UNIVERSITY OF CALIFORNIA
Santa Barbara

Automated Configuration and Deployment of Applications in Heterogeneous Cloud Environments

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Chris Bunch

Committee in Charge:

Professor Chandra Krintz, Chair

Professor Amr El Abbadi

Professor John Gilbert

November 2012
UCSB Technical Report 2013-02

The Dissertation of
Chris Bunch is approved:

Professor Amr El Abbadi

Professor John Gilbert

Professor Chandra Krintz, Committee Chairperson

November 2012

Automated Configuration and Deployment of Applications in Heterogeneous Cloud Environments

UCSB Technical Report 2013-02

by

Chris Bunch

Dedication and Gratitude

I dedicate this dissertation to my family: my wife, Alexandra, my parents, Aisha and David, my brother, Michael, and my grandparents, Connie and Charles, for their unconditional support and encouragement throughout all stages of my education.

I am deeply grateful to Chandra Krintz for all of the support, guidance, mentorship, and help that she has provided during the entire process.

I would like to thank John Gilbert and Amr El Abbadi for serving on my Ph.D. committee.

I am grateful to Khawaja Shams and Mike Aizatsky for being my mentors during my internships at JPL NASA and Google, respectively.

Finally, I would like to thank the staff, faculty, and fellow graduate students at the Computer Science department at UC Santa Barbara for their support and the opportunity to pursue this work.

Acknowledgements

The text of Chapters 3–6 is in part a reprint of the material as it appears in the conference proceedings listed below. The dissertation author was the primary researcher while the co-author listed on each publication directed and supervised the research which forms the basis for these chapters.

Chapter 3: Publication [15] in the 5th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2012).

Chapter 4: Publication [19] in the Special Issue on Data Intensive Computing in the Clouds (DataCloud 2012).

Chapter 5: Publication [20] in the Special Issue on Interoperability, Federation, Frameworks and Application Programming Interfaces for Infrastructure-as-a-Service (IaaS) Clouds (in submission).

Chapter 6: Publication [18] in the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013) (in submission).

Curriculum Vitæ

Chris Bunch

Education

2012	**Doctor of Philosophy in Computer Science,**
	University of California, Santa Barbara.
2012	**Master of Science in Computer Science,**
	University of California, Santa Barbara.
2007	**Bachelor of Science in Computer Science,**
	California State University, Northridge.

Experience

2007 – 2012	**Graduate Research Assistant,**
	University of California, Santa Barbara.
2011	**Intern,**
	Google, San Francisco, CA.
2010	**Intern,**
	Jet Propulsion Laboratory, Pasadena, CA.

Awards

2008	**Certificate of Teaching Excellence,**
	University of California, Santa Barbara.

Publications

Chris Bunch and Brian Drawert and Navraj Chohan and Andres Riofrio and Chandra Krintz and Linda Petzold: "MEDEA: A Pluggable Middleware System for Interoperable Program Execution Across Cloud Fabrics." *In the Journal of Grid Computing Special Issue: Interoperability, Federation, Frameworks and Application Programming Interfaces for IaaS Clouds, 2013 (in submission)*

Chris Bunch and Brian Drawert and Navraj Chohan and Chandra Krintz and Linda Petzold: "Exodus: An Application Programming Interface for Cost-Aware, Cloud-Aware Program Execution." *In the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2013 (in submission)*

Chris Bunch and Brian Drawert and Navraj Chohan and Andres Riofrio and Chandra Krintz and Linda Petzold: "MEDEA: A Pluggable Middleware System for Portable Program Execution." *In the 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2013 (in submission)*

Chris Bunch and Vaibhav Arora and Navraj Chohan and Chandra Krintz and Shashank Hegde and Ankit Srivastava: "A Pluggable Autoscaling Service for Open Cloud PaaS Systems." *In the 5th IEEE/ACM International Conference on Utility and Cloud Computing (UCC), 2012*

Navraj Chohan and Anand Gupta and Chris Bunch and Sujay Sundaram and Chandra Krintz: "North by Northwest: Infrastructure Agnostic and Datastore Agnostic Live Migration of Private Cloud Platforms." *In the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud), 2012*

Navraj Chohan and Anand Gupta and Chris Bunch and Kowshik Prakasam and Chandra Krintz: "Hybrid Cloud Support for Large Scale Analytics and Web Processing." *In the 3rd USENIX Conference on Web Application Development (WebApps), 2012*

Chris Bunch and Brian Drawert and Navraj Chohan and Chandra Krintz and Linda Petzold and Khawaja Shams: "Language and Runtime Support for Automatic Configuration and Deployment of Scientific Computing Software over Cloud Fabrics." *In the Special Issue on Data Intensive Computing in the Clouds (DataCloud), 2012*

Chris Bunch and Navraj Chohan and Chandra Krintz: "Supporting Placement and Data Consistency Strategies using Hybrid Clouds." *In the IEEE Aerospace Conference, 2012*

Chris Bunch and Chandra Krintz: "Enabling Automated HPC / Database Deployment via the AppScale Hybrid Cloud Platform." *In the 1st Workshop on High-Performance Computing meets Databases (HPCDB), 2011*

Navraj Chohan and Chris Bunch and Chandra Krintz and Yoshihide Nomura: "Database-Agnostic Transaction Support for Cloud Infrastructures." *In the 4th International Conference on Cloud Computing (CLOUD), 2011*

Chris Bunch and Navraj Chohan and Chandra Krintz and Khawaja Shams: "Neptune: A Domain Specific Language for Deploying HPC Software on Cloud Platforms." *In the 2nd Workshop on Scientific Cloud Computing (ScienceCloud), 2011* ***(Best Paper Award)***

Chris Bunch and Jonathan Kupferman and Chandra Krintz: "Active Cloud DB: A RESTful Software-as-a-Service for Language Agnostic Access to Distributed Datastores." *In the International Conference on Cloud Computing (ICST CloudComp), 2010*

Sylvain Hallé and Taylor Ettema and Chris Bunch and Tevfik Bultan: "Eliminating Navigation Errors in Web Applications via Model Checking and Runtime Enforcement of Navigation State Machines." *In the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2010*

Chris Bunch and Navraj Chohan and Chandra Krintz and Jovan Chohan and Jonathan Kupferman and Puneet Lakhina and Yiming Li and Yoshihide Nomura: "An Evaluation of Distributed Datastores Using the AppScale Cloud Platform." *In the 3rd International Conference on Cloud Computing (CLOUD), 2010*

Navraj Chohan and Chris Bunch and Sydney Pang and Chandra Krintz and Nagy Mostafa and Sunil Soman and Rich Wolski: "AppScale: Scalable and Open AppEngine Application Development and Deployment." *In the International Conference on Cloud Computing (ICST CloudComp), 2009*

Field of Study: Computer Science

Abstract

Automated Configuration and Deployment of Applications in Heterogeneous Cloud Environments

Chris Bunch

Cloud computing is a service-oriented approach to distributed computing that provides users with resources at varying levels of abstraction. Cloud infrastructures provide users with access to self-service virtual machines that they can customize for their applications. Alternatively, cloud platforms offer users a fully managed programming stack that users can deploy their applications to and scales without user intervention. Yet challenges remain to using cloud computing systems effectively. Cloud services are offered at varying levels of abstraction, meter based on vendor-specific pricing models, and expose access to their services via proprietary APIs. This raises the barrier-to-entry for each cloud service, and encourages vendor lock-in.

The focus of our research is to design and implement research tools to mitigate the effects of these barriers-to-entry. We design and implement tools that service users in the web services domain, high performance computing domain, and general-purpose application domain. These tools operate on a wide variety of cloud services, and automatically execute applications provided by users, so that the user does not need to be conscientious of how each service operates and meters. Furthermore, these tools

leverage programming language support to facilitate more expressive workflows for evolving use cases.

Our empirical results indicate that our contributions are able to effectively execute user-provided applications across cloud compute services from multiple, competing vendors. We demonstrate how we are able to provide users with tools that can be used to benchmark cloud compute, storage, and queue services, without needing to first learn the particulars of each cloud service. Additionally, we are able to optimize the execution of user-provided applications based on cost, performance, or via user-defined metrics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing is a service oriented approach to distributed computing wherein vendors lease resources to users on a metered, pay-per-use basis. This enables companies to quickly acquire resources (e.g., virtual machines, storage) and release them when they are no longer needed. To date, cloud computing has mostly been seen in use by web service companies, who can use this elasticity to acquire machines in response to increased web traffic, without having to pay the full cost for those machines.

In response to the increasing number and low cost of cloud service offerings, other communities are investigating the feasibility of cloud services within their domains. One such community are computational scientists, who utilize computational resources to simulate, investigate, and experiment with scientific processes. These processes may rely on data collected with "traditional" scientific devices (e.g., microscopes, chemical interactions), but may be modelled via computer to better understand the underlying phenomena involved.

Yet using cloud services for scientific computing is a far from trivial undertaking. Scientists may not have the same levels of experience with configuring, debugging, installing, and maintaining complex programs as a full-time system administrator, and the time it would take to acquire this experience is time that could otherwise be spent furthering their own scientific research. Furthermore, in the case of student scientists, these workers are transient in nature, and the focus that they put into their research tends to come at the expense of a proper knowledge transfer process. This results in a longer learning curve for new students, and makes it more difficult to properly evaluate new cloud technologies. The primary quantities that must be considered to make this evaluation are:

- **The capabilities of the service to use.** Not all services are identical offerings, and can operate at varying layers of abstraction and require differing amounts of maintainence. Infrastructure-as-a-Service (IaaS) offerings, such as Amazon EC2 [2] and Google Compute Engine [47], provide access to virtual machines, metered on a per-hour basis. At a higher level of abstraction exists Platform-as-a-Service (PaaS), which provides access to full runtime stacks. These stacks can offer the traditional three tier-web deployment strategy, in which users can present web pages to users from one or more application servers, store and retrieve data via a persistent database, and cache frequently accessed data. Providers, such as Google App Engine [44], deny users access to individual machines, but in-

stead enable users to focus only on their applications, which the provider then can scale on the user's behalf. Finally, Software-as-a-Service (SaaS) providers offer end-users a single application that they can configure for their needs.

- **The cost model of the service to use.** As a motivating example, consider virtual machine (IaaS-level) services. Amazon EC2 and Google Compute Engine meter on a per-hour basis, while Microsoft Azure [70] meters on a per-wall-clock-hour basis. Alternatively, Google App Engine meters on a per-minute basis. These varying cost models must be taken into consideration to minimize the cost incurred to use cloud resources. Resources that are metered on a finer granularity (shorter amounts of time) encourage users to acquire more resources, use them immediately, and release them, while resources metered on courser granularities (longer amounts of time) encourage users to acquire less resources and spread out resource usage over that quantum.

- **The application programming interfaces (API) that exist to connect the user's application to the cloud service.** Cloud vendors provide first-party library support to access their services for users of certain, but not all, programming languages. Third-party support exists for a wider array of programming languages, but often lags behind first-party support in terms of feature sets and overall quality. This means that the program the scientist is developing is not guaranteed to

be compatible with all cloud services, and that an investigation must be launched to determine which services (and which APIs within each service) can be utilized within the language that the scientist's application is written in.

Once the scientist decides which of these services they want to utilize, they must then implement and maintain their system utilizing each of the chosen technologies. If transitioning off of an existing system, then the scientist must port their system to the new technology. The time and engineering costs that have been invested in learning each of these technologies is not directly transferrable to other technologies: while other competitors may be abstractly similar (i.e., Amazon EC2 and Google Compute Engine both offer virtual machines), in practice their APIs are incompatible, requiring an expert to refactor the code base when porting to other services. Finally, the scientist must spend additional time to transfer the knowledge of how to maintain their application (which now utilizes a new set of services) with others.

The above process results in the creation of a system that is optimized for, and thus only supports, a single scientific application. This system requires a system administrator to maintain, and is not reusable for other applications. In practice, this is because systems are not typically designed to be automatically configured and deployed. This would require the system to be made general-purpose, to enable arbitrary programs to be deployed, and may do so at the cost of the performance of hosted applications (as application-specific information may be lost in the generalization process).

As a result of trends between cloud service offerings, it is increasingly common for scientists to leverage several cloud products to solve a single problem, and to not generalize their application to service other, possibly related, problems. Therefore, these solutions tend to lack *automation* with respect to both *configuration* and *deployment*, and are dependent (or "locked-in", in the cloud vernacular) upon certain vendor's offerings. These offerings are heterogeneous in terms of the *services offered* (e.g., virtual machines at the IaaS layer, runtime stacks at the PaaS layer, and applications at the SaaS layer), their *cost models* (e.g., per-minute metering, per-hour pricing, or per-API-request pricing), and the *interfaces* (tied to one or more programming languages) that can be utilized to access them.

1.1 Thesis Question

The primary research question that we explore in this dissertation can be stated as follows:

> *How can we enable scientific applications to be executed on cloud systems, by automatically configuring and deploying applications across cloud offerings that vary based on the type of service offered, cost model employed, and APIs via which services are exposed?*

To answer this question, we design, implement, and evaluate *open source Platform-as-a-Service* solutions that automatically configure and deploy applications from various application domains. Our goal is to execute applications intelligently, considering

Domain	Language / Platform Support
Web Services	AppScale (3)
High Performance Computing	Neptune (4)
Arbitrary Applications	MEDEA (5), Exodus (6)

Table 1.1: Design space in automated application execution that we investigate. Each row lists the domain that this work addresses as well as the solution that we design, implement, and evaluate our support for. In parentheses, we show the chapter number that describe the corresponding systems that we contribute.

both how to execute applications as well as how to optimally do so with respect to performance, cost, or user-defined metrics. We leverage programming language support to simplify how users specify that programs should be run via cloud services, and investigate PaaS support for *web service applications, high performance computing applications, and general purpose applications*.

Table 1.1 summarizes the *design space* that we cover with this dissertation. A primary goal of this work is to provide *pluggable* systems that expert users can impart information into, that can then be automatically leveraged for non-expert users and the community at large. The aim is to provide a research tool that can be used to evaluate cloud service offerings for applications whose underlying usage patterns may vary greatly between one another.

Another key goal of our research is to provide *programming language support* to enable Turing-complete specifications of scientific workflows. This enables scientists to dynamically indicate when their computations have finished, to consult expert users via e-mail or other existing infrastructures (as data sets may be too complex to analyze

in a purely programmatic fashion), or to run certain computations only as long as they can be done quickly or inexpensively.

This work aims to target a diverse array of application domains, to maximize the impact of the systems contributed here. We design and implement AppScale (Chapter 3) to aim at targetting web service applications to improve the Quality-of-Service that they provide to users while minimizing the cost incurred to do so. We direct our focus to high performance computing applications, which form a crucial core of scientific applications and have, to-date, been primarily discussed outside of the context of cloud systems. We intend to simplify their often complex deployment via expressive programming language support, and do so without sacrificing performance, via the Neptune domain specific language (Chapter 4). This enables users to write programs that interactively investigate the results of their experiments, and launch new experiments in response to these results. Finally, MEDEA and Exodus (Chapters 5 and 6, respectively) are our efforts to target general-purpose applications, widening the reach of our contributions and maximizing the types of scientific research that can be performed. We intend to do so while preserving the ease of use that scientists (and users at large) have come to expect from their systems, and while preserving the research contributions of our systems that serve web service and high performance computing applications.

1.2 Dissertation Organization

We organize the remainder of this dissertation as follows. We begin by providing background information, discussing terminology, state-of-the-art systems, open problems, and limitations in automatically configuring and deploying applications across cloud systems in Chapter 2. Chapters 3–6 describe the four systems that we contribute to address our thesis question and that represent separate points in the design space shown in Table 1.1. In this table, the parenthesized values correspond to the chapter numbers that detail solutions for the domain in question. Each of these four chapters motivate the particular problems they aim to solve, discuss how they are designed and implemented to solve these forward new types of science to be performed, evaluate applications that their systems support, discuss related work, and conclusions. Chapter 3 focuses on web service applications, while Chapter 4 focuses on high performance computing applications. Arbitrary applications are discussed in Chapters 5 and 6. Chapter 7 summarizes our contributions and discusses future research directions.

Chapter 2

Background

In this chapter, we provide background on, and survey the state-of-the-art in, middleware systems that are used to automatically configure and deploy applications in the web services and scientific computing domains. Of particular interest to us are systems deployed on statically allocated resources (e.g., grids, clusters) as well as on dynamically allocated resources (e.g., clouds). We overview recent advances in automatic program execution and deployment as well as the limitations found in each of these systems.

2.1 Application Execution via Static Resources

In this section, we overview systems that allow for the automated configuration and deployment of programs in the context of statically acquired resources. These resources may be classified as grids or clusters, but fundamentally are static in size: users access

a fixed number of machines, which can only change due to the influence of a system administrator (occurring infrequently). We provide background on systems harnessing these resources, in the context of web services and scientific computing, and discuss their limitations.

2.1.1 Web Services

Web services are offered by organizations to both internal and external-facing users, with resources typically hosted on-premise or in an organization-owned datacenter. Users tend to be served web traffic via the standard, three-tier deployment strategy, in which users access a load balancer (first tier), which routes them to one or more application servers (second tier), which store/retrieve data via one or more database servers (third tier). Each of these three tiers has performance characteristics (with respect to CPU, memory, and I/O usage) and usage patterns that depend the behavior of its accessing tier.

The widespread usage of web services has led to a number of software stacks emerging solely to support it. The perhaps most well-known is the Linux-Apache-MySQL-PHP (LAMP) stack, which provides users with a fully open source stack that is therefore free to use. Best practices and additional software packages have since emerged that try to simplify the configuration and deployment process for LAMP-stack applications, and to try to extend the range of programs supported in the stack.

Chef [26] and Puppet [76] are two efforts that aim to simplify application configuration and deployment through the use of domain specific languages. Users write scripts ("recipes" in Chef and Puppet's nomenclature) that describe which software should be installed, how to install it, and how to deploy it. These systems work on a single machine as well as in distributed environments. Although simplifying the deployment process, these systems still require the presence of an expert user, who must optimally place and configure their components as well as maintain Chef or Puppet themselves in a distributed environment.

Research efforts have taken a largely orthogonal approach to these domain specific languages, and have instead aimed to improve resource usage within each tier and across tiers. [92] utilizes results from queueing theory to model the three-tier web deployment strategy as three queueing systems, each with their own production and consumption rates. Their system does not support the LAMP stack, but supports a similar stack through the use of Apache as the load balancer, Tomcat as the application server (hosting applications written in the Java programming language), and MySQL as the database server. They contribute scaling algorithms for scenarios when each tier can and cannot benefit from the use of hot spares to elastically scale up and down, as long as the applications supported have been "well-profiled" and generate accurate heuristics for their algorithms.

In practice, profiling these applications to acquire these heuristics is a non-trivial task. One effort that addresses this problem is [12]. This work argues that the only way to properly profile an application is to instrument its production environment, and use that data to determine when to scale resources up and down. In lieu of the scalers proposed by [92], [12] uses hot spares as a safeguard if their autoscaler consumes too many resources for profiling purposes (which would otherwise leave too few resources available to users).

2.1.2 Scientific Computing

Grid computing services are offered by organizations to their internal users with resources typically hosted on-premise or in an organization-owned datacenter. System administrators determine which software packages are installed and supported on these machines, which can be programmatically acquired and released by users. These systems have largely been utilized for scientific computing applications, with a specific emphasis on the field of high-performance computing (HPC).

The most well-known interface to machines hosted in a grid is the Portable Batch System (PBS) [75]. PBS enables users to acquire machines hosted within a grid, execute one or more programs on them, and release those machines. The Simple API for Grid Applications (SAGA) [60] project aims to fulfill a similar goal, but instead of

requiring users to learn a new job description language, provides APIs within the Java, C, and Python programming languages to facilitate greater ease-of-use.

Extensions in this space have been largely concerned with expanding the types of programs that can be run and with improving resource utilization when a large number of programs are run. BatchPipes and Swift [35] provide users with the ability to "chain" the execution of many programs together and specify interdependencies amongst programs via XML, while systems like Pegasus [34] can consume these workflow descriptions and attempt to optimize their execution for some set of metrics (e.g., end-to-end execution time, grid resource usage). Both of these systems can utilize "community grid" systems like Condor [90], wherein idle resources (e.g., terminals in a university that are not in use by students) are employed by the grid for execution of submitted jobs, until a user resumes use of their terminal (at which point the job is checkpointed and aborted).

Other efforts have focused on restricting the runtime stack to optimize shared cluster usage, or altering the runtime stack to focus on non-traditional hardware profiles. In the former category are projects like Mesos [56], which limits the runtime stack to MPI and Hadoop MapReduce programs, to attempt to improve CPU utilization on company-owned clusters. In the latter category are the Anyscale Many-Task Computing Engine (AME) [99] and StratUm [73]. AME seeks to resolve engineering difficulties that have arisen from deploying grid software onto supercomputers by taking into account

supercomputer-specific information, while StratUm implements a meta-scheduler to dispatch biochemical computing applications across one or more community grids.

2.1.3 Limitations

Mainstream cluster and grid offerings to date offer mature software solutions, providing stable access to on-premise hardware. In the context of providing automated application deployment, these systems possess the following key limitations:

- Resources are statically utilized within grid and cluster systems. System administrators may add and remove resources, but this is a relatively rare event, and more relevantly, resources can not be added or removed programmatically. In the context of web applications, this means that users try to avoid downtime by provisioning for the maximum amount of traffic they could face (as opposed to the current amount of traffic they face). This causes resources to be wasted during non-peak hours.

- Software stacks must be maintained by system administrators on-premise. In the context of HPC applications, this means that users can only deploy applications written in supported frameworks (and only certain versions of those frameworks), even if other frameworks offer better performance or if newer versions of supported frameworks are available. Furthermore, system administrators are

responsible for maintaining the health and availability of the cluster or grid itself. End-users with sufficient technical knowledge or system expertise have no mechanisms to alleviate this, even at smaller scales.

Extant systems have begun to address these limitations by utilizing virtualization. This technology enables machines to be emulated as programs, known as virtual machines, and when architecture support for virtualization is enabled, virtual machines suffer little performance degradation compared to their non-virtualized counterparts.

The following section details how the grid and cluster computing fields have evolved by utilizing virtualization to create a new field of study, known as cloud computing. This field is still in its infancy, but many of the software offerings detailed here (e.g., SAGA, Swift) have already begun to support cloud services to enable programs to be configured and deployed automatically.

2.2 Application Execution via Dynamic Resources

In this section, we overview systems that allow for programmatic interaction and deployment of programs in the context of dynamically acquired resources. These resources may be classified as Infrastructure-as-a-Service clouds or Platform-as-a-Service clouds, which provide scalable access to virtual machines or full software stacks, re-

spectively. We provide background on systems harnessing these resources, detail what application domains they serve, and enumerate their limtations.

2.2.1 Programmatic Cloud Interaction

Cloud computing services are offered by vendors to the public on a pay-per-use basis or within private institutions, often hosted on-premise. These services are typically classified at two tiers: Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS). At the IaaS layer, virtual machines are offered to users. Users receive root-level access to these machines, and have both the freedom and the responsibility to utilize them correctly for their needs. Public offerings can typically scale to as many machines as the end user can afford, while private offerings scale as far as the total size of the on-premise deployment.

The Amazon Elastic Compute Cloud [2], often abbreviated to simply Amazon EC2, is the oldest and most well-known IaaS offering. Machines with different hardware profiles ("instance types" within the EC2 nomenclature) are offered to users and are metered on a per-hour basis. For example, a user who uses a single machine for an hour and a half would be charged for two full hours of use. Amazon EC2 offers virtual machines in an on-demand fashion as well as in an auction-style offering, known as Spot Instances [3]. In contrast to the on-demand instance offering, the price that a user pays for a Spot Instance is not constant, but instead is set by Amazon in an opaque

fashion, based on excess machine capacity. Users place bids for these instances, and as long as the user's bid exceeds the price set by Amazon, they are granted access to the machines in question. Spot Instances tend to cost less than On-Demand instances, but can be reclaimed by Amazon at any time, limiting the types of applications that can use them effectively.

A number of software packages provide interfaces to Amazon EC2 for users of different programming languages. `boto`[13] provides this functionality for users of the Python programming language, while the RightScale Gems [79] fulfil this role for the Ruby programming language. Furthermore, while the SAGA [60] project originally targetted grid interaction, it has been repurposed since the introduction of Amazon EC2 to also interact with it.

2.2.2 Automated Service Deployment

While IaaS offerings provide low-level access to virtual machines in an on-demand fashion, Platform-as-a-Service (PaaS) offerings provide scalable access to a fully customized software stack. PaaS offerings vary in the types of autoscaling mechanisms that are exposed to end users, in a manner proportional to the customizability of hosted programs.

For example, Google App Engine [44] provides a PaaS offering that originally only hosted applications written in Python 2.5. Furthermore, only web applications could be

hosted on Google App Engine, and only API calls on Google's whitelist could be used. In practice, this disallows file system access, socket creation, or the persistence of data in any fashion other than a Google-hosted database or a transient caching layer. Web responses were also initially limited to 30 seconds in length. This restricted runtime environment forces the web server to be stateless and its performance to be predictable, allowing Google App Engine to easily scale any hosted application, regardless of its actual content. Furthermore, this scaling can be performed transparently, without the need for the user to indicate scaling rules (which they may be unfamiliar with or be unqualified to instruct Google's closed source platform on). Since its initial introduction, Google App Engine has expanded to also support applications written in the Java and Go programming languages, with similar runtime restrictions.

Alternatively, Microsoft Azure [70] provides a PaaS offering that allows users to host applications of any programming language, without restrictions. For automated scaling to occur, a rule language is exposed to users, who must then dictate a series of rules that indicate to the platform when resources should be scaled up or scaled down. As the platform can be running code in any language, the scaling rules cannot be tied to user-specific code, and must rely on conditions involving the state of Azure-hosted services. In practice, this means that applications can scale up or down based on CPU and memory usage of hosted VMs, or the state of the queue and storage services provided by Microsoft Azure (e.g., scale up if the storage service has been accessed

more than X times in the last Y minutes). This increases the amount of complexity that the end user has to deal with to properly host applications on this platform, but correspondingly increases the variety in the types of applications that can be hosted.

The Nimbus project [61] provides two offerings that are aimed at bringing the types of automated service deployment offered by a PaaS to IaaS systems. The first offering, `cloudinit.d`, provides an API that can be utilized to launch and configure services in a cloud IaaS. Users write scripts that utilize `cloudinit.d` to acquire virtual machines and use them in their applications. In contrast, the second offering, the Nimbus Context Broker, shifts the complexity of writing scripts to utilize virtual machines onto the cloud IaaS itself. The user can then simply ask for a "pre-configured", specialized virtual machine, and the Context Broker will acquire a "vanilla", base virtual machine and customize it accordingly. Both offerings do not provide autoscaling, and thus may not be true PaaS offerings, but attempt to bridge the gap between IaaS and PaaS.

Conversely, Elastisizer [53] does offer a PaaS-like system that attempts to automatically deploy Hadoop MapReduce programs [49] to Amazon EC2, in a manner similar to Amazon Elastic MapReduce [4]. Both systems provide automated configuration and deployment for Hadoop MapReduce, but Elastisizer adds capabilities to intelligently place Map and Reduce tasks on virtual machines based on CPU, memory, or I/O load.

Another offering that allows for automated program execution within a cloud PaaS is the Google App Engine Pipeline API [46]. Users write Python or Java code (as is re-

quired by Google App Engine) and indicate which functions are to be chained together, in a manner similar to that of the workflow systems detailed previously. The core difference comes in that users write code in a Turing-complete language, allowing dynamic workflows that can consult humans (e.g., via e-mail or instant messaging services) if the data to analyze is too complex for a program to analyze alone.

In a similar manner to the workflow systems utilized for statically acquired sets of resources (e.g., grids and clusters), Amazon also provides a workflow service that harnesses EC2 to run computations. This service, known as Amazon Simple Workflow Service (SWF) [7], enables users to statically define workflows, which are then executed on machines hosted within Amazon or on-premise (which must be administered manually).

2.2.3 Limitations

Mainstream cloud offerings to date offer a valuable first step in offering unprecedented amounts of raw compute capacity to the community at large (e.g., scientists, system administrators, end-users). However, these offerings as a whole tend to fall into one of the two following ideologies:

- **Generalize, at the cost of specialization.** Cloud Infrastructure-as-a-Service offerings tend to fall into this category, in which users are offered root-level virtual machine access and have the ability to do anything, but consequently are required

to do everything to produce a scalable solution. This requires users to become system administrators (an often difficult and costly endeavor) and produce solutions that are not extensible to inclusion or use by other software artifacts.

- **Specialize, at the cost of generalization.** Cloud Platform-as-a-Service offerings tend to fall into this category, in which users are offered a specific software stack and cannot modify it in any way. This alleviates users of the burden of becoming system administrators, but now requires users to (1) evaluate if their application can run within the allowed software stack, and (2) rewrite their application to run effectively on the cloud platform.

Some work has been done by others [29] to provide a "middle-ground" between these offerings, wherein the platform is customizable and yet system administration is not the user's responsibility. Yet the ability to customize the software stack in these offerings comes at the cost of auto-scaling, and require extant, non-free solutions to partially mitigate this issue. The requirements for a state-of-the-art research cloud platform should therefore incorporate at least some of the following goals and features:

- **Open source.** As a research tool, the ability to modify the platform at will makes it feasible as a tool for conducting scientific experiments. This requires us to produce a tool that is open to the public, and that anyone can use, at no cost to them,

to test and validate our theories through rigorous experiment and observation, and give users the ability to create and test their own theories.

- **One-button deployment.** The research tool will only be useful to users if it is simple to utilize. Any barriers-to-entry will preclude users from harnessing this tool, and thus is detrimental to their ability to use it as a scientific tool.

- **Extensible to different software stacks.** The research tool should not require users to conform to it, but instead, conform to the user's programs in a reasonable fashion. The initial offering may not support every software stack, but is open to customization by the community-at-large.

- **Extensible within supported software stacks.** The research tool should enable sufficiently interested users to customize it to add domain-specific library support (e.g., for high performance computing, for image processing) as they require.

- **Auto-scale for supported software stacks.** Extending the research tool to add additional functionality should not come at the cost of losing autoscaling capabilities, and thus the tool should be able to acquire and release cloud resources to best serve user requests.

- **Pluggable to different cloud services.** Harnessing the resources of a single set of cloud resources encourages vendor lock-in, which harms the portability of

supported applications. It is thus imperative that the research tool be able to run on resources hosted in public clouds (off-premise) or private clouds (on-premise), which may possible provide differing APIs.

Prior work in this field tends to provide a small subset of these requirements, and no single offering satisfies all of these requirements. Currently, the key limitations of existing work are:

- Many offerings are sold as commercial-off-the-shelf (COTS) products or are remotely hosted by the vendor, so their source code is not open to inspection or extension. This precludes their use as the primary research tool, as it harms the ability to run experiments that are repeatable over long periods of time (as the vendor is incentivized to improve or otherwise alter their services to better serve their customers).

- Ease of installation and use has not, to date, been a first-class feature of existing offerings. This means that the sheer complexity of existing software has come with a correspondingly complex installation process. This hampers the ability to utilize these tools as a vehicle for scientific research, and has been a barrier-to-entry for all but the most technically savvy system administrators.

- Platforms-as-a-Service, by their very definition, offer scalable access to full software stacks. As the majority of mature PaaS offerings are hosted within a ven-

dor's datacenter(s), this places the onus of securing these machines on the vendor, and disincentivizes them to experiment with a wider array of software stacks, programming languages, and libraries. Similarly, there is an incentive to only release features that will be used by the majority of customers, making them less attractive to use by researchers (who may want to experiment at all layers of the software stack, and do not have the same budget as an enterprise customer).

- Platform-as-a-Service offerings tend to restrict the supported software stack to enable autoscaling. Alternatively, some offerings choose the reverse decision: to disable autoscaling to enable a wider array of software stacks. A first-class research tool should seek to provide both autoscaling and more than a single software stack, and investigate how to do so in the general case (so that it can be adopted, evaluated, and improved upon by others).

- Commercial vendors are monetarily incentivized to create "lock-in", and create incompatible APIs and cost models for what are conceptually similar services (e.g., FIFO queues). This makes it difficult for even expert users to determine which services are the best for their application and usage pattern. Furthermore, due to the rapidly evolving nature of these applications, which services may be "best" (e.g., w.r.t. price, performance, ease of use) may change over time, and the cost of changing from one provider to another (via refactoring, system adminis-

tration, etc.) tends to greatly outweigh the cost of paying more for services from the original vendor. This disincentivizes competition amongst cloud vendors, especially the vendor who maintains the largest market share (who can simply dictate which APIs should be used, instead of creating open APIs with other vendors).

The systems described in Chapters 3, 4, and 5 address these limitations by designing and implementing pluggable middleware systems that enable expert users to inject their own software stacks and autoscale them as desired. Once an expert user does this, users at all skill levels can take advantage of this work in their own research or commercial applications.

Chapter 3

A Pluggable Autoscaling Service for Open Cloud PaaS Systems

In this chapter, we present the design, implementation, and evaluation of a pluggable autoscaler within an open cloud platform-as-a-service (PaaS). We redefine high availability (HA) as the dynamic use of virtual machines to keep services available to users, making it a subset of elasticity (the dynamic use of virtual machines). This makes it possible to investigate autoscalers that simultaneously address HA and elasticity. We present and evaluate autoscalers within this pluggable system that are HA-aware and Quality-of-Service (QoS)-aware for web applications written in different programming languages, automatically (that is, without user intervention). Hot spares can also be utilized to provide both HA and improve QoS to web users. Within the open source AppScale PaaS, utilizing hot spares within the HA-aware autoscaler can reduce the amount of time needed to respond to node failures by an average of 48%, and can in-

crease the amount of web traffic that the QoS-aware autoscaler serves to users by up to 32%.

As this autoscaling system operates at the PaaS layer, it is able to control virtual machines and be cost-aware when addressing HA and QoS. Therefore, we augment these autoscalers to make them cost-aware. This cost awareness uses Spot Instances within Amazon EC2 to reduce the cost of machines acquired by 91%, in exchange for an increase in startup time. This system facilitates the investigation of new autoscaling algorithms by others that can take advantage of metrics provided by different levels of the cloud stack (IaaS, PaaS, and SaaS).

3.1 Introduction and Motivation

While cloud IaaS and PaaS systems have seen sizable increases in usage, they have also suffered from a number of outages [32] [45], with some lasting several days [41] [55]. The remedy to the problem of single-datacenter failures (as recommended by IaaS vendors) is to utilize resources across multiple datacenters, and to use autoscaling products (e.g., RightScale, CloudWatch) to provide fault detection, recovery, and elasticity. Yet to make these offerings general-purpose, for use with services written in any programming language, the metrics with which these products can autoscale are limited and statically defined. In practice, these systems are usually

rule-based and can scale on coarsely-defined, VM-level metrics, including CPU usage, memory usage, and system load. Furthermore, the closed source nature of these offerings makes them inextensible and precludes their use by the community at large (e.g., researchers, developers, system administrators) to perform autoscaling on applications written in different programming languages, based on application-specific metrics.

As a motivating example, consider a typical application utilizing an IaaS and a LAMP stack. Once this application becomes popular, the developer or system administrator needs to manually scale this application out by hand, which (at the bare minimum) requires them to become experts at scaling load balancers, application servers, and database nodes. By contrast, if the application itself runs at the PaaS layer, then the burden of autoscaling is removed from the developer and placed onto the PaaS vendor. Furthermore, the runtime restrictions that PaaS providers enforce mean that the application itself does not need to be modified to facilitate scaling.

We mitigate the problem of autoscaling by reinterpreting high availability (HA) under the veil of elasticity, and proposing a *pluggable* autoscaling service that operates within at the PaaS layer. Operating at the PaaS layer enables the autoscaling tool to use high-level, application-specific metrics (e.g., database or API usage) as well as low-level, cloud-specific metrics (e.g., hypervisor or cloud IaaS scheduling decisions). Furthermore, because the autoscaling tool operates at the PaaS layer, it can perform both inter-VM scaling and intra-VM scaling. Additionally, we elect to utilize the Google

Role Name	Implemented Via
Load Balancer	`haproxy`
AppServer	Modified AppServer
Database	*Pluggable* [16]
AppCaching	`memcached`
Service Bus	VMWare RabbitMQ
Metadata	Apache ZooKeeper
AppController	Ruby daemon

Table 3.1: A listing of the roles within the AppScale PaaS and the open source technologies that implements them.

App Engine PaaS, so that our autoscaling service can operate on the one million active applications that currently run on Google App Engine [72].

This work targets the AppScale and Eucalyptus [71] cloud systems, but the techniques detailed here are extensible to other PaaS/IaaS systems. AppScale, originally detailed in [28] and extended in [16][27], is an open source implementation of the Google App Engine APIs. This enables any application written for Google App Engine to execute over AppScale without modification. As AppScale is open source, it is extensible to other application domains; in [19], it was extended to support high-performance computing (HPC) frameworks, including MPI [48] and X10 [25]. AppScale runs over the Amazon EC2 public cloud IaaS as well as the Eucalyptus private cloud IaaS, an open source implementation of the EC2 APIs.

We begin by detailing the design of our pluggable autoscaling service and its implementation within the open source AppScale PaaS. We then evaluate autoscalers that implement support for HA, Quality-of-Service (QoS), and cost awareness. We discuss

this support within the open source Eucalyptus IaaS as well as over the closed source Amazon EC2 IaaS. We then discuss related work and conclude.

3.2 Design

This work redefines HA as the acquisition of virtual machines to keep services available to end-users, making it a special case of elasticity (the acquisition and release of virtual machines). We discuss how we use this idea within a cloud PaaS to provide HA via elasticity and our implementation of this idea in the open source AppScale PaaS. We then detail the pluggable autoscaling system that AppScale enables, along with a number of autoscalers that can be used within AppScale to provide HA, Quality-of-Service (QoS), and cost awareness for hosted applications.

3.2.1 Role System

The goal of our work is to use elasticity to implement HA. To support this aim within a cloud PaaS, it is necessary to support HA for the full software stack that a cloud PaaS provides for its users. The approach that we take within the AppScale PaaS is what we call a *role system*, where each part of the software stack is designated by a unique *role* that indicates what responsibilities it takes on and how it should be "started" (configured and deployed) and "stopped" (its tear-down process). Scripts are included

in the AppScale code base that indicate how each role can be started and stopped, as needed. A list of the roles supported by the AppScale PaaS, their functionality, and the open source software packages that implement these roles are detailed in Table 3.1.

Roles are started and stopped on each node by a Ruby daemon named the AppController. Users detail the "placement strategy" (a map indicating which nodes run each set of roles) for their AppScale deployment and, using a set of command-line tools, pass this information to an AppController. The AppController then sends this information to all other AppControllers in the system, and starts all the roles for its own node. Because the AppController itself is "role-aware", start and stop scripts can take advantage of this to enforce dependencies between roles. A common dependency is the reliance of the AppServer on the Database, AppCaching, and Service Bus roles, which are all required for the AppServer to start correctly.

As an example of how users specify roles in their placement strategy, consider the following AppScale configuration file (specified in the YAML [96] format):

```
---
:load_balancer:
- node-1
:app_server:
- node-2
- node-3
```

```
- node-4
:database:
- node-5
- node-6
- node-7
```

Here, the user has specified that they wish to have a single load balancer role, three application server roles, and three database roles. This configuration file is used by the AppControllers to generate the AppScale deployment shown in Figure 3.1. Note that users need not indicate here which cloud IaaS they run over, as this is abstracted away from them and whatever cloud IaaS credentials they make available to AppScale are used to acquire resources. This role system greatly simplifies configuration and deployment for the user, as it is the PaaS's responsibility to administer these services. In this scenario, the user has not specified where the AppCaching, Service Bus, and Metadata roles should be run, so the AppControllers place them automatically to enable the system to start correctly. This behavior can be overriden to fail if all roles are not explicitly specified, or customized to allow researchers to consider the performance implications of running more instances of each distributed role in the AppScale PaaS.

Each role that runs within the AppScale PaaS (except the Metadata role) writes metrics about its usage to the Metadata role. Within AppScale, this service is im-

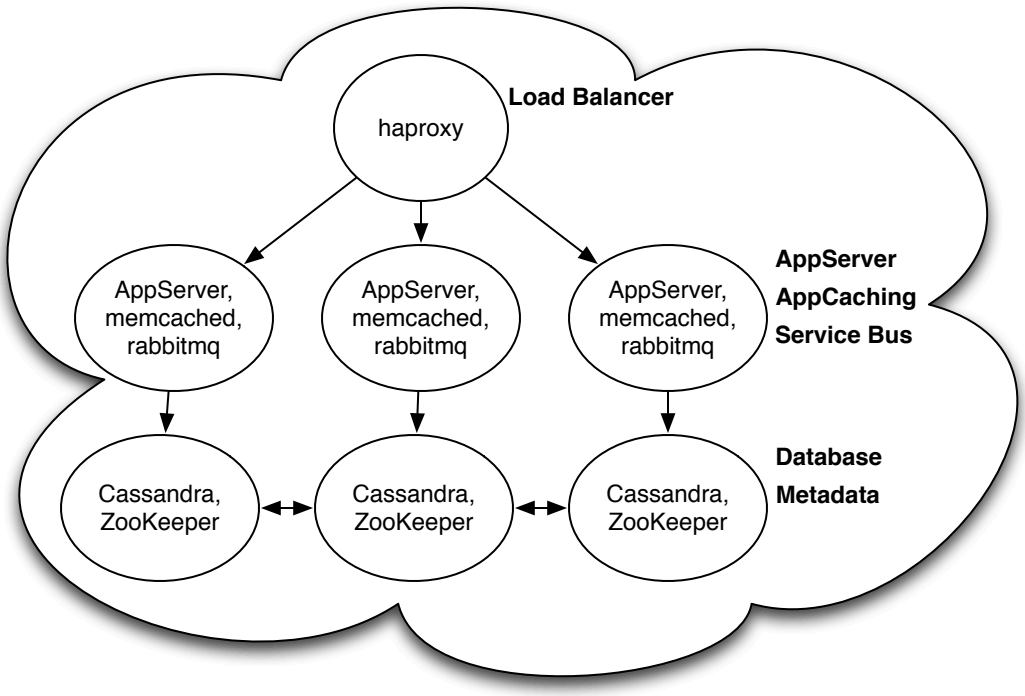

Figure 3.1: A sample placement layout for the AppScale PaaS, where the user has explicitly requested a single Load Balancer, three AppServers, and three Database roles. The AppCaching, Service Bus, and Metadata roles are implicitly added by the AppController if not explicitly placed.

plemented via Apache ZooKeeper, an implementation of the Paxos [65] algorithm based on Google's Chubby service [22]. To maintain correctness for data stored within ZooKeeper, a quorum must be achieved on read and write requests, so a majority of nodes running the Metadata role must always be alive at all times.

3.2.2 Using Role Metadata to Support Pluggable Autoscaling

Storing metrics about every role within the AppScale PaaS enables any role to gather metrics about the current state of the AppScale deployment. As the AppController role is responsible for starting and stopping all other roles within its node, we extend it here to make it also responsible for maintaining all roles within its node. Furthermore, we make AppControllers responsible for maintaining HA within the AppScale PaaS as a whole. Specifically, after each AppController starts all the roles necessary for its own node, it creates a persistent connection with the Metadata service (an **ephemeral link** in ZooKeeper terminology), so that it if that node ever fails, the link will be disrupted and every other AppController will be notified of its failure.

Every AppController then enters a heartbeat loop, where it performs the following activities:

- Write its own metrics to the Metadata service.
- Ask the *autoscaler* if new nodes should be spawned, and if so, how many are required and the roles they should take on.
- Acquire that many nodes, start the AppController role on each of them, and instruct each AppController which roles it should start.

The *autoscaler* is a thread within the AppController that is responsible for making scaling decisions within the AppScale PaaS. Because it can access the Metadata service,

it can view metrics about any role and any node within the AppScale PaaS, and because it runs within the AppController, it can spawn new nodes via the underlying IaaS and configure them accordingly. Specifically, the types of metrics that are available to the autoscaler are:

- Application-level metrics: Information about hosted Google App Engine applications, including their CPU, memory, I/O, and API usage (e.g., datastore, caching, e-mail). The number of application servers serving each application is also available, as well as the average request latency.

- PaaS-level metrics: Information about the virtual machines hosting AppScale. This includes the CPU, memory, disk, and network I/O usage on each machine, as well as role-specific statistics (e.g., Load Balancer usage, Metadata usage) and historical data about previous scheduling decisions (e.g., the times/dates of previous node failures).

3.3 Framework Instantiations

The pluggable autoscaling system designed here can make scaling decisions based on application and PaaS-level statistics. We next detail how certain combinations of these metrics can be utilized to implement autoscaling algorithms to serve complementary use cases within the AppScale PaaS.

3.3.1 HA and QoS-Aware Autoscalers

One autoscaler supported within AppScale is HA. This autoscaler polls the Metadata service for a list of all the nodes that have registered itself as being alive, and looks for persistent connections named after each of those nodes. If any of those connections are missing (e.g., because a node has failed), then the autoscaler polls the metadata service to see which roles that node was hosting and returns that information to the AppController's main thread. The AppController then spawns nodes to take the places of each failed node, with each failed role.

Another autoscaler that is supported within AppScale is QoS enforcement. This autoscaler service polls the Metadata service for data reported by the Load Balancer role (implemented by the `haproxy` daemon) about how many requests have been served in the last t seconds (a customizable value that defaults to 10 seconds) for each AppServer and how many are currently enqueued over the last t seconds. It then uses an exponential smoothing algorithm to forecast how many requests to expect for the next t seconds, via the following formulae:

$$r_0 = 0; q_0 = 0 \tag{3.1}$$

$$r_{t+1} = \alpha * r_{t-1} + (1 - \alpha) * r_t \tag{3.2}$$

$$q_{t+1} = \alpha * q_{t-1} + (1 - \alpha) * q_t \tag{3.3}$$

where r refers to the number of requests served, and where q refers to the number of requests enqueued. If r_{t+1} or q_{t+1} exceed a customizable threshold (defaulting to 5 for each), then the autoscaler decides that an AppServer needs to be added within the system. If there is enough CPU and memory free on any node currently running (metrics reported by each AppController), then the AppServer is added on a currently running node. The CPU and memory thresholds vary for AppServers in different programming languages because the CPU and memory footprints differ significantly between the Python, Java, and Go AppServers, as evaluated in Section 3.4.3.

If there is not enough CPU and memory free on any currently running node, the autoscaler reports that a new node needs to be spawned to host an AppServer role. This autoscaler considers both intra-VM scaling (scaling within a node) and inter-VM scaling (scaling among nodes), in that order. Intra-VM scaling decisions are considered every minute, while inter-VM scaling decisions are considered every 15 minutes (customizable values).

This autoscaler also uses the above formulae to scale AppServers down. If r_{t+1} or q_{t+1} fall below a customizable threshold (defaulting to 5 for each), then the autoscaler determines that an AppServer needs to be removed from its node, as there is not enough traffic to justify the CPU and memory footprint that it consumes.

Finally, it is important to stress that we are not limited to the use of a single autoscaler within the AppScale PaaS. For scenarios when more than one autoscaler is

used, each autoscaler is invoked in the order that the researcher provides. In the open source branch of AppScale that we extend as part of this work, we default to utilizing both the HA-aware and QoS-aware autoscalers. Additionally, the open, pluggable nature of the autoscaler proposed here makes it amenable to the vast amount of existing research on resource scheduling [92][95][86][69].

3.3.2 A Cost-aware Autoscaler

As AppScale operates at the PaaS layer, it is responsible for the acquisition and utilization of IaaS resources (e.g., virtual machines). Therefore, we have the opportunity to provide an autoscaler that can make decisions with not just performance in mind, but also monetary cost. For example, Amazon EC2 charges users on a per-hour basis. If the QoS-aware autoscaler described previously were to decide that resources it acquires are no longer needed, it would terminate them without realizing that keeping the resources until the end of the hour is free under the Amazon pricing model, and that there is no gain from terminating them before this hour price boundary.

We therefore augment the HA-aware, QoS-aware autoscaler used within AppScale to also be cost-aware in the following ways. Whenever a resource would normally be terminated by the QoS-aware autoscaler, it is instead relieved of all of its roles (that is, the stop scripts are called for each role it runs) and the node becomes a hot spare. This hot spare can then be utilized by the HA-aware autoscaler to quickly respond to a

node failure or by the QoS-aware autoscaler to quickly respond to increased web traffic. As we always run the HA-aware autoscaler before the QoS-aware autoscaler, the HA-aware autoscaler gets priority over these machines, but this behavior can be reversed if desired.

Amazon EC2's standard offering provides users with instances in an on-demand fashion. However, they do also offer an auction-style product, Spot Instances [3] (SI), which users acquire by placing bids. If the bid that the user places is above the market price for a particular instance type (classified by CPU and memory), then the user wins the auction and gets the instance. If the market price (dictated by Amazon in an opaque fashion) ever rises above the user's bid, then the resource is reclaimed and the user is refunded for any partial hours used. As these instances can cost substantially less than the standard, on-demand instances, we propose a cost-aware autoscaler. This autoscaler is able to automatically place bids and utilize SIs for both the HA autoscaler and the QoS autoscaler. To avoid losing instances to rising market prices, the cost-aware autoscaler searches through a history of successful bid prices and bids 20% above the average SI price paid (a customizable metric). We evaluate the performance and monetary cost impacts on the AppScale PaaS in Section 3.4.3.

The HA-aware, QoS-aware, and cost-aware autoscaler is open sourced as part of this research. Future work will examine a Azure-aware autoscaler that takes its pricing model into account, as well as the inclusion of Google's new IaaS offering, Compute

Engine [47], which abstractly works in a similar fashion as the Amazon and Microsoft offerings but has different prices associated with the machines it offers.

3.3.3 Manual Administrator Intervention

While the pluggable autoscaling system and sample autoscalers proposed in this work do provide automated resource management within the AppScale PaaS, there may be conditions where a cloud administrator may wish to perform manual scaling. In these scenarios, the cloud administrator typically has some knowledge or metrics that the autoscaler is not aware of and needs to scale up some part of the system.

For example, a company hosting an application may aggressively market their application to the public and thus expect a steep increase of traffic. While the QoS autoscaler above (or variations of it) may be able to reactively deal with the increased amount of traffic, it may drop some traffic before it finishes scaling up. Therefore, the company may want to proactively add AppServers or Database nodes to serve their application.

This work addresses this category of use cases by exposing **autoscaling as a service**, enabling administrators to proactively scale the system up as needed. Specifically, we extend the AppScale command-line tools (similar conceptually to the EC2 tools for AWS) with a new tool, `appscale-add-nodes`. Users give this tool a YAML file that indicates the placement strategy for the new nodes, in a manner identical to that used when starting up AppScale normally. If a user wishes to serve the use case pre-

viously described and add two AppServers and two Database nodes, they could give `appscale-add-nodes` the following placement strategy:

```
---
:node-1:
- appserver
:node-2:
- appserver
:node-3:
- database
:node-4:
- database
```

Users need not learn and specify all the dependencies for each role. Although the user above did not specify where the Service Bus should run, if the AppServer requires it to run on the same node, the AppController will configure and deploy it automatically. Alternatively, users who want to add virtual machines to an AppScale deployment but may not be certain where they could be best utilized can specify that the role be `open`, making it a hot spare that the AppControllers can assign roles to as needed.

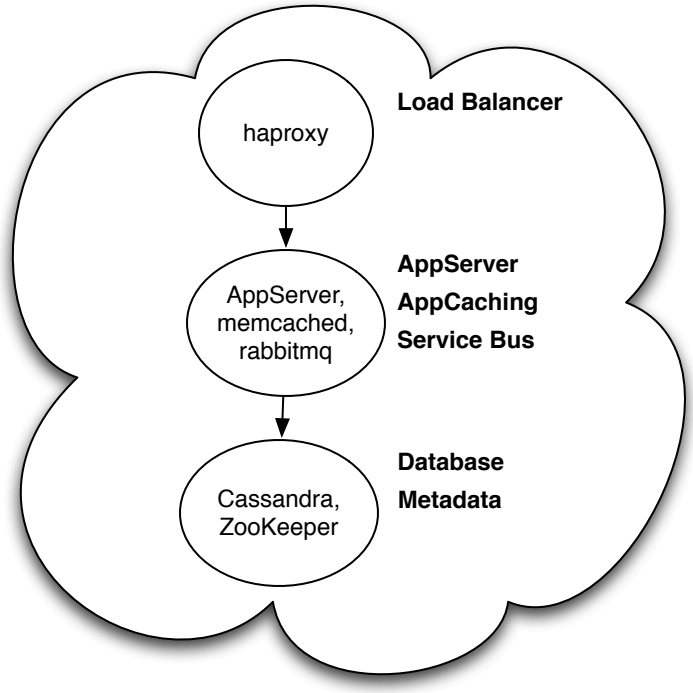

Figure 3.2: Placement strategy used for the experiments in Section 3.4. One node is used to host each role, to maximize the impact of failures on the AppScale PaaS.

3.4 Experimental Evaluation

We next empirically evaluate our proposed autoscalers within AppScale. We begin by presenting our experimental methodology and then discuss our results.

3.4.1 Methodology

To evaluate the pluggable autoscaler system put forth by this work, we use sample Google App Engine applications provided by Google. We use implementations of the standard Guestbook application written in Python and Java. Upon each request to this

application, it queries the database for the most recent posts and displays them to the user. This application is indicative of Google App Engine application usage as a whole, as it utilizes the Datastore and Memcache APIs to serve data dynamically to users.

We host this application on AppScale via the placement strategy shown in Figure 3.2. We intentionally minimize the number of roles implementing each service to maximize the impact of scaling decisions and node failures on the system as a whole. We also utilize Cassandra to implement the Database role, as it is the default within AppScale.

3.4.2 Experimental Autoscaler Results

To evaluate the HA autoscaler, we run AppScale within Amazon EC2 and kill the AppServer running the Python and Java Guestbook applications. Figure 3.3 shows a breakdown of how long it takes for the HA autoscaler to recover from this failure. As a majority of the time is spent acquiring a new virtual machine from EC2, we also use the `appscale-add-nodes` tool to proactively add a hot spare to the system, and find that it significantly reduces the time needed to recover from failures. The presence of a hot spare does not have a significant impact on the other phases in Figure 3.3, however. Furthermore, as the price of an `m1.large` instance (the instance type we use in these experiments) is currently $0.32/hour, having a hot spare always present increases the hourly cost to run this AppScale deployment by 33%, from $0.96 to $1.28. In practice,

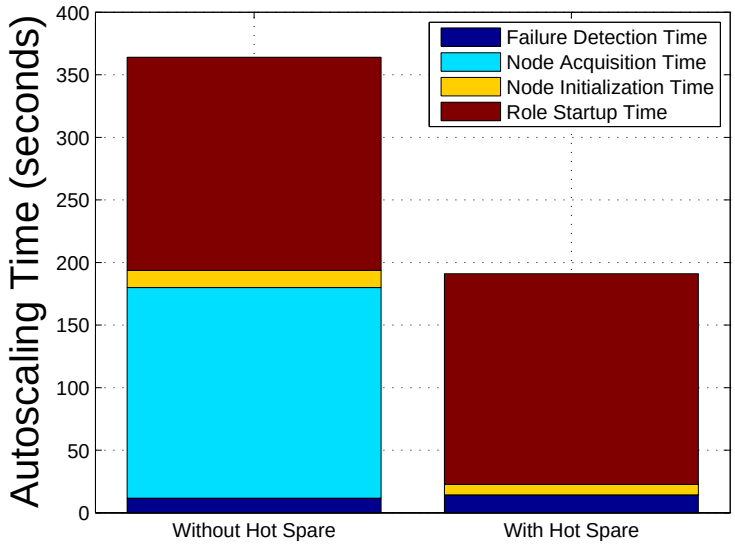

Figure 3.3: Average time for the HA autoscaler to recover from the loss of a single node running the AppServer role within the AppScale PaaS, over the Amazon EC2 IaaS. Recovery time also indicates if a hot spare was available when the failure was detected.

the additional $0.32 incurred to have a hot spare available is likely to be insignificant compared to the opportunity costs due to lost business from the added downtime.

To evaluate the QoS autoscaler, we use the Apache Benchmark tool [1] to dispatch 40,000 web requests to the Python and Java guestbook applications (70 concurrently), and measure how long it takes for AppScale to serve these requests. The results of five runs of this experiment are shown in Figure 3.4. The first bar in each graph measures the time for AppScale to process the 40,000 web requests without the QoS autoscaler, as a baseline set of values.

The second bar in each graph uses the QoS autoscaler and only considers inter-VM scaling. It performs significantly better for the Python Guestbook application (but not for the Java Guestbook application). For both applications, the high request rate and high number of users enqueued at the load balancer cause the QoS autoscaler to quickly acquire more nodes to run AppServer roles, which in turn allows more requests to be served at a time. However, the Java AppServer is faster due to the performance difference between the Java and Python languages, and the Java AppServer is able to use multithreading, while the Python AppServer is limited to a single thread. Although the QoS autoscaler attempts to alleviate this problem by adding AppServers within a virtual machine, it is only able to do it up to a limit (the available CPU and memory on that machine).

Paradoxically, the faster Java AppServer processes the 40,000 web requests before the newly spawned AppServers can have a significant impact (hence the similarities between Java QoS-off and Java QoS-on). To reduce the spawning time of these AppServers and increase their impact, we use the `appscale-add-nodes` command-line tool to add a hot spare to the AppScale deployment before running Apache Benchmark. The results, shown in the third bar, detail a significant improvement for both the Python and Java Guestbook applications when a hot spare is used. Like in the HA autoscaler, the constant presence of a hot spare increases the cost to run the AppScale deployment, but is far less than the costs of business lost due to downtime.

Finally, the fourth bar utilizes the QoS autoscaler to only perform intra-VM scaling. It performs similarly to the scenario where the inter-VM scaler is utilized with a hot spare, and incurs a lower monetary cost (due to not needing the hot spare). Work is ongoing to consider the performance implications of utilizing the inter-VM scaler and intra-VM scaler simultaneously.

3.4.3 Experimental Metrics Results

We next move on to the gathering and reporting of metrics not traditionally considered by autoscaling algorithms, and their use in ongoing research into autoscalers used by the pluggable autoscaling solution proposed here. We begin by examining the CPU and memory footprint of the Python, Java, and Go AppServers, whose information is stored automatically in the Metadata service. The AppController queries the operating system every 30 seconds for this information (a customizable interval), and the average of ten of these queries for steady-state AppServers is shown in Table 3.2.

We begin by noting that the Go AppServer within AppScale utilizes the Python AppServer as a proxy for RPC calls, so the Go AppServer always requires a Python AppServer to be present. Table 3.2 shows both the CPU and memory taken for the standalone Go AppServer and its combined footprint in its production form, when it requires the Python AppServer. As the standalone Go AppServer memory footprint is

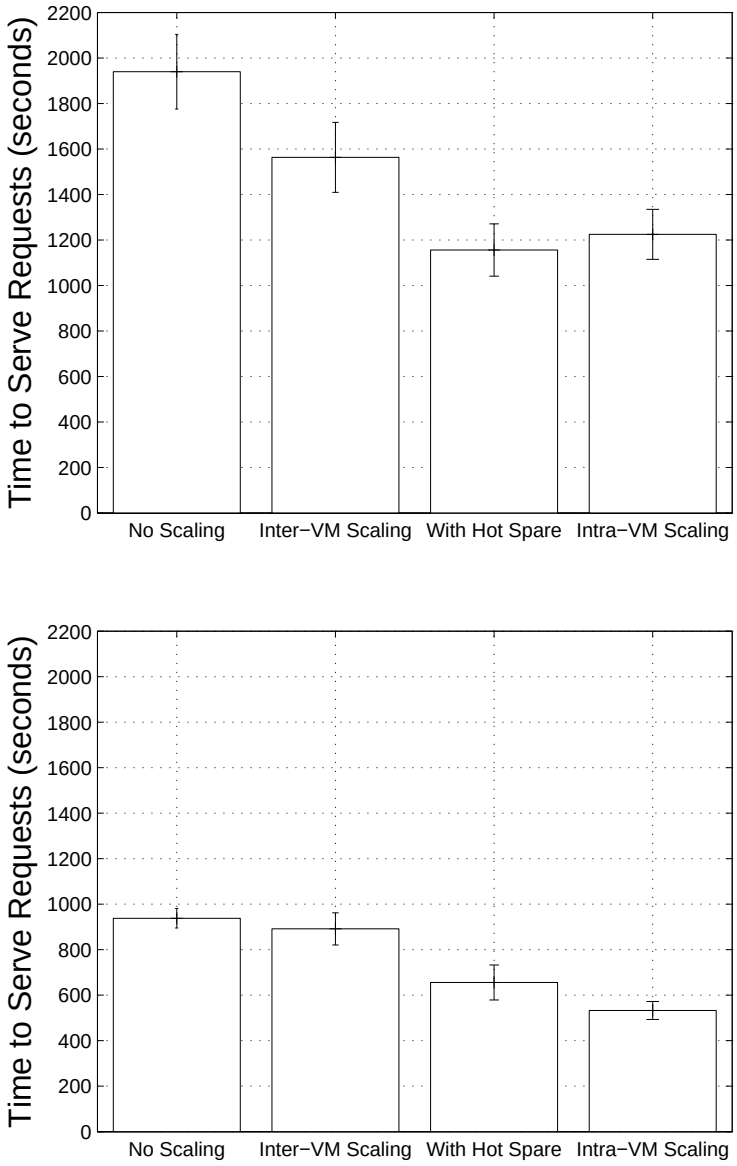

Figure 3.4: Average time for AppScale to serve 40,000 web requests to the Python (Left) and Java (Right) Guestbook applications. We consider the case when the QoS autoscaler is off (the default before this work), when it is on (our contribution), and when we proactively start a hot spare to be used by the QoS autoscaler. Each value represents the average of five runs.

Language	CPU Footprint (%)	Memory Footprint (%)
Python	0.00 ± 0.00	2.40 ± 0.00
Java	0.64 ± 0.08	8.10 ± 0.00
Go	0.00 ± 0.00	0.10 ± 0.00
Go+Python	0.00 ± 0.00	2.50 ± 0.00

Table 3.2: Average CPU and memory footprint consumed by the Python, Java, and Go AppServers while in a steady state. Each value represents the average of ten runs.

significantly less than the Python AppServer memory footprint, we are examining how to remove the dependency on the Python AppServer.

For both the Python and Go AppServers, its steady-state requires no CPU footprint and a significantly smaller CPU and memory footprint than the Java AppServer. However, as Figure 3.4 demonstrated, this smaller footprint may have come at the expense of the ability to handle a production web service workload.

An alternative metric that is simple yet powerful for a PaaS-layer autoscaler to measure is virtual machine startup time compared to cost incurred. Here, we use our cost-aware scheduler to acquire Amazon EC2's on-demand instances and Spot Instances (SIs) automatically (betting 20% above the market price in the manner described previously), and report both the time taken for the instances to boot up and the monetary cost incurred for one hour's use of these machines. Table 3.3 shows the average results of running this experiment ten times.

Table 3.3 shows two clear trends. First, on-demand instances can be acquired extremely quickly, with low variance in both the time and cost incurred to utilize these

Instance Type	Time to Acquire Instances (sec)	Monetary Cost ($)
On-Demand	37.03 ± 1.36	0.3200 ± 0.0000
Spot	411.31 ± 103.61	0.0299 ± 0.0002

Table 3.3: Time and monetary cost incurred for the cost-aware scheduler to utilize Amazon EC2 on-demand and spot instances. These results reflect the average of ten runs, with the `m1.large` instance type in the AWS East Coast region.

machines. Second, the SIs take an order of magnitude longer to acquire, but cost an order of magnitude less. This makes SIs an ideal target to be used as hot spares within the AppScale PaaS, to be aggressively spawned and used to reduce the amount of time needed to recover from failures or ensure a higher QoS.

3.5 Related Work

This work proposes and implements a pluggable autoscaling solution that can be utilized for fault tolerance as well as elasticity. The fields of fault tolerance and elasticity have been well-studied, and a number of research efforts are conceptually similar to the work proposed here.

The VGrADS [77] and MODISAzure [66] projects are two research efforts aimed at providing fault tolerance to e-Science applications running over cloud resources. VGrADS is aimed at using cloud resources according to grid computing's best practices, where resources are acquired in bulk, utilized, and discarded. That is the opposite mentality of the work performed here, where cloud resources are acquired and used ac-

cording to cloud computing's best practices, where resources are acquired elastically, as needed. The difference is notable in a public cloud environment where resources are not free, as compared to the grid environment (where large clusters are available for use, free of charge). Accordingly, VGraDS does not consider the cost of utilizing the Amazon EC2 IaaS, whereas we are cognizant of it and consider the use of Spot instances to reduce the price that end-users pay. MODISAzure is similarly only interested in how clouds can be utilized for their end-users (in their case, the Microsoft Azure IaaS), and is not concerned with the cost of actually using these resources.

CloudScale [86] and [92] focus on developing new elasticity techniques for web applications. Two main differences exist between these works and the pluggable autoscaler system proposed here. First, this work is the first that we know of to actually run within a cloud PaaS. Despite its name, CloudScale does not run within a cloud IaaS or PaaS, but instead utilizes virtual machines managed by the Xen hypervisor (which is necessary but not sufficient for cloud computing). [92] also targets machines running over Xen, but does not use it as a mechanism for research, opting instead to focus on elasticity algorithms customized for the three-tier web deployment strategy (load balancer, application server, and database). Second, these efforts do not seek to provide a pluggable autoscaling solution for researchers to experiment with and test with. They seek to provide novel autoscaling algorithms, and thus do not compete with the system

proposed here. Work is ongoing to adapt the algorithms proposed in these works as autoscalers and evaluate their effectiveness within the AppScale PaaS.

Cloud vendors have also proposed products and solutions to the issues of fault tolerance and elasticity for their users. Amazon EC2 and Microsoft Azure both provide users with virtual machine-level high availability by always running a hot spare [14], although this leaves the responsibility of utilizing that machine in the user's application to the user. RightScale [78], enStratus [40], Scalr [84], and Kaavo [59] seek to provide high availability and elasticity through virtual machine-level metrics (CPU, memory, disk usage), and if the user utilizes LAMP stack components, the health of those components can also be utilized when making scaling decisions. Yet these systems are closed source systems, often only allowing simple rules to be utilized to make scaling decisions (e.g., scale up if load exceeds 30%). In contrast, the open pluggable autoscaling system proposed here is extensible to the years of scheduling research done by the community at large and provides new avenues of research to be performed.

3.6 Summary and Conclusions

The flexible role system and pluggable autoscaler implemented here abstracts away the complexities of deploying and managing applications distributed across machines in cloud services. The motivation behind this pluggable autoscaler is to enable users to

experiment with the vast array of research on autoscaling and experimentally validate how the performance and cost of autoscalers for their application and specific workloads. We utilize this pluggable autoscaler to implement HA and QoS autoscalers, and make them cost-aware when running in a public cloud. We evaluate the performance of the HA and QoS autoscalers under varying workloads, for applications written in different programming languages. This pluggable autoscaler provides users with the ability to easily test autoscalers for their applications, and to quantify how scaling decisions impact their application's performance and cost it incurs to the user that hosts it.

The text of this chapter is, in part, a reprint of the material as it appears in [15].

Chapter 4

Language and Runtime Support for Automatic Configuration and Deployment of Scientific Computing Software over Cloud Fabrics

In this chapter, we present the design and implementation of Neptune, a simple, domain-specific language based on the Ruby programming language. Neptune automates the configuration and deployment of scientific software frameworks over disparate cloud computing systems. Neptune integrates support for MPI, MapReduce, UPC, X10, StochKit, and others. We implement Neptune as a software overlay for the AppScale cloud platform and extend AppScale with support for elasticity and hybrid execution for scientific computing applications. Neptune imposes no overhead on application execution, yet significantly simplifies the application deployment process, enables portability across cloud systems, and promotes lock-in avoidance by specific cloud vendors.

4.1 Introduction and Motivation

Beyond the differences between clouds and grids, there are three barriers to the adoption of cloud computing for the execution of distributed, cluster-based, scientific applications. First, cloud systems currently in use have been designed for the execution of applications from the web services domain. As a result, developers must implement additional services and frameworks to support applications from other domains. Such infrastructure (tools, services, packages, libraries) presents challenges to efficient reuse, and requires non-trivial installation, configuration, and deployment efforts to be repeatable. Second, cloud systems today are vastly diverse between one another, and code written for one system is not easily portable to other cloud systems, despite using common services and APIs provided by the cloud system. Differing interfaces can impose large learning curves and lead to lock-in – the inability to easily move from one cloud system to another. Third, the self-service nature of cloud infrastructures require significant user expertise to manipulate, control, and customize virtual machines (the execution unit of cloud infrastructures), making them inaccessible to all but expert users [57].

The goal of our work is to reduce the real-world impact of these barriers-to-entry and to facilitate greater use of cloud fabrics by the scientific computing community. This is also part of an effort to enable a cost-effective computation alternative to that

of the cluster that is still viable for large scale scientific problems. Toward this end, we present and evaluate Neptune, a domain-specific language for automatically configuring and deploying disparate cloud-based services and applications. Neptune is a high-level language that is a superset of Ruby [81], a dynamic, open source programming language that is easy to learn and facilitates programmer productivity. Neptune adds to Ruby a series of keywords and constructs that developers use to describe a computational job at a very high level. Neptune executes any Ruby code using the Ruby interpreter and uses this job description along with a set of API calls to build, configure, and deploy the services, libraries, and virtual machines necessary for the distributed execution of complex scientific applications and systems over cloud platforms. Neptune abstracts away all of the low level details of the underlying cloud platforms (and by extension, cloud infrastructures) and provides a single, simple interface with which developers can deploy their applications. Neptune thus enables application portability across clouds and precludes lock-in to any single cloud vendor. Moreover, developers can use Neptune to employ multiple clouds concurrently (hybrid cloud computing), without application modification.

To enable this, Neptune interfaces to the AppScale [27, 28, 64] cloud platform. AppScale is a distributed, scalable software system that exposes a set of popular cloud service APIs (based on those of Google App Engine), and executes over the Amazon Web Services and Eucalyptus [71] clouds.

In this paper we present the design and implementation of Neptune, as well as a set of AppScale extensions that enable automatic configuration and deployment of scientific applications. These extensions include dynamic instantiation of virtual machines, placement of application and cloud service components within virtual machines for elasticity, and hybrid cloud computing. We extend AppScale with a set of popular software systems that are employed by a wide range of scientific application domains, such as MPI, UPC, and MapReduce for general-purpose HPC, as well as more science-specific toolkits such as StochKit [83] for stochastic biochemical simulation, DFSP [36] for spatial stochastic biochemical simulation, and dwSSA [31] for the estimation of rare event probabilities. Moreover, Neptune's design makes it straightforward for users to add additional frameworks, libraries, and toolkits.

In the sections that follow, we describe the design and implementation of Neptune, and our extensions to the AppScale cloud platform. We then empirically evaluate Neptune using distributed HPC frameworks, stochastic simulation applications, and different placement strategies. We then present related work and conclude.

4.2 Neptune

Neptune is a domain-specific language that gives cloud application developers the ability to easily configure and deploy computational science software over cloud sys-

tems. Configuration refers to writing the configuration files that HPC software requires to execute in a distributed fashion, while deployment refers to starting HPC services in the correct order, to enable user code to be executed. Neptune operates at the cloud platform layer (runtime system level) so that it can control infrastructure-level entities (virtual machines) as well as application components and cloud services.

4.2.1 Syntax and Semantics

The Neptune language is a metaprogramming extension of the Ruby programming language. As such, it is high-level and familiar, and can leverage a large set of Ruby libraries to interact with cloud infrastructures and platforms. Moreover, any legal Ruby code is also legal within Neptune programs, enabling users to use Ruby's scripting capabilities to quickly construct functioning programs. The reverse is also true: Neptune can be used within Ruby programs, to which it appears to users as a library that can be utilized in the same fashion as other Ruby libraries.

Neptune uses a reserved keyword (denoted throughout this work via the `neptune` keyword) to identify services within a cloud platform. Legal Neptune code obeys the following syntax, where e represents the empty string:

```
S -> neptune :type => T


T -> mpi, M1 | x10, M1 | upc, M1 | mapreduce, R1
```

```
T -> erlang , E1 | ssa , S1 | appscale , P1

T -> set-acl , A1 | get-acl , G1 | get-output , O1 | compile , C1


M1 -> :code => 'location', M2

M2 -> :nodes_to_use => int , M3

M3 -> :output => 'location', M4

M3 -> :output => 'location', M5

M4 -> :procs_to_use => int , M5

M5 -> K1 | e


R1 -> :input => 'location', R2

R2 -> :output => 'location', R3

R3 -> :nodes_to_use => int , R4

R4 -> :mapreducejar => 'location', R5

R5 -> :main => 'classname', R6

R6 -> K1 | e


E1 -> :code => 'location', E2

E2 -> :output => 'location', E3
```

```
E3 -> :nodes_to_use => int, E4

E4 -> K1 | e


S1 -> :nodes_to_use => int, S2

S2 -> :tar => 'location', S3

S2 -> :tar => 'location', S4

S3 -> :simulations => int, S5

S4 -> :trajectories => int, S5

S5 -> :output => 'location', S6

S6 -> K1 | e


K1 -> :storage => 'appdb',

K1 -> :storage => 's3', K2

K1 -> :storage => 'gstorage', K2

K1 -> :storage => 'walrus', K2

K2 -> :EC2_ACCESS_KEY => 'key', K3

K3 -> :EC2_SECRET_KEY => 'key', K4

K4 -> :S3_URL => 'url'
```

```
P1 -> :nodes_to_use => {P2}, P3

P2 -> :cloud int => int, P3

P3 -> :add_component => 'load_balancer', P4

P3 -> :add_component => 'appengine', P4

P3 -> :add_component => 'database', P4

P4 -> :time_needed_for => float


A1 -> :output => 'location', A2

A2 -> :acl => 'public'

A2 -> :acl => 'private'


G1 -> :output => 'location'


O1 -> :output => 'location', O2

O1 -> :output => 'location'

O2 -> :save_to_local => 'location'


C1 -> :code => 'location', C2

C2 -> :main => 'file', C3
```

```
C3 -> :output => 'location', C4

C4 -> :copyto => 'location'
```

The semantics of the Neptune language are as follows: each valid Neptune program consists of one or more `neptune` invocations, each of which indicate a job to run in a cloud. The `type` marker indicates the name of the job (e.g., MPI, X10), and is associated with a set of parameters that are necessary for the given invocation. This design choice is intentional: not all jobs are created equal, and while some jobs require little information be passed, other job types can benefit greatly from increased information. As a further step, we leverage Ruby's dynamic typing to enable the types of parameters to be constrained by the developer. If the user specifies a Neptune job but fails to provide the necessary parameters, Neptune informs them which parameters are required and aborts execution.

The value that the invocation returns is also extensible, but by default, a Ruby hash is returned, whose items are job specific. In most cases, this hash contains a key named `:success` whose Boolean value corresponds to whether or not the request succeeded. Other scenarios allow for additional parameters to be included. For example, in the scenario where the invocation asks for the access policy for a particular piece of data stored in the underlying cloud platform, there is an additional key named `:acl` whose value is the current data access policy.

Finally, when the user wishes to retrieve data via a Neptune job, the invocation returns the location on the user's filesystem where the output can be found. Work is in progress to expand the number of failure messages to give users more information about why particular operations failed (e.g., if the data storage mechanism was unavailable or had failed, or if the cloud platform itself was unreachable in a reasonable amount of time), to enable Neptune programs written by users to become robust, and to adequately deal with failures at the cloud level. The typical format of a user's Neptune code is thus of the following form:

```
result = neptune :type => :mpi,
 :code => '/code/powermethod',
 :nodes_to_use => 4
if result[:success]
 puts 'Your MPI job is now in progress.'
else
 puts 'Your MPI job failed to start.'
end
```

4.2.2 Design Choices

It is important to contrast the decision to design Neptune as a domain specific language with other configuration options that use XML or other markup languages [63].

These languages work well for configuration but, since they are not Turing-complete programming languages, they are bound to their particular execution model. In contrast, Neptune's strong binding to the Ruby programming language enables users to leverage Neptune and its HPC capabilities to easily incorporate it into their own codes. For example, Ruby is well known for its Rails web programming framework [82], and Neptune's interoperability enables Rails users to easily spawn instances of scientific software without explicit knowledge of how Neptune or the scientific computing software operates.

Markup and workflow languages are powerful in the types of computation that they enable. Similarly, Neptune allows arbitrary computation to be connected and chained to one another. The following example shows how the output of a MapReduce job can be used as the input to a X10 job. Here, the MapReduce job produces a graph representing links between web pages, while the X10 code takes this graph and performs a shortest-path algorithm from all nodes to one another. As Neptune does not automatically resolve data dependencies between jobs, we manually delay the execution of the X10 job until after the MapReduce job has completed and generated its output.

```
neptune :type => :mapreduce,
 :input => '/rawdata/webdata',
 :output => '/output/mrgraph',
 :mapreducejar => '/code/graph.jar',
```

```
 :main => 'main',
 :nodes_to_use => 64


# wait for the mapreduce job to finish
loop {
 result = neptune :type => :get-output,
  :output => '/output/mrgraph'
 if result[:success]
   break
 end
 sleep(60)
}


neptune :type => :mpi,
 :input => '/output/mrgraph',
 :output => '/output/shortestpath',
 :code => '/code/ShortestPath',
 :nodes_to_use => 64
```

To enable code reuse, we allow operations to be reused across multiple Neptune job types. For example, retrieving data and setting ACLs on data are two operations that occur throughout all the job types that Neptune supports. Thus, the Neptune runtime enables these operations to share a single code base for the implementation of these functions. This feature is optional: not all software packages may support ACLs and a unified model for data output, so Neptune gives developers the option to implement support for only the features they require, and the ability to leverage existing support as needed.

4.3 Implementation

To enable the deployment of Neptune jobs, the cloud platform must support a number of primitive operations. These operations are similar to those found in computational grid and cluster utilities, such as the Portable Batch System [75]. The cloud platform must be able to receive Neptune jobs, acquire computational resources to execute jobs on, run these jobs asynchronously, and place the output of these jobs in a way that enables users to retrieve them later or share them with other users. For this work, we employ the AppScale cloud platform to add these capabilities.

AppScale is an open-source cloud platform that implements the Google App Engine APIs. Users deploy applications using AppScale via either a set of command-line tools

or a web interface. An AppScale cloud consists of one or more distributed database components, web servers, and a monitoring daemon (the AppController) that coordinates services across nodes in the AppScale cloud. AppScale implements a wide range of datastores for its database interface via popular open source technologies. As of its most recent release (AppScale 1.5), it includes support for HBase, Hypertable, MySQL Cluster, Cassandra, Voldemort, MongoDB, MemcacheDB, Scalaris, and Amazon SimpleDB. AppScale runs over virtualized and un-virtualized cluster resources as well as over the Amazon EC2 and Eucalyptus cloud infrastructures. The full details of AppScale are described in [28, 16].

The execution of a Neptune job follows the pattern shown in Figure 4.1. The user invokes the `neptune` executable on a Neptune script they have written, which results in a SOAP message being sent to the Neptune runtime (a separate thread in AppScale's AppController service). In the case of a compute job, the Neptune runtime acquires nodes to run the code over, configures them for use, and executes the code, storing the output for later retrieval. In the case of a data input or output job, the Neptune runtime stores or retrieves the data via the datastore.

In this section, we overview the AppScale components that are impacted by our extensions enabling customized placement, automatic scaling, and Neptune support, the AppScale command-line tools and the AppController

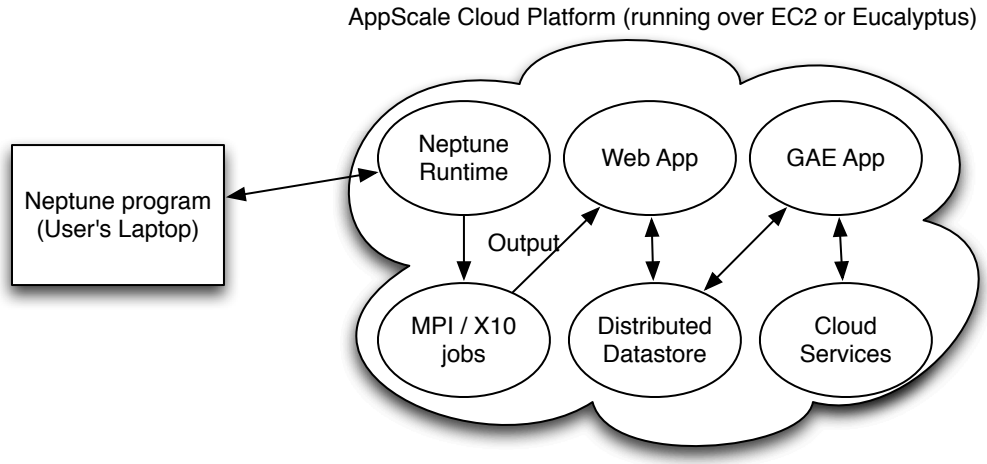

Figure 4.1: AppScale cloud platform with Neptune configuration and deployment support.

4.3.1 Cloud Support

Our extensions to AppScale facilitate interoperation with Neptune. In particular, we modify AppScale to acquire and release machines used for computation, and to enable static and dynamic service placement. To do so, we modify two components within AppScale: the AppScale Tools and the AppController.

AppScale Tools

The AppScale Tools are a set of command-line tools that developers and administrators can use to manage AppScale deployments and applications. In a typical deployment, the user writes a configuration file specifying which node in the system is the "master" node and which nodes are the "slave" nodes. Prior to this work, this

meant that the master node always deployed a Database Master (or Database Peer for peer-to-peer databases) and AppLoadBalancer to handle and route incoming user requests, while slave nodes always deployed a Database Slave (or Database Peer) and AppServers hosting the user's application.

We extend this configuration model to enable users to provide a configuration file that identifies which nodes in the system should run each service (e.g., Database Master, Database Slave, AppLoadBalancer, AppServer). For example, users can specify that they want to run each service on a dedicated machine by itself. Alternatively, users could specify that they want their database nodes running on the same machines as their AppServers, and have all other components running on another machine. We also allow users to designate certain nodes in the system as "open", which tells the AppController that this node is free to use for Neptune jobs (a hot spare).

We extend this support to enable hybrid cloud deployment of AppScale, in which nodes are not limited to a single cloud infrastructure. Here, users specify which nodes belong to each cloud infrastructure, and then export environment variables that correspond to the credentials needed for each cloud. This is done to mirror the styles used by Amazon EC2 and Eucalyptus. One potential use case of this hybrid cloud support is for users who have a small, dedicated Eucalyptus deployment and access to Amazon EC2: these users could use their Eucalyptus deployment to test and optimize their code, and deploy to Amazon EC2 when more nodes are needed. Similarly, Neptune

users can use hybrid cloud support to run jobs in multiple availability zones simultaneously, providing them with the ability to run computation as close as possible to their data. For scenarios where the application to be deployed is not a compute-intensive application (e.g., web applications), it may be beneficial to ensure that instances of the application are served in as many availability zones as possible, to ensure that users always have access to a nearby instance. This deployment strategy gives users some degree of fault-tolerance, in the rare cases when an entire availability zone is down or temporarily inaccessible [54].

AppController

The AppController is a monitoring service that runs on every node in an AppScale deployment. It configures and instantiates all necessary services, which typically involves starting databases and running Google App Engine applications. AppControllers also monitor the status of each service it runs, and periodically send heartbeat messages to other AppControllers to aggregate this information. It currently queries each node to learn its CPU, memory, and hard drive usage, although it is extensible to collecting other metrics.

Our extensions enable the AppController to receive and understand RPC (via SOAP) messages from Neptune and to coordinate Neptune activities across other nodes in an AppScale deployment. Computational jobs and requests for output data run asyn-

chronously within AppScale, and do not block the user's Neptune code. All Neptune requests are authenticated with a secret established when starting AppScale, and are performed over SSL to prevent request sniffing.

If running in hybrid cloud deployments, AppScale spawns machines for each cloud in which the user has requested machines, with the credentials that the user has provided. Any hot spares (machines indicated as "open") are acquired before new nodes are spawned. The AppController records which cloud each machine runs in, so that Neptune jobs can ask for nodes within specific cloud or more than one cloud. Additionally, as cloud infrastructures currently meter on a per-hour basis, we have modified the AppController to be cognizant of this and reuse virtual machines between Neptune jobs. Within AppScale, any virtual machine that is not running a Neptune job at the 55-minute mark is terminated; all other machines are renewed for another hour.

Administrators query AppScale via either the AppScale Tools or the web interface provided by the AppLoadBalancer. These interfaces inform administrators about the jobs in progress and, in hybrid cloud deployments, which clouds are running which jobs. These interfaces do not actually run Neptune jobs or interact with them, but simply describe their status as reported to them by the AppController.

A perk of offering this service at the cloud platform layer is that the platform can profile the usage patterns of the underlying system and act accordingly (since a well-specified set of APIs are offered to users). We provide customizable scheduling mecha-

nisms for scenarios when the user is unsure how many nodes are required to achieve optimal performance. This use case is unlikely to occur for highly tuned codes, but more likely to occur within HTC and MTC applications, where the code may not be as well tuned for high performance. Users only need specify how many nodes the application can run over, a required parameter because Neptune does not perform static analysis of the user's code, and oftentimes specific numbers of nodes are required (e.g., powers of two). Neptune then employs a hill-climbing algorithm to determine how many machines to acquire: given an initial guess, Neptune acquires that many machines and runs the user's job, recording the total execution time for later use. On subsequent job requests, Neptune tries the next highest number of nodes, and follows this strategy until the execution time fails to improve. Our initial release of Neptune provides scheduling based on total execution time, total cost incurred (e.g., acquire more nodes only if it costs less to do so), or a weighted average of the two. This behavior is customizable, and is open to experimentation via alternative schedulers.

More appropriate to scientists using cloud technologies is the ability to automatically choose the type of instance acquired for computation. Cloud infrastructure providers offer a wide variety of machines, referred to as "instance types", that differ in terms of cost and performance. Inexpensive instance types offer less compute power and memory, while more expensive instance types offer more compute power and memory. If the user does not specify an instance type to use, Neptune will automatically acquire a

compute-intensive instance. A benefit of this strategy is that since these machines are among the more expensive machines available, the virtual machine reuse techniques we employ amortize their cost between multiple users for jobs that do not run in 60-minute increments (the billing quantum used in Amazon EC2).

AppServer

The AppServer is a modified version of the Google App Engine SDK that runs a user's App Engine application. Applications can be written in Python, Java, or Go, and can utilize APIs that provide a variety of features, including storage capabilities (via the Datastore and Blobstore) and communication capabilities (via Mail and XMPP).

For this work, we modify the AppServer to add an additional API: the Neptune API. This API allows users to initiate Neptune jobs from within App Engine applications hosted on AppScale, and thus provides a mechanism by which web applications can execute high performance computation. This also opens up HPC to greater audiences of users, including those who want to run their codes from different types of platforms (e.g., via their smartphone or tablet computer).

4.3.2 Job Data

Clouds that run Neptune jobs must allow for data stored remotely to be imported and used as job inputs. Jobs can consume zero or more files as inputs, but always

produce exactly one piece of output, a string containing the standard out generated by the executed code. Neptune refers to data as three-tuple: a string containing the job's identification number, a string containing the output of the job, and a composite type indicating its access policy. The access policy used within Neptune is similar to that of the access policy used by Amazon's Simple Storage Service [5]: a particular piece of data can be tagged as either private (only visible to the user that uploaded it) or public (visible to anyone). Data is by default private but can be changed by the user, via a Neptune job. Similarly, data is referenced as though it were on a file-system: paths must begin with a forward-slash ('/') and can be compartmentalized into folders in the familiar manner. The data itself is accessed via a Google App Engine application that is automatically started when AppScale starts, and can be stored internally via AppScale or externally via Amazon S3. This allows jobs to automatically save their outputs in any datastore that AppScale supports, or any service that is API-compatible with Amazon S3 (e.g., Google Storage, Eucalyptus Walrus). The Neptune program to set the ACL of a particular piece of data to be public is:

```
neptune :type => 'set-acl',
 :output => '/mydata/nqueens-output',
 :acl => 'public'
```

Just as a Neptune job can be used to set the ACL for a piece of data, a Neptune job can also be used to get the ACL for a piece of data:

```
acl_data = neptune :type => 'get-acl',
 :output => '/mydata/nqueens-output'
puts 'The current ACL is:' + acl_data[:acl]
```

Retrieving the output of a given job is also done via a Neptune job. By default, it returns a string containing the results of the job. As many jobs return data that is far too large to efficiently be used in this manner, a special parameter can be used to instead indicate that it should be copied to the local machine. The following Neptune code illustrates both use cases (note that the # character is Ruby's comment character):

```
# for a job with small output
result = neptune :type => 'get-output',
 :output => '/mydata/boo'
puts 'Output is: ' + result[:output]
# for a job with much larger output
result = neptune :type => 'get-output',
 :output => '/mydata/boo-large',
 :save_to_local => '/shared/boo-large.txt'
if result[:success]
 puts 'Output copied successfully.'
end
```

4.3.3 Employing Neptune for HPC Frameworks

To support HPC applications within cloud platforms, we *service-ize* them for use via Neptune. Specifically, Neptune provides support for MPI, X10, MapReduce, UPC, and Erlang, to enable users to run arbitrary codes for different computational models. While these general purpose languages and frameworks are useful for the scientific community as a whole, Neptune also seeks to engender support from the biochemical simulation community. These groups of HPC perform simulations via kinetic Monte Carlo methods (specifically, the Stochastic Simulation Algorithm), and often need to run a large number of these simulations (on a minimum order of 10^5) to gain statistical accuracy. Neptune supports use of StochKit, a general purpose SSA implementation, as well as DFSP and dwSSA, two specialized SSA implementations.

As users may not have these libraries and runtimes installed locally, Neptune also provides the ability to remotely compile their code (required for the non-SSA computational models), and is extensible to support non-compute intensive application domains, such as web services.

MPI

The Message Passing Interface (MPI) [48] is a popular, general purpose computational framework for distributed scientific computing. The most popular implementation is written in a combination of C, C++, and assembly. Implementations exist for

many other programming languages, such as Fortran, Java, and Python. AppScale employs the C/C++ version, enabling developers to write code in either of these languages to access MPI bindings within AppScale. The developer uses Neptune to specify the location of the compiled application binary and output data, and this information is sent from Neptune to the AppController.

Following the MPI execution model, one compute node is designated as a master node, and all other nodes are referred to as slave nodes. The master node starts up NFS on all nodes, mounts a shared filesystem on all slave nodes, runs `mpdboot` on its own node, and executes the user's code on its node via `mpiexec`, piping the output of the job to a file on its local filesystem. Once it has completed, the master node runs `mpdallexit` and stores the standard output and standard error of the job (the results) in the database that the user has requested, for later retrieval. An example of how a user would run an MPI job is as follows:

```
neptune :type => :mpi,
 :code => '/code/powermethod',
 :nodes_to_use => 4,
 :output => '/output/powermethod.txt'
```

In this example, we specify the location where the compiled code to execute is located (stored via a previous Neptune job). The user also indicates how many machines are required to run their MPI code and where the output of the job should be placed.

Note that this program does not use any inputs, nor need to write to any files on disk as part of its output. Neptune can be extended to do so, if necessary. We also can designate which shared file system to use when running MPI. Currently, we support NFS and are working on support for the Lustre Distributed File System [68].

We also note that many HPC applications require a high performance, low latency interconnect. If running over Amazon EC2, users can acquire this via the Cluster Compute Instances they provided, and in Eucalyptus, a cloud can be physically constructed with the required network hardware. If the user does not have access to this type of hardware, and their program requires it, their program may suffer from degraded performance, or may not run at all.

X10

While MPI is suitable for many types of application domains, one demand in computing has been to enable programmers to write fast, scalable code using a high-level programming language. In addition, as many years of research have gone into optimizing virtual machine technologies, it is also desirable for a new technology to be able to leverage this work. In this spirit, IBM introduced the X10 programming language [25], which uses a Java-like syntax, and can execute transparently over either a non-distributed Java backend or a distributed MPI backend. The Java backend enables

developers to develop and test their code quickly, and utilize Java libraries, while the MPI backend allows the code to be run over as many machines as the user can acquire.

As X10 code can compile to executables for use by MPI, X10 jobs are reducible to MPI jobs. Thus the following Neptune code deploys an X10 executable that has been compiled for use with MPI:

```
neptune :type => :mpi,
 :code => '/code/NQueensDist',
 :nodes_to_use => 2,
 :output => '/output/nqueensx10.txt'
```

With the combination of MPI and X10 within Neptune, users can trivially write algorithms in both frameworks and (provided a common output format exists) compare the results of a particular algorithm to ensure correctness across implementations. One example used in this paper is the $n - queens$ algorithm [80], an algorithm that, given an chess board of size $n \times n$, determines how many ways n queens can be placed on the board without threatening one another. The following Neptune code illustrates how to verify the results produced by an MPI implementation against that of an X10 implementation (assuming both codes are already stored remotely):

```
# run mpi version
neptune :type => :mpi,
```

```
  :code => '/code/MpiNQueens',
  :nodes_to_use => 4,
  :output => '/mpi/nqueens'


# run x10 version
neptune :type => :mpi,
  :code => '/code/X10NQueens',
  :nodes_to_use => 4,
  :output => '/x10/nqueens'


# wait for mpi version to finish
loop {
  mpi_data = neptune :type => 'output',
    :output => '/mpi/nqueens'
  if mpi_data[:success]
    break
  end
  sleep(60)
}
```

```
# wait for x10 version to finish
loop {
 x10_data = neptune :type => 'output',
  :output => '/x10/nqueens'
 if x10_data[:success]
  break
 end
 sleep(60)
}

if mpi_data[:output] == x10_data[:output]
 puts 'Output matched!'
else
  puts 'Output did not match.'
end
```

Output jobs return a hash containing a `:success` parameter, indicating whether or not the output exists. We leverage this to determine when the compute job that generates this output has finished. The `:output` parameter in an `output` job contains a string

corresponding to the standard out of the job itself, and we use Ruby's string comparison operator (==) to compare the outputs for equality.

MapReduce

Popularized by Google in 2004 for its internal data processing [33], the map-reduce programming paradigm (MapReduce) has experienced a resurgence and renewed interest. In contrast to the general-purpose message passing paradigm embodied in MPI, MapReduce targets embarrassingly parallel problems. Users provide input, which is split across multiple instances of a user-defined Map function. The output of this function is then sorted based on a key provided by the Map function, and all outputs with the same key are given to a user-defined Reduce function, which typically aggregates the data. As no communication can be done by the user in the Map and Reduce phases, these programs are highly amenable to parallelization.

Hadoop provides an open-source implementation of MapReduce that runs over the Hadoop Distributed File System (HDFS) [50]. The standard implementation requires users to write their code in the Java programming language, while the Hadoop Streaming implementation faciliates writing code in any programming language. Neptune has support for both implementations. Users provide a Java archive file (JAR) for the standard implementation, or Map and Reduce applications for the Streaming implementation.

AppScale retrieves the user's files from the desired data storage location, and runs the job on the Neptune-specified nodes in the system. In particular, the AppController contacts the Hadoop JobTracker node with this information, and polls Hadoop until the job completes (indicated by the output location having data written to it). When this occurs, Neptune copies the data back to a user-specified location. From the user's perspective, the necessary Neptune code to run code written with the standard MapReduce implementation is:

```
neptune :type => :mapreduce,
 :input  => '/input/input-text.txt',
 :output => '/output/mr-output.txt',
 :mapreducejar => '/code/example.jar',
 :main => 'wordcount',
 :nodes_to_use => 4
```

As was the case with MPI jobs, the user specifies where the input to the MapReduce job is located, where to write the output to, and where the code to execute is located. Users also specify how many nodes they want to run their code over. AppScale normally stores inputs and outputs in a datastore it supports or Amazon S3, but for MapReduce jobs, it also supports the Hadoop Distributed File System (HDFS). This can result in Neptune copying data to HDFS from S3 (and vice-versa), but an extra pa-

rameter can be used to indicate that the input already exists in HDFS, to skip this extra copy operation.

Unified Parallel C

Unified Parallel C [38] is a superset of the C programming language that aims to simplify HPC applications via the Partitioned Global Address Space (PGAS) programming model. UPC allows developers to write applications that use shared memory in lieu of the message passing model that other programming languages offer (e.g., MPI). UPC also can be deployed over a number of runtimes; some of these backends include specialized support for shared memory machines as well as optimized performance when specialized networking equipment is available. UPC programs deployed via Neptune can use any backend supported by the underlying cloud platform, and as we use AppScale in this work, three backends are available: the SMP backend, optimized for single node deployments, the UDP backend, for distributed deployments, and the MPI backend, which leverages the mature MPI runtime.

UPC code can be deployed in Neptune in a manner analogous to that of other programming languages. If a UPC backend is not specified in a `Makefile` with the user's code, the MPI backend is automatically selected. As we have compiled our code with the MPI backend, the Neptune code needed is identical to that used in MPI deployments:

```
result = neptune :type => :mpi,
 :code => '~/ring-compiled/Ring',
 :nodes_to_use => 4,
 :procs_to_use => 4,
 :output => '/upc/ring-output'

# inspect is Ruby's method to print a hash
puts result.inspect
```

As shown here, users need only specify the location of the executable, how many nodes to use, and where the output should be placed. We extend the MPI support that Neptune offers to enable users to specify how many processes should be spawned. This allows for deployments where the number of processes is greater than that of the number of available nodes (and are thus overprovisioned), and can take advantage of scenarios where the instance types requested have more than a single core present.

Erlang

Erlang [9] is a concurrent programming language developed by Ericsson that uses a message passing interface similar to that of MPI. While other HPC offerings try to engender a larger user community by basing their language's syntax, semantics, or runtime on that of C or Java (e.g., MPI, UPC, and X10), Erlang does not. The stated

reason for this is that Erlang seeks to optimize the user's code via the single assignment model, which enables a higher degree of compile-time optimization than the model used by C-style languages.

While Erlang's concurrent programming constructs extend to distributed computing, Erlang does not provide a parallel job launcher analogous to those provided by MPI (via `mpiexec`), Hadoop MapReduce, X10, and UPC. These job launchers do not require the user to hardcode IP addresses in their code, as is required by Erlang programs.

Due to this limitation, we support only the concurrent programming model that Erlang offers. We are currently investigating ways to automate the process for the distributed version. Users write Erlang code with a `main` method, as is standard Erlang programming practice, and this method is then invoked by the AppScale cloud platform on a machine allocated for use with Erlang.

The Neptune code needed to deploy a piece of Erlang code is similar to that of the other supported languages:

```
neptune :type => :erlang,
 :code => '~/ring-compiled/ring.beam',
 :output => '/erlang-output.txt',
 :nodes_to_use => 1
```

In this example, we specify that we wish to use a single node, the path on the local filesystem where the compiled code can be found, and where the output of the execution should be placed.

Compilation Support

Before MPI, X10, MapReduce, UPC, or Erlang jobs can be run, they require the user's code to be compiled. Although the target architecture (the machines that AppScale runs over) may be the same as the architecture that the scientist has compiled their code on, it is not guaranteed to be so. It is therefore necessary to offer remote compilation support, so that no matter what platform the user runs, whether it be a 32-bit laptop, a 64-bit server, or even a tablet computer that has a text editor and internet connection, code can be compiled and run. The Neptune code required to compile a given piece of source code is:

```
result = neptune :type => :compile,
 :code => '~/ring',
 :main => 'Ring.x10',
 :output => '/output/ring',
 :copyto => '~/ring-compiled'

puts result.inspect
```

This Neptune code requires the user to indicate only where their code is located and which code is the main executable (as opposed to being a library or other ancillary code). Scientists may provide `makefile`s if they like. If they do not, Neptune attempts to generate one for them based on the file's extension or its contents. Neptune cannot generate `makefile`s for all scenarios, but can do so for many scenarios where the user may not be comfortable with writing a `makefile`.

StochKit

To enable general purpose SSA programming support for scientists, Neptune provides support for StochKit, an open source biochemical simulation software package. StochKit provides stochastic solvers for several variants of the Stochastic Simulation Algorithm (SSA), and provides the mechanisms for the stochastic simulation of arbitrary models. Scientists describe their models by specifying them in the Systems Biology Markup Language (SBML) [58]. In this work, we simulate a model included with StochKit, known as `heat-shock-10x`. This model is a ten-fold expansion of the system that models the heat shock response in *Escherichia coli* [39]. Figure 4.2 shows results from a statistical analysis on an ensemble of simulated trajectories from this model.

Typically scientists utilizing the SSA run a large number of simulations to ensure enough statistical accuracy in their results. As the number of simulations to run may

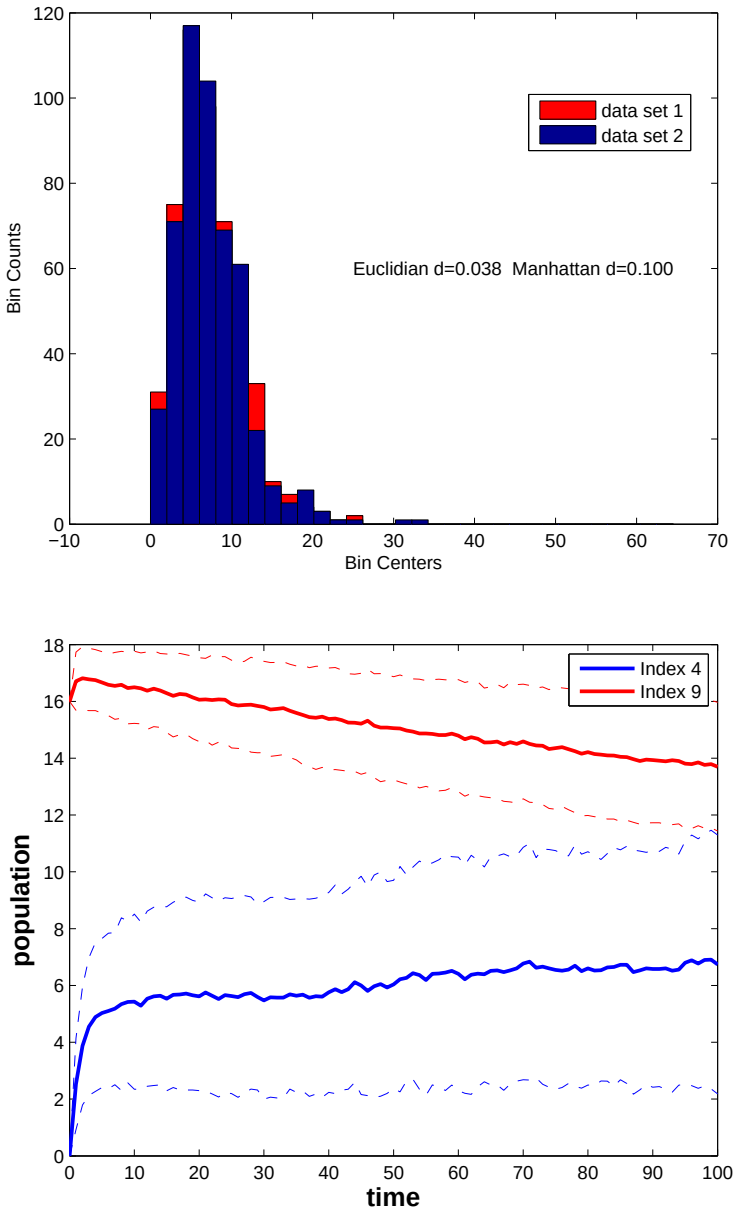

Figure 4.2: Plots showing statistical results from StochKit stochastic simulations of the heat shock model. (Left) Comparison of probability density histograms from two independent ensembles of trajectories, and the histogram distance between them. The histogram self-distance is used to determine the confidence for a given ensemble size. (Right) Time-series plots of the mean (solid lines) and standard-deviation bounds (dashed lines) for two biochemical species.

not be known a priori, scientists often have to run a number of simulations, see if the requested confidence level has been achieved, and if this has not occurred, the process repeats. The Neptune code required to do this is trivial:

```
confidence_needed = 0.95
i = 0
loop {
 neptune :type => :ssa,
  :nodes_to_use = 4,
  :tar => '/code/ssa.tar.gz'
  :simulations = 100_000,
  :output = '/mydata/run-#{i}'

 # wait for ssa job to finish
 loop {
  ssa_data = neptune :type => 'get-output',
   :output => '/mydata/run-#{i}'
  if ssa_data[:success]
   break
  end
  sleep(60)
```

```
 }


 confidence_achieved = ssa_data[:output]
 if confidence_achieved>confidence_needed
   break
  else
   puts 'Sufficient confidence not reached.'
 end
 i += 1
}
```

To enable StochKit support within Neptune, we automatically install StochKit within newly created AppScale neptune/images by fetching it from a local repository. It is placed in a predetermined location on the image and made available to user-specified scripts via its standard executables. It is possible to require users to run a Neptune `:compile` job that would install StochKit in an on-demand fashion, but we elect to preinstall it, to reduce the number of steps required to run a StochKit job. Additionally, while forcing a compilation step is possible, the user's StochKit code often consists of biochemical models and a `bash` script, which do not need to be compiled to execute and thus do not fall under the domain of a `:compile` job.

As StochKit does not run in a distributed fashion, the AppController coordinates the machines that the user requests to run their SSA computation. For the example above, in which four nodes are to be used to run 100,000 simulations, Neptune instructs each node to run 25,000 simulations.

DFSP

One specialized SSA implementation supported by Neptune is the Diffusive Finite State Projection algorithm (DFSP) [36], a high-performance method for simulating spatially inhomogenous stochastic biochemical systems, such as those found inside living cells. The example system that we examine here is a biological model of yeast polarization, known as the G-protein cycle example, shown in [36]. Yeast cells break their spatial symmetry and polarize in response to an extra-cellular gradient of mating pheromones. The dynamics of the system are modeled using the stochastic reaction-diffusion master equation. Figure 4.3 shows visualizations from stochastic simulations of this model.

The code for the DFSP implementation is a tarball containing C language source and an accompanying `makefile`. The executable produces a single trajectory for each instance that is run. As this simulation is a stochastic system, an ensemble of independent trajectories are required for statistical analysis; 10,000 trajectories are needed to minimize error to acceptable levels. The Neptune code needed to run this is:

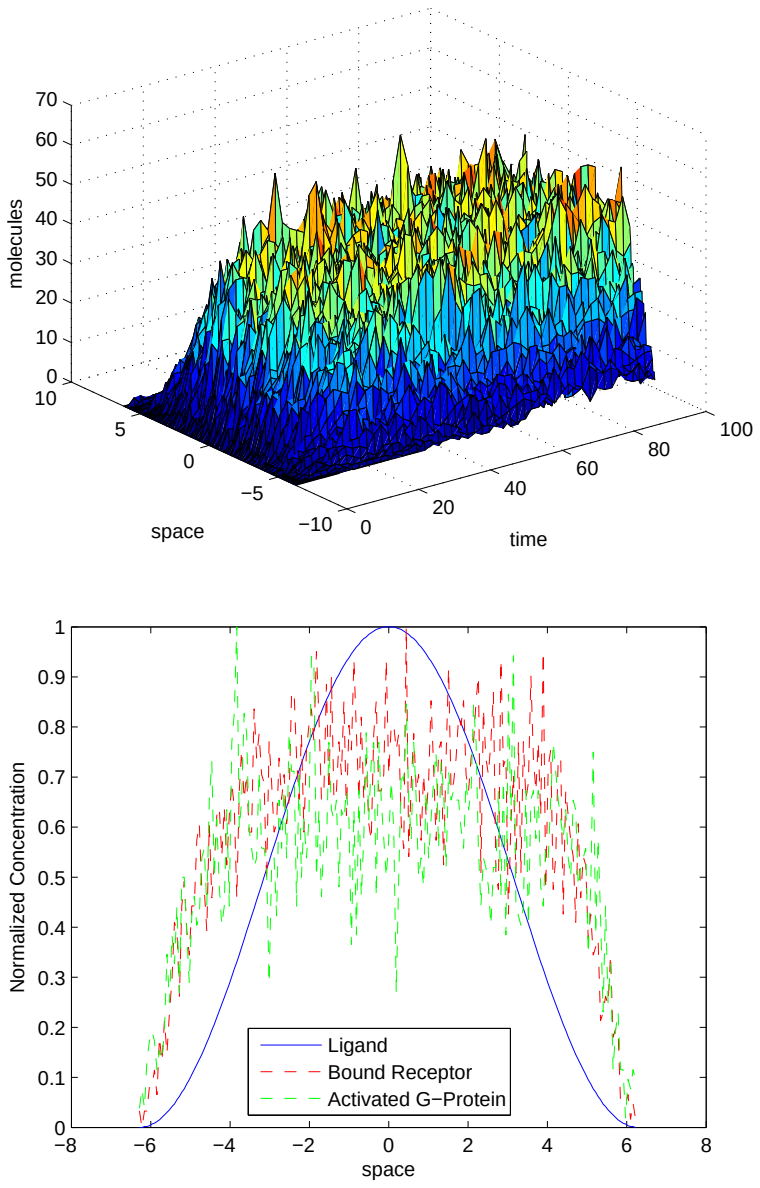

Figure 4.3: Two plots of the DFSP example model of yeast polarization. (Left) Temporal-Spatial profile of activated G-protein. Stochastic simulation reproduces the noise in the protein population that is inherent to this system. (Right) Overlay of three biochemical species populations across the yeast cell membrane: the extra-cellular pheromone ligand, the ligand bound with membrane receptor, and the G-protein activated by a bound receptor.

```
status = neptune :type => :ssa,
 :nodes_to_use => 64,
 :tar => '/code/dfsp.tar.gz',
 :output => '/outputs/ssa-output',
 :storage => 's3',
 :EC2_ACCESS_KEY => ENV['S3_ACCESS_KEY'],
 :EC2_SECRET_KEY => ENV['S3_SECRET_KEY'],
 :S3_URL => ENV['S3_URL'],
 :trajectories => 10_000,
puts status.inspect
```

In this example, the scientist has indicated that they wish to run their DFSP code, stored remotely at `/code/dfsp.tar.gz`, over 64 machines. The scientist here has also specified that their code should be retrieved from Amazon S3 with the provided credentials, and that the output should be saved back to Amazon S3. Finally, the scientist has indicated that 10,000 simulations should be run. The storage-specific parameters used here are not specific to DFSP or SSA jobs, and can be used with any type of computation.

To enable DFSP support within Neptune, we automatically install support for the GNU Scientific Library (GSL) when we generate a new AppScale image. The user's

DFSP code can then utilize it within its computations in the same fashion as if it were installed on their local computer. Neptune does not currently provide a general model for determining library dependencies, as versioning of libraries can make this problem difficult to handle in an automated fashion. However, Neptune does allow an expert user to manually install the required libraries a single time and enable the community at large to benefit.

dwSSA

Another specialized SSA implementation we support within Neptune is the dwSSA, the doubly weighed SSA coupled with the cross-entropy method. The dwSSA is a method for accurate estimation of rare event probabilities in stochastic biochemical systems. Rare events, events with probabilities no larger than 10^{-9}, often have significant consequences to biological systems, yet estimating them tends to be computationally infeasible. The dwSSA accelerates the estimation of these rare events by significantly reducing the number of trajectories required. This is accomplished using importance sampling, which effectively biases the system toward the desired rare event, and reduces the number of trajectories simulated by several orders of magnitude.

The system we examine in this work is the birth-death process shown in [31]. The rare event that this model attempts to determine is the probability that the stochastic fluctuations of this system will double the population of the chemical species. The

model requires the simulation of 1,000,000 trajectories to accurately characterize the rare event probability. The code for this example is a coupled set of source files written in R (for model definition and rare event calculations) and C (for efficient generation of stochastic trajectories). The Neptune code needed to run the dwSSA implementation is identical to that of DFSP and StochKit: users simply supply their own tarball with the dwSSA code in place of a different SSA implementation. As each dwSSA simulation takes a trivial amount of time to run, we customize it to take, as an input, the number of simulations to run. This minimizes the amount of time wasted setting up and tearing down the R environment.

To enable dwSSA support within Neptune, we automatically install support for the R programming language when we generate a new AppScale image. We also place the R executables in a predetermined location for use by AppScale and Neptune and use R's batch facilities to instruct R to never save the user's workspace (environment) between R executions, as is the default behavior.

4.3.4 Employing Neptune for Cloud Scaling and Enabling Hybrid Clouds

Our goal with Neptune is to simplify configuration and deployment of HPC applications. However, Neptune is flexible enough to be used with other application domains. Specifically, Neptune can be used to control the scaling and placement of

services within the underlying cloud platform. Furthermore, if the platform supports hybrid cloud placement strategies, Neptune can control how services are placed. This allows Neptune to be used for both high throughput computing (HTC) and many task computing (MTC). In the former case, resources can be claimed from multiple cloud infrastructures to serve user jobs. In the latter case, Neptune can be used to serve both compute-intensive jobs as well as web service programs.

To demonstrate this, we use Neptune to enable users to manually scale up a running AppScale deployment. Users need only specify which component they wish to scale up (e.g., the load balancer, application server, or database server) and how many of them they require. This reduces the typically difficult problem of scaling up a cloud to the following Neptune code:

```
neptune :type => :appscale,
 :nodes_to_use => {:cloud1 => 3,
                   :cloud2 => 6},
 :add_component => 'appengine',
 :time_needed_for => 3600
```

In this example, the user has specified that they wish to add nine application servers to their AppScale deployment, and that these machines are needed for one hour. Furthermore, three of the servers should be placed in the first cloud that the platform is running over, while six servers should be placed in the second cloud. Defining which

cloud is the "first cloud" and which cloud is the "second cloud" is done by the cloud administrator, via the AppScale Tools (see Section 4.1.1). This type of scaling is useful when the amount of load in both clouds is known: here, this is useful if both clouds are over-provisioned, but the second is either expecting greater traffic in the near future or is sustaining more load than the first cloud.

Scaling and automation are only amenable to the same degree as the underlying services allow for. For example, while the Cassandra database allows nodes to be added to the system dynamically, users cannot add more nodes to the system than already exist (e.g., in a system with N nodes, no more than $N-1$ nodes can be added at a time) [24]. Therefore, if more than the allowed for number of nodes are needed, either multiple Neptune jobs must be submitted or the cloud platform must absorb this complexity into its scaling mechanisms.

4.3.5 Limitations

Neptune enables automatic configuration and deployment of software by a cloud platform to the extent that the underlying software allows. It is thus important to make explicit scenarios in which Neptune encounters difficulties, as they are the same scenarios in which the supported software packages are not amenable to being placed in a cloud platform. From the end-users we have designed Neptune to aid, we have experi-

enced three common problems that are not specific to Neptune or to distributed systems (e.g., clouds, grids) in general:

- Codes that require a unique identifier, whether it be an IP address or process name to be used to locate each machine in the computation (e.g., multi-node Erlang computations). This is distinct from the case where the framework requires IP addresses to be hardcoded, as these frameworks (like MPI) do not require the end-user's code to be modified in any way or be aware of a node's IP address.

- Programs that have highly specialized libraries for end-users but are not free / open-source, and thus are currently difficult to dynamically acquire and release licenses for.

- Algorithms that require a high-speed interconnect that run in a cloud infrastructure that does not offer one. These algorithms may suffer from degraded performance or may not work correctly at all. The impact of this can be mitigated by choosing a cloud infrastructure that does provide such an offering (e.g., Cluster Compute Instances for Amazon EC2, or a Eucalyptus cloud with similar network hardware).

We are investigating how to mitigate these limitations as part of our future work. For unique identifiers, it is possible to have Neptune take a parameter containing a list of process identifiers to use within computation. For licensing issues, we can have

the cloud fabric make licenses available on a per-use basis. AppScale can then guide developers to clouds that have the appropriate licenses for their application.

4.3.6 Extensibility

Neptune is designed to be extensible, both in the types of job supported and the infrastructures that it can harness. Developers who wish to add support for a given software framework within Neptune need to modify the Neptune language component as well as the Neptune runtime within the cloud platform that receives Neptune job requests. In the Neptune language component, the developer needs to indicate which parameters users need to specify in their Neptune code (e.g., how input and output should be handled), and if any framework-specific parameters should be exposed to the user. At the cloud platform layer, the developer needs to add functionality that can understand the particulars of their Neptune job. This often translates into performing special requests based on the parameters present (or absent) in a Neptune job request. For example, MapReduce users can specify that the input be copied from the local file system to the Hadoop Distributed File System. Our implementation within AppScale skips this step if the user indicates that the input is already present within HDFS. Once a single, expert developer has added support for a job type within Neptune and AppScale, it can then be automatically configured and deployed by the community at large, without requiring them to become an expert user.

4.4 Evaluation

We next use Neptune to empirically evaluate how effectively the supported services execute within AppScale. We begin by presenting our experimental methodology and then discuss our results.

4.4.1 Methodology

To evaluate the software packages supported by Neptune, we use benchmarks and sample applications provided by each. We also measure the cost of running Neptune jobs with and without VM reuse.

To evaluate our support for MPI, we use a Power Method implementation that, at its core, multiplies a matrix by a vector (the standard `MatVec` operation) to find the absolute value of the largest eigenvalue of the matrix. We choose this code over more standard codes such as the Intel MPI Benchmarks because it tests a number of the MPI primitives working in tandem, producing a code that should scale with respect to the number of nodes in the system. By contrast, the Intel MPI Benchmarks largely measure interprocess communication time or the time taken for a single primitive operation, which is likely to scale negatively as the number of nodes increase (e.g., barrier operations are likely to take longer when more nodes participate). We use a 6400x6400

matrix and 6400x1 vector to ensure that the size of the matrices evenly divides the number of nodes in the computation.

For X10, our evaluation uses an NQueens implementation publicly available from the X10 team that is optimized for multiple machines. To ensure a sufficient amount of computation is available, we set $n = 16$, thus creating a 16x16 chessboard and placing 16 queens on the board. For comparison purposes with MPI, we also include an optimized MPI version publicly made available be the authors of [80]. It is also set to use a 16x16 chessboard, using a single node to distribute work across machines and the others to perform the actual work involved.

To evaluate our support for MapReduce, we use the publicly available Java WordCount benchmark, which takes an input data set and finds the number of occurrences of each word in that set. Each Map task is assigned a series of lines from the input text, and for every word it finds, it reports this with an associated count of one. Each Reduce task then sums the counts for each word and saves the result to the output file. Our input file consists of the works of William Shakespeare appended to itself 500 times, producing an input file roughly 2.5GB in size.

We evaluate UPC and Erlang by means of a Thread Ring benchmark, and compare them to reference implementations in MPI and X10. Each code implements the same functionality: a fixed number of processes are spawned over a given number of nodes, and each thread is assigned a unique identifier. The first thread passes a message to the

next thread, who then continues doing so until the last thread receives the message. The final thread sends the message to the first thread, connecting the threads in a ring-like fashion. This is repeated a given number of times to complete the program's execution.

In our first Thread Ring experiment, we fix the number of messages to be sent to 100 and fix the number of threads to spawn to 64. We vary the number of nodes to use between 1, 2, 4, 8, 16, 32, and 64 nodes, to determine the performance improvement that can be achieved by increasing the amount of available computation power.

In our second Thread Ring experiment, we fix the number of messages to be sent to 100, and fix the number of nodes to use at 8 nodes. We then vary the number of threads to spawn between 2, 4, 8, 16, 32, and 64 threads, to determine the impact of increasing the number of threads that must be scheduled on a fixed number of machines.

Our third Thread Ring experiment fixes the number of nodes to use at 8 nodes, and fixes the number of threads to use at 64 threads. We then vary the number of messages to send between 1, 10, 100, 1000, and 10000, to determine the performance costs of increasing the number of messages that must be sent around the distributed thread ring in each implementation.

For our SSA codes, DFSP and dwSSA, we run 10,000 and 1,000,000 simulations, respectively, and measure the total execution time. As mentioned earlier, previous work in each of these papers indicate that these numbers of simulations are the minimum that scientists typically must run to achieve a reasonable accuracy.

We execute the non-Thread Ring experiments over different dynamic AppScale cloud deployments of 1, 4, 8, 16, 32, and 64 nodes. In all cases, each node is a Xen guestVM that executes with 1 virtual processor, 10GB of disk (maximum), and 1GB of memory. The physical machines that we deploy VMs to execute with 8 processors, 1TB of disk, and 16GB of memory. We employ a placement strategy provided by AppScale where one node deploys an AppLoadBalancer (ALB) and Database Peer (DBP), and the other nodes are designated as "open" (that is, they can be claimed for any role by the AppController as needed). Since no Google App Engine applications are deployed, no AppServers run in the system. All values reported here represent the average of five runs.

For these experiments, Neptune employs AppScale 1.5, MPICH2 1.2.1p1, X10 2.1.0, Hadoop MapReduce 0.20.0, UPC 2.12.1, Erlang R13B01, the DFSP implementation graciously made available by the authors of the DFSP paper [36], the dwSSA implementation graciously made available by the authors of the dwSSA paper [31], and the StochKit implementation publicly made available on the project's web site [89].

4.4.2 Experimental Results

We begin by discussing the performance of the MPI and X10 Power Method codes within Neptune. We time only the computation and any necessary communication required for the computation; thus, we exclude the time to start NFS, to write MPI con-

figuration files, and to start prerequisite MPI services. Figure 4.4 presents these results.

Table 4.1 presents the parallel efficiency, given by the standard formula:

$$E = \frac{T_1}{pT_p} \tag{4.1}$$

where E denotes the parallel efficiency, T_1 denotes the running time of the algorithm running on a single node, p denotes the number of processors used in the computation, and T_p denotes the running time of the algorithm running on p processors.

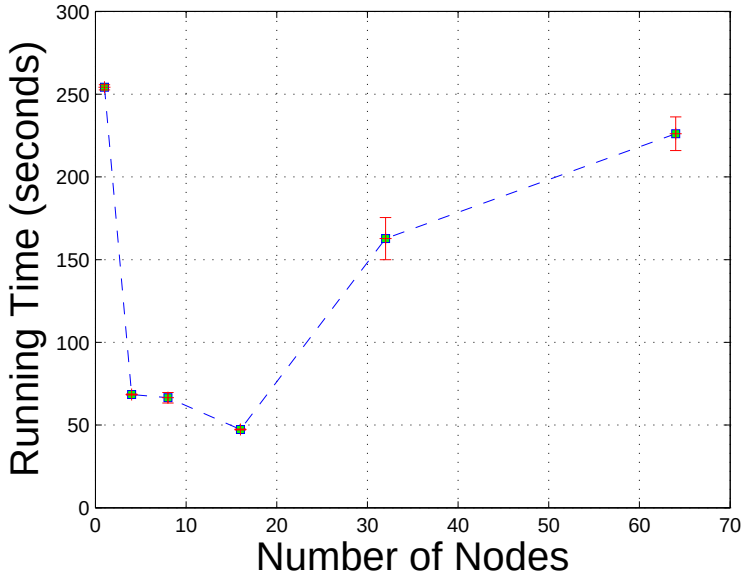

Figure 4.4: Average running time for the Power Method code utilizing MPI over varying numbers of nodes. These timings include running time as reported by MPI_Wtime and do not include NFS and MPI startup and shutdown times.

Both Figure 4.4 and Table 4.1 show clear trends: speedups are initially achieved as nodes are increased to the system, but the decreasing parallel efficiencies show that this

Table 4.1: Parallel efficiency for the Power Method code utilizing MPI over varying numbers of nodes.

# of Nodes	MPI Parallel Efficiency
4	0.9285
8	0.4776
16	0.3358
32	0.0488
64	0.0176

scalability does not extend up through 64 nodes. Furthermore, the running time of the Power Method code increases after using 16 nodes. Analysis using VAMPIR [93], a standard tool for MPI program visualization, shows that the collective broadcast calls used are the bottleneck, becoming increasingly so as the number of nodes increase in the system. This is an important point to reiterate: since Neptune simply runs supported codes on varying numbers of nodes, the original code's bottlenecks remain present and are not optimized away.

The MPI and X10 n-queens codes encounter a different type of scaling compared to our Power Method code. Figure 4.5 shows these trends: the MPI code's performance is optimal at 4 nodes, while the X10's code performance is optimal at 16 nodes. The X10 n-queens code suffers substantially at the lower numbers of nodes compared to its MPI counterpart; this is likely due to its relatively new work-stealing algorithm, and is believed to be improved in subsequent versions of X10. This is also the rationale for the larger standard deviation encountered in the X10 test. We omit parallel efficiencies for this code because the MPI code dedicates the first node to coordinate the computation,

which precludes us from computing the time needed to run this code on a single node (a required value).

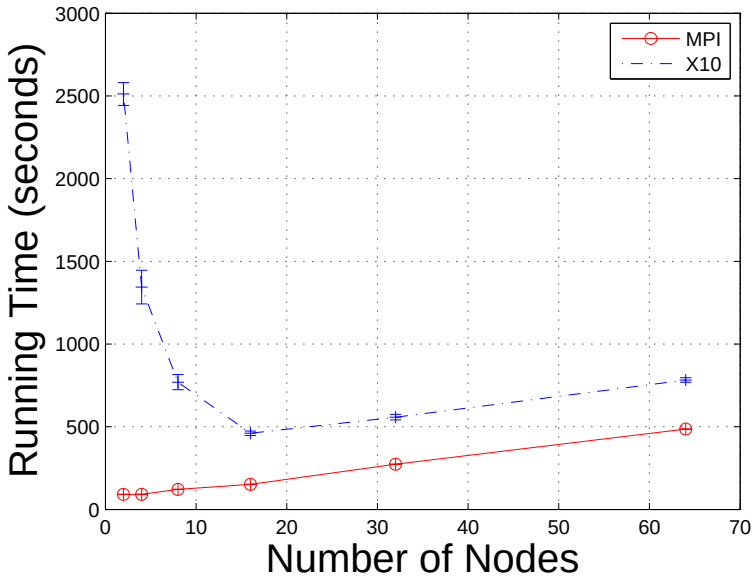

Figure 4.5: Average running time for the n-queens code utilizing MPI and X10 over varying numbers of nodes. These timings include running time as reported by MPI_Wtime and $System.nanoTime$, respectively. These times do not include NFS and MPI startup and shutdown times.

MapReduce WordCount experiences a superior scale-up compared to our MPI and X10 codes. This is largely because this MapReduce code is optimized by Hadoop and does not communicate between nodes, except between the Map and Reduce phases. Figure 4.6 and Table 4.2 show the running times of WordCount via Neptune. As with MPI, we measure computation time and not the time incurred starting and stopping Hadoop.

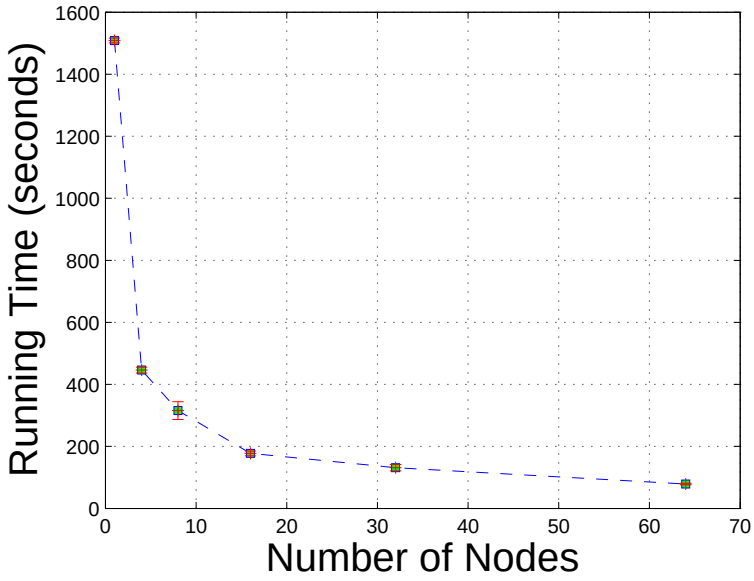

Figure 4.6: Average running time for WordCount utilizing MapReduce over varying numbers of nodes. These timings include Hadoop MapReduce run times and do not include Hadoop startup or shutdown times.

Figure 4.6 and Table 4.2 show opposing trends compared to the MPI results. With our MapReduce code, we see consistent speedups as more nodes are added to the system, although with a diminishing impact as we add more nodes to the system. This is clear from the decreasing parallel efficiencies, and as stated before, these speedups are not related to MapReduce or MPI specifically, but are due to the programs evaluated here. WordCount sees a superior speedup compared to the Power Method code due to the reduced amount of communication and larger amounts of computation. We also see smaller standard deviations when compared with the Power Method MPI code, as the communication is strictly dictated and optimized by the runtime itself.

Table 4.2: Parallel efficiency for WordCount using MapReduce over varying numbers of nodes.

# of Nodes	Parallel Efficiency
4	0.8455
8	0.5978
16	0.5313
32	0.3591
64	0.3000

Table 4.3: Parallel efficiency for the Thread Ring code utilizing MPI, X10, and UPC over varying numbers of nodes.

# of Nodes	MPI	X10	UPC
1	1.0000	1.0000	1.0000
2	0.5000	0.5000	0.5000
4	0.4518	1.1251	0.0014
8	0.3068	1.2663	5.6904e-04
16	0.1955	1.6518	2.0528e-04
32	0.1189	1.9190	7.0025e-05
64	0.0642	25.1618	9.8221e-06

In our first Thread Ring experiment, we measure time taken to send 100 messages through a ring of 64 threads. We vary the number of nodes used between 1, 2, 4, 8, 16, 32, and 64. Figure 4.7 shows the amount of time taken for implementations written in X10, MPI, and UPC, while Table 4.3 shows the parallel speedup achieved. Both the MPI and X10 codes improve in execution time as nodes are added. While the X10 code achieves a better parallel efficiency than the MPI code, it is on average one to three orders of magnitude slower. The reason behind this has been explained by the X10 team: the X10 runtime currently is not optimized to handle scenarios where the system is overprovisioned (e.g., when the number of processes exceeds the number of nodes).

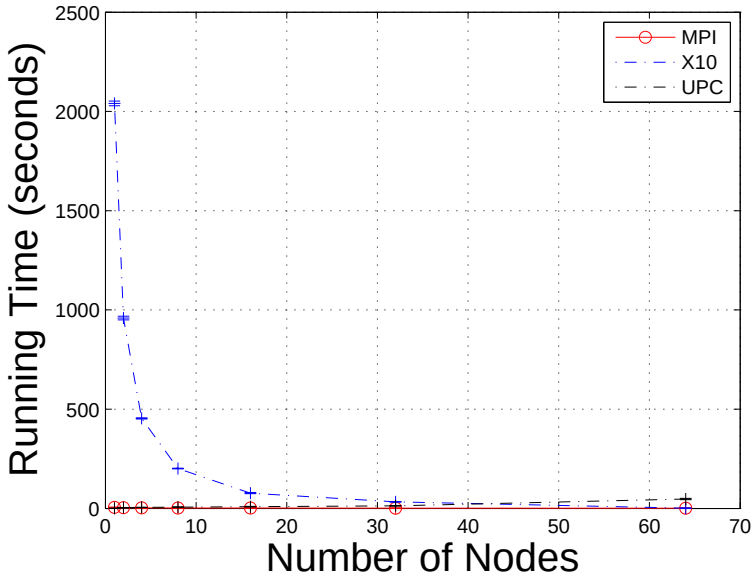

Figure 4.7: Average running time for the Thread Ring code utilizing MPI, X10, and UPC over varying numbers of nodes. These timings only include execution times reported by each language's timing constructs.

This is confirmed by the scenario in which 64 nodes are used: here, the system is not overprovisioned and runs in an equivalent amount of time as the MPI code. The UPC code exhibits a very different scaling pattern compared to the MPI and X10 codes: as it relies on synchronization via barrier statements, it runs quickly when the number of nodes is small, and becomes slower as the number of nodes increases.

Our second Thread Ring experiment fixes the number of nodes at the median value (8 nodes), and measures the amount of time needed to send 100 messages through thread rings of varying sizes. Here, we vary the sizes between 8, 16, 32, 64, and 128 threads. The results of this experiment for the X10, MPI, and UPC codes are shown in

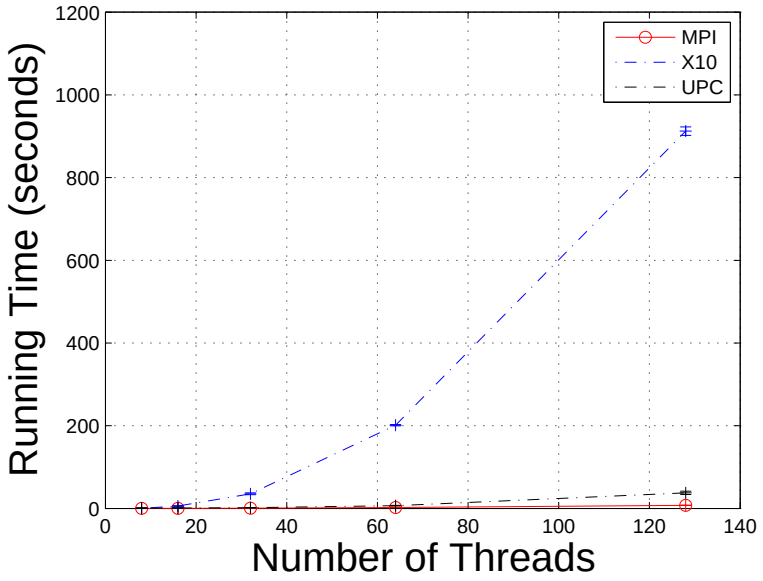

Figure 4.8: Average running time for the Thread Ring code utilizing MPI, X10, and UPC over varying numbers of threads. These timings only include execution times as reported by each language's timing constructs.

Figure 4.8. As expected, all codes become slower as the size of the thread ring grows. The overall execution time is fastest for the MPI code, followed by that of the UPC and X10 codes. The reason for these differences is identical to that given previously: the UPC code relies on barriers. As the number of threads increases, it becomes more expensive to perform these barrier operations. The X10 code is also overprovisioned in most cases, so it slows down in these scenarios as well. In the scenario when it is not overprovisioned (e.g., when there are 8 threads and 8 nodes), the X10 code performs on par with the MPI code.

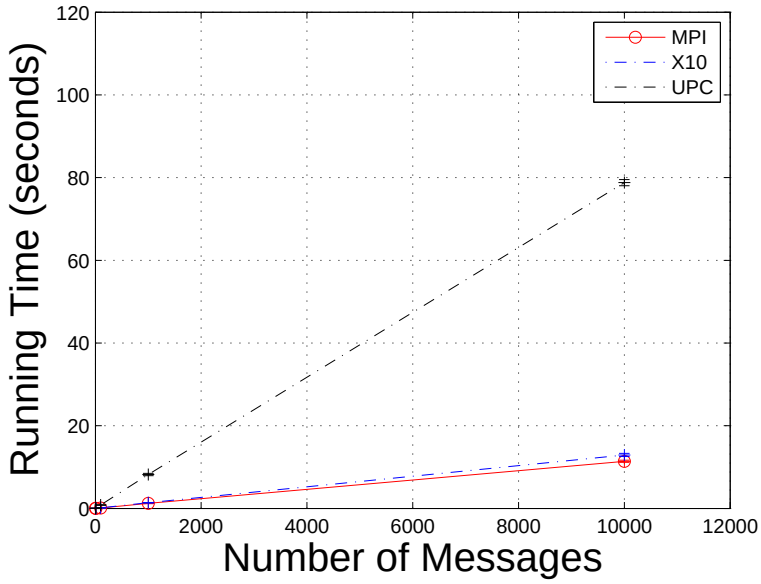

Figure 4.9: Average running time for the Thread Ring code utilizing MPI, X10, and UPC over varying numbers of messages. These timings only include execution times as reported by each language's timing constructs.

Our third Thread Ring experiment fixes the number of nodes at the median value (8 nodes) once again and measures the amount of time needed to send a varying number of messages through the thread ring. Specifically, we vary the number of messages to be sent between 1, 10, 100, 1000, and 10000 messages for the X10, MPI, and UPC codes. Figure 4.9 shows the results of this experiment: for all codes, excluding the single message scenario, the time to send additional messages increases linearly. Unlike the other benchmarks, the X10 and UPC codes perform within an order of magnitude of the MPI code. For the X10 code, this is because all machines are well-provisioned (specifically because we run 8 threads over 8 nodes), avoiding the performance degradation that the

other experiments revealed. The UPC code also maintains relatively close performance to the MPI and X10 codes due to the low number of nodes: the barriers, which are the bottleneck of the UPC code, are inexpensive when a relatively small number of nodes are used.

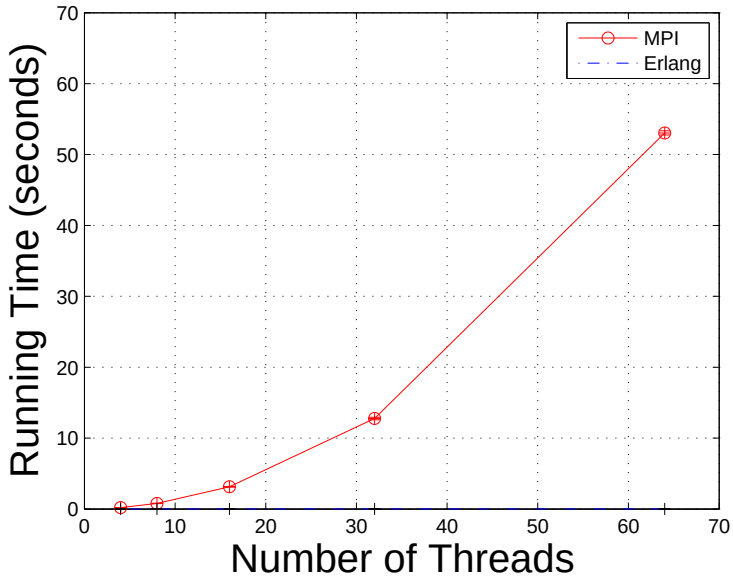

Figure 4.10: Average running time for the single node Thread Ring code utilizing MPI and Erlang over varying numbers of threads. These timings only include execution times as reported by each language's timing constructs.

To evaluate the performance of our Erlang code, we compare our Erlang Thread Ring implementation with that of our MPI code deployed over a single node. We fix the number of messages to send at 1000 and vary the number of threads that make up the ring between 4, 8, 16, 32, and 64. The results of this experiment are shown in Figure 4.10. The Erlang code scales linearly, and performs two to three orders of

magnitude faster than the MPI code for all numbers of threads tested. This is likely due to Erlang's long history as a concurrent language and lightweight threading model, which makes it highly amenable to this experiment. Similarly, MPI is designed to be a distributed programming language, and as we have seen in the other experiments, suffers performance degradations when overprovisioned. We are looking into support for Threaded MPI (TMPI) [85], which provides optimizations for the concurrent use case shown here.

We next analyze the performance of the specialized SSA packages supported by Neptune. This includes the specialized implementations present in DFSP and dwSSA, which here focus on the yeast polarization and birth-death models discussed previously.

Like the MapReduce code analyzed earlier, DFSP also benefits from parallelization and support via Neptune. This is because the DFSP implementation used has no internode communication during its computation, and is embarrassingly parallel. In the DFSP code, once each node knows how many simulations to run, they work with no communication from other nodes. Figure 6.2 and Table 6.1 show the running times for 10,000 simulations via Neptune. Unlike MapReduce and MPI, which provide distributed runtimes, our DFSP code does not, so we time all interactions once AppScale receives the message to begin computation from Neptune until the results have been merged on the master node.

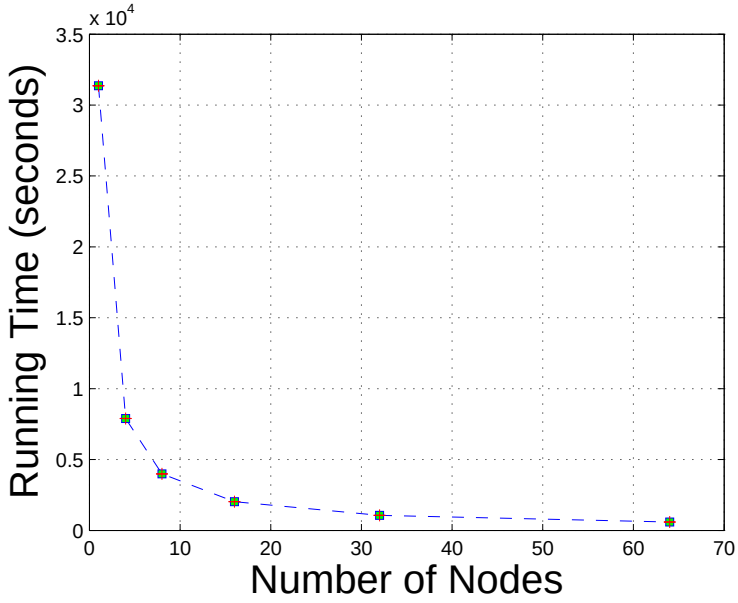

Figure 4.11: Average running time for the DFSP code over varying numbers of nodes. As the code used here does not have a distributed runtime, timings here include the time that AppScale takes to distribute work to each node and merge the individual results.

Figure 6.2 and Table 6.1 show similar trends for the DFSP code as seen in MapReduce WordCount. This code also sees a consistent reduction in runtime as the number of nodes increase, but retains a much higher parallel efficiency compared to the MapReduce code. This is due to the lack of communication within computation, as the framework needs only to collect results once the computation is complete, and does not need to sort or shuffle data, as is needed in the MapReduce framework. As less communication is used here compared to WordCount and Power Method MPI codes, the DFSP code exhibits a smaller standard deviation, and a standard deviation that tends to decrease with respect to the number of nodes in the system.

Table 4.4: Parallel efficiency for the DFSP code over varying numbers of nodes.

# of Nodes	Parallel Efficiency
4	0.9929
8	0.9834
16	0.9650
32	0.9216
64	0.8325

Table 4.5: Parallel efficiency for the dwSSA code over varying numbers of nodes.

# of Nodes	Parallel Efficiency
4	0.7906
8	0.4739
16	0.3946
32	0.2951
64	0.1468

Another example that follows similar trends to the DFSP code is the other Stochastic State Algorithm, dwSSA, shown in Figure 4.12 and Table 4.5. This code achieves a reduction in runtime with respect to the number of nodes in the system, but does not do so at the same rate as the DFSP code, as can be seen through the lower parallel efficiencies. This is because the execution time for a single dwSSA trajectory is much smaller than a single DFSP trajectory, which results in wasted time setting up and tearing down the R environment.

4.4.3 VM Reuse Analysis

Next, we perform a brief examination of the costs of the experiments in the previous section if run over Amazon EC2, with and without the VM reuse techniques described

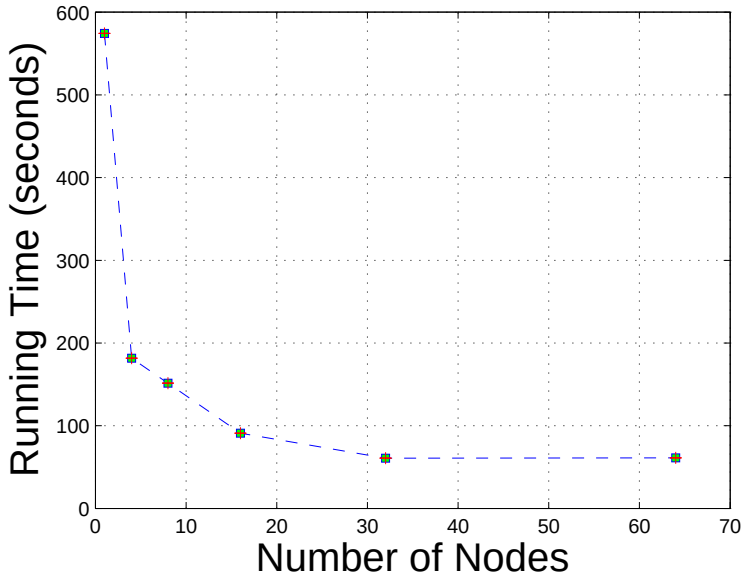

Figure 4.12: Average running time for the dwSSA code over varying numbers of nodes. As the code used here does not have a distributed runtime, timings here include the time that AppScale takes to distribute work to each node and merge the individual results.

previously. The VMs are configured with 1 virtual CPU, 1 GB of memory, and a 64-bit platform. This is similar to the Amazon EC2 "Small" machine type (1 virtual CPU, 1.7 GB of memory, and a 32-bit platform) which costs $0.085 per hour.

Each PowerMethod, MapReduce, DFSP, and dwSSA experiment is run five times at 1, 4, 8, 16, 32, and 64 nodes to produce the data shown earlier, while each NQueens experiment is run five times at 2, 4, 8, 16, 32, and 64 nodes. We compute the cost of running these experiments without VM reuse (that is, by acquiring the needed number of machines, running the experiments, and then powering them off) compared to the cost with VM reuse (that is, by acquiring the needed number of machines, performing

Table 4.6: Cost to run experiments for each type of Neptune job, with and without reusing virtual machines.

# Type of Job	Cost with VM Reuse	Cost without VM Reuse
PowerMethod	$12.84	$64.18
NQueens(MPI)	$12.92	$64.60
NQueens(X10)	$13.01	$64.60
MapReduce	$13.01	$64.18
DFSP	$35.70	$78.63
dwSSA	$12.84	$64.18
Total	$100.32	$400.37

the experiment for all numbers of nodes, and not powering them off until all runs complete). Note that in the reuse case, we do not perform reuse between experiments. For example, the Neptune code used to run the experiments for the X10 NQueens code is:

```
[2,4,8,16,32,64].each { |i|
 5.times { |j|
  neptune :type => :x10,
   :code => '/code/NQueensDist',
   :nodes_to_use => i,
   :output => '/nqueensx10/#{i}/#{j}'
 }
}
```

Table 4.6 shows the expected cost of running these experiments with and without VM reuse. In all experiments, employing VM reuse greatly reduces the cost. This is

largely due to inefficient use of nodes without reuse, as many scenarios employ large numbers of nodes to run experiments that only run for a fraction of an hour (VMs are charged for by AWS by the hour). All of the experiments except for DFSP also cost roughly the same because they use similar numbers of CPU-hours of computation within AWS and thus are similarly priced. We see much greater variation in time and cost on a per-minute or per-second pricing model instead of a per-hour pricing model.

4.5 Related Work

An early version of this work was presented at the Workshop on Scientific Cloud Computing (ScienceCloud) and was entitled "Neptune: A Domain Specific Language for Deploying HPC Software on Cloud Platforms" [17]. The work developed by others that is most similar to Neptune is `cloudinit.d` from Nimbus [61]. `cloudinit.d` provides an API that users employ to automatically launch, configure, and deploy nodes in a cloud infrastructure. In contrast to Neptune, `cloudinit.d`'s programming model places the onus of configuration and deployment on the user who writes `cloudinit.d` scripts. Neptune takes an alternate approach, hiding the complexity behind correct configuration and deployment.

Other works exist that provide either language support for cloud infrastructures or automated configuration or deployment, but not both. In the former category ex-

ist projects like SAGA [60], the RightScale Gems [79] and `boto` [13]. SAGA enables users to write programs in C++, Python, or Java that interact with grid resources, with the recent addition of support for cloud infrastructure interaction. A key difference between SAGA and Neptune is that SAGA is conceptually designed to work with grid resources, and thus the locus of control remains with the user. The programming paradigm embodied here serves use cases that favor a static number of nodes and an unchanging environment. Conversely, Neptune is designed to work over cloud resources, and can elastically add or remove resources based on the environment. The RightScale Gems and `boto` are similar to SAGA but only provide interaction with cloud infrastructures (e.g., Amazon EC2 and Eucalyptus).

In the latter category exist projects such as the Nimbus Context Broker [61] and Mesos [56]. The Nimbus Context Broker automates configuration and deployment of otherwise complex software packages in a matter similar to that of Neptune. It acquires a set of virtual machines from a supported cloud infrastructure and runs a given series of commands to unify them as the user's software requires. Conceptually, this is similar to what Neptune offers. However, it does not offer a language by which it can be operated, like Neptune and SAGA. Furthermore, the Nimbus Cloud Broker, like SAGA, does not make decisions dynamically based on the underlying environment. A set of machines could not be acquired, tested to ensure a low latency exists, and released within a script running on Nimbus Cloud Broker. Furthermore, it does not employ

virtual machine reuse techniques such as those seen within Neptune. This would require a closer coupling with supported cloud infrastructures or the use of a middleware layer to coordinate VM scheduling, which would effectively present a cloud platform.

Like the Nimbus Context Broker, Mesos also automates configuration and deployment of complex software packages, but aims to do so for only a very specific set of packages (MapReduce, MPI, Torque, and Spark). It requires supported packages to be modified, and once they are "Mesos-aware", they can be utilized towards goals of better resource utilization for the cluster as a whole and better performance for individual jobs. Mesos also positions itself in the cluster computing space, in which jobs can dynamically scale up and down in the number of nodes that they use, but where the cluster as a whole must be statically partitioned. Cluster administrators can manually add or remove nodes, but the size of the cluster as a whole tends to remain static. This is in contrast to the cloud model employed by Neptune, where the number of nodes is in flux and is controllable by Neptune itself.

4.6 Summary and Conclusions

Neptune provides users with a domain specific language that abstracts away the complexities of deploying and utilizing high performance computing services within cloud platforms. The motivation behind Neptune is to enable users to program with

HPC frameworks without first having to learn how to install, configure, deploy, and maintain the often complex runtimes associated with these frameworks. Neptune aims to achieve this goal by repurposing the AppScale cloud platform to automatically configure and deploy HPC frameworks, on behalf of the user, who need only indicate how many nodes are required to execute their application. We evaluate Neptune by utilizing sample applications from the MPI, X10, MapReduce, UPC, and Erlang general-purpose HPC frameworks, as well as StochKit, DFSP, and dwSSA, scientific applications that serve the computational systems biology community. Neptune enables users to utilize these HPC frameworks over varying numbers of nodes with minimal effort, simply, uniformly, and scalably.

The text of this chapter is, in part, a reprint of the material as it appears in [19].

Chapter 5

MEDEA: A Pluggable Middleware System for Interoperable Program Execution Across Cloud Fabrics

In this chapter, we present MEDEA, an execution model for automatically executing programs, written in various programming languages, portably over disparate cloud fabrics. MEDEA abstract away the details of cloud-based program execution by providing language support (to enable applications to be programmatically described) as well as runtime support that abstracts away implementation-specific details of cloud-based queuing, compute, and storage services. To facilitate cross-cloud interoperability, MEDEA plugs a wide range of compute, storage, and FIFO queue offerings into this system, including those provided by Amazon Web Services, Microsoft Azure, Google App Engine, and AppScale. By doing so, MEDEA relieves developers of the burden of having to become experts with each cloud system on which they wish to run. To investigate the potential of MEDEA, we employ the system for a number of different

use cases in which we compare and contrast supported clouds in terms of price and performance using various applications, domains, and programming languages.

5.1 Introduction and Motivation

Both IaaS and PaaS systems export programmatic access to a wide range of scalable services via well-defined APIs. Increasingly, these exported technologies are similar across cloud offerings and include storage, FIFO queues, and execution services. Although such technologies simplify distributed application deployment, their APIs, language bindings, performance, scale, and cost models all differ significantly across clouds.

The plethora of IaaS and PaaS options, implementations, and restrictions makes it challenging and time consuming for new and expert users alike to determine which set of services is best for a particular application, for some definition of "best" (e.g., price, performance, scale, ease of use). Moreover, once users choose a service and code their application to that interface and configure it for that system, they become "locked in"

As a motivating example, consider a typical user who wishes to start utilizing cloud services. They must first evaluate which cloud services they wish to use in their application, which can be at varying layers of abstraction and require differing amounts of maintainence. Once the user decides which services they want to utilize, they must

then implement and maintain their system utilizing each of the chosen technologies. If transitioning off of an existing system, then the user must port their system to the new technology. The time and engineering costs that the user has invested in learning each of these technologies is not directly transferrable to other technologies: while other competitors may be abstractly similar (i.e., Amazon EC2 and Google Compute Engine both offer virtual machines), in practice their APIs are incompatible, requiring an expert user to refactor the code base when porting to other services. Finally, the user must spend additional time to transfer the knowledge of how to maintain their application (which now utilizes a new set of services) with others.

This work attempts to alleviate these problems by proposing an **interoperable, portable, reusable execution model for cloud systems**. Using this model, which we call MEDEA, users describe the execution environment of their programs in a high level scripting language. This metadata includes details about the program (name, executable, arguments, etc.) and the user's account credentials for each cloud they wish to use. This scripting language support then communicates with the MEDEA deployment engine. This deployment engine is a software overlay that exports an abstract interface for managing jobs in FIFO queues, for executing jobs, and for persisting program output and profile information in cloud storage so that it can be easily accessed by users, post-execution.

The deployment engine plugs in different cloud services to provide the implementation of these operations. By doing so, MEDEA is able to deploy arbitrary programs over different cloud systems without requiring modification to the programs themselves, providing interoperability between supported cloud services. That is, we offload the complexities of using different compute, storage, and queue cloud services onto MEDEA. As a result, developers can use MEDEA-compatible services to avoid lock-in, to compare and contrast different cloud offerings (their restrictions, costs, performance, etc.), and to evaluate hybrid cloud deployment of their applications, easily and portably.

To implement MEDEA, we leverage the open source AppScale cloud platform [16, 27] and the Neptune HPC configuration language [19]. The plugins that we integrate into the MEDEA deployment engine include compute, storage, and FIFO queue services from Amazon Web Services [8], Microsoft Azure [70], Google App Engine [44], and AppScale. To investigate the potential of MEDEA, we employ the system for a number of different use cases in which we compare and contrast supported plugins in terms of price and performance using various applications, domains, and programming languages.

In summary, we contribute:

- A pluggable cloud software overlay that automates configuration and deployment of applications over cloud compute, storage, and queue services provided by Amazon, Google, Microsoft, and on-premise clouds. This support enables

developers to evaluate and switch between different cloud services without modifying their applications.

- Scripting language support to simplify interaction with the pluggable software overlay that enables the construction of dynamic workflows by users.

- An experimental evaluation of a variety of software packages (from scientific as well as general-purpose application domains) across multiple cloud vendors, which investigates the performance and the monetary cost incurred to execute applications via popular cloud services.

In the sections that follow, we present the design and implementation of the MEDEA execution model. We describe how we plug in different cloud services with MEDEA. We then investigate the cost and performance of a number of different use cases enabled by MEDEA, empirically evaluate its use in different hybrid cloud configurations, and for programs written in different languages. We then discuss related work and conclude.

5.2 Design

By unifying cloud program execution under the MEDEA execution model, we aim to make existing user code interoperable between disparate cloud services. Pushing the complexity of cloud services into an abstract software layer reduces the complexity that

must be present in user-facing code. This also increases portability, reduces lock-in to a particular vendor's services, and enables users to benchmark their applications without needing to become experts with each technology they wish to utilize. In this work, we focus on providing such support for three different and common cloud services:

- Compute services for execution of user code
- Storage services for data persistence
- Queue services which provide a FIFO queue abstraction

The MEDEA execution model uses a combination of these three services to manage and execute programs over supported cloud fabrics. Moreover, it does in a way that hides the details of the implementation of each service, so that users can employ them for execution of their programs without having any knowledge or direct experience with them – users need only have credentials for each cloud she wishes to use.

We depict the design of the MEDEA execution model in Figure 5.1. The MEDEA execution model consists of two components. The first is scripting language support that enables developers to specify the execution environment and deployment preferences for their programs. The second is a deployment engine that plugs in cloud service support to execute applications. We first overview the MEDEA scripting language support and then describe the MEDEA deployment engine.

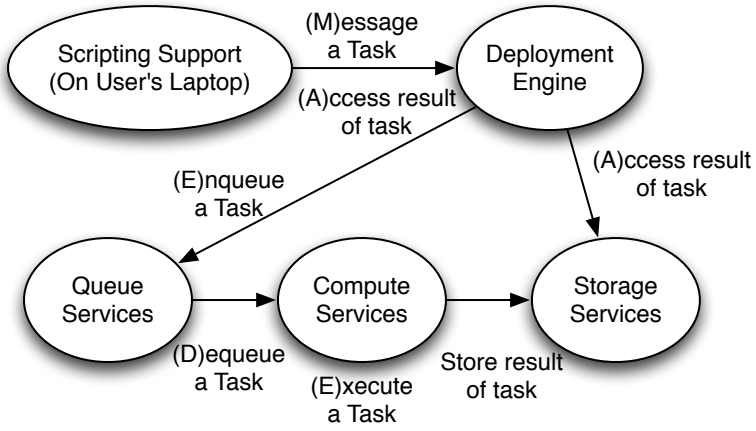

Figure 5.1: Overview of the design of the MEDEA execution model.

To make the use of these services portable and simple, the MEDEA scripting support consists of a single function with which users specify the program they wish to run, its inputs, location, and executable (if any), as well as the names of the service plugins they wish to employ for compute, storage, and queuing. The type of programs that MEDEA currently supports are those that take zero or more arguments as inputs, that communicate only with persistent services, and that generate output through the standard output and standard error streams.

Consider a user who wishes to run a Python n-body simulation in Amazon EC2, store its output in Amazon S3 [5], and have workers in EC2 poll for tasks (here, the n-body simulation is the task) via Amazon SQS [6]. Normally they would need to become familiar with the APIs of each service, their pricing models, and best practices. Once this is done, the user is then locked-in to these three services. In contrast, using

a domain specific language to specify the execution environment of a program reduces the amount of work required to execute the following code:

```
result = babel(
  :executable => "python",
  :code => "/home/user/nbody.py",
  :compute => "ec2",
  :storage => "s3",
  :queue => "sqs")

puts result.stdout
puts result.stderr
```

The formal syntax of calls to `babel` is as follows, where e represents the empty string:

```
S -> babel B | babel T1

B -> :executable => 'binary', C

C -> :code => 'location', A
```

```
A -> :argv => "arguments", E

A -> :argv => ["arguments"], E


E -> :compute => 'ec2', S

E -> :compute => 'azure', S

E -> :compute => 'app-engine', S

E -> :compute => 'euca', S


S -> :storage => 'appdb', Q

S -> :storage => 's3', Q

S -> :storage => 'waz-storage', Q

S -> :storage => 'gstorage', Q

S -> :storage => 'walrus', Q


Q -> :queue => 'rabbitmq', O1

Q -> :queue => 'sqs', O1

Q -> :queue => 'azure-q', O1

Q -> :queue => 'gae-pull-q', O1
```

```
O1 -> :instance_type => 'machine', O2 | e

O2 -> :max_nodes => int, O3 | e

O3 -> :worker => 'inline' | e


T1 -> :type => 'output', T2

T2 -> 'output' => 'location'
```

Users indicate what binary executes their program in the cloud compute service, where the code to execute is located on their local machine, and which compute, storage, and queue services should be used. Users also provide their credentials to each service as environment variables or as additional parameters. Code can be written in any language, as long as the compute service has the correct binary installed to execute it. Our library support for this function validates the submitter's cloud credentials and verifies that the user's code exists on their local computer. Once this is done, it packages this information and sends it to the MEDEA deployment engine.

The object that is returned from calls to `babel` can be used to manually poll for the result of the job. To execute the program using a different service, the developer need only change the value of a function argument for the compute, storage, and/or queue services. For example, changing the value of `:compute` above from `ec2` to `azure`, causes MEDEA to execute the program in Microsoft Azure instead of Amazon EC2.

The returned object provides methods that store the task's standard output and standard error streams. MEDEA also profiles the execution of the task and returns various performance metrics to the user as a field in this object called `metadata`. This latter support enables users to extend their scripts to interrogate the differences between the multiple cloud services to compare and contrast them and to identify the most appropriate one for their application.

We implement the MEDEA scripting language support by repurposing Neptune, a domain specific language that automates the configuration and deployment of high-performance computing applications. Our extensions implement this new function (function semantics, and library support) to facilitate execution of arbitrary user programs. Prior to this work, the implementation of Neptune only allowed for a specialized, statically chosen set of HPC frameworks to be configured and deployed. Arbitrary programs could not be executed, even if the compute service supported it, as the original implementation required an expert user to dictate how each framework executes code (e.g., to run MPI, users must first start NFS, mount a shared filesystem, start the MPI Process Daemon, and so on). In contrast, this work enables any executable installed on a compute service to be utilized automatically by our scripting language support.

5.3 Implementation

The MEDEA scripting language support communicates with the MEDEA deployment engine. The deployment engine provides a software layer that abstracts away common services required for execution of arbitrary programs over cloud compute, storage, and queue services. We plug in actual cloud services to this layer to provide the implementation for each of these operations.

The MEDEA deployment engine employs two key abstractions: the Task Manager (which delegates tasks to clouds) and the Task Worker (which executes the task). This scripting language support and deployment engine perform five steps (which form the acronym MEDEA) to execute programs portably:

1. the scripting language support (M)essages the Task Manager with the program to execute, described by a MEDEA script,
2. the Task Manager (E)nqueues the task to a queue service,
3. a Task Worker (D)equeues a task from a queue service,
4. a Task Worker (E)xecutes the task,
5. the developer (A)ccesses the result of the task from their local computer (from within a MEDEA script).

We implement the MEDEA deployment engine as a web service within the AppScale cloud platform. This platform automatically deploys and starts the MEDEA deployment engine when an AppScale cloud is instantiated. We also provide plugins into the deployment engine's abstractions for each service (queue, compute, and storage) that implement the necessary functionality.

5.3.1 Pluggable Queue Support

When a Task Manager receives a request to run a task from a user, it examines the `:queue` parameter in the user's task to determine which cloud queue the task should be placed on. Acceptable values are:

- `"rabbitmq"` for RabbitMQ, hosted within AppScale (the default)
- `"sqs"` for Amazon Simple Queue Service (SQS)
- `"azure-q"` for Microsoft Azure Queue Service
- `"gae-pull-q"` for Google App Engine's pull queue

These queues provide a scalable FIFO queue service where items can be pushed to or popped from. The Task Manager employs the Factory design pattern, thus, as long as supported queues implement a common API (push/pop), the Task Manager can access them without needing to be concerned with their underlying implementation details.

Once the Task Manager uses a QueueFactory to get a connection to the necessary queue service, it pushes the task to that queue service and returns an acknowledgement to the user's local computer that the task has been started.

Task Workers periodically query the Task Manager for a list of all the queues that tasks can be found on, as well as the cloud credentials needed to access each queue. This is necessary because two users may have different credentials to the same queue service. Each Task Worker uses the same QueueFactory as the Task Manager to get a connection to each queue service and pops off one item of work per core on its machine.

5.3.2 Pluggable Compute Support

After pushing the task onto the specified queue service, the Task Manager ensures that Task Workers are running in the specified compute service. For example, if a user has specified that a task should be executed in Amazon EC2, the Task Manager will ensure that one or more Task Workers are running in Amazon EC2. To provide this functionality, the Task Manager keeps metadata about the number of workers in each cloud and utilizes a ComputeFactory to interact with cloud compute services, based on the value of the `:compute` parameter in the user's task. Acceptable values are:

- `"ec2"` for Amazon EC2, hosted within AppScale (the default)
- `"azure"` for Microsoft Azure

- `"app-engine"` for Google App Engine
- `"euca"` for Eucalyptus

For Amazon EC2, the Task Manager uses the EC2 command-line tools to dynamically spawn or terminate virtual machines. Once virtual machines have been spawned, a Task Worker is started on it, who then polls the Task Manager for work as previously descibed. The Task Manager is also cost-aware, so it does not terminate Task Workers once they have completed a task. Because Amazon EC2 charges on a per-hour basis, the Task Manager terminates Task Workers only near the end of the hour, and only if they are not in use at that time.

Amazon EC2 enables users to remotely log into machines and directly execute programs via the familiar Linux programs `ssh` and `scp`. In contrast, Microsoft Azure and Google App Engine do not support this functionality, as Azure deploys Windows virtual machines, and App Engine does not allow access to the hosted machine at all. To enable interoperable program execution, we contribute `Oration`, a tool that automatically generates Task Workers that execute user-provided applications in different cloud execution systems. Oration takes, as inputs, the name of the cloud to execute the application in, the name of the function to execute, and the name of the file that function can be found in, and then constructs a “cloud-ready” Task Worker that utilizes best

practices from that cloud to execute the user's program. This Task Worker implements the following API:

1. `PUT /task`: Given a function name and its inputs, runs the function stores its output for later retrieval.

2. `GET /task`: Given the name of the task, checks to see if the task is still running, has completed, or has failed.

3. `PUT /data`: Given a location to store data and the data to store, saves the data for later use.

4. `GET /data`: Given a location to read from, returns either the given data (if it exists) or a null value (if it does not exist).

For Microsoft Azure, Windows virtual machines are procured (as opposed to Linux virtual machines in Amazon EC2), so the bootup script we include starts by installing language support for each runtime we wish to execute tasks with (by default, this supports Python and Java, but is extensible to other languages). Microsoft Azure also follows a per-hour pricing model, but in contrast to Amazon EC2, it is a per-wall-clock-hour pricing model. The following process is used to implement MEDEA support on Microsoft Azure:

1. the scripting language support (M)essages the Task Manager with the program to execute, described by a MEDEA script,

2. the Task Manager uses Oration to construct a Microsoft Azure compatible Task Worker and uploads it to Microsoft Azure. The Task Manager then (E)nqueues the task by performing a `PUT /task` on the remotely-hosted web application, which will schedule a background task with the Microsoft Windows Azure Queue Service API. If the task requires any files as inputs, the Task Manager uses `PUT /data` calls to move inputs from the datastore specified to the Windows Azure Storage Service.

3. the Task Worker (D)equeues the task and spawns up workers by performing a `POST /task` to the application server.

4. the Task Worker (E)xecutes the task and stores the output via the Azure Storage Service, a key-value datastore that uses a get/put interface.

5. the developer (A)ccesses the result of the task from their local computer (from within a MEDEA script). The Task Manager retrieves the result by performing a `GET /data` on the remotely-hosted web application.

Finally, the Google App Engine PaaS provides autoscaling, and does not allow its users to programmatically dictate the number of instances that are used. It also employs a restricted runtime that can only execute tasks written in Python, Java, and Go, so we provide specialized Task Workers in those languages to execute Python, Java, and Go tasks. Google App Engine charges on a per-minute pricing model, as opposed to the

per-hour pricing model employed by Amazon EC2 and Microsoft Azure. The following process is used to implement MEDEA support on Google App Engine:

1. the scripting language support (M)essages the Task Manager with the program to execute, described by a MEDEA script,

2. the Task Manager uses Oration to construct a Google App Engine compatible Task Worker and uploads it to Google App Engine. The Task Manager then (E)nqueues the task by performing a `PUT /task` on the remotely-hosted application, which will schedule a background task with the Google App Engine Task Queue API. If the task requires any files as inputs, the Task Manager uses `PUT /data` calls to move inputs from the datastore specified to the Google App Engine Datastore.

3. the Task Worker (D)equeues the task and spawns up workers by performing a `POST /task` to the application server.

4. the Task Worker (E)xecutes the task and stores the output via the Datastore API, an object datastore that uses a get/put interface.

5. the developer (A)ccesses the result of the task from their local computer (from within a MEDEA script). The Task Manager retrieves the result by performing a `GET /data` on the remotely-hosted web application.

Regardless of where the task executes, the Task Worker collects the following data as outputs and metadata from the task:

- The standard output produced by the task.
- The standard error produced by the task.
- The time taken to execute the task.
- The time taken from when the Task Manager received the task to when the task finished executing.
- The time taken to retrieve the code and inputs from the datastore service.
- The time taken to dequeue the task off the queue service.
- Information about the processors on this machine (the contents of `/proc/cpuinfo`).
- Information about memory on this machine (the contents of `/proc/meminfo`).
- Information about disk usage on this machine (the result of `df -h`).

The types of data collected is extensible. In particular, we are looking to extend this system with information about the cost incurred to run the task once cloud providers make this information available programmatically (as opposed to performing estimates or downloading bills from a web page, as is currently done).

5.3.3 Pluggable Storage Support

Once a Task Worker finishes executing one or more tasks, it uses a StorageFactory to get access to a supported storage service. The user indicates which storage service is to be used via the `:storage` parameter, with acceptable values being:

- `"appdb"` for the datastore hosted within AppScale (the default)
- `"s3"` for Amazon Simple Storage Service (S3)
- `"waz-storage"` for Microsoft Azure Storage Service
- `"gstorage"` for Google Cloud Storage
- `"walrus"` for Eucalyptus Walrus

The Task Worker then stores three files in the specified storage service, containing the standard output of the task, the standard error of the task, and the task's metadata (performance profile). At this point, if the user's script accesses the `babel` function's return value, the calls will succeed and return this information to the user.

5.4 Evaluation

We next use our support for MEDEA within AppScale to empirically evaluate how effectively tasks execute within cloud IaaS and PaaS offerings. We begin by evaluating

our pluggable queue support, continue by evaluating a computational systems biology application, and conclude by evaluating implementations of the n-body benchmark application.

5.4.1 Pluggable Queue Evaluation

We begin by using the pluggable queue support that the MEDEA execution model enables to compare the performance and cost of different cloud queue offerings. We investigate one internal and four external pull queue services: RabbitMQ (internal to AppScale), Amazon SQS, Microsoft Azure Storage Queue, and Google App Engine's pull queue. We employ Amazon S3 as the storage service for each task and deploy an AppScale cloud over Amazon EC2, in the manner shown in Figure 5.2. Specifically, we instruct AppScale to automatically deploy a single virtual machine instance as the Task Manager, and in all of our Neptune job requests, we indicate that no more than two Task Workers should be dynamically acquired and used (to limit the monetary costs we can incur). The Task Manager creates Task Workers whenever it detects that the number of tasks waiting to be executed in all queues is non-zero. For Task Workers, we utilize Amazon's `m2.4xlarge` instance type, each of which has 8 virtual cores and 68GB of memory. This instance type is one of the more powerful machines offered by Amazon, and costs $1.60 per hour to lease.

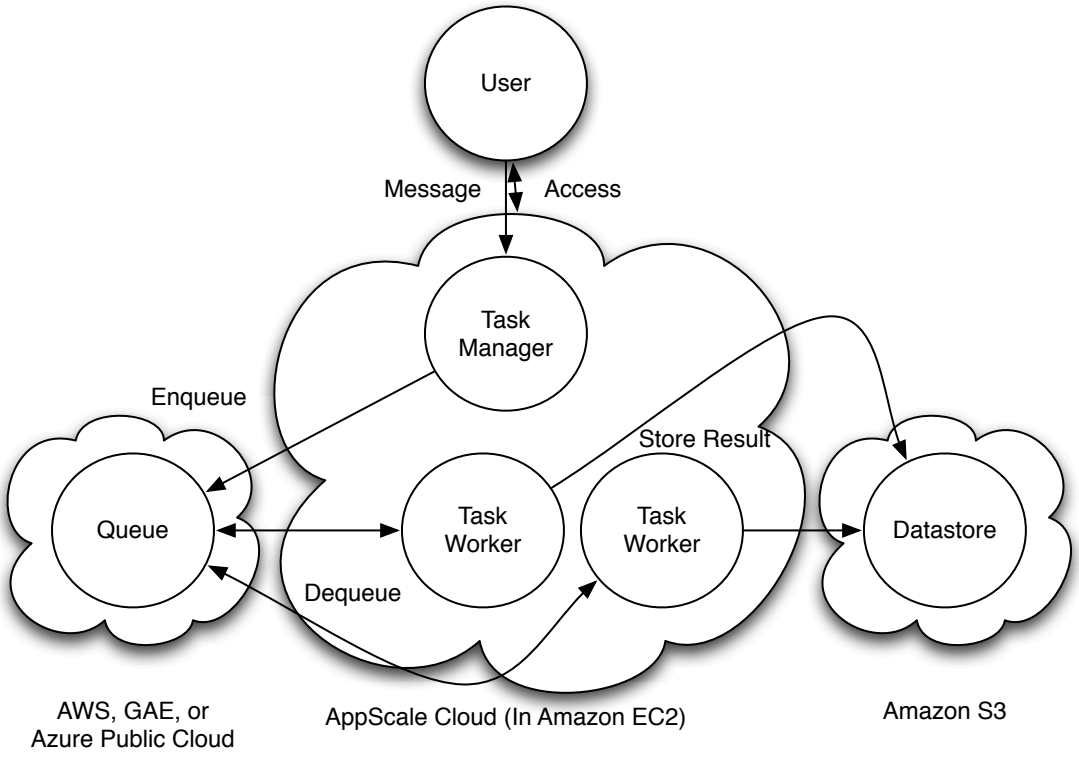

Figure 5.2: Deployment strategy used for the n-body simulation benchmark to evaluate different pull queue technologies.

For this evaluation, we run ten instances of our n-body simulation program (ten tasks) in parallel and report the time that Task Workers spend dequeuing tasks from the queue. Note that the task's payload is nearly constant for all queues used, and only varies when more or fewer credentials are needed to access the queue. The Neptune code that we run for each queue to dispatch the tasks and report the resulting time incurred is:

```
task_info = []
10.times { |i|
  task_info << babel(params)
}


task_info.each { |task|
  if task.return_value != 0
    abort(``Analysis failed: '' +
      task.stderr)
  end
  puts task.queue_pop_time()
}
```

The results of running this code for our n-body simulation on each of the four supported queues is shown in Figure 5.3. RabbitMQ performs the best, because Task

Workers in EC2 always have a RabbitMQ server running on their local machine and thus either need only talk to it or to another machine in the AppScale deployment a short distance away. This improves performance but at the cost of fault-tolerance: in the rare case of an availability zone failing in Amazon EC2, it would also cause our RabbitMQ servers to fail with it. Conversely, Amazon SQS and Microsoft Azure Storage Queue have added fault-tolerance, but perform an order of magnitude slower than RabbitMQ, but outperform Google App Engine's pull queue. This is likely due to the latency between our Task Workers and Google App Engine's pull queue.

These results should not be considered a final evaluation of available cloud queuing services, as such services are constantly upgraded and evolve and improve over time. However, since AppScale can be used at any time, users can employ it to snapshot the current performance of the various queue offerings, evaluate that tradeoff against the cost of using the queue, and choose any queue implementation on demand.

5.4.2 Computational Systems Biology Evaluation

We next evaluate the compute engine offerings that are enabled by making AppScale MEDEA-compatible. The application we use for this study is a Stochastic Simulation Algorithm (SSA) [42]. SSA is form of kinetic Monte Carlo simulation used extensively in computational systems biology. These algorithms are embarrassingly parallel and probabilistic in nature, and require a large number of independent simula-

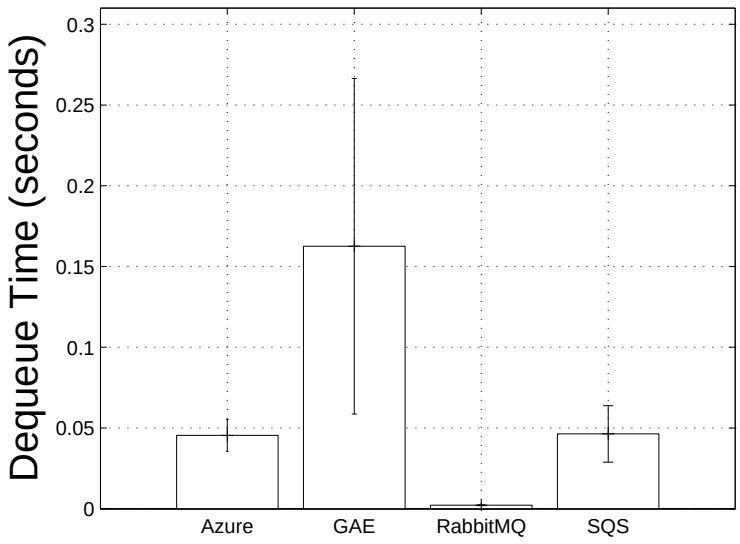

Figure 5.3: Average dequeue time for the Python n-body simulation, when task data is stored in Azure Storage Queue, Google App Engine's pull queue, RabbitMQ, and SQS. Each value shown here represents the average of ten runs.

tions to be executed to achieve an acceptable level of statistical accuracy. The specific algorithm we focus on is the Diffusive Finite State Projection Algorithm (DFSP) [36], which simulates spatially inhomogeneous stochastic biochemical systems. In our study, this algorithm is used to simulate a model of the mating pheromone induced G-protein cycle in budding yeast. We employ this application because it is a canonical example of a compute and data-intensive eScience workflow, thus allowing us to illustrate the performance and cost benefits of executing scientific applications via cloud-based systems. However, it is also an example of an application that is not a web service and thus is

not likely to have a user-written MVC interface (which the Task Manager automatically constructs).

Our evaluation considers the Amazon EC2 IaaS, Google App Engine PaaS, and Microsoft Azure IaaS. For the IaaS offerings, we must manually choose the number of instances (virtual machines) that execute tasks, so we experiment with the performance and cost implications of using 1, 2, 4, and 8 workers. For the Google App Engine PaaS, users cannot dictate the exact number of instances to be used (as it dynamically scales up and down in response to user traffic). To provide a fair comparison, we use the `m1.small` instance type within Amazon EC2, the Small instance type within Microsoft Azure, and the `F1` instance type within Google App Engine. These instances minimize the cost incurred to end users, and provide a comparable amount of CPU and memory between one another.

Figure 5.4 shows the time taken to run a varying number of tasks within Amazon EC2 and Microsoft Azure, for varying numbers of workers. As we increase the number of simulations, we see a roughly linear increase in the amount of time taken to execute these tasks. We also note a standard deviation proportional to the number of tasks run. For Amazon EC2, this is due to the performance variability of tasks that execute within it, a result that has been confirmed by the works of others [10] [62]. As we increase the number of workers used to execute tasks, we also note a corresponding speedup in the total execution time. Note that the x-axis is on a logarithmic scale.

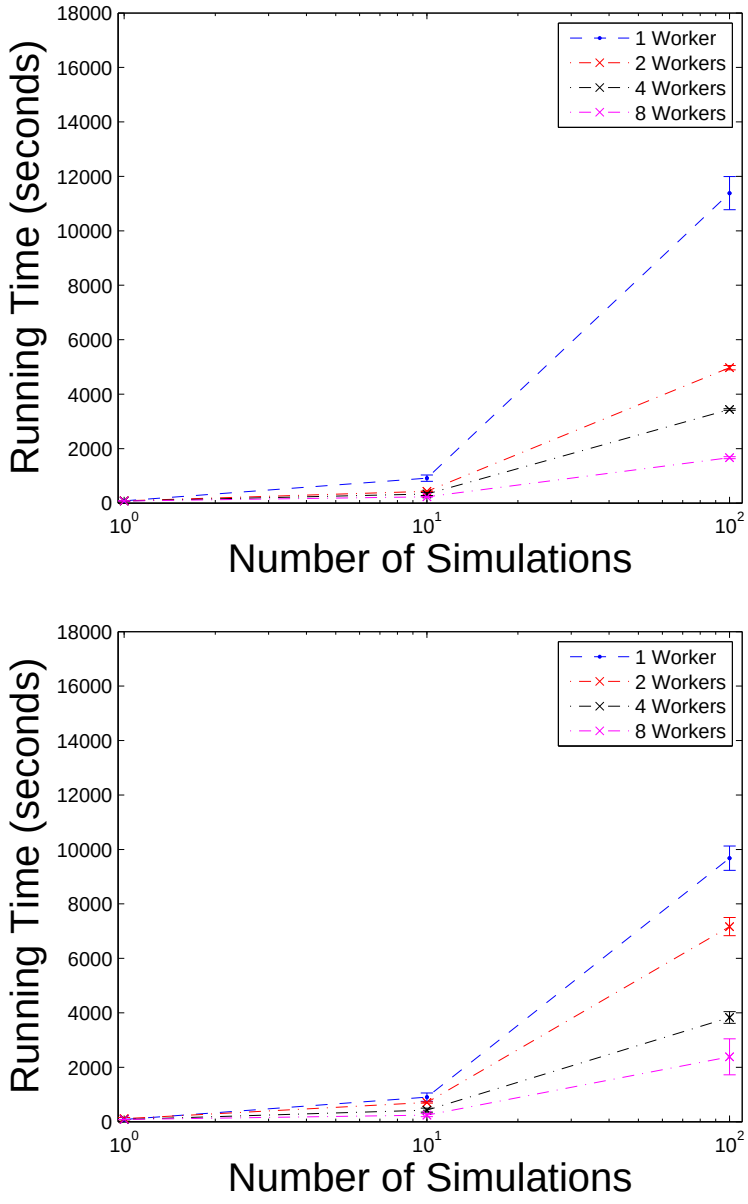

Figure 5.4: Running time for execution of SSA simulations in Amazon EC2 (left) and Microsoft Azure (right), when a varying number of workers are utilized. Each value represents the average of five runs. Note that the x-axis is on a logarithmic scale.

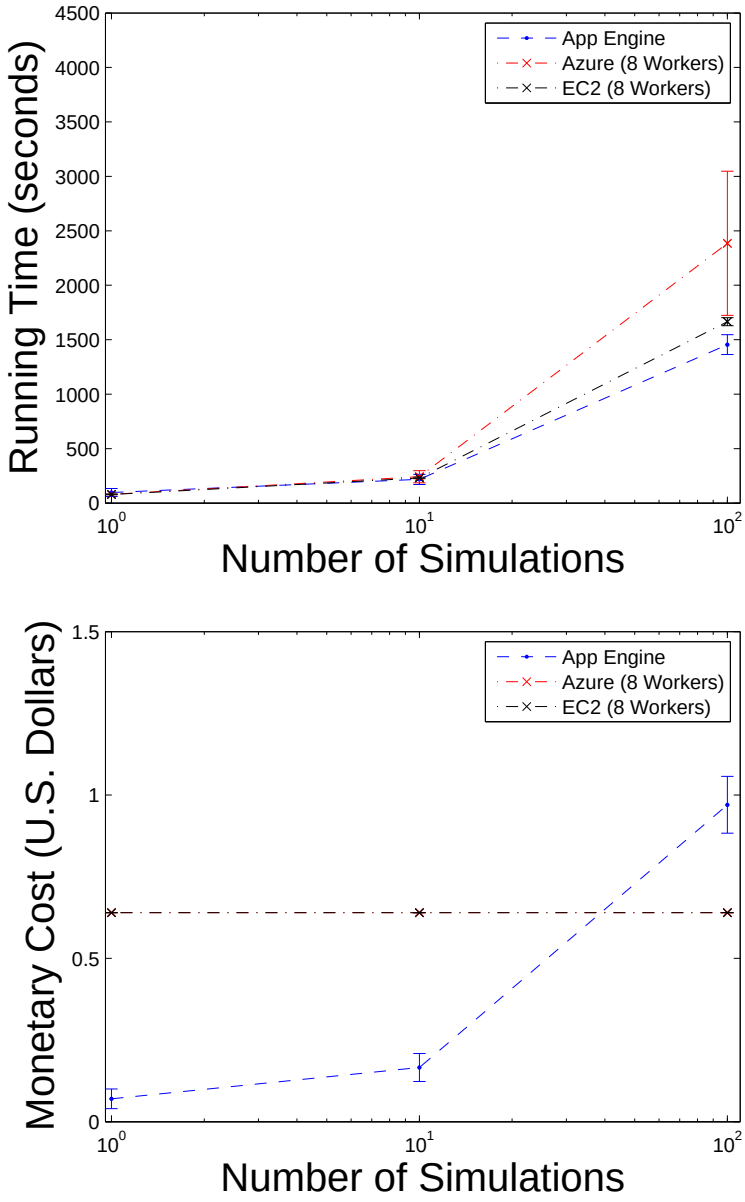

Figure 5.5: Running time (left) and monetary cost incurred (right) for execution of a varying number of SSA simulations in Amazon EC2, Microsoft Azure, and Google App Engine. Each value represents the average of five runs.

Figure 5.5 shows the time taken and the cost incurred when using the maximum number of workers in Amazon EC2 and Microsoft Azure (here, 8 workers) and compares it with Google App Engine, which autoscales and does not allow us to dictate the exact number of workers to use. We note that Google App Engine performs the best of the three engines compared here, and because of its per-minute pricing model, costs less than the other offerings for the lower numbers of simulations. Amazon EC2 and Microsoft Azure both cost the same, which is simply the price of eight machines for a single hour. All three cost a similar amount when the total execution time approaches an hour, which agrees with the per-hour pricing model employed by Amazon EC2 and Microsoft Azure.

5.4.3 Programming Language Shootout Evaluation

In the previous sections, we showed that the MEDEA execution model can be used to enable programs to be executed simply and easily over disparate cloud systems. In this section, we use AppScale's MEDEA support to compare the performance and cost of using different programming language implementations of programs over different public cloud fabrics.

It can be useful to test the performance of a given language, which itself evolves into numerous versions over time. Additionally, creators of a new programming language may wish to compare the performance of their language with other programming lan-

guages on a set of reference implementations. In the spirit of the Computer Language Benchmarks Game [91], we can use AppScale (augmented with MEDEA) to provide a community cloud PaaS that can be used to benchmark algorithms with implementations in different languages on various cloud compute, storage, and queue services.

We evaluate AppScale's MEDEA support in this use case in Figure 5.6. Here, we have taken eleven implementations of the n-body simulation benchmark from [91], written in programming languages of varying programming paradigms, type checking systems, and other language-level design and implementation details. This data shows that most of the implementations of this benchmark perform within the same order of magnitude, with the exceptions of Python and Ruby, which perform two orders of magnitude slower than the others. These results are roughly in agreement with the values published by [91].

While the MEDEA execution model provides users with support for different programming languages and different programming models, it also enables users to investigate and understand the monetary costs of using a particular programming language in a public cloud setting. Moreover, it enables users to investigate the costs of the different pricing models employed by public cloud vendors.

For example, the cost to run the n-body benchmark in different languages using AppScale over Amazon EC2 is shown in Table 5.1. We consider both the cost to run each benchmark via an hourly pricing model (the standard employed by Amazon) and a

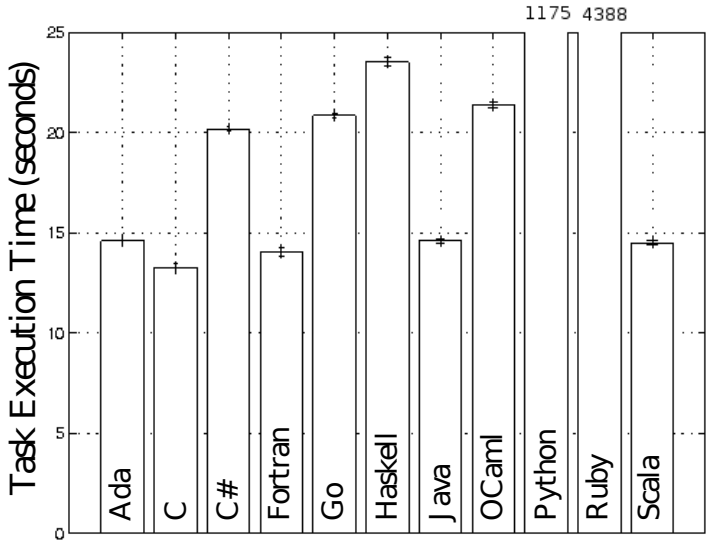

Figure 5.6: Average running time for implementations of the n-body benchmark in different programming languages. Only the time taken to execute the task is considered here. This does not include the time taken to message the system, enqueue the task, dequeue it, or the final result in the remote datastore. Each value represents the average over ten runs.

per-second pricing model (similar to the per-minute pricing model employed by Google App Engine). For the hourly pricing model, all of the benchmarks employed ran within a single hour (except for Ruby), and thus cost $1.80 to run. For Ruby, it took more than an hour to run, so we were assessed charges for two hours of computation, a total of $3.60. If Amazon were to employ a per-second pricing model (as shown in the table), the results exhibit larger differences between language technologies. Specifically, C is the cheapest, with Fortran, Java, Scala, and Ada closely following it. Python and Ruby perform the slowest, costing one to two orders of magnitude more to run.

AppScale, with MEDEA support, thus provides users with a tool that they can use to measure the costs of running their application in a given language or under the different pricing models employed by cloud vendors. Such a tool is important for the investigation of new pricing models and to assess application costs when pricing models change.

Next, we consider the performance and cost of running the Python and Java n-body simulations in Amazon EC2, Google App Engine, and Microsoft Azure. We elect to use only Python and Java (as opposed to all the languages we have implementations for) because Google App Engine only supports programs written in Python, Java, and Go. Here, we utilize a `m1.large` instance in Amazon EC2, a `F4` instance in Google App Engine, and an `Extra Small` instance in Microsoft Azure. We vary the number

Language	Cost Per Task
Ada	\$0.0076 $\pm$ \$0.0002
C	\$0.0069 $\pm$ \$0.0002
C#	\$0.0105 $\pm$ \$0.0000
Fortran	\$0.0073 $\pm$ \$0.0003
Go	\$0.0105 $\pm$ \$0.0000
Haskell	\$0.0120 $\pm$ \$0.0000
Java	\$0.0075 $\pm$ \$0.0000
OCaml	\$0.0110 $\pm$ \$0.0000
Python	\$0.5876 $\pm$ \$0.0057
Ruby	\$2.1944 $\pm$ \$0.0198
Scala	\$0.0075 $\pm$ \$0.0000

Table 5.1: Average monetary cost (in U.S. dollars) incurred to run the benchmarks shown in Figure 5.6 via a per-second pricing model. These costs only include the cost incurred for the virtual machines used. Each value shown here represents the average cost incurred over ten runs.

of bodies to simulate between 5×10^3 and 5×10^7, and run each simulation ten times, reporting the average and standard deviation.

The average running time for the n-body simulation benchmark is shown in Figure 5.7. Amazon EC2 performs the fastest at the lower number of bodies to simulate because it does not dispatch workers to a queue and backend storage service - it simply runs them as it receives them. At the higher number of bodies to simulate, the queue and storage service time no longer dominates the total execution time, and the three services perform roughly the same to one another. We were unable to run the Python n-body simulation at 5×10^7 bodies, because our instances used more than 512MB of memory (the maximum memory allowed for F4 instances) and were killed by the App Engine runtime. Even without this memory restriction, it would have likely taken more

than 10 minutes to execute (the maximum time allowed for background tasks to execute within Google App Engine) and still have been killed by the Google App Engine runtime.

The average cost to run the n-body simulation benchmark is shown in Table 5.2. Amazon EC2 and Microsoft Azure charge users on a per-hour basis, and because all of the n-body simulation times for the Python and Java implementations ran in less than an hour, we were charged for a full hour in these systems. This was $0.32 for a `m1.large` instance in Amazon EC2, and $0.02 for an `Extra Small` instance in Microsoft Azure. Google App Engine charges on a per-minute basis, and because all of the Java n-body simulation times ran in less than a minute, we were charged for a full minute in Google App Engine (as opposed to a full hour in Amazon EC2 and Microsoft Azure). As we used the most expensive instance type in Google App Engine (the `F4` instance type), we were charged $0.0013 for the minute that our program took to execute. We used the same instance type for our Python n-body simulation, but as the larger number of bodies to simulate took more than a single minute to execute, we were charged for more than a single minute of time. Table 5.2 shows the cost incurred by simulating 5×10^7 bodies.

These cost values are not intended to reflect the optimal costs of running the n-body simulation code in Amazon EC2, Google App Engine, and Microsoft Azure. We could have picked instance types that cost less within each of these providers, which could

Cloud Service	Cost
Amazon EC2	\$0.3200 $\pm$ \$0.0000
Google App Engine (Java)	\$0.0013 $\pm$ \$0.0000
Google App Engine (Python)	\$0.0049 $\pm$ \$0.0006
Microsoft Azure Worker Roles	\$0.0200 $\pm$ \$0.0000

Table 5.2: Monetary cost incurred to run the n-body simulation code shown in Figure 5.7 across Amazon EC2, Google App Engine, and Microsoft Azure. Costs are assessed on a per-hour basis for Amazon EC2 and Microsoft Azure, and on a per-minute basis for Google App Engine. The value presented for the Python Google App Engine simulation reflects only the most expensive simulation size (all others are identical to the Java Google App Engine simulation).

have then increased the total execution time for each, which could have then increased the cost incurred (depending on the pricing model used). As was the case for the cloud queue services, implementations of MEDEA provides users with a system that can be used to snapshot the performance and cost of using a cloud IaaS or PaaS system to execute their code.

5.5 Extending MEDEA

The MEDEA execution model enables cloud interoperability for supported programs. In this section, we consider extensions to this model to facilitate greater portability across and ease of use of cloud systems. The extensions we consider in the subsections that follow include simplifying the use of the MEDEA scripting language component via a parallel future construct, task inlining (bypassing the queuing system in some cases), making task deployment more efficient via batching, and utilizing

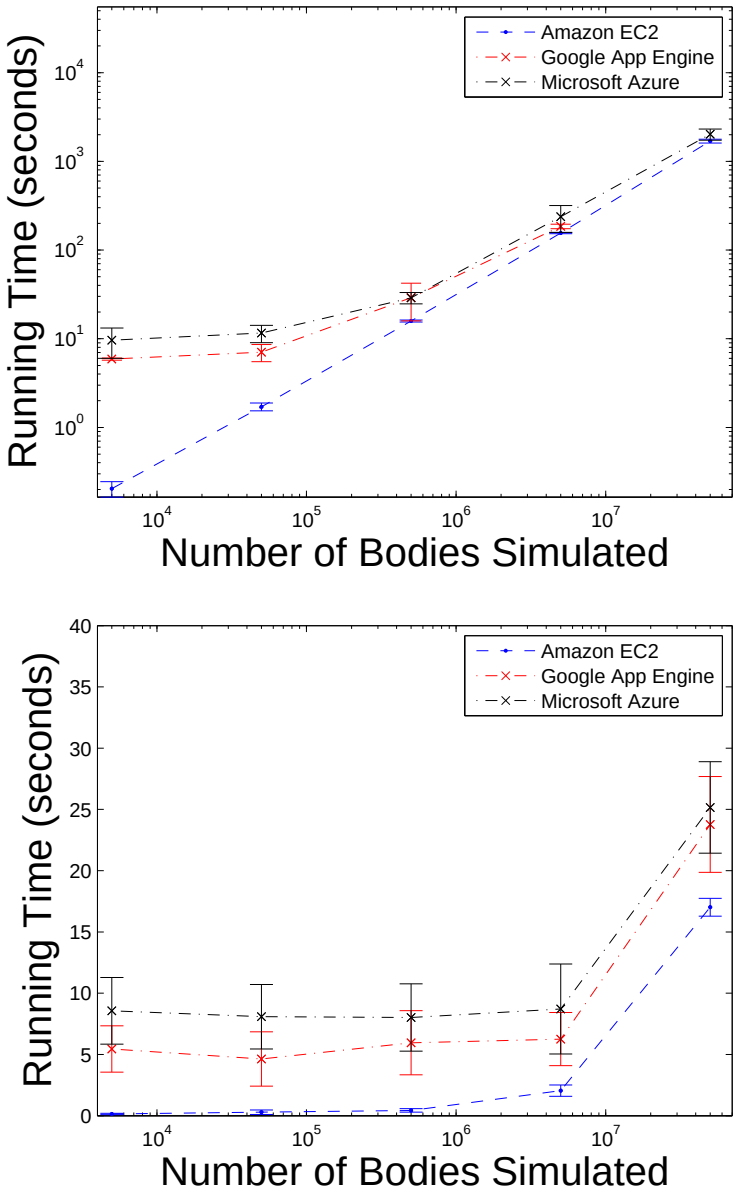

Figure 5.7: Running time for execution of n-body simulations written in Python (left) and Java (right), using Amazon EC2, Google App Engine, and Microsoft Azure. Note that both axes are on logarithmic scales.

caching on Task Workers to eliminate unnecessary data retrieval. Throughout this section, we also consider the impact of these optimizations on the popular MapReduce programming model, popularized in [33], with an emphasis on the single embarrassingly parallel applications that this model supports.

5.5.1 Automatic Polling via Futures

We begin by considering ways to improve the use of the MEDEA scripting language component for distributed, multi-cloud application deployment. Towards this end, the result of an invocation of `babel()` returns an object that encapsulates information about the task's execution. Users can poll for the output of the task from within a MEDEA script, to determine when a task has completed. The MEDEA script for doing so would look similar to the following:

```
result = babel(params)
output_params = params.dup
output_params[:type] = "output"
loop {
  if babel(output_params)[:done]
    break
  end
  sleep(10)
```

```
}

puts result.stdout

puts result.stderr

puts result.metadata
```

To automate the process of polling (reducing the amount of work a user must perform) and to enable the script to do other work while waiting for the task to finish (e.g. execute more tasks), we investigate implementing the `babel` function as a future [51, 98, 97]. A future is a simple and elegant programming language construct that enables developers to introduce asynchronous computation into their programs.

To enable this in our scripting language support, we modify the design and implementation of the `babel` function to return a future for the object (the task's result) instead of the object itself. When `babel` is invoked, a background thread is spawned that dispatches a message to the MEDEA Task Manager, polls for its output, and blocks if the user calls any of its methods or accesses any of its fields before the task has completed. We employ Ruby's metaprogramming features to implement implicit future semantics for the `babel` function, so users need not know that the object they are accessing is a future. The previous example, which used polling, can be rewritten when futures are used, as follows:

```
result = babel(params)
```

```
puts result.stdout  # this will block until
puts result.stderr  # the task completes
puts result.metadata
```

5.5.2 Inlining Task Execution

MEDEA provides a task execution model that utilizes a distributed queue service to pass information between the Task Manager and Task Workers. Yet for short-running tasks, the overhead incurred by storing tasks in a queue may be longer than running the task immediately within the Task Manager. Therefore, we enable users to specify the value of `:worker` to be `inline` to indicate that the task should be "inlined" - that is, it should not follow the standard MEDEA execution model, and instead should be immediately executed inline within the Task Manager.

To evaluate the benefits and drawbacks of task inlining in MEDEA, we deploy a set of tasks that count the number of words in an input corpus using the map-reduce programming model [33]. Here, each Map task performs a word count on the works of William Shakespeare (roughly 5MB in size), and each Reduce task aggregates the results from each Map task. Our extensions to the MEDEA scripting language support that facilitate the use of futures enables supported programs to be "chained" together in a manner similar to that of a workflow system, except that this system is fully Turing-

complete, as opposed to the XML-based systems that most workflow systems employ. Here, we pass the output of each Map task as an input to the final Reduce task. The MEDEA script for this MapReduce job looks like the following:

```
common_params = {
    :storage => "s3",
    :queue => "sqs",
    :instance_type => "m2.4xlarge",
    :max_nodes => 3,
    :worker => "inline"
}

map_params = common_params.dup
map_params[:code] = "/.../wc.py"
map_params[:argv] = ["/.../shakespeare.txt"]

num_mappers.times { |i|
  param_list << map_params
}
map_tasks = babel(param_list)
```

```
outputs = map_tasks.map { |task|
  task.output_location
}
reduce_params = common_params.dup
reduce_params[:code] = "/.../reduce.py"
reduce_params[:argv] = outputs
reduce_task = babel(reduce_params)
```

In this experiment, we vary the number of Map tasks dispatched when inlining is used and when it is not used, and report the results in Figure 5.8. The data shows that when we inline a small number of tasks, inlining performs better than the non-inlined case, but as we inline more tasks, it causes a near-linear slowdown on the system (as all inlined tasks are run on the Task Manager, who runs them serially). In the non-inlined case, the number of tasks we run are smaller than the number of available cores, so the total execution time is roughly constant.

As part of ongoing and future work, we are investigating how to automatically detect when a task can and should be inlined vs deployed via the tasking system. Such support will remove the burden from the programmer to decide when it is best to do so. Since the MEDEA Task Manager collects performance data and task behavior, we will use this information to guide this inlining functionality that we currently have in place.

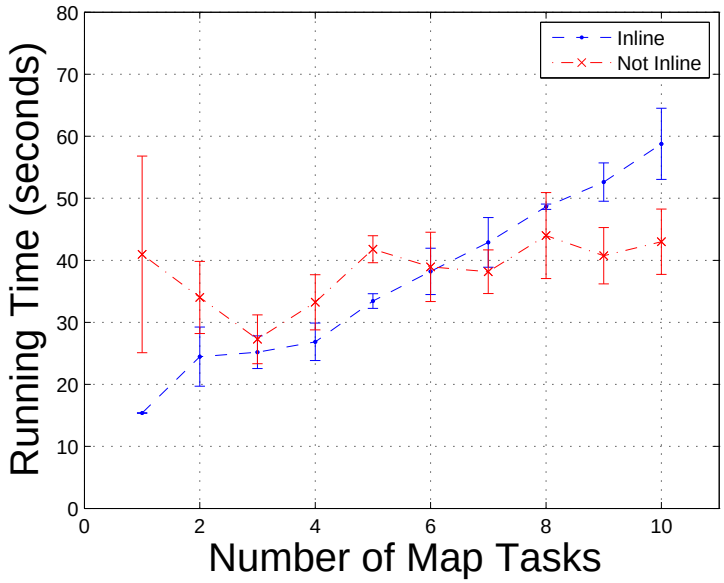

Figure 5.8: Average end-to-end time to run inlined and non-inlined tasks for the Python MapReduce WordCount code for varying numbers of Map tasks and a single Reduce task. Each value here represents the average of five runs.

As a first step, we augment the Task Manager to automatically inline up to one task per core on its node (to avoid CPU thrashing from overprovisioning tasks).

5.5.3 Batch Task Execution

As many real-world use cases need to run more than a single task, the ability to batch task invocations can be useful. Our next MEDEA extension therefore facilitates batch task invocation.

To enable this, we modify the invocation of `babel` to take advantage of Ruby's duck typing capabilities so that it can receive either a Ruby hash (a single task invo-

cation) or a Ruby array of hashes (multiple task invocations) as arguments. In case of the latter, the multiple task requests are dispatched all at once to the MEDEA Task Manager within AppScale, and a Ruby array of futures of task objects is returned as a result. A code example that runs ten n-body simulations in Google App Engine and prints their outputs is:

```
tasks = []
10.times { |i|
  tasks << babel(params)
}
tasks.each { |task|
  puts task.stdout
}
```

That example dispatches 10 tasks individually to MEDEA to be executed, and prints the result of each task. Alternatively, the 10 tasks could be dispatched in a single batch request as follows:

```
param_list = []
10.times { |i|
  param_list << params
}
```

```
tasks = babel(param_list)

tasks.each { |task|

  puts task.stdout

}
```

Figure 5.9 shows the performance improvements that are possible from batching requests for the Python MapReduce WordCount code. When only a single Map task is used, the two systems perform roughly equally. However, as the number of tasks to run increases, batching the tasks into a single request saves a significant amount of time. The amount of time spent is linear in the number of tasks in both cases, as the MEDEA scripting language checks that the inputs and code to run are in the remote datastore (or copy them to the datastore if they are on the local disk), and that the output location specified does not exist (to avoid accidentally overwriting existing data).

5.5.4 Caching Support

Many use cases, such as those in the MapReduce programming paradigm, can execute many instances of a single program (here, the Map program) on a single machine. Our final MEDEA extension is therefore concerned with providing caching for programs and inputs on machines.

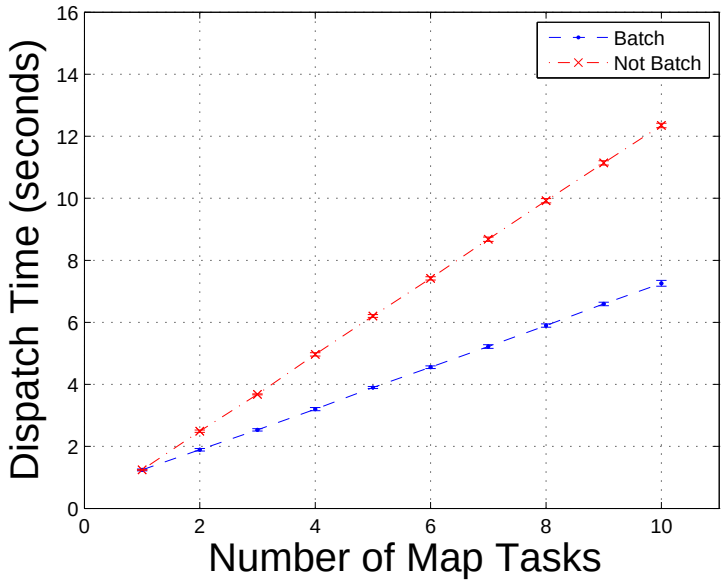

Figure 5.9: Average time to dispatch requests in a batched fashion and a non-batched fashion for the Python MapReduce WordCount code for varying numbers of Map tasks. Each value here represents the average of ten runs.

To implement caching support, whenever Task Workers would normally download a program or an input file, they first check to see if they have the file already stored locally in `/var/cache/medea`. If so, they do not attempt to download the file again (otherwise, they download the file from the remote datastore as usual).

Figure 5.10 shows the results of executing WordCount Map tasks over the baseline MEDEA system, as well as the performance improvements that occur when we batch the tasks into a single request. Finally, we also consider the performance improvements of using batching as well as caching the Map program and its input file (the works of William Shakespeare). Although batching does improve performance (by 23% at 64

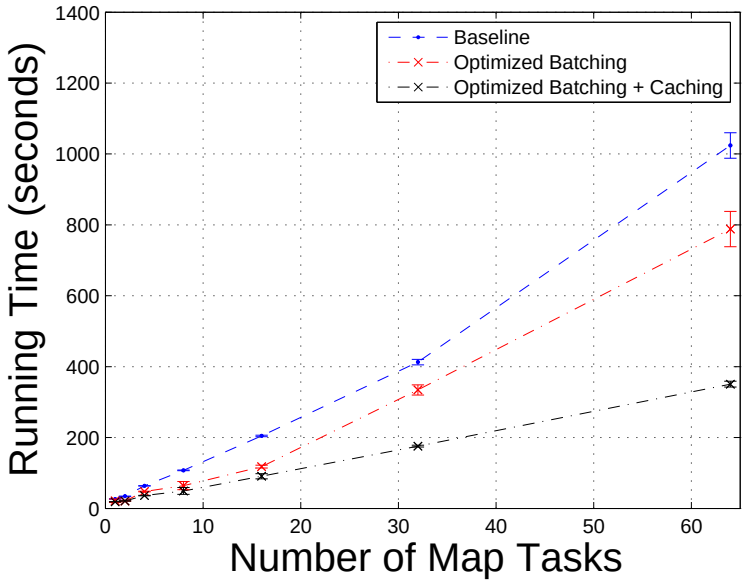

Figure 5.10: Average time taken to execute a varying number of WordCount Map tasks, when the baseline system is used, when batch task support is enabled, and when batch task support and caching is employed. Each value here represents the average of five runs.

Map tasks), adding caching support has a much greater impact on total execution time (by 65% at 64 Map tasks). This is because the input file is 5MB in size, so not re-downloading it for every Map task reduces the normal "download-execute" process to simply "execute".

5.6 Related Work

The MEDEA execution model builds upon and expands upon the works of others. Primarily, MEDEA is implemented by repurposing AppScale and Neptune. App-

Scale enables automatic deployment of Google App Engine applications written in Python, Java, and Go, and Neptune expands this to support HPC applications written in MPI [48], UPC [38], X10 [25], KDT [67], and others. The implementation of MEDEA via AppScale and Neptune goes further by enabling automated execution of programs written in any programming language, across compute, storage, and queue services offered by Amazon, Google, and Microsoft. Furthermore, our extensions automatically collect and expose metadata about the program, allowing users to write programs that quantify the performance characteristics of the programs they execute.

MEDEA is inspired in part by the YCSB project [30] and its successor, YCSB++ [74]. These projects enable users to benchmark popular non-relational datastores (e.g., HBase [52], Cassandra [23]) on a consistent workload to provide information about their underlying performance characteristics. MEDEA goes a step in an orthogonal direction: instead of providing a system that can be used to benchmark datastores in a single cloud IaaS, implementations of MEDEA can be used to benchmark compute, storage, and queue services tied together in a single cloud IaaS or PaaS, or utilized as a hybrid cloud.

Elastisizer [53] provides users with the ability to automatically acquire IaaS resources and run tasks over them, and like Neptune, provides a language-like interface to abstract away resource usage. Elastisizer differs from MEDEA in two critical ways. First, Elastisizer can run only Java tasks that conform to the Hadoop MapReduce framework / programming model, whereas our implementation of MEDEA can

run tasks written in any programming language, in any programming model. Secondly, Elastisizer's declarative language serves a different purpose than Neptune does. Elastisizer enables users to query the system about the performance of their tasks for certain data sets, while Neptune enables users to specify the tasks themselves and chain them together with other tasks.

Other offerings provide either a library or runtime component similar in nature to MEDEA. At the library level, the Google App Engine Pipeline API [46] offers developers writing Python or Java applications the ability to chain together functions that should be asynchronously executed. Pipeline differs from MEDEA in that it is not a published work and that its primary implementation runs on a closed runtime stack within Google's infrastructure. Furthermore, arbitrary languages are not supported, as the Google App Engine runtime stack only supports applications written in Python and Java that use whitelisted APIs. Finally, only functions that were uploaded with the user's application can be executed (that is, arbitrary Python or Java functions cannot be uploaded and executed).

In a similar vein, Pegasus [34] and Swift [35] allow users to specify an execution plan (typically in XML) to connect programs together. In contrast to our language support, these execution plans are not Turing-complete, which prevents them from being used in scenarios where the result of a computation can cause an arbitrary piece of code to be executed or require some type of human interaction (which may be the case when

an expert user is needed to analyze the result of a computation). Furthermore, these systems are not designed to be pluggable in nature: they intend only to utilize a single, statically owned set of resources to run applications.

Workflow systems execute and connect programs together automatically, which is conceptually similar to what MEDEA offers. AME [99], Condor [90], StratUm [73], and Amazon Simple Workflow Service (SWF) [7] are recent works that seek to address this problem, for differing domains. AME is designed to run on supercomputers, where millions of cores may be present, while Condor and StratUm utilize grids, which do not provide elasticity and thus do not allow users to dynamically acquire nodes. While Amazon SWF does operate within a cloud environment, it is specialized to the Amazon cloud, which encourages lock-in to Amazon's compute, storage, and queue services. Furthermore, the specification language that connects computation together in Amazon SWF is not Turing-complete, limiting the types of computation that can be run in a manner similar to Pegasus and Swift.

5.7 Summary and Conclusions

MEDEA provides users with an execution model whose implementation automatically deploys applications to compute, storage, and queue services offered by popular cloud IaaS and PaaS systems. The motivation behind MEDEA is to reduce the com-

plexity with learning the myriad APIs, cost models, and best practices needed to utilize these cloud services effectively. To achieve this goal, MEDEA leverages the Neptune domain specific language and uses it to provide users with a Turing-complete language that they can program in to detail what services should execute their application, and not have to be concerned about learning how to execute their application. Our evaluation of MEDEA shows that, while cloud systems may perform similarly for a given piece of code, they can vary greatly with respect to the price users pay to run their code in these systems, due to the pricing models that clouds enforce. MEDEA enables users to evaluate these systems easily for their own applications, and enables users to compare and to contrast the performance and cost of each of these services, as the services and their applications evolve over time.

The text of this chapter is, in part, a reprint of the material as it appears in [20].

Chapter 6

Exodus: An Application Programming Interface for Cost-Aware, Cloud-Aware Program Execution

In this chapter, we present Exodus, an application programming interface for automatically estimating the total execution time and cost incurred to execute applications via cloud Infrastructure-as-a-Service offerings. Exodus abstracts away the details of cloud-based program execution by providing library support to enable users to programmatically dictate (via a Turing-complete language) how to optimally execute their applications. Exodus provides optimizers that schedule application execution based on performance as well as cost, and considers heterogeneous resource types provided by the Amazon EC2 public cloud and the Eucalyptus private cloud. By doing so, Exodus provides users with the ability to estimate how long it will take and how much it will cost to execute their application over cloud resources, without first needing to become an expert in each cloud service that they wish to utilize. We investigate the potential of

Exodus by employing it for computational systems biology applications (where scientists are typically not cloud experts) written in low-level and high-level programming languages. Exodus is able to predict total execution time with 2%-16% error for the applications surveyed, when optimizing for total execution time or cost.

6.1 Introduction and Motivation

The myriad types of cloud service offerings and instance types that each provider offers makes it challenging and time consuming for new and expert users alike to determine which cloud, instance type, and how many instance types are optimal for their application. Furthermore, what is "optimal" varies from one user to another, and can include minimizing overall cost incurred, minimizing the total amount of time needed to execute the user's code, or a variety of user-defined metrics (e.g., execution must finish before a particular deadline). The complexity of estimating how much it will cost to run in the cloud is therefore often done in an ad-hoc manner, typically resorting to back-of-the-envelope style calculations, which in practice can be extremely inaccurate.

This work attempts to reduce this complexity by proposing an **application programming interface for cloud-aware program execution** that can consider performance, cost, or user-defined metrics. This interface, which we call Exodus, profiles bag-of-tasks applications and automatically determines the ideal instance type to use.

Optimization is performed based on total execution time, monetary cost, or via user-provided metrics. After determining which resources are optimal to execute a user's application, Exodus executes the application without user intervention.

Applications can be executed in the Amazon EC2 public cloud as well as in on-premise Eucalyptus clouds. In both cases, Exodus automatically determines how to use resources optimally to execute a user's application, and uses the AppScale cloud platform [16] [27] to automatically configure and deploy applications. To implement the Exodus API, we leverage the Neptune HPC configuration language [19] [21]. To investigate the potential of Exodus, we employ the system for a number of different use cases, in which we compare and contrast scientific and general-purpose applications in terms of performance, price, and a weighted average of the two in the Amazon EC2 public cloud.

In summary, we contribute:

- The design of an application programming interface that enables users of any programming language to automatically determine which cloud resources are optimal for their application, for some user-provided definition of "optimal".

- An implementation of this API in the Neptune domain specific language, that automatically executes applications via Amazon EC2 and Eucalyptus.

- An experimental evaluation of bag-of-tasks applications from computational systems biology as well as general purpose applications, demonstrating how Exodus is able to automatically determine the instance types needed and the number of instances needed to minimize the cost to execute these programs or maximize their performance via popular cloud infrastructures.

In the sections that follow, we present the design and implementation of Exodus. We describe how we provide an API that is able to determine how to optimally execute a user's program, for various definitions of "optimal". We then investigate the cost and performance of executing programs via Exodus by evaluating scientific and general purpose applications written in different programming languages. We then discuss related work and conclude.

6.2 Design

By providing a solution at the level of an application programming interface (API), we aim to abstract away the complexities associated with cost and performance management of user-provided applications. This enables users to focus on programming their applications, instead of having to spend time to become an expert on each cloud service they wish to potentially use. This also increases application portability, as appli-

cations can be executed over supported clouds without needing to manually port their application from one cloud to another. In this work, we focus on providing an API that:

- Estimates the total execution time and monetary cost of executing a user's application via a cloud IaaS
- Uses time and cost estimates to optimize the execution of a user's application, via standard or user-defined metrics
- Is aware of the differing types of hardware profiles ("instance types") offered by cloud IaaS vendors, specifically how they vary with respect to performance and cost
- Automatically executes applications via a cloud IaaS

Our realization of this API in Exodus is designed to accomplish the above to facilitate cloud program execution that is both cost-aware and performance-aware, and can leverage existing research that schedules applications based on performance characteristics [94] [11]. The latter feature is designed to provide a "pluggable" optimizer, to serve use cases where application-specific metrics drive the underlying execution.

One such use case is a deadline: a scientist may have a paper deadline at a certain time, and needs the execution of their experiments to finish before that deadline, regardless of the cost incurred. Alternatively, a company may have a budget that they

have allocated for use with public clouds, and may want to execute as many programs as they can without exceeding that budget. Other use cases may evolve over time, so our API must be pluggable to accommodate user-provided metrics.

The Exodus API consists of a single function call that users invoke to specify the execution environment, deployment preferences for their program, and credentials for each cloud IaaS they wish Exodus to consider. The execution environment includes information about where the program is stored locally, a list of arguments to invoke the program with, and (if necessary) the name of the executable that should be used to run the program. The formal syntax of calls to `exodus` is as follows:

```
S –> exodus E


E –> :executable => 'binary', C


C –> :code => 'location, A


A –> :argv => "arguments", N

A –> :argv => ["arguments"], N

A –> :argv => Proc.new { |i| user code }, N


N –> :num_tasks => int, O
```

```
O -> :optimize_for => :performance, M

O -> :optimize_for => :cost, M

O -> :optimize_for => Proc.new { |t, c| user code }, M


M -> :max_instances => int, I1


I1 -> :clouds_to_use => [I2]

I2 -> :AmazonEC2, I2 | :AmazonEC2

I2 -> :Eucalyptus, I2 | :Eucalyptus
```

The execution of invocations to Exodus follows the pattern shown in Figure 6.1. Once the user's program invokes Exodus, the runtime validates the parameters that specify their execution environment, deployment preferences, and cloud credentials. Then the runtime dispatches a message to a specialized service running within the AppScale cloud platform, which is then charged with the task of acquiring virtual machines in each cloud the user specifies, with the given credentials. The application is then executed, the result of the job (its standard output and standard error) saved to the cloud storage service that the user requests, and is retrieved by Exodus and passed to the user (if their program requests it).

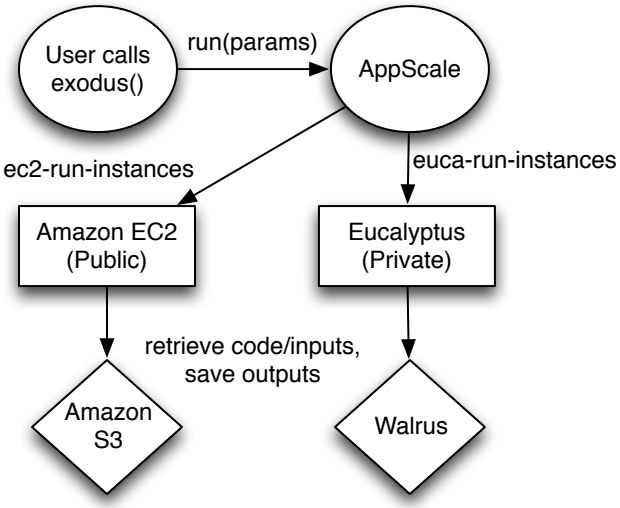

Figure 6.1: An overview of how Exodus abstracts away cloud IaaS interaction via the AppScale cloud platform.

6.3 Implementation

Exodus provides an application programming interface to users that abstracts away how to optimally deploy applications via cloud Infrastructure-as-a-Service offerings. Our implementation of this API in the Neptune domain specific language, itself a superset of the Ruby programming language, facilitates the use of Turing-complete programs to execute programs (as opposed to XML-based or rule-based solutions). This enables users to deploy their applications and take arbitrary actions based on their resulting outputs. For example, scripts that interact with Exodus can e-mail the outputs of their code to a user for expert analysis, or integrate with other libraries to perform statistical analysis on a user's behalf.

We detail the types of parameters that Exodus API calls require, how profiling information is generated, and how this information is harnessed to optimally decide how to execute user-provided applications.

6.3.1 Application Programming Interface

Our implementation of the Exodus API provides users of the Neptune domain specific language with a single function call, named `exodus`, that users invoke to specify the execution environment, deployment preferences, and credentials for each cloud IaaS they wish Exodus to consider. Specifically, the following parameters are required to specify the execution environment that the user's code should run under:

1. `:code` - The location on the local filesystem where the user's code is located.
2. `:argv` - An Array of Strings that represents the arguments that should be passed when the code is executed. Any files specified here are copied to the remote cloud service prior to code execution.

Users must then specify deployment preferences for their application, which include:

1. `:num_tasks` - The number of times that Exodus should execute this code in a cloud service, for bag-of-tasks applications.

2. `:optimize_for` - The definition of how to optimally execute the user's application. Users may provide the Ruby Symbol (String constants) `:performance` to indicate that their application should be executed as quickly as possible, or `:cost` to indicate that their application should be executed as inexpensively as possible. Users may also provide a Function (a Proc in Ruby nomenclature) that is given Exodus' projected total execution time and cost incurred as inputs, and returns a Float, which our optimizer will attempt to minimize amongst the instance types available.

3. `:max_instances` - The maximum number of instances that should be utilized to execute a user's code over. By default, 19 instances are set as an upper limit for Amazon EC2 (since 20 instances are the maximum number a user can acquire without contacting Amazon beforehand, and one instance is dedicated within AppScale to application management), but a user may wish to limit the number of instances employed to execute their applications.

Additionally, users must also specify credentials for each cloud IaaS that they wish Exodus to consider executing programs over. This is done via the `:clouds_to_use` parameter, which can include `:AmazonEC2` for the publicly available Amazon Elastic Compute Cloud and `:Eucalyptus` for an on-premise Eucalyptus cloud.

Both of the clouds that we currently support require similar credentials to be specified, including certificates, private keys, and access keys. Exodus validates that each set of credentials given are able to access the cloud service before attempting to execute the user's application.

For applications that require unique identifiers or seeds for random-number generators, executing a number of identical applications via Exodus would produce identical results. To better serve these applications, we amend the `:argv` parameter to also accept a Function from users. This function is invoked once for every task that Exodus should execute, and receives the sequence ID of this task. This enables applications to seed their random-number generator with the sequence ID and produce unique results in their applications.

6.3.2 Cloud-Aware Program Execution

Once a user invokes `exodus` in their Neptune script, the Neptune runtime (extended in this work to support Exodus API calls) checks to see if the code has profiling data available. The current implementation checks on the local filesystem for this information, but is extensible to utilize remotely available sources. If no profiling data is found, the Neptune runtime then invokes the profiler that the user has specified. This work contributes two profilers: NaïveCPUProfiler and RemoteCloudProfiler.

The Neptune runtime implements NaïveCPUProfiler by invoking one or more copies of the application locally (once by default) to generate this profiling information, noting both the total execution time (average execution time if more than one run is performed) and the speed of the CPU on the local machine. This information is written to the local filesystem for future deployments.

The Neptune runtime then examines the `:clouds_to_use` parameter, and for each cloud specified, determines which instance type, and the number of machines for that instance type is optimal to execute the desired number of invocations of the user's program. It begins executions by estimating the total execution time needed to execute `:num_tasks` programs on each instance:

$$t_{np} = \frac{: num\_tasks \times t_1}{n \times p} \tag{6.1}$$

where t_n represents the time required (in seconds) to execute `:num_tasks` programs on n machines, with p processors each, and t_1 represents the time needed to execute one program on one machine with one processor. To estimate the amount of time needed to execute one program on each instance type, we scale the local execution time of the profiled application to the CPU speed of each instance type:

$$t_{remote} = \frac{cpu_{remote} \times t_{local}}{cpu_{local}} \tag{6.2}$$

where t_{remote} represents the estimated time to execute the user's application on a CPU at a clock speed of cpu_{remote} MHz, and t_{local} represents the time needed to execute the user's application locally on a CPU at a clock speed of cpu_{local} MHz.

Our optimizer estimates the total execution time via Equation 6.1 for each instance type, when between 1 and a maximum (max) number of instances are utilized. Our application executor within the AppScale cloud platform dedicates a single node to manage applications.

Next, our optimizer estimates the cost incurred for each instance type, and for between 1 and max instances at each instance type to execute the user's application. This cost estimate is given by:

$$c_{np} = Ceiling(t_{np}, 3600) \times n \times c_{1,EC2} \tag{6.3}$$

where c_{np} represents the estimated cost to utilize n instances, each with p processors, with a cost of $c_{1,EC2}$ U.S. dollars for one hour of use. This pricing model is specific to the per-hour metering that Amazon EC2 employs, and is easily adaptable to the cost model that Google App Engine employs (per-minute, with a minimum charge of 15 minutes) as follows:

$$c_{np} = Max(15 * 60, Ceiling(t_{np}, 60)) \times n \times c_{1,GAE} \tag{6.4}$$

For the Eucalyptus IaaS offering, we assume that it is hosted on-premise, and is offered freely to all users. This means that we set c_1 to 0.

Once we have calculated t_{np} and c_{np} for each instance type, for each number of instances we can acquire, we compute an "aggregate score" that factors in both total execution time and cost incurred:

$$s_{np} = \alpha t_{np} + (1 - \alpha) c_{np} \tag{6.5}$$

where s_{np} represents the aggregate score for executing our code over n instances, each with p processors, and α is a value between zero and one that indicates whether we should bias the calculation towards being performance-effective or cost-effective. For the case when a user indicates `:optimize_for => :cost`, the optimizer sets $\alpha = 0$, thus considering only cost in Equation 6.5. Similarly, if a user indicates `:optimize_for => :performance`, the optimizer sets $\alpha = 1$, thus considering only total execution time (performance) in Equation 6.5. Users may also explicitly state α's value.

The Neptune runtime then calculates the aggregate score for each instance type, for each number of nodes available, for each cloud that the user wishes to execute code over. It finds the minimum value, and if `:recommend_only => true` is set,

reports the projected execution time and cost, aggregate score, recommended instance type and number of nodes, and cloud to use.

Instead of scaling solely on CPU speed, as NaïveCPUProfiler does, we also provide RemoteCloudProfiler. RemoteCloudProfiler sends a SOAP message to the AppScale cloud platform that instructs it to acquire one of each instance type within each cloud the user has specified to use, and records the total execution time to execute one invocation of the user's program. This process is not free in public clouds: for the Amazon EC2 public cloud, profiling an application that executes in less than one hour on all instance types would charge the end-user $6.71. Work is ongoing to utilize Spot Instances to reduce this cost, although this is only applicable for Amazon EC2, and would significantly increase profiling time (as Spot Instances are not always immediately made available).

If `:recommend_only` is not set (or is set to `false`), then the Neptune runtime proceeds to execute the user's application. To do so, it constructs `:num_tasks` Neptune tasks and executes them, via the AppScale cloud platform (as detailed in [21]). The Exodus API then returns an Array of length `:num_tasks`, which users can access to query the standard output and standard error of their programs.

6.3.3 Pluggable Optimizers

To leverage the vast array of existing literature on program optimization as well as to support unique use cases with Exodus, we provide a pluggable optimizer within Exodus. To that end, Exodus enables users to specify their own aggregate scoring function that should be executed in lieu of the one in Equation 6.5. The user's function must receive two inputs: the projected total execution time and the projected cost. The user's function should return an Integer or Float value, so that the Neptune runtime can attempt to find the minimum aggregate score across each instance type, for each number of nodes available.

As an example, consider the use case where a user wishes to run their application and does not wish the total cost to exceed one dollar. The user could invoke the Exodus API as follows, to serve this use case:

```
exodus(
 :code=>"/home/user/g-protein.dfsp",
 :optimize_for=>Proc.new{|time, cost|
  if cost > 100 # cents
   return INFINITY
  else
   return cost
```

```
  end
 }
)
```

Here, the user passes in a function that overrides the built-in optimizers provided by Exodus. This function is fairly straightforward: if the estimated cost exceeds one dollar, it returns a score of infinity, effectively eliminating that instance type and number of instances from being considered for execution. In all other cases, it simply returns the cost to execute the code, so the least expensive instance type and number of instances will still be considered.

A similar use case encountered by scientists is the desire to execute an application by a certain deadline, but still try to do so as inexpensively as possible. We can thus amend the previous example to serve that use case as follows:

```
exodus(
 :code=>"/home/user/g-protein.dfsp",
 :optimize_for=>Proc.new{|time, cost|
  if Time.now + time > DEADLINE
   return INFINITY
  else
   return cost
```

```
    end
  }
)
```

This example returns a score of infinity (eliminating this instance type and number of instances) if the projected time would cause the user to miss their deadline, and if not, we return the projected cost. Therefore, this example attempts to minimize cost while meeting the deadline. If we replaced `cost` with `time` in that example, it would instead attempt to minimize only the time spent, possibly incurring a substantially larger bill.

6.4 Evaluation

We next use our implementation of Exodus to empirically evaluate how well it optimizes program execution within cloud IaaS offerings. Because Exodus is implemented via the Neptune domain specific language, it is able to leverage the open source AppScale cloud platform to automatically configure and execute programs over IaaS resources. We begin by evaluating Exodus' ability to select resources and execute scientific and general-purpose applications, and then proceed to analyze the accuracy of Exodus' profilers.

6.4.1 Scientific Application Evaluation

We begin by using the optimizer that Exodus provides to compare the performance and cost of different IaaS offerings when executing scientific applications. We investigate two scientific applications that use a specific type of kinetic Monte Carlo algorithm (known as the Stochastic Simulation Algorithm [42]), as these applications are representative of probablistic applications that scientists execute. These applications are from the field of computational systems biology, and require a large number of independent simulations to be executed (due to their probablistic nature) to achieve an acceptable level of statistical accuracy.

The two scientific applications we focus on here are the Diffusive Finite State Projection Algorithm (DFSP) [36] and the doubly weighted SSA coupled with the cross-entropy method (dwSSA) [31]. DFSP simulates spatially inhomogeneous stochastic biochemical systems. The application that we test here employs DFSP to simulate a model of the mating pheromone induced G-protein cycle in budding yeast. The dwSSA is a method for accurate estimation of rare event probabilities in stochastic biochemical systems; the application employs dwSSA to simulate the birth-death process described in [31]. In this process, there is a rare event that can occur that doubles the population of the chemical species; accurately characterizing the rare event's probability is the goal of the application.

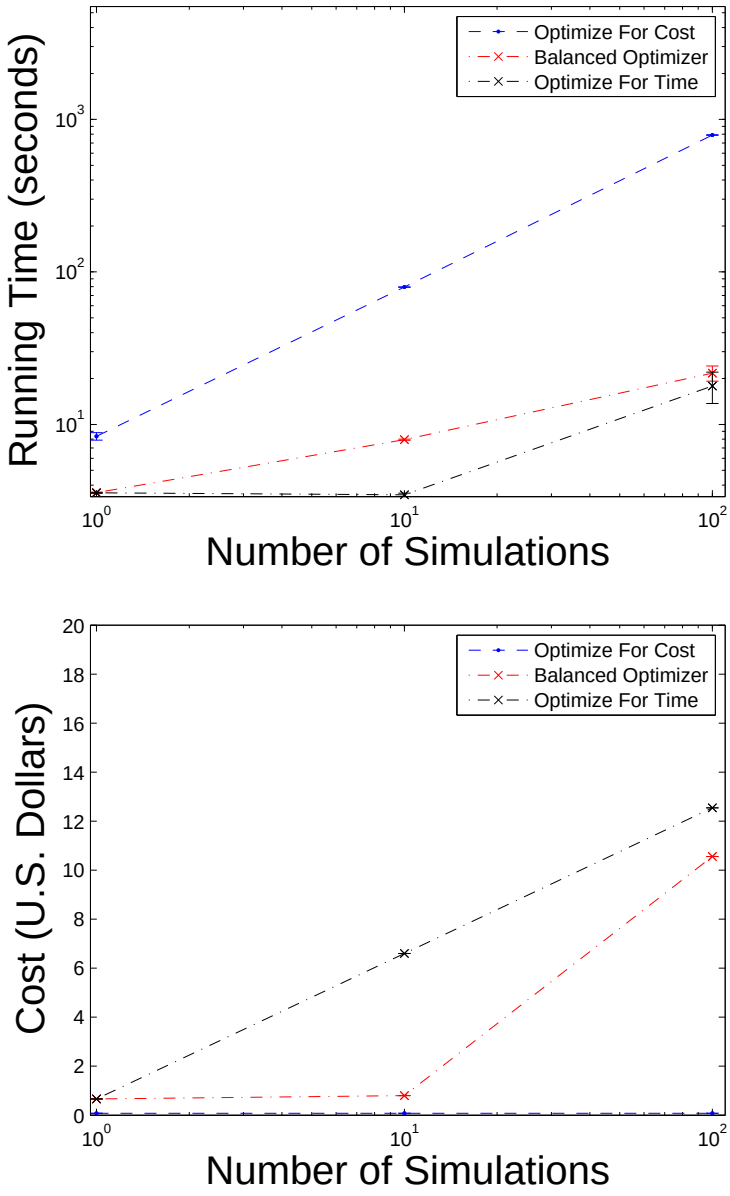

Figure 6.2: Running time (left) and cost incurred (right) for execution of DFSP simulations in Amazon EC2. We vary the optimizers used to schedule application execution between the cost-focused optimizer, the time-focused optimizer, and an optimizer that sets $\alpha = 0.5$. Each value represents the average of five runs. Note that in (left), both axes are on a logarithmic scale, and in (right), the x-axis is on a logarithmic scale.

For our first experiment, we execute 1, 10, and 100 DFSP simulations via Amazon EC2, and measure how long it takes to do so when we optimize our application to run quickly ($\alpha = 1$), run inexpensively ($\alpha = 0$), and a balance of the two ($\alpha = 0.5$). We execute each of these experiments five times and report the average and standard deviation. Here, we consider only the time spent executing tasks and omit the time spent dispatching simulations from Neptune, processing them in AppScale (hosted in EC2), and storing their results in S3.

The results of our DFSP evaluation are shown in Figure 6.2. As expected, the optimizer that focuses on fast execution performs the fastest, but at the greatest cost to the user, as it acquires 1 of the most expensive instances (`c1.xlarge` in Amazon EC2) to execute 1 simulation, 10 instances to execute 10 simulations, and 19 instances (the maximum that we can acquire from Amazon without contacting them to request more) to execute 100 simulations. Similarly, the optimizer that focuses on executing code inexpensively accrues the smallest costs, but executes code slower than the other optimizers. This is because it always acquires 1 of the least expensive instances (`m1.small`).

Figure 6.2 shows interesting behavior for our optimizer that sets $\alpha = 0.5$. With respect to total execution time it performs similarly to the performance-focused optimizer, but it is able to execute code at less cost to the user. This is because the NaïveCPUProfiler estimates total execution time to be linear with CPU speed, so it

incorrectly estimates that the fastest instance type in Amazon EC2 is proportionally faster than their slower (and cheaper) counterparts.

We next move on to evaluating our support of the birth/death model via the dwSSA. We increase the number of simulations we perform here to 10^5 per execution of our R script, and vary the number of simulations we perform between 10^5, 10^6, and 10^7. We use the same three optimizers from the DFSP evaluation, focusing on executing code quickly, inexpensively, and a balance of the two. This code requires a unique seed to be passed to it to seed its random number generator, so we amend the previous Exodus API invocation as follows to pass in this seed:

```
exodus(
  :executable=>"Rscript",
  :code=>"/home/user/run_cewssa.r",
  :argv=>Proc.new { |i| [i] },
  :optimize_for=>:cost,
  :num_tasks=>NUM_TASKS
)
```

Strictly speaking, this is not truly random (as simulations receive monotonically increasing numbers as their seeds), and we could replace i with $rand()$ to instead call the pseudo-random number generator. However, this would increase the complexity of the underlying code, as we would now have to make sure the value returned by $rand()$

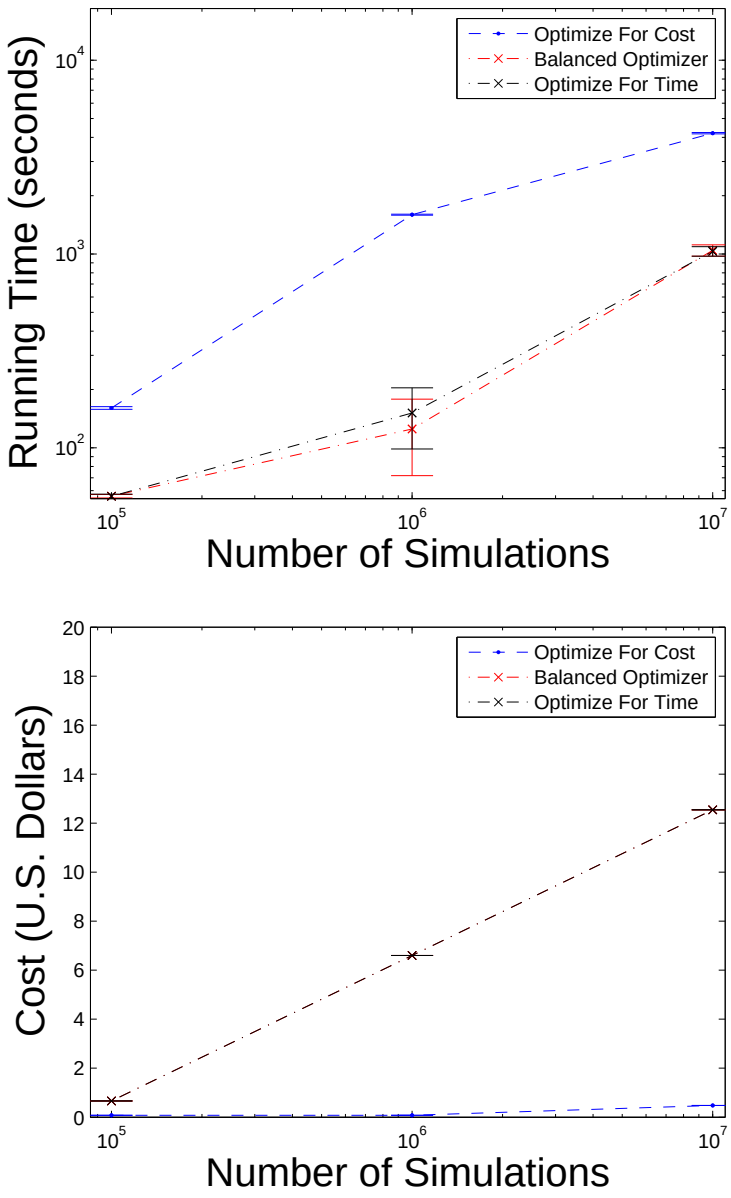

Figure 6.3: Running time (left) and cost incurred (right) for execution of dwSSA simulations in Amazon EC2. We vary the optimizers used to schedule application execution between the cost-focused optimizer, the time-focused optimizer, and an optimizer that sets $\alpha = 0.5$. Each value represents the average of five runs. Note that in (left), both axes are on logarithmic scales, and in (right), the x-axis is on a logarithmic scale.

was a seed not passed to any other simulation (as then the simulations would return the same result, and not provide unique simulation data).

The results of executing the dwSSA simulation in Amazon EC2 are shown in Figure 6.3. Here, we see that the performance-oriented optimizer and the optimizer that balances performance and cost choose the same instance type and number of instances, so they cost the same for all simulations and perform roughly similarly. We see the same expected trends as we did for the DFSP simulation in Figure 6.2: the performance-oriented optimizer executes code the fastest, but at the greatest cost to the user, and the cost-oriented optimizer executes code at the least expense to the user, but in the slowest amount of time.

The primary difference with respect to the cost-oriented optimizer is that, unlike with DFSP, the cost-oriented optimizer does not always pick 1 of the smallest instance types for all executions. This is because it projects the total execution time for the 100 programs to execute (where each program performs 10^5 simulations) at being greater than an hour to execute, and thus could use a single machine for several hours or several machines for two hours. Our tie-breaking mechanism comes into play here: whenever there are multiple instance types and number of instances that charge equally, the cost-focused optimizer chooses the option that performs the fastest.

6.4.2 General-Purpose Application Evaluation

We continue by comparing the performance and cost of the Amazon EC2 public cloud to execute general-purpose applications. We investigate two applications here, originally proposed by the seminal MapReduce paper [33], the WordCount and Grep benchmarks, written in the Python programming language. As the original Google paper indicates, these applications are representative of programs written in the MapReduce programming paradigm. For the WordCount benchmark, we pass as input the works of William Shakespeare, and allow it to count the number of occurences of each word in that input corpus.

For the Grep benchmark, we again pass as input the works of William Shakespeare, and ask the user to tell us which words they would like the benchmark to search for. In this evaluation, we only ask Grep to return lines that contain the word "Hamlet". The Exodus API invocation for this benchmark is as follows (where `gets()` is the Neptune function that queries the user for a line of input):

```
exodus(
  :executable=>"python",
  :code=>"/home/user/grep.py",
  :argv=>["/tmp/input.txt", gets()],
  :optimize_for=>:cost,
```

Simulations	Execution Time	Naïve (Error)	Remote (Error)
1	3.56 s	1.68 s (52%)	3.63 s (2%)
10	3.45 s	1.68 s (51%)	3.63 s (5%)
100	17.86 s	8.84 s (50%)	19.12 s (7%)

Table 6.1: A comparison of the time taken to execute the DFSP application in the Amazon EC2 public cloud with estimates provided by Exodus' profilers (NaïveCPUProfiler and RemoteCloudProfiler).

```
  :num_tasks=>NUM_TASKS
)
```

The results of executing the WordCount and Grep applications in Amazon EC2 are shown in Figure 6.4. Here, we see a linear relationship between the number of tasks executed and the total execution time. This is to be expected, as the cost-oriented optimizer estimates that the total execution time will be less than one hour for both the WordCount and Grep applications, and thus always picks one `m1.small` instance. Therefore, this one instance simply polls Amazon SQS for more work and executes the varying number of Map tasks in serial.

6.4.3 Error Analysis

We next move on to evaluating how effective Exodus' NaïveCPUProfiler and RemoteCloudProfiler are at correctly estimating total execution time and cost. We perform this analysis for the DFSP application with the performance-oriented optimizer, and the dwSSA application with the cost-oriented optimizer, in the Amazon EC2 public cloud.

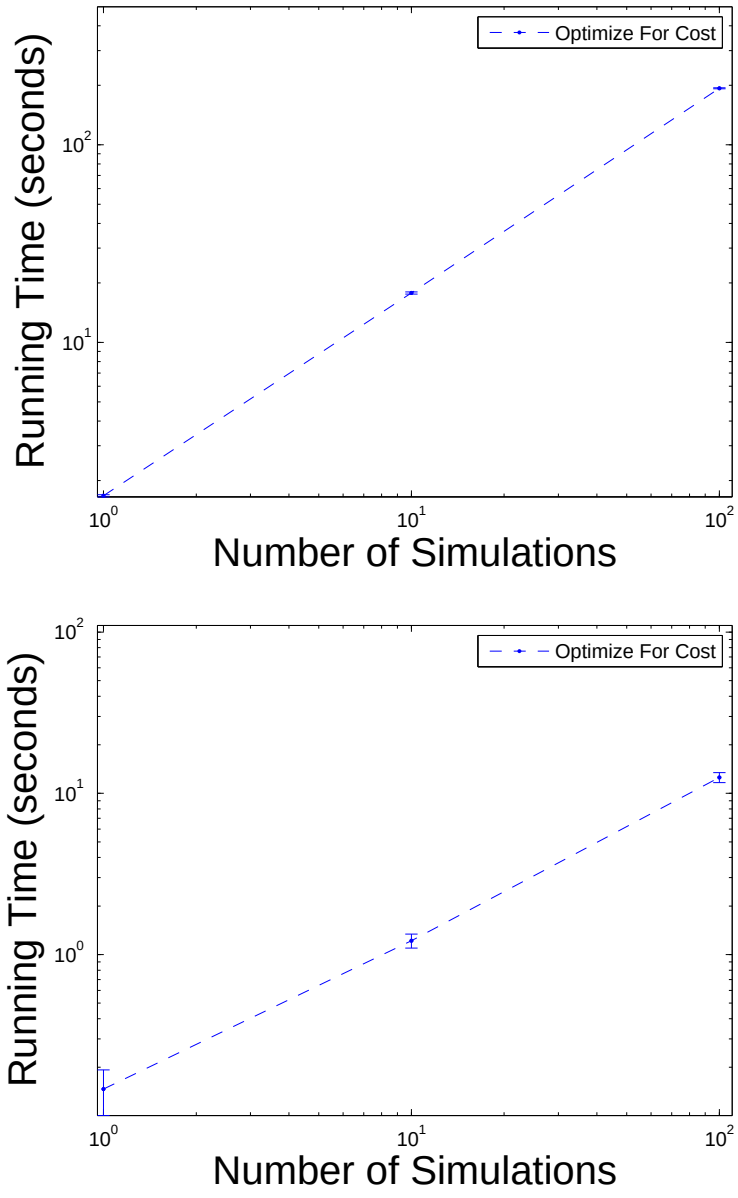

Figure 6.4: Running time for execution of WordCount (left) and Grep (right) applications in Amazon EC2. We fix the optimizer to focus on optimizing cost incurred, and vary the number of Map tasks executed. Each value represents the average of five runs. Note that both axes are on logarithmic scales.

Simulations	Execution Time	Naïve (% Error)	Remote (% Error)
1	160.45 s	101.94 s (36%)	145.50 s (9%)
10	1594.27 s	1019.44 s (36%)	1455.04 s (9%)
100	12504.72 s	10194.43 s (18%)	14550.40 s (16%)

Table 6.2: A comparison of the time taken to execute the dwSSA application in the Amazon EC2 public cloud with estimates provided by Exodus' profilers (NaïveCPUProfiler and RemoteCloudProfiler).

We begin by evaluating Exodus' profilers with the DFSP application, shown in Table 6.1. We note that, as expected, the performance-oriented optimizer always picks the fastest instance types (`c1.xlarge`), and uses as many instances as possible to execute the given simulations (1, 10, and 19 instances for 1, 10, and 100 simulations, respectively). However, we note that the profilers differ significantly in their estimates for the total execution time. The NaïveCPUProfiler has an average error of 51% when estimating DFSP execution time, but is able to do so at no cost to the user. In contrast, the RemoteCloudProfiler provides a significantly smaller error to the user (4.67%), but costs $6.71 to the user to execute (one hour of time on each instance in Amazon EC2).

We continue by evaluating Exodus' profilers with the dwsSA application, shown in Table 6.2. We again note that, as expected, the cost-oriented optimizer always picks the cheapest instance types (`m1.small`), and uses one of them to execute 10^5 and 10^6 simulations. However, due to the significant error in NaïveCPUProfiler's estimation of the total execution time (30%, on average), it incorrectly recommends to utilize three instances in the case of 10^7 simulations. It does so, believing that they will finish in

3398.15 seconds (thus charging the user for three compute-hours of time), but due to this estimation error, actually executes for 4494.96 seconds, which on three machines charges the user for two hours on each machine, or six compute-hours of time. The RemoteCloudProfiler, in contrast, provides a significantly lower estimation error (11.33%, on average) and instead utilizes one instance for four hours, charging the user a total of four compute-hours (albeit at an increased cost to the user to profile the application itself).

6.5 Related Work

The Exodus application programming interface builds upon and expands upon the work of others. Exodus was implemented by repurposing the Neptune domain specific language, which provides users with a Turing-complete language to specify how programs should be executed in clouds. The actual execution of these programs is then handled by the AppScale cloud platform. This is because its PaaS-level offering automatically manages, configures, and deploys IaaS-level resources. This work goes further with Neptune and enables it to predict the performance and cost incurred to run a user's application.

RO-BURST [94] aims to solve the problem of cost estimation in a similar vein as Exodus. However, RO-BURST differs from Exodus in three critical ways. First,

while RO-BURST claims to solve the problem of cost estimation for IaaS vendors (specifically citing Amazon EC2), their evaluation does not use any cloud services. Instead, it assigns a cost to the owner of the datacenter for every machine that they would have to purchase to execute the given workload (at $100 per server or hard drive). This style of cost estimation ignores the various instance types and pricing models popularized by Amazon EC2, which is considered when executing applications via Exodus.

Second, RO-BURST cannot perform cost estimation *a priori*, and requires intense workload characterization. Each of the twenty workloads in [94] were run for twenty days, and analyzed every five minutes to gather the 5,760 data points used to perform RO-BURST's cost estimation. In contrast, Exodus is able to estimate total execution time and performance with only a single, local execution of the user's program, and can significantly reduce the error of this estimate by utilizing cloud resources (at a minor cost to the user).

Third, it is not clear how users interact with RO-BURST. [94] does not explain how users interface their programs to it and receive cost estimates, and is not designed to be extensible with respect to how a user's code can be optimized. In contrast, Exodus provides a clear API that estimates the cost of executing a user's application, and, because it is implemented via a Turing-complete language, provides users with the ability to provide application-specific optimizers. This engenders support for a wider array of

use cases than is currently possible, and facilitates greater experimentation with optimizers on a per-application basis.

Alternatively to RO-BURST, [11] provides and evaluates a middleware offering that is able to execute MapReduce programs over local clusters and in the Amazon EC2 public cloud. It considers time deadlines and cost budgets when executing a user's MapReduce program. Three primary differences separate this work from Exodus. First, [11] only targets MapReduce programs, while Exodus can execute any application that eventually terminates, regardless of programming paradigm. Second, [11] considers only cost or execution time exclusively of one another, while Exodus is able to balance the two or enable a user to plug in their own function to optimize program execution. Third, [11] does not consider heterogeneous resource types in Amazon EC2, and limits itself to the fastest, most expensive resource type in Amazon EC2 (c1.xlarge), which is not well-suited to use cases where minimizing cost is paramount.

6.6 Summary and Conclusions

Exodus provides users with an API whose implementation automatically determines which cloud resources are optimal to use for their application, as well as how many resources to utilize. The motivation behind Exodus is to reduce the complexity associated with opaque cost models for the wide array of services offered by cloud ven-

dors. To serve this goal, Exodus enables users to define what "optimal" means to them by writing functions in a Turing-complete language, providing users with the ability to optimize execution on an application-specific basis. We evaluate how Exodus is able to effectively recommend resource usage for scientific and general-purpose applications, serving a variety of use cases. Exodus is transparently integrated with the Neptune domain specific language and automatically utilizes the AppScale cloud platform to harness IaaS-layer resources in Amazon EC2 and Eucalyptus, without user intervention. This engenders efficient use of cloud resources without requiring users to become an expert with each cloud technology they wish to consider executing their applications over, and saves users time and money when executing their applications.

The text of this chapter is, in part, a reprint of the material as it appears in [18].

Chapter 7

Conclusion

In this dissertation, we investigate how to simplify the deployment of scientific applications on cloud systems, which offer differing services, meter via competing pricing models, and provide programmatic access via vendor-specific APIs. Cloud services are seeing increased usage for a number of reasons. First, cloud services offer simple access to a number of familiar abstractions, including compute, storage, and queue services. These services are made available to the public at a previously unprecedented scale. Second, cloud services charge on a pay-per-use basis, providing users with potentially vast amounts of resources for only as long as they wish to pay for them. This allows scientists to temporarily acquire large numbers of compute nodes for their computations without needing their organization to permanently acquire and maintain them. Similarly, it provides scientists with the ability to experiment with a small number of resources, at a proportionally lower cost to their organization. Third, cloud services expose access to resources to users via first-party APIs, providing scientists with a simple

way to access these services, if their application is written in a language compatible with those APIs.

The goal of our work is to utilize the automated configuration and deployment capabilities that cloud Platform-as-a-Service (PaaS) offers to benefit scientific applications. We investigate new techniques to further this end at both the PaaS-level and at the programming language level. More specifically, we design, implement, and evaluate PaaS solutions that automatically configure and deploy applications from various application domains. These extensions are described in detail in Chapters 3–6 and provide new PaaS and programming language support for cloud-based execution of:

- **Web-enabled applications.** We design and implement a pluggable autoscaler at the PaaS layer, and thus has access to information at the entire runtime stack on each machine that hosted applications execute over. This autoscaler can therefore make scaling decisions based on Infrastructure-level information (pertaining to virtual machines), Platform-level information (pertaining to load balancers, application servers, and databases), and Software-level information (pertaining to the hosted applications themselves). We leverage our pluggable autoscaler to provide high availability, enforce Quality-of-Service requirements for hosted applications, and to do both of these in a cost-aware fashion. This cost awareness can be utilized to reduce the cost incurred to host applications, by colocating critical services on a smaller number of nodes.

- **High performance computing applications.** To facilitate greater ease of use when deploying HPC applications, we develop a hybrid PaaS/programming language solution, named Neptune. Neptune provides a programming language that scientists can write Turing-complete scripts in to deploy their applications to supported clouds, as well as PaaS-level support that automatically acquires resources, configures them, executes their applications, and releases those resources. Neptune's PaaS-level support considers the cost models of cloud services when acquiring and releasing resources, enabling resources to be used as hot spares. This reduces the amount of time needed to execute programs and amortizes the cost of resources over repeated executions.

- **General purpose applications.** We design a specialized program execution model, named MEDEA, whose PaaS-level implementation automatically incorporates compute, storage, and queue services from popular cloud vendors. This enables scientists to evaluate their applications over these services without needing to become familiar with the intricacies and best practices of each service, and facilitates portability of these applications between services. Exodus provides programming language support that predicts how to best execute a scientist's applications, based on performance, cost, or user-defined metrics. We investigate how to leverage MEDEA's PaaS-level implementation to gather this information for scientists automatically, and for a wide array of cloud services.

Finally, we provide detailed empirical evaluations of our solutions and show that they enable scientific applications to be executed in the web service, high performance computing, and general purpose application domains. In addition, the automation with respect to application configuration and deployment at the PaaS and programming language levels facilitates greater ease of use by scientists when using cloud services.

7.1 Contributions and Impact

In this section, we summarize our main contributions and discuss their impact. Our primary contribution is Platform-as-a-Service and programming language support within and across cloud compute, storage, and queue offerings, which enable automatic configuration and deployment of applications in the web service, high performance computing, and general purpose domains. Other contributions that we make in this dissertation include performance and cost awareness for hosted applications, increased portability of applications between otherwise incompatible Infrastructure-as-a-Service vendors (including on-premise offerings), and something else.

The results of our research have been appeared in the proceedings of high-quality, peer-reviewed conferences and journals. Combining the automated configuration and deployment capabilities that Platform-as-a-Service cloud computing offers with the ease of use and expressitivity of a Turing-complete programming language has never

been investigated in the literature before. In particular, our original Neptune publication was granted the Best Paper Award at HPDC's ScienceCloud based on its novelty and on the strength of its contributions.

Besides their scientific impact, our contributions have a significant practical value. The systems produced as part of our research have been released and actively maintained as open source projects since their inception. To date, the AppScale PaaS has been downloaded over 10,000 times and has an active worldwide user community. In summary, the key contributions we make with this disseration are:

- **A pluggable autoscaling system.** We contribute an open source, pluggable autoscaler that runs at the cloud PaaS layer. By realizing high availability (HA) as being part of maintaining an elastic cloud PaaS, we are able to provide an extensible autoscaling solution that adds both HA-awareness as well as QoS-awareness for web applications. We find that utilizing hot spares within our system can decrease the amount of time needed to recover from certain types of failures by an average of 48%, with a slight increase in the monetary cost that the end-user incurs. Similarly, a slight increase in monetary cost can be utilized to ensure a higher QoS to end users, with an increased performance of up to 32% for the applications tested.

  We also contribute a cost-aware autoscaler that is able to automatically save users 91% for the instances utilized in the AppScale PaaS for the HA-aware or QoS-

aware autoscalers, albeit with an order of magnitude increase in the amount of time needed to respond to failures or low QoS. We contribute all of these autoscalers to the open source AppScale code base, to engender new types of research as well as the inclusion and experimentation of existing scaling algorithms within cloud PaaS systems.

- **A domain specific language to automate program execution.** We develop and implement Neptune, a Domain Specific Language (DSL) that abstracts away the complexities of deploying and using high performance computing services within cloud platforms. We integrate support for Neptune into AppScale, an open-source cloud platform and add cloud software support for MPI, X10, MapReduce, UPC, Erlang, and the SSA packages StochKit, DFSP and dwSSA. Neptune allows users to deploy supported software packages over varying numbers of nodes with minimal effort, simply, uniformly, and scalably.

  We also contribute techniques for placement support of components within cloud platforms, while ensuring that running cloud software does not negatively impact other services. This entails hybrid cloud placement techniques, facilitating application deployment across cloud infrastructures without modification. We implement these techniques within AppScale and provide sharing support that allows users to share the results of Neptune jobs, and to publish data to the scien-

tific community. The system is flexible enough to allow users to reuse Neptune job outputs as inputs to other Neptune jobs.

- **An execution model for heterogeneous cloud execution.** We develop MEDEA, an execution model whose implementation automatically deploys user programs to cloud IaaS and PaaS systems (compute, storage, and queue services), without requiring that users modify their applications. To provide this pluggable service, MEDEA repurposes an open source cloud PaaS and domain specific language to enable arbitrary programs to be deployed and executed. Our implementation of MEDEA encapsulates such programs automatically so that they can be executed over a wide variety of cloud systems, and can execute programs on-premise or off-premise in Amazon EC2, Google App Engine, Microsoft Azure, or some combination.

  We experiment with and evaluate MEDEA's implementation using a number of different programs, programming languages, benchmarks, and use cases. We find that while cloud systems may perform similarly for a given piece of code, they can vary greatly with respect to the price users pay to run their code in these systems, due to the pricing models that clouds enforce. Overall, the MEDEA execution model significantly simplifies and makes cloud IaaS and PaaS systems portable and reusable through abstraction and a cloud PaaS system. With our

implementation of MEDEA, users can "snapshot" the performance and cost of their programs in cloud systems, and run them where it is fastest or cheapest to do so.

- **A cost and performance aware application programming interface.** We contribute Exodus, an application programming interface (API) whose implementation in the Neptune domain specific language automatically determines how to optimally execute user programs over cloud IaaS systems. Users may decide that "optimally" executing their program means that it should be executed quickly, inexpensively, or via user-provided metrics. Enabling users to provide these metrics via a Turing-complete language improves the expressitivity of the optimizer itself, and provides users with the ability to optimize execution on a application-specific basis.

  We experiment with and evaluate Exodus' implementation using a number of different scientific applications, written in different programming languages and solving non-trivial biochemical problems. Exodus is able to correctly select the optimal instance type and number of instances for its performance and cost-oriented optimizers with 2%-16% error for the applications evaluated in this work. Furthermore, we find that there is a significant, but quantifiable and predictable, difference between the number of machines and machine type to use

when executing applications quickly or inexpensively. Exodus provides the ability to estimate this quantity *a priori*, without requiring users first to become an expert with each cloud IaaS system that they wish to consider executing their application over. This increases the portability of their applications, enabling users to execute their programs wherever it is fastest or cheapest to do so.

Our contributions advance the state-of-the-art in Platform-as-a-Service cloud computing primarily by enabling scientific applications to be executed over cloud services, which provide differing services, assess fees via unique pricing structures, and supply users with access to their services by means of programmatic API support. Our contributions include techniques and implementations that have the potential to impact scientists, end users, cloud vendors, as well as researchers operating within the Infrastructure-as-a-Service and Platform-as-a-Service communities.

7.2 Future Research Directions

In this section, we identify several avenues for future research work. Our contributions described in this dissertation motivate and facilitate designing and building new systems that further advance the state-of-the-art in cloud Platform-as-a-Service, Infrastructure-as-a-Service, programming languages, and beyond. We discuss a number of research directions that we believe are worth exploring based on our empirical

results and observations as well as design and implementation intuition that we have gained while developing the systems we described in Chapters 3–6. We overview both extensions to our contributions and completely new research projects along with their potential impact.

The pluggable autoscaling system provided by AppScale can be a starting point for a number of different research paths. We identify and briefly overview the most interesting and promising ones below.

- **Pluggable autoscaling for applications outside the web services domain.** Autoscaling systems have been well investigated for web service applications, and for programs within specialized frameworks. The MapReduce programming model, and its open source implementation in Hadoop MapReduce, in particular have benefitted from research on how to serve programs that conform to its requirements. Yet an investigation has not been done to enable autoscalers from previous research systems to be interchanged (or "plugged in") or combined in a manner similar to that which is done with AppScale's pluggable autoscaling system. Such a system would foster greater research, collaboration, and use of the autoscalers themselves as scientific tools, wherein costs could be minimized or a specified Quality-of-Service could be maintained for supported, possibly general-purpose, applications.

- **A language for autoscaling.** Domain specific languages are becoming increasingly common, in response to programming languages with powerful metaprogramming capabilities. The Ruby programming language in particular provides both high writability and metaprogramming constructs that have facilitated its use for web service applications (e.g., Sinatra [87], Slim [88]), process monitoring (e.g., God [43]), and role-playing games that execute within interactive terminals (e.g., Dwemthy's Array [37] for the Interactive Ruby Shell). Yet the literature has not, to date, seen the use of a domain specific language that aids users in designing, implementing, and evaluating autoscalers for applications of any domain. APIs and rule-based systems currently are provided for autoscaling, but do not provide the flexibility or fine-grained scaling capabilities that a Turing-complete PaaS-level language solution could support.

- **Autoscaling based on service placement at the IaaS layer.** The pluggable autoscaler that this work contributes operates at the Platform-as-a-Service layer, which is abstracted away from the underlying Infrastructure-as-a-Service. This lack of cooperation between the PaaS and IaaS layers means that the PaaS could aggressively spawn virtual machines to provide high availability, but the IaaS could be oblivious of this goal and place these virtual machines on a single hardware rack. This would then defeat the purpose of providing high availability, as the system is reliant on the survival of the hardware rack itself. If the pluggable

autoscaler could instead communicate with the IaaS layer and indicate where virtual machines should be placed, then end users could be served more reliably.

We believe that leveraging domain specific languages to provide simple, powerful interactions with Platform-as-a-Service offerings (which in turn can automatically configure and deploy applications across hybrid cloud deployments) can be extended further. Below, we describe a number of potential research directions that seem worth exploring and may lead to interesting results.

- **Adaptive profiling for application execution.** Executing applications locally to predict performance encountered in cloud compute services causes errors in the predictions given by our Exodus system. One way we minimize this error is to execute the profiling runs on the actual cloud compute resources, thus acquiring a more accurate performance profile. However, this solution is valid only if instances within a given instance type perform similarly, which [10] and [62] have shown is not always the case. Therefore, future work is needed to adaptively schedule application execution among machines that are advertised as being the same by a cloud vendor but in reality perform significantly differently.

- **Cost-aware fault tolerance for application execution.** Research has been done that details how execution can be performed in a fault-tolerant manner for a number of classes of applications. Yet these works have not considered what the

monetary cost of fault-tolerance is – that is, the cost to maintain more resources as hot spares or to have speculatively executing tasks. Therefore, a system that could optimally heal from faults with respect to cost (perhaps at the expense of performance) would be a beneficial research tool for executing scientific applications.

- **Budgetting Exodus programs as a whole.** While the Exodus system does provide users with the ability to indicate that executed programs should cost no more than a certain limit, it does not provide the ability to perform this in aggregate. What this means is that scientists cannot specify their budget for all of their computations, and must individually break their budget down into chunks for each application to run. Providing users with the ability to run functions (which themselves could execute more than one application via cloud services) with a budget on all computations that it executes would alleviate this problem, and relieve scientists of the burden of breaking up their budget if more than one set of applications needs to be run.

- **Deadlines for Exodus programs.** Conceptually similar to setting a program-wide cost budget, scientists also need to be able to specify that their program should complete by a certain deadline. Research is needed to investigate how to

minimize the amount of error in Exodus predictions and thus ensure that execution plans can be produced that obey these deadlines.

In summary, this dissertation work open up several promising research opportunities in Platform-as-a-Service cloud computing. Moreover, they lay the foundation for further improvements in cloud service offerings, cost model analysis, and API interoperability, as well as automation and negotiation of the use of these services across cloud providers.

Bibliography

[1] Apache HTTP Server Benchmarking Tool. "http://httpd.apache.org/docs/2.0/programs/ab.html".

[2] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[3] Amazon EC2 Spot Instances. "http://aws.amazon.com/ec2/spot-instances".

[4] Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`.

[5] Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`.

[6] Amazon Simple Queue Service (SQS). "http://aws.amazon.com/sqs".

[7] Amazon Simple Workflow Service (SWF). "http://aws.amazon.com/swf".

[8] Amazon Web Services. `http://aws.amazon.com/`.

[9] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent Programming in ERLANG, 1993.

[10] S. K. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. *MMsys*, pages 35–46, 2010.

[11] T. Bicer, D. Chiu, and G. Agrawal. Time and cost sensitive data-intensive computing on hybrid clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 636–643, Washington, DC, USA, 2012. IEEE Computer Society.

[12] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 1–6, New York, NY, USA, 2009. ACM.

[13] Boto. "http://code.google.com/p/boto/".

[14] Building fault-tolerant applications on aws. White Paper, Oct. 2011.

[15] C. Bunch, V. Arora, N. Chohan, C. Krintz, S. Hedge, and A. Srivastava. A Pluggable Autoscaling Service for Open Cloud PaaS Systems. In *The 5th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, Nov. 2012.

[16] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura. An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. In *IEEE International Conference on Cloud Computing*, Jul. 2010.

[17] 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.

[18] C. Bunch, B. Drawert, N. Chohan, C. Krintz, and L. Petzold. Exodus: An Application Programming Interface for Cost-Aware, Cloud-Aware Program Execution. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) (in submission)*, May 2013.

[19] 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. *Journal of Grid Computing*, 10:23–46, 2012. 10.1007/s10723-012-9213-8.

[20] C. Bunch, B. Drawert, N. Chohan, A. Riofrio, C. Krintz, and L. Petzold. MEDEA: A Pluggable Middleware System for Interoperable Program Execution Across Cloud Fabrics. In *Journal of Grid Computing Special Issue: Interoperability, Federation, Frameworks and Application Programming Interfaces for IaaS Clouds (in submission)*, 2013.

[21] C. Bunch, B. Drawert, N. Chohan, A. Riofrio, C. Krintz, and L. Petzold. MEDEA: A Pluggable Middleware System for Portable Program Execution. In *27th IEEE International Parallel & Distributed Processing Symposium (in submission)*, 2013.

[22] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.

[23] Cassandra. "http://incubator.apache.org/cassandra/".

[24] Cassandra Operations. "http://wiki.apache.org/cassandra/Operations".

[25] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[26] OpsCode. "http://www.opscode.com/chef/".

[27] N. Chohan, C. Bunch, C. Krintz, and Y. Nomura. Database-Agnostic Transaction Support for Cloud Infrastructures. In *IEEE International Conference on Cloud Computing*, Jul. 2011.

[28] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *ICST International Conference on Cloud Computing*, Oct. 2009.

[29] CloudFoundry. "http://www.cloudfoundry.com/".

[30] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[31] B. J. Daigle, M. K. Roh, D. T. Gillespie, and L. R. Petzold. Automated estimation of rare event probabilities in biochemical systems. *J. Phys. Chem.*, 2011.

[32] Power Outage Affects Amazon Customers. "http://www.datacenterknowledge.com/archives/2012/06/15/power-outage-affects-amazon-customers/".

[33] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating System Design and Implementation(OSDI)*, pages 137–150, 2004.

[34] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny. Pegasus : Mapping Scientific Workflows onto the Grid. In *Across Grids Conference*, 2004.

[35] J. Dias, E. Ogasawara, D. de Oliveira, F. Porto, A. L. Coutinho, and M. Mattoso. Supporting dynamic parameter sweep in adaptive and user-steered workflow. In *Proceedings of the 6th workshop on Workflows in support of large-scale science*, WORKS '11, pages 31–36, New York, NY, USA, 2011. ACM.

[36] B. Drawert, M. J. Lawson, L. Petzold, and M. Khammash. The diffusive finite state projection algorithm for effficient simulation of the stochastic reaction-diffusion master equation. *J. Phys. Chem.*, 132(7), 2010.

[37] Dwemthy's Array. "http://mislav.uniqpath.com/poignant-guide/dwemthy/".

[38] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: a NPB experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[39] H. El-Samad, H. Kurata, J. C. Doyle, C. A. Gross, and M. Khammash. Surviving heat shock: Control strategies for robustness and performance. *Proceedings of the National Academy of Sciences of the United States of America*, 102(8):2736–2741, 2005.

[40] EnStratus. "http://enstratus.com/".

[41] Final Thoughts on the Five-Day AWS Outage. "http://www.eweek.com/c/a/Cloud-Computing/Final-Thoughts-on-the-FiveDay-AWS-Outage-236462/".

[42] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[43] God Monitoring Framework. "http://god.rubyforge.org/".

[44] Google App Engine. `http://code.google.com/appengine/`.

[45] Java App Engine Outage, July 14, 2011. "http://googleappengine.blogspot.com/2011/07/java-app-engine-outage-july-14-2011.html".

[46] Google App Engine Pipeline API. "http://code.google.com/p/appengine-pipeline/".

[47] Google Compute Engine. "http://cloud.google.com/products/compute-engine.html".

[48] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[49] Hadoop. `http://hadoop.apache.org/core/`.

[50] Hadoop Distributed File System. "http://hadoop.apache.org".

[51] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[52] HBase. "http://hadoop.apache.org/hbase/".

[53] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[54] Heroku Learns from Amazon EC2 Outage. "http://searchcloudcomputing.techtarget.com/news/1378426/Heroku-learns-from-Amazon-EC2-outage".

[55] Heroku: Widespread Application Outage. "https://status.heroku.com/incident/151".

[56] 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.

[57] Engaging the Missing Middle. "http://www.hpcinthecloud.com/features/Engaging-the-Missing-Middle-in-HPC-95750644.html".

[58] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, , the rest of the SBML Forum:, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J.-H. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. Le Novre, L. M. Loew, D. Lucio, P. Mendes, E. Minch, E. D. Mjolsness, Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[59] Kaavo. "http://www.kaavo.com/".

[60] H. Kaiser, A. Merzky, S. Hirmer, G. Allen, and E. Seidel. The SAGA C++ reference implementation: a milestone toward new high-level grid applications. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[61] 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.

[62] Y. E. Khamra, H. Kim, S. Jha, and M. Parashar. Exploring the performance fluctuations of hpc workloads on clouds. *CloudCom*, pages 383–387, 2010.

[63] G. Koslovski, T. T. Huu, J. Montagnat, and P. Vicat-Blanc. Executing distributed applications on virtualized infrastructures specified with the VXDL language and managed by the HIPerNET framework. In *ICST International Conference on Cloud Computing*, 2009.

[64] C. Krintz, C. Bunch, and N. Chohan. AppScale: Open-Source Platform-as-a-Service. Technical Report UCSB Technical Report 2011-01, Univ. of California, Santa Barbara, Jan 2011.

[65] L. Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems*, 1998.

[66] J. Li, M. Humphrey, Y.-W. Cheah, Y. Ryu, D. Agarwal, K. Jackson, and C. v. Ingen. Fault tolerance and scaling in e-science cloud applications: Observations from the continuing development of modisazure. In *Proceedings of the 2010 IEEE Sixth International Conference on e-Science*, ESCIENCE '10, 2010.

[67] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *SIAM Conference on Data Mining (SDM)*, 2012 (accepted).

[68] Lustre. "http://www.lustre.org/".

[69] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[70] Microsoft Azure Service Platform. "http://www.microsoft.com/azure/".

[71] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *IEEE International Symposium on Cluster Computing and the Grid*, 2009. `http://open.eucalyptus.com/documents/ccgrid2009.pdf`.

[72] Google I/O 2012 Keynote Transcription. "http://oakleafblog.blogspot.com/2012/07/google-io-2012-day-2-keynote-by-urs.html".

[73] P.-O. östberg, A. Hellander, B. Drawert, E. Elmroth, S. Holmgren, and L. Petzold. Abstractions for scaling escience applications to distributed computing environments; a stratum integration case study in molecular systems biology. *Proceedings of BIOINFORMATICS 2012, International Conference on Bioinformatics Models, Methods, and Algorithms*, pages 290–294, 2012.

[74] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, New York, NY, USA, 2011. ACM.

[75] Pbspro home page – `http://www.altair.com/software/pbspro.htm`.

[76] IT Automation Software for System Administrators. "http://puppetlabs.com/".

[77] L. Ramakrishnan, C. Koelbel, Y. suk Kee, R. Wolski, D. Nurmi, D. Gannon, G. Obertelli, A. Yarkhan, A. Mandal, T. M. Huang, K. Thyagaraja, and D. Zagorodnov. Vgrads: Enabling e-science workflows on grids and clouds with fault tolerance.

[78] RightScale. "http://www.rightscale.com/".

[79] RightScale. RightScale Gems. "http://rightaws.rubyforge.org/".

[80] T. J. Rolfe. A Specimen MPI Application: N-Queens in Parallel. *inroads (bulletin of the ACM SIG on Computer Science Education)*, 40(4), 2008.

[81] Ruby language. "http://www.ruby-lang.org".

[82] Ruby on Rails. "http://www.rubyonrails.org".

[83] K. R. Sanft, S. Wu, M. Roh, J. Fu, R. K. Lim, and L. R. Petzold. StochKit2: Software for Discrete Stochastic Simulation of Biochemical Systems with Events. *Bioinformatics*, 2011.

[84] Scalr. "http://scalr.net/".

[85] K. Shen, H. Tang, and T. Yang. Adaptive two-level thread Management for fast MPI execution on shared memory machines. In *Proceedings of ACM/IEEE SuperComputing '99*, 1999.

[86] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[87] Sinatra. "http://www.sinatrarb.com/".

[88] A Fast, Lightweight Template Engine for Ruby. "http://slim-lang.com/".

[89] StochKit. "http://www.cs.ucsb.edu/ cse/StochKit/".

[90] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobbs Journal*, February 1995.

[91] The Computer Language Benchmarks Game. "http://shootout.alioth.debian.org/".

[92] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1), Mar. 2008.

[93] W. E. Nagel and A. Arnold and M. Weber and H.-Ch. Hoppe and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, 1996.

[94] J. Wang, R. Hua, Y. Zhu, J. Wan, C. Xie, and Y. Chen. Ro-burst: A robust virtualization cost model for workload consolidation over clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 490–497, Washington, DC, USA, 2012. IEEE Computer Society.

[95] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15:757–768, 1999.

[96] YAML. "http://yaml.org/".

[97] L. Zhang, C. Krintz, and P. Nagpurkar. Language and virtual machine support for efficient fine-grained futures in java. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 130–139, Washington, DC, USA, 2007. IEEE Computer Society.

[98] L. Zhang, C. Krintz, and P. Nagpurkar. Supporting exception handling for futures in java. In *Proceedings of the 5th international symposium on Principles and*

practice of programming in Java, PPPJ '07, pages 175–184, New York, NY, USA, 2007. ACM.

[99] Z. Zhang, D. S. Katz, M. Ripeanu, M. Wilde, and I. T. Foster. Ame: an anyscale many-task computing engine. In *Proceedings of the 6th workshop on Workflows in support of large-scale science*, WORKS '11, pages 137–146, New York, NY, USA, 2011. ACM.