Abstract

Service model stack diagrams are sometimes presented when cloud systems are discussed. These diagrams are based on service models defined by the National Institute of Standards and Technology and inspired by the desire to build cloud applications out of interchangeable parts. We consider the possibility of the three service models forming a stack. Placing existing software in the model or defining implementations for the model is hard because it is so general. We introduce terms solution and component to pinpoint partial layer implementations and pieces of software within the layers. We take into account the need to define interfaces in an iterative manner as we learn about the requirements.

Keywords: cloud, software, definition, layer model

1 Introduction

Cloud computing refers to a system distribution model where some computer resources are provided by a third party for use over the network [Armbrust et al. 2010; Vaquero et al. 2008]. The US based National Institute of Standards and Technology (NIST) defines three service models for cloud computing [Mell and Grance 2009]. The service models are used to categorize service providers based on the abstraction level of the provided service. Infrastructure as a service (IaaS) providers abstract physical hardware from their customers and provide virtual machines. Platform as a service (PaaS) providers abstract multiple computing nodes of the infrastructure into a single application development platform. Software as a service (SaaS) providers run services that are useful to the end users, mapping computations into meaningful things such as an email or a photo album.

Technology stacks are motivated by the success of the TCP/IP stack which is based on the more abstract OSI model that defines abstraction layers for data communication systems. In a stack model the user uses services provided by the top layer, and the top layer uses services provided by the lower layers. The layered stack representation implies that implementations of different layers are interchangeable. The borders between layers symbolize interfaces that implementations of the different layers use to communicate with each other. The goal of the model is to define these interfaces to allow competition on providing implementations for different layers.

We consider the possibility of the three service models forming a stack. A service of any of the three service models has its limitations. None of them can support all use cases or applications. Therefore, for example PaaS services are often not interchangeable. In practice the platform’s type is defined by the kind of applications it supports. Thus the service model stack must be refined to accommodate multiple possibly overlapping elements on each layer. We introduce some new terminology to clarify the model.

2 Service Model Stack

Service model stack diagrams are sometimes presented when cloud systems are discussed [Lenk et al. 2009; Rimal et al. 2009; Bhardwaj et al. 2010]. We have included a picture of a service model stack as Figure 1. While the service model definition helps us categorize different types of cloud services, a stack model may help us visualize how some service models are more abstract than others. The stack model also implies that the upper layers are implemented on top of the lower layers. An SaaS email application may be implemented on top of a PaaS offering, while the PaaS provider may use services from an IaaS partner to acquire resources based on the timely need.

Placing existing software in the service model stack is hard because the model is so general. Let us consider a database system for example. It is possible to provide a database as an SaaS service by providing raw database access directly to the user. A database system can provide some support for defining application specific user interfaces to the user. This would turn it into a PaaS offering. Finally a database system often provides long term storage which implies it is part of the storage infrastructure, making it an IaaS service. A database system might be very limited either in its user friendliness, its support for custom applications or its support for complex computations. Thus a database system might not alone fulfill the requirements for a layer in the service model stack.

Defining what an implementation of a layer in the service model stack would look like seems impossible. An implementation of the SaaS layer needs to contain all possible software applications needed by humankind. The PaaS layer needs to support building all of those applications, and the IaaS layer needs to have the correct abstraction for supporting all the applications and platforms. If we can not define what the implemented layers should do we can not define how the interfaces between those layers would look like.

It is impossible to define interfaces for all future needs. Thus designing cloud interfaces needs to be an incremental process. We are seeing some competition between Open Cloud Computing Interface (OCCI) and Elastic Compute Cloud (EC2) API. In addition to the competition in defining the interfaces it is also useful to have some competition in implementing those interfaces. Some competition currently exists between Eucalyptus, OpenStack and Amazon Web Services (AWS) on the IaaS layer, but also between Google App Engine, Heroku and Cloudfoundry on the PaaS layer. To keep the competition alive we need to allow incomplete, scenario-specific interfaces to be defined and implemented.

3 Solution Stacks

We suggest that term solution be used when referring to competing implementations within a layer. Solutions need not be full implementations of the service model layer they target. Instead solutions may produce the cloud services needed for a specific use case.
Two SaaS solutions, say Facebook and Google Docs, may implement completely orthogonal parts of the same layer. This does not stop two IaaS solution providers behind OpenStack and Eucalyptus from working together on their shared functionalities. Depending on timely motivations the two service providers may either compete in defining a better interface or strive to standardize known good parts of the interface.

Standardizing solution interfaces allows other solutions on the layer above to be built without knowing internal structure of the services provided at the lower layer. Ideally such standards would make it possible to move from one service provider to another. While solutions may overlap in functionality and interfaces, their non-functional qualities can still be different. For example two storage services could have different reliability guarantees. Combining this with common interfaces allows competition between solutions providing similar services.

We suggest that term component be used for referring to a piece of software used to provide a service. Components need not be designed for one layer specifically. Thus the database system from our example in the previous chapter could be used for implementing solutions on multiple layers. In practice it may be desirable to further limit the scope when designing new software components.

When we talk about components and solutions we are talking about the technical parts that are used by a service provider to provide a service. For a piece of software to become part of a service, someone needs to run it. For example AWS is a live service and is not considered a solution in this sense while the software used to run AWS definitely counts as a solution. While the software for a solution may be publicly available, all components and solutions need not be published. Some companies may use in-house developed components in their solutions. This classification should not be affected by such details.

![Figure 2: Component/solution model with practical examples.](image-url)

Figure 2 visualizes how solutions and components fit into the service model stack. The figure shows the theoretical stack model next to some examples of solutions and components on different layers. On the top there is a software project management solution called Launchpad. It is used by software developers for bug tracking, software translation, and release planning. Malone is a bug tracking application while Rosetta is used for translating software user interfaces into foreign languages. Google App Engine is a web application platform. Its applications are written in the Java programming language and it uses High Replication Datastore (HRD) for long term storage. Our example IaaS solution is a combination of OpenStack and GlusterFS. OpenStack is used to manage virtual machines while GlusterFS is used for long term storage. The components within each solution in the figure are examples and do not compose the whole solution. For example Google App Engine uses Python for some applications and Launchpad requires some combination of Linux, Apache, MySQL and Perl/Php/Python (LAMP) components to run.

### 4 Conclusion

We studied how a cloud service stack differs from traditional networking stacks with its vague interfaces. We noticed some problems with the service model stack being too general for discussing cloud interfaces. We suggested the term solution to be used for the compilation of software required for providing a service and the term component to be used for pieces of software in such compilations.

Building cloud services from stackable parts would seem to require standardization for interfaces between the layers in the stack. At the moment it seems too early to say whether or not clouds will eventually become truly stackable in a way that would allow migration between providers of lower levels. This may not only be a question of interfaces, but also about other types of co-operation between service providers.

### Acknowledgements

The authors would like to thank Ville Palkosaari, Sami Saada, Adam J. Oliner and Flutra Osmani for constructive feedback on the original draft.

### References


