ijecs open access www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 2 Issue 12, Dec.2013 Page No. 3374-3383

Moving Towards IT platforms to the cloud Computing & deep understanding with IT platform architecture transformation

Er. Shalu soni, Er. Sahil Verma, Er. Vishal Gupta

M-Tech Student

M-Tech (C.S.E.)

M-Tech (C.S.E.)

Introduction

Companies can greatly reduce IT costs by offloading data and computation to cloud computing services. Still, many companies are reluctant to do so, mostly due to outstanding security concerns. A recent study [2] surveyed more than 500 chief executives and IT managers in 17 countries, and found that despite the potential benefits, executives "trust existing internal systems over cloud-based systems due to fear about security threats and loss of control of data and systems". One of the most serious concerns is the possibility of confidentiality violations. Either maliciously or accidentally, cloud provider's employees can tamper with or leak a company's data. Such actions can severely damage the reputation or finances of a company. In order to prevent confidentiality violations, cloud services' customers might resort to encryption. While encryption is effective in securing data before it is stored at the provider, it cannot be applied in services where data is to be computed, since the unencrypted data must reside in the memory of the host running the computation. In Infrastructure as a Service (IaaS) cloud services such as Amazon's EC2, the provider hosts virtual machines (VMs) on behalf of its customers, who can do arbitrary computations. In these systems, anyone with privileged access to the host can read or manipulate a customer's data. Consequently, customers cannot protect their VMs on their own. Cloud service providers are making a substantial effort to secure their systems, in order to minimize the threat of insider attacks, and reinforce the confidence of customers. For example, they protect and restrict access to the hardware facilities, adopt stringent accountability and auditing procedures, and minimize the number of staff who have access to critical components of the infrastructure [8]. Nevertheless, insiders that administer the software systems at the provider backend ultimately still possess the technical means to access customers' VMs. Thus, there is a clear need for a technical solution that guarantees the confidentiality and integrity of computation, in a way that is verifiable by the customers of the service.

Outside-in and inside-out architecture styles

Architecture styles define families of software systems in terms of patterns for characterizing how architecture components interact. They define what types of architecture components can exist in architectures of those styles, and constraints on how they may be combined. They define how components may be combined together for deployment. They define how units of work are managed, e.g., whether they are transactional (n-phase commit). And they define how functionality that components provision may be composited into higher order functionality and how such can be exposed for use by human beings or other systems. The *Outside-In* architectural style is inherently top-down and emphasizes decomposition to the functional level but not lower, is service-oriented rather than application-oriented; factors out policy as a first-class architecture component that can be used to govern transparent performance of service-related tasks; and emphasizes the ability to adapt performance

to user/business needs without having to consider the intricacies of architecture workings1. The counter style, what we call *Inside-Out*, is inherently bottom-up and takes much more of an infrastructural point of view as a starting point, building up to a business functional layer. Application platforms constructed using *client server*, *object-oriented*, and *2/3/n-tier* architecture 1 An styles are those to which we apply the generalization *Inside-Out* because they form the basis of enterprise application architectures today, and because architectures of these types have limitations that require transformation to scale in a massive way vis-à-vis Outside-In platforms. Implementation of an Outside-In architecture results in better architecture
layering and factoring, and interfaces that become more *business* than *data* oriented. Policy becomes more explicit, and is exposed in a way that makes it easier to change it as necessary. Service orientation guides the implementation, making it more feasible to integrate and interoperate using commodity infrastructure rather than using complex and inflexible application integration middleware. As a rule, it is simpler to integrate businesses at functional levels than at lower technology layers where implementations might vary widely. Hence we emphasize decomposition to the functional level, which often is dictated by standards within a market, regulatory constraints on that market, or even accounting (AP/AR/GL) practices. For a much more detailed discussion of Outside-In versus Inside-Out
architecture styles, please see the working paper we call "Web Services
2.0"vii.

Clouds and service grids

Since a widely accepted industry definition of cloud computing — beyond a relationship to the Internet and Internet technologies — does not exist at present, we see the term used to mean hosting of hardware in an external data center (sometimes called infrastructure as a service), utility computing (which packages computing resources so they can be used as a utility in an always on, metered, and elastically scalable way), platform services (sometimes called middleware as a service), and application hosting (sometimes called software or applications as a service). The potential of cloud computing is not limited to hosting applications in someone else's data center, though cloud offerings can be used in this way to elastically manage computing resources and circumvent the need to buy
new infrastructure, train new people, or pay for resources that might only be used periodically. Special file system, persistence, data indexing/search, payment processing, and other cloud services can provide benefits to those who deploy platforms in clouds, but their use often requires modifications to platform functionality so that it interoperates with these services. Before the term cloud, the term service grid was sometimes used to define a managed distributed computing platform that can be used for business as well as scientific applications. Said slightly differently, a service grid is a manageable ecosystem of specific services deployed by service businesses or utility companies. Service grids have been likened to a power or utility grid … always on, highly reliable, a platform for making managed services
available to some user constituency. When the term came into use in the IT domain, the word service was implied to mean Web service, and service grid was viewed as an infrastructure platform on which an ecology of services could be composed, deployed, and managed. The phrase service grid implies structure. While grid elements, servers together with functionality they host within a service grid, may be heterogeneous vis-à-vis their construction and implementation, their presence within a service grid implies manageability as part of the grid as a whole. This implies that a capability exists to manage grid elements using policy that is external to implementations of services in a service grid
(at the minimum in conjunction with policy that might be embedded in
legacy service implementations). And services in a grid become candidates
for reuse through service composition; services outside of a grid also are
candidates for composition, but the service grid only can manage services
within its scope of control. Of course, service grids defined as we have
above are autonomic, can be recursively structured, and can collaborate in
their management of composite services provisioned across different grids.
Clouds and service grids both have *containers*. In clouds, *container* is used
to mean *a virtualized image containing technology and application stacks*.
The container might hold other kinds of containers (e.g., a J2EE/Java EE

application container), but the cloud container is *impermeable*, which means that the cloud does not directly manage container contents, and the cloud contents do not participate in cloud or container management.

In a service grid, *container* is the means by which the grid provides underlying infrastructural services, including security, persistence, business transaction or interaction life cycle management, and policy management.

In a service grid, it is possible for contents *in* a container to participate in grid management as a function of infrastructure management policies harmonized with business policies like service level agreements. It also is possible that policy external to container contents can *shape2* how the container's functionality executes. So a service grid container's wall is permeable vis-à-vis policy, which is a critical distinction between clouds and service grids3.

2 The sense of the word shape is consistent with how policy is applied in the telecom world where, for example, bandwidth might be made available to users during particular times in the day as a function of total number of users present.

3 Cloud management typically is exposed by the cloud vendor through a dashboard. Vendors like Amazon also make functionality underlying the dashboard available as Web services such that cloud users' functionality could programmatically adjust resources based on some internal policy. A service grid is constructed to actively manage itself as a utility of pooled resources and functionality for all grid users. Hence, a service grid will require interaction with functionality throughout the grid and determine with the use of policy extension points whether resource supply should be adjusted.

Cloud computing

A *cloud*, as defined by the cloud taxonomy noted earlier, *is not necessarily a service grid*. There is nothing in cloud definitions that require all services hosted in them to be manageable in a consistent and predetermined way4. There is no policy engine required in a cloud that is responsible to harmonize policy across infrastructure and business layers within or across its boundaries, though increased attention is being given software vendors to policy-driven infrastructure management. Clouds are not formed with registries or other infrastructure necessary to support service composition and governance.

However, a service grid can be formed by implementing a cloud architecture, adding constraints on cloud structure, and adding constraints on business and infrastructure architecture layers so that the result can be managed as both a technology *and* a business platform.

Architecture transformation

How to construct an Outside-In architecture that meets next century computing requirements is a topic that requires debate. Should we leverage our past investments in infrastructure, bespoke software development, and third party software products? If so, how can we self-fund this and how long will it take? Or do we go back to the IT funding well with rationale that defends our need now to develop a new service platform and jettison that multimillion-dollar investment we just barely finished paying off?

The answer is *it depends*. We've seen both approaches taken. And we've seen that development of a new platform is no longer as drastic as it sounds.

Transforming an existing architecture

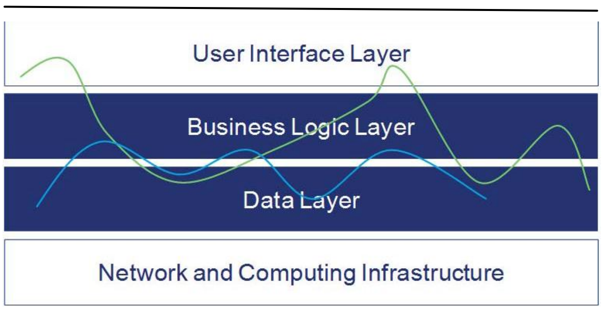
It is enticing to think that one could implement an Outside-In architecture simply by wrapping an existing Inside-Out application platform with Web service technologies to service-enable it.

Not quite.

It *is* possible to do that *and then* evolve the Inside-Out architecture to an Outside-In one as budget and other resources allow using a strategy very similar to Shinsei's *business interface strategy* discussed in the introduction of this paper. But the fact that an Inside-Out architecture typically is *not* service-oriented — even though it might be possible to access application functionality *using* Web services — suggests that just using the wrapper strategy will not yield the benefits of a full Outside-In architecture implementation, and compensation for Inside-Out architecture limits may even be more costly than taking an alternative approach.

To illustrate the process of converting an Inside-Out architecture to an Outside-In one, we consider how a typical Web application platform could be converted to an Outside-In architecture in which some Web application accesses all critical business functionality through a Web services layer, and Web services are hosted in a cloud, a service grid, or internally.

From a layered perspective, a Web application usually can be described by a graphic of a three-tiered architecture like the one below.



At the top of the graphic we see a user interface layer, which usually is implemented using some Web server (like Microsoft's IIS or Apache's HTTP Web server) and scripting languages or servlet-like technologies that they support. The second layer, the business logic layer, is where all business logic programmed in Java, C#, Visual Basic, and php/python/perl/ tcl (or pick your favorite programming language that can be used to code libraries of business functionality) is put. The data layer is where code that manipulates basic data structures goes, and this usually is constructed using object and/or relational database technologies. All of these layers are deployed on a server configured with an operating system and network infrastructure enabling an application user to access Web application functionality from a browser or rich internet client application.

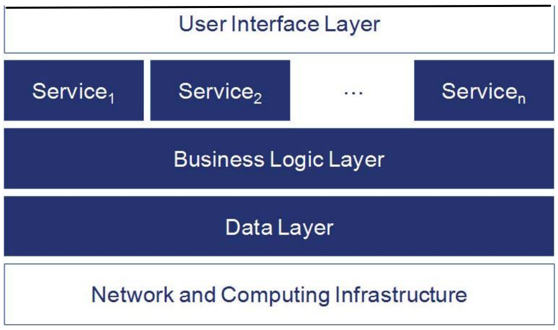
The blue and red lines illustrate that business and data logic sometimes are commingled with code in other layers of the architecture, making it difficult to modify and manage the application over time (code that is spread out and copied all over the architecture is hard to maintain). Ideally, the red and blue lines would not exist at all in this diagram, so it is here where we start in the process of converting this Inside-Out architecture to an Outside-In one.

4 This should not suggest that clouds and elements in them are not managed, because they are. Service grids, however, impose an autonomic, active, and policy-based management strategy on all of the elements within their scope of control so that heterogeneous application and technology infrastructure can be managed through a common interface that can be applied to fine-grained grid elements as desired or necessary.

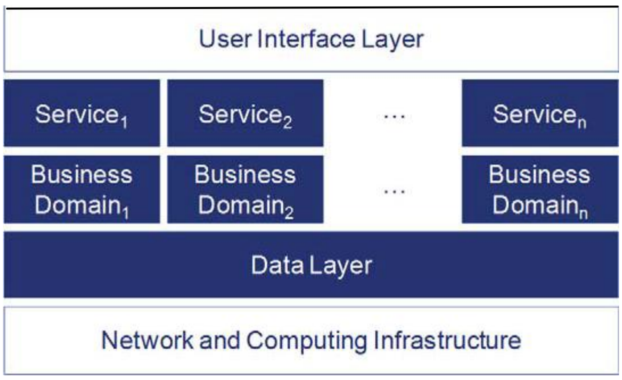
Addressing architecture layering and partitioning

The first step of transitioning from one architecture style to another is to correct mistakes relating to layering wherever possible. This requires code to be cleaned and commented, refactored, and consolidated so that it is packaged for reuse and orderly deployment, and so that cross-layer violations (e.g., database specifics and business logic are removed from the UI layer, or business logic is removed from the data layer) are eliminated.

Assuming layering violations are addressed, it makes sense then to introduce a service application programming interface (API) between the User Interface Layer and the Business Logic Layer as shown in the slightly modified layer diagram below:



The service layer illustrated here is positioned between the User Interface and lower architecture layers as the *only* means of accessing lower level functionality. This means that the concerns of one architecture layer do not become or complicate the concerns at other levels. But while we may have cleaned up layering architecture violations, we may not have cleaned up *partitioning* violations. Partitioning refers to the "componentizing" or "modularizing" of business functionality such that a component in one business functional domain (e.g., order management) accesses functionality in another such domain (e.g., inventory management) through a single interface (ideally using the appropriate service API). Ensuring that common interfaces are used to access business functionality in other modules eliminates the use of private knowledge (e.g., private APIs) to access business functionality in another domain space. Partitioning also may be referred to as *factoring*. When transitioning to a new architecture style, the first stage of partitioning often is implemented at the Business Logic Layer, resulting in a modified architecture depicted as follows:



The next phase of transformation focuses attention on partitioning functionality in the database so that, for example, side effects of inserting data into the database in an area supporting one business domain does not also publish into or otherwise impact the database supporting other business domains.

Why go to such trouble?

Because it is possible to transition the architecture in Figure 1 to become like one of the depictions below. Figure 4 illustrates a well-organized platform that might be centrally hosted.

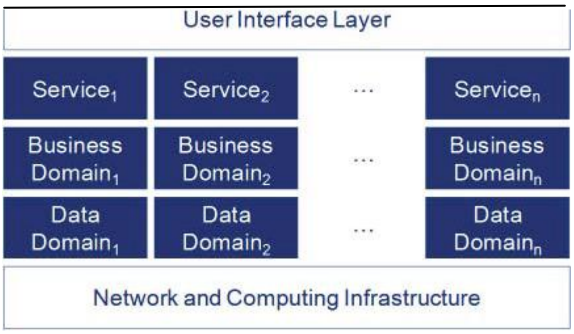
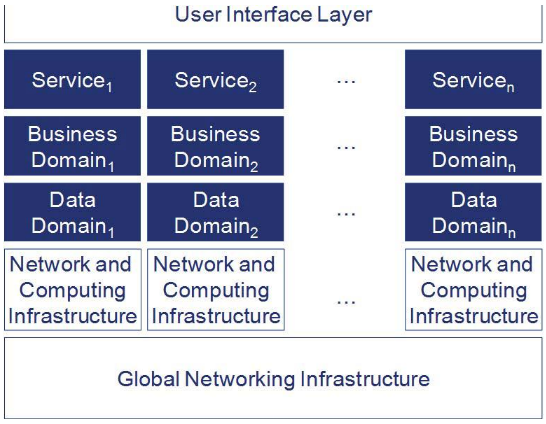


Figure 5 illustrates a well organized platform that could be hosted in a service grid or even many service grids.



Figures 4 and 5 make it simple to see that services and their supporting business logic and data functionality could be replaced easily with an alternative service implementation without negatively impacting other areas of the architecture, *provided that* functionality in one service domain is accessed by another service domain *only through the service interface*.

And such capability is required in order to simplify management of an application portfolio implemented on such an architecture as well as distribute and federate service implementations.

Externalizing policy

The next step toward implementing an Outside-In architecture is to external both business and infrastructure policies from any of the functionality provisioning services illustrated in the figures above.

Our use of the word *policy* connotes constraints placed upon the business functionality of a system, harmonized with constraints on the infrastructure (hardware and software) that provisions that functionality.

These constraints could include accounting rules that businesses follow, role-based access control on business functionality, corporate policy about the maximum allowable hotel room rate that a nonexecutive employee could purchase when using an online reservation service, rules about peak business traffic that determine when a new virtualized image of an application system should be deployed, and the various infrastructural policies that might give customer *A* preference over customer *B* should critical resource contention require such.

Policy extension points provide the means by which policy constraints are exposed to business and corresponding infrastructural5 functionality and incorporated into their execution. They are not configuration points that are usually known in advance of when an application execution starts and that stay constant until the application restarts. Rather, policy extension points are dynamic and late bound to business and infrastructural functionality, and they provide the potential to *dynamically shape* execution of it within the deployment environment's runtime.

Externalizing policy highlights a significant distinction between Inside-Out and Outside-In architecture styles. Inside-out architectures usually involve legacy applications in which policy is embedded and thus externalizing it is — at best — very difficult. Where application policies differ in typical corporate environments, it becomes the responsibility of integration middleware to implement policy adjudication logic that may work well to harmonize policies over small numbers of integrated systems, but this will not generalize to manage policy in larger numbers of applications as would be the case in larger value chains. To illustrate the problem of scaling systems where policy is distributed throughout it, consider the system illustrated in Figure 6.

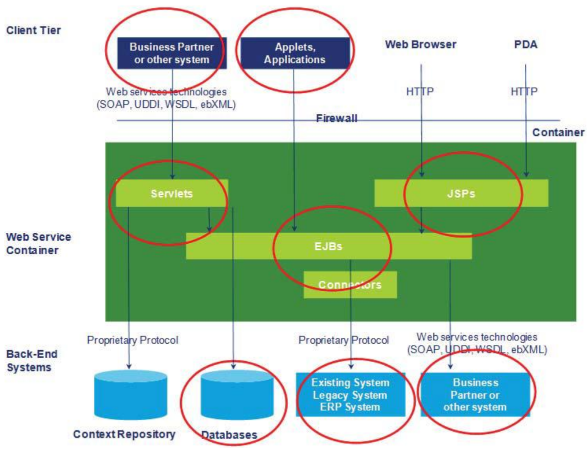
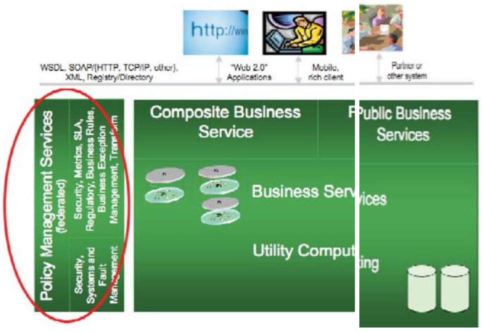


Figure 6 illustrates a system where business policy exists in multiple locations of the architecture as indicated by areas outlined in red. Scaling this architecture would be disastrous because policy would be distributed as copies (or, worst case, as different code bases) over a very complex deployment environment. But a well-factored environment like the ones illustrated in Figures 4 and 5 have business logic located in a single logical architecture layer and, from it, policy can be externalized with the development of adapters or similar architecture components that play the role of policy extension points described above. Once this is accomplished, the architecture we started with now begins to resemble the architecture illustrated in Figure 7 below, in which policy has been externalized, possibly federated, and put under the control of policy management services. Once policy from business functionality is externalized, it can be harmonized with infrastructure policy as feasible/desired.

Replacing application functionality with (composite) services

The final step in transforming an Inside-Out platform to an Outside-In platform is to replace *business application code that coordinates invocation of multiple services* with *composite service* if this is possible.

In Figure 7 we use the term *composite service* to mean business services formed by combining other business services (or methods thereof) together to form coarse (larger) business functions that are peer with application functionality. For example, we might see services to manage order fulfillment, invoice submission and payment processing, orchestrations with which billing staff use to prepare for invoicing, logistics planning, and so forth. As a kind of mental mapping between Figures 1 and 7, the composite service functionality in Figure 7 maps to business logic that has *leaked* into Web pages of the Web application in Figure 1 (shown with red and blue lines) that are used to manage order fulfillment, invoice submission, etc.



Orchestration is often equated to workflows used to coordinate some ordering of service method invocations. Workflow and other business process management technologies are now well-known within today's corporations. Workflow engines for Web services have been commoditized through open source initiatives and by commercial software vendors. These engines make it possible to implement composite Web services as either state machine or sequential workflows. Use of state machine flows makes it possible to avoid prescriptively dictating how systems interoperate. They also provide the opportunity to incorporate human intelligence tasks to help resolve exception conditions that often emerge from composite services or straight through processing flows6.

Starting from scratch — maybe easier to do, but sometimes hard to sell

Many CIOs and IT executives hope that the costs and risks of transforming a legacy platform architecture to an Outside-In one can be amortized over time, and who can blame them. Most have probably spent a considerable sum developing the current architecture, so the last thing any IT executive wants to ask for is new budget sufficient to fund still more infrastructurelevel activities or require their companies to choose between new functionality or resolved infrastructure issues.

But we have experienced many changes in the technology world during the last 20 years that strongly suggest there is value in at least considering whether implementing Outside-In architectures from scratch would be worthwhile. An interesting catch here is that this argument could have been made and *was* made at each new stage of development over the last 20 years. *Why is the story now so different?* Because today's context versus just a few years ago is qualitatively different. Significant broadband capacity, economic storage (both self- and cloud-hosted), cheap memory and modern caching services, commodity 64-bit operating systems, XML accelerators and sophisticated application protocol management capabilities, commoditized integration/interoperability technologies, virtualization and utility computing, cloud and service grid computing, and other relatively recent innovations challenge the traditional wisdom that it is better to evolve and extend an existing platform than it is to create a new one that could circumvent problems from retrofitting an existing architecture in ways quite counter to its original design.

Coupled with these advances are elaborations of *industry* domains in the form of *industry* or *business solution maps*. These maps are used by consulting companies and software vendors to provide business process oriented views of industry, define roles played and responsibilities performed within business processes, begin (at least) to build out functional decompositions of the industry domain, and map processes to technology solutions where feasible. Using these maps as starting points streamlines process and data mapping efforts that used to take months to even several years to perform (in larger companies), and results in a detailed functional view that is necessary to build a well-formed Outside-In architecture.

Building from scratch is really not the same as starting with nothing but a blank sheet of paper. While it is unusual to find a company able to take a purely greenfield approach (unless it is a startup), there are ways for established businesses to get comfortable with taking a greenfield approach to developing an Outside-In architecture, and subsequently developing a strategy to implement it even if using components of existing platforms.

Concluding remarks

Transforming an Inside-Out architecture to an Outside-In architecture can be a lengthy process — it is a function of existing system complexity, size, and age. One company who shared with us its experiences when making such a transition was Rearden Commerce (Rearden). Prior to three and a half years ago, Rearden's architecture was composed like many of the Web applications we see today: three-tiered, open source Web and application server technologies, and a relational database. Rearden's Web application exposed a framework to which merchant clients could interface to Rearden "services" or functions. Rearden's management team had the foresight to recognize the company's need to create a platform (not just an application), and the corresponding need to make architecture changes to support more rapid development and simpler deployment of new services.

By this time, Rearden already had clients, so it understood that change had to be made transparently to its user base whenever possible or in a way that the user base viewed as a positive upgrade of capability to which they could migrate as doing so became expedient to their business.

Rearden strengthened its leadership team with technologists who had participated in Web service infrastructure companies and could guide

in Rearden's architecture modernization. This new leadership team
undertook a transformation of the company's three-tiered architecture to
a service-oriented one over a two-year period using a process like the one
described above. At the end of the two-and-a-half-year period, Rearden
had transformed its traditional Web application architecture to a service
oriented one with externalized policy management.

When performing an architecture transformation, is it necessary that *all*
architecture components are *entirely* transformed — as was the case with
Rearden? If there was queue-based middleware in the old architecture,
should it be replaced? Should all *old* applications be replaced with custom
applications having appropriate policy extension points?

6 Ultimately, it may prove necessary to incorporate a constraint engine into the way that services are composited to harmonize policies and dynamically
govern execution of the composite..

Cloud computing

The answer to these questions is *it depends*. Certainly it is possible to
replace enterprise application integration technologies with commodity
or open source technologies, simplify them, or maybe — in some cases
— even eliminate them. It is unlikely that middleware supporting reliable
messaging and long-lived business transactions between business partners
needs to be totally replaced in or removed from an Outside-In architecture.

But its use can be couched in ways that eliminate tight coupling between
partners, and commingling of business policy with integration functionality
that makes partner integration difficult to change as policies change or as
a partner networks expand.

Taking an Outside-In point of view requires that we separate concerns
from the start. Application platforms should be viewed as distributed
from their beginning rather than be made so after the fact by attaching
some distribution layer to them. We must understand how we have
permitted business security and access control models to be built into
our architectures and how, now that technology innovations enable us
to challenge these limits, we must remove them from our computing
platforms to realize business agility goals that will be demanded of an
architecture in the twenty-first-century. Technologies we've used in
the past can be useful to us in the future. Success in implementing an
Outside-In architecture is less a function of technology than it is of a
business and technology architecture vision that forces business and
technology architects to view business capabilities from a global, outside in
and top down perspective.

REFERRENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A.

Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M.

Zaharia. Above the Clouds: "A Berkeley View of Cloud

computing". Technical Report No. UCB/EECS-2009-28,

University of California at Berkley, USA, Feb. 10, 2009.

[2] L. Vaquero, L. Merino, and J. Caceres. "A break in the

clouds: towards a cloud definition". SIGCOMM Comp.

Communications Review, vol. 39, pp. 50—55 (2009).

[3] L. Youseff, M. Butrico, and D. Da Silva. "Toward a Unified

Ontology of Cloud Computing," Grid Computing

Environments Workshop (GCE '08), pp. 1—10 (2008).

[4] P. Mell and T. Grance. “The NIST Definition of Cloud

Computing”. National Institute of Standards and Technology

(2009).

[5] Shufen Zhang, Shuai Zhang, Xuebin Chen, Shangzhuo Wu.

“Analysis and Research of Cloud Computing System”.

Instance 2010 Second International Conference on Future

Networks, pp. 88 – 92.

[6] Xu Lei, Xin Zhe, Ma Shaowu, Tang Xiongyan. “Cloud

Computing and Services Platform Construction of Telecom

Operator”. Broadband Network & Multimedia Technology,

2009. IC-BNMT '09. 2nd IEEE International Conference on

Digital Object Identifier, pp. 864 – 867.

[7] Thomas Dietinger. “Aspects of e-Learning Environments”.

Dissertation for the Award of the Academic Degree Doctor

of Technical Sciences at Graz University of Technology.

Retrieved January 12, 2008, from: http://www.iicm.tugraz.

ac.at/thesis/tdieting_diss.doc

[8] Ministry of Education Government of People’s Republic

http://www.moedu.gov.uk/.