2012

Abstraction Driven Application and Data Portability in Cloud Computing

Ajith Harshana Ranabahu
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/779

This Dissertation is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu, library-corescholar@wright.edu.
Abstraction Driven Application and Data Portability in Cloud Computing

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

By

AJITH H. RANABAHU
B.Sc., University of Moratuwa, 2003

2012
Wright State University
WRIGHT STATE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

December 31, 2012


__________________________________________
Amit P. Sheth, Ph.D.
Dissertation Director

__________________________________________
Arthur A. Goshtasby, Ph.D.
Director, Computer Science Ph.D. Program

__________________________________________
Andrew T. Hsu, Ph.D.
Dean, School of Graduate Studies

Committee on Final Examination

__________________________________________
Amit P. Sheth, Ph.D.

__________________________________________
Krishnaprasad Thirunarayan, Ph.D.

__________________________________________
Keke Chen, Ph.D.

__________________________________________
E. Michael Maximilien, Ph.D.
ABSTRACT


Cloud computing has changed the way organizations create, manage, and evolve their applications. While many organizations are eager to use the cloud, tempted by substantial cost savings and convenience, the implications of using clouds are still not well understood. One of the major concerns in cloud adoption is the vendor lock-in of applications, caused by the heterogeneity of the numerous cloud service offerings. Vendor locked applications are difficult, if not impossible to port from one cloud system to another, forcing cloud service consumers to use undesired or suboptimal solutions.

This dissertation investigates a complete and comprehensive solution to address the issue of application lock-in in cloud computing. The primary philosophy is the use of carefully defined abstractions in a manner that makes the heterogeneity in the clouds invisible.

The first part of this dissertation focuses on the development of cloud applications using abstract specifications. Given the domain specific nature of many cloud workloads, we focused on using Domain Specific Languages (DSLs). We applied DSL based development techniques to two domains with different characteristics and learnt that abstract driven methods are indeed viable and results in significant savings in cost and effort. We also showcase two publicly hosted Web-based application developments tools, pertaining to the two domains. These tools use abstractions in every step of the application life-cycle and allow domain experts to conveniently create applications and deploy them to clouds, irrespective of the target cloud system.

The second part of this dissertation presents the use of process abstractions for application deployment and management in clouds. Many cloud service consumers are focused on specific application oriented tasks, thus we provided abstractions for the most useful cloud interactions via a middleware layer. Our middleware system not only provided the independence from the various process differences, but also provided the means to reuse known best practices. The success of this middleware system also influenced a commercial product.
# Contents

1 Introduction 1
   1.1 Significance ................................................. 6
   1.2 Scope ...................................................... 7
      1.2.1 Justification on Scope ............................. 10
   1.3 Dissertation Contributions ............................. 13
   1.4 Dissertation Organization ............................. 14

2 Cloud Computing: Challenges and Opportunities 15
   2.1 What is Cloud computing? ............................... 15
   2.2 Benefits of Cloud Computing ........................... 17
   2.3 Challenges in Cloud Computing ....................... 19
   2.4 Cloud Growth Forecast ................................. 22
   2.5 Chapter Summary ......................................... 23

3 Using Abstractions as a Solution Strategy 24
   3.1 A Typical Application Life-cycle ...................... 24
   3.2 What Activities should be Abstracted? ................ 26
   3.3 A Two Pronged Strategy to Apply Abstractions ......... 27
   3.4 Prior Work: Other Abstract Based Approaches ....... 28
      3.4.1 Code Generation Systems for Service Oriented Architectures 28
      3.4.2 Code Generation Systems for Applications .......... 29
      3.4.3 The Philosophy of Software Factories .............. 30
   3.5 Abstractions in Cloud System Interactions .......... 31
      3.5.1 Cloud Brokers ...................................... 31
      3.5.2 Unified Cloud Programming Interfaces ............ 32
   3.6 An Augmented Application Life-Cycle .................. 33
   3.7 Achieving Portability .................................... 34
   3.8 Chapter Summary ......................................... 35

4 Platform Agnostic Cloud Application Development 36
   4.1 Preliminaries ............................................. 36
      4.1.1 A Background on Domain Specific Language ...... 36
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of cloud types, providers and user responsibilities</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>Cisco Cloud report</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>The typical top-down cloud application life-cycle</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>The augmented cloud application life-cycle</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Standard metamodel for Mathematical expressions</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>The modeling hierarchy</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Four types of semantics with respect to a Numerical data processing application</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Enhanced Metamodel for Mathematical expressions</td>
<td>56</td>
</tr>
<tr>
<td>4.5</td>
<td>Metamodels, models and transformations</td>
<td>57</td>
</tr>
<tr>
<td>4.6</td>
<td>An example map-reduce task graph</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Model-View-Controller design pattern</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Generated components for the Task Manager application</td>
<td>72</td>
</tr>
<tr>
<td>5.3</td>
<td>The high level architecture of the MobiCloud system</td>
<td>73</td>
</tr>
<tr>
<td>5.4</td>
<td>Augmented high level architecture for extensions</td>
<td>76</td>
</tr>
<tr>
<td>5.5</td>
<td>Various views of the composer.</td>
<td>80</td>
</tr>
<tr>
<td>5.6</td>
<td>The generated application, running on an Android device</td>
<td>81</td>
</tr>
<tr>
<td>5.7</td>
<td>Architecture of the SCALE Implementation</td>
<td>82</td>
</tr>
<tr>
<td>5.8</td>
<td>Overview of the SCALE composer</td>
<td>87</td>
</tr>
<tr>
<td>5.9</td>
<td>A typical SCALE composition</td>
<td>91</td>
</tr>
<tr>
<td>5.10</td>
<td>Lines of Code statistics of MobiCloud generated applications</td>
<td>92</td>
</tr>
<tr>
<td>5.11</td>
<td>Lines of Code statistics of MobiCloud generation templates</td>
<td>95</td>
</tr>
<tr>
<td>5.12</td>
<td>Lines of Code statistics of SCALE generated applications</td>
<td>96</td>
</tr>
<tr>
<td>5.13</td>
<td>LOC statistics of SCALE Generators</td>
<td>97</td>
</tr>
<tr>
<td>6.1</td>
<td>The placement of cloud middleware</td>
<td>107</td>
</tr>
<tr>
<td>6.2</td>
<td>Altocumulus Architecture</td>
<td>111</td>
</tr>
<tr>
<td>6.3</td>
<td>Altocumulus Dashboard</td>
<td>112</td>
</tr>
<tr>
<td>6.4</td>
<td>Structure of a Best Practice</td>
<td>118</td>
</tr>
<tr>
<td>6.5</td>
<td>Illustration of Different Best Practices</td>
<td>121</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Features and capabilities of the MobiCloud toolkit ........................................ 79
5.2 Selected code metrics of MobiCloud applications ............................................. 94
5.3 Template and Resource LOC counts for MobiCloud ......................................... 96
5.4 Selected code counts for SCALE operators ....................................................... 97
5.5 Template and Resource LOC counts for SCALE ............................................... 98
5.6 Selected Running times of SCALE applications ............................................... 99
Acknowledgement

This dissertation is the result of years of hard work as well as the contributions of many others, to whom I am greatly in debt.

Foremost, I would like to thank my advisor, Dr. Amit Sheth for his guidance and advice, both in technical matters as well as in life. He has been instrumental in opening many doors for me and without his efforts and guidance, I would not have reached this milestone. He was the primary driver of my research. We’ve had many enlightening discussions, most of them technical, yet some of the non-technical ones played important roles in my life decisions.

Next, I would thank Dr. Michael ’Max’ Maximilien, mentor and friend. Working with Max has been the key differentiator in my research life, providing the basis for this research work. The two internships with Max at IBM Research and our subsequent collaborations have shaped my research extensively. Max literally took me under his wing and treated me as a colleague rather than an intern. It is his influence that showed me the many important industry problems that I could concentrate on.

Dr. Krishnaprasad Thirunarayan, committee member and close collaborator, has also been a key driver of my research and deserves special gratitude. During our collaborations, Dr. Prasad read all my papers from top to bottom, giving critical comments on every aspect of them. His vast knowledge in languages has been the primary reason that this dissertation is at its current elegant state.
I would also like to thank Dr. Keke Chen, committee member and collaborator. Dr. Chen has been the critical expert that helped me organize some of my early work.

Dr. Paul Anderson, with whom I’ve had a very successful collaboration, deserves special gratitude for persuading me to consider life sciences as an application area. This ultimately resulted in the SCALE project, and contributed to a chain of relevant publications.

Dr. Karthik Gomadam, my early room-mate and close collaborator, helped tremendously to shape my research life. Our collaborations led to a number of well regarded publications as well as a life-long friendship. I would also like to extend my gratitude to other close friends and colleagues, Dr. Meenakshi Nagarajan, Dr. Cartic Ramakrishnan, Christopher Thomas, Pablo Mendes and Dr. Ramakanth Kavuluru for their valuable advice and comments during the numerous discussions and impromptu gatherings we had over the years.

I’m also grateful to Dr. Sanjiva Weerawarana for inspiring me to follow higher education. Without his influence, I would not have followed a doctoral degree in the first place. Dr. Weerawarana not only introduced me to Dr. Sheth but also discussed with me, at length, the numerous social and personal issues that may arise during a long and arduous graduate student life. Without his keen insights, I would not have understood the importance of a doctoral degree and numerous opportunities it would open.

A very special note of gratitude goes to my dear wife Dharshani Herath and baby daughter Sanuthi Lisara Ranabahu. Without Dharshani’s love and care, this dissertation would never have been completed. Sanuthi, even before seeing the light of this world, has been the inspiration of my life.
My dear friend and mentee Ashwin Manjunatha also deserves a note of gratitude for the initial work in getting some of the projects off the ground. Our collaborations yielded many important tools and publications, many of them contributing to this dissertation as well.

Last but not least, I would like to thank my friends and fellow lab-mates at Kno.e.sis: Delroy Cameron, Prateek Jain, Cory Henson, Ashutosh Jadhav, Harshal Patni, Hemant Purohit, Kalpa Gunaratna, Pavan Kapanipathi, Pramod Anantharam, Raghava Mutharaju, Sarasi Lalithsena, Sujan Perera, Vinh Nguyen, Wenbo Wang, Lu Chen, Sanjaya Wijeratne and Lakshika Balasuriya for the stimulating discussions and for the good times we have had working together in the last six years. I’ve thoroughly enjoyed your friendship and the time we spent on and off Kno.e.sis.
Dedicated to my parents Mutubanda and Nandawathie for their unconditional support to follow my dreams.
Introduction

Cloud computing is one of the most notable evolutions in computing. Availability of seemingly unlimited, readily provisioned, pay per use computing resources has not only spawned a number of new industries but has also changed the mindset of many information technology (IT) centric organizations. Large institutions now offload their overflow computing requirements to clouds while technology startups use them to establish their IT infrastructure without a heavy up front capital expenditure.

The mass adoption of clouds however, does not imply that all the challenges in using computing clouds are well understood. Clouds offer access to cheap, abundant computing resources, but the most appropriate utilization of these resources is still limited by the unavailability of relevant software stacks. For example, infrastructure as a service (IaaS) clouds offer the ability to quickly and programmatically provision computing instances but it is up to the user programs to make use of this capability, say for dynamic load balancing using replicated computing instances. In this case, it is the responsibility of the cloud service consumers to manage the infrastructure as they see fit.

The difficulty of utilizing a pure infrastructure based cloud solution has prompted some cloud service providers to offer platform services, referred to as platform as a service (PaaS). In a cloud
platform, the difficulties in scaling and load management are transparent to certain types of user programs, e.g., Web applications. User programs merely adhere to a set of predefined software frameworks and the platform takes care of the otherwise mundane tasks such as load balancing. These platforms however, only support a limited set of application types. For example, Google App Engine (GAE)[1], one of the leading cloud platform service providers, only supports Java and Python Web applications and mandates the use of Google specific data storage solutions for persistence. A similar offering by Amazon, the Amazon Elastic Beanstalk[2] is more flexible but supports only three Web development software stacks as of June 2012.

With the availability of cheap cloud infrastructure, a third type of service offering has also become popular. Known as software as a service (SaaS), this service offering covers cloud based, multi-tenant software stacks. These are typically services that were considered back-office, such as email and document management. SaaS has only limited customizations and there is no concept of user-defined applications.

Vendors offering such differentiated cloud services with unique features has caused the current cloud computing landscape to include a large number of heterogeneous service offerings. These service offerings include infrastructure oriented services (providing basic computational resources such as virtual machines), cloud computing platform services (specialized software development platforms) and multi-tenant cloud based software services (customizable document management and email services). The downside of these differences is that, when user-defined applications are considered, they result in application architectures dictated by service provider specific features, ultimately resulting in non-portable, vendor-locked applications.

To illustrate this with an example, consider the case of a one developer startup, an entity with
one application and a single developer. Many such startups prefer using GAE for the cost benefits\(^1\) and easy programming. However, GAE used to force the use of Google BigTable [3] and thus, the data representation and manipulation inside the applications were strongly tied to BigTable. When the application outgrows the capabilities of GAE, migrating to a more scalable and reliable infrastructure cloud service is desirable. However, such a migration would require extensive reprogramming, primarily on the data access and manipulation components. These concerns are in fact voiced by a number of developers publicly, reflected in public forums as well as market research reports[4].

A number of publicly recorded incidents provide evidence to the seriousness of vendor-lock-in of applications. Three such incidents are listed below.

1. The Amazon Elastic Compute Cloud (EC2) became unavailable on April 21st, 2011 for about 12 hours due to a network misconfiguration [5]. Many popular startups, including Foursquare, Reddit, and Quora, were unable to function during this period. None of these services were able to restore their functionality until EC2 was fixed as a consequence of the tight coupling of the software back-ends to EC2.

2. Amazon EC2 was partially crippled due to heavy thunderstorm knocking out power to a data center on June 29th 2012[6]. Backup power systems failed, effecting a large group of consumers including Netflix and Pinterest. The service disruption lasted about 3 hours.

3. The Microsoft Azure cloud became unavailable for about 3 hours on February 28th, 2012 due to a leap year (February 29th) time calculation bug in the Azure platform software [7].

\(^1\)GAE provides about 5 million page views per month at no cost, as per a non-authoritative source http://bit.ly/Kyr1WI (Last retrieved June 2012)
All Microsoft cloud services were not restored until a fix was deployed. Microsoft later issued service credits for all the customers affected by this outage.

All the cloud applications in the above scenarios were locked into their host clouds, either by software platform restrictions (Azure) and/or use of cloud specific features (EC2). These scenarios indeed highlight the importance of the portability of an application in a cloud context.

Taking a more pragmatic view, one can see a *difference in perspective* where some of the major issues stem from. There are two primary parties who can be considered stakeholders in a cloud software environment.

1. **Cloud service consumers**: Consists of parties who use cloud resources to run applications. They primarily take an *application oriented perspective*, i.e. the focus is on the application. As long as the expected functionality of the application is available, the underlying software stack or the actual service provisioning mechanism is of lesser importance. For example, many small to medium businesses care about having a Web presence, yet are not particular about the software stack being used (Say .NET, Java or Python) or the actual service hosting mechanism (Say cloud or dedicated server) as long as the Web presence goals are met.

2. **Cloud service providers**: Parties that provide cloud services. They typically take a *utilization perspective*, i.e. the focus is on maximal utilization of resources. This is evident in service provisioning and resource management mechanisms of many clouds. For example, Amazon EC2 allows fine grained control over the virtual machine import and export. This feature is useful to optimize the backup and restore functions, yet many users would not use nor care about these features in a simple application deployment.
This difference in perspective ultimately results in an *impedance mismatch*, a difference in what is needed and what is provided. Cloud service consumers are required to take into account internal details of the cloud software environment, tightly coupling their application design and development process to a selected cloud environment. This is indeed an important issue to address. Cloud computing is projected to reach over $60 billion market capitalization in 2014 [4]. It is also predicted that over 50% of the total computational workloads will be on computing clouds by that time [8].

This dissertation takes an application oriented perspective and focuses on cloud applications. The systematic approach described here is based on two ideas.

1. Use well defined programming abstractions to specify the application behavior rather than describe its implementation. The specifications are sufficiently high level such that cloud-specific features are not directly exposed, thus the consumers are *not* locked into a vendor or a particular software environment.

2. Use appropriate process abstractions to allow high level operations with clouds rather than micro-operations. These process abstractions provide high-level interaction patterns, leaving cloud-specific operations transparent.

The main goal of this dissertation is to construct and evaluate a methodology to develop, deploy and manage cloud applications in a *cloud environment agnostic fashion*. 

5
1.1 Significance

The ability to develop cloud applications easily and port them across different cloud software environments with little or no effort offers significant advantages over the current state-of-the-art methods. The following factors are the most significant, within the scope covered in this dissertation (see Section 1.2).

1. **Usability and Adoption**: The ability to develop a cloud application taking a domain centric perspective, has profound effects on the usability of the cloud. The cloud, as of now, is an option for the highly technical consumers, i.e. developers. Typical, non-technical, consumers do not go beyond cloud based software services (SaaS), yet there is large interest in specific domains to use the cloud. For example, in eScience (scientific experimentation with extremely large datasets), there is significant interest in using clouds to lower the processing time and/or cost. The majority of the scientists working in core science domains however, are not capable of composing their own distributed programs to make the best use of cloud resources. Giving these scientists the ability to compose programs in their domain terms has a significant impact on cloud adoption.

2. **Economic Advantage**: The ability to deploy a compatible implementation in a target cloud, without reprogramming has an impact economically. The reduction in the development effort (since only one version has to be maintained) results in significant cost savings, both in the initial development as well as during maintenance. Other cost saving side-effects include the ability to federate applications, i.e., deploy functionally equivalent versions of the same application across different environments to mitigate failures.
While there may be other derived benefits, these two factors can be treated as the most significant umbrella advantages in avoiding lock-in.

1.2 Scope

Given that the term cloud and cloud application are overloaded and broad (the subsequent chapters provide concrete definitions for these terms), it is important to explicitly state the scope of this dissertation.

The following considerations are in scope of this dissertation.

1. Development of monolithic, domain specific applications: The term monolithic means the ability of the application components to be packaged to a single bundle. Note that this does not mean strictly self-contained. For example, the generated application may contain procedures to connect to an external database (rather than containing an inbuilt database, a feature of a self-contained application) and the necessary database scripts. A deployment process aware of these scripts could execute them to create the initial database prior to deploying the application, when the database is available as a service (or trivial to install).

The term domain specific refers to the anticipated function of the application. The applications considered are to solve problems that are restricted to specific domains. For example, the research discussed in this dissertation includes computational biology applications that are only useful in that domain.

2. Deployment and management of bundled applications in heterogeneous cloud environments.

Deployment refers to setting up the application in the cloud environment. For some cloud
environments, the deployment may be merely executing a command but for others, deployment may include a lengthy set-up procedure. *Management* refers to maintenance tasks that are typically done after the application is deployed, for example taking data backups.

3. *Migration of bundled applications between heterogeneous cloud environments.* This type of migration takes into account at least two aspects.

(a) *Code translation:* Each cloud environment has different capabilities regarding the type of application that can be deployed. In an infrastructure cloud, applications written in a large variety of programming languages may be deployed, given the proper support environment (interpreters, compilers etc.) can be successfully configured. On the other hand, a platform cloud environment typically limits the usable programming language to a handful. When migrating across such clouds, a code translation to a supported programming language may be required.

A related issue in the same context is language feature suppression. Some cloud environments limit the use of certain programming language features and impose conditions on using environment specific libraries. For example, Google App Engine (GAE) supports the Java programming language, except certain features. One of the notable ones is Java native interface (JNI) that allows a Java program to reach down to the native platform, bypassing the virtual machine. Similarly, GAE mandates the use of GAE specific libraries for certain common tasks, such as spawning a new thread. The code translation also needs to consider such environment specific feature suppressions and restrictions.

(b) *Data transformation:* Each cloud environment supports (or mandates) different data
storage options. When migrating an application to a different cloud environment, migrating the data becomes an important issue, often when the application has accumulated useful data. In many domains, the accumulated data is considered as the most valuable asset. Data migration is based on the following two considerations.

i. Transforming the data schema.

ii. Transforming the data instances.

Data schema is the structure and the organization of the data values and instances are the data values.

The following considerations are not in the scope of this dissertation.

1. Development of complex multi-tier applications that depend on a number of (possibly distributed) support components to function.

An abstraction based generation mechanism is capable of supporting a simple multi-tier application (consisting of two or three layers). However, any application that consists of more than three (possibly loosely coupled) components/layers is considered too complex for our purposes. Note that this does not imply it is impossible to generate such an application using an abstract specification. Simply, the details needed in the abstract specification becomes significantly larger, making it complex, thus defeats the purpose of introducing the abstractions in the first place.

For example, an abstraction based mechanism can generate a typical three tier Web application, consisting of a database, an application server and a UI. This set up can easily be migrated across different cloud environments, replacing each tier with an environment-
supported configuration. However, there may be other components/tiers that may be required in a more rigorous, industry grade configuration. For example, many real world applications use independent authentication services, either as an internal component of the system (such as the Lightweight Directory Access Protocol[9]) or as a completely external service (such as Open Authentication[10]). In either case, this would be a difficult component to migrate across cloud environment given the number of dependencies and system requirements of such components.

2. Reverse engineering of existing cloud applications. Reverse engineering refers to taking an arbitrary cloud application and constructing an abstracted form, such that the original functionality is replicated when the abstract form is transformed. Since this dissertation addresses domain specific applications, it is impossible to reverse engineer an arbitrary cloud application. This is discussed in detail in Section 4.2.4.4.

This dissertation does not cover any form of reverse engineering.

1.2.1 Justification on Scope

For large software projects with a significant number of components, it is extremely difficult to port the entire software stack from one platform to another. These projects embrace the fact that lock-in cannot be avoided and failures are unavoidable, thus have procedures in place to compensate for them. Clearly this type of project requires significant funding and effort and are generally managed by large, well funded organizations that have the resources and the leverage to cater for such additions.
On the other hand, there are less complex cloud applications developed and maintained by small development teams. These applications typically solve a domain specific problem. This type of application is more likely to face portability issues, given the lack of resources and expertise in cloud based software engineering. The solutions presented in this dissertation are more suited towards such domain specific problems.

An excellent example for this can be found in the biology domain, as discussed briefly earlier. A significant amount of computer based statistical processing is required to get the results from many modern biological experiments that generate large numerical datasets. This branch of biology is called computational biology for its requirement of heavy computations.

There are three facts that stand out about computational biology.

1. *Computational biology tasks requires a significant amount of computations to make useful interpretations*: Computational biologists typically collect and process large amounts of numerical data. For example, Nuclear magnetic resonance (NMR) based computational biology experiments generate numerical data sets typically in the range of few megabytes per sample. Given the variations of radiation and other parameters, one real world experiment may contain data in the order of gigabytes. Running conventional computations (using a single computer) on this type of data and making useful interpretations sometimes takes several days [11].

2. *Biologists prefer specialized and domain specific tools, even if they are slow*: Many biological computational processes use tools built using platforms like MATLAB. These tools are easy to use and program, yet are slow and unable to directly take advantage of the cheaply
available computational power. Domain scientists are hesitant to migrate computational processes to external sources such as clouds, primarily due to the overhead of such a move and the possible inconvenience in using them.

3. Many computational biology research teams are not very large: Except for a handful of large organizations, many computational biology research teams (especially in academia) are not very large, with reference to other scientific research groups. Thus, these teams focus on the most effective utilization of their effort.

Computational biology is an ideal domain to apply the abstraction based methods. On one hand, the required computations are heavy and they can benefit from using the cloud to speed up the data processing. On the other hand, the domain experts are not highly experienced in computing but are comfortable using domain specific tools. An abstract based solution applies well to this kind of scenario.

An example to the contrary is Netflix, an online video streaming company. An organization like Netflix also benefits from using the cloud but require their systems to be highly available. Netflix famously switched to Amazon EC2 from their own data centers [12] and rebuilt their entire system to use the Amazon infrastructure cloud platform. Netflix has invested heavily in the cloud migration and also built enough safeguards to cater to failures. Netflix was able to survive some high profile failures of the Amazon cloud [13], providing evidence that it is possible to cater to a failure even when the software stack is vendor-locked. This however came at a significant cost and is not affordable for smaller groups or startups. There is also the high complexity associated with this type of software environment that requires significant human effort.
1.3 Dissertation Contributions

This dissertation makes the following contributions.

1. Introduce a general strategy to develop, deploy, manage and migrate cloud applications using well-designed abstractions. These abstractions take two forms.

   (a) **Domain Specific Languages (DSLs):** A DSL is a restricted language, often used to specify specifications in a highly restricted context.

   (b) **Cloud Middleware:** Middleware refers to a broad range of software stacks that provide abstractions over operational aspects of different (heterogeneous) application programming interfaces (APIs).

2. Investigate the theoretical and practical details of using DSLs for cloud application development. We focus only on two salient aspects that are considered to be the building blocks of cloud program portability.

   (a) **Functional abstractions:** Functional abstractions provide high level specifications of the core business logic of a program that governs part of the applications behavior.

   (b) **Data abstractions:** Data abstractions refer to the high level specifications of the core data structures and the data elements to be persisted.

We evaluate this approach using two practical projects.

(a) **MobiCloud:** A cloud-mobile hybrid application generator.

(b) **SCALE:** A scientific data processing application generator.
3. Investigate the theoretical and practical details of using middleware abstractions for cloud application deployment and management. We present and evaluate the Altocumulus cloud middleware system. Altocumulus is an experimental cloud middleware system that supported basic application related operations over 3 cloud environments.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. In a nutshell, except for Chapter 2, other chapters discuss the above mentioned contributions systematically. Chapter 4 and Chapter 5 discuss the theoretical and practical aspects of the application development strategy and separated for clarity.

Chapter 2 presents a short background on cloud computing and discuss the challenges and opportunities.

Chapter 3 presents the core philosophical ideas behind this dissertation and establishes a high-level overview.

Chapter 4 discusses the use of DSLs in application generation in detail.

Chapter 5 presents the two projects, MobiCloud and SCALE, that use the DSL based techniques.

Chapter 6 discusses the applicability of cloud middleware and the Altocumulus middleware system.

Chapter 7 presents the conclusion of the dissertation.
Cloud Computing : Challenges and Opportunities

This chapter briefly introduces cloud computing and discusses the benefits, challenges and opportunities in cloud computing.

2.1 What is Cloud computing?

Cloud computing (or just cloud) is a term originally coined by network professionals to refer to the unknown complex web of connected computers that lie beyond an organization. However, the terms cloud and cloud computing are now used to refer to many different systems, often in completely different contexts. For example, the Apple iCloud [14] is a multi tenant remote storage system that can be coupled to a number of Apple devices. The focus of iCloud is on storage, thus many non-technical device users associate the term cloud with iCloud-like storage systems. On the other hand, Amazon Elastic Compute Cloud (EC2) uses the term cloud in several completely different senses, to refer to computation units as well as scalable storage.

To avoid any misrepresentations, we follow the definition from the National Institute of Stan-
2.1. WHAT IS CLOUD COMPUTING?

National Institute of Standards and Technology (NIST) [15] as the governing description of cloud computing.

“cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”

This definition encapsulates the various aspects of cloud computing well and also highlights four important characteristics.

1. **Shared pool of computing resources**: Cloud computing is inherently multi-tenant and provides resources, may it be servers, storage or applications, from a shared pool. This provides the *economy of scale*, thus the resources are relatively inexpensive.

2. **On-demand network access**: The resources can be accessed over the Internet, often with relative anonymity of where they are located.

3. **Rapid provisioning**: Provisioning or initiation of resources can be done quickly. This is primarily supported by resource virtualization.

4. **Minimal management effort and service provider interaction**: One of the major characteristics of cloud computing is this minimal interaction. Most cloud systems offer service interfaces, thus one can even manage computing resources programmatically.

NIST also identifies three types of clouds based on the service model; Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) that has been mentioned before. Figure 2.1 shows what we consider to be the user (i.e. cloud application developer) responsibility for each of these cloud types with a representative list of providers.
2.2 Benefits of Cloud Computing

A number of publications discuss the advantages of cloud computing. These advantages can be broadly consolidated under two umbrellas.

1. **Cost Advantage**: Cost saving, either in terms of hardware or man power seems to be the primary driving factor for many businesses. With cloud computing, there is no need to physically acquire hardware or a maintenance staff - computing power can be purchased on a pay-per-use basis. This simplifies the costs and also solves most of the scalability issues. A fitting example in this regard is the case of text extraction from Hillary Clinton’s White house schedules [16]. Washington Post engineer Peter Harkins used 200 nodes in Amazon
EC2 cloud to convert over 17,000 pages of non-searchable PDF documents to a searchable text database within 9 hours. The cost for this conversion was just $144.62, a significantly less expense than using a dedicated set-up. More importantly, the constraint on time was overcome by using a large number of readily provisioned computing nodes. This example highlights the cost saving in resource usage as well as the reduction of *time-to-market*, an important consideration to many commercial entities.

2. **Effort Advantage**: Convenience, in terms of software configuration, scaling and general maintenance of the software setup is significantly improved in a cloud computing environment. For example, with an Infrastructure cloud, images of virtual machines can be stored and cloned easily. This allows the software setup to be made once and then simply replicated. Scaling and maintaining the software environment is even easier with a platform cloud. Platform clouds have made such tasks transparent and thus, service consumers are not responsible for them.

Additional advantages that fall under both these umbrellas are as follows.

1. **Software Accessibility**: Clouds enable access to expensive software on a pay per use basis. For example, Amazon EC2 provides standard machine images pre-configured with IBM software (such as IBM DB2 database, IBM Websphere server) at a slightly higher hourly rate [17] rather than requiring the traditional heavy upfront licensing fees. This allows businesses to use and evaluate high quality commercial software and avoid any costly upfront licensing fees. An added benefit is the simplification of the configuration process. Many of these software suites are available as pre-configured machine images and can be instantiated to directly obtain a working system.
A related advantage is the possibility of conveniently instantiating a distributed system. For example, configuring a Hadoop based distributed processing cluster is not easy. The ability to use pre-configured machine images and the presence of programmatic access allows such systems to be instantiated rapidly and conveniently. An example is Amazon’s elastic map-reduce (EMR) [18] where users can spin up an on-demand Hadoop [19] cluster by simply specifying the type and the number of processing nodes required.

2. **Energy Conservation**: The rising energy costs and the environmental concerns have highlighted the importance of the highest utilization of computing resources. Cloud computing enables the utilization of unused computing power, primarily via virtualization. In the classic paper by Barham et al., virtualization was introduced as a viable method of increasing resource utilization while only sacrificing marginal levels of performance [20]. Many large organizations, including business entities and government agencies, have already begun to embrace cloud computing as a *green computing* initiative.

### 2.3 Challenges in Cloud Computing

Cloud computing, although slated for an impressive growth, face numerous challenges. Following is a list of broad categories of these challenges, ordered by their importance. The order is determined by condensed responses of 785 technology professionals at 39 high-profile enterprise technology companies by Northbridge [21].

- Security
- Regulatory / Compliance issues
2.3. CHALLENGES IN CLOUD COMPUTING

- Lock-in
- Interoperability
- Privacy
- Network Bandwidth
- Reliability
- Complexity
- Pricing
- Other (specified reasons)
- Expense
- Other (unspecified reasons)

These challenges can be broadly grouped under the following umbrellas.

1. **Security and Compliance**: There are a number of concerns of using cloud for data storage, either as part of an application or as a storage of its own. Many fear about the data security since it is not possible to make physical protection measures (unlike the lock-and-key protections in private premises).

A number of regulations that govern the storage of certain types of data is also a major consideration. For example, The Health Insurance Portability and Accountability Act (HIPAA) governs the storage and use of health care data and any application that stores health care
data is required to follow HIPAA rules. Specifically HIPAA states that “A covered entity must maintain reasonable and appropriate administrative, technical, and physical safeguards to prevent intentional or unintentional use or disclosure of protected health information in violation of the Privacy Rule and to limit its incidental use and disclosure pursuant to otherwise permitted or required use or disclosure”[22]. Providing such strict data safeguards is highly difficult in a cloud computing setting.

There are some suggested methods to use cloud computing environments and build HIPAA compliant solutions [23]. However there are debates over the level of compliance and the potential pitfalls of such solutions and many have opted not to adopt cloud computing based solutions for health related applications.

2. **Software related Consideration**: Major software related issues are vendor lock-in and interoperability.

Many cloud applications are developed to exploit certain cloud specific features and as a result, are not portable. These features may include cloud specific programming libraries, cloud specific storage systems or other services (such as queuing) or authentication that may be used either as an option or by mandate. For example, the use the Amazon queuing service may simplify an Amazon bound application, yet will effectively lock-in the application to the Amazon cloud environment.

There are also interoperability concerns in pushing applications to the cloud, primarily due to the lack of control over certain aspects of the environment. For example, many platform clouds do not allow fine grained control over system level configurations and thus, may not be able to have proper system level configurations, required for certain inter-operations.
3. **Other Considerations**: Issues such as application complexity and system reliability are also of importance. Most of these considerations would be simplified when the cloud eco-system stabilizes, typically by standardization.

### 2.4 Cloud Growth Forecast

Despite the challenges, the previously mentioned advantages have boosted the growth of cloud computing to an impressive level. A number of reputed market research firms have predicted this trend.

One of the notable examples is the Forrester report that predicts a total cloud market expansion beyond $60 billion by the end of 2012 [4]. This cloud market includes all types of cloud resources including storage. Given the improvements in networking, such cloud storage systems are now practical and are been used for many different scenarios. For example, it is now possible to maintain a personal music collection in a cloud storage and play on a multitude of devices on demand, demonstrated by commercial systems such as iCloud and Google Play.

A more specific prediction is highlighted in the Cisco cloud report [8]. Cisco report states that over 50% of the total workloads will be processed in the cloud by 2014 as illustrated in Figure 2.2. This specifically highlights the potential of the cloud to capture the space of computations and application hosting. The case with the traditional data center is that the bulk of computational power is unused (as discussed earlier under power considerations). Cloud computing has been the key discovery that made the data center more profitable as well as environmentally friendly.
2.5. Chapter Summary

This chapter presented a brief summary of cloud computing, first by defining it and then highlighting the benefits, challenges and opportunities. We direct the readers to the excellent article by Armbrust et.al, which discusses the various aspects of cloud computing in great detail for a thorough background [24]. The next chapter focuses on introducing the philosophy of abstractions and its applicability in cloud computing.

Figure 2.2: Cisco global cloud index outlining the workload distribution prediction: Data for this chart extracted from the Cisco cloud report [8].
Using Abstractions as a Solution Strategy

Abstractions are present in a number of domains, in a variety of forms. These abstractions are of different granularities and are applied in vastly different contexts. In this chapter we investigate where abstractions can be applied in the context of cloud computing, how they can be infused into the application life-cycle and how the different abstractions can be combined into a comprehensive solution. The purpose of this chapter is to set the foundation of this dissertation, i.e., present the core philosophical ideas of this dissertation.

3.1 A Typical Application Life-cycle

Figure 3.1 illustrates a typical life-cycle of a cloud application. Unlike applications for other environments (Say a personal computing environment), a cloud application considers deployment and management of the application as important post-development activities. In many cases, the deployment process requires careful planning and skilled personnel. The management process also requires similar planning and skills. Thus, the effort required in application deployment and management are non-trivial for a cloud application.

Also, the process is cyclic, i.e. the develop-deploy-manage cycle repeats until the desired
3.1. A TYPICAL APPLICATION LIFE-CYCLE

stability is achieved.

Note that there is a tacit assumption of the application life-cycle being top-down, i.e. the design being carried out up-front. While some top-down approaches are viewed as heavy, what is advocated in this dissertation is not the traditional big design up front (BDUF) style development. The presence of abstractions fully support agile development with simple, evolving designs, albeit applicable only in a domain specific context.

Figure 3.1: The typical top-down cloud application life-cycle
3.2 What Activities should be Abstracted?

We looked at the main activities in the typical application life-cycle, focusing our attention on instances where platform heterogeneity becomes relevant. Each step of the cloud application life-cycle suffers from consequences in platform heterogeneities.

**Developing the application**: Constructing the application or coding it, requires some level of platform awareness. When an infrastructure cloud is the target, the applications can be developed fairly generically, i.e., using standard development platforms or even commercial software libraries. The exception to this rule is when the application uses vendor specific support services. For example, an application targeting the Amazon cloud environment may use the Amazon SimpleDB [25] to store data or Amazon simple queue service (SQS) [26] to manage message queues. In this case, the application depends on Amazon specific libraries (and services) and hence becomes locked to Amazon.

When a platform cloud is targeted, coding the application becomes much more restrictive. Apart from the limitations on the software stack, there are other issues such as non-standard configuration files. For example, GAE supports only Java, Python and the experimental Go language. In case of Java, GAE requires the application to be bundled as a web archive (war) file and include an `appengine-web.xml` file (a non-standard configuration file) within a specific directory in the war file.

**Deploying the application**: Deploying an application refers to getting the application running on the target cloud. This is a straightforward activity in a platform cloud due to its application oriented nature. For example, the GAE standard development kit includes the `appcfg` tool,
3.3. A TWO PRONGED STRATEGY TO APPLY ABSTRACTIONS

Deploying an application to an infrastructure cloud requires more operations. In case of Amazon EC2, one would have to instantiate the correct Amazon Machine Image (AMI), install the necessary support software (if the AMI did not include them) and deploy the application artifacts via the platform specific deployment mechanism. The deployment mechanism may require the application files to be copied to a specific file system location and/or execute specific commands.

Managing the application: Managing applications refer to operations carried out after the application is deployed. These include taking data snapshots, managing the scale and undeploying the application in some cases.

Depending on the target cloud one may have special tools to perform these operations or use of a more system oriented workflow. For example, when taking a data snapshot in GAE, the GAE appcfg tool provides the necessary commands to do so. In contrast, when using Amazon EC2, one may have to connect to the respective virtual machine and execute an environment specific command sequence. This command sequence depends on the type of database being used.

3.3 A Two Pronged Strategy to Apply Abstractions

We apply two types of abstractions to the above identified activities. These are as follows.

1. For programming tasks (i.e. application design and development), we use DSLs.
DSLs can be considered mini languages with high levels of abstraction, targeted towards a domain. Chapter 4 presents the background in DSLs and dives deep into the language transformations.

2. For workflows and processes (i.e. application deployment and management), we use middleware.

Middleware is a software layer, typically used to provide abstractions (of various granularities) over complex and heterogeneous platforms. We applied the same principle in this research and introduced a cloud middleware to cover the various differences in cloud interactions. Chapter 6 provides extensive detail on the practical use of cloud middleware.

3.4 Prior Work: Abstract Based Approaches for Application Generation

There are multiple frameworks that generate (full or partial) programs based on an abstract specification. We list some of the well known types below.

3.4.1 Code Generation Systems for Service Oriented Architectures

A number of frameworks that implement remote communications (RPC) or support service oriented architectures (SOA) contain code generation tools to generate concrete code by compiling an interface definition. Almost all these tools generate code with place holders for the real service implementation or consumption. We list some of the well known code generation tools below.
CORBA  Common Object Request Broker Architecture (CORBA) uses a special language called Interface Definition Language (IDL) to define interfaces [27]. The IDL scripts are used with an IDL compiler to generate executable code for the targeted platform. Many major programming environments include IDL compilers in their development kits, for example, the Java Development Kit (JDK) includes the CORBA IDL compilers as part of the standard binaries [28].

Web services  Web service toolkits (primarily based in SOAP) also include code generation tools. Web Services Description Language (WSDL) for Web services plays a similar role to IDL by providing a system-agnostic service description. WSDL files can be processed to generate code as a service client (stub) or a skeletal service implementation (skeleton). Some of these tools include Apache Axis2 Web service engine [29] and .NET code generation tools [30].

An important point to note is that all these abstract specifications focus only on providing a programmer friendly interface for the abstract interface definition. The code being generated are either skeletons or stubs, support components that may be used in either accessing the service or constructing the service.

3.4.2  Code Generation Systems for Applications

There are a number of tools and frameworks that use code generation to create fully fledged applications. We present the details of one tool that represents the general characteristics of the such tools.

Google Web Toolkit (GWT) is an Asynchronous Java script and XML (AJAX) development
tool from Google, targeted for Java Developers [31]. Entire Web applications, including both the GUI and the business logic, are written in Java using the GWT API. GWT offers developers two important conveniences.

1. GWT compiles the Java files related to the UI into compact, optionally obfuscated, JavaScript files that support multiple browsers. This process offers developers a way to overcome the browser compatibility problems without maintaining multiple versions of the same UI.

2. During the development phase, the back-end is not programmed separately from the front-end. In fact the application appears monolithic during development. When the application is compiled, a Java script based front-end and a Java servlet based back-end is generated, ensuring the communication interfaces are fully compatible. This relieves developers from lengthy debug and test cycles on the service interface.

GWT however, is not based on a DSL. The abstract notions present in GWT are all through Java based objects and has a steeper learning curve.

### 3.4.3 The Philosophy of Software Factories

In the context of DSL based application generation, one of the most relevant philosophical ideas were expressed by Jack Greenfield et.al. [32]. They advocated a DSL driven, factory-like approach to software construction where each component of a complex software system is constructed using DSLs (and/or tools). While this style of software construction has not caught up, the ideas presented in software factories marked an important philosophy change.
The ideas presented in this dissertation are inspired by a similar philosophy. However, our proposals consider development in tightly scoped domains rather than in a general context.

3.5 Use of Abstractions in Cloud System Interactions

There are two broad classes of abstract driven cloud interactions.

3.5.1 Cloud Brokers

Third party middleware solutions for cloud computing with limited capabilities have become popular commercially. Known as cloud brokers, many of these services focus only on IaaS clouds and are mostly value added services over the base cloud interface. Cloud brokers usually distinguish their services by providing convenient user interfaces or special capability to provision and manage multiple instances at once. Commercially successful cloud broker services include Rightscale [33] and Scalr [34].

Features of a typical brokering service is listed below. These features are representative of most of the cloud brokers available commercially.

- Brokering services provide management capabilities to maintain deployments. Cloud deployments involve multiple software bundles, scripts and credential details. Cloud brokers usually offer a workspace-like environment to conveniently manage multiple artifacts relevant to different aspects of a deployment.
Brokers have capability to access multiple clouds of the same type, usually IaaS clouds. This is convenient at a system administration level, giving system administrations a single view of all the resources.

A level of automation is provided by adding monitoring capabilities and notifications over the resources. These added capabilities may come in terms of reusable scripts. For example, Rightscale provides such capabilities via rightscripts, an enhanced version of shell scripts.

Although these services offer conveniences over the typical cloud interactions, they do not offer true migration capability. A migration needs moving code and data, a capability not provided by any of these brokers. Some brokers allow migration at virtual machine level (i.e. a snapshot of a running virtual machine may be migrated to a different infrastructure cloud). Such lower level migrations may require a number of post-migration activities to establish a fully functional application.

### 3.5.2 Unified Cloud Programming Interfaces

These are of similar capabilities to brokers but are focused on developers. These interfaces are typically provided as programming language libraries and are predominantly focused on IaaS clouds. Notable example include Apache Libcloud [35] (for the Python programming language), Fog [36] (for Ruby programming language) and Juju [37].

Typical features of cloud programming libraries are as follows.

- All these libraries include unified programming constructs to represent the most useful cloud
components. For example, an instance is represented as a class in both Libcloud and fog libraries, allowing easy conceptual design of scripts.

- These libraries focus on providing convenient programming interfaces to the operations rather than value added services. For example, there are no built-in monitoring services, yet the capabilities of these libraries can be used to build such a service.

Similar to the brokers, programming interfaces also suffer from their inability to provide application level migrations.

3.6 An Augmented Application Life-Cycle

We introduce an augmented application life-cycle, based on the two primary modes of abstractions we identified before. As discussed in Section 3.1, this application life-cycle is light-weight but top down. In other words, albeit simple, there is the need of the high level organization up-front during the process. The program specifications in abstract form can be thought of as models, thus this approach may also be called a model-driven development process. This key steps in the DSL/middleware based application life cycle process are the following.

1. Develop programming code and data abstractions using the appropriate DSLs.

2. Deploy the artifacts using the middleware.

3. Manage deployed artifacts using the middleware.

This life-cycle is illustrated in Figure 3.2. The rest of this dissertation builds upon this top-down application life-cycle.
The activities are now based on abstractions and significantly less complex than learning each individually. For example, the special language presented in Section 5.1 (MobiCloud) is easy to learn and can be mastered in a few hours. This is significantly less effort than programming directly on GAE, Android or other cloud/mobile development platform.

3.7 Achieving Portability

This augmented life-cycle supports portability in the following way.

1. Applications are developed using DSLs. Thus, building an executable version of the application for a different cloud is simply a targeted code generation (Note the assumption that

Figure 3.2: The augmented cloud application life-cycle. Abstractions are applied at every activity of the life-cycle to decouple the platform dependent operations.
the target platform is supported by the generator) and all that is required to keep track of is
the DSL script. All the effort is focused on building the DSL script(s) and the effort required
to re-target the application is minimal. Thus the applications are portable in terms of their
code, data schema and design.

2. Any data accumulated for the applications can be automatically transformed to the re-targeted
platform, when a migration happens. While this dissertation does not focus on this data
transformational aspect, the generation mechanism can certainly support it. This is possible
because the transformations are known to the generator and thus, it can create configurable
transformations to and from a common data model.

3. Deployment and management happens through a unified interface. The operational differ-
ences between the clouds are minimal and thus the deployment and management processes
are also learn-once, i.e. the knowledge in the process is portable across cloud environments.
If the middleware layer is API driven, frequent operations can be scripted. In that sense, an
operational process script also becomes portable.

### 3.8 Chapter Summary

This chapter presented the philosophy of applying abstractions at various activities in the appli-
cation life-cycle. The two main techniques are using DSLs and middleware, where the typical
application life-cycle can be modified to be driven completely by abstractions.

The next chapter discusses the DSL based application development process in detail.
Platform Agnostic Cloud Application

Development

This chapter covers the theoretical details of using abstractions in the development process. It includes the background details of programming abstractions as well as the limitations in transforming high level specifications to executable instructions.

4.1 Preliminaries

4.1.1 A Background on Domain Specific Language

In order to understand the use of domain specific abstract specifications, we present a systematic background discussing the idea of abstraction as well as the traditional language theory.

4.1.1.1 What is an Abstract Concept

We clarify the use of the word abstract concept by using a modified version of the definition provided by Kleppe [38].
“The abstraction level of a concept present in a software language as the amount of detail required to either represent (for data) or execute (for processes) this concept in terms of the perceived (virtual) zero level.”

The perceived zero level in this case refers to the base line that the abstractions are measured from. The absolute zero line for a software language is the computer hardware. Yet, with the advancement and sophistication of high level computer languages and their tools, the zero line may be considered at a much higher level. This elevated zero line is what is referred to as a virtual zero line [38].

To illustrate this further, consider the object oriented programming paradigm where it is typical for computer programs to be modelled and programmed using objects. In this case, all the program design happens assuming objects to be the primitive building blocks, thus perceiving a virtual zero line at the level of objects. The objects, defined further in terms of data structures to hold its state and methods to transform its state, will obviously need to be mapped to memory and instructions that can be executed on hardware. However such transformations are facilitated by well established software frameworks (compiler, linker libraries) and can be performed mechanically without human intervention. Hence, the program designer can conveniently assume objects to be the lowest level of abstraction.

4.1.1.2 Traditional Language Specification

Language theory states that a language specification (L) requires three elements to be described:

1. An abstract syntax model (ASM): this is the high level model of the language, often in-
visible and used directly inside the language interpreter mechanism. Also known as the conceptual model, the ASM can be represented as a directed, labeled graph.

2. **One or more concrete syntax models (CSM):** This is generally the syntax seen by the programmers and what is typically referred to as *the language*. A single ASM may have more than one related CSM.

3. **Set of transformations (mapping):** A mapping from the ASM to the CSM (defined per CSM) specifying the conversion of the ASM to concrete syntax and vice versa. These transformations are reversible, i.e., a program representation can be transformed losslessly from ASM to CSM and vice versa.

Language literature also discusses three more elements, relevant to a language specification:

1. **A semantic description:** a description of the meaning of the program or model, including a model of the semantic domain.

2. **Required language interfaces:** a definition of what the programs need from programs written in other languages.

3. **Offered language interfaces:** a definition of what parts of the programs are available to programs written in other languages.

The semantic description warrants our special attention. A typical language specification consists of a formal syntactic specification accompanied by a rigorous but informal semantic specification. This is partly due to the familiarity with other language designs, and the accessibility of
the prose version of the semantics of the language to the programmers at large. Formal specifications are indispensable only for the specialized developmental activities such as the construction of compilers and interpreters, program transformation and verification tools, etc. In other words, a clear semantic description of the language is essential, yet a formal one may not be presented in all cases.

There are three types of semantics discussed by Nielson and Nielson [39] in the context of traditional language theory.

1. **Operational Semantics**: Meanings for program phrases defined in terms of the steps of computation they can take during program execution.

2. **Axiomatic Semantics**: Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties.

3. **Denotational Semantics**: Concerned with giving mathematical models of programming languages. Meanings for program phrases defined abstractly as elements of some suitable mathematical structure.

These semantic representations are complementary to each other. The use of a given type of semantic representation depends on the context of interest. For example, when the partial correctness of the programs are of interest, axiomatic semantic representation is preferred over the others [39].

The readers are directed to Kleppe [38] for a thorough coverage of the language theory concepts discussed above. This dissertation focuses on a special branch of languages called **domain**
specific languages, described in detail in Section 4.1.1.3. Other languages that are not DSLs, are referred to as general purpose programming languages (GPPL).

### 4.1.1.3 What is a Domain Specific Language?

Van Deursen et al. [40] state that

> “a domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain”.

A domain in this case is the set of entities and their associated operations, selected specifically to represent and operate on these entities in a restricted context. Domains can be of varying degrees of granularity. Some domains are highly constrained, while others have a much larger scope. A fitting example in this case is Mathematics. While Mathematics can be considered a large domain as a whole, specialized sub-fields of Mathematics such as Matrix Algebra or Statistics can be considered as domains with more restrictive scope.

While GPPLs consider the integrity of the CSM as a significant aspect (thus the emphasize on grammars and ultimately the semantics of a program), DSLs rarely focus on these traditional aspects. This is partly due to the fact that many DSLs are created as subsets of GPPLs, assuming the presence of a well defined grammar. Instead, the focus of the DSL is the domain model the DSL represents. we discuss these facts further in the subsequent sections.
4.1.1.4  Modeling and Metamodelling

A model can be thought of as an abstraction of a system and/or its environment. [41] In software engineering, the term model is frequently used to refer to graphical representations or abstract versions of the program code such as in Unified Modeling Language (UML) [42]. In practice, a model can be represented in any form. For example, many models are stored using specialized textual languages that represent the salient features of the models.

A metamodel defines the abstractions used by the modeling notation. The metamodel acts as the schema for a model, defining the permissible components and the constraints applicable on the model. Thus, based on conformity, one can create a hierarchy of models [43] as illustrated in Figure 4.2a. Specifications higher than meta-metamodels are typically not useful in the context of model driven development.

4.1.1.5  Metamodelling and DSLs

Metamodels are considered an integral part of this research since a DSL can be considered a representation of a domain model. We assert that the ASM of the DSL is the domain metamodel.

To illustrate this using an example, consider the simple domain of unary and binary mathematical expression. The high level concepts that encapsulate this domain are operator and expression. Expression may further be specialized as unary and binary expressions and number may also be added as a subclass of expression to support literal values. These concepts, arranged in a linked graph (Figure 4.1) defines the metamodel for mathematical expressions. This metamodel is in fact
Figure 4.1: Standard metamodel for Mathematical expressions

listed by Kleppe [38] as an example. The original metamodel illustrated by Kleppe uses UML notation and contains more details (such as cardinality). Some of these details are omitted for brevity and this model is depicted as a simplified directed, labelled graph.

Now assume that a language is needed to describe mathematical expressions. This requires a modeling of the relationship between various components of mathematical expressions and representing these expressions in a syntax agnostic manner, i.e. an ASM for this language needs to be constructed. It is easy to see that the components of this ASM are essentially the same as the metamodel. For example, each literal number present in a mathematical expression will need to be represented as an instance of \textit{number} type, which is one of the components defined in the expression metamodel.

\subsection*{4.1.1.6 Model Transformations}

A Model transformation is a mapping from one model to another, defined on the metamodels but operated on the respective metamodel instances. This relationship is illustrated in Figure 4.2b.

It is worth noting that there are multiple methods of model transformations, graph based transformations being only one of them. The work covered in this dissertation, graphs are selected
as the primary representation of the models and hence the focus is limited to graph transformations. Czarnecki et al. [41] provide a thorough list of transformational techniques used in model transformations. Also, a specific transformation technique, say a rule based implementation is not considered. The focus of this work is only on the conditions and special requirements that apply to these transformations, rather than how they are performed.

4.1.1.7 Semantics of a DSL: Metamodeling vs Traditional Semantics

Metamodeling is the preferred way of establishing semantics of a DSL. This is an alternative to the rigorous semantic representations used in traditional language construction, discussed earlier in this section. Since GPPLs are not confined to a domain, they require a domain independent specifications to establish the semantics of the program formally.

The closest semantic representation to metamodeling is denotational semantics, briefly mentioned in Section 4.1.1.2. A metamodel can be thought of a denotational semantic representation (albeit less rigorous) in a domain context.
4.1.2 Formal Definition of a Cloud Application

Before attempting to generate a cloud application, we need to investigate the nature of a typical cloud application. We first identify the different aspects a cloud application represents and then proceed to construct the definition based on these aspects.

4.1.2.1 Aspects of a Cloud Application

To establish a formal definition for a cloud application, the aspects to be modeled need to be identified. Inspired by the area of aspect oriented programming and following a similar line of design in the area of services \[44, 45\], four types of semantic aspects for a cloud application are identified \[46\].

They are,

- **Data aspects**: These include the definitions of data structures, relationships across multiple data structures as well as restrictions on the access of some of the data items.

- **Functional (Logic and process) aspects**: These are the details pertaining to the core functionality (commonly referred to as the *business logic*) of the application. Unlike in a service, the functional and execution semantics are tightly tied together for an application. For example, a service may delegate the exceptions to an external service although exceptions are an integral part of the core functionality of an application and seldom designed separately.

- **Non-functional aspects**: These are details not-directly relevant to the business logic but requires consideration at a different level. Examples include access control and logging.
While these are not part of the core functions, an application nevertheless requires these aspects to be defined. Some of these considerations may require certain libraries or internal code changes. A typical example is logging where the application internally implements the execution points for logging but the users get to control the granularity of the log entries such as INFO (informational content) vs ERROR (only errors).

- **System aspects**: Relevant system considerations include deployment descriptions and dependency management. These considerations are neither relevant for the business logic nor the non-functional considerations but become important when the application starts running on a system. The number of computing nodes, the separation of software components are some of the considerations to be made as part of the system aspect.

These aspects are generally orthogonal to each other, thus can be addressed independently. For example, many non-functional aspects, such as security, are layered on top of the functional aspects and can be varied while all other aspects remain unchanged. Similarly, the same application can be deployed with a variety of system configurations, while the functional, data or non-functional aspects remain unchanged.

It is important to note that while these are generally independent, dependencies may exist at different layers of development. For example, enabling a different QoS requirement may not effect other components at design level, yet may require a configuration change in the system configuration at development/deployment levels.
4.1.2.2 An Example of the Aspects

Figure 4.3 illustrates each of these aspects superimposed over the application components of a service-oriented numerical data processing application (This example is based on a real application that provides a service interface to a private Hadoop cluster, providing numerical data processing capabilities for a MATLAB based front-end system). The core function expected from this application is numerical data processing, thus the implementation of this logic is considered to be part of the functional aspects. The representation and storage of the numerical data is considered under the data aspects. QoS capabilities of the interface are considered part of the non-functional aspects and the system configuration is considered as part of the system aspects.

To illustrate the independence of these aspects, consider changing the security of the service interface from HTTP to HTTPS (HTTP traffic over a secure socket layer). Such a change neither affects the function nor the data structures of the application. Similarly, system configurations can
change while the business logic, data structures and the service interface remain unchanged.

4.1.2.3 Principle Assertions

The following are the driving assertions in this work that establishes the importance of the aspect based separation of concerns. We assert that

1. Each aspect can be expressed using a DSL.

   Although a DSL is only capable of expressing a specification confined to a possibly narrow domain, each aspect of the application can be expressed via such a specification. Data specifications are the easiest to understand in this context since many applications specify their data schema and manipulation via SQL. Similarly, it is possible to either use or create a DSL that expresses the program logic, non-functional details and the system configurations.

2. The abstract syntax model (ASM) of each of these DSLs (i.e. the respective domain metamodels) can be represented using a graph.

   While there are many other possible alternatives to represent an ASM, the graph representation is the most generic and the most flexible. Graph representations offer easy visualization and also multiple forms of serialization. Thus, throughout this dissertation, a language ASM is always represented using a graph.

Definition 1 formally presents the fact that a language ASM can be represented as a graph.

**Definition 1** Abstract Syntax Model (ASM) is a directed, labeled graph \( G = (V, E, l_E, l_V) \), where
• $V$ is a set of vertices,

• $E$ is a set of ordered pairs of vertices, called edges,

• $l_E$ is a labeling function, defined on $E$ and applies to $E$, called edge labels.

• $l_V$ is a labeling function, defined on $V$ and applies to $V$, called vertex labels.

The interpretation of the meaning of the vertices, edges and labels of an ASM graph is dependent upon the domain that is represented.

### 4.1.2.4 Formal Definition for a Cloud Application

Based on the above assertions, the following formal definition of a cloud application is established:

**Definition 2** A cloud application $CA$ is represented by the four tuple $CA = \langle G_{data}, G_{func}, G_{qos}, G_{sys} \rangle$ where

• $G_{data}$ is the ASM graph for the DSL representing data,

• $G_{func}$ is the ASM graph for the DSL representing functional details,

• $G_{qos}$ is the ASM graph for the DSL representing non-functional details,

• $G_{sys}$ is the ASM graph for the DSL representing the system configuration.

Definition 2 presents a strategy to separate the different concerns of a cloud application. Specifications covering each of these aspects should be present to create an executable cloud application. Application developers however, may not explicitly specify all of them, where the cloud
environment would make a reasonable assumption. For example, in our numerical data processing application, the developers may not explicitly specify the QoS parameters. In this case, the cloud environment would supply the QoS guarantees it is configured to provide by default.

**4.1.2.5 Addressing Cloud Specific Features**

A careful reader may have realized that no specific references has been made to the distributed nature of some of the clouds. Indeed, this definition deliberately omits such details. The purpose of this definition is to separate the aspects that can be further analyzed.

Explicit parallelism may be modeled as part of different semantic aspects of Definition 2. For example, the system semantics may include a horizontally replicating configuration providing parallelism at the system level but this parallelism may not be visible (nor effect) any of the other semantic aspects. Similarly, data storage may be performed on a distributed database, yet the rest of the system may well be completely isolated and none of the other components would be affected.

To exemplify this further, consider the GAE application model. GAE applications are well shielded from the distributed nature of the platform, as well as the data store. In fact, building an application for GAE is very similar to building a non-distributed Web application, except for the use of several special libraries. On the other hand, building a scalable Web application for Amazon EC2 would require explicitly addressing the replication issues or using a service, such as Amazon Elastic Beanstalk [2] that provide load balancing features.
4.1.3  A Background on Data Modeling

Analogous to language theory, data modeling theory defines three types of schema. A schema is a specification of data elements and their relationships.

1. **Conceptual Schema**: a high level model of the data organization. Usually a conceptual schema does not represent any details of the data organization, just the entities that need representation and their relationships. A popular representation of the conceptual schema is the entity-relationship model [47].

2. **Logical Schema**: the data organization, conforming to a modeling paradigm. The modeling simply known as *data modeling* usually refers to the logical schema. A logical schema is usually expressed using the relational model, first discussed by Codd [48], and later incorporated into major database systems.

3. **Physical Schema**: The data organization in terms of storage and representation. The physical schema considers storage aspects such as partitioning and distribution.

In practice, it is rare to separately model all the three schemas. Many organizations only create the conceptual schema and use computer aided database management tools to transform the conceptual schema to a logical schema. The physical schema is often hidden and the pertinent aspects of a physical nature are managed by a database management system, often automatically.

Cloud data management solutions are focused on availability and have given up some of the behaviors the traditional database systems have considered essential (See Section 4.1.4.1 for a detailed discussion). Not surprisingly, the data modeling strategies for cloud databases are also...
4.1. PRELIMINARIES

less-rigid than the traditional data modeling approaches. The most prominent cloud data modeling approach is the Entity-attribute-value (EAV) data model. Alternative terms such as *key-value data model* and *document-oriented data model* are also used to refer to EAV data models. EAV based storage systems are also sometimes called *schemaless* databases due to their lack of a rigid schema.

An EAV data model considers entity as the primary building block. An *entity* is simply a collection of data attributes, a unique key attribute often being the only mandatory attribute. Each entity may carry different attributes and adding or removing an attribute from an entity does not require changes in other stored entities. This type of storage can be visualized as a matrix where each row represents a document and each column represents an attribute type. A rendering of a document oriented data store as a matrix would yield a large (typically sparse) matrix and thus, an EAV storage may even be treated as a matrix mathematically.

EAV data models may support declarative language access. Although all the features of the relational data model based operations cannot be supported, an *SQL-like* declarative language can be provided to access entities. For example, Google Big Table supports an SQL-like declarative query language called GQL [49] which has a different syntax for filtering (WHERE clause) and does not support operations such as joins, yet provides a familiar syntax for most developers.

It is also important to note that the context of this dissertation only includes the data stores with data management features where the above mentioned schemas can be applied. File system like storage solutions such as Amazon S3 [50] and the Windows Azure blob storage [51], are also marketed as data storage solutions. However these storage solutions do not offer advanced data management functions beyond the capabilities of a file system and thus are not considered in the context of this dissertation.
4.1.4 Other Considerations

4.1.4.1 Relevance of the Consistency Availability Partition-tolerance (CAP) Theorem

When focusing on cloud applications, an important factor to consider is the effect of CAP theorem [52]. The Consistency, Availability and Partition tolerance (CAP) theorem states that only two of the CAP factors can be achieved at once. For example, if consistency and availability are required, such a system cannot be made to tolerate partitions. On the other hand, a partition tolerant system can be made to be highly available but cannot provide strict consistency guarantees. First stated by Eric Brewer as a conjecture, it was later formally proved by Gilbert and Lynch [52].

It is important note that CAP theorem does not imply total exclusion. Instead it implies various strengths. To take the example of a highly available, partitioned data storage, these data storages do offer a level of consistency (in fact a completely inconsistent data storage will be useless). However that level of consistency is much weaker when compared to a traditional RDBMS system that provides ACID guarantees.

CAP theorem is the fundamental governing principle in distributed database systems. It explains the reason for the weaker data consistency models, particularly the eventually consistent model [53] followed by many highly available cloud data stores. These data stores are typically distributed (i.e. partitioned) and thus have to sacrifice either availability or consistency according to CAP theorem. Many data stores of this type, such as Google BigTable and Amazon Simple DB, have given up strict consistency guarantees in favor of high availability. The original purpose of such data stores is to serve highly popular Internet applications where availability is critical, thus their weaker consistency is justified.
The impact of CAP theorem on cloud applications is that it restricts the type of applications that can use the cloud, primarily based on the applications data storage requirements. In other words, it provides a means to determine the suitability of an application for the cloud [54]. For example, a banking application or an air ticket reservation system which values data consistency over availability may not be successfully moved to the cloud. On the other hand, data collected for a restaurant rating system (such as Yelp![55]) are well suited for the cloud since this type of data can tolerate a lower grade of consistency.

In reality, many current cloud software environments offer traditional RDBMS services (Google SQL services, Amazon RDS etc) and it is possible to place even an application with strict consistency requirements in the cloud. But these applications are limited in their ability to scale (since achieving consistency means sacrificing availability).

The relevance of the CAP theorem (and other principles of data consistency that has been established, based on it) is that it forces this research to focus only on specific types of applications that can tolerate the weaker consistency data models. In others words, the transformational mechanisms covered in this dissertation do not provide the magical ability to overcome CAP constraints when applications are migrated across platforms. The applications in question should be able to tolerate the weaker data store consistency guarantees in the first place, if they are targeted towards such data stores during an application migration.

4.1.5 Problem Formation

There are multiple challenges in using DSLs, ranging from pure technical challenges that affect the range of possible solutions to social issues that affect the adoption. This dissertation considers
the following broad challenges to be the most significant in the context of cloud programs.

1. **Applicability of DSLs**: The types of domains that DSLs are most effective.

2. **Effort of creating DSLs**: The effort trade-off in creating the DSL infrastructure and associated tooling.

3. **Usability of DSLs**: The human perception of the DSL, in terms of its complexity and applicability. A DSL should be balanced in terms of complexity and completeness.

The first two challenges, applicability and effort requirement, can be addressed reasonably. This dissertation presents techniques where applicability and effort can be quantified. The third challenge, usability, however is hard to address. Usability is highly subjective, often depending on a number of non-technical factors.

Given that the two most significant actions are *code migration* and *data migration*, the theoretical focus is on the model transformation process. Thus, the two theoretical details this thesis plans to investigate are the following:

1. **Properties of the language model transformation from the user domain ASMs to distributed ASMs, suitable for cloud environment implementation.**

2. **Properties of the schema transformation from a conceptual schema to a logical schema to suit the respective software environment.**

In this dissertation, we treat both these transformations as a language model transformation problem.
4.2 Applying DSLs to Develop Cloud Applications

4.2.1 Language Model Transformation

In this section, the language transformational features are investigated in detail by using a symbolic representation.

The focus is on the transformation of the functional language ASM from the users domain to the cloud environment, i.e., the transformation of the $G_{\text{func}}$, introduced in Definition 2. Although $G_{\text{func}}$ exists as part of the cloud application and may be intermingled with other details such as data definitions, theoretically it is possible to address the functional transformation in isolation. $G_{\text{func}}$ is chosen without the loss of generality, i.e. some of the requirements relevant to functional specification transformations are also applicable to other graphs in Definition 2.

The transformation from the domain model to a cloud-supported implementation model depends heavily on the details of the domain metamodel. In other words, the domain metamodels must be detailed enough so that a meaningful transformation can be made. This realization leads to the primary principle that source metamodel graphs need different vertices for semantically distinct language constructs, regardless of their syntactic representation.

It is typical for ASMs to focus purely on giving an abstraction of the CSM, where syntactically similar constructs are modeled indistinctively. The simple expression metamodel, introduced in Figure 4.1 (Section 4.1.1.4) is used as an example. In this model, the concept operator is represented by a single vertex despite the fact that many semantically different operators may exist. For example, increment and decrement operators represent completely different tasks but are represented as a single vertex in the typical metamodel since their representations are similar in a
concrete syntax. Figure 4.4 illustrates an enhanced metamodel that defines each operator as a distinct vertex. Usually this level of detail is considered excessive and unnecessary in syntactically driven ASMs.

![Enhanced Metamodl for Mathematical expressions, showing the sub expressions](image)

**Figure 4.4: Enhanced Metamodel for Mathematical expressions, showing the sub expressions**

### 4.2.2 Requirements on ASM transformations for Cloud implementations

In this section, the governing principles are formally presented as *requirements* and *rationalies*. While these are not as rigorous as theorems and proofs, they can still be considered the governing principles of this research.

Consider $G_d$ as the ASM of the domain, $G_c$ as the ASM of the cloud, $G_{d(meta)}$ and $G_{c(meta)}$ as the respective metamodels, represented as graphs. Consider the transformation $T_{d\rightarrow c}$, denoting the source and target as the domain and cloud respectively. Thus, $T_{d\rightarrow c}$ is defined using $G_{d(meta)}$ and $G_{c(meta)}$ but applies to $G_d$ and $G_c$ respectively. The relationship between these models is illustrated in Figure 4.5.

**Condition 1** $G_{d(meta)}$ must define distinct vertices for each semantically distinct domain concept.
4.2. APPLYING DSLS TO DEVELOP CLOUD APPLICATIONS

Figure 4.5: Relationship between metamodels, models and transformations, represented symbolically

Requirement 1 states that the source metamodel should have elements for each and every distinct domain concept that may implement a semantically distinct operation.

**Rationale 1** Assume that there is a vertex $\lambda_{\text{meta}}$ in $G_{\text{d meta}}$ that has two interpretations. Then there exists at least two vertices in $G_d$, Say $\lambda_1$ and $\lambda_2$, that comply to $\lambda_{\text{meta}}$ but has two interpretations, hence should map to two different vertices in $G_c$. However, since there is one vertex in $G_{\text{d meta}}$, only one mapping can exist for it in $T_{d-c}$.

Thus, trivially $T_{d-c}$ cannot manage different mappings to $\lambda_1$ and $\lambda_2$ unless they map to two different meta concepts in $G_{\text{d meta}}$.

**Condition 2** $T_{d-c}$ is a surjective (onto) mapping, i.e. all vertices in $G_c$ must be defined by the transformation.

Requirement 2 highlights the fact that the transformation must yield a complete target graph. This does not mean that all vertices in the source graph will be mapped. Rather, the result of the
transformation, i.e. the target graph that gets created as a result, should be complete. This is a different way of specifying that the transformation should provide sensible defaults to avoid an incomplete target graph.

**Rationale 2** Assume that the mapping is not surjective.

Then $G_c$ is missing at least one vertex $\lambda$ needed to complete the model and thus, $G_c$ is incomplete.

Then $G_c$ cannot be converted to a working program.

Thus, In order to have an executable program, the mapping must be surjective.

As a result of Requirement 2, we can derive lemma 1.

**Lemma 1** $T_{d\rightarrow c}$ is not reversible.

Lemma 1 states that the transformation is not reversible. This can be trivially rationalized by considering the properties of a general surjective mapping, except for the special case of the mapping being bijective. The usual case in this context is that the target language is almost always at a lower level of abstraction which makes the bijective case non-existent (the transformation can be considered bijective when the models being translated are at equal levels of abstraction, enabling a lossless conversion in both directions).

### 4.2.3 Considering Explicit Parallelism

These base requirements result in more restrictive requirements that apply into domain dependent contexts. One of the important domains is the translation to a map-reduce program, which is a common requirement when the parallel nature of infrastructure clouds need to be exploited.
First, the concept of the map-reduce task graph is introduced. A map-reduce task graph, also called a *physical plan* is a task graph representing the map and reduce task sequences for a given operation. For almost all practical cases, a single map or reduce combination is insufficient and requires a combination of multiple map and reduce tasks. The map-reduce task graph represents this sequence of map and reduce tasks.

Figure 4.6 illustrates a map-reduce task graph generated by the PIG compiler for a statistical operator, sum normalization. PIG [56] is a SQL like DSL for map-reduce tasks. The illustrated operation is a common statistical operation in data processing and this particular example is taken from a scientific data processing application (See Chapter 5).

Figure 4.6: An example map-reduce task graph (plan) for a sum normalization operation

When requirements 1 and 2 are applied to the special case of translating the DSL to a map-reduce program, we derive the following condition, *applicable only when transforming a domain model to a map-reduce model.*

**Condition 3** All vertices of $G^*_{d}$ must be mappable to the components of a map-reduce task model.

Requirement 3 states that all domain concepts should have a translation that can map it to a map-reduce task graph. An operator may map to one or more map-reduce cycles.
4.2. APPLYING DSLS TO DEVELOP CLOUD APPLICATIONS

4.2.4 Significance

There are at least two important practical impacts of these theoretical investigations.

First, these theoretical findings determine the type of domains that this methodology is sensible (depending on the effort and other trade-offs) to be applied. For example, some domains may not have suitable operators that can be transformed to Map-reduce style implementations, limiting their applicability in cloud environments that can exploit the parallelism. Second, they also establish limitations on what can be performed, i.e. operations such as reverse engineering. These limitations are important to assess the applicability of using abstractions.

4.2.4.1 Domain Modeling Requirements

The first impact of these conditions can be seen in the effort required for domain modeling (Tacitly assuming that the application specification can indeed be converted to a cloud environment). Condition 1 requires domain modelers to identify the semantically distinct concepts and incorporate them to the metamodel appropriately. Such a task obviously requires more effort than typically anticipated and adds additional overhead at the design phase. While such extra work is feasible in restricted domains, it may require considerable effort in domains that have a larger scope, encapsulating a large number of concepts.

One method suggested in this dissertation is to use the Effort Trade-off as a means of determining the suitability of the domain. This requires a preliminary metamodel of domain. Given the large number of modeling and DSL creation tools, it is assumed that a preliminary metamodel can be constructed quickly and without significant commitment. The following section describes
a systematic method of determining the effort trade-off.

### 4.2.4.2 Determining the Effort Trade-off

To determine the trade-off in effort, a single indicator $R$ is introduced. The purpose of $R$ is to determine the effort trade-off in a single target platform, assuming the domain has already being modeled.

1. $LOC_{templates}$: LOC count of the templates.
2. $LOC_{generated}$: LOC of the generated code.
3. $LOC_{dsl}$: LOC of the DSL script.

The following assumptions are made

1. *The base code generation framework (parsing, syntax tree generation etc) is in place.* Thus the effort required is limited to the creation of templates.

2. *The effort required to create the templates can be estimated by the lines of code (LOC) count.*

   This is somewhat naïve since LOC does not indicate the complexity of the code. However, it can be measured easily and rapidly, making it suitable for this type of indicator.

3. $LOC_{dsl} < LOC_{generated}$.

These metrics are combined to form $R$, using Equation 4.1.

$$R = \left( \ln \frac{LOC_{dsl}}{LOC_{generated}} - \ln \frac{1}{LOC_{templates}} \right)^{-1} \quad (4.1)$$
4.2. APPLYING DSLS TO DEVELOP CLOUD APPLICATIONS

The rationale behind organizing these metrics in Equation 4.1 is as follows. The ratio between $LOC_{dsl}$ and $LOC_{generated}$ is a direct indication of saving in effort gained by using the DSL. However the generation capability of the DSL comes via the templates and thus, the effort required in creating the templates should be discounted from the gain. Arranging these metrics using their logarithms ensure $R$ is normalized and remains between 0 and 1, given the assumptions.

For practical cases, lower $R$ values are better. While it is difficult to denote a typical range ($R$ highly depends on the characteristic of the domain and the target platforms), higher $R$ values suggest either extensive template code efforts (with respect to the generated code) or large DSL code scripts that would offset the advantage gained from introducing a DSL in the first place.

4.2.4.3 Limitations in Transforming to Explicit Parallelism

Condition 3 states that the a complete transformation from the domain model to the map-reduce task model should exist, in order to implement explicit parallelism. In other words, every domain concept represented in the DSL should have an equivalent operation or a combination of operations in map-reduce.

In practice this is hard to achieve, even in a limited domain. The reason for this can be attributed to the DSL feature gap where all required features may not be supported by the implemented features. This implies that it is always possible for some desired features to be not covered by the DSL.

An example for this can be found in the context of the SCALE project (Chapter 5). Some specialized statistical operators used in NMR data processing, such as orthogonal projection on latent structures (OPLS)[11] have no parallel implementations yet (OPLS is an iterative process
and it may be possible to convert it to an explicitly parallel version. It has not been formulated into a explicitly parallel process yet). Thus, the original SCALE language does not support OPLS as an operator, even though it is highly desired.

The impact of this is felt when a desired feature becomes essential. In that case only a portion of the program can be distributed. As in the case of SCALE, the code generators have modifications that allow them to incorporate sequential (non distributed) code segments to the programs. This is very inefficient (for example, the data has to be pulled from the distributed file system to the local file system, processed and put back to the distributed file system for a non-distributed operation), yet deemed necessary in extreme cases.

### 4.2.4.4 Reverse Engineering

Lemma 1 states that it is unreliable (or impossible in most cases) to reverse the transformation. This means that trying to reverse engineer programs and generate a DSL representation is not possible.

In practice, one may be able to glean a reasonable set of abstractions of a limited set of existing applications. It may also be possible to reconstruct an abstract form from an application if it was originally created using abstractions and includes transformational annotations. Transformational annotations are special fragments of meta-data that does not effect the functionality of the application (such as comments) but are useful only to a reverse transformer. This however should not be taken as a general property. The abstractions in the scope of this research are domain focused and can only be converted to an executable program by incorporating significant (often assumed) details. It is simply not possible to take a program pertaining to an undisclosed domain and convert
4.3 Chapter Summary

This chapter discussed, in detail, the theoretical aspects of domain based application development and the consequences of using these techniques for cloud applications.

- We discussed the concept of four types of semantic aspects (Data, functional, non-functional, system) in a cloud applications.

- We established three conditions that govern the transformation process.

- We established that these theoretical conditions give rise to practical limitations in cloud applications, particularly for actions such as reverse engineering.

We look at two practical applications in the next chapter to investigate the application of these principles in practice.
Platform Agnostic Application Development in Practice

This chapter presents two applications of applying abstractions to the development process. These two applications follow the principles established in the previous chapter yet address completely different needs in two disparate domains.

5.1 MobiCloud - Cloud-Mobile Hybrid Application Generator

MobiCloud is a unique solution that simplifies the development of multi-platform applications that are connected to cloud back-ends [57, 58]. These applications are named cloud-mobile hybrids (CMH), due to their nature of having a cloud centric back-end and a mobile device based front-end. At least 3 challenges exist today in developing CMH applications.

1. The multitude of existing clouds offer different paradigms, programming environments and persistence storage. The heterogeneity present in the core cloud services effectively locks the developers to a particular vendor.
2. A number of mobile development platforms exist today, each with different development environments, Application Programming Interfaces (API) and programming languages. Fragmentation of APIs even within a single platform forces mobile application developers to focus on only specific platforms and versions [59]. The current practice in the industry is to concentrate the development efforts on selected mobile platforms, leaving out a significant portion of devices and platforms in the market.

3. Developing the back-end and front-end as separate components require managing the communication interfaces. The presence of Remote Procedure Calls (RPC) makes the whole development process tedious, even with an arsenal of sophisticated tools at a developer’s disposal. The separation of the front-end and the back-end is also a source of version conflicts with clients and services where the service API has to be maintained at the level of the least capable client. Introducing changes to the service API would break the existing clients, a common problem faced by many of the mobile application vendors.

MobiCloud addresses these issues by providing a model driven development strategy. This is based on a DSL and the generation machinery capable of using the DSL to generate separate front-end and back-end applications and the required service interfaces. A MobiCloud generated application bundle includes a mobile application package and a cloud application package.

The cloud application package includes

- A data storage mechanism tied to the storage technology that suits the target cloud environment.

- A service layer capable of exposing the operations on the data store.
The mobile application package includes

- A service access layer in the targeted front-end capable of accessing the services defined on the server side.
- User front-end components.

5.1.1 Model-View-Controller Design Pattern

The most appropriate design pattern for user interface based applications has been identified as the Model-View-Controller (MVC) pattern. Figure 5.1 illustrates the major components present in a MVC based design.

![Model-View-Controller design pattern](image)

*Figure 5.1: Model-View-Controller design pattern*

*Model* represents a data-structure that holds a neutral representation of the data items pertinent to the application. A *view*, representing the data in a format suitable to the user, observes the model and updates itself. Any interactions with the view are processed via a *controller* which modifies the model and avoids any inconsistent states by restricting the operations on the model. When observing the model is expensive, the controller may notify the view of the status of an operation, instructing the view to refresh.
The advantage of the MVC pattern is the separation of concerns and the relative ease of modifying each component. For example, the same model can be rendered via different views by only changing the view component. Similarly, new operations and constraints can be implemented by modifying the controller, keeping the other components untouched. A partial practical example that illustrates this is a chart added to a spreadsheet. The data is stored as a table but multiple types of charts can be added over the same data table. Modifying the data automatically modifies the views, i.e. the associated charts. This example is partial since a distinctive controller component is missing, yet it provides a good overview of why the separation of concerns is important and useful.

The MVC pattern has been the basis for many of the current Web application frameworks. Some notable examples include the Oracle Application Development Framework, Apache Struts [60], and Ruby on Rails [61].

5.1.2 MobiCloud DSL

The Metamodel used in constructing the MobiCloud DSL is in fact the MVC design pattern, thus Figure 5.1 also illustrates the Metamodel for the MobiCloud DSL. It provides constructs for each of the three key components: model, view, and controller. Each of these constructs act as place holders to collect details of the respective component. Following is a brief description of the major constructs of the MobiCloud DSL.

**Metadata** A collection of key-value pairs indicating meta-data associated with this application. There are no enforced meta-data values, but depending on the choice of the targets, certain meta-data values may be deemed mandatory.
Models  Defines each model with a name and a list of key-value pair attributes. The key-value pairs indicate the attribute name and the data type of the attribute. In this example greeting is the name of the model and it has one string attribute called message. A single script can include any number of models. The name of the model acts as a unique identifier for the model and is used as a reference in others sections of the DSL script. Models may translate to data objects on both the front-end and the back-end to represent the same data structure.

Controllers  Defines actions on models. These actions may be predefined or customized, as discussed later in this chapter. The generated controller components typically only affect the back-end, i.e. the controller is placed *behind* the service interface.

Views Defines GUI components, translated to the necessary code, that generate a suitable rendering on the targeted platform. The visual components of the views are implied from the action and the model the view is associated with. For example, a :retrieve operation implies that attributes of a model object needs to be displayed. Hence, the view contains labels (or other suitable components) to display the attribute values.

Extensions  A list of extensions that are active in the current script. Based on the active extensions, the interpretation of the models, views or controllers may be different.

Recipe Encapsulates all other constructs and acts as the wrapper for the constructs mentioned before.

We developed the MobiCloud DSL in stages. The first generation (MobiCloud I) consists of only the create, retrieve, update and delete (CRUD) operations. The second generation (MobiCloud
II) consists of extensions that enable a variety of additional capabilities such as the use of predefined models, views, or controllers and a wide range of custom operations. There is no difference in basic constructs between the generations, except for the *extensions* construct.

### 5.1.3 MobiCloud : First Generation

Listing 5.1 depicts a first generation MobiCloud script for a simple task manager application, intended to illustrate relationships between the components as well as the constructs of the DSL. It includes :

1. A *model* with four attributes, used to store task details (lines 4 to 7). This construct expresses the data structure that is needed to store the required application data.

2. A *controller* with two actions for creating and retrieving tasks (lines 9 to 12). Given that the data structure of interest is *task* (indicated by using the model name in actions in lines 10 and 11), the controller auto-generates the operations for creating and retrieving a task data structure.

3. Two *views* with basic user interfaces to add and retrieve tasks (lines 14 to 19). The views also assume, based on the referred action and the model, the appropriate UI rendering. For example, the create view includes text boxes (or equivalent UI component in the target platform) for string attributes.
Listing 5.1: The DSL script for the *task manager* application in MobiCloud. Extra line breaks have been
inserted for formatting.

```plaintext
recipe(:todolist) do
  metadata({:id => 'task-manager'})
  # models
  model(:task, {:name => :string, :description => :string, :time => :date, :location => :string})
  # controllers
  controller(:taskhandler) do
    action:create, :task
    action:retrieve, :task
  end
  # views
  view:add_task, {:models => [:task], :controller => taskhandler, :action => create}
  view:show_tasks, {:models => [:task], :controller => taskhandler, :action => retrieve}
end
```
Figure 5.2: Generated components for the Task Manager application. The model contributes to two equivalent data structures while the controller primarily contributes to a back-end service wrapper. The view mainly contributes to the mobile front-end portion, creating the UI.

### 5.1.4 First Generation Architecture

The first generation MobiCloud standard tools are based on the general architecture, illustrated in Figure 5.3.

As the first step of the generation, the DSL script is parsed and an object representation is created. This object representation is used inside a set of target specific template processors to generate the code. The template processors share some commonality, yet use templates specific to the target platform at the point of generating concrete code.
5.1. MOBICLOUD

5.1.5 MobiCloud: Second Generation

The second generation MobiCloud (MobiCloud II) added an extension capability to allow predefined models, views, or controllers to be available to the language, simply by requiring the extension in a script.

Listing 5.2 shows a partial code fragment from the second generation MobiCloud, exemplifying the use of an extension. In general, an extension adds specific capabilities to the generator. For the illustrated URL extension, the generators receive the ability to generate controller code to fetch contents of a URL, and optionally, assign values from the output to a specified model. Fetching the contents of a URL is a capability present in all cloud platforms but each platform carries its own quirk; for example, GAE requires the use of the Google URL fetch library to access external URLs while EC2 has no such restriction.
Listing 5.2: Using the URL fetch extension in MobiCloud. Extra line breaks have been inserted for formatting.

```ruby
# Generic HTTP fetcher
# exemplified using Yahoo's network time fetcher

recipe : http_fetch do
  # Enabling generic http extension
  extensions ['http']
  # metadata
  metadata({:id => "http_fetcher"})
  # models
  model : time_value, {:ts => :int}
  # controllers
  controller : time_manager do
    action : fetch_time, :time_value, {:type=>'http',
      :params => {:appid => 'xxx'},
      :return_mapping => {:ts=> '/Result/Timestamp'}}
    end
  # views
  view : view_time, {:model =>:time_value, :controller => :time_manager,
    :action => :fetch_time}
end
```
Listing 5.3: A Salesforce contact extraction application written using the MobiCloud DSL. The salesforce extension adds extra actions as well as predefined models. Extra line breaks have been inserted for formatting.

``` Jahreszeit | MOBICLOUD |
---|---|

```
5.1.5.1 Augmenting the Architecture to Integrate Extensions

Extensions are integrated to the generators via predefined hooks. The most common method of integration is to globally augment the semantic object model of the parsed DSL by inserting pre-defined models or views during the post model creation hook. For generation tasks that require specific code to be inserted, such as for controllers, targeted extension hooks are used. These are platform specific, i.e., they are required to follow the specific code guidelines for the respective platform. For example, the URL Fetch extension illustrated in the example modifies the controller templates in both GAE and EC2 to insert customized versions of a URL data extraction function. Extensions can also insert extra libraries to the respective projects. This extension mechanism is illustrated in Figure 5.4.

Figure 5.4: Augmented high level architecture to support extension processing. Extensions augment the parsed model globally or locally, and/or modifying specific code segments in the templates
5.1.5.2 Enterprise Integration

An important addition, possible via the extensions mechanism, is enterprise integration. One such featured extension is the Salesforce extension that allows one to integrate Salesforce.com data, such as a contact list. Salesforce [62] is a popular enterprise application platform to create business applications and provides many support services such as authentication, scaling etc. Salesforce integration requires a significant learning and debugging effort, especially the OAuth [10] based authentication mandated by Salesforce for selected content. Using the MobiCloud Salesforce extension saves significant time and effort in developing integrated applications by shielding developers from the intricacies of the OAuth mechanism. The extension adds the necessary libraries and user interfaces for authentication with Salesforce, augmenting the base controllers and views, as well as predefined data structures needed to extract Salesforce data.

Listing 5.3 illustrates the DSL code for an application that displays a Salesforce contact list on a mobile device. The following extra components are generated when this DSL is used with the code generators.

- All required data structures to represent a Salesforce contact. In this case, data structures for an organization and a contact are generated. The attributes for these data structures are defined by following the Salesforce service descriptions.

- The views for the corresponding Salesforce models.

- A view that acts as the front-end for the OAuth based authentication. When users try to perform actions that need authentication, they are automatically redirected to this authentication view.
• A controller component (a servlet in the case of a Java Web application) that handles the authentication. This controller caches the credential data following the OAuth protocol. It also acts as the callback endpoint for the OAuth handler.

• The necessary data storage provisions for the models as well as the credential caches.

Some extensions under development will add support for custom data type inclusions, UI customization and integration of popular support services including maps and social network integrations, etc.

5.1.6 Tools

The base MobiCloud toolkit consists of a command line compiler and code generator. Table 5.1 outlines the features and capabilities of the MobiCloud standard tool kit.

In a nutshell, the tool kit is capable of producing Android and Blackberry applications as front-ends and Google App Engine (GAE) and Amazon EC2 applications as back-ends, in the general case. The generated applications are readily deployable, either using the MobiCloud tools or using the respective cloud or mobile development kits (SDK).

5.1.7 MobiCloud Composer

Although the DSL itself is simple and concise, an MVC design becomes much more palatable when done graphically. Such graphical design also alleviates a significant portion of the language learning curve. Thus, we created two Web-based applications to compose and store MobiCloud applications.
<table>
<thead>
<tr>
<th>Feature</th>
<th>MobiCloud Generation</th>
<th>Supported Front-end Platforms</th>
<th>Supported Back-end Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for CRUD operations</td>
<td>I</td>
<td>Android (version 1.6 upwards) - The generated applications are compatible with the latest version of Android due to Androids backward compatibility</td>
<td>Google App Engine (Version 1.5.1) - uses the Google BigTable as the data storage and generates Java Data Objects (JDO) compliant annotated classes for data persistence. Amazon EC2 / Local server - Standard Java Servlet/JSP based application, using relational database. SQL scripts are generated to create and pre-populate the database, using MySQL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blackberry (version 6) - experimental</td>
<td></td>
</tr>
<tr>
<td>Friend of a Friend (FOAF) Extension</td>
<td>II</td>
<td>This extension activates only in the global extension processor (Figure 5.4), as a result supports all platforms</td>
<td></td>
</tr>
<tr>
<td>Provides access to standard models for Person and Organization data structures based on the FOAF specification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>URL Access Extension</td>
<td>II</td>
<td>Android (Version 1.6 upwards)</td>
<td>Google App Engine (Version 1.5.1)</td>
</tr>
<tr>
<td>Provides URL access capability. Supports XML responses and XPath based data extraction</td>
<td></td>
<td></td>
<td>Amazon EC2</td>
</tr>
<tr>
<td>Salesforce Extension</td>
<td>II</td>
<td>Android (Version 1.6 upwards)</td>
<td>Google App Engine (Version 1.5.1)</td>
</tr>
<tr>
<td>Provides access to contacts in a Salesforce account. Supports OAuth authentication through the browser as well as the mobile device</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Features and capabilities of the MobiCloud toolkit

The composer is a graphical tool that can be used to generate MobiCloud code using graphical components. Figure 5.5 illustrates the Web based user interface of the composer. Graphical icons representing model, view, and controller constructs can be dragged onto a canvas and connected to create the required configuration. The code is created on the fly and displayed below in the code window.
(a) MobiCloud composer user interface. The canvas at the center contains draggable widgets that are plumbed together to graphically depict their relationships.

(b) The graphical composition of the task manager application listed in Listing 5.1

Figure 5.5: Various views of the composer.
(c) Target Selection dialog, illustrating the different target options

(d) Different options for the user, once the targets are selected. They can download the source code or compile and deploy the application directly

Figure 5.5: Various views of the composer.
The advantage of this graphical UI is two fold. First, it alleviates the need to learn the syntactic details of the language. A composition can be made entirely in the graphical form, without writing any code. Use of graphical expressions makes it easier to visualize the application and hence results in faster composition. The second is that it facilitates other convenient additions such as direct deployment and packaging. The composer provides packaging for selected mobile platforms and also the direct deployment capability to clouds. For example, when the Android application is compiled via the composer, it creates an Android package (APK file) that can be installed in a compatible device.

The composer tools are available online for public use [63]. The current graphical composer is only MobiCloud I capable. A text based composer is available for MobiCloud II compositions.
5.1.8 MobiCloud Catalog

MobiCloud catalog is a storage and catalog solution for MobiCloud scripts. Users can store MobiCloud scripts, either manually or by exporting directly from the composer. The composer also has integrated features to search and import code from a catalog. The users can simply search for publicly available scripts and import them to the composer, without leaving the composer. A MobiCloud catalog instance is available for public use [64].
5.2 Scalable Cloud Application Generator - SCALE

Scalable Cloud Application Generator (SCALE) is an ongoing effort between Kno.e.sis computer scientists and their biomedical collaborators to provide a DSL based solution to create cloud-based statistical programs for bioinformaticians. SCALE addresses the implementation of statistical work flows for processing Nuclear Magnetic Resonance (NMR) spectroscopy data [65, 66].

5.2.1 NMR Spectroscopy

NMR spectrometer is a device that analyses biological samples using electromagnetic radiation. The physical phenomena in a nutshell is that when atoms are excited using a magnetic field, the atoms absorb and re-emit electromagnetic radiation in a frequency dependent on the properties of the atom and the strength of the excitement. Thus it is possible to look at the electromagnetic radiation signature and determine the chemical composition of a substance.

NMR spectrometers typically output a large amount of numerical data as a result of an analysis. The output is represented in a spectrum where peaks represent the response of a chemical to the strong magnetic field emitted by the NMR spectrometer. These spectra however, often include several thousand overlapping resonances, making it extremely difficult to identify individual peaks and compare them. NMR spectral data is subjected to a number of statistical processes before a human expert performs the peak identification process [11]. Note here that interpreting an NMR spectrum still remains a human oriented task. Almost all the computational techniques focus on noise reduction and peak amplification, ultimately reducing the complexity for the human expert.

In a nutshell, SCALE defines a DSL upon a number of base operators for NMR data process-
5.2. SCALABLE CLOUD APPLICATION GENERATOR - SCALE

The DSL implemented using a Ruby based DSL that reads like English to reduce the cryptic nature of the script. This script can be compiled into a Hadoop/Mapreduce program or a desktop program, based on the requirements of the scientist.

5.2.2 Comparing SCALE With Similar Approaches

While there are many successful generic workflow languages and tools (e.g. Taverna [67]), the nature of the typical processing task in metabolomics and similar eScience domains makes it difficult to use an off-the-shelf workflow solution. Following three aspects are important in this context.

1. Analysis is often performed over data sets of various sizes. These may range from a few hundred kilobytes to a few hundred megabytes. The scientists need a flexible approach that lets them use an appropriate type of computing resources depending on the data size.

2. Many institutions that perform analysis require the data be processed and kept in-house for legal reasons. This limits the processing options available to the scientist and makes service based workflow solutions difficult or impossible to use.

3. Almost all of these institutions have their own type of high performance computing infrastructure. The differences between these computing resources ultimately result in highly platform specific processing implementations. This is a serious hindrance for inter-institute collaboration.

SCALE is designed from ground up, to support the following characteristics.
5.2. SCALABLE CLOUD APPLICATION GENERATOR - SCALE

Provide domain specific abstractions, comprehensive to the domain scientists. One of the most important requirements of SCALE is to be comprehensive to the scientists so that they can compose the workflow by themselves. This gives the scientists the flexibility to modify the workflow as they see fit and also reduces the time to create it.

Platform agnostic, i.e. the created workflows can be converted to run on multiple platforms. Almost all solutions available to the scientist are focused on specific platforms or specialized tools. Writing an executable workflow that runs on a distributed computing platform like Hadoop is difficult, even to a seasoned programmer and clearly not an expertise a biologist would need. SCALE provides the ability to have one script and convert this script to functionally equivalent programs that run on distributed computing platforms or even a desktop.

Have minimal local tooling requirements. One of the most important issues is the required local tool set up. Many existing workflow solutions rely on heavy upfront tool installations and often, these installations are difficult to configure. The SCALE tooling efforts deliberately focused on Web based technologies so that the local tooling requirement becomes trivial.

5.2.3 Architecture

Similar to MobiCloud, SCALE DSL scripts are processed using a common parser generator infrastructure that creates an object representation of the constructs. This representation is then passed through the required platform specific generators to create the code. The generator core is designed in a way that facilitates adding a new platform specific generator. Figure 5.7 illustrates the high level architecture of SCALE.
5.2.4 The SCALE DSL

The SCALE DSL is based on the following base operators. These operators are categorized into families to highlight the presence of similar types of operations but using different techniques. Note that these operators are particularly relevant in the context of NMR data processing.

- **Normalization** \((N)\): This family of operators act on spectra on a per-spectrum basis to make the samples directly comparable to each other. For example, the most common operation in this category is sum normalization where the individual readings of the sample are divided by its sum to make the sum equal to 1. Other normalization techniques includes normalization by weight.

- **Correction** \((C)\): This family of operators remove errors introduced by measuring equip-
A common error in the NMR equipment is the slight variation of the baseline, commonly known as the *baseline shift*. Sub-operators of this family include baseline correction and phase correction.

- **Quantification** ($Q$): This family of operators reduce the dimensionality of the data and attempt to extract or approximate metabolite concentrations. Sub-operators of this family include binning and targeted profiling.

- **Scaling** ($S$): This family of operators control the weighting of features before a multivariate statistical or pattern recognition technique is applied. Sub-operators of this family include auto-scaling, pareto-scaling, and mean-centering.

- **Mining** ($M$): This family of operators selects the significantly responding metabolites/features for a given experiment. Sub-operators of this family include t-test, and partial least squares with variable selection.

- **Visualization** ($V$): This family of operators output a visualized representation of the data and/or results. Sub-operators of this family include principal component analysis and partial least squares scores plot.

- **Transformation** ($T$): This family of operators perform data transformations, such as Fourier transforms.

A typical data processing task can be composed by using these operators in a sequence. For example, the normalizing, binning and scaling a dataset can be represented as a composite function, as illustrated in Equation 5.1. $S$ and $S'$ are the input and output data sets respectively.
5.2. SCALABLE CLOUD APPLICATION GENERATOR - SCALE

\[ S' = Q_{\text{autoscaling}}(B(N_{\text{sum}}(S))) \]  

(5.1)

SCALE DSL consists of functions that resemble these operators. Only extra functions present are for the data reading and writing. Following is a list of some of the operators available.

**load_data**: Loads data from various sources and formats. The default behavior is to load data from the respective default file system. The format argument is optional and the default is comma separated values (csv) files.

**sum_normalize**: Sum normalize a dataset. Sum normalization refers to summing up the data rows (assuming the data of each sample is present in a row) and dividing each value by the sum. This forces sum of each row to be 1 (thus normalized).

**auto_scale**: Performs an auto scaling operation. Auto scaling is also a normalizing operation but it normalizes the standard deviation of the data. This is done by calculating the standard deviation of a row and dividing each value by this value. This makes the standard deviation of the values 1.

**store_data**: Stores the data in the respective file system. The current behavior is limited to file storage.

Listing 5.4 illustrates using the SCALE DSL to perform a sum normalization over a NMR spectral dataset. Note the function-like usage of the constructs where the intermediate values are held in variables.
5.2. SCALABLE CLOUD APPLICATION GENERATOR - SCALE  

Listing 5.4: Sum normalization implemented using the SCALE DSL

```ruby
# Load, do a sum normalization and store the results in a file
original_data = load_data(:raw_data_file, { :format => "csv" })

# sum normalize
normalized = sum_normalize(original_data)

# write the file
store_data(:normalized_data_file, normalized)
```

5.2.5 Tools

The base SCALE toolkit is a command-line program generator, capable of using a DSL script, such as the one in Listing 5.4, to generate multiple applications with the same functionality. The current SCALE toolkit supports three platforms.

**Desktop / Single computer**: SCALE can generate a desktop application that use the Ruby language. Ruby interpreters are available for all major operating systems. Also Ruby is a very readable language and makes it easier to modify the scripts if the need arises.

**Hadoop Cluster**: SCALE is capable of generating a Java application, based on the Hadoop framework. Hadoop is an open source distributed computation framework. The generated application can run on a local cluster or on a public cloud such as Amazon EC2 where Hadoop is well supported [18].

**Windows Azure**: SCALE can generate a C# application, based on the Daytona framework by Microsoft Research. Daytona is a distributed computation framework, similar to Hadoop, but built to run on the Windows Azure cloud platform.
5.2. SCALABLE CLOUD APPLICATION GENERATOR - SCALE

Figure 5.8: Overview of the SCALE Web based composer. The panels resemble a familiar IDE environment. This composer is based on the MobiCloud composer and shares some UI and back-end components.

These three platforms offer scientists a good balance of the computational coverage. They can try out a smaller dataset in their desktops and then submit a larger dataset to a cluster of choice, either internally or externally.

5.2.5.1 SCALE composer

In order to support the goal if minimum local tooling, the primary composer for SCALE is developed as a Web application. The scientists can compose the work-flow, download an appropriate implementation and optionally deploy to a supported public cloud, all without having a specific local tool set. Figures 5.8 illustrates the user interface of the SCALE composer, where users can compose their code by dragging and dropping icons representing operators, similar to MobiCloud. Figure 5.9 illustrates a composition.
Figure 5.9: SCALE Web based composer depicting a typical composition of loading, sum normalizing, auto scaling and writing back a dataset
5.3 Evaluation

Following evaluations are performed.

1. A code metrics evaluation of DSLs and generated applications. These code statistics are primarily lines of code (LOC) counts, highlighting the human effort needed to create them manually.

2. A code metrics evaluation of the generation mechanism. These primarily include the LOC counts for the templates and the resource files that get copied verbatim.

3. A timing test with three versions of the generated applications for a selected SCALE DSL script. The objective of this test is to highlight the advantage of using the cloud for large datasets.

4. Subjective evaluation of the tools obtained by exposing the tools to the community.
5.3. EVALUATION

5.3.1 Evaluating MobiCloud Generated Applications

This section includes the code metrics relevant to the MobiCloud project. Table 5.2 includes the code metrics for 4 different MobiCloud applications. Figure 5.11 illustrates these statistics as a graph.

<table>
<thead>
<tr>
<th>Script</th>
<th>Description</th>
<th>DSL Lines of Code</th>
<th>Models</th>
<th>Views</th>
<th>Controllers</th>
<th>Target</th>
<th>Generated Lines of Code (Java, JSP and XML)</th>
<th>Ratio of DSL to Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Manager</td>
<td>A task manager application that stores and retrieves tasks (todo items)</td>
<td>11</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td>541 Blackberry 329 EC2 156 GAE</td>
<td>1:49</td>
</tr>
<tr>
<td>Shop Manager</td>
<td>An application to keep track of jobs and customers for a mechanics shop</td>
<td>17</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td></td>
<td>1244 Blackberry 592 EC2 628 GAE</td>
<td>1:73</td>
</tr>
<tr>
<td>URL Fetcher</td>
<td>Fetches and displays timestamp values from a Yahoo Web service</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>486 Blackberry 100 EC2 289 GAE</td>
<td>1:54</td>
</tr>
<tr>
<td>Salesforce Contacts</td>
<td>Fetches and displays the contact list from a Salesforce account</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>794 Blackberry - EC2 - GAE</td>
<td>1:88</td>
</tr>
</tbody>
</table>

Table 5.2: Selected code metrics of MobiCloud applications

The LOC counts indicate that consistently the code generation has a significant (at least 11 to 1) advantage over *hand-coding* the application. This is consistent with the expected advantage of using a DSL. This effort advantage improves significantly when complex applications are considered, especially where extensions are in use. In the selected examples, the Salesforce extension generated the largest (and the most complex) code output, giving a 153 to 1 advantage over hand
coding the application. This illustrates the advantage of using the DSL for complex tasks where the abstractions provide transparency over the actual complexity of the code.
5.3.2 Evaluating MobiCloud Generation Mechanism

Table 5.3 shows the code metrics of the MobiCloud code generation templates. The same statistics are illustrated in Figure 5.11.

<table>
<thead>
<tr>
<th>Target Platform</th>
<th>Template LOC</th>
<th>Resources LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>1162</td>
<td>0</td>
</tr>
<tr>
<td>EC2</td>
<td>519</td>
<td>10</td>
</tr>
<tr>
<td>BlackBerry</td>
<td>380</td>
<td>0</td>
</tr>
<tr>
<td>GAE</td>
<td>789</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 5.3: Template and Resource LOC counts for MobiCloud

These statistics indicate that the effort required to create a template in average (about 720 lines) is only slightly higher than the effort of creating a typical (non-trivial but not complex) application (roughly around 540 lines of code as per the previous statistics). Note that in this case, much of the templates are dynamic, thus the resource LOC counts are negligible.
5.3.3 Evaluating SCALE Generated Applications

This section includes the code metrics relevant to the SCALE project. Table 5.4 includes the code metrics for 3 different operators. Figure 5.12 illustrates these statistics as a graph.

<table>
<thead>
<tr>
<th>Operation</th>
<th>DSL</th>
<th>Windows Azure</th>
<th>Hadoop (Local cluster/ EC2)</th>
<th>Desktop (Ruby)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Code</td>
<td>Ratio</td>
<td>Code</td>
<td>Ratio</td>
</tr>
<tr>
<td>Sum Normalization</td>
<td>3</td>
<td>847</td>
<td>1:282</td>
<td></td>
</tr>
<tr>
<td>Auto Scaling</td>
<td>3</td>
<td>848</td>
<td>1:283</td>
<td></td>
</tr>
<tr>
<td>Sum normalize then auto scale</td>
<td>4</td>
<td>938</td>
<td>1:235</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Selected code counts for SCALE operators

![Figure 5.12: Lines of Code statistics of SCALE generated applications](image)

Similar to the MobiCloud case, the code output far outweighs the effort to create the DSL script. This advantage is readily visible in the more complex cases (such as Hadoop where the code ratio ranges from 1:255 to 1:318) than the desktop version (1:26 to 1:31).
5.3.4 Evaluating SCALE Generation Mechanism

Table 5.5 shows the code metrics of the SCALE code generation templates. The same statistics are illustrated in Figure 5.13.

<table>
<thead>
<tr>
<th>Target Platform</th>
<th>Template LOC</th>
<th>Resources LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure</td>
<td>454</td>
<td>736</td>
</tr>
<tr>
<td>Hadoop</td>
<td>434</td>
<td>886</td>
</tr>
<tr>
<td>Ruby</td>
<td>133</td>
<td>90</td>
</tr>
</tbody>
</table>

Table 5.5: Template and Resource LOC counts for SCALE

Even in this case, the effort in creating the templates (and the resources) is roughly 120% to 150% the effort of writing one program. For example, in case of Hadoop, the total LOC for both the templates and resources is about 1300 and the typical LOC for a generated program is around 1000.
5.3.5 SCALE Generated Application Performance

Apart from the code metrics, the performance of the generated applications was also measured.

Table 5.6 illustrates these runtimes for SCALE applications. MobiCloud applications were not scrutinized for their runtimes since there is no heavy back-end processing intended by them.

<table>
<thead>
<tr>
<th>Task</th>
<th>File size</th>
<th>Hadoop (Local Cluster)</th>
<th>Ruby</th>
<th>Windows Azure</th>
<th>File</th>
<th>File</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>File handling</td>
<td>File handling</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>processing</td>
<td>processing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum normalization,</td>
<td>2.8</td>
<td>2</td>
<td>104</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>then auto scale</td>
<td>76</td>
<td>5</td>
<td>165</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>152</td>
<td>6</td>
<td>185</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>76</td>
<td>5</td>
<td>165</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>152</td>
<td>6</td>
<td>185</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: Selected Running times of SCALE applications, based on the input file size. All file sizes are in Megabytes and timings in seconds

For smaller file sizes, the local processing is faster. This is evident by the 2 seconds local processing time for a 2.5MB input file. The delay may very well be the distribution overhead which offsets the cluster advantage. However, when the file size increases, the local processing time exponentially increase. This is when the cluster advantage becomes clear. For example, the processing time changes only by 20 seconds when the input file size doubles from 76MB to to 152MB.

Windows Azure timings are higher than of a local cluster, which may very well be due to network latencies and other communication overhead.

These results verify that there is a definite advantage in using cloud based solutions. Also the flexibility in using the cluster based processing only when necessary improves the usability of the software.
5.3.6 Discussion on Code Metric Results

The code metric results can be summarized as follows.

1. Generating code is advantageous within the scope of a domain. The actual effort saving depends on the target. This is the expected behavior in using a DSL in the first place.

2. The effort to create the templates and the resource files for the code generation mechanism are manageable in the considered domains. This justifies the use of a DSL and the effort to create the DSL driven generation mechanism.

It should be noted that LOC is not the most representative measure of coding effort. There are other metrics, such as cyclomatic complexity that better represent the code complexity. As discussed in Section 4.2.4.2, LOC can be calculated easily and rapidly and provides a reasonable (although not the most accurate) indication of the effort required.

In these experiments, the most relevant effort that is not accounted for is the effort in algorithm conversion to parallel architectures. This effort is typically encountered when building the templates. Also during every code generation, the users automatically get the benefit of this conversion, yet it is not quantified by the LOC counts in the generated code.
5.3.7 Evaluating The MobiCloud Experience

After exposing the MobiCloud tools to the developer community (via the online tools as well as a demonstration in a leading conference on cloud computing) two concerns were highlighted.

5.3.7.1 Action and UI Customizations

The first concern is regarding the inability of MobiCloud to customize either the UI or the back-end actions. Many mobile developers considered the lack of UI customization a serious lack of capability. Addition of extensions capable of augmenting the model, view or controller functions is primarily a response to address this limitation.

5.3.7.2 Exploiting Target Specific Features

By the very nature of design of MobiCloud, it is not meant to explicitly exploit target specific features or specify custom code to be inserted on specific target platforms. For example, if the developer wants a specific library call to be made when a particular cloud, say GAE, is targeted, such code has to be inserted after the initial generation process. The problem however is that when such code is not part of the original specification, it may become a source of incompatibility at a later stage of development. Inserting platform specific code also limits the ability of the DSL to be truly platform independent.

5.3.7.3 Favorable Properties

MobiCloud was observed to have the following favorable properties.
5.3. EVALUATION

1. The DSL is simple and no heavy upfront design is required. This approach meshes well with Agile development techniques which value quick iterations over heavy up front designs.

2. The tool performance is sufficient to do rapid prototyping. The typical code generation task takes only a few seconds.

3. The generated code may be used as boilerplate, i.e., developers can use MobiCloud to simply get rid of the repetitive programming and focus on the more creative aspects, such as customizing the UI.

HTML 5 has come up as a strong alternative to cross platform applications. However, HTML 5 still lacks strong platform integration features and is not able to replace the user experience of a native mobile application. Thus it is safe to assume that a tool like MobiCloud, with native front-end generation capabilities, will be relevant, in spite of advanced Web standards such as HTML 5. MobiCloud can be simply extended to generate an HTML 5 front-end, if such a capability is desired.

5.3.7.4 The Case of the Smallest Common Subset

During initial public exposures of MobiCloud, many developers have raised the issue of just supporting the smallest common subset of features and thereby limiting the usefulness of the DSL. This aspect however, does not limit the capabilities of the DSL severely in practice because most of the generally used features not directly present can always be simulated or approximated in an indirect way during the customized compilation process. In other words, all mobile platforms support features that are deemed essential and the code generators can include graceful degradation.
of application features when the target platform does not have the required hardware or software capabilities. Such a strategy also applies to the cloud platforms that support features with similar semantics, yet are different syntactically and structurally.

### 5.3.8 Evaluating the SCALE Experience

SCALE toolkit was subjectively evaluated by a set of researchers at the bioinformatics division of a research lab. These researchers were from a variety of backgrounds, ranging from biology to computer science.

#### 5.3.8.1 Absence of Operators

Many observers noted the absence of some useful operators. Analogous to the case of the smallest common subset, it was observed that some operators, such as OPLS, are more difficult to convert to parallel algorithms, and when converted, result in an inefficient implementation. While this limits the potential of this approach, it is still possible to make use of the parallelization capabilities of distributed computing platforms. The initial implementation focused on only the operators where parallelism can be exploited.

#### 5.3.8.2 Full lifecycle Support

Another important aspect was the need of integrated deployment and support utilities. Since the use of the tools was in the hands of non-computer experts, it was deemed extremely important to have a full working solution generated and deployed directly through the user interface.
5.4 Chapter Summary

This chapter discussed the details of two projects that used abstractions to implement program generation for clouds.

- MobiCloud, the cloud-mobile hybrid application generator, includes the generation capabilities to support two mobile platforms and two cloud platforms. MobiCloud addresses a specific application generation need in the consumer space.

- SCALE is a conceptually similar project that applies the same principles to create large scale data processing programs.

- Examining the code metrics in both these projects provide evidence that it is advantageous to use DSL driven development in these domains.

- Subjective evaluations by users reveal that there are other considerations if these projects are to be accepted to become mainstream.
Platform Agnostic Cloud Application

Deployment and Management

This chapter discusses, in detail, the use of middleware in cloud computing to achieve platform agnostic application deployment and management. Unlike the previous chapter, the theoretical aspects of Middleware are only discussed briefly. The major portion of this chapter is dedicated to describe the Altocumulus cloud middleware system.

6.1 Preliminaries

6.1.1 What Actions Benefit from Abstractions?

There are many activities that take place in a cloud service interaction. While most of these activities are dictated by the cloud platform, there are two main categories of activities required in an application oriented context.

Application Deployment: This refers to placing an application in the cloud environment. Some cloud environments (PaaS) offer direct operations for deploying an application while in an
infrastructure cloud, many operations precede the activity of the actual application placement. This heterogeneity is a significant barrier to entry for amateur cloud users.

**Application Management**: This refers to the on-going activities performed on an application that is already placed inside a cloud environment. Some of these activities include periodic tasks like taking data backups as well as load based tasks such as scaling.

These two categories of activities are the most important in the general cloud application life-cycle (Figure 3.1, discussed in Chapter 3), thus this chapter focuses on providing abstractions relevant to these activity categories.

Note that there is a tacit assumption about the presence of a service interface. While a cloud platform, by definition, provides a programmatic interaction method, not all clouds expose their actual service API to the public. For example, GAE provides a toolkit that wraps the Google specific service interfaces. In this case, although direct service calls are not possible, the tools can be further wrapped to provide a similar experience.

Also note that there is an assumption about the human involvement in these activities, either by using a human interface or a programmatic interface. Thus, the objective of the middleware is to support the human in performing these complex tasks in a convenient and repeatable manner. The term *human* is used casually to refer to technical users capable of understanding the flow of the operations at a higher level but are not experts that understand intricate details.
6.1.2 Placement and The General Architecture of Cloud Middleware

A middleware layer is capable of providing a uniform interaction protocol to multiple cloud interfaces and includes the ability to deploy, and manage cloud applications using a uniform interface.

The structure and the general architecture of this type of middleware is illustrated in Figure 6.1.

The middleware layer includes adapters that are capable of interacting with specific cloud service interfaces. These adapters are developed using a common interface facing the middleware internals. Some of these adapters may also have wrapped a composition of services to compensate to mimic the required functionality. The core of the middleware simply dispatches the operations to the required cloud interface by invoking the relevant methods in the cloud adapter.

The user facing components include service and/or human interfaces to deploy and manage
applications. The deployment interface would require the application as a bundle (i.e. all the components required for the application is a single file). This may be an application specific archive, say a Web Archive (war) file or a middleware specific archive format.

Note that the major advantage of this type of architecture is the relative convenience of adding support to a new cloud interface. The addition of a new cloud merely requires a new cloud adapter. This leads to a linear growth in the required effort, rather than a quadruple growth. In other words, this reduces the complexity in catering for new clouds since the effort required is focused on creating a single adapter.

6.1.3 Challenges

Similar to the DSL issues, there are a number of challenges in taking this approach that can be categorized under the following broad topics.

1. **Applicability of Middleware**: The applicability of middleware is determined by the types of operations that is required and permitted by the clouds. Unlike the case of the DSLs, the operations are determined by the available (and permitted) operations and their composability from the cloud interfaces. For example, in a users perspective, the operation they want to accomplish is getting an application up and running in a cloud environment. This is not very complicated in a PaaS cloud environment. However for an IaaS cloud environment, the middleware should select a suitable machine image and set up the necessary base software to *install* the application. Thus, the operations being abstracted need to have equivalent operations or compositions from the cloud system. While this is possible in most cases, there are special considerations to be made when interfacing certain cloud systems.
2. **Usability of Middleware**: Similar to the case of the languages, the usability of the middleware is determined by the balance of complexity and completeness. This is an important consideration given our assumption on the involvement of the human in designing and executing the process. For example, if an activity requires 30 parameters to be supplied, the middleware would be deemed unusable if all these parameters were presented to the user. Instead, it should provide sensible defaults and categorizations such that the complexity can be managed by a human.

How and to what extent these challenges can be overcome are discussed in detail in Section 6.2.

### 6.1.4 Repetitive Operations: Using Patterns

An advantage in using a middleware to deploy applications is the possibility of enforcing a known pattern. The idea of such a pattern is to easily replicate a configuration that is known to work and repeat them when required. These patterns have the following properties.

1. **Ability to store a number of data items** Certain cloud environments require a large number of configuration parameters, thus the pattern should be able to provide enough place holders to store the necessary data values. These data values may be logically segregated. For example, access credentials may be stored under a credentials collection and scaling rules may be stored under management rules.

2. **Contain reasonable defaults** Given that the number of parameters can be quite large, it is inconvenient, if not impossible, to supply all of them during an interaction. It is essential that
sensible defaults are in place. These defaults may also be dynamic, i.e. changed according to the relevant high level activity. For example, the default machine image used in a deployment may be changed based on the type of the application being deployed.

3. **Comprehensive** Given that these patterns are created, maintained and used by humans, the organization of the pattern should be reasonably comprehensive to a technical user.

## 6.2 Altocumulus - Applying Middleware to Deploy Cloud Applications

Altocumulus is an IBM research cloud middleware layer that provides abstractions for a set of critical operations on multiple cloud platforms [68, 69]. Altocumulus was successful in providing a manageable deployment and management solutions for IBM internal use. IBM Workload Deployer [70, 71], part of its smart cloud initiative, is a direct descendant and productized version of the research project, discussed briefly later in this chapter.

IBM Altocumulus uses a distributed architecture that divides the platform into three components:

- **Dashboard** is the principal interaction points for end users. There, users can: a) select the best practices that they want to use, b) provide credentials for the different clouds to target, and c) deploy and manage their applications into different clouds.

- **API and API Tester** is a Web application that exposes the Altocumulus REST API and gives an easy UI to invoke the API and browse documentation about it.
Core is the back-end orchestrator of the platform. It implements the API which is how the Dashboard (and other Altocumulus applications) can communicate with the Core. The Core contains an elaborated set of scripts, rules, and cloud adapters to allow execution of actions seamlessly across different clouds for different users.

Figure 6.2 illustrates this architecture.

Figure 6.3 illustrates the screen of deployment view. Although deployed in completely different clouds and the operations need to be different, these deployments appear similar in a users perspective.
6.2. Altocumulus Platform Support

Following cloud platforms are supported in Altocumulus.

**Amazon EC2 (and clones)** Amazon EC2 is one of the earliest cloud platforms and has a comprehensive service interface. The early entry of Amazon to the cloud space has prompted many communities to adopt the Amazon API as a de-facto API and build clones of the EC2 system. A prominent system in this regard is Eucalyptus [72], a complete open source cloud system that any organization can take and use to build an EC2-like cloud. Altocumulus included support to EC2 and EC2 clones via the EC2 Ruby library. While the Ruby library provided convenient programming abstractions, this library uses service calls underneath to perform the operations.

**Google App Engine (GAE)** The tools for GAE are executables and included as part of its stan-
standard development kit. There is no clear service API provided. Thus, Altocumulus created programming abstractions by wrapping the command line tools with a library. This approach is the most convenient to implement, yet may not be scalable in a large scale due to high system resource requirements. For example, each deployment job starts an invisible shell in the server, which takes significantly more system resources than making a service call.

**IBM HiPODS (High Performance On Demand Solutions) Cloud** HiPODS cloud was the IBM private cloud solution (All IBM private cloud efforts are now unified under the IBM PureSystems) and performed similarly to EC2. There are however two major differences.

1. The HiPODS cloud did not have a detailed published service API. The typical way to interact with it was through a dashboard application. Altocumulus reverse engineered the dashboard application (which used JSON messages to communicate) and wrapped it in a programmatic interface to build an adapter.

2. The HiPODS philosophy included a reservation time period feature for computing instances. The time requirement of a computing instance had to be specified upfront and the instance automatically terminates when the time period expires. Altocumulus used a periodic update of the time periods for HiPODS instances, giving the users the illusion of EC2-like functionality.

Apart from these major differences, the HiPODS instances behave in the same manner as the EC2 (or any other infrastructure cloud) instance.
6.2.2 Altocumulus Operations

Altocumulus supports the following operations.

**Deploy application bundles** The idea of *deployment* is to make the application operational in the given cloud environment. In many PaaS clouds, this is simply executing a command using the provided cloud interaction tools. In most IaaS, this requires the installation of a number of support software components and executing scripts to configure them, before the application bundle is placed in the virtual machine.

**Perform management operations** The types of management operations primarily included data snapshots and status checks. Creating images (taking a snapshot of the virtual machine) is supported only on IaaS clouds. Similar to the case of deployment, management operations in some clouds (primarily PaaS clouds) are mostly one or more commands in the provided interaction tools. In infrastructure clouds, there are no standard tools to mimic these functions, thus Altocumulus inserts scripts into the virtual machines to support some of these operations. For example, in case of EC2, a script in inserted into the virtual machine to use the mysql\*dump tool (in case MySQL is used as the database) to generate data backups.

**Undeploy applications** Undeploying an application is making the application inactive. In case of an IaaS cloud, the instance can be terminated, removing all traces of the applications execution (If data and other accumulated information such as logs are required afterwards, they need to be preserved, either by extracting them explicitly or taking an image of the virtual machine). In case of PaaS, this was achieved by inserting a simple no-op application in the original application space.
6.2.3 Other Altocumulus Features

There are other Altocumulus features that makes it convenient to manage large deployments.

Social Features The deployments are either private or public, in terms of visibility. For a public deployment, users other than the deployer can check the status and the configuration of the deployment and even add comments. This makes it convenient in either the public or corporate level where the users have the flexibility to check the health of a deployment. A public deployment may not have all its actions exposed publicly unless the deployer specifies them to be.

Philosophy of Credential Bundles Credential Bundles are simply collections of usernames, passwords, certificates and/or keys used to access the clouds. Since these are reused many times, Altocumulus offered a feature for users to bundle them and reuse them as needed. A credential bundle is a collection of key-value pairs, the value being either a string or a binary object, such as a certificate.

Cost Estimation Comparison Altocumulus also offered a cost comparison feature where the users can manually compare the costs in using a particular cloud. The cost model used was not dynamic and could not track the cost of a deployment. However, the users could reasonably estimate the cost of a deployment if the parameters of the deployment were well understood.

Section 6.2.4 describes, in detail, the most important feature, the best practice model.
6.2.4 Altocumulus Best Practice Model

Altocumulus introduced the idea of a best practice to cloud deployments. This best practice model is designed around three principles that stem from the properties of patterns discussed in Section 6.1.4.

1. **Provide logically arranged place holders for different types of data required for a cloud deployment.** This principle is based on the fact that there are significant number of different data items that is needed for a deployment. Although these data items may vary significantly between different deployments, they can be logically grouped and arranged.

2. **Be applicable to different cloud paradigms.** This principle states that such a grouping needs to be compatible with the different cloud paradigms, specifically IaaS and PaaS types.

3. **Have a reasonable balance between completeness and complexity.** This principle states that this grouping needs to be reasonably complete but not overly complex since it is meant ultimately for a human. This is especially important considering that various components of a best practice would evolve over time, i.e. will have different versions. The goal here is that a user would be able to redeploy a cloud application with an updated best practice using a one-click action. This can be significantly important when a new version of a best practice component is released to cover a vulnerability, e.g. a patch solving a security vulnerability.

An Altocumulus Best Practice (BP), in its simplest form is a particular sequence of process steps (selected either automatically or manually) that best suits the cloud management task at hand. A BP however, differs from a simple workflow since it may additionally include provisions to define other aspects of a deployment such as scaling. A BP consists of two primary components.
6.2. ALTOCUMULUS

- **Topology**: An indication of the structure of a particular deployment. In the simplest form, the topology can be a single instance or an application space. However, the topologies can be arbitrarily complex. In section 6.2.5, several common topologies are discussed.

- **Scaling Strategy**: Encapsulates the process for scaling. For some clouds, such as platform clouds, the scaling strategy would be very simple. However, for infrastructure clouds (IaaS providers), the scaling strategy includes details on what resources to manage when a scale-out or scale-in operation is required. The scaling strategy is tightly coupled with the topology.

The Scaling Strategy includes customizable *rules* that can be used to control the behavior of the scaling process. For example, for a strategy of scaling horizontally by replicating, the rules would control the load limit to spawn a new replication and the maximum number of replications allowed. These limits may be set by a user or a process.

The Topology is based on *instance groups*. An instance group refers to a set of functionally equal compute nodes or a cluster. Each instance group attaches to a *configuration bundle* that in turn includes an ordered set of *configurations*. A *configuration* represents a single step in a simplified work flow of setting up the node. For example, installing a database software is a configuration. Configurations (optionally) have parameters, most of them user tunable. To use the above example, the default user name and password for the database would be part of the user tunable configuration parameters. Figure 6.4 illustrates the structure of a Best Practice in UML notation.

While some simple BPs have only one instance, advanced BPs allow deployments to be spread across groups of instances that can grow and shrink according to some rules. Due to the complexity of building a custom BP from scratch, however, the Altocumulus platform contains a number of pre-built BP templates for the most common cases. Regular users are allowed to customize these BP
templates and only privileged users can construct them from scratch. We further discuss the issues involved in custom BPs in section 6.2.5.4.

### 6.2.5 Example Altocumulus Best Practices

This section presents three examples that clearly illustrate the flexibility of the best practice model and its coverage of the design principles stated in section 6.2.4. Some of the pain points these BPs try to overcome is also presented.

#### 6.2.5.1 A Single Instance Best Practice

This is the most simplistic BP where all the configurations apply to a single instance. There is no scaling strategy. The topology is in its simplest form where one instance contains everything. To cater for such simple scenarios, the Altocumulus platform includes an empty scaling strategy and a fixed topology.
A popular use of compute clouds, especially among the scientific researchers, is to use them for parallelizable computations using the map-reduce paradigm [73]. Apache Hadoop [19] is one of the popular map-reduce computation frameworks used for large computations. Typically when setting up a map-reduce task the major pain point is setting up the management nodes and the worker nodes (worker nodes are generally large in numbers). The number of nodes for a computation task is generally fixed and does not fluctuate during a computation.

The fixed cluster BP supports this type of set up where the nodes may be heterogeneous but fixed in number. Currently this BP supports only the Hadoop framework and includes a multi-instance star topology with 1) A central management node and 2) A number of worker nodes. The scaling strategy is empty since there is no dynamic scaling involved. This type of setup is illustrated in figure 6.5a.

A typical framework based (such as Apache struts [60], Ruby on Rails [61] and Python Django [74]) Web site is two tiered, i.e. they have a presentation front-end that displays content fetched from a database back-end. Majority of public Web sites such as wikis, forums or blogging platforms are structured this way. The popular open source multi purpose Web platform Drupal [75] is one of the prominent examples for a two tiered Web application platform, also called a Content Management System (CMS). Scaling strategies for such two tiered Web applications have been well studied and one of the accepted strategies is to replicate the presentation component under a load balancer. The pain points in such a scaling process includes replicating the complete configuration of the
presentation component which is typically a script language based application, and updating the load balancer configuration with the details of newly added (or removed) instances.

This BP includes a multi-instance layered topology that includes 3 layers

1. A load balancing layer

2. A presentation / Application Server (AS) layer

3. A Database layer

The scaling strategy is horizontal replication and includes rules that trigger replications. Figure 6.5b illustrates this architecture.
6.2. ALTOCUMULUS

(a) Star Topology for Hadoop

(b) Layered Topology for Horizontal scaling

Figure 6.5: Illustration of Different Best Practices
6.2.5.4 Custom Best Practices

Ultimately the intention of the best practice model is to provide cloud users with reasonable power and flexibility in cloud application management. However creating an advanced BP template needs a significant level of knowledge and testing. For example creating the fixed cluster BP described in section 6.2.5.2 requires in depth understanding of the Hadoop framework. Due to this reason the BP creation capability in altocumulus was limited only to privileged users. The regular users can always customize existing templates. The use of BPs, even with this limitation, has proved to be highly useful.

6.2.6 Feedback on Altocumulus

Altocumulus was demonstrated in several conferences and internally in IBM via the IBM Technology Adoption Program (TAP). TAP deployment and the demonstrations helped gather a important feedback in applying a middleware platform in the cloud context.

1. Altocumulus cannot address application lock-in. During the TAP deployment, many users attempted deploying incompatible applications to cloud platforms (say an incompatible web archive file to GAE). Altocumulus have no provisions to warn against such incompatibility and was only able to give a failed deployment message. Even worse, some applications failed at runtime due to these incompatibilities giving no hints to the user that the application was in fact incompatible during the deployment.

This underscores the importance of the ability to understand the internals of the application and possibly the ability to transform it at a middleware level. In fact, the experience from
Altocumulus played an important role in starting this research to find a completely new application development paradigm to generate portable cloud applications.

2. Hybrid cloud deployments are an important use case to address. Many users with enterprise experience discussed the importance of the ability to use a mix of public and private clouds given the mounting privacy and data protection considerations. Altocumulus can be used to manage such a hybrid deployment over multiple clouds (via an appropriate best practice), yet the deployer need to have clear understanding of the different components used in the deployment.

3. Cost considerations are important. Many users preferred an up-front cost estimate (with minimum user input) for a given application. The cost modeling in Altocumulus was useful but not comprehensive. Users preferred to have the ability to do cost estimation using an application profile, before it was deployed.

6.2.7 Productization of Altocumulus

The philosophical ideas Altocumulus and some of the functionality are incorporated into the IBM PureSystems private cloud system [76]. IBM PureSystems is a complete hardware and software solution to support the enterprise private cloud computing niche.

Altocumulus functionality is integrated into the PureSystems software stack for the application deployment and management work-flow. It is included as part of the IBM workload deployer (IWD) [71]. IWD is a complete appliance by itself with bundled software where the following features are influenced by Altocumulus.
6.3. **CHAPTER SUMMARY**

- Pattern based application deployments.

- Multi cloud (provider) deployments. Each cloud environment is called a provider within IWD.

- The concept of *bundles*. These primarily include software bundles and credential bundles.

Extensive details about IWD and its functionality is included in the IWD technical manual [71].

### 6.3 Chapter Summary

This chapter discussed the details of the cloud middleware system *Altocumulus*.

- Altocumulus has the ability to support selected cloud operations to three different cloud platforms, namely Amazon EC2, GAE and IBM HiPODs.

- Altocumulus promotes pattern based deployments by introducing the idea of *best practices*.

- Some Altocumulus features are incorporated into the software stack in the commercial product, IBM workload deployer.
Conclusion

This dissertation presented an abstract driven methodology to develop, deploy and manage cloud applications in a platform agnostic manner. This work focused on addressing these tasks in a domain specific context, using DSLs and middleware as the core technological bases. Three projects, IBM Altocumulus, MobiCloud and SCALE are presented in detail, evaluating different aspects of this research.

The following lessons are learned by the MobiCloud and SCALE projects.

1. There are theoretical limitations in applying DSLs to cloud application development.

2. DSL based software development is feasible in the cloud context, applied within the scope of a domain.

3. DSL based software development yields clear advantages, primarily in terms of user effort, even with the limitations.

4. Perceived usability and the presence of tools is important in adopting a DSL in practice.

Lessons learned by the Altocumulus project are
1. Middleware based operational abstractions are feasible in the cloud context.

2. These abstractions can be packaged for reuse, reducing the user complexity. Such packaging also improves quality and reduces mistakes.

3. These abstractions are useful and are incorporated into commercial cloud related software products.
Bibliography


