FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Deploying a Web Service Runtime Environment into the Cloud

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Anton Korosec, BSc
Matrikelnummer 0325672

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.Ass. Mag. Philipp Leitner

Wien, 01.12.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Anton Korosec
Tellgasse 14/11
1150 Wien

Abstract

This thesis evaluates the benefits and drawbacks of deploying an existing server application into a commercial cloud computing offering.
The 'Vienna Runtime Environment for Service-oriented Computing' (VRESCo) is being developed on the Institute of Information Systems at the Vienna University of Technology and is a Web service runtime environment, which addresses some of the current challenges in the Service-oriented Computing research (e.g., dynamic binding/invocation, metadata support).
The goal is to deploy VRESCo onto the Amazon Web Services (AWS), which offer a set of services for a computing platform in the cloud. Not only should the application run on a virtual machine instance within the Amazon cloud, but the current database should also be replaced by one of Amazon's cloud database services.
Besides several drawbacks (e.g., latency, database limitations) it is expected that VRESCo will overall benefit from the advantages that a cloud architecture has to offer (scalability, no administration, low costs, etc.).
After an introduction into the used AWS services and their underlying concepts, the steps undertaken to deploy VRESCo into Amazon's cloud architecture are presented. It has especially been focused on the comparison and evaluation of the various cloud databases offered by Amazon and the techniques (caching, lazy loading, etc.) that were used in order to achieve the best performance.
Overall, the results show that AWS is suitable for building proper cloud architectures, which are capable of substituting the need for local resources. The services are not only easy to use, but also flexibly customizable to the application's needs.

Kurzfassung

Diese Diplomarbeit untersucht die Vor- und Nachteile des Einsatzes eines kommerziellen Cloud-Computing Angebots bei einer bestehenden Server-Anwendung.
Das 'Vienna Runtime Environment for Service-oriented Computing' (VRESCo) wird auf dem Institut für Informationssysteme der Technischen Universität Wien entwickelt und ist eine Web-Service-Laufzeitumgebung, die sich einigen der aktuellen Herausforderungen im Bereich der service-orientierten Architekturen (z.B. dynamisches Binden/Aufrufen, Metadaten-Unterstützung) annimmt.
Das Ziel ist es, VRESCo mit Hilfe der 'Amazon Web Services' (AWS), die eine Reihe von Diensten für eine Rechenarchitektur in der Cloud bereitstellen, anzubieten. Die Anwendung soll nicht nur auf einer virtuellen Maschinen-Instanz innerhalb der Amazon-Cloud laufen, sondern es soll auch die aktuelle Datenbank durch einen der von Amazon angebotenen Cloud-Datenbank-Dienste ersetzt werden.
Neben einigen Nachteilen (z.B. Latenz, Datenbank-Einschränkungen) wird erwartet, dass VRESCo insgesamt von den Vorteilen, die eine Cloud-Architektur mit sich bringt (Skalierbarkeit, keine Verwaltung, geringe Kosten, etc.), profitiert.
Nach einer Einführung in die verwendeten AWS Dienste und deren zugrunde liegenden Konzepte werden die Schritte, die unternommen wurden, um VRESCo innerhalb Amazons Cloud-Architektur bereitzustellen, vorgestellt. Der Fokus liegt dabei vor allem auf dem Vergleich und der Bewertung der von Amazon angeboten Cloud-Datenbanken, sowie auf den Techniken (Caching, Lazy-Loading, etc.), die eingesetzt wurden, um eine möglichst optimale Leistung zu erzielen.
Insgesamt zeigen die Ergebnisse, dass AWS für den Aufbau von Cloud-Architekturen, die es ermöglichen auf lokale Ressourcen zu verzichten, geeignet ist. Die Dienste sind nicht nur einfach zu bedienen, sondern lassen sich auch flexibel an die Bedürfnisse der Anwendung anpassen.

Danksagung

Die vorliegende Diplomarbeit stellt das Ende meines Studiums dar und steht für einen weiteren Höhepunkt in meiner Ausbildung. Weiters steht diese Arbeit für mich auch für den Übergang vom jetzigen Studentenleben zum anstehenden Berufsleben.
Ich möchte hiermit die Gelegenheit nutzen und mich bei allen Personen bedanken, die mich auf diesem Weg stets unterstützt haben und es mir überhaupt erst ermöglicht haben, dass ich so weit kommen konnte.
Meinen Eltern danke ich für all die Bestärkung und Unterstützung, die ich erfahren durfte. Nicht nur für die finanzielle und moralische Unterstützung, sondern vor allem auch dafür, dass sie mir alle Möglichkeiten geebnet haben, bin ich ihnen unendlich dankbar. Ich möchte mich auch für deren Verständnis bedanken, dass ich mir während meines Studiums all die Zeit nehmen konnte, die ich für meine Bildung und Ausbildung als für notwendig erachtet habe.
Ganz besonders möchte ich mich auch bei meiner Schwester Ana für all die Motivation und Inspiration bedanken. Durch die Gespräche mit ihr habe ich so viel über die wirklich wichtigen Dinge im Leben gelernt.
Weiters möchte ich mich auch bei meinen Betreuen, Prof. Dr. Schahram Dustdar und Mag. Philipp Leitner, für die ausgezeichnete Betreuung meiner Diplomarbeit, die professionellen Hilfestellungen und das stets prompte Feedback bedanken.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

In the recent years there has been quite a hype about *cloud computing* [62]. This paradigm represents the idea of omitting to maintain own IT infrastructure (computing power, database storage, etc.), but rather renting it from a vendor and accessing it via network (typically the Internet). The infrastructure, which resides 'somewhere out there' is dynamically rented on demand and only used when needed. The author of [24] states that cloud computing points us the direction in that computer architectures will be developing in the near future and that it raises computational capacity onto the same level as electricity is nowadays. This means that although you could generate your own electricity, you certainly would not do it, as it is more applicable to purchase it from a vendor.

One of the main advantages of cloud computing is that you do not need to make any upfront investments before you can start using the IT capacity you need. This enables a whole new perspective and potential for businesses and application/service developers. Rather than having to buy and set-up expensive and maybe unprofitable servers, it is possible to just start using them without actually having to own them. By using such a just-in-time infrastructure you also do not need to bother about it after it has served its purpose and you have no use for it anymore. Furthermore, there is no need for over-provisioning and maintaining IT infrastructure that is hardly used and just has the purpose to manage potential peaks in usage.

Another important motive for using cloud computing is the fact that there is no need to care about setting-up and maintaining the used infrastructure. Depending on the cloud service, the vendor usually takes care of its whole management. Cloud computing also comes coupled with a certain pay-as-you-use policy, which means that it is usually strictly usage-based and you only pay for the computational capacities that you have actually used.

The number of users and IT companies that are recognizing the cloud's potential increases, and so cloud computing turns out to establish itself more and more. While this new paradigm helps to handle your own IT infrastructure more efficiently, on the other hand it also raises questions about security and safety: How save is the data after putting it into the cloud and exclusively accessing it via the Internet?

Service-oriented Architecture (SOA) [75] is another paradigm that has received a lot of attention in the last couple of years. The motivation behind using services lies within the idea of making software components dynamically usable and flexible. Other advantages that characterize services are clearly defined interfaces and their general independence from a specific platform.

Service registries play an important role when it comes to mediation between the so-called *service providers* and the *service requestors*, which are the clients using the services. With the help of service registries the basic idea is actually made possible, whereby in a SOA the services are first of all published, then dynamically found, afterwards bound and finally executed.

The *'Vienna Runtime Environment for Service-oriented Computing' (VRESCo)* [66] is developed at the Institute of Information Systems on the Vienna University of Technology. Besides being just another service registry, VRESCo focuses on the original ideas and goals of a SOA and additionally offers unique features for the provisioning and the usage of services (e.g., dynamic binding and invocation, service composition and versioning, support for meta-data, extended searchability, notification on events, etc.).

The goal of this thesis is to deploy VRESCo into the cloud. For this purpose, Amazon has been chosen as the vendor for the used cloud computing infrastructure. Besides being currently the largest and most important vendor for commercial cloud services, Amazon has been also chosen because of the variety of its offered services.

With its *Amazon Web Services (AWS)* [10], Amazon offers a series of services that make it easy to rent inexpensive computing capacity in the form of virtual machine instances for swapping out data and computation into the cloud (*Amazon Elastic Compute Cloud (EC2)* [3]). Besides a relational database service (*Amazon Relational Database Service (RDS)* [5]), also a database service based on key-value pairs (*Amazon SimpleDB* [8]), which especially convinces by its enormous scalability and flexibility, is offered.

1.1 Motivation

The motivation in deploying VRESCo into the cloud is to benefit from the advantages that a cloud computing solution has to offer. As already mentioned, there are no up-front investments for the needed infrastructure and computational capacities. Because of the usage-based and pay-as-you-go policy that often goes along with cloud computing, a solution involving cloud computing is in most cases cheaper and more profitable than a traditional architecture.

Another significant argument is that the cloud capacities are designed to be very scalable. If needed, it is possible to dynamically rent huge amounts of storage capacities and enormous computational power. This is very essential when it comes to rapidly handle certain peaks of load and demand by being able to temporary scale the used infrastructure up.

A further benefit is the omission of certain administrative tasks when working with computational capacity in the cloud. As mentioned, it is not necessary to set-up or maintain any hardware components, or when it comes to the level of software it is not needed to take care of any installations or updates. When using a database service in the cloud, the database administrator's task of keeping the database 'up and running', and for example to index the data for upcoming searches could be omitted. It is also clear that management tasks as backing-up and replicating data are executed more efficiently when done automatically in the cloud.

When deploying VRESCo into the Amazon cloud, not only a server instance of the application should be run from a rented virtual machine instance, but also the underlying database layer should be replaced by an appropriate cloud database service.

There are several approaches on how to run the application's database from within the Amazon cloud. Besides setting-up and running a database server in a rented Amazon EC2 machine instance, Amazon also offers self-managed cloud database services. These services basically differ in their fundamental paradigms. While Amazon RDS is a fully-featured relational database, Amazon SimpleDB features a simple key-value store. In this thesis the use of these services in conjunction with VRESCo will be evaluated.

When swapping out VRESCo's database, the overall goal will be to exclusively focus on the application's Data Access Layer (DAL). The idea behind this is to keep VRESCo separated from the decision of what database is actually going to be used. A possibility to easily and transparently choose among the desired databases will be offered by introducing an appropriate 'switch' in the application's configuration file. The end user is then able to flexibly decide about to either use his database on the local machine or to make use of one of the mentioned Amazon databases in the cloud.

Because VRESCo currently runs upon a relational database, a particular challenge will be to fully substitute it by the Amazon SimpleDB service. The objective is to support the features that are offered by the current database and that are already used by the application. The SimpleDB service seems at first to be rather unsuitable for the use with VRESCo because of its missing support for the features of a relational database (e.g., relational database-schema, transactions, etc.). The application will, by using it, still benefit from the various other advantages it has to offer (e.g., high availability, theoretically limitless scalability, no database-administration, very low costs, etc.).

1.2 Contribution

Next to a theoretical overview over the definitions and concepts of SOA and cloud computing, an understanding for their goals and advantages will be given. Besides also presenting the current state-of-the-art of the used cloud infrastructure services, the characteristics and uses of VRESCo will be presented.

The goal of this thesis is to gradually evaluate the steps of deploying an existing server application – in this case VRESCo – onto Amazon's cloud services. It should be started by rewriting the existing database layer, which is currently based on a relational database, and to substitute it by Amazon SimpleDB. The objective is to only work on the application's 'lower' database layer and to preserve the existing database schema along with its functionality.

When exchanging the local database server by an external database service, it is also expected to decrease the overall performance of the application because any read- and write-operation on the database will be handled over the Internet. Several techniques, like lazy loading and various caching mechanisms will be applied to overcome these performance issues.

Besides the use of SimpleDB, VRESCo's behavior will also be examined when working upon Amazon's Relational Database Service as primary database. A significant advantage of this service is that it offers a fully-featured relational database and completely takes care of its administration.

After swapping out VRESCo's database, the actual server application will itself be deployed into the cloud with the use of a rented virtual machine instance (Amazon's EC2 service). Amazon states that the communication within its cloud – between two of its services (in this case between EC2 and SimpleDB or RDS, respectively) – has the latency of a Local Area Network (LAN). It is expected that this step leads to a better performance because of the minimized latency of the database communication. Furthermore, there are also no charges for the messages sent between two Amazon services interacting with each other as long as they communicate within the cloud.

Finally the goal of this thesis is to gradually evaluate and compare VRESCo with the use of the mentioned services and optimizations upon a beforehand introduced use case scenario. The overall evaluation shows that the best results, when deploying an existing application that is originally based on a relational database, are achieved by using the Amazon RDS service in conjunction with Amazon EC2.

1.3 Organization

The structure of the remainder of this thesis will be outlined in the following:

- Chapter 2 will give definitions and describe the core concepts of Service-oriented Architecture (SOA) and cloud computing. Next to describing their features and benefits, their similarities and especially their connectivity will be addressed. Because Amazon has been chosen in the practical part of this thesis as the vendor for the used cloud computing services, the Amazon Web Services (AWS) will be discussed in this chapter as well. This will include a presentation of their current scope of functionality and their specific features. It should also be discussed why these services were used and how they are actually used.

- In Chapter 3 related work that has been done by the scientific community will be outlined. Similar uses of Amazon's cloud services should be mentioned alongside already published experience with the use of cloud infrastructures. Furthermore, also popular uses of the AWS services will be covered.

- The Web service runtime environment, VRESCo, upon which has been worked during this thesis will be presented in Chapter 4. Besides describing its architecture and the underlying data model, some of its outstanding features will be covered in more detail.

- Chapter 5 is at first going to describe the alternative approaches of using the Amazon Web Services to deploy VRESCo into the cloud. Afterwards, each step of doing so will be presented. Helpful optimizations throughout this process will be covered as well as various other mechanisms (caching techniques, etc.).

- In Chapter 6 a possible use case scenario for VRESCo will be introduced. By using this example the overall performance of the application will be evaluated during the several steps of its deployment into the cloud. Besides just comparing the performance, the focus will also be on interpreting the applications behavior. Further limitations of the implementation will be covered as well.

- In Chapter 7 the steps undertaken to deploy VRESCo into the Amazon cloud will be recapitulated. To conclude the thesis, the results of the evaluation will be summarized and a recommendation about the use of the Amazon Web Services will be given. Possible future approaches to optimize and enhance the current results are going to be mentioned.

2 State of the Art Review

The following chapter describes the fundamental paradigms (*Service-oriented Architecture* and *cloud computing*) that build the basis for the subject of this thesis. After outlining their ideas and characteristics, the services (*Amazon Web Services*) which have been used during this thesis will be presented.

2.1 Service-oriented Architecture

In a Service-oriented Architecture (SOA), services represent the fundamental elements of the architecture and are used to build applications. SOA is a way of reorganizing software applications and infrastructure into a set of interacting services [74].

2.1.1 Services

The 'Organization for the Advancement of Structured Information Standards' (OASIS), which is a consortium that aims to develop Web service standards, defines a service as follows:

> "A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description." [39]

This definition states that a service basically consists of a *service interface* and its *service implementation*, which are both clearly separate from each other. The interface defines the identity of the service and together with an additional description defines how the service is to be invoked. The implementation, as the name states, actually implements what the service is intended to do and is usually hidden to the 'rest of the world'. This guarantees that external components do not know about the actual implementation of the service interface. They can just care about the expected result of the functionality, which is encapsulated behind the service interface.

According to [75], a service represents "a reusable unit of business-complete work" and is characterized by the following properties:

- *self-contained*: The service maintains its own state and acts autonomously. Services must not require any knowledge at the client or server side. Within a

SOA the services are *loosely coupled*. The term *coupling* indicates the degree of dependency that two systems (or in this case services) have on each other. Loose coupling provides a level of flexibility and interoperability, which allows to connect and interact more ‘freely’ [73].

- *platform-independent*: The interface of a service is limited to platform-independent assertions and the invocation mechanisms should comply with widely accepted standards. Services should be as technology neutral as possible and must be invocable through lowest common denominator technologies [74].
- *location transparent*: Services should be dynamically located, invoked and eventually combined. This is especially made possible in SOA by storing the service information in a central repository (*service registry*). A service could then be dynamically found and invoked by clients – regardless of its actual location [75].

In [74] the author points out that services come in two flavors: *simple* and *composite*. Simple services are atomic in their nature and typically exhibit a request/reply mode of operation. In contrast, composite services involve the assembly and interaction of other existing services, which have possibly even originated from multiple service providers. A common example is a business service, which composes several simple services that each accomplish a specific business task, such as order taking, order tracking, order billing and so on.

2.1.2 Definition and Concepts of a SOA

As already mentioned and as the name indicates, a SOA is centered around services. [73] states that “SOA is a meta-architectural style that supports loosely coupled services to enable business flexibility in an interoperable, technology-agnostic manner.” Another more general definition for SOA is given by [32]:

> “Service-oriented architecture presents an approach for building distributed systems that deliver application functionality as services to either end-user applications or other services.” [32]

The foundations for a SOA are three primary roles. These are the *service provider*, the *service registry* and the *service requestor*, which represents the client. A service provider holds the service’s logic and is responsible for publishing the appropriate service description in a service registry. A service requestor requests the invocation of a certain service and therefore needs first of all to be able to find the description of the desired service and to bind to it afterwards. It is possible that a service requestor is simultaneously also a service provider by not only requesting services but also providing them itself, which is the case with composite services.

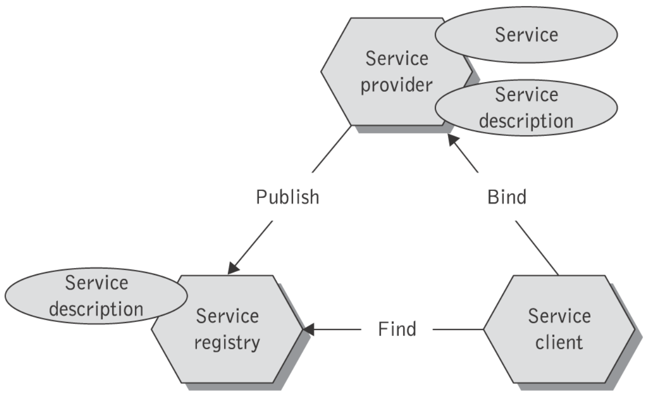


Figure 1: The SOA Triangle (from [73])

From this short description of the basic entity types of a SOA it can already be identified that there is a certain principle of how these primary roles interact. This paradigm will be referred to as *'publish-find-bind'*-paradigm and is depicted in the so-called 'SOA Triangle' (see Figure 1). In the remainder of this section, the roles, the operations and the artifacts that are involved in a SOA will be described.

These roles are:

- *Service Provider*: The service provider is the owner of the provided service and implements the underlying logic. It is a network-addressable entity that hosts and controls the access to its service. The provider is responsible for describing the service and to register this information into the service registry, so that a service requestor is able to discover and invoke the service. The service provider accepts and executes the requests from the requestor.
- *Service Registry*: A service registry is a searchable directory where it is possible for service requestors to search for previously published service descriptions. By finding an appropriate service description the requestor obtains information on how to bind to a service provider and invoke the desired service. The service provider is responsible for providing the required infrastructure to enable to publish and search for services. It also prescribes a certain format for the service descriptions, upon which the providers have to comply when publishing their services. In SOA the service registry is an important role, which actually enables to decouple the service requestor and the provider. Without the registry they would need to know about each other at compile-time. In a SOA the service requestor is able to discover and invoke a desired service at run-time without previously even knowing about its provider.

- *Service Requestor*: The service requestor, which is also referred to as *service consumer*, represents the client that is in need for certain functionality in form of a service. After searching the service registry for a suitable service and discovering the appropriate service description, it can use this information to bind to the actual service.

Each entity in a SOA can take one or even multiple of these three described roles. In Figure 1 it is depicted how these roles relate to each other in a SOA. Three primary operations must take place to fulfill the SOA-triangle. These are publication of the service descriptions, finding the service descriptions, and binding or invocation of services based on their service description [73]. The specific types of operations will be described in the following:

- *Publish*: The goal of the publishing-operation is that the service provider publishes its provided service into a service registry, so that a service requestor is able to discover and invoke it. The actual publishing precisely consists of two other operations, namely describing the service itself and the actual registration of the service. So in order to publish a service to the service registry the service provider needs to properly describe the service (e.g., information regarding the service provider, the nature of the service, implementation details and invocation methods, etc.). The description has to comply with the format that the service registry prescribes in order to be searchable. After having described the service, the service provider is able to register the service by storing the service description information into the service registry.
- *Find*: The finding-operation consists of querying the registry for services matching the needs of the service requestor and selecting the desired service from the search results afterwards. A query consists of several search criteria (e.g., type of service, technical service characteristics, etc.) and is executed against the service description information in the registry that were entered by the service provider.
- *Bind and Invoke*: During the binding-operation the service requestor interacts with the actual service a run-time. It therefore uses the technical information and the binding details from the service description which it gathered from the service registry. Besides the direct invocation of the service, it is also possible to use the service registry as a mediator for all the communication between the service requestor and the service (provider) [73].

According to the author of [74], the roles and operations within a SOA act upon the following service artifacts:

- *Service description*: The service description usually contains information about the service interface, its capabilities (conceptual purpose and expected results), the behavior during execution, and its quality of service (functional and non-functional quality attributes; e.g., cost, performance, security, availability, etc.). The description specifies the way a service requestor will interact with the service provider and specifies the format of the request and response from the service [74]. The publication of this information to a service registry is necessary for discovering, selecting and binding to the service. In order to fulfill its purpose it is essential that the description complies with a format that is prescribed by the service registry.
- *Service (implementation)*: The implementation realizes a specific service interface whose implementation details are hidden to the 'rest of the world'. It is the actual service, which is hosted and made available by the service provider for the use through the published service description.

In [32] the authors state that SOA is able to help organizations to succeed in the dynamic business landscape of today. SOA allows businesses to be ready for the future and provides the flexibility and responsiveness that is critical in order to remain competitive. The ability to compose services and to reuse them in general reduces the time needed to go through the SW-development life-cycle and leads to rapid development and faster time-to-market. Another benefit of the increased potential for reuse is the reduction of costs. Since one of the key principles of SOA is the clear separation of the service specification and its implementation, the impact when infrastructure and implementation changes occur is minimized. Through this the overall complexity and an eventual integration becomes more manageable. It is also possible for organizations to make use of the additional layer of abstraction provided by SOA and to wrap existing IT investments as services in order to leverage these assets.

2.1.3 Web Services

One possibility to realize SOA are *Web services* [73]. It is important to point out that Web services are not the only technology that can be used to implement a SOA. Web services have also been used to implement architectures that are not Service-oriented. The W3C's Web Services Architecture Working Group defines a Web service as follows:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface

> described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." [45]

For describing, advertising, discovering and binding Web services in a decentralized, distributed Service-oriented environment the key standards are *WSDL*, *SOAP* and *UDDI* [89]. The interoperability of these standards supports the basic activities of a SOA, namely the previously presented 'publish-find-bind'-paradigm. The Web service standards are completely independent of programming language, operating system and hardware and are based on open technologies [32].

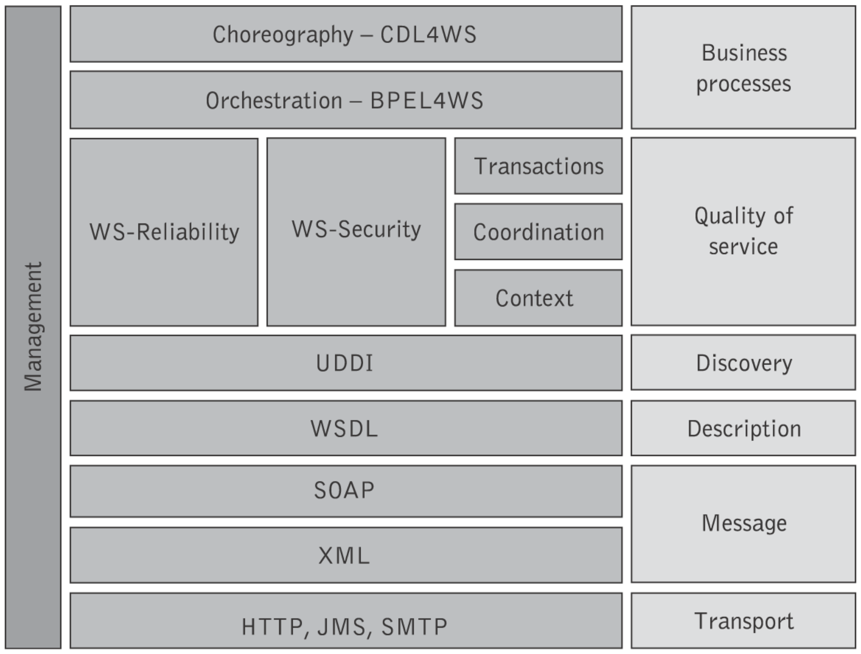


Figure 2: The Web Services Technology Stack (from [73])

A comprehensive picture of the Web services technology stack is given in Figure 2. One can see that the *Extensible Markup Language (XML)* [93], which provides a cross-platform approach to data encoding and formatting, is used as the fundamental building block for nearly every other layer. In the following the mentioned key standards will be briefly presented:

- *SOAP*: SOAP [95] is a messaging protocol for exchanging structured information. It uses XML for data encoding and is not bound to any specific transport protocol, although it is typically used on top of HTTP to carry its data. SOAP is not tied to any operating system or programming language and is designed to be simple and extensible.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">

  <env:Header> <!-- optional -->
    <!-- header blocks go here ... -->
  </env:Header>

  <env:Body>
    <!-- payload or Fault element goes here ... -->
  </env:Body>
</env:Envelope>
```

Listing 1: Structure of a SOAP Message (from [73])

A SOAP message has a simple structure and consists of a so-called *Envelope*, which contains an optional *Header*- and a mandatory *Body*-element (see Listing 1). The header contains blocks (*header blocks*) of information on how the message needs to be processed and is therefore possibly changed between the transport layers. While the header holds information relevant for the endpoints (or intermediate transport points), the main XML data (*payload*) of the message is placed inside the body. The body-element may contain an arbitrary number of *body elements* and either holds the application-specific data or a fault message.

- *WSDL*: The 'Web Service Description Language' (WSDL) [94, 96] is an XML-based specification schema that is used for describing the interface of a Web service. Its purpose is to specify *what* a certain Web service does, *where* it is located and *how* it is invoked.

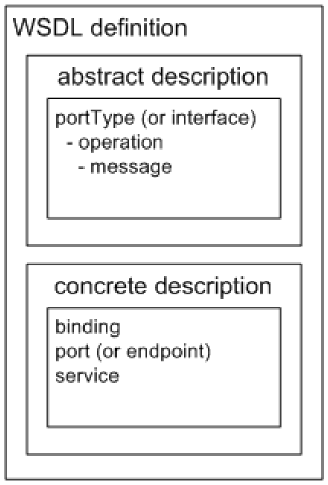


Figure 3: WSDL Document (Consisting of Abstract and Concrete Parts)

A complete WSDL service description provides two pieces of information: an application-level service description, or abstract interface, and the specific protocol-dependent details that users must follow to access the service at concrete service endpoints [27]. By separating the abstract definition of endpoints and messages from their concrete network deployment or data format bindings, certain reusability among the various elements of the description is achieved. Figure 3 outlines the basic structure of a WSDL document and illustrates the separation of the individual descriptions. The further elements of a WSDL description are presented in more detail in [73].

- *UDDI*: The 'Universal Description, Discovery and Integration' (UDDI) [37] is a platform-independent and XML-based registry mechanism that is used to look-up Web service descriptions. It is a specification proposed by the 'Organization for the Advancement of Structured Information Standards' (OASIS) on how to implement Web service registries. UDDI specifies the way to store and retrieve information about Web services (`businessService`), their corresponding service providers (`businessEntity`) and their technical entry points (`bindingTemplate` and `tModel`). It does not only provide a repository for WSDL documents, but rather has its own general-purpose model for capturing the information. A UDDI registry has a SOAP interface and provides APIs for registering and discovering Web services.

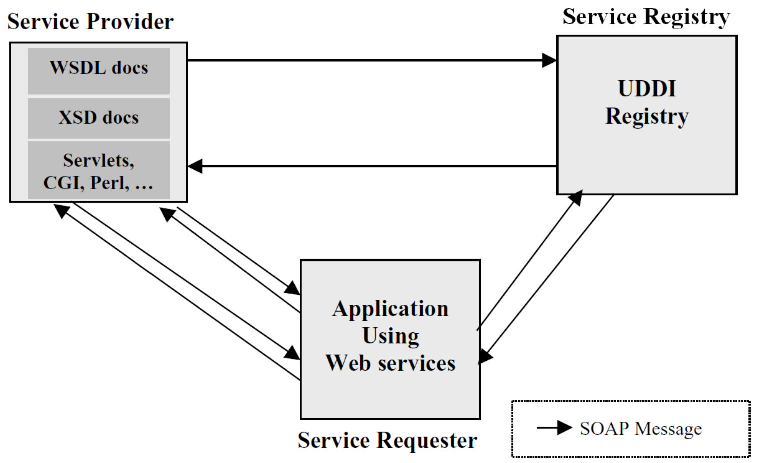


Figure 4: How SOAP, WSDL and UDDI Relate to Each Other in SOA (from [89])

Figure 4 depicts the roles of these key Web service standards within the previously introduced 'SOA Triangle' (see Figure 1) and illustrates how they relate to each other. The importance of a (UDDI) service registry should once again be emphasized, as it plays an important role within the interaction of these three standards.

> "SOAP, which is built on XML, defines a simple way to package information for exchange across system boundaries. [...] WSDL is an XML

grammar for specifying properties of a Web Service. The UDDI specifications define a next-layer-up that allows two companies share a way to query each others services and to describe their own services." [89]

2.2 Cloud Computing

Before presenting the cloud services that were used during the work on this thesis, the basic concepts of cloud computing should be discussed in this section. Besides giving a satisfying definition and outlining the key characteristics and features, certain challenges and risks of using cloud computing will be mentioned too.

According to the authors of [76], *cloud computing* is one of the most vague 'technique terminologies' in the world of information technology right now. One reason for this is that cloud computing can be used in many application scenarios, another reason is that cloud computing is currently hyped by lots of companies for business promotion [76].

Cloud computing is not actually a revolutionary recent development, but it is rather the combination of many preexisting technologies and the result of the continuous evolution of data management technology [100]. This means that although cloud computing is getting a lot of attention from the IT community nowadays, some technologies on which it draws on are not new and have matured at different rates and in different contexts.

Besides technologies like Service-oriented Architectures (see Section 2.1) and Web services (see Section 2.1.3), cloud computing is also based on virtualization technologies. The term *virtualization* refers to the "abstraction of compute resources (CPU, storage, network, memory, application stack and database) from applications and end users consuming the service" [61]. The idea is to combine and manage physical resources as so-called *pools*. Upon request, it is possible to dynamically create a suitable platform for a specific application out of these resource-pools. Instead of a physical machine, a virtual machine is used [20].

The author of [24] stresses the potential impact of cloud computing on information technology. He equates the rise of cloud computing in the information age to the electrification in the industrial age. Before electrification, companies had to generate their own power (steam engines, dynamos). After electric utilities started to offer cheap power, it not only changed how businesses operated, but also set off a chain reaction of economic and social transformations. Cloud computing is the beginning of a similar change within information technology, where organizations do not have to provide their own computing resources anymore, but rather just – similar to power today – simply plug in to the cloud for the resources they need. Computing turns into a utility.

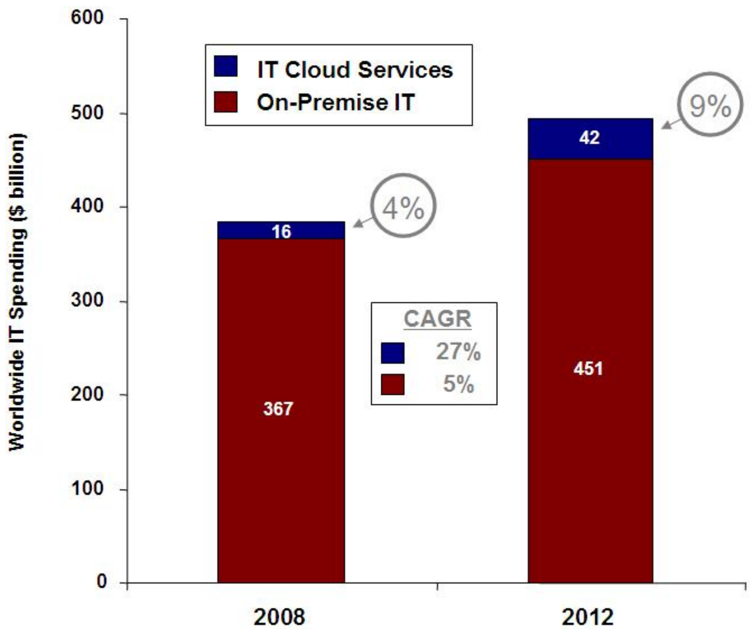


Figure 5: IT Spending by Consumption Model (from [49])

Cloud computing is forecasted for high growth and the authors of [99] state that "there is no doubt that cloud computing has a bright future". According to the 'International Data Corporation' (IDC), which is a market research and analysis company specialized in information technology, cloud services are expected to grow at a 'compound annual growth rate' (CAGR) of 27% in the next years (see Figure 5). The spending on traditional, on-premise IT is expected to grow only at a CAGR of around 5%, which is over five times less [49].

2.2.1 Definition

There are many interpretations about the actual meaning of *cloud computing*, but there is no standardized or consistent definition of the term. The basic idea behind cloud computing is to enable the offering and usage of IT infrastructure as services via the Internet.

In [90] the authors studied more than 20 different definitions of cloud computing in order to achieve a consensus definition and to extract a minimum definition containing the essential characteristics. They state that although there are many definitions on what cloud computing is, they all seem to focus on just certain aspects of the technology. Their proposed definition of the cloud concept is as follows:

> "Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs." [90]

As further result they conclude that the set of features that most closely resembles a minimum definition containing the essential characteristics would be: scalability, a pay-per-use utility model and virtualization.

Another very prominent definition of cloud computing, which is often cited in the scientific literature, is the definition given by the 'National Institute of Standards and Technology' (NIST):

> "Cloud computing is a model for enabling 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." [62]

Additionally, the 'NIST definition of Cloud Computing' also describes several essential characteristics, service models and deployment models of the cloud concept. According to its definition, cloud computing is composed of the following essential characteristics:

- *On-demand self-service*: A consumer is able to automatically acquire all the computing capabilities (e.g., server time, network storage) he needs. For this purpose he is not required to humanly interact in any way with the provider of the service.
- *Broad network access*: All the capabilities are available and accessed through the network. They are accessed through standardized mechanisms that "promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs)" [62].
- *Resource pooling & Multi-tenancy*: In order to serve multiple consumers, the computing resources offered by a provider are necessarily pooled. Different physical and virtual resources are dynamically allocated and assigned according to the consumers demand. Typical examples of computing resources are

processing time, storage, memory, network bandwidth and virtual machines. Due the resource pooling "the customer generally has no control or knowledge over the exact location of the provided resources" [62]. It is still often possible to specify desired locations at a higher abstraction-level, like country or region.

- *Massive scalability*: With cloud computing the capabilities and resources available for provisioning often appear to be unlimited to the consumer. Furthermore, it appears that the capabilities can be purchased in any quantity and at any given time [62].
- *Rapid elasticity*: The customer needs to be able to quickly scale (out and in) according to the actual demand. Therefore it is necessary to rapidly and elastically provision and release capabilities. In some cases this is even done automatically.
- *Measured Service*: Typically the use of a resource is controlled and optimized automatically and by the cloud system itself. In order to obtain a certain transparency for both the provider and the consumer it needs to be able to monitor and control the usage of the utilized service.

Very similar are the key attributes identified by the authors of [61]. In addition to the 'NIST definition of Cloud Computing', they stress that a certain *'pay as you go'* or *pay-per-use* policy is another essential attribute of computing in the cloud. It means that the consumers are only paying for the resources that they are actually using and only for the time they require them.

In [30] the author even goes one step further and identifies *little or no commitment* as being another important aspect and essential characteristic of cloud computing.

The idea behind cloud computing is often confused with grid computing [41], which is a much older concept. Because clouds and grids both share similar visions, the distinction between the two concepts is not often clear. According to [40] a grid is a system that:

> "coordinates resources that are not subject to centralized control, using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service." [40]

This means that grid computing is mainly about the joint use of shared resources. Distributed and parallel computing is achieved by composing a cluster of loosely coupled computers. As opposed to that, with cloud computing there is a clear distinction between the supplier, who holds the resources under centralized control and the consumers.

Grid computing is primarily used in academic settings and one of its most prominent examples is the *SETI@home* project [98]. A more in-depth analysis, highlighting the similarities and differences of the two paradigms is carried out in [90].

2.2.2 Categories

There are several possibilities and dimensions to classify cloud computing offerings. Besides the categorization according to the application domain (e.g., high-performance computing, analytics, finance, web), two other commonly used ways should be presented in the following. Figure 6 illustrates the relationship between services, uses and types of clouds.

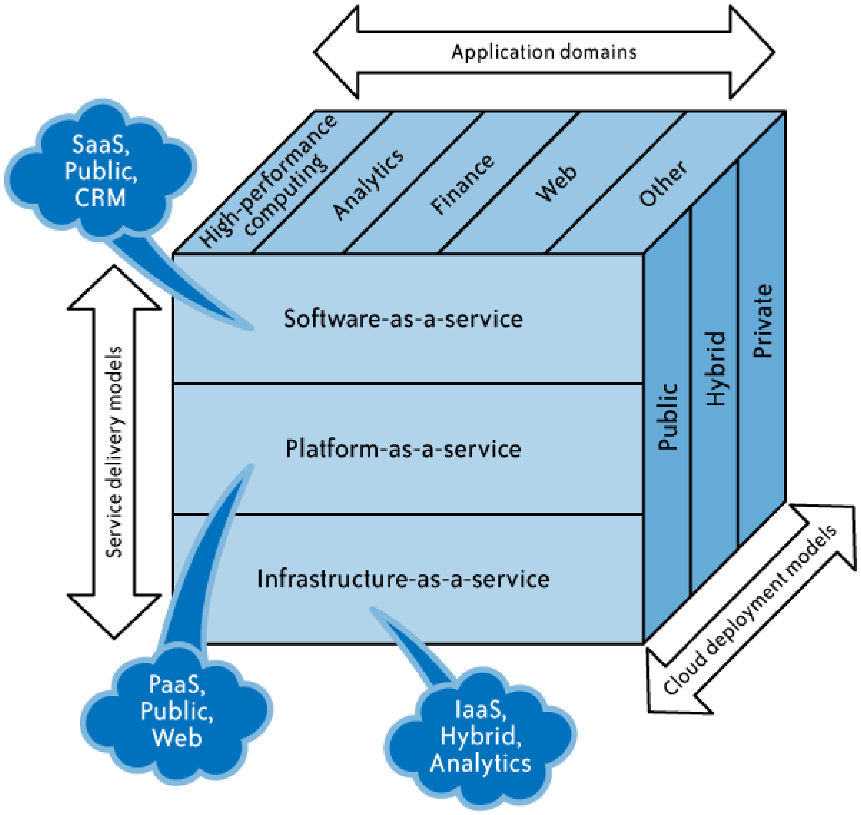


Figure 6: Categories of Clouds (from [61])

One aspect upon which cloud architectures can be viewed is the organizational point-of-view. It differentiates between the separation of organizational entities (vendors and customers). The categorization is based on the service boundary and classifies cloud computing according to its deployment method, which is *public*, *private* or *hybrid* [76]:

- *Public Cloud*: With public clouds (or external clouds) the vendor and the potential customers do not belong to the same organizational entity. It is basically cloud computing in its traditional sense, where the vendor shares his resources by offering publicly accessible cloud services to external parties. These customers are then able to use them on a self-service basis over the Internet.

  The third-party vendor not only hosts, operates and manages the cloud, but is also responsible for the security management. Which means that "the customer of the public cloud service offering has a low degree of control and oversight of the physical and logical security aspects" [61].

- *Private Cloud*: A private cloud (or internal cloud) is built and operated by enterprises for their own use and purpose. This means that in this case both the vendor and the customer belong to the same organizational entity.

  The main argument for operating an own private cloud is security. In a private cloud the control over the data does not exceed organizational boundaries and therefore stays with the customer. Further benefits for managing data and processes within the own organization would be no restrictions of the network bandwidth and no legal requirements that might entail when using public cloud services across open and public networks [79].

  The downside is that because the organization has to buy, build and manage its own cloud, it does not benefit from lower upfront investments and less management.

- *Hybrid Cloud*: Hybrid clouds are scenarios in which services from both public and private clouds are used. A hybrid environment may consist of multiple internal and/or external providers where the resources are shared between public and private clouds.

  The authors of [61] write that with a hybrid cloud "organizations might run non-core applications in a public cloud, while maintaining core applications and sensitive data in-house in a private cloud". Another scenario would be to solely rely on private resources during normal operation and to swap (uncritical) processes and data to a public cloud at certain peaks.

Another way to classify cloud computing is according to the service type and its delivery model. This rather technical point-of-view focuses on functional properties and classifies cloud services as follows:

- *Infrastructure-as-a-Service (IaaS)*: IaaS is, as its name states, the delivery of computer infrastructure as a service. Customers are offered an abstract view on rentable hardware (servers, mass-storage, network, etc.), which is housed by the vendor.

"The IaaS model is similar to utility computing, in which the basic idea is to offer computing services in the same way as utilities." [61] While with Infrastructure-as-a-Service the provider is in complete control over the infrastructure and it abstracts the user from details about the physical computing resources and their exact location, with utility computing the user wants to be in control over the geographical location of the infrastructure and to know what runs on each server.

At the most basic level of cloud computing, developers are offered virtual machine instances. These instances behave like dedicated servers and are controlled by the customers, who have the full responsibility for their operation. Additional to that, an interface for managing the resources is offered, which allows to allocate resources, create and delete images of operating systems, scale claimed resources, define network topologies and so on.

Besides features like scalability and pay-per-use that were already mentioned in Section 2.2.1, the IaaS model also often enables instant access to the latest 'best-of-breed' technology solutions for just a fraction of their actual cost.

- *Platform-as-a-Service (PaaS)*: An additional abstraction level to IaaS is added by Platform-as-a-Service. In the PaaS model the vendor offers a development environment to the application developer, who then develops applications and offers these as services through the cloud platform. The environment abstracts machine instances and other technical details from the developer, who is in contrast to IaaS not concerned with matters of allocation [60].

  Furthermore, the idea is to "provide developers with a platform including all the systems and environments comprising the end-to-end life cycle of developing, testing, deploying and hosting of sophisticated web applications as a service" [79]. This means that the vendor provides development toolkits, standards for the development and channels for distribution.

  The PaaS model can help to slash development time and consequently enables the rapid propagation of software applications and services. Further benefits are the leveraging of already established channels for costumer acquisition and the offering of several readily available (development) tools and services.

  Besides the benefits of the PaaS model and the development environment that comes along with it, developers often have to deal with some constraints that this environment imposes on their application design (e.g., key-value stores instead of relational databases) [60].

- *Software-as-a-Service (SaaS)*: SaaS offerings are software applications in the cloud, primarily addressed directly at the end user. Usually these services are of potential interest to a wide variety of users and represent an alternative to locally run applications.

With Software-as-a-Service the customer does not purchase any software, but rather rents it for use on a subscription basis, which is in some cases even free for limited use. What differentiates SaaS from traditional websites or web applications is that SaaS mainly interfaces with the information and data (e.g., documents) of the user and offers him online resources and storage.

There is typically no need for the customer to load software onto his own hardware and on customer-side no need for local software installations or allocations of the needed resources. Furthermore, the customer does not need to take care of updating or eventually even patching the application, as it might be the case with traditional desktop applications. With SaaS, "the purchased service is complete from a hardware, software, and support perspective" [61].

Prominent examples for SaaS are the online alternatives of typical office applications such as word processors [90]. Two concrete examples with very rich and complex functionalities are *Google Docs* [43] and *Salesforce.com* [85]. With *Google Docs* offering, as it states, "online documents, spreadsheets, presentations, surveys, file storage and more" and *Salesforce.com* distributing business software ('Customer Relationship Management') as Software-as-a-Service on a subscription basis.

In [79] the authors state that Software-as-a-Service is "heralded by many as the new wave in application software distribution".

Various cloud service offerings fall into different levels of abstraction and are aimed at different market segments, which is depicted in Figure 7. The figure shows the different actors involved in computing in the cloud and their categorization into roles. The separation of roles helps to identify the different stakeholders and their individual interests.

It is clear that the *vendor* is in charge for providing the demanded resources, no matter if on IaaS-, PaaS- or SaaS-level. In a private cloud both the vendor and the consumer belong to the same organizational entity. Still, within each cloud the role of the provider can only be occupied by the vendor itself, who therefore also controls the cloud.

Cloud service offerings at the level of Software-as-a-Service are mostly aimed at the so-called *end users*. Whereas services at IaaS- and PaaS-level are mostly intended for *developers*. These developers utilize the provided resources and then build the services for the end users. The SaaS offerings are commonly built on basis of PaaS and/or IaaS offerings and are hosted and probably even managed by the vendor.

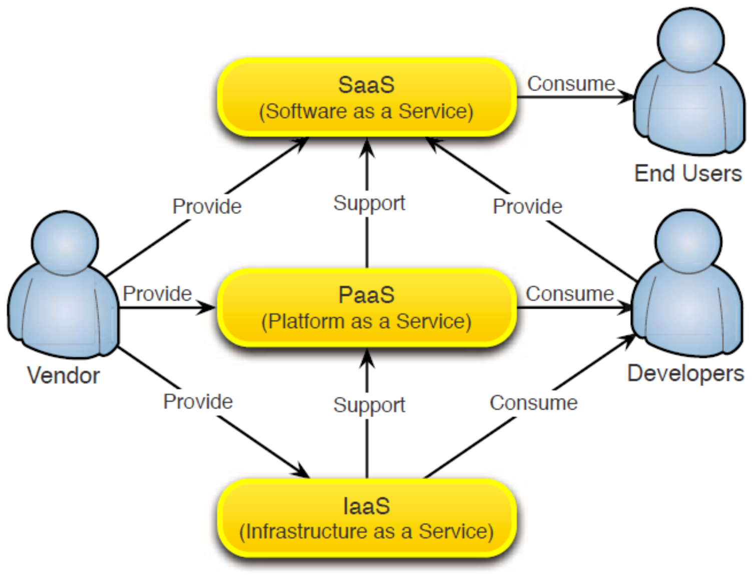


Figure 7: Abstractions of Cloud Computing (from [60])

Actors can take on multiple roles. This makes it possible that for example a vendor also develops services for end users. Another scenario would be the developer utilizing cloud services that were developed by other developers in order to build his own.

2.2.3 Advantages

One of the central arguments for customers to rely on cloud computing are lower IT costs. This is mainly achieved by the fact that there are almost no upfront investments into the IT infrastructure necessary. In [91] the author states that “to build a large-scale system it may cost a fortune to invest in real estate, hardware (racks, machines, routers, backup power supplies), hardware management (power management, cooling), and operations personnel”. Through computing in the cloud all these capital expenditures could be avoided because there is mostly no need to purchase any additional hardware or software.

Because the cloud infrastructure is commonly billed only according to its actual use, it is no longer an investment, but could rather be treated as an expense. The pay-per-use model means that the customer is not liable for the entire infrastructure that he may use, but is only being billed for the fraction that is used by him [91]. This ‘pay as you go’ policy often manages it to keep the ongoing IT costs low.

All the mentioned advantages make cloud computing especially very interesting for smaller and emerging businesses (e.g., start-ups) because they lower the barrier to entry and open up prospects that would maybe not be possible without. Even companies that could afford to invest into the needed IT infrastructure benefit from omitting several rounds of management approvals before the actual project could get started [91]. It is also a fact that by swapping out your infrastructure the number of needed IT staff personnel is being reduced and you do not carry the overhead of additional salaries, benefits, insurance and building space that goes along with traditional IT system [92]. Furthermore, you do not need to worry about setup, management and constant maintenance.

Another significant advantage of cloud computing is that because of its high flexibility, companies can acquire computing and development services as needed and on demand. This not only shortens the time to complete a particular project, but also reduces the so-called 'time to market' and helps to stay competitive. By scaling the utilized service depending on the actual demand, users can easily handle certain peaks or cope with eventually higher seasonal demands. Cloud computing also enables the customer to always use and work with the latest technology and provides him with access to supercomputer-like power at just a fraction of its actual cost.

In [91] the author states the following dilemma: "In the past, if you got famous and your systems or your infrastructure did not scale you became a victim of your own success. Conversely, if you invested heavily and did not get famous, you became a victim of your failure." Through the 'just-in-time infrastructures', which are made possible by cloud architectures, the risk of failing because of one of the addressed scenarios is rather low because you only scale as you grow.

It is essential to add that cloud infrastructures can be relinquished in the same and easy way as they were acquired in the first place – usually in minutes [91]. This makes cloud computing very interesting for projects that would otherwise not pay off because of the large IT investments that are necessary in order to carry out a particular task. After usage, users do not carry any ballast in form of unutilized IT infrastructure and do not need to further manage or care about such.

The use of cloud computing has also a considerable effect on the consumer's governance structure. Traditionally, most IT organizations govern the technology layers shown in Figure 8 on their own. Cloud computing however has the effect of reducing the complexity of the consumer's IT infrastructure by moving the level of control towards the cloud vendor. This is especially the case when moving from IaaS to PaaS to SaaS [61].

According to [91], a further benefit of cloud computing is that it has the potential for shrinking the processing time of compute- or data-intensive jobs in conjunction with parallelization: "An elastic infrastructure provides the application with the ability

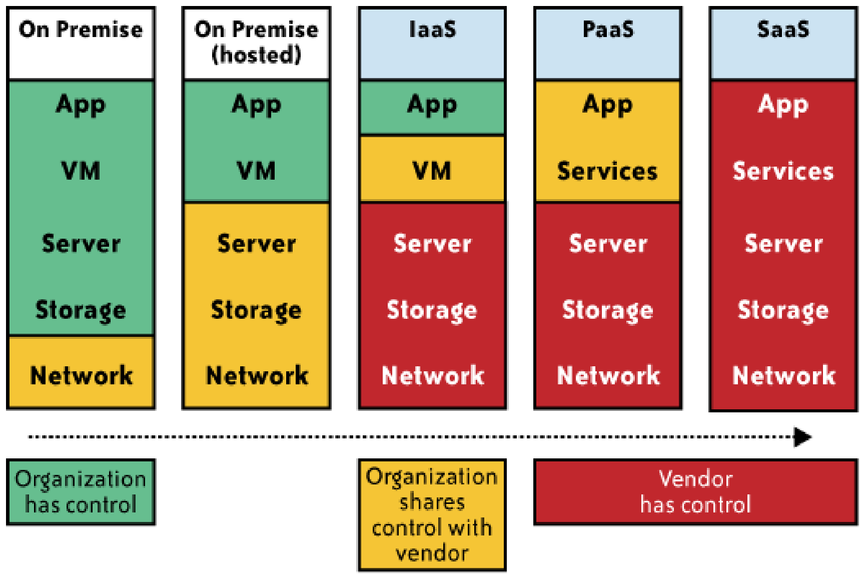


Figure 8: Impact of Cloud Computing on Governance Structure (from [61])

to exploit parallelization in a cost-effective manner reducing the total processing time." [91]

But not only customers can potentially benefit from cloud computing. This new paradigm comes in also very handy for larger companies that have already invested in a considerable IT infrastructure in order to fulfill the demands of their own customers. Because such an infrastructure is designed to bear up against certain peaks in the customers demand, it mainly stays unutilized during normal operation. In order to gain profit out of already undertaken investments and to maximize its 'Return-of-Investment' (ROI), it just seems natural to offer parts of the unused resources as cloud architecture to potential new customers.

2.2.4 Challenges

Although there are many benefits of computing in the cloud, there are also some significant barriers to its adoption. Two of its most significant challenges are security and privacy.

Most of the security and privacy issues in cloud computing are caused by the lack of control over the physical infrastructure and the cloud's multi-tenancy model. In [51] the authors provide an overview over the technical security issues of cloud computing environments and present possible existing and upcoming threats to its security.

It is important to add that many of these issues are not specific to cloud computing, but rather relate to the underlying security problems of Web services, browsers and so on. As cloud computing makes heavy use of these, the security issues become more significant [54]. An example is a 'Denial of Service' (DoS) attack against a resource, where the user ends up paying for the attack because of the increased usage it has caused.

The authors of [51] also identify that "completely relying the own data and execution tasks to an external company, eventually residing in another country with a different regulatory environment, may cause companies not to consider Cloud Computing". The major barrier is, in other words, the lack of knowledge about the full details of how and where the own data is being processed. In [25] the authors write that the potential of the cloud is not yet being realized and that "Lack of control in the cloud is the major worry. One aspect of control is transparency in the cloud implementation – somewhat contrary to the original promise of cloud computing in which the cloud implementation is not relevant."

Security issues are very critical from an enterprise perspective and the risks of potential security and privacy threats, like data exposure, may even exceed the possible benefits of cloud computing. Another point is that some data (e.g., financial and health sector) may have to be maintained within a specific jurisdiction and is simply not transferable to the cloud [54].

In [97] the authors state that the sharing of virtual machine (VM) images is "one of the fundamental underpinnings of cloud computing". According to them, virtual machine image sharing unavoidably introduces new security risks and in their article they elaborate these from different perspectives. The practicality of mounting cross-VM attacks in existing third-party clouds is explored in [80]. The authors show that it might be possible for an attacker to "penetrate the isolation between VMs and violate customer confidentiality" [80].

Besides the mentioned issues it needs to be said that it is still cheaper to implement security on a larger scale. Start-ups and individuals may eventually benefit from the expertise and greater security measures provided by cloud operators. Furthermore, they could use cloud computing to eliminate the costs of needing to develop a secure infrastructure on their own.

It is in general important to settle the question about security responsibility, which is also asked by the author of [53]: "Does using a cloud environment alleviate the business entities of their responsibility to ensure that proper security measures are in place for both their data and applications, or do they share joint responsibility with service providers?"

Besides mentioned barriers, like security concerns and third-party data control, the continuity of cloud services is another important issue. In this case users are concerned about the reliability of the actual server uptime and the availability of their critical applications and data in the cloud.

In the past there have been several documented incidents where popular cloud services have been unavailable to their users for hours. [61] reports about continued DDoS ('Distributed Denial of Service') attacks on the Amazon Web Services (AWS), which were used during the work on this thesis and will be presented in Section 2.3. A couple of further outages (some even ranging up to 8 hours) of individual AWS services are listed in [18]. In [76] even a 22 hours long down-time of one of the cloud services provided by Microsoft is mentioned.

Another common barrier to the adoption of cloud computing is the potential lock-in of customers data. The APIs for computing in the cloud are essentially proprietary and there have been no agreements on the standardization of the external interface. This means that it is currently not able for customers to easily extract their processes and data from one site and to run them on another. As a result, once a customer starts to use the service of a cloud computing provider, he is most likely to be locked-in by the provider [76].

Although the customer's lock-in may be attractive to the cloud computing provider, cloud computing users are "vulnerable to price increases, to reliability problems or even to providers going out of business" [18]. There have been incidents of companies loosing their customer's data and having to shut down because of the latter reason. An obvious solution to counteract would be to standardize the APIs for cloud computing in order to "enable migration and plug and play of cloud components" [61].

A frequently cited list of the "top 10 obstacles for the adoption of cloud computing" has been compiled by the authors of [18] and mentions further challenges, like performance unpredictability and data transfer bottlenecks.

The environmental impact of cloud computing is another very important issue and it would be irresponsible to neglect it nowadays. The authors of [60] state that a major concern is "the ever-increasing carbon footprint from the exponential growth of the data centres required for Cloud Computing". Furthermore, they add that "while these issues are endemic to Cloud Computing, they are not flaws in the Cloud conceptualisation, but the vendor provision and implementation of Clouds" [60].

IBM even claims that the IT is becoming 'greener' with the advance of cloud computing and the coherent potential to "increase the utilization rate of server, storage, network and so on" through virtualization technology [101].

2.3 Amazon Web Services

The *Amazon Web Services (AWS)* [10] are a collection of infrastructure Web services (Infrastructure-as-a-Service) delivered over the Internet by *Amazon.com*. This suite of cloud services allows third-party developers to access and build applications on top of Amazon's 'battle proved' IT infrastructure.

Being one of the world's largest online retail platforms, Amazon made huge investments into its IT infrastructure in order to bear up against significant peaks in their customer's demand, which usually emerge around holidays (Christmas, Thanksgiving Day, etc.). During normal operation most of the infrastructure remains unused and does not even reach close to its maximum capacity.

Jeff Bezos, the chief executive officer (CEO) and founder of Amazon, stated that before introducing its cloud computing offerings, the Amazon servers happened to ran at even less than 10 percent of their capacity [24]. By renting out its system to others, Amazon is able to boost its resource utilization and to slash the overall price for computing, not just for its customers but also for itself.

Today, Amazon advanced to one of the most important commercial providers for cloud services and was among the first to recognize this potential in order to make profit out of it. The company claims to have "spent over a decade and hundreds of millions of dollars building and managing [its] large-scale, reliable and efficient IT infrastructure" [15]. With AWS, developers and companies of all sizes can now take advantage of this global computing infrastructure and benefit from Amazon's experience and investment.

In the remainder of this section, the features and possibilities of AWS will be discussed. Amazon currently offers a variety of cloud services, including, among others, storage, computing power, messaging and database services. The focus will be on the presentation of the services that were used during the work on this thesis, namely the *Amazon Elastic Compute Cloud (EC2)*, *Amazon SimpleDB* and the *Amazon Relational Database Service (RDS)*. Other Amazon Web Services are not within the scope of this thesis and will not be covered.

2.3.1 Features

The following are characteristics provided by AWS that Amazon claims are "unique among all vendors in the cloud computing landscape" [15]:

- *Flexible*: With the Amazon Web Services there is no need for large investments into new architectures, programming languages or operating systems. Rather these platform-agnostic services allow the customers to "choose the programming models, languages, and operating systems they are already using or that are best suited for their project" [15]. It is essential that developers "can bring their existing skills and knowledge to the platform" and are not limited by the fact that they have to learn and adapt to the platform [15].

In [15] Amazon claims that AWS's flexibility enables "that migrating legacy applications to the cloud is easier and much less expensive" because there is no need of re-writing existing applications and that they can easily be moved into the cloud. Furthermore, it is stated that AWS "can be used to run almost anything – from full web applications to batch processing to offsite data backups" [15].

- *Cost-effective*: One of the important advantages when working with AWS is that there is no need for any upfront investments. Furthermore, the customer is not bound to any long-term commitments or minimum spending, but rather just 'pays as he goes'. Customers can start to work with the provided cloud services in minutes and are able to terminate their relationship with AWS at any time.

  As with cloud computing, there is no need to consider or even pay for costs that go alongside owning and operating an own IT infrastructure (e.g., power, cooling, real estate, IT administration staff) when using AWS.

  "AWS provides businesses with the increased agility needed to be able to instantly scale their infrastructure up or down based on their unique demands. This business agility can often be a point of cost savings itself. When a business is able to respond quickly to changes, no matter how large or small, it can take on new opportunities and meet business challenges that could drive revenue and reduce costs." [15]

- *Scalable and elastic*: AWS as cloud services can help to elastically "scale computing resources up and down easily and with minimal friction" [15]. This is essential when the customer's application needs to cope with certain usage peaks and unexpected demands.

  "At the same time, the cloud is also useful as a resource for executing mission-critical, short-term jobs." [15]

  To allocate resources with AWS it only needs to involve simple API calls. Amazon supports seamless scaling and additionally offers load balancing and auto-scaling features.

- *Secure*: "In order to provide end-to-end security and end-to-end privacy, AWS builds services in accordance with security best practices, provides appropriate security features in those services, and documents how to use those features.

In addition, AWS customers must use those features and best practices to architect an appropriately secure application environment." [15]

Besides leading the way to secure services and data privacy for its customers, Amazon also stresses its use of multiple layers of operational and physical security to ensure the integrity and safety of their customer's data. Additionally to that it refers to successfully obtained security certifications and accreditations to demonstrate the security of its infrastructure and services.

Readers interested in details about the security of the Amazon Web Services should be referred to the 'AWS Security Whitepaper' [14].

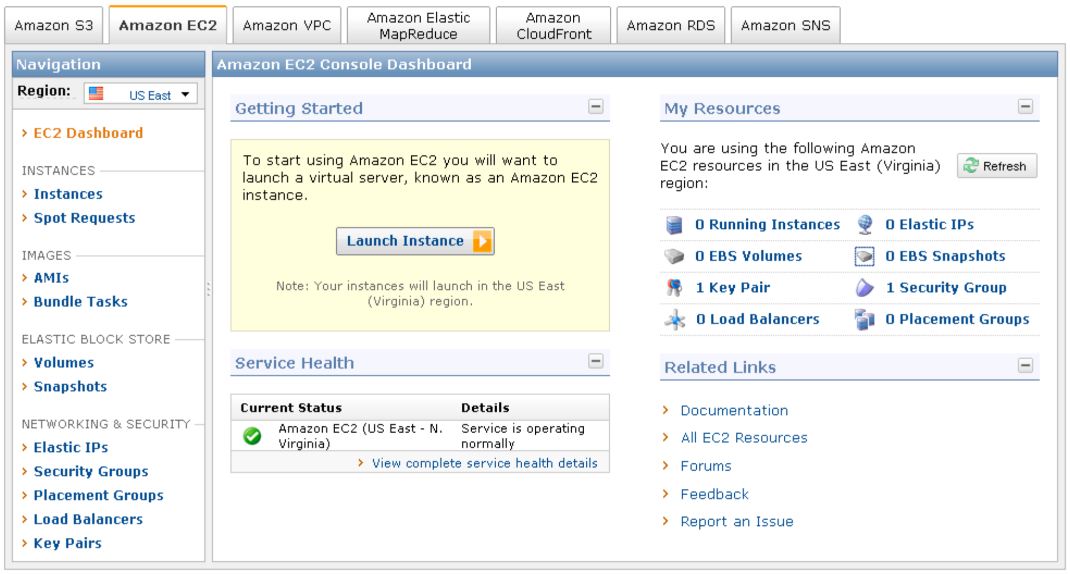
- *Experienced*: Amazon manages the multi-billion dollar business Amazon.com and serves millions of customers. By running the world's largest retail platform for over 16 years, Amazon has gained immense reputation and expertise. AWS was built on top of this experience and benefits from the company's infrastructure management skills and capabilities. The company itself states that "AWS has been operating since 2006 and today serves hundreds of thousands of customers worldwide" [15].

  Another appealing point is that AWS is keen on constantly delivering new services and 'highly innovative new features'. This has been also experienced during the work on this thesis. Not only new services and features, but also continuous price-reductions stood out. The authors of [18] write that "heavy users of AWS saw storage costs fall 20% and networking costs fall 50% over the last 2.5 years, and the addition of nine new services or features to AWS over less than one year". Furthermore, they state that "in less than two years, Amazon Web Services increased the number of different types of compute servers ('instances') from one to five, and in less than one year they added seven new infrastructure services and two new operational support options" [18].

It is important to point out that at the time of writing, some of the Amazon cloud services (e.g., SimpleDB, Relational Database Service) are still in *beta stadium*. This means that some features of the service are still evolving and that there is an eventual risk of problems that may make the service unsuitable for use in production systems [68].

Amazon offers a whole range of cloud services for various different purposes. A great convenience is that all Amazon Web Services are designed to integrate easily and work perfectly in conjunction with other AWS services. Applications running fully within the Amazon cloud (e.g., a SimpleDB request from within an application that runs in an EC2 instance) will provide near-LAN latency. Another advantage is that the data transferred between Amazon Web Services within the same region is free of charge.

The services are fully manageable with simple API calls. For this purpose Amazon offers three separate APIs: *REST*, *Query* and *SOAP* [68]. Although the underlying API operations are largely the same, each interface provides a slightly different way of interacting with a service. Based on these APIs there are several command line tools available in order to serve as client interfaces to the services. These tools help to configure and manage the services, their features and options.



To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Note: Your instances will launch in the US East (Virginia) region.

Current Status	Details
Amazon EC2 (US East - N. Virginia)	Service is operating normally

› View complete service health details

You are using the following Amazon EC2 resources in the US East (Virginia) region:

Figure 9: AWS Management Console

In addition to that there are also several programming libraries in various programming languages available, which help developers to easily integrate and interact with the Amazon Web Services from within their application. Another possibility is to access, manage and even monitor part of the cloud services from within Amazon's 'AWS Management Console' [11], which is a web-based user interface (see Figure 9).

2.3.2 Amazon EC2

The *Amazon Elastic Compute Cloud (EC2)* [3] is a Web service that provides resizable compute capacity in the cloud. It provides an environment for running servers on demand, where each virtual server is manageable just like a physical machine.

Customers are able to start as many virtual servers as needed and are able to scale the computing power up or down by using more or less powerful virtual server types. At the time of writing, Amazon offers the EC2 instance types shown in Table 1. The table lists the available instances together with the memory, compute capacity and instance storage provisioned for each of these. One *EC2 Compute Unit (ECU)* provides the equivalent CPU capacity of a 1.0-1.2 GHz '2007 Opteron' or '2007 Xeon' processor [3].

Instance Type	**Memory**	**Capacity**	**Storage**
Micro Instance	613 MB	2 ECU	
Small Instance	1.7 GB	1 ECU	160 GB
Large Instance	7.5 GB	4 ECU	850 GB
Extra Large Instance	15 GB	8 ECU	1690 GB
High-Memory Extra Large	17.1 GB	6.5 ECU	420 GB
High-Memory Double Extra Large	34.2 GB	13 ECU	850 GB
High-Memory Quadruple Extra Large	68.4 GB	26 ECU	1690 GB
High-CPU Medium	1.7 GB	5 ECU	350 GB
High-CPU Extra Large	7 GB	20 ECU	1690 GB
Cluster Compute Quadruple Extra Large	23 GB	33.5 ECU	1690 GB

Table 1: EC2 Instance Types

Amazon provides the ability to place instances in multiple locations and EC2 is currently available in the regions: US East (Northern Virginia), US West (Northern California), EU (Ireland), and Asia Pacific (Singapore) [3].

Besides the local instance storage, Amazon also offers so-called *Amazon Elastic Block Stores (EBS)*, which can be mounted as devices by EC2 instances. Such EBS volumes provide persistent block level storage for EC2 instances and are independent from the life of an instance. Amazon allows to create EBS storage volumes up to 1 TB and they behave like raw, unformatted block devices that can be seen as hard drives.

Amazon Machine Images (AMI) are complete snapshots of EC2 instances at a certain point in time. These snapshots capture the root file system, including the operating system, software, configuration and even data of an instance and serve as a boot disk when launching one.

Various pre-built AMIs for all kinds of purposes and with different pre-configured operating systems, like *Windows Server 2003/2008*, *Red Hat Enterprise Linux*, and *Oracle Enterprise Linux* are offered. Certain AMIs also come with pre-installed applications (e.g., *Apache HTTP*, *Ruby on Rails*, *Windows Media Server*) or databases. Besides using existing AMIs, users can also create images from scratch or customize existing ones. These can then be stored and even shared with other EC2 users.

The prices[1] for on-demand EC2 instances at the time of writing are shown in Table 2. To simplify matters, the listed prices are valid for machine instances, running a Windows operating system and with default configuration. Users are billed monthly

[1]These prices are correct as of November 2010. Refer to [3] to confirm the current pricing.

and the service charges are based on the number of hours that an instance is running.

Instance Type	**Price Per Hour**
Micro Instance	$0.03
Small Instance	$0.12
Large Instance	$0.48
Extra Large Instance	$0.96
High-Memory Extra Large	$0.62
High-Memory Double Extra Large	$1.24
High-Memory Quadruple Extra Large	$2.48
High-CPU Medium	$0.29
High-CPU Extra Large	$1.16

Table 2: EC2 On-Demand Instance Pricing

Besides the hourly usage fee, service charges also include charges for the data transfer, which is based on the volume of data transferred to or from an EC2 instance and currently costs around $0.10 per GB. Between Amazon EC2 and other Amazon Web Services (within the same region) there is no charge for any data transfer.

Next to the mentioned *On-Demand Instances*, Amazon also started to offer new types of instances, namely *Reserved Instances* and *Spot Instances*. Reserved instances are purchased for 1 or 3 year terms with a one-time payment. After this low one-time fee, the instance is reserved to the user, who is in turn receiving a significant discount on the hourly usage charge for that instance. With spot instances users bid on unused EC2 capacity and run those instances as long as their bid exceeds the current spot price, which changes periodically and is based on supply and demand.

The Amazon Elastic Compute Cloud is additionally offering several further features. The *Auto Scaling* option allows to automatically scale the EC2 capacity up or down according to pre-defined conditions. This ensures that used instances scale up seamlessly during demand spikes in order to maintain performance. Afterwards, auto scaling automatically scales down capacity in order to minimize costs.

Elastic Load Balancing is an optional feature that helps to automatically distribute incoming traffic across multiple Amazon EC2 instances in order to provide a greater fault tolerance.

2.3.3 Amazon SimpleDB

Amazon SimpleDB is a “highly available, scalable, and flexible non-relational data store that offloads the work of database administration” [8]. The database is designed to be simple and to minimize the administrative overhead involved in managing the data.

SimpleDB allows the user to store “small pieces of textual information in a simple database structure that is easy to manage, modify and search” [68]. Amazon stores the user’s data in a secure and redundant way and even automatically takes care of indexing the stored information, which allows to carry out queries more efficiently.

It is important to understand that SimpleDB is a key-value data store and therefore differs from a traditional ‘Relational Database Management System’ (RDBMS). On the one hand, it is not able to offer the same features (e.g., complex transactions or joins) and all the possibilities of a relational database. On the other hand, its simplicity and flexibility offer whole new perspectives to the user, when storing his data.

“A traditional relational database requires a sizable upfront capital outlay, is complex to design, and often requires extensive and repetitive database administration.” [15] Amazon states that “SimpleDB eliminates the administrative overhead of running a highly-available production database, and is unbound by the strict requirements of a RDBMS” [8]. In general there are a number of major differences between the two approaches that may be considered as either benefits or drawbacks, depending on the requirements of the application.

One of the key differences between SimpleDB and a traditional database is that in SimpleDB items are stored in a rather hierarchical structure and not a table. The three main resources provided are *domains*, *items* and *attributes*:

- *Domains*: “A domain is a named container for related data.” [68] Domains are basically used to partition data sets that are logically distinct. Queries can be executed against a domain, but cross-domain searches are not allowed. Domains are collections of items.
- *Items*: Items represent individual objects that contain one or more attribute name-value pairs. “Each item has a name that uniquely identifies it within the domain.” [68]
- *Attributes*: “An attribute is an individual category of information that is stored within an item. Each attribute has a name that uniquely identifies it within the item and it has one or more text string values associated with this name.” [68] In other words, attributes represent categories of data that can be assigned to items, and values represent instances of attributes for items.

Figure 10 shows the hierarchy for a simple database of widgets.

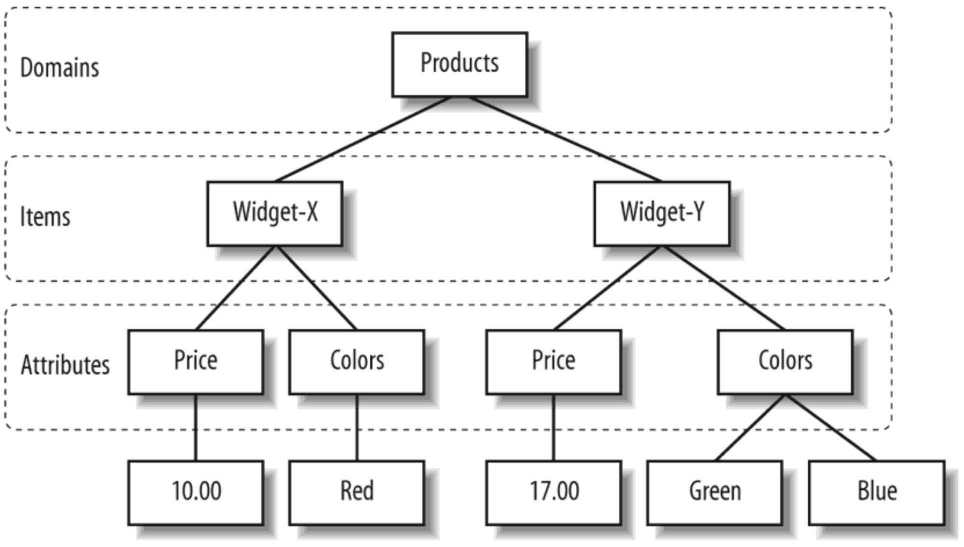


Figure 10: Example Hierarchy of SimpleDB Resources (from [68])

Because there is no predefined database or table schema, any item can have a different set of attributes from the other items. This flexibility means that the user is able to alter the structure and content of the database whenever this is needed. He is free to rearrange attributes and values as he adds new data elements or as the application changes.

For instance, an attribute called 'warranty' could without any difficulty be added to the `Widget-X`-item from Figure 10. This can be done at any time and without the other widgets needing to adapt the enhancement.

Besides benefits like simplicity and flexibility, a schema-less database requires additional awareness from the developer. It is entirely up to the developer to ensure that all stored items comply with the intended schema. Working with SimpleDB means working without the safety-net of a predefined schema and that means that the services will not complain on any mistakes being made (e.g., forgetting to store attributes, misspelling attribute names) [68].

Another significant difference between SimpleDB and a traditional database is that all data is exclusively stored as textual data. SimpleDB does, beyond text strings, not support any data types like Booleans, integers, floating-point numbers or dates. This means that the developer has to take care of proper encoding before saving his data into the database and decoding after retrieving it (e.g., negative number offsets or zero-padding).

When it comes to querying, SimpleDB is again limited to comparing the attributes based on their lexicographical ordering. In this case the simplicity of SimpleDB's data model eventually again requires additional efforts from the developer.

Amazon SimpleDB provides a simple Web services interface to store, modify, query and retrieve data sets easily. Its API will be outlined in the following [9]:

- *CreateDomain, ListDomains, DeleteDomain*: Creating, listing and deleting domains.
- *PutAttributes*: Adds, modifies or removes data within a SimpleDB domain. An item is either added, if it does not already exist, or updated by adding or updating its attributes, otherwise.
- *BatchPutAttributes*: Multiple put-operations are generated in a single call in order to achieve greater overall throughput. The operation either succeeds or fails as a whole: There are no partial puts.
- *DeleteAttributes*: Removes one or more attributes associated with an item from a domain. If all attributes of an item are deleted, the item is deleted itself.
- *GetAttributes*: Retrieves a subset or all attributes and values of a specific item.
- *Select*: A select-operation returns a set of attributes for specific items that match a given select-expression. The supported expressions are similar to the standard *SQL SELECT* statements (e.g., `select target from domain_name where query_expression`).
- *DomainMetadata*: Returns information about the domain, including when the domain was created, the number of items and attributes, and the size of attribute names and values.

Besides its huge scalability there are some limitations according the data storable in SimpleDB. Currently, individual domains are enabled to hold a maximum of 1 billion attributes and to grow up to 10 GB each. Furthermore, per default each user is initially allocated a maximum of 100 domains, which is still enhanceable upon request. Other constraints limit the length of attribute names and values, and restrict 256 attribute-value pairs per item [9].

Users of SimpleDB are billed monthly based on three criteria: the amount of storage they have used ($0.25 per GB-month), the numbers of hours of machine utilization time their operations have consumed ($0.14 per machine-hour consumed), and the amount of data transferred into or out of the service (sliding scale, ranging from $0.08 to $0.15 per GB).[2]

[2]These prices are correct as of November 2010. Refer to [8] to confirm the current pricing.

As with other Amazon Web Services, there is no minimum fee and no long-term commitment. In addition to that, SimpleDB currently offers a 'free tier' where the users pay no charges on the first 25 utilized machine hours and 1 GB of storage.

2.3.4 Amazon RDS

The *Amazon Relational Database Service (RDS)* [5] is a Web service that provides an industry-standard relational database in the cloud. RDS makes it easier to set up, operate and scale the database server. Furthermore, it manages common time-consuming database administration tasks, automatically takes care of backs up, and maintains the database software.

In contrast to Amazon's other database services, RDS offers a fully-featured *MySQL 5.1* [70] database. The advantage is that users already familiar with MySQL can without any difficulty replace their local MySQL database with an RDS instance. Programming code, applications and tools that are already in use with the existing MySQL database work seamlessly with Amazon RDS.

At the time of writing, Amazon offers the RDS instance classes shown in Table 3. One *EC2 Compute Unit (ECU)* provides, as with the EC2 instance types, the equivalent CPU capacity of a 1.0-1.2 GHz '2007 Opteron' or '2007 Xeon' processor [5].

Instance Type	**Memory**	**Capacity**	**Price**
Small DB Instance	1.7 GB	1 ECU	\$0.11
Large DB Instance	7.5 GB	4 ECU	\$0.44
Extra Large DB Instance	15 GB	8 ECU	\$0.88
High-Memory Extra Large DB	17.1 GB	6.5 ECU	\$0.65
High-Memory Double Extra Large DB	34 GB	13 ECU	\$1.30
High-Memory Quadruple Extra Large DB	68 GB	26 ECU	\$2.60

Table 3: RDS Instance Classes

For each instance class, RDS provides the ability to select from 5 GB to 1 TB of associated database storage capacity.

The prices listed in the table are to be paid for the compute capacity by the hour the database instance runs. In addition to that, users are charged a storage rate (\$0.10 per GB-month) and a data transfer rate for data transferred into or out of the service (sliding scale, ranging from \$0.08 to \$0.15 per GB).[3]

As with the Elastic Compute Cloud, Amazon RDS also offers the possibility to

[3]These prices are correct as of November 2010. Refer to [5] to confirm the current pricing.

purchase reserved database instances. The user makes a one-time up-front payment for a database instance and reserves it for a one- or three-year term. The result is that he is charged a significantly lower rate for its utilization.

Furthermore, two different replication features are offered with RDS: *Multi-AZ Deployments* help to enhance the availability and protect against unplanned outages. *Read Replicas* make it easy to scale out beyond the capacity constraints of a single database instance for read-heavy database workloads.

3 Related Work

In this chapter an overview over the work that was recently done by the scientific research community and is related to the topic of this thesis will be given. It can be identified that the concerned literature dealing with commercial cloud computing offerings mainly focuses on evaluating the usage of these. It either gives a general analysis over the offered functionalities and possibilities, or compares them according to performance and costs.

Similar to a large part of this thesis, there are also a couple of papers that especially focus on cloud databases and even deal with the possibility of featuring the transactional guarantees of a (traditional) relational database in conjunction with simple key-value stores in the cloud.

One section of this chapter will mention some popular uses of the AWS services.

It has to be mentioned that many papers focus on and cover the *Amazon Simple Storage Service (S3)* [7]. S3 is a simple web storage and it is primarily intended for the storage of data objects (e.g., media files). It provides the ability to store large amounts of data reliably and with high availability.

The S3 storage model is a simple two-level hierarchy, where the users may create *buckets* and place data *objects* (up to 5 GB each) into these buckets. Strings are used as keys for both buckets and objects.

3.1 Evaluation and Performance Analysis

In the following several works that evaluate the use of cloud computing will be mentioned. In general, the goals of these works are similar to the intention of Section 6.1, which deals with the quantitative evaluation of the used cloud services. It has to be mentioned that the concerned literature especially focuses on particular subjects, like the evaluation of the use of cloud computing services for scientific computing. Many of the cited papers also include evaluations that address further aspects of computing in the cloud, than were within the scope of this thesis (e.g., cost analysis).

In [59] the authors investigate how they can build a web server farm in the cloud. A dynamic switching architecture that dynamically switches among several configurations depending on the detected workload and traffic pattern is proposed. They discuss the switching criteria and how to use the cloud's dynamic capability to implement the architecture. Furthermore, the authors present a benchmark performance study on various cloud components and show that the performance results reveal several limitations. They conclude that there is currently not a single configuration that can satisfy all traffic scenarios.

The use of clouds in a way that strikes the right balance between cost and performance is discussed in [29]. The authors study the cost/performance trade-offs of different execution and resource provisioning plans. Their results show that by provisioning the right amount of storage and compute resources, costs can be reduced with no significant impact on the application performance. Furthermore, they state that cloud computing offers a cost-effective solution for data-intensive applications, because they have experienced that the storage costs were insignificant compared to the CPU costs. As the cloud field matures, they expect to see a more diverse selection of fees and quality of service guarantees for the different resources and services provided by clouds.

There are several papers that evaluate the use of cloud computing services for scientific computing. Such a comprehensive performance analysis of the EC2 service for scientific computing is given in [71]. The results of the paper indicate that the current cloud services need an order of magnitude in performance improvement to be useful to the scientific community. Besides the performance, the authors also write that the reliability of the tested cloud is rather low. They conclude that although computing in the cloud is insufficient for scientific computing at large, it may still appeal to scientists that need resources immediately and temporarily.

A similar evaluation is found in [46], where a case study of the use of EC2 for scientific computing is presented. The author evaluates the cloud service according its performance, cost and even usability (learning, remote access, ease of use). The case study concludes that EC2 provides a feasible and cost-effective model in many application areas. Although he stresses the advantages of services like EC2 and S3, the author's opinion is that these services will not replace dedicated clusters or large shared super-computer facilities.

After studying the use of cloud computing for scientific workflows in [52], the authors conclude that clouds are a viable alternative for running scientific workflow applications and that the performance is reasonable given the resources available. They are also satisfied with the costs and only critic the relatively high costs for data transfer.

In [21], the authors state that traditional benchmarks are not sufficient for analyzing the novel cloud services. In their paper they present some initial ideas of how such a new benchmark should look like in order to better fit to the characteristics of cloud computing (e.g., scalability, pay-per-use, fault tolerance). Because dynamic allocation and de-allocation of resources as well as the pay-as-you-go model are inherent features of these services, an important difference to most existing benchmarks is that this new cloud benchmark should not require using a static configuration of software and hardware components.

3.2 Cloud Databases

Besides the evaluation of cloud offerings in conjunction with concrete scenarios, many authors specifically deal with the analysis and the usage of cloud database services.

[72] evaluates S3's ability to provide storage support to large-scale science projects from a cost, availability, and performance perspective. The paper evaluates whether S3 is a feasible and cost-effective alternative for offloading storage from in house maintained mass storage systems for today's scientific collaborations. Besides an independent characterization of S3 in terms of user-observed performance, the costs of outsourcing the storage functions to S3 are also evaluated. Furthermore, the authors also discuss the functionality and security features of the service. They conclude that the current S3 design needs to be improved and give recommendations for the next-generation storage utility services.

In order to evaluate S3, the authors used the Amazon EC2 service, which was also used during the work on this thesis. They state that the primary relevance of EC2 is that there are no bandwidth charges for data sent between services within the cloud and that as a result the data saved within the Amazon cloud can be "cheaply processed using virtual EC2 hosts" [72]. It is also mentioned that the "observed availability from EC2 was quite high" and that their "experience using EC2 was excellent": "Although Amazon makes no guarantee regarding the durability or availability of any EC2 instance, we ran instances for a total of 6 CPU months and had only one case in which an instance was rebooted without our initiation." [72]

The recently published paper [55] evaluates the current state-of-the-art of cloud database services – including Amazon SimpleDB, Amazon RDS, Amazon S3, and others. The questions raised by the authors are: How well do the offerings scale with an increasing workload? How expensive are these offerings and how does their cost/performance ratio compare? How predictable is the cost with regard to changes in the workload?

The authors state that even though the cloud services look similar from the outside, they vary dramatically when it comes to end-to-end performance, scalability and cost: "While most (traditional) general-purpose database systems (e.g., DB2, MySQL, Oracle 11, Postgres, SQL Server) share roughly the same 'textbook' architecture and data structures, the differences in the implementation of cloud services are immense" [55].

In order to carry out their tests with the Amazon SimpleDB service, they needed to adapt certain layers of their test application. While doing so, they describe what was also experienced in a similar way during the work on this thesis: “Since SimpleDB does not support SQL, SQL operators such as joins and aggregation had to be implemented at the application level. To do so, we implemented a (Java) library with these SQL operations and manually optimized SQL queries (i.e., join orders and methods). Obviously, this approach resulted in shipping all the relevant base data from SimpleDB to the application servers and resulted in poorer performance as the query shipping approach supported by full-fledged SQL database systems.” [55]

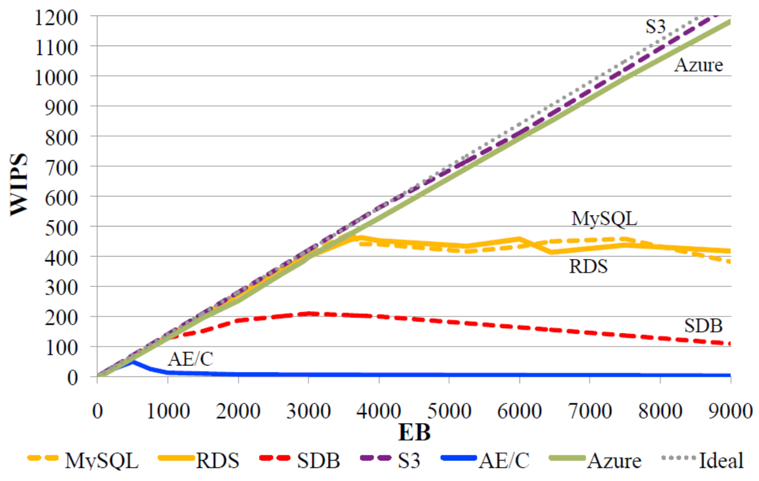


Figure 11: Comparison of Architectures (from [55])

After presenting the different offerings and vendors, a comparison of the architectures (see Figure 11) is presented, next to a detailed cost analysis. The authors state that the current cloud market is still immature and that they observed that most services had significant scalability issues. They stress that “S3 is the only variant that is based on an architecture that has no bottlenecks” [55], which is also reflected in the figure. It still needs to be mentioned that in contrast to S3, the Amazon services SimpleDB and RDS are still in beta stadium. The authors admit that “the fundamental question of what the right data management architecture for cloud computing is, could not be answered” [55].

When revisiting distributed database architectures that are nowadays used in cloud computing, the authors of [55] present how caching can be integrated at the database server layer (see Figure 12) and write that “caching can also help the cloud computing promises with regard to cost and scalability” [55]. The described concept of caching is similar to the one used in this thesis.

Although S3 is mostly used to store multimedia documents (e.g., videos, photos, audio), which are shared by a community of people and rarely updated, the purpose

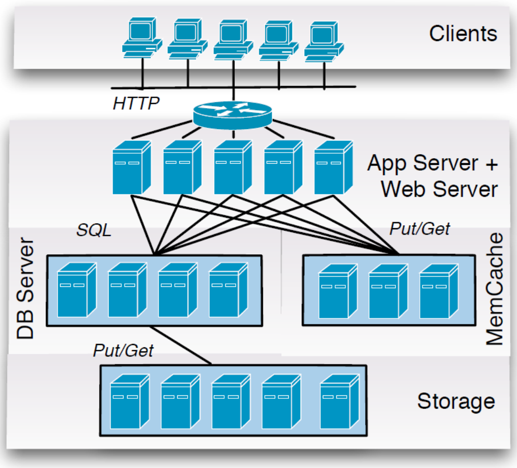


Figure 12: Distributed Database Architecture and Caching (from [55])

of paper [22] is to "demonstrate the opportunities and limitations of using S3 as a storage system for general-purpose database applications which involve small objects and frequent updates". Read, write and commit protocols are presented, and furthermore, the cost, performance and consistency properties of such a storage system are studied. The authors state that because the majority of the cloud databases were designed to be cheap and highly available, they are often slow and sacrifice consistency. Their work's goal is to preserve the scalability and availability of a distributed system like S3 and to achieve the same level of consistency as a database system.

After evaluating their implementation, they observed high execution times and write: "These high execution times, however, were expected [...] S3 has about two to three orders of magnitude higher latency than an ordinary local disk drive [...]. Despite these high execution times, we believe that the results are acceptable in an interactive environment such as a Web 2.0 application." [22]

The authors of [28] have a similar intention. They state that although services like SimpleDB are highly scalable, they do not provide transactional guarantees. In their paper they propose *ElasTraS*, which stands for 'Elastic Transactional Data Store' and is a data store that is "elastic along the same lines as the elastic cloud, while providing transactional guarantees" [28].

3.3 Popular Uses of AWS

In this section two of the most prominent and often cited uses of the AWS services will be presented.

Further case studies and businesses that have successfully applied the Amazon Web Services are listed and presented under [13].

3.3.1 Animoto

The early utilizations of cloud computing were web applications that were developed by companies with only limited IT resources. Public cloud services were used in order to rent a flexibly scalable infrastructure with low entry barriers and at low costs.

One of the prime examples of the advantages of cloud infrastructure is the success of *Animoto* [17]. Animoto is a web application that “automatically produces beautifully orchestrated, completely unique video pieces from your photos, video clips and music” [17]. After users provide content to the service, Animoto analyzes it and generates an animated video. In the case of Animoto, cloud services were used to handle an unexpectedly high demand by its users.

The start-up company was founded in 2006 and its web application is built on top of various Amazon Web Services, including Amazon EC2 and S3 [16]. Before its actual breakthrough, the company used to serve an average of about 5 000 users a day with at most two video renderings per minute [47]. The company solely relied on Amazon’s servers and utilized around 50 EC2 machine instances at that point. Animoto does not host any servers, but rather outsources its computing power to the Amazon cloud.

After the company started to advertise its web application with the help of ‘social media’, the service went ‘viral’ and the demand for it dramatically increased over night. Within only three days the company had to serve up to 750 000 users and at one point even 25 000 people used Animoto within a single hour [47]. At the peak, 450 videos were rendered per minute and the company’s EC2 server usage increased in a way that they had to scale from 50 up to 3 500 server instances (see Figure 13).

Figure 13 also shows that there are some significant drops in the graph representing the demand. This means that the Amazon services are able to automatically scale-down, if there is less demand for computational power (e.g., middle of the night). As Amazon follows a strict pay-as-you-go policy, Animoto does not have to further care or pay for the servers after they were un-deployed and returned back to Amazon.

The authors of [59] write that “such a dramatic change in the infrastructure requirement would mean either gross underprovisioning or gross overprovisioning if a fixed set of capacity is provisioned”. Furthermore, they conclude that “an infrastructure

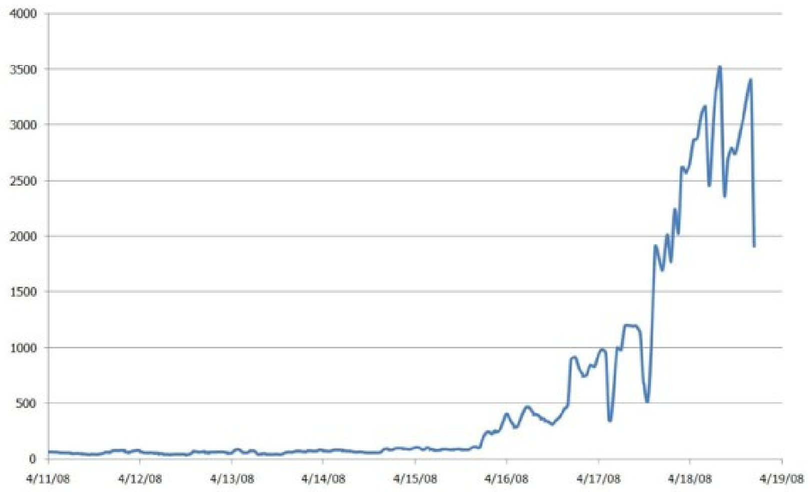


Figure 13: Animoto's EC2 Instance Usage (from [16])

cloud, such as Amazons EC2/S3 services, is a promising technology that can address the inherent difficulty in matching the capacity with the demand" [59].

One of the lessons learned from this example is that cloud computing can apparently help companies to be prepared for their immediate breakthrough, without having to fear that they are under-provisioning and in this case maybe even missing their opportunity for success. Furthermore, with computing in the cloud there is opposed to traditional infrastructure no need to plan and acquire any physical resources, which demanded capacity could not be foreseen anyway.

As it is practically not able with the use of traditional infrastructure to scale in the degree of Animoto's case, cloud services in fact especially clear the way for emerging start-up businesses.

3.3.2 TimesMachine

A further often cited use of the Amazon Web Services is the *TimesMachine* project by the 'New York Times' [88]. The idea behind TimesMachine is to make archived news articles from the years 1851 to 1980 available through the Internet.

In the initial step, around 11 million articles (4 TB of data) were available as scanned TIFF images and the goal was to create PDF files based on these images. To generate the PDF version of an article, the numerous TIFF images of which it is composed of needed to be scaled and composed accordingly.

The conventional approaches to generate PDF files from the scans of the articles proved to be very slow and expensive. For processing such an amount of data, new servers would have needed to be acquired and the whole project would have taken months [20]. The use of cloud services promised to process the whole articles within days and for only a fraction of the costs.

The 4 TBs of source data were uploaded to the Amazon S3 service and code was written in order to run on numerous EC2 instances simultaneously. After reading the source data and creating a PDF, the program would store the result back into S3. Around 100 EC2 instances were used during the project and it was able to process all 11 million articles in just under 24 hours [87]. While processing, another 1.5 TB of resulting PDF files were generated.

The TimesMachine project is a prime example for a one-time batch job, which would maybe not be profitable without the use of cloud infrastructure. The project team achieved to successfully carry out the generation of the PDFs in a cost-efficient way, without the need to invest in potentially unprofitable hardware.

4 Background

The Web service runtime environment, *VRESCo*, upon which has been worked during this thesis will be presented in the following. After describing its architecture and the underlying data model, some of its features will be covered in more detail.

4.1 Vienna Runtime Environment for Service-oriented Computing (VRESCo)

As described in the previous section, three roles are needed to fulfill the SOA triangle. In [66] the authors claim that in practice most of the SOA applications only rely upon the service provider and the service requestor – neglecting the role of a service registry. By leaving out the service registry, the exchange of the service information needs to already happen at design-time. Since the service requestors need to know about the exact static address of the service, this procedure is obviously contrary to the core principles of Service-oriented Architectures (SOA).

Another significant issue for SOA is the general shortcoming of current service registries. Although Web service registries, like UDDI [37] and ebXML [38] have gone through standardization processes and are considered as standards, the reality is that both are rarely used. In fact, the public UDDI registry, which was hosted by companies like IBM, Microsoft and SAP, has been shutdown in 2005 [66].

To fill a gap in current Service-oriented Architectures and to 'recover the broken SOA triangle' the authors of [65] and [66] came up with the idea for the *'Vienna Runtime Environment for Service-oriented Computing' (VRESCo)*. VRESCo has been developed on the 'Vienna University of Technology' and aims to solve some of the current challenges in Service-oriented Computing. Among other features it includes support for service versioning, service composition, dynamic binding and dynamic invocation. Furthermore also dynamic searching, querying and notification are supported. In this section the basic architecture of VRESCo will be presented alongside some of its basic principles and features.

4.1.1 Architecture

The overall system of VRESCo is implemented in the C# programming language and uses the 'Windows Communication Foundation' (WCF). Figure 14 shows an overview of the VRESCo architecture. The *VRESCo Runtime Environment* is a server application that is invocable using the *VRESCo Client Library*. The VRESCo

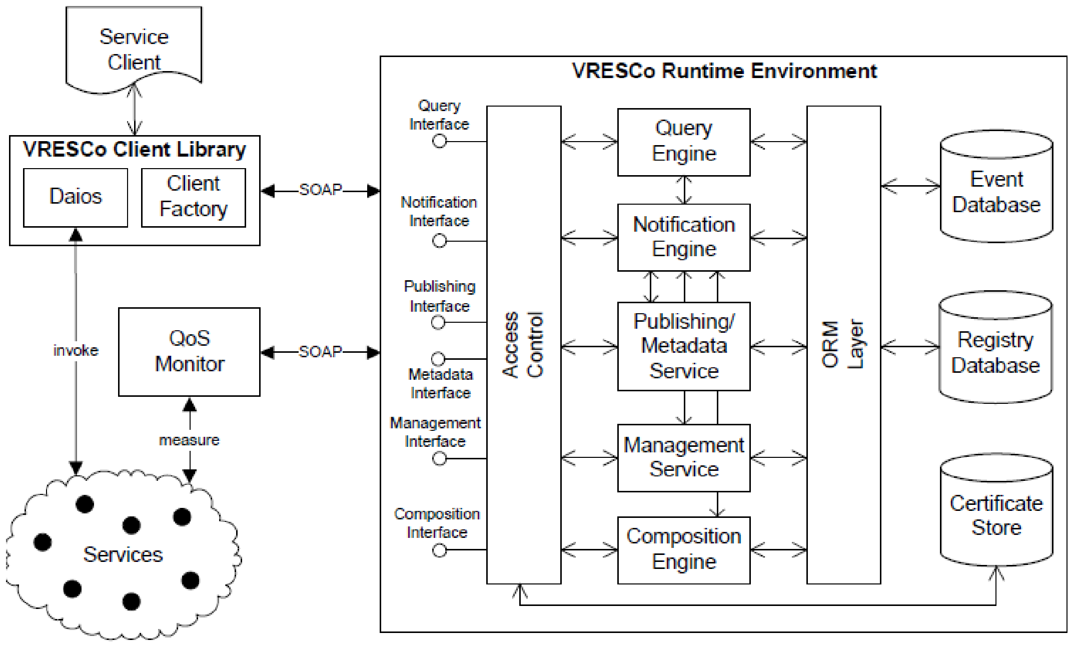


Figure 14: VRESCo Architecture Overview (from [65])

runtime environment is built on top of various services and engines. Each of the core services is exposed as a Web service and could be accessed either by using SOAP or the client library.

In the following, the core services and components of which the VRESCo runtime environment is composed of will be presented in short:

- *Query Engine*: The query engine is used to query the registry database and search for published services. Querying could be either used to find matches within the service descriptions and their attributes (e.g., quality of service attributes) or to find services using a full text search. There is also a specialized query language offered, namely the *VRESCo Query Language (VQL)* [56]. VQL queries are based on the entities and relations of the data model and therefore abstract from the concrete database schema that is used to store the model entities.

- *Notification Engine*: The notification engine informs users about events that occur during runtime [48]. The user is either notified per E-Mail or Web service notification whenever certain events of interest occur. Possible events are the availability of a new service, a change in a service interface or the change of the QoS of a service [64].

- *Publishing/Metadata Service*: This service is used to publish services and metadata into the registry. The service offers two different interfaces for either adding entries to the *metadata model* or for registering a service description using the *service model*. The difference and purpose of these two models will be described in Section 4.1.2.

There is a clear distinction between services and revisions in the VRESCo service model. A service is available in one or more revisions and a service revision contains all the information (e.g., reference to the WSDL file) that is necessary to invoke a service. Service revisions are the basis for the service versioning, supported in VRESCo. It is able to branch and merge service revisions and to tag them according to their current version (e.g., custom tags or default tags, like `INITIAL`, `LATEST`, `DEPREC`, etc.) [57].

- *Management Service*: The management service is responsible for managing and storing user information, like names and passwords. It furthermore also handles the access rights to the VRESCo runtime environment.
- *Composition Engine*: The composition engine provides mechanisms to compose services by specifying hard and soft constraints on QoS attributes. A service composition is conducted by a composition request, which is specified in a domain specific language. Such a request specifies the functional requirements for the features, their composition and certain non-functional requirements (e.g., QoS values) [81, 82].

4.1.2 Service Metadata Model

In a SOA it is usual that service functionality is offered by several different service providers. Although the basic functionality of these services is the same, they may still differ in their QoS attributes (e.g., response time, availability, price) and would therefore not be equally interesting for the service consumer. These services may furthermore even differ in their technical implementation. The service could have different input and output parameters, where the parameters differ in type, order or number.

The result is that the service consumer needs to decide which specific service he wants to use at design time in order to know how to interact with it in his application. Because there is no generic way to interact with these services, the technical side of the interaction with a service needs to be 'hard-coded' within the application. It could eventually result in disadvantages for the service consumer, when during run-time a service provider registers a new service that is better according to its QoS attributes (e.g., cheaper). In order to use it, the service consumer would need to adapt its application code, which is certainly not desirable.

To overcome the described issue and to decouple the service consumer from a specific service, VRESCo introduces a new abstraction layer to hide the technical service details. The result is that the service consumer does not build its application against a concrete service, but rather against an abstract service, which offers the desired functionality. Another advantage is that because the service consumer interacts with only the abstract version of a service, an eventual exchange of the concrete service is transparent.

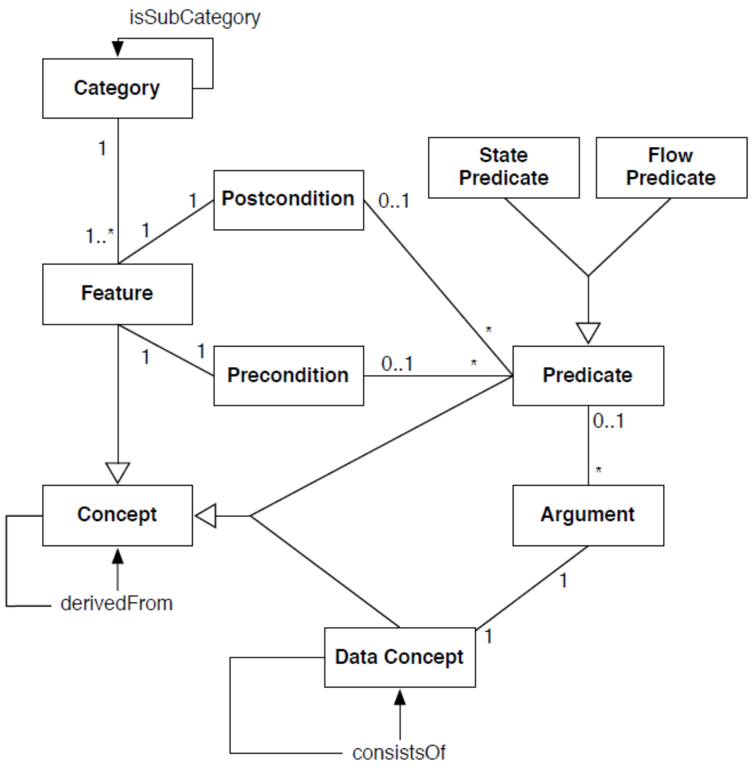


Figure 15: VRESCo Metadata Model (from [83])

Figure 15 shows the basic metadata model used in VRESCo to model services, their features, pre- and postconditions. A detailed description of the model and its building blocks is given in [83].

If a service provider registers a service to VRESCo, it needs to be mapped to the metadata model. The mapping between a concrete service model and the metadata model is depicted in Figure 16. Although the service model, which basically follows the Web service based notation as introduced by WSDL, and the metadata model in VRESCo are different, they can both be transformed into one another. Services are grouped into categories and service operations are mapped to features, as they represent concrete implementations of it. The input and output parameters of the service operations are mapped using data concepts. Again a more detailed description of the mapping between the two models can be found in [83].

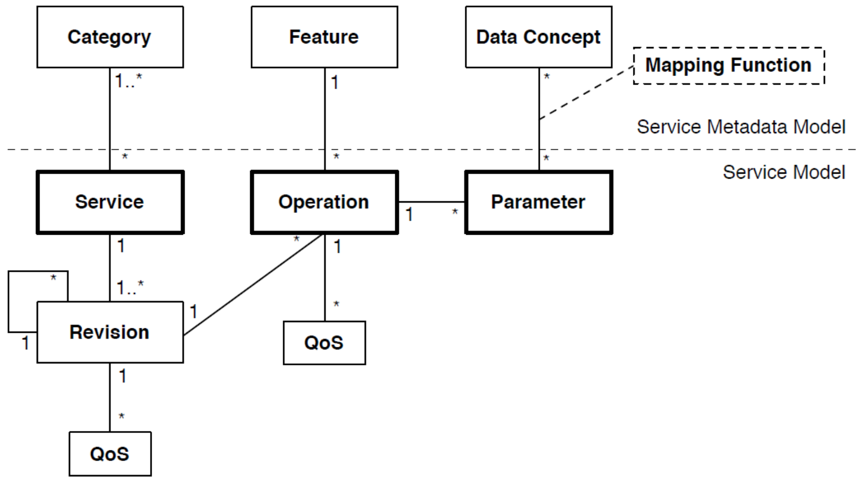


Figure 16: Mapping of Service Model to Metadata Model (from [65])

In Figure 16 it can be also seen that besides the functional attributes described in the model, there is also a set of QoS attributes associated with each service revision and service operation. The QoS attributes could be either specified manually or measured automatically using the QoS monitor, which is integrated into VRESCo and was introduced in [84]. The pre-defined QoS attributes are *Price*, *Reliable Messaging*, *Security*, *Latency*, *Response Time*, *Availability*, *Accuracy* and *Throughput.* Since the QoS attributes are part of the service model, QoS-related criteria can be included in VQL queries, which aids QoS-based service selection [48].

4.1.3 Dynamic Binding

The support for dynamic binding was one of the core motivations for the VRESCo project. *Dynamic binding* (or *late binding*) is the process of linking an abstract service to a concrete service instance at execution time [66]. The idea is that the service consumers can at runtime search for a service with the desired functionality and criteria (e.g., QoS attribute), and dynamically bind to it. Furthermore, this rebinding to a more suitable service (on change of QoS attributes, etc.) should be transparent to the service consumer.

The transparent selecting and dynamic binding to concrete service instances according to some QoS criteria is referred to as *QoS-based dynamic binding* [66]. In practice a QoS monitor (see Figure 14) monitors the QoS attributes of a number of services that are running on one or more hosts [84]. The dynamic binding mechanism is then used to select a specific service, which is best fitting according to the specified selection strategy. VRESCo currently offers various rebinding strategies, like *Periodic*, *OnDemand*, *OnInvocation* and *OnEvent*. Each of these strategies has different advantages and disadvantages, which are broader discussed in [48].

4.1.4 Dynamic Invocation

The invocation of services is usually done in a very static manner. Alone the need to generate stubs out of a WSDL service description in order to invoke a service operation is the evidence for a lack of dynamism of current Web service technologies. The goal is to invoke a service by just using its endpoint, the operation name and the required input message. Because there is no dynamic way in doing this, the result is that the applications are very inflexible, since the service invocation needs to be specified at design-time.

The currently predominant Web service implementations rely on service stubs, which leads to less dynamic applications and does not comply with the initial idea of a SOA. To overcome this issue, another goal of VRESCo is to support the stubless, protocol-independent and message-driven invocation of services.

Dynamic invocation in VRESCo is achieved by using the DAIOS ('Dynamic, Asynchronous and Message-oriented Invocation of Web Services') framework [58]. The idea of this framework is to provide a stubless way to use services and to hide their technical implementation details. The communication is carried out using DAIOS messages, which are then transformed to a format that is understood by the service. DAIOS additionally features the support for one-way and asynchronous communication. A detailed presentation of DAIOS is not within the scope of this thesis and the interested reader should be referred to [58] for more details.

5 Design and Implementation

The overall goal of this thesis is to deploy the web service runtime environment VRESCo, which has been introduced in Section 4.1, into the cloud. The motivation behind this is to benefit from the various advantages of computing in the cloud. These benefits were already covered in Section 2.2.3 and include massive scalability, no maintenance and administration efforts, and the overall inexpensiveness of the solution.

To achieve this goal, Amazon has been chosen as the vendor for the cloud services used during the work on this thesis. The decision to use the Amazon Web Services (see Section 2.3) is based on their popularity and the variety of the offered services. Amazon is not only the largest vendor for commercial and public cloud offerings, but it also offers a comprehensive cloud computing platform (compute, database, storage, messaging, payment services, etc.).

Besides just deploying the regular VRESCo server application into the Amazon cloud and running it from within a virtual machine instance, the underlying database has to be considered as well. There are at least three different options of how to deploy VRESCo's database into the Amazon cloud:

- *Running a database server (MySQL [70], etc.) from within an Amazon EC2 machine instance and managing the database remotely.* This requires launching an EC2 instance and manually installing and starting a database server. Besides the benefits that EC2 offers as a scalable virtual machine instance, the database still has to be administered and maintained (e.g., patching, indexing, performance tuning) by the user.
- *Using the SimpleDB service, which features a simple key-value data store in the cloud.* This opens new perspectives and offers various significant advantages, like flexibility, enormous scalability and inexpensiveness. A drawback of this approach is that SimpleDB's data model is only based on key-value pairs and is certainly not relational. It does not offer the same features as an RDBMS and requires a lot more work from the developer in order to compensate for its simplicity.
- *Swapping the database to the fully-relation database service Amazon RDS.* The Relational Database Service is a self-managed and fully-featured MySQL database in the Amazon cloud. The advantages and disadvantages of this approach are both conditional upon the fact that RDS is a relational database. On the one hand, this service features all the common features of a traditional relational database (e.g., transaction support). On the other hand, it is

restricted in terms of flexibility and eventually even scalability.

The first option is rather trivial and is not going to be considered in this thesis. The remote server and its enormous scalability are great, but because it takes the user to administer and maintain a database, it does not take advantage of the cloud to its full potential. In the remainder of this chapter, the latter two solutions will be presented and it will be successively shown how VRESCo was deployed into the Amazon cloud.

The differences between the mentioned database approaches are once again summed up in Table 4. It can be seen that although the hardware is automatically provisioned in all three approaches, only with the services SimpleDB and RDS automatic software updates and data backup/replication is carried out. Additionally to that, with the SimpleDB service, the cloud even takes care of data indexing and query tuning.

Db on Amazon EC2	**Amazon SimpleDB**	**Amazon RDS**
Automated hardware provisioning	Automated hardware provisioning	Automated hardware provisioning
User-controlled software updates/patching	Automated software updates/patching	Automated software updates/patching
User initiated backups or snapshots	Automated geo-redundant replication	Automated backups (administered by user) and user initiated snapshots
User responsible for indexing, query tuning	Automated indexing, query tuning	User responsible for indexing, query tuning

Table 4: Differences Between AWS Database Solutions (from [2])

After the deployment of the database, the deployment of the overall VRESCo server application will be covered. The application is going to be run from within an Amazon EC2 instances and will interact with the databases from within the Amazon cloud.

5.1 Using Amazon SimpleDB as Database

SimpleDB and its underlying data model have been described previously in Section 2.3.3. It propagates the essence of 'real' cloud computing by providing massive scalability and a strict pay-per-use policy. Furthermore, it is managed by the cloud provider and there is no need for any installation, update or data indexing efforts.

On the one hand the goal of this approach was to fully substitute VRESCo's relational database with the SimpleDB service. On the other hand the objective was to support the features that were offered by the initial database and that were already used by the application. This was overall successfully achieved with the exception of a few limitations (transaction support, etc.) that are going to be mentioned below.

The current implementation that enables the usage of SimpleDB as a substitution for the previously used relational database can not be seen as a library. It is specialized to work in conjunction with VRESCo's data object model and has been adjusted to the application in order to exhaust its possibilities and features. There is also no existing 'rdbms2simpledb'-library that aims for the same objective, and features the same complexity and power of the current solution.

To integrate SimpleDB into VRESCo, which is implemented in the C# programming language, and to use its services from within the application, parts of the 'AWS SDK for .NET' library [12] offered by Amazon were used.

To improve the solutions performance and to maximize its efficiency, an additional caching layer was implemented. This managed to significantly improve the applications performance, when using SimpleDB as the database and running the VRESCo server from outside the Amazon cloud.

It has to be mentioned that an overall goal of this implementation was to mainly focus on the application's Data Access Layer (DAL). Further programming logic and services were not meant to be adjusted in order to achieve better performance and efficiency. The reason was to keep VRESCo separated from the decision of what database is actually going to be used and to introduce the possibility to flexibly choose the desired database paradigm (either an RDBMS or SimpleDB). This is achieved by introducing a 'switch' (`key='SimpleDB' value='on'`) in the application's configuration file, which makes the end user decide about the actual database that is going to be used.

Besides the work on the DAL, also parts of VRESCo's query engine were enhanced in order to feature support for VQL (see Section 4.1.1) when using SimpleDB as the database. It has to be mentioned that there is currently no full support for VQL in conjunction with SimpleDB and only the processing for very basic VQL-queries is possible.

5.1.1 Extending the Data Access Layer

VRESCo was initially built around a relational MSSQL [70] database. In order to use SimpleDB's key-value concept, the Data Access Layer had to be enhanced and several *Data Access Objects (DAO)* [86] needed to be introduced.

The purpose of the DAO software design pattern is to hide actual data source implementation details from the client. The interface exposed by the Data Access Object to the clients does not change when the underlying data source implementation changes. This allows to make adjustments on the Data Access Layer in order to support different storage schemes without affecting its clients or business components [1].

VRESCo's Data Access Layer currently exposes a generic interface (`IGenericDAO`) with common methods for storing, retrieving and deleting objects from a database. The interface and its featured methods were implemented for an abstract data object in the `AbstractGenericDAO` to support the interaction with a relational MSSQL database (see Figure 17).

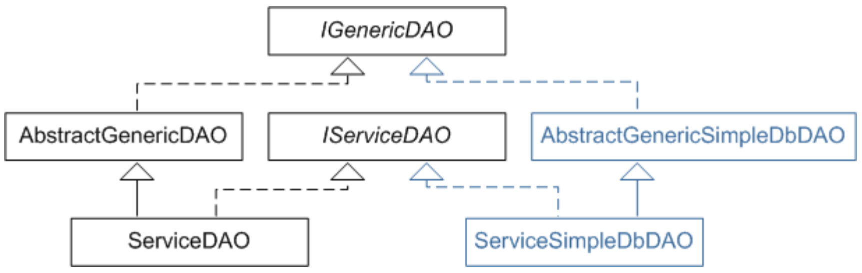


Figure 17: Extended VRESCo DAO Model

Additionally, VRESCo contains several specific Data Access Objects for every object type featured in its service model (see Section 4.1.2). Each of these model DAOs extends the generic `AbstractGenericDAO`, and implements further methods that are specific for the data access of the concerned model object and that are exposed by its DAO-interface.

In Figure 17, an extract is shown that depicts the described relation of the DAOs for the `Service` object: the `ServiceDAO` extends the `AbstractGenericDAO` and implements the methods exposed by its `IServiceDAO` interface.

In order to feature SimpleDB as a database, an enhancement needed to be made by introducing new DAO instances that specifically handle the interaction between VRESCo's business logic and the new database paradigm. This has been achieved by adding a new generic DAO (`AbstractGenericSimpleDbDAO`), which implements the exposed generic DAO-operations in conjunction with SimpleDB and represents the core piece of this solution.

Besides that, for every featured object type, several model related DAOs had to be introduced. These DAOs again handle the model object specific database operations, but are intended to interact with using SimpleDB as data store. Figure 17 shows the described extensions by depicting the newly introduced and SimpleDB specific

DAOs (`AbstractGenericSimpleDbDAO` and `ServiceSimpleDbDAO`) on the right side of the outline of VRESCo's initial Data Access Layer. Again, for just illustrational purposes, only a small extract is shown by focusing on the `Service` type and leaving the others out.

5.1.2 Mapping the Relational Data Model to SimpleDB

Within the original Data Access Layer, the properties of the relational model objects and their relations were annotated with NHibernate attributes [19, 69] in order to make this information accessible to NHibernate when interacting with the underlying database. The objects are stored to and retrieved from the database according to the annotated attributes.

With the introduction of SimpleDB as the database, the goal was to fully feature the original relational service model. The challenge was to extend the Data Access Layer in order to support the mapping of the existing relational data model to SimpleDB's rather primitive key-value paradigm. To achieve this, several NHibernate attributes were reused in order to not need to alter the existing model objects and to be able to handle the object relations within the new SimpleDB-DAOs.

The primary question was to determine how to generally save an object instance within the SimpleDB data model, which was described earlier in Section 2.3.3. In order to achieve the best result, several alternatives needed to be considered and evaluated according their scalability, efficiency, and danger to conflict with SimpleDB's limitations. There are basically three possible ways to store objects in SimpleDB. The different approaches are depicted in Figure 18 and are described in the following:

- *Use separate domains for every object type and store each object instance within its own separate item in the corresponding domain.* The effect is that the data stored within SimpleDB is distributed over several domains. This approach is recommended by Amazon because it promises to be efficient. Furthermore, it is, compared to the other alternatives, more intuitive and less complex, which allows the database operations to be simpler and even faster.

  Currently, SimpleDB does not support cross-domain searches (e.g., joins within a select-statement). This makes it necessary to carry out multiple consecutive queries in order to query data that is distributed across several domains. The result is that retrieving objects that for example relate to others, eventually lacks in efficiency and in performance.

- *Save all objects in only one domain ('vresco') and store each object instance within its own separate item.* This means that all objects (e.g., `s1`, `s2`, `sc1`,

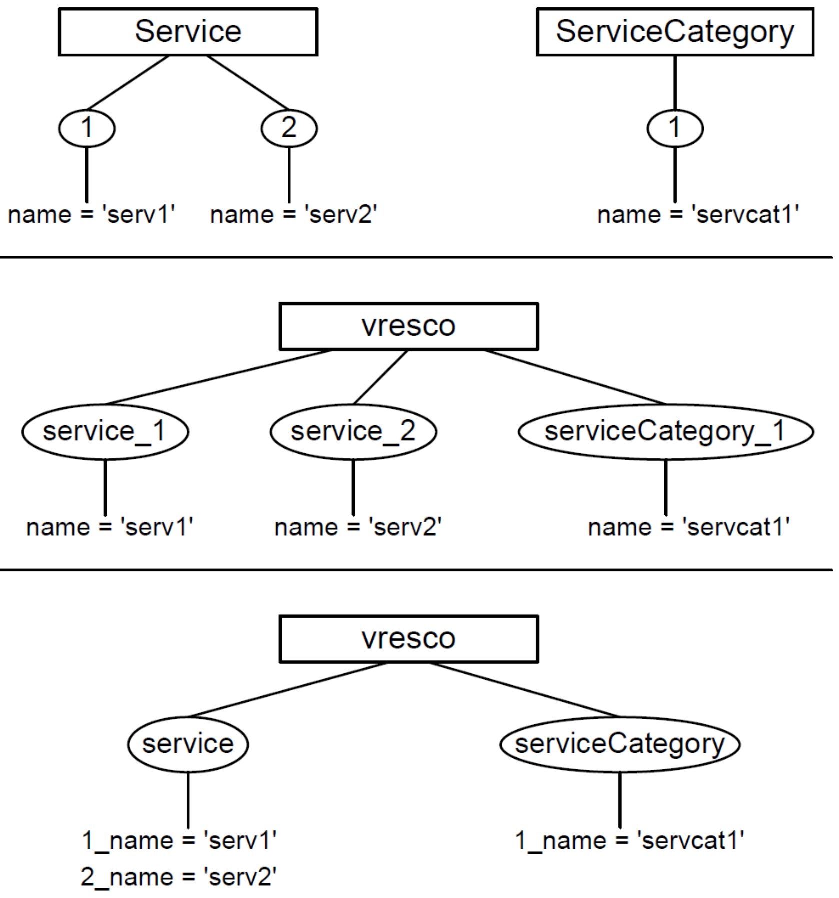


Figure 18: Different Approaches to Store Objects into SimpleDB

etc.) are stored within different items. Every item additionally needs to carry a prefix (the name of the object's type) in order to carry the information about its corresponding object type (e.g., 'service_1', 'service_2', 'serviceCategory_1', etc.).

The major benefit of this approach would be to allow queries across multiple object types. Opposed to the first approach, everything is saved in only one domain and a single select-statement would be now able to query all stored data.

Another benefit of this approach is also that it would allow better usage of SimpleDB's *BatchPutAttributes* operation. This would offer the possibility to implement a persist operation that saves objects – even of different types – in a single transaction at once (e.g., persisting an object together with its `cascadedSave`-related property objects).

The drawback of this approach is that there is a danger of conflicting with the limitations of SimpleDB's domains: a maximum of 10 GB of data and 1 billion attributes per domain. Furthermore, it is also expected that a query is carried out slower when issued upon a domain that alone contains all the data, compared to the approach where the data is split upon several domains and each of these only holds a fraction of the data.

- *Save all objects in only one domain ('vresco') and only introduce items for the different object types.* This means that all objects (`s1`, `s2`, etc.) of the same type (`Service`) are saved within the type's item ('service'). The information of a particular object is stored as attribute-values within its type's item.

  In order to work, every attribute-key would need to carry a prefix to identify its value as the information that belongs to a certain property of a certain object instance (e.g., the name of a service `s1` is assigned to the attribute-value '1_name' within the item 'service').

  For applications that require storing lots of data, this approach would eventually not work. The reasons are the earlier mentioned limitations of SimpleDB, like the constraints that restrict the name lengths and the maximum number of attribute name-value pairs per item.

  Furthermore, this approach would require complex operations in order to map the stored values to their corresponding objects and would potentially cause a lot of overhead when saving and especially when retrieving data from the database.

Although it does not enable to save and retrieve data belonging to different types in a transactional manner, it was still decided to implement the first approach. The reason is to avoid the danger of conflicting with SimpleDB's limitations and to profit from the efficiency of distributing the stored data across several domains.

Each domain holds data from objects that belong to the same type. A domain is named according to the full class name (including the name of its package) of the corresponding object type (e.g., 'VReSCO.Contracts.Core.Service').

The items holding data of specific object instances are named after the object's unique *ID*. For instance, a service with the ID '1234' would be saved in its corresponding item '1234' within the domain 'service'.

5.1.3 Mapping of NHibernate Attributed Properties

Model objects, their properties and relations are annotated with attributes in order to help NHibernate map the objects with their representation in the MSSQL database. With using SimpleDB as database these attributes are reused in order to achieve the same relational mapping within the new DAOs and the underlying SimpleDB data model.

Within the newly introduced SimpleDB-DAO a property of an object that is being stored to or retrieved from the database, is only handled under the condition that it has been initialized and that the NHibernate attribute `[Property]` or one of the NHibernate attributes describing a relation (`[ManyToOne]`, `[OneToMany]` or `[ManyToMany]`) is attached to it. The property's information is stored as value to an attribute-key that is named after either the property or a related name that is retrieved from the property's NHibernate attribute and helps to unambiguously relate to it.

As mentioned in Section 2.3.3, SimpleDB only allows to store exclusively textual information as attribute-values. This means that for saving data types other than *strings*, a certain encoding (and decoding) has to be considered.

Because there is no need for comparing or sorting the attribute-values when querying, certain techniques presented in [68] for consigning the ordering of actual property values to the lexicographical ordering of their string representations (e.g., negative number offsets or zero-padding), were not used. The encoding is based on the property's type and is kept very simple. In most cases it affects primitive types and therefore only consists of the value's string representation.

In SimpleDB an object is stored within an item that is named after the string representation of the object's ID. When an object's property holds an object of another model type (e.g., `[ManyToOne]`), then the relation is commonly mapped by storing the string representation of the referenced object's ID as attribute-value within the referencing object's SimpleDB item. If the property holds a whole collection of such, then multiple IDs are assigned to the attribute-key accordingly.

The opposite is done when an object is retrieved from the database. In order to build it, the data for its properties is being read from SimpleDB. The assigned values are created based on the retrieved strings and according to their corresponding property's type.

The mapping used with this implementation is not only based on the mentioned NHibernate attributes, but also on several parameters that may be set for these. For example, the property of an object will be checked prior to saving, if it has been assigned a value, when its attribute demands it to be `NotNull`. `Cascade` (e.g., save or delete) parameters on certain property attributes also propagate the corresponding changes further to the referenced object.

Another very important attribute parameter that is considered when mapping the relational model to SimpleDB is the declaration of a property relation as being `Inverse`. A further comprehensive overview over the reused attributes that are considered when mapping the data model is given in Table 5 and Table 6.

Attribute	Description
`[Id]`	This attribute marks the property as being the ID of the object.
`[Property]`	The annotated property contains only a 'simple' type (integer, float, date, etc.). It does not relate to other types. The property carries a `Column` and eventually also a `NotNull` parameter.
`[OneToMany]`	Indicates that the property holds a collection of objects and relates to another type from the model.
`[ManyToOne]`	The property relates to another object by referencing a model type. Additionally it may define the parameters `Cascade` and `NotNull`.
`[ManyToMany]`	The attribute defines a many-to-many relation between two object types from the model. It contains a `Column` parameter.

Table 5: Essential Attributes for Property Mapping

The ID of an object is stored in a property that is marked with the `[Id]` attribute. The generation of IDs is done within the `AbstractGenericSimpleDbDAO` when saving an object into SimpleDB. The current implementation assigns a random numeric value to the corresponding property.

An alternative would be to implement counters in order to assign definitely unique and ascending values. There could either be only one counter for all types, or one separate counter per type and domain. The drawback of the alternative approaches may be additional overhead which would be caused when checking for the latest ID. Although it is very unlikely, it needs to be mentioned that with the current implementation of assigning random IDs, the possibility of leading to clashes (assigning the same random ID more than once) exists.

5.1.4 Mapping of Relations

If an object type inherits and extends another type and no specific DAO has been implemented to take care of its database operations, then the mapping of the subtype relation has to be handled within the generic `AbstractGenericSimpleDbDAO`.

When saving such an object into SimpleDB, it is stored into the domain of the

Attribute	**Description**
[Column]	The attribute belongs to the [ManyToOne] attribute and its parameter Name defines the attribute-key when mapping the property to SimpleDB. The NotNull parameter may also be set.
[Key]	The name of the mapped property is defined with the attribute's parameter Column. The attribute belongs to the attributes [OneToMany] and [ManyToMany].
[Bag]	The attribute belongs to the attributes [OneToMany] and [ManyToMany] and may define the parameters Cascade and Inverse. (The Inverse parameter only in conjunction with the [OneToMany] attributes.)
[Element]	This attribute is used to annotate that a property contains a collection of strings. It defines the parameter Column, which carries the name of the mapping.

Table 6: Further Attributes Considered for Property Mapping

upper type. Additionally its concrete type is stored as attribute-value under a key '_type'. (For example an object of the type PostCondition is saved into the domain 'condition' and has a string representation of its type (e.g., 'PostCondition') stored under the attribute key '_type'.) When getting the object data out of the database, the resulting object has to be explicitly cast into an object of the value of '_type'.

Before handling the mapping of relations, the meaning of an *inverse relation* has to be understood. As seen in the previously presented tables, properties that are annotated with [OneToMany] attributes and relate to other model types can be declared as Inverse by setting the parameter in the [Bag] attribute that goes along with the relation.

An *inverse relation* means that the property of an object, which contains another object of the model, does not store the reference within the database representation of the object. Rather the object referenced from the property stores the reference to the object in which it is contained itself. The reference is stored under an attribute-key with the name taken from the Column parameter of the [Key] attribute. The differences when mapping an inverse relation are shown in Table 7, where the mappings of a sample model depicted in Figure 19 are shown.

Almost all of the [OneToMany] attributed relations in the current data model are inverse relations and are additionally coupled with the Cascade parameter. This cascaded operation ensures that changes are propagated and that the relation is assuredly stored.

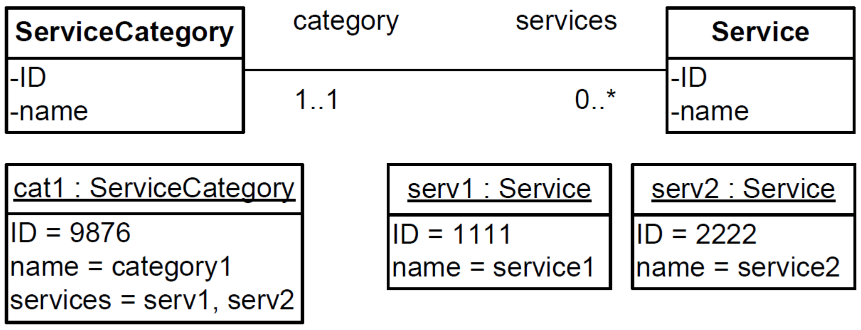


Figure 19: Relation Between `ServiceCategory` and `Service`

Domain	**Item**	**Attribute-key**	**Attribute-value**
Relation mapping of 'cat1' (from Figure 19):			
servicecategory	9876	name services	category1 1111, 2222
service	1111	name	service1
	2222	name	service2
Inverse relation mapping of 'cat1' (from Figure 19):			
servicecategory	9876	name	category1
service	1111	name category	service1 9876
	2222	name category	service2 9876

Table 7: Sample Mapping of Attributes

When getting the objects for an inversely related property out of the database, they usually need to be queried within a separate step (e.g., for the services in the `ServiceCategory` 'cat1' from Figure 19: `select * from service where category = 9876`) because their relation is not stored within the object itself.

With many-to-many relations the references are stored on both sides of the relations. This means that when retrieving a many-to-many related property of an object, it is not sufficient to only get the properties stored within the same object. Rather the comprehensive result is the aggregation of the references stored within the objects on both sides of the relation. To retrieve all properties a similar query is carried out as when getting the properties of an inverse relation.

After extending the DAO and handling the mapping of the data model, various other techniques were implemented in order to improve the performance of the application. These techniques will be presented in the following and include lazy loading when retrieving properties, and various caching mechanisms.

5.1.5 Lazy Loading

As described in the previous section, when retrieving an object from SimpleDB, several sequential requests are necessary in order to get all its property objects from the database and to build it completely. The problem is that these several requests have a negative effect on the performance and significantly slow down the application.

Furthermore, there may be also situations where these successive request can even cause loops within the DAO operations (e.g., retrieving a `ServiceCategory` object from the database – querying for its services – retrieving each of its `Service` objects from the database – querying for their category – and so on), if not prevented.

In [42] the author presents the *Lazy Loading* design pattern, which is used to prevent loading a huge number of related objects, when only one object is actually being retrieved. He writes that a "*Lazy Load* interrupts this loading process for the moment, leaving a marker in the object structure so that if the data is needed it can be loaded only when it is used" [42].

There are several ways to implement this pattern and the approach carried out in this solution is the use of so-called *ghost* objects [42]. A ghost is the real object in a partial state and without holding any data, except for its ID. When accessing one of its fields for the first time, it loads its full state and the full data into its fields.

In this solution ghosts are always created when the property object of another object is being retrieved from SimpleDB. The ghost that is assigned to the retrieved object's property is an instance of the property type with the appropriate ID and a 'lazy-loading' flag being set. The 'lazy-loading' flag indicates that an object is a ghost and that it has not been fully loaded yet. Again, the same is done when the property holds a collection.

```
public ServiceCategory Category
{
  get
  {
    if (category != null && category.LazyLoad)
      category = (new ServiceCategorySimpleDbDAO()).GetById(category.Id);

    return category;
  }
}
```

Listing 2: Getter for Property Holding a `ServiceCategory` Object

Because the objects are not fully loaded when retrieved from the database through one of the DAO operations, the objects retrieve the full data on their own and therefore need to know how and where to get it. This required the adaptation of all the ‘getter’ operations of the properties within the types of the VRESCo data model. Listing 2 shows the adapted get-method for a property that holds an object of the type `ServiceCategory`. The new getter for a collection of `Service` objects is shown in Listing 3.

```
public IList<Service> Services
{
  get
  {
    if (services != null)
    {
      IList<Service> tempServices = new List<Service>();

      foreach (var s in services)
      {
        if (s != null)
        {
          if (s.LazyLoad)
            tempServices.Add((new ServiceSimpleDbDAO()).GetById(s.Id));
          else
            tempServices.Add(s);
        }
      }

      services = tempServices;
    }
    else
    {
      services = new List<Service>();
    }

    return services;
  }
}
```

Listing 3: Getter for Property Holding a Collection of `Service` Objects

5.1.6 Caching with Memcached

In order to improve the performance when using SimpleDB as a database from within a VRESCo server application that is not hosted inside the Amazon cloud, the possibility to cache the data retrieved from the database was implemented. A *cache* is a mechanism used to store frequently used information in a readily accessible place to reduce the need to retrieve that information repeatedly.

In [68] the author suggests to use *Memcached* in conjunction with SimpleDB in order to achieve better results. Memcached [63] is a high-performance in-memory key-value store that is widely used and helps to reduce repeated database queries. It uses a client-server architecture, where the server maintains a key-value associative array, which is populated and queried by the client. The cache is intended to store small chunks of arbitrary data (e.g., strings, objects) and was designed to be very fast.

The interface of Memcached exposes basic functions for getting, setting and deleting data from the cache. In this solution the data is cached right after it is retrieved from the database (e.g., query with ‘select’ or getting an object). The basic functionality of a cache is shown in Listing 4.

```
function getById(id) {
  // try to get from cache:
  obj = memcached.get(id);

  if (!obj) {
    // nothing found -> get from database:
    obj = loadFromDatabase(id);

    // store into cache (for next get):
    memcached.put(id, obj);
  }

  return obj;
}
```

Listing 4: Basic Cache Functionality

After the data is cached, subsequent calls do not need to get it from the database anymore, but rather consult the cache, which is sure faster. To keep the cached data accurate and up-to-date, it needs to be deleted from the cache, if it changes for some reason (e.g., a client updates the data in the database). This ensures that the client is always forced to fetch the updated and most current data out of the database.

Every value or object stored into the cache is associated with a unique key. Additionally, Memcached also offers the possibility to define when the cached data is to expire automatically. For the sake of simplicity this has not been considered in the current implementation. The cached data is not meant to expire by itself, but is rather deleted by the application when a possible update invalidates it.

Preventing the cache from getting ‘dirty’ requires keeping track of the cached data and deleting it, if it does not represent the latest replication of the database data anymore. Because Memcached is kept very simple and only features a rather primitive model, the developer has to take care of this.

The current implementation keeps track of the cached data by storing lists with references to it. Table 8 gives an overview over the data that is cached within Memcached as part of this solution.

The lists that are being cached are necessary in order to delete all the cached objects

Cached data	**Key**	**Value**
objects	the object's ID	the object
queries	a query's unique key	the result of the query (as list containing either the resulting objects or only the keys to their caches)
lists to keep track of the cached objects that contain properties with an inverse relation	a concrete type	a list containing the keys of cached objects
lists to keep track of the queries that have been cached	a concrete type	a list containing the keys of cached queries
lists containing the types that are *inversely related* to other types	a concrete type	a list containing types that are *inversely related* to the type

Table 8: Data Cached Within Memcached

or queries of a certain type, if certain database updates result in invalidating their corresponding replications in the cache. If for example an object is updated because some of its properties were changed, then all cached query results concerning the object's type need to be deleted from the cache. This mechanism is kept for efficiency reasons rather simple and it is not examined if the concerned queries are really affected or not.

There are two possible ways to cache the results of a query. One is to cache a list containing the keys of the objects that have resulted from the query only, and to cache each of these objects separately. The other possibility is to cache the query result as a whole. The latter is done when the data retrieved through the query is expected to be not in a complete state. For example, when only specific properties are requested within a query (e.g., `select itemName() from services where ...`) and the resulting objects are only partially loaded. The retrieved data is then only processed within the scope of the query and there is no need to cache the partial query result objects separately.

Another challenge of this solution was the caching of objects that are inversely related to others. As described before, the information about such a relation is not stored within the object that contains other objects in its property, but rather in the latter objects itself. This means that if inverse relations are not treated separately and one of the referenced objects changes, the property where these are referenced and therefore the whole containing object eventually becomes invalid within the cache.

In this solution objects with inverse relations are cached too. Additionally, the key of the object is also listed under the object's type within a cached list, which is needed in order to keep track of these objects and to be able to delete them, if they have outdated because of updates that have affected their type.

To be able to check which types and consequently which objects are potentially affected by the change of an object, there are other lists that store the inverse type relations into the cache. If an object of an inversed type is updated, all the cached data that eventually stores this object in an inverse way is deleted from the cache. Not only the objects of the affected type, but also the query results issued and cached upon it, are deleted.

If for example objects of the type `ServiceRevision` contain properties that store collections of `Tag` objects and this relation is defined as *inverse*, then the reference for this relation is stored within the individual 'tag' items in SimpleDB. Furthermore, when these objects are cached, the relation from type `Tag` to `ServiceRevision` is also stored in the just mentioned cached list. If an object of the type `Tag` changes and is updated, then this list helps to track the relation to the `ServiceRevision` objects, which are deleted from the cache.

The reason for explicitly needing to cache the relation between two inversely related types is because although an object stores another one within its properties (e.g., a `Tag` object stores a `ServiceRevision`), it is not known to the DAO of the type (`Tag`) that the state of the object from the other type (`ServiceRevision`) depends on it in the first place.

Because the many-to-many related properties of an object can contain the aggregation of the references which are stored on both sides of the relation, these many-to-many relations are also handled as if they were inverse relations, and the objects are cached and treated accordingly as described.

To better illustrate the purpose of caching the mentioned lists, the method for removing the obsolete cache entries (query results and objects) of a certain type is presented in 'pseudo-code' in Listing 5. This function is called from within the abstract DAO (with the parameter `deleteAllObjsOfType` set to 'false') after persisting or deleting an object from the database.

As with SimpleDB, the use of the additional caching option can be enabled and disabled using a simple 'switch' (`key='Memcached' value='on'`) from within the application's configuration file.

```
function RemoveInvalidsFromCache(string type, bool deleteAllObjsOfType)
{
  if (deleteAllObjsOfType)
  {
    list = memcached.get("obj_"+type);
    if (list)
    {
      foreach(o in list)
        memcached.delete(o);

      memcached.delete("obj_"+type);
    }
  }

  list = memcached.get("qry_"+type);
  if (list)
  {
    foreach(q in list)
      memcached.delete(q);

    memcached.delete("qry_"+type);
  }

  list = memcached.get("rel_"+type);
  if (list)
  {
    memcached.delete("rel_"+type);

    foreach(t in list)
      RemoveInvalidsFromCache(t, true);
  }
}
```

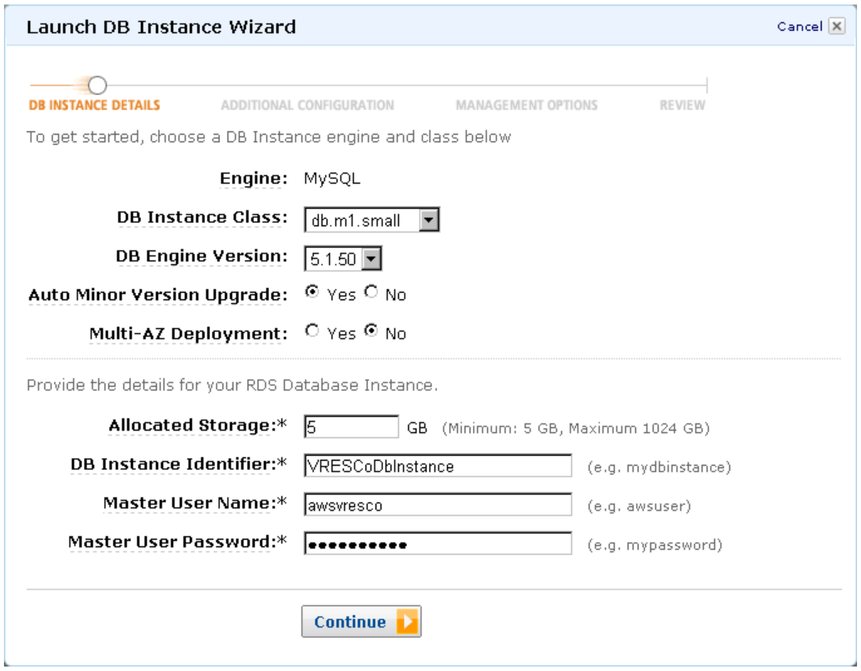
Listing 5: Method for Deleting Invalid Data from Cache

5.2 Using Amazon RDS as Database

Amazon launched the Relational Database Service during the work on SimpleDB as the database for VRESCo. It has been decided to use and evaluate both database approaches in this thesis.

To launch an Amazon RDS instance is fairly simple and can be done in a couple of minutes. Because a fully-featured MySQL database is offered by the service, there is almost no need to make any changes in the VRESCo implementation in order to adapt it to the new database.

There are several ways to launch a concrete RDS instance. For this purpose Amazon offers either several programming libraries or even a special command line tool, which offers simple commands for managing the database instances. Another possible way to launch, configure and manage an instance is to use the 'AWS Management Console' [11], which features an intuitive web-based user interface.



To get started, choose a DB Instance engine and class below

Engine:	MySQL
DB Instance Class:	db.m1.small
DB Engine Version:	5.1.50
Auto Minor Version Upgrade:	[✓ Yes] No
Multi-AZ Deployment:	Yes [✓ No]

Provide the details for your RDS Database Instance.

Allocated Storage:*	5 GB	(Minimum: 5 GB, Maximum 1024 GB)
DB Instance Identifier:*	VRESCoDbInstance	(e.g. mydbinstance)
Master User Name:*	awsvresco	(e.g. awsuser)
Master User Password:*	••••••••••	(e.g. mypassword)

Figure 20: Launching an RDS Instance

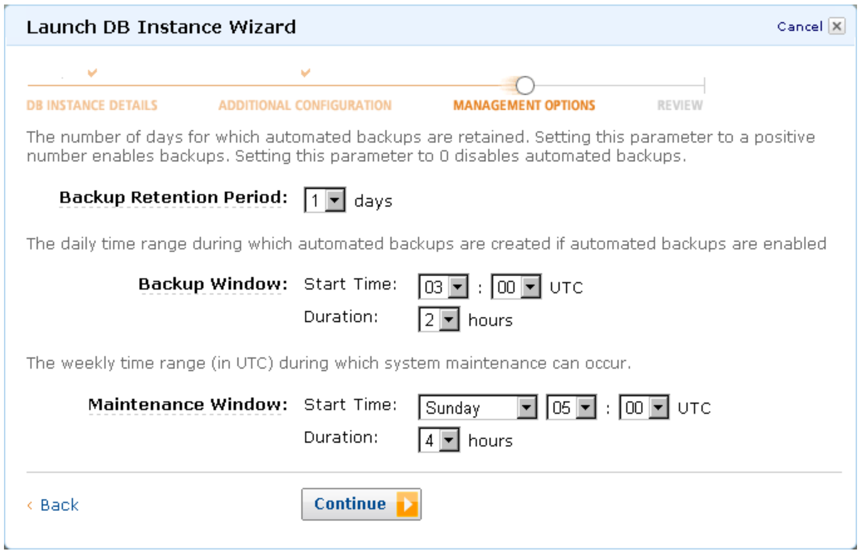
When launching an RDS instance via this management console, the user is asked to specify the details about the desired instance (e.g., instance type, allocated storage, port). Furthermore, the user needs to enter a database instance identifier, and the master username and password, which are then used to log on to the database instance (see Figure 20).

Before the instance is launched, there is also the possibility to provide additional configuration information and to specify certain management options (e.g., backup retention period, time windows for backup/maintenance), which is shown in Figure 21. It may take a few minutes before the database instance is created by Amazon and is ready to be used.

The management console also helps to authorize access to the RDS instance by assigning it to a definable security group. This allows to specify either a single IP address or a range of IP addresses that are allowed to connect to it. It is then possible to connect to the instance that is made available through an endpoint of the form `mydbinstance.myaccountid.us-east-1.rds.amazonaws.com`.

```
Data Source = mydbinstance.myaccountid.us-east-1.rds.amazonaws.com;
    Database = vresco; User Id = myid; Password = mypass
```

Listing 6: Example NHibernate Connection String


Launch DB Instance Wizard

Cancel

DB INSTANCE DETAILS | ADDITIONAL CONFIGURATION | MANAGEMENT OPTIONS | REVIEW

The number of days for which automated backups are retained. Setting this parameter to a positive number enables backups. Setting this parameter to 0 disables automated backups.

Backup Retention Period: 1 days

The daily time range during which automated backups are created if automated backups are enabled

Backup Window: Start Time: 03 : 00 UTC
Duration: 2 hours

The weekly time range (in UTC) during which system maintenance can occur.

Maintenance Window: Start Time: Sunday 05 : 00 UTC
Duration: 4 hours

‹ Back Continue

Figure 21: Defining Management Options for an RDS Instance

To use the launched instance in conjunction with VRESCo, only a few lines of the NHibernate configuration file need to be adjusted. Besides defining to use the `MySqlDataDriver` and `MySQL5Dialect`, the NHibernate connection string needs to be altered (see Listing 6).

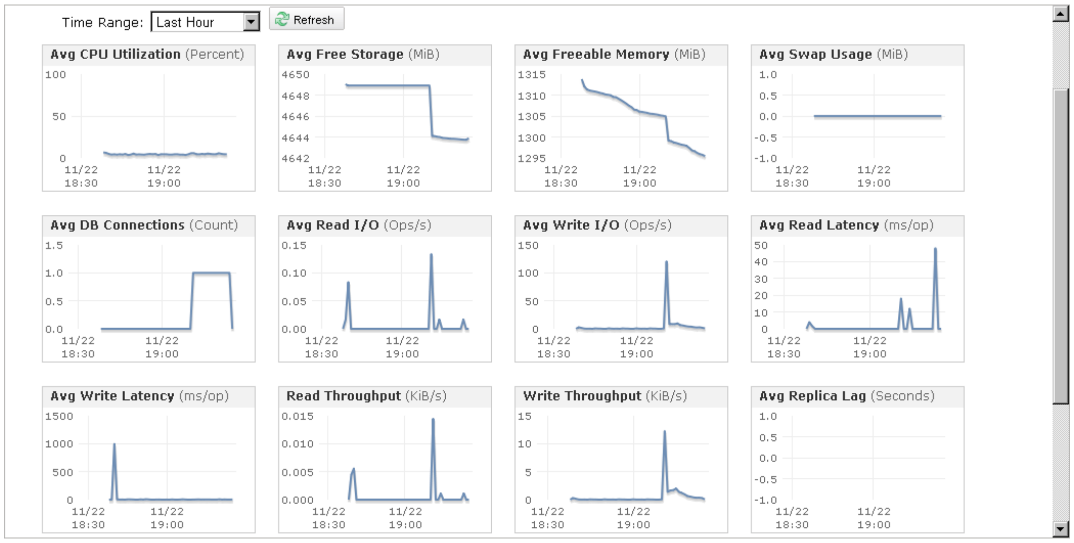


Figure 22: Monitoring an RDS Instance

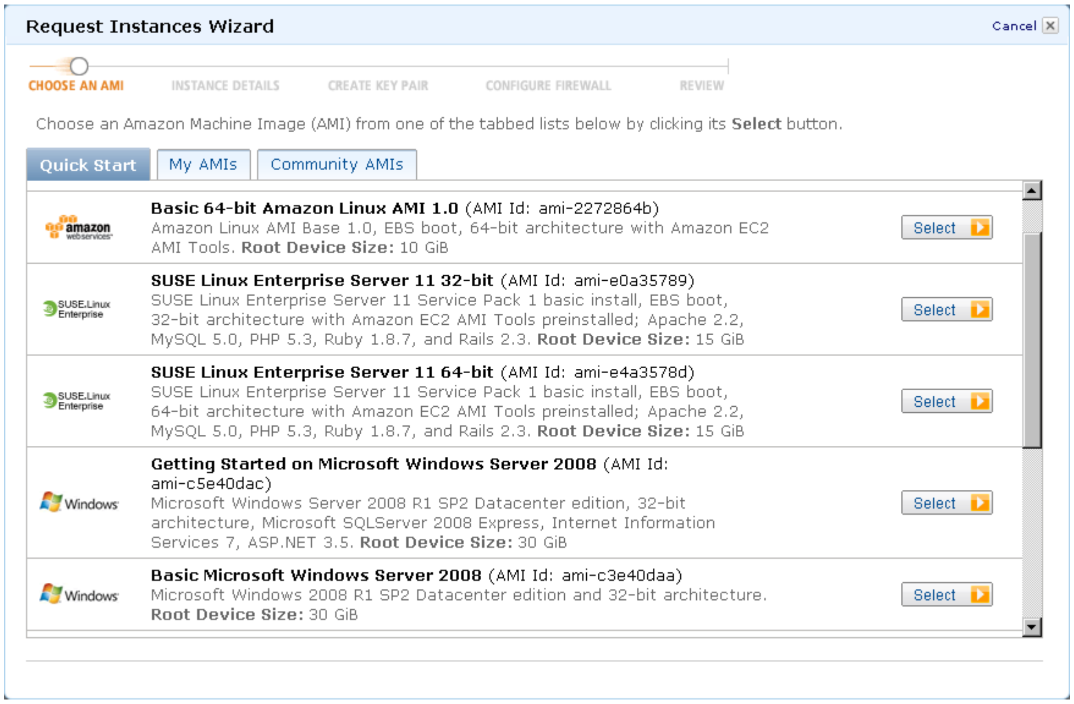
After that, VRESCo can be run as usual either from outside the Amazon cloud or from within an Amazon EC2 instance (see following section). While running an RDS database instance, it can be constantly monitored using the AWS Management Console (see Figure 22). Further features and options of the Relational Database

Service were not used and are not within the scope of this thesis.

5.3 Deploying VRESCo to Amazon EC2

The previously introduced cloud database services can be used from an application that runs outside the Amazon cloud. However, Amazon promises a better performance due to lower latencies when using the services from within the cloud. Furthermore, the application benefits from the various advantages that computing in the cloud has to offer, like instant scalability and not needing to maintain local servers.

The process for launching an Amazon EC2 instance is similar to the launching of an Amazon RDS instance. Amazon again offers several libraries, command line tools and the intuitive web-based management console. The console can be used to search for available Amazon Machine Images and to easily launch an EC2 instance (see Figure 23).



Choose an Amazon Machine Image (AMI) from one of the tabbed lists below by clicking its **Select** button.

Basic 64-bit Amazon Linux AMI 1.0 (AMI Id: ami-2272864b)
Amazon Linux AMI Base 1.0, EBS boot, 64-bit architecture with Amazon EC2 AMI Tools. **Root Device Size:** 10 GiB

SUSE Linux Enterprise Server 11 32-bit (AMI Id: ami-e0a35789)
SUSE Linux Enterprise Server 11 Service Pack 1 basic install, EBS boot, 32-bit architecture with Amazon EC2 AMI Tools preinstalled; Apache 2.2, MySQL 5.0, PHP 5.3, Ruby 1.8.7, and Rails 2.3. **Root Device Size:** 15 GiB

SUSE Linux Enterprise Server 11 64-bit (AMI Id: ami-e4a3578d)
SUSE Linux Enterprise Server 11 Service Pack 1 basic install, EBS boot, 64-bit architecture with Amazon EC2 AMI Tools preinstalled; Apache 2.2, MySQL 5.0, PHP 5.3, Ruby 1.8.7, and Rails 2.3. **Root Device Size:** 15 GiB

Getting Started on Microsoft Windows Server 2008 (AMI Id: ami-c5e40dac)
Microsoft Windows Server 2008 R1 SP2 Datacenter edition, 32-bit architecture, Microsoft SQLServer 2008 Express, Internet Information Services 7, ASP.NET 3.5. **Root Device Size:** 30 GiB

Basic Microsoft Windows Server 2008 (AMI Id: ami-c3e40daa)
Microsoft Windows 2008 R1 SP2 Datacenter edition and 32-bit architecture. **Root Device Size:** 30 GiB

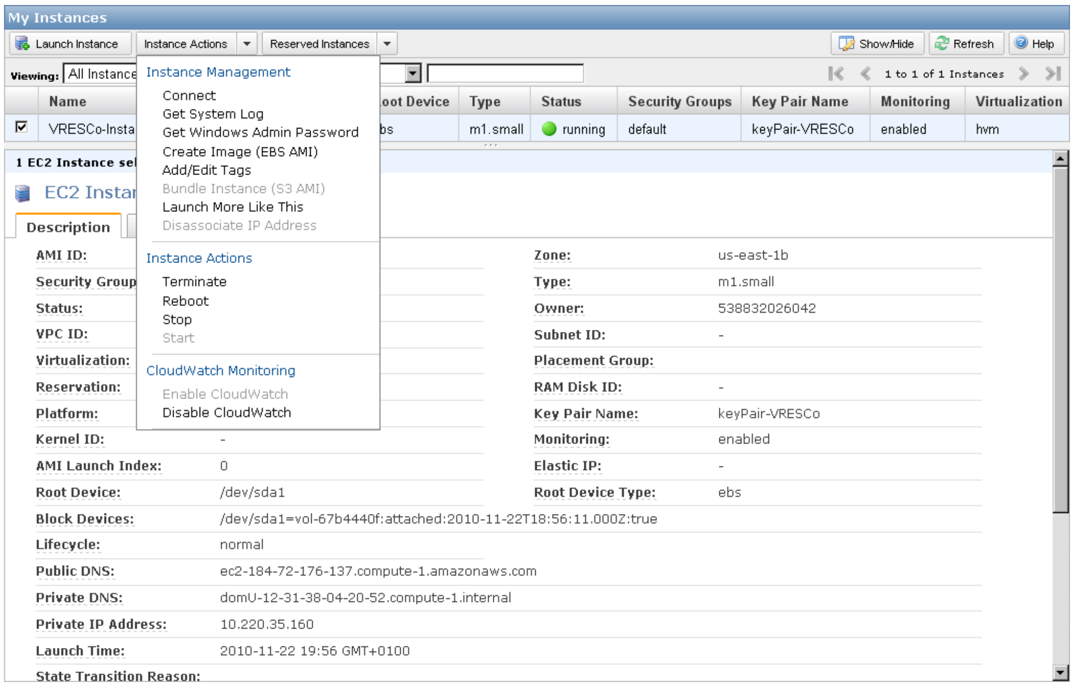
Figure 23: Choosing an AMI

When specifying the details (instance type, AMI, etc.) of the instance that is going to be launched, Amazon's management console asks to define and assign the instance to a security group. A security group defines certain rules for the firewall, which then decides over the delivery of the incoming traffic for the instance.

Before launching, the user also needs to create a *key pair*. A key pair is a security credential that is similar to a password and which is used to securely connect to the EC2 instance. Once the key pair is created, the private key from the key pair is needed to be saved locally by the user in order to use it to authenticate when connecting to the instance.

When the instance is launched, it again takes a few minutes before it is ready and can be used. To connect to it the user needs its corresponding endpoint address, which is given after the successful launch of the EC2 instance. There are several possible ways of how to connect and manage the server instance: the 'Windows Remote Desktop', 'secure shell' (ssh), and more.

After eventually configuring basic server settings and copying all the necessary application data to the instance, the VRESCo application server can be run. There is no need for any further adjustments in order to use the presented database solutions because VRESCo runs within the EC2 instance just as on a local physical machine. The EC2 instance can be monitored from within the AWS Management Console and the user can connect to it at any time, if he needs to undertake any further configuration or management tasks (see Figure 24).



Name	oot Device	Type	Status	Security Groups	Key Pair Name	Monitoring	Virtualization
✓ VRESCo-Insta	bs	m1.small	● running	default	keyPair-VRESCo	enabled	hvm

AMI ID:		Zone:	us-east-1b
Security Group		Type:	m1.small
Status:		Owner:	538832026042
VPC ID:		Subnet ID:	-
Virtualization:		Placement Group:	
Reservation:		RAM Disk ID:	-
Platform:		Key Pair Name:	keyPair-VRESCo
Kernel ID:	-	Monitoring:	enabled
AMI Launch Index:	0	Elastic IP:	-
Root Device:	/dev/sda1	Root Device Type:	ebs
Block Devices:	/dev/sda1=vol-67b4440f:attached:2010-11-22T18:56:11.000Z:true		
Lifecycle:	normal		
Public DNS:	ec2-184-72-176-137.compute-1.amazonaws.com		
Private DNS:	domU-12-31-38-04-20-52.compute-1.internal		
Private IP Address:	10.220.35.160		
Launch Time:	2010-11-22 19:56 GMT+0100		
State Transition Reason:			

Figure 24: Managing an EC2 Instance

As described in one of the previous sections, there are many further EC2 features available. The evaluation of features like Elastic Block Storage, automatic scaling, load balancing and many more is not within the scope of this thesis.

6 Evaluation

Before discussing a few limitations and shortcomings of the presented solution, an example use case will be introduced in order to further evaluate the effects of the undertaken architectural changes.

6.1 Quantitative Evaluation

In this subsection an example use case is going to be presented. The next step will be to run this scenario upon different variations of the VRESCo architecture that were presented and introduced during the work on this thesis. This will show the effects of the different services and aspects of computing in the cloud, and their impacts on the performance of the overall application.

Not only will the influences of the different databases (the initial relational database and the Amazon cloud database services: SimpleDB and RDS) be measured, but it will be also tested how the application behaves when it additionally runs from within the cloud. Additionally the potential benefits of the rather sophisticated caching that was implemented to support the application when it uses the SimpleDB service will be quantified.

The performance and efficiency of the current implementation will be evaluated according the time the use case requires to run through and the number of issued database instructions. Each database service will be tested with and without the combination of VRESCo running from within an Amazon EC2 instance.

6.1.1 Introducing Example Use Case

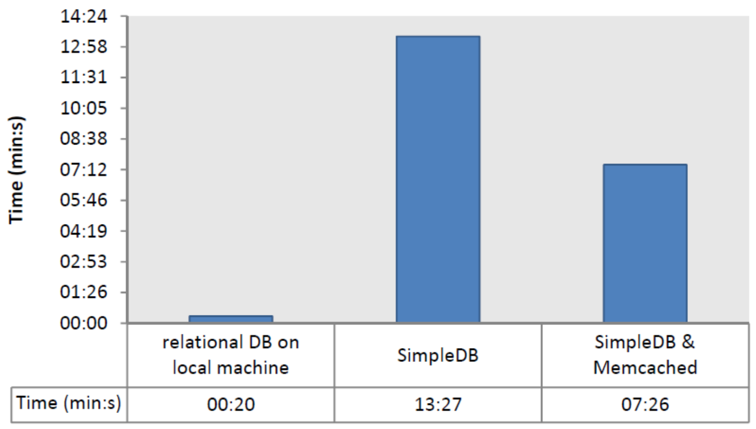
It has to be mentioned that the VRESCo implementation was not evaluated based on the performance of specific operations (e.g., create service revision, append revision tags). Rather various operations that involve a wide range of the underlying data model were covered in one comprehensive use case. This approach does not give any detailed information about certain 'bottlenecks' within the application, but the measurements are still representative of the performance characteristics of the different architectural approaches.

To evaluate the current implementation, a use case resembling the infrastructure of an example telecommunications company (TELCO) was used. Basically the services exposed by VRESCo are used to store users and services to the registry. Furthermore, some concrete service revisions are added to the earlier defined services, and several tags and QoS values are appended to these revisions. The example is reused from [64, 66] and its full listing can be found in Appendix B.

The following measurements are carried out five times and were then aggregated in order to get a representative average value. The tests undertaken within the Amazon cloud were carried out by using machines of the default instance type (small Amazon EC2 instance) in the default region ('US East'). The local tests were carried out on an equivalent notebook computer with a 1.86 GHz processor and 1.25 GB of RAM memory. The Internet connection of the local machine has a bandwidth of about 25 Mbit/s.

6.1.2 Use Case Running on SimpleDB

The first step was to substitute VRESCo's original database with the Amazon SimpleDB service. It is important to keep in mind that the SimpleDB service is used from the local machine, which is outside the Amazon cloud. The results after running the TELCO example are summarized in Figure 25.



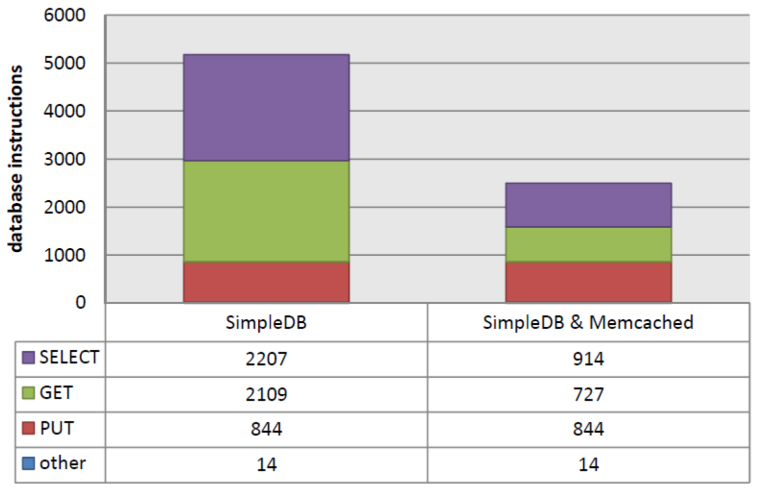
	relational DB on local machine	SimpleDB	SimpleDB & Memcached
Time (min:s)	00:20	13:27	07:26

Figure 25: Running the TELCO Use Case Using SimpleDB

Before introducing the Amazon cloud services, the TELCO example was run on the local computer with the initial VRESCo implementation using a relational database on the same machine, and needed 20 seconds to run through. It can be seen that after substituting the database with the cloud database service, the processing takes much longer. A significant loss in performance was expected because of the fact that part of the central architecture has been swapped to a remote service and all the crucial database instructions are carried out over the Internet.

For the sake of completeness it needs to be mentioned that the SimpleDB domains that were used to hold objects from the VRESCo data model, were created prior the execution of the tests. The instruction for creating a domain takes a couple of seconds and because all the applications using VRESCo are based upon the same pre-defined data model, these domains were initially created outside the test scope.

After introducing the caching option using Memcached in order to prevent that identical and avoidable database instructions are issued, the use case was run and measured again. The results depicted in Figure 25 show that Memcached achieved to significantly improve the applications performance compared to only relying on SimpleDB. Already processed data from SimpleDB is replicated at the local cache server and is retrieved without needing to issue a repeated request over the Internet and eventually loosing performance.



	SimpleDB	SimpleDB & Memcached
🟪 SELECT	2207	914
🟩 GET	2109	727
🟥 PUT	844	844
🟦 other	14	14

Figure 26: SimpleDB Instructions When Running the TELCO Use Case

Figure 26 shows the number of database instructions that were issued upon SimpleDB when the TELCO example was run. It can be seen that the use of Memcached enables to spare more than half of the retrieving database instructions that were originally necessary in this example.

It needs to be mentioned that this TELCO example does not reflect the caches full potential and that even greater improvements were experienced with different use cases during the work on this thesis. With use cases that heavily query the database, even almost three quarters of the database instructions were able to be omitted.

After running the example with VRESCo using SimpleDB in conjunction with Memcached, the caching statistics listed in Table 9 were observed. It can be seen that there is a large number of total set commands compared to the actual db instructions issued upon SimpleDB. This is the case because the statistic also counts the instructions that were issued in order to maintain the previously presented helper lists used

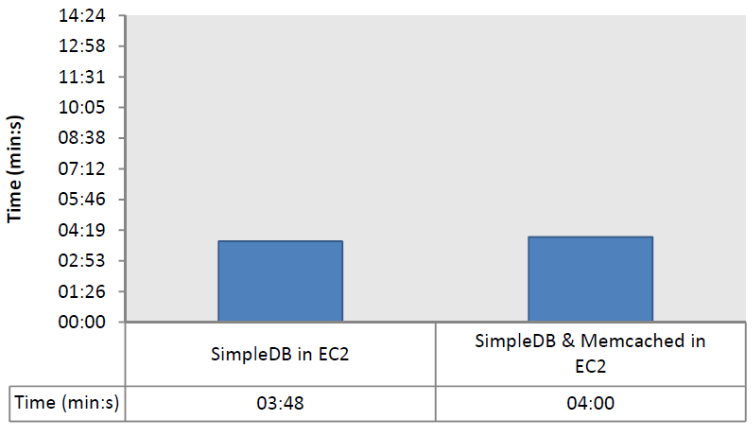
number of set commands	3967
number of get commands	8906
hits on get commands	6221
misses on get commands	2685
number of delete commands	2027
hits on delete commands	1651
misses on delete commands	376
current bytes	250056
total bytes read	4883217
total bytes written	6161777
current number of items	133
total number of items	3967

Table 9: Memcached Statistics After Running the TELCO Use Case

for keeping track of the cached data. The same cause also affects the number of delete commands.

The impact of these additionally cached lists presented in Section 5.1.6 also reflects itself in the large number of total items that were cached during the processing – opposed to the number of items cached after execution.

The ratio between the hits and misses of the issued 'get' commands means that the vast majority of the requested data was found in the cache and did not needed to be retrieved from SimpleDB.



	SimpleDB in EC2	SimpleDB & Memcached in EC2
Time (min:s)	03:48	04:00

Figure 27: Running the TELCO Use Case Within EC2 and Using SimpleDB

The next step was to run VRESCo and the TELCO example entirely from within the Amazon cloud. In addition to using SimpleDB as the database, the VRESCo server application was deployed into the cloud. A default machine instance was created in

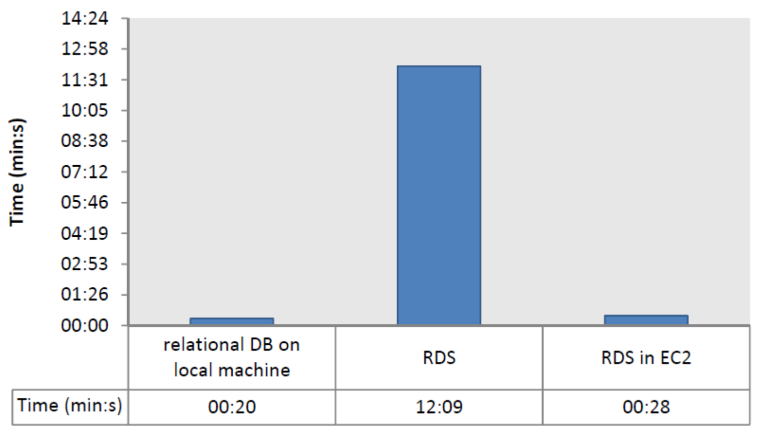
EC2 and the necessary application data was copied to it. The measured results from running the TELCO use case are shown in Figure 27.

Amazon advertises the close integration of its services and even promises near-LAN latency for the internal communication between them. Again, the results show that the performance has improved and that by avoiding the Internet as potential 'bottleneck' a significant enhancement was able.

When additionally running and using a Memcached server, the caching does not improve the applications performance significantly. In some cases, as with the TELCO example, it even slows down the execution. The reason is because of the mentioned overhead that comes with the caching option. On EC2 the connection to SimpleDB already has near-LAN latency and therefore this additional overhead is noticed.

6.1.3 Use Case Running on RDS

Next, the TELCO use case was tested against VRESCO running upon the Relational Database Service. It can be seen in Figure 28 that the result from working with the remote database while running the VRESCo application on a local machine is not satisfying. The reason for this is clearly that the use of the remote service requires all database operations to be carried out over the Internet. In order to still improve the performance of this architectural scenario, various NHibernate caching and pooling options could be tried.



	relational DB on local machine	RDS	RDS in EC2
Time (min:s)	00:20	12:09	00:28

Figure 28: Running the TELCO Use Case Using RDS

A further approach to receive a way better performance is to run VRESCo from within an EC2 instance and to keep the communication between the application and its database within the cloud. As can be seen in the figure, the time that is needed to run the TELCO example almost reaches into the range of when running VRESCo and the underlying database locally.

6.1.4 Discussion

The Relational Database Service used by an application that is running from within the cloud has provided the best measured results for the TELCO use case in terms of performance. The efforts necessary for creating the instances are minimal and there is no further need to make any changes within the original implementation in order to be able to swap the database and use VRESCo as usual.

Although RDS seems to be the first choice for applications that rely on an RDBMS, SimpleDB may also be a considerable alternative. As seen in the results of the evaluation, the service is not able to compete with the performances achieved when using the relational databases. Still, it was shown that the additional use of caching or the deployment of the application into an Amazon EC2 instance can significantly speed it up. Memcached helps to prevent the repeated issuing of already executed retrieve-instructions, and the individual database instructions upon SimpleDB are executed faster when issued by an application that runs in an EC2 instance from within the cloud.

The evaluation has also shown that using RDS from outside the cloud leads to very unsatisfying results in terms of performance. When running upon one of the relational databases, VRESCo uses NHibernate to interact with it. NHibernate causes additional overhead in order to manage the relational features (e.g., transactions) and it needs to be considered that all of these instructions are carried out over the Internet, causing additional traffic and slowing down the application's overall performance.

6.2 Limitations

A few mentionable shortcomings of the current implementation will be listed in the following. The intention is to note some of the current drawbacks of the solution and to help to decide if the advantages of using a highly available and scalable cloud service are still outweighing its shortcomings. Because the usage of Amazon RDS and EC2 did not involve any extensive adaptation of the initial VRESCo implementation, the following points mostly concern the usage of SimpleDB:

- Because SimpleDB is not a relational database, there is no support for transactions as with when using NHibernate on top of the original database. The problem is that it is not possible to guarantee that the persisting of an object that again contains other objects within its properties is going to be consistent. The reason is that there is no way to persist an object and further cascaded persists within a single transactional scope.

If the alternative approach with storing all SimpleDB items in only one SimpleDB domain would have been realized, then there would be the possibility to store several different objects with SimpleDB's *BatchPutAttributes* instruction at once.

- Since transactions are not supported, this also means that a *commit* terminating such a NHibernate transaction does not ensure that the updated data is also going to be persisted within SimpleDB. It is therefore necessary and essential to explicitly state in the code to persist the altered data objects.

- Although SimpleDB was designed for extensive parallelization when saving, getting and searching for data, the problem is that it is not fully possible to benefit from this advantage. VRESCo's current data model and the fact that objects are stored within separate domains according to their types, limits the scope of individual database operations. Consequently, several sequential requests are necessary when retrieving objects from the database and it is not possible to improve the efficiency by using parallel threads.

- Because a remote service is involved, unpredictable latencies that open a new set of potential error conditions need to be considered (e.g., network connectivity, request timeouts and remote errors due to overload or throttling). In this case the database, which is one of the applications core layers, is accessed via network. Fault tolerance, recovery and other techniques are necessary to ensure consistency within the application.

  Another point is that SimpleDB limits the maximum size of data that is retrievable from the database at once. This means that when reading a larger block of data, it needs to be retrieved using multiple consecutive requests. All this again requires additional work by the developer in order to make up for the limitations and potential error conditions of the service.

- One of the already mentioned limitations is that the maximum length of a SimpleDB attribute value is 1024 bytes. In order to handle longer values a possible solution would be to use other cloud storage services from Amazon in addition to SimpleDB, which were designed to store larger amounts of data. A pointer to the saved data block is then stored as attribute value. The result is that it would need two round trips when writing or reading the property data.

  Another alternative approach would be to split the value and store each segment into separate attributes. Additional splitting and joining logic would be needed.

- The current implementation does not support the option `deleteOrphan` for relations within the VRESCo data model. To 'delete the orphan' means that not anymore referenced elements of a one-to-many relation should be deleted from the database. The object contained within another one is not able to exist without the containing object or at least its relation.

  In the current implementation there is no support for tracking the internal relations and it is not able to tell if a certain object is really not referenced by any other object in the database. The result is that such an object is still stored in SimpleDB, but is not used or referenced again. The only side-effect to this is that it is still saved within the database and reserves space without being actually needed.

- Furthermore, also not all of the exceptions thrown by NHibernate are supported. It is currently not able to check if all of the references to an object have been removed, if it is being deleted. Therefore no `ObjectDeletionException` will be thrown as it would be the case with NHibernate when using a relational database.

- Only very restricted support for VQL (see Section 4.1.1) has been added at the moment.

- When using the DAOs with SimpleDB it has been experienced that sometimes a couple of reoccurring and eventually avoidable database instructions repeat itself without any reason and that the concerned data has not changed in-between. A possible explanation is that an object's property which is marked with `Cascade` propagates the database operation further to an unchanged property object. In the next step this leads the object to be updated – even if there have not been any changes to it.

  A possible but very complex solution to overcome this issue would be to track and detect the actual changes made upon an object, and to only persist these.

- Because the object IDs are assigned by random, they are neither starting with '0' nor are they ascending. It is therefore not possible to check for example for the first item of a certain type that was stored to the database.

- There is no fixed order among the multiple values of a SimpleDB attribute. A certain object is not per se at the same position within the collection of a property when it is retrieved from the database, as it has been when it was stored. In other words, it is not guaranteed that the order of the objects within the running application is the same as within the database.

7 Conclusion and Future Work

Besides the current hype around cloud computing, there are some considerable benefits to it and it is conceivable that cloud computing services open whole new perspectives and possibilities. The goal of this thesis was to explore the chances and advantages of deploying an existing application into the cloud. The Amazon Web Services have been used to achieve this goal because of their popularity and the general variety of their offering.

The focus was specifically on deploying VRESCo's database layer into the cloud and to substitute its original local database with an appropriate cloud database service from Amazon. One of the approaches was to use Amazon SimpleDB, which is a massively scalable key-value store. Although it is very flexible, the programmer has a lot of work to do in order to compensate for its simplicity. Furthermore, not all of the features from the original relational database are supported by the current implementation.

The evaluation presented in Chapter 6 showed that very satisfying results were achieved with Amazon's RDS service, which offers a fully-featured relational database in the cloud. To replace the local database with the remote database service, only a small adjustment in the application's configuration is necessary.

Summarizing it can be stated that SimpleDB is still a great alternative to the commonly used RDBMS. It is very powerful and many great advantages result from its simplicity and flexibility (e.g., schema-less). Still it has to be kept in mind that it is certainly not relational and that its usage may eventually involve more programming efforts. This thesis has also shown that SimpleDB's efficiency can be enhanced drastically by using Memcached in conjunction with it.

When considering to deploy an application into the cloud, the 'pros and cons' have to be individually considered (e.g., flexibility versus more work). For applications that are designed from scratch, the key-value database in the cloud (SimpleDB) is definitely a considerable and potentially beneficial option. Already existing applications (e.g., VRESCo) that are based on an RDBMS are eventually better off when sticking to the Relational Database Service and running them from within Amazon EC2.

7.1 Future Work

However, there are a couple of further possible optimizations and improvements that have not been explored during this thesis. It would be interesting to see how the application's performance is affected by the use of the further available EC2 options (e.g., auto scaling, load balancing). When working with the remote RDS service, it could be also examined if certain caching and pooling features of NHibernate have any positive effects on the performance. Another interesting aspect would also be to evaluate the use of cloud computing services according to their actual cost.

When using SimpleDB with VRESCo, it has to be kept in mind that a wise usage of the DAOs is able to lessen the number of database operations that are about to be performed, which certainly has an effect on the applications overall performance. When analyzing VRESCo's log files after running use cases, it can be seen that still many redundant and reoccurring database instructions are carried out. A possible task could be to improve VRESCo's service implementations in order to use the DAOs more efficiently.

Concerning the use of SimpleDB it would be also great to implement the possibility of issuing several instructions within a single transactional scope. By working on the data model, it would be eventually able to support both transactions and the comprehensive use of parallelization. Most of the currently issued database operations are exclusively sequential and depend on each other. To be able to concurrently issue multiple operations would make it possible to further benefit from SimpleDB's paradigm, which encourages to excessively make use of parallelization.

Another open point is the not yet realized full support for VQL queries when using the DAOs based on SimpleDB.

Appendix

A List of Abbreviations

AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
CAGR	Compound Annual Growth Rate
CPU	Central Processing Unit
DAL	Data Access Layer
DAO	Data Access Object
DB	Database
EBS	Amazon Elastic Block Store
EC2	Amazon Elastic Compute Cloud
ECU	EC2 Compute Unit
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ID	Identification
IDC	International Data Corporation
IP	Internet Protocol
IT	Information Technology
LAN	Local Area Network
NIST	National Institute of Standards and Technology
OASIS	Organization for the Advancement of Structured Information Standards
PaaS	Platform as a Service
QoS	Quality of Service
RAM	Random-Access Memory
RDBMS	Relational Database Management System
RDS	Amazon Relational Database Service
REST	Representational State Transfer
S3	Amazon Simple Storage Service
SaaS	Software as a Service
SimpleDB	Amazon SimpleDB
SLA	Service Level Agreement
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
SoC	Service-oriented Computing
SQL	Structured Query Language
TELCO	Telecommunications Company
UDDI	Universal Description, Discovery and Integration

Continued on Next Page...

URI	Uniform Resource Identifier
VM	Virtual Machine
VQL	VRESCo Querying Language
VRESCo	Vienna Runtime Environment for Service-oriented Computing
W3C	World Wide Web Consortium
WCF	Windows Communication Foundation
WSDL	Web Services Description Language
XML	Extensible Markup Language

Table 10: List of Abbreviations

B Complete TELCO Use Case

```
////////////////////////////////////////
// insert categories
////////////////////////////////////////

var partnerServiceCat = new ServiceCategory("PartnerService");
var paymentServiceCat = new ServiceCategory("PaymentService");
var phoneNrPortingCat = new ServiceCategory("PhoneNumberPorting");
var shippingServiceCat = new ServiceCategory("ShippingService");
var smsServiceCat = new ServiceCategory("SMSService");
var supplierCat = new ServiceCategory("SupplierService");

////////////////////////////////////////
// insert users
////////////////////////////////////////

// create a user group for telcos
var telcoGroup = new UserGroup();
telcoGroup.Groupname = "TelcoPartners";
telcoGroup.Description = "This group consists of collaborating TELCOs";
telcoGroup = userManager.CreateNewUserGroup(telcoGroup);

// create telco providers and its contact persons
var telco1 = new User();
telco1.Username = "mobilkom";
telco1.Password = "sdjaksdjaskd";
telco1.Company = "Mobilkom Austria";
telco1.Firstname = "Oliver";
telco1.Lastname = "Moser";
telco1.Usergroup = telcoGroup;
telco1 = userManager.CreateNewUser(telco1);

var telco2 = new User();
telco2.Username = "orange";
telco2.Password = "dajsdkajsdad";
telco2.Company = "Orange";
telco2.Firstname = "Rudolf";
telco2.Lastname = "Moser";
telco2.Usergroup = telcoGroup;
telco2 = userManager.CreateNewUser(telco2);

var telco3 = new User();
telco3.Username = "tmobile";
telco3.Password = "fskfasdafas";
telco3.Company = "T-Mobile";
telco3.Firstname = "Hans";
telco3.Lastname = "Moser";
telco3.Usergroup = telcoGroup;
telco3 = userManager.CreateNewUser(telco3);

var shippingUser = new User();
shippingUser.Username = "ups";
shippingUser.Password = "dajsdkajsdad";
shippingUser.Company = "UPS";
shippingUser.Firstname = "Hans";
shippingUser.Lastname = "Schnellsender";
shippingUser.Usergroup = telcoGroup;
shippingUser = userManager.CreateNewUser(shippingUser);
```

```

var supplierUser = new User();
supplierUser.Username = "nokia";
supplierUser.Password = "dajsdkajsdad";
supplierUser.Company = "NOKIA";
supplierUser.Firstname = "Hans";
supplierUser.Lastname = "Wurst";
supplierUser.Usergroup = telcoGroup;
supplierUser = userManager.CreateNewUser(supplierUser);

var paymentUser = new User();
paymentUser.Username = "mastercard";
paymentUser.Password = "dajsdkajsdad";
paymentUser.Company = "Mastercard";
paymentUser.Firstname = "Hans";
paymentUser.Lastname = "Viekohl";
paymentUser.Usergroup = telcoGroup;
paymentUser = userManager.CreateNewUser(paymentUser);

var smsUser = new User();
smsUser.Username = "smsat";
smsUser.Password = "dajsdkajsdad";
smsUser.Company = "SMS.AT";
smsUser.Firstname = "Hans";
smsUser.Lastname = "Viekohl";
smsUser.Usergroup = telcoGroup;
smsUser = userManager.CreateNewUser(smsUser);

////////////////////////////////////////
// insert test user groups
////////////////////////////////////////

var superUsers = new UserGroup();
superUsers.Description = "Advanced users but not admins";
superUsers.Groupname = "Superusers";
superUsers = userManager.CreateNewUserGroup(superUsers);

var guests = new UserGroup();
guests.Description = "Guest users";
guests.Groupname = "Guests";
guests = userManager.CreateNewUserGroup(guests);

var normalUsers = new UserGroup();
normalUsers.Description = "Normal users";
normalUsers.Groupname = "Users";
normalUsers = userManager.CreateNewUserGroup(normalUsers);

var createServices = new Claim(Claim.Resources.Service, Claim.ClaimTypes.
    Create);
var readServices = new Claim(Claim.Resources.Service, Claim.ClaimTypes.Read
    );
userManager.AddClaimToUserGroup(createServices, telcoGroup);
userManager.AddClaimToUserGroup(readServices, telcoGroup);

////////////////////////////////////////
// define QoS model
////////////////////////////////////////

var doubleConcept = new DataConcept();
```

```
doubleConcept.Name = "double";

var stringConcept = new DataConcept();
stringConcept.Name = "string";

var intConcept = new DataConcept();
intConcept.Name = "int";

var boolConcept = new DataConcept();
boolConcept.Name = "bool";

var processingTime = new Property();
processingTime.Description = "Time needed to carry out the operation for a
    specific request";
processingTime.Name = "ProcessingTime";
processingTime.DataConcept = doubleConcept;

var wrappingTime = new Property();
wrappingTime.Description = "Time needed to wrap/unwrap the XML structure of
    the request";
wrappingTime.Name = "WrappingTime";
wrappingTime.DataConcept = doubleConcept;

var executionTime = new Property();
executionTime.Description = "Time needed to execute the service (
    ProcessingTime + 2 * WrappingTime)";
executionTime.Name = Constants.QOS_EXECUTION_TIME;
executionTime.DataConcept = doubleConcept;

var latency = new Property();
latency.Description = "Time needed for the SOAP request to reach its
    destination";
latency.Name = Constants.QOS_LATENCY;
latency.DataConcept = doubleConcept;

var responseTime = new Property();
responseTime.Description = "Time needed for sending a message from the
    client until the response returns from to the service (ExecutionTime +
    2 * Latency)";
responseTime.Name = Constants.QOS_RESPONSE_TIME;
responseTime.DataConcept = doubleConcept;

var roundTripTime = new Property();
roundTripTime.Description = "Total Time consumed from issuing a request on
    the client to receiving the result from the service (ResponseTime + 2 *
    WrappingTime)";
roundTripTime.Name = "RoundTripTime";
roundTripTime.DataConcept = doubleConcept;

var throughput = new Property();
throughput.Description = "The number of requests that can be handled per
    second";
throughput.Name = Constants.QOS_THROUGHPUT;
throughput.DataConcept = doubleConcept;

////////////////////////////////////////
// insert some sample TELCO test services
////////////////////////////////////////

```

```
// publish service 1
var aps1 = new Service();
aps1.Name = "NumberPortingService";
aps1.Description = "A number porting service written with Axis 1";

var portingService1 = new ServiceRevision();
portingService1.Service = aps1;
portingService1.Wsdl = "http://vresco.vitalab.tuwien.ac.at:8081/axis/
    services/NumberPortingService?wsdl";
portingService1.Contract = "PortingService1";

var sr = publisher.CreateNewService(aps1, phoneNrPortingCat, telco1,
    portingService1);
phoneNrPortingCat = sr.Service.Category;

// publish service 3
var aps3 = new Service();
aps3.Name = "PortingService3";
aps3.Description = "Yet another number porting service written with .NET/
    WCF.";

var portingService3 = new ServiceRevision();
portingService3.Service = aps3;
portingService3.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30023/TELCO/
    PortingService3?wsdl";
portingService3.Contract = "PortingService3";

publisher.CreateNewService(aps3, phoneNrPortingCat, telco2, portingService3
    );

// publish service 5
var shippingService = new Service();
shippingService.Name = "ShippingService";
shippingService.Description = "UPS Shipping service implementation.";

var shippingServiceRev = new ServiceRevision();
shippingServiceRev.Service = shippingService;
shippingServiceRev.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30032/PARTNER
    /ShippingService?wsdl";
shippingServiceRev.Contract = "ShippingService";

publisher.CreateNewService(shippingService, shippingServiceCat,
    shippingUser, shippingServiceRev);

// publish service 6
var paymentService = new Service();
paymentService.Name = "PaymentService";
paymentService.Description = "A credit card payment service from Paylife.";

var paymentServiceRev = new ServiceRevision();
paymentServiceRev.Service = paymentService;
paymentServiceRev.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30033/PARTNER/
    PaymentService?wsdl";
paymentServiceRev.Contract = "PaymentService";

publisher.CreateNewService(paymentService, paymentServiceCat, paymentUser,
    paymentServiceRev);

// publish service 7
```

```
var supplierService = new Service();
supplierService.Name = "SupplierService";
supplierService.Description = "A Supplier service from another TELCO.";

var supplierServiceRev = new ServiceRevision();
supplierServiceRev.Service = supplierService;
supplierServiceRev.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30031/PARTNER
    /SupplierService?wsdl";
supplierServiceRev.Contract = "SupplierService";

publisher.CreateNewService(supplierService, supplierCat, supplierUser,
    supplierServiceRev);

// publish service 8
var smsService = new Service();
smsService.Name = "SMSService";
smsService.Description = "An SMS service from sms.at.";

var smsServiceRev1 = new ServiceRevision();
smsServiceRev1.Service = smsService;
smsServiceRev1.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30011/SMSAT/
    SMSService1?wsdl";
smsServiceRev1.Contract = "ISMSService";
smsServiceRev1 =  publisher.CreateNewService(smsService, smsServiceCat,
    smsUser, smsServiceRev1);
smsServiceCat = sr.Service.Category;

// define some tags for the SMS service
var t1 = new RevisionTag();
t1.Property = new Property("sms service");

var t2 = new RevisionTag();
t2.Property = new Property("messaging");

publisher.AppendRevisionTags(smsServiceRev1.Id, new RevisionTag[] {t1, t2})
    ;

// publish service 9
var smsService2 = new Service();
smsService2.Name = "SMSService";
smsService2.Description = "An SMS service from Mobilkom.";

var smsServiceRev2 = new ServiceRevision();
smsServiceRev2.Service = smsService;
smsServiceRev2.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30012/TELCO/
    SMSService2?wsdl";
smsServiceRev2.Contract = "ISMSService";

smsServiceRev2 = publisher.CreateNewService(smsService2, smsServiceCat,
    telco1, smsServiceRev2);

ServiceRevision smsServiceRev3 = new ServiceRevision();
smsServiceRev3.Service = smsService;
smsServiceRev3.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30013/TELCO/
    SMSService3?wsdl";
smsServiceRev3.Contract = "ISMSService";

smsServiceRev3 = publisher.AddRevision(smsServiceRev2, smsServiceRev3);

```

```
////////////////////////////
//// Publish case study version graph including tags and QoS attributes
////////////////////////////

////       v1
////      /  \
////     v3   v2
////     |     |
////     |    v4
////     |     |
////     |    v5
////     v6

var portingService = new Service();
portingService.Name = "NumberPortingService";
portingService.Description = "A number porting service for our case study";

// now we insert the revisions and chain them
// initial version
var v1 = new ServiceRevision();
v1.Wsdl = "http://vresco.vitalab.tuwien.ac.at:8081/axis/services/
    NumberPortingService?wsdl";
v1.Contract = "default";
v1.Binding = "default";
v1 = publisher.CreateNewService(portingService, phoneNrPortingCat, telco3,
    v1);

// v2
var v2 = new ServiceRevision();
v2.Wsdl = "http://vresco.vitalab.tuwien.ac.at:8081/axis/services/
    NumberPortingService2?wsdl";
v2.Contract = "default";
v2.Binding = "default";
v2 = publisher.AddRevision(v1, v2);

// v3
var v3 = new ServiceRevision();
v3.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30023/TELCO/PortingService3?
    wsdl";
v3.Contract = "default";
v3.Binding = "default";
v3 = publisher.AddRevision(v1, v3);

// v4
var v4 = new ServiceRevision();
v4.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30024/TELCO/PortingService4?
    wsdl";
v4.Contract = "default";
v4.Binding = "default";
v4 = publisher.AddRevision(v2, v4);

// v5
var v5 = new ServiceRevision();
v5.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30025/TELCO/PortingService5?
    wsdl";
v5.Contract = "default";
v5.Binding = "default";
v5 = publisher.AddRevision(v4, v5);

```

```
// v6
var v6 = new ServiceRevision();
v6.Wsdl = "http://vresco.vitalab.tuwien.ac.at:30026/TELCO/PortingService6?
    wsdl";
v6.Contract = "default";
v6.Binding = "default";
v6 = publisher.AddRevision(v3, v6);

// set service
service = v6.Service;

/////////////////////////////////////////
// insert custom tags
/////////////////////////////////////////

// tags for v1
var idTag = new RevisionTag();
idTag.Property = new Property("v1");
var stableTag = new RevisionTag();
stableTag.Property = new Property(RevisionTag.DefaultTags.STABLE.ToString()
    );
var jaxrpcTag = new RevisionTag();
jaxrpcTag.Property = new Property("jaxrpc");
publisher.AppendRevisionTags(v1.Id, new RevisionTag[] {idTag, stableTag,
    jaxrpcTag});

// tags for v2
idTag = new RevisionTag();
idTag.Property = new Property("v2");
var altTag = new RevisionTag();
altTag.Property = new Property("alt");
publisher.AppendRevisionTags(v2.Id, new RevisionTag[] {idTag, altTag,
    jaxrpcTag});

// tags for v3
idTag = new RevisionTag();
idTag.Property = new Property("v3");
var wcfTag = new RevisionTag();
wcfTag.Property = new Property("wcf");
publisher.AppendRevisionTags(v3.Id, new RevisionTag[] {idTag, wcfTag});

// tags for v4
idTag = new RevisionTag();
idTag.Property = new Property("v4");
publisher.AppendRevisionTags(v4.Id, new RevisionTag[] {idTag, altTag,
    wcfTag});

// tags for v5
idTag = new RevisionTag();
idTag.Property = new Property("v5");
publisher.AppendRevisionTags(v5.Id, new RevisionTag[] {idTag, altTag,
    wcfTag});

// tags for v6
idTag = new RevisionTag();
idTag.Property = new Property("v6");
publisher.AppendRevisionTags(v6.Id, new RevisionTag[] {idTag, wcfTag});

/////////////////////////////////////////
```

```
// insert random QoS values
/////////////////////////////////////////

var random = new Random(DateTime.Now.Millisecond);

// insert qos values for v1
var executionTimeQoS = new RevisionQoS();
executionTimeQoS.Property = executionTime;
executionTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v1.Id, executionTimeQoS);

var latencyQoS = new RevisionQoS();
latencyQoS.Property = latency;
latencyQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v1.Id, latencyQoS);

var responseTimeQoS = new RevisionQoS();
responseTimeQoS.Property = responseTime;
responseTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v1.Id, responseTimeQoS);

var throughputQoS = new RevisionQoS();
throughputQoS.Property = throughput;
throughputQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v1.Id, throughputQoS);

// insert qos values for v2
executionTimeQoS = new RevisionQoS();
executionTimeQoS.Property = executionTime;
executionTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v2.Id, executionTimeQoS);

latencyQoS = new RevisionQoS();
latencyQoS.Property = latency;
latencyQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v2.Id, latencyQoS);

responseTimeQoS = new RevisionQoS();
responseTimeQoS.Property = responseTime;
responseTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v2.Id, responseTimeQoS);

throughputQoS = new RevisionQoS();
throughputQoS.Property = throughput;
throughputQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v2.Id, throughputQoS);

// insert qos values for v3
executionTimeQoS = new RevisionQoS();
executionTimeQoS.Property = executionTime;
executionTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v3.Id, executionTimeQoS);

latencyQoS = new RevisionQoS();
latencyQoS.Property = latency;
latencyQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v3.Id, latencyQoS);

responseTimeQoS = new RevisionQoS();
```

```
responseTimeQoS.Property = responseTime;
responseTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v3.Id, responseTimeQoS);

throughputQoS = new RevisionQoS();
throughputQoS.Property = throughput;
throughputQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v3.Id, throughputQoS);

// insert qos values for v4
executionTimeQoS = new RevisionQoS();
executionTimeQoS.Property = executionTime;
executionTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v4.Id, executionTimeQoS);

latencyQoS = new RevisionQoS();
latencyQoS.Property = latency;
latencyQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v4.Id, latencyQoS);

responseTimeQoS = new RevisionQoS();
responseTimeQoS.Property = responseTime;
responseTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v4.Id, responseTimeQoS);

throughputQoS = new RevisionQoS();
throughputQoS.Property = throughput;
throughputQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v4.Id, throughputQoS);

// insert qos values for v5
executionTimeQoS = new RevisionQoS();
executionTimeQoS.Property = executionTime;
executionTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v5.Id, executionTimeQoS);

latencyQoS = new RevisionQoS();
latencyQoS.Property = latency;
latencyQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v5.Id, latencyQoS);

responseTimeQoS = new RevisionQoS();
responseTimeQoS.Property = responseTime;
responseTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v5.Id, responseTimeQoS);

throughputQoS = new RevisionQoS();
throughputQoS.Property = throughput;
throughputQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v5.Id, throughputQoS);

// insert qos values for v6
executionTimeQoS = new RevisionQoS();
executionTimeQoS.Property = executionTime;
executionTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v6.Id, executionTimeQoS);

latencyQoS = new RevisionQoS();
latencyQoS.Property = latency;
latencyQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v6.Id, latencyQoS);

responseTimeQoS = new RevisionQoS();
responseTimeQoS.Property = responseTime;
responseTimeQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v6.Id, responseTimeQoS);

throughputQoS = new RevisionQoS();
throughputQoS.Property = throughput;
throughputQoS.DoubleValue = random.Next(1000);
publisher.AppendRevisionQoS(v6.Id, throughputQoS);
```

Listing 7: Complete Listing of the TELCO Use Case (from [64,66])

References

[1] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns: Best Practices and Design Strategies.* Sun Microsystems, Inc., Mountain View, CA, USA, 2003.

[2] Amazon Web Services Developer Community. Structured Data Storage. `http://developer.amazonwebservices.com/connect/entry.jspa?externalID=3087`. Visited on 01-November-2010.

[3] Amazon.com, Inc. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`. Visited on 01-November-2010.

[4] Amazon.com, Inc. Amazon Elastic Compute Cloud: Developer Guide. `http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-dg.pdf`. Visited on 01-November-2010.

[5] Amazon.com, Inc. Amazon Relational Database Service (Amazon RDS). `http://aws.amazon.com/rds`. Visited on 01-November-2010.

[6] Amazon.com, Inc. Amazon Relational Database Service: Developer Guide. `http://awsdocs.s3.amazonaws.com/RDS/latest/rds-dg.pdf`. Visited on 01-November-2010.

[7] Amazon.com, Inc. Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3`. Visited on 01-November-2010.

[8] Amazon.com, Inc. Amazon SimpleDB. `http://aws.amazon.com/simpledb`. Visited on 01-November-2010.

[9] Amazon.com, Inc. Amazon SimpleDB: Developer Guide. `http://awsdocs.s3.amazonaws.com/SDB/latest/sdb-dg.pdf`. Visited on 01-November-2010.

[10] Amazon.com, Inc. Amazon Web Services. `http://aws.amazon.com`. Visited on 01-November-2010.

[11] Amazon.com, Inc. AWS Management Console. `http://aws.amazon.com/console`. Visited on 01-November-2010.

[12] Amazon.com, Inc. AWS SDK for .NET. `http://aws.amazon.com/sdkfornet`. Visited on 01-November-2010.

[13] Amazon.com, Inc. Case Studies. `http://aws.amazon.com/solutions/case-studies`. Visited on 01-November-2010.

[14] Amazon.com, Inc. Amazon Web Services: Overview of Security Processes. `http://media.amazonwebservices.com/pdf/AWS_Security_Whitepaper.pdf`, 2010. Visited on 01-November-2010.

[15] Amazon.com, Inc. Overview of Amazon Web Services. `http://media.amazonwebservices.com/AWS_Overview.pdf`, 2010. Visited on 01-November-2010.

[16] Animoto Inc. Amazon.com CEO Jeff Bezos on Animoto. `http://animoto.com/blog/company/amazon-com-ceo-jeff-bezos-on-animoto`. Visited on 01-November-2010.

[17] Animoto Inc. Animoto – The End of Slideshows. `http://animoto.com`. Visited on 01-November-2010.

[18] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf`, February 2009. Visited on 01-November-2010.

[19] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[20] C. Baun, M. Kunze, J. Nimis, and S. Tai. *Cloud Computing – Web-basierte dynamische IT-Services*. Springer-Verlag Berlin Heidelberg, 2010.

[21] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the Weather tomorrow?: Towards a Benchmark for the Cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, pages 9:1–9:6, New York, NY, USA, 2009. ACM.

[22] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a Database on S3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 251–264, New York, NY, USA, 2008. ACM.

[23] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25(6):599–616, 2009.

[24] Nicholas Carr. *The Big Switch: Rewiring the World, from Edison to Google*. W.W. Norton & Company, 2008.

[25] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 85–90, New York, NY, USA, 2009. ACM.

[26] Brian F. Cooper, Eric Baldeschwieler, Rodrigo Fonseca, James J. Kistler, P. P. S. Narayan, Chuck Neerdaels, Toby Negrin, Raghu Ramakrishnan, Adam Silberstein, Utkarsh Srivastava, and Raymie Stata. Building a Cloud for Yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43, 2009.

[27] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[28] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud '09, Berkeley, CA, USA, 2009. USENIX Association.

[29] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The Cost of Doing Science on the Cloud: The Montage Example. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

[30] Dave Durkee. Why Cloud Computing Will Never Be Free. *Queue*, 8(4):20–29, 2010.

[31] Schahram Dustdar and Martin Treiber. A View Based Analysis on Web Service Registries. *Distrib. Parallel Databases*, 18(2):147–171, 2005.

[32] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling. Patterns: Service-Oriented Architecture and Web Services. `http://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf`, 2004. Visited on 01-November-2010.

[33] Justin R. Erenkrantz, Michael Gorlick, Girish Suryanarayana, and Richard N. Taylor. From Representations to Computations: The Evolution of Web Architectures. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 255–264, New York, NY, USA, 2007. ACM.

[34] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall Professional Technical Reference, 2005.

[35] Thomas Erl. *SOA – Entwurfsprinzipien fuer serviceorientierte Architektur*. Addison-Wesley Verlag, 2008.

[36] Roy T. Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[37] Organization for the Advancement of Structured Information Standards (OASIS). UDDI Version 3.0.2 Specification. `http://uddi.org/pubs/uddi_v3.htm`, 2004. Visited on 01-November-2010.

[38] Organization for the Advancement of Structured Information Standards (OASIS). ebXML Registry Services and Protocols Version 3.0. `http://docs.oasis-open.org/regrep/v3.0/specs/regrep-rs-3.0-os.pdf`, 2005. Visited on 01-November-2010.

[39] Organization for the Advancement of Structured Information Standards (OASIS). Reference Model for Service Oriented Architecture 1.0. `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`, 2006. Visited on 01-November-2010.

[40] Ian Foster. What is the Grid? A Three Point Checklist. `http://dlib.cs.odu.edu/WhatIsTheGrid.pdf`, 2002. Visited on 01-November-2010.

[41] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.

[42] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[43] Google, Inc. Google Docs – Online documents, spreadsheets, presentations, surveys, file storage and more. `http://docs.goole.com`. Visited on 01-November-2010.

[44] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web services architecture. *IBM Syst. J.*, 41(2):170–177, 2002.

[45] World Wide Web Consortium (W3C) Web Services Architecture Working Group. Web Services Architecture. `http://www.w3.org/TR/ws-arch`, 2004. Visited on 01-November-2010.

[46] Scott Hazelhurst. Scientific computing using virtual high-performance computing: a case study using the Amazon Elastic Computing Cloud. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, SAICSIT '08, pages 94–103, New York, NY, USA, 2008. ACM.

[47] Hearst Seattle Media, LLC. Hello Animoto, an Amazon Web Services darling. `http://blog.seattlepi.com/amazon/archives/142569.asp`. Visited on 01-November-2010.

[48] W. Hummer, P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. VRESCo – Vienna Runtime Environment for Service-oriented Computing. In Schahram Dustar and Fei Li, editors, *Service Engineering: European Research Results*. Springer, 2010.

[49] International Data Corporation (IDC). IDC on "the Cloud". `http://www.idc.com/research/cloudcomputing/index.jsp`, 2008. Visited on 01-November-2010.

[50] Information Systems Institute at the Technical University of Vienna. VRESCo – Vienna Runtime Enviroment for Service-oriented Computing. `http://www.infosys.tuwien.ac.at/prototyp/VRESCo`. Visited on 01-November-2010.

[51] Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. On Technical Security Issues in Cloud Computing. In *CLOUD '09: Proceedings of the 2009 IEEE International Conference on Cloud Computing*, pages 109–116, Washington, DC, USA, 2009. IEEE Computer Society.

[52] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Phil Maechling. Scientific Workflow Applications on Amazon EC2. In *Proceedings of the Workshop on Cloud-based Services and Applications in conjunction with 5th IEEE International Conference on e-Science*, e-Science 2009, 2009.

[53] Lori M. Kaufman. Data Security in the World of Cloud Computing. *IEEE Security and Privacy*, 7(4):61–64, 2009.

[54] Ali Khajeh-Hosseini, Ian Sommerville, and Ilango Sriram. Research Challenges for Enterprise Cloud Computing. *CoRR*, abs/1001.3257, 2010.

[55] Donald Kossmann, Tim Kraska, and Simon Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 579–590, New York, NY, USA, 2010. ACM.

[56] Thomas Laner. VQL – A View-based Querying Approach for the VRESCo Runtime. Master's thesis, Vienna University of Technology, 2009.

[57] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. End-to-End Versioning Support for Web Services. In *SCC '08: Proceedings of the 2008 IEEE International Conference on Services Computing*, pages 59–66, Washington, DC, USA, 2008. IEEE Computer Society.

[58] Philipp Leitner, Florian Rosenberg, and Schahram Dustdar. Daios: Efficient Dynamic Web Service Invocation. *IEEE Internet Computing*, 13(3):72–80, 2009.

[59] Huan Liu and Sewook Wee. Web Server Farm in the Cloud: Performance Evaluation and Dynamic Architecture. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 369–380, Berlin, Heidelberg, 2009. Springer-Verlag.

[60] Alexandros Marinos and Gerard Briscoe. Community Cloud Computing. In *CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing*, pages 472–484, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance*. O'Reilly Media, Inc., 2009.

[62] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, Version 15. `http://csrc.nist.gov/groups/SNS/cloud-computing`, 2009. Visited on 01-November-2010.

[63] memcached – a distributed memory object caching system. `http://memcached.org`. Visited on 01-November-2010.

[64] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Advanced Event Processing and Notifications in Service Runtime Environments. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 115–125, New York, NY, USA, 2008. ACM.

[65] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. *IEEE Transactions on Services Computing*, 99(PrePrints), 2010.

[66] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken SOA triangle: a software engineering perspective. In *IW-SOSWE '07: 2nd international workshop on Service oriented software engineering*, pages 22–28, New York, NY, USA, 2007. ACM.

[67] Rao Mikkilineni and Vijay Sarathy. Cloud Computing and the Lessons from the Past. In *Proceedings of the 2009 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*, WETICE '09, pages 57–62, Washington, DC, USA, 2009. IEEE Computer Society.

[68] James Murty. *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly Media, Inc., March 2008.

[69] NHibernate Project. NHibernate Forge. `http://nhforge.org`. Visited on 01-November-2010.

[70] Oracle Corporation. MySQL – The worlds's most popular open source database. `http://www.mysql.com`. Visited on 01-November-2010.

[71] Simon Ostermann, Alexandru Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In *Proceedings of the Cloudcomp 2009*, CloudComp '09, 2009.

[72] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for Science Grids: a Viable Solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, DADC '08, pages 55–64, New York, NY, USA, 2008. ACM.

[73] Michael P. Papazoglou. *Web Services: Principles and Technology.* Pearson Education Limited, 2007.

[74] Mike P. Papazoglou. Service -Oriented Computing: Concepts, Characteristics and Directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, Washington, DC, USA, 2003. IEEE Computer Society.

[75] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

[76] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud Computing: An Overview. In Martin Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 626–631. Springer Berlin / Heidelberg, 2009.

[77] George Reese. *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud.* O'Reilly Media, Inc., 2009.

[78] Leonard Richardson and Sam Ruby. *RESTful Web Services.* O'Reilly Media, Inc., 2007.

[79] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *NCM '09: Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, pages 44–51, Washington, DC, USA, 2009. IEEE Computer Society.

[80] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, New York, NY, USA, 2009. ACM.

[81] Florian Rosenberg, Predrag Celikovic, Anton Michlmayr, Philipp Leitner, and Schahram Dustdar. An End-to-End Approach for QoS-Aware Service Composition. In *EDOC'09: Proceedings of the 13th IEEE international conference on Enterprise Distributed Object Computing*, pages 128–137, Piscataway, NJ, USA, 2009. IEEE Press.

[82] Florian Rosenberg, Philipp Leitner, Anton Michlmayr, Predrag Celikovic, and Schahram Dustdar. Towards Composition as a Service – A Quality of Service Driven Approach. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1733–1740, Washington, DC, USA, 2009. IEEE Computer Society.

[83] Florian Rosenberg, Philipp Leitner, Anton Michlmayr, and Schahram Dustdar. Integrated Metadata Support for Web Service Runtimes. In *EDOCW '08: Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops*, pages 361–368, Washington, DC, USA, 2008. IEEE Computer Society.

[84] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 205–212, Washington, DC, USA, 2006. IEEE Computer Society.

[85] Salesforce.com, Inc. Salesforce.com – CRM Software & Online CRM System. `http://www.salesforce.com`. Visited on 01-November-2010.

[86] Sun Developer Network (SDN). Core J2EE Patterns – Data Access Object. `http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`. Visited on 01-November-2010.

[87] The New York Times Company. Self-Service, Prorated Supercomputing Fun! `http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun`. Visited on 01-November-2010.

[88] The New York Times Company. Times Machine – New York Times. `http://timesmachine.nytimes.com`. Visited on 01-November-2010.

[89] Aphrodite Tsalgatidou and Thomi Pilioura. An Overview of Standards and Related Technology in Web Services. *Distrib. Parallel Databases*, 12(2-3):135–162, 2002.

[90] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[91] Jinesh Varia. Cloud Architectures. `http://media.amazonwebservices.com/AWS_Cloud_Architectures.pdf`, 2008. Visited on 01-November-2010.

[92] Toby Velte, Anthony Velte, and Robert Elsenpeter. *Cloud Computing: A Practical Approach*. McGraw-Hill, Inc., New York, NY, USA, 2010.

[93] World Wide Web Consortium (W3C). Extensible Markup Language (XML). `http://www.w3.org/XML`. Visited on 01-November-2010.

[94] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 2001. Visited on 01-November-2010.

[95] World Wide Web Consortium (W3C). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). `http://www.w3.org/TR/soap12-part1`, 2007. Visited on 01-November-2010.

[96] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. `http://www.w3.org/TR/wsdl20`, 2007. Visited on 01-November-2010.

[97] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing Security of Virtual Machine Images in a Cloud Environment. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 91–96, New York, NY, USA, 2009. ACM.

[98] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. SETI@HOME—massively distributed computing for SETI. *Computing in Science and Engg.*, 3(1):78–83, 2001.

[99] Shuai Zhang, Shufen Zhang, Xuebin Chen, and Xiuzhen Huo. Cloud Computing Research and Development Trend. In *ICFN '10: Proceedings of the 2010 Second International Conference on Future Networks*, pages 93–97, Washington, DC, USA, 2010. IEEE Computer Society.

[100] Shufen Zhang, Shuai Zhang, Xuebin Chen, and Shangzhuo Wu. Analysis and Research of Cloud Computing System Instance. In *ICFN '10: Proceedings of the 2010 Second International Conference on Future Networks*, pages 88–92, Washington, DC, USA, 2010. IEEE Computer Society.

[101] Jinzy Zhu, Xing Fang, Zhe Guo, Meng Hua Niu, Fan Cao, Shuang Yue, and Qin Yu Liu. IBM Cloud Computing Powering a Smarter Planet. In *CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing*, pages 621–625, Berlin, Heidelberg, 2009. Springer-Verlag.