# Architecture Patterns of Web Services Applications

Neil B. Harrison

Utah Valley University

[neilmoose@hotmail.com](mailto:neilmoose@hotmail.com)

## Abstract

With the advent of web services platforms, the architectures of applications deployed on these platforms have evolved from architectures found in early software systems. A study of reference architectures provided by Amazon Web Services showed eight architecture patterns. These patterns are similar to the original software architecture patterns, but they may be sufficiently different to be considered separate patterns in their own right. If they are not new patterns, it is still worthwhile to document them as modern uses of legacy architecture patterns. This paper presents the following web services architecture patterns Broker-Server, Advanced Repository, and Front-End Broker.

## Introduction

In recent years, the landscape of software applications has changed dramatically. Applications are no longer standalone software systems residing a single computer or a self-contained network. The rise of the Internet has enabled systems to be distributed in ways thought unthinkable not long ago. For example, all computing used to be done locally: if you wanted to use a particular application, it had to be installed on your own computer (or server). But cloud computing has introduced a new paradigm of computation. It is no longer necessary to install a program on a local machine to use it; instead, the program can run (and be managed) on a remote computer through the web.

For the purposes of this work, web services are scoped to those provided by Amazon Web Services (AWS), as well as any others that follow the same model. A typical AWS application consists of an application hosted by Amazon, accessed through the web. The application contains some sort of gateway (such as web servers), one or more application servers, and one or more database servers. The gateway usually provides security as well as load balancing, the application servers often dynamically scale, and the database servers often provide high reliability through some form of replication.

There are two tightly coupled architectural facets of web services applications. In a sense, each makes the other viable. The first is that computation can happen in the cloud: an application doesn’t run on a local computer, including a local server, but rather runs on an arbitrary computer “in the cloud”, managed by a provider’s web services software. Second, the web service provider has numerous components which facilitate the running and management of a client’s application.

Because the computing paradigm for web services is different from legacy computing, the basic architectures may also have changed from common architectures. There may be new architecture patterns, or variations on existing patterns. This paper is all about these architecture patterns.

These patterns come from studies of Amazon Web Services architectures. There is considerable documentation of AWS-based architectures available at [www.amazon.com/architecture](http://www.amazon.com/architecture). The following AWS architectures were studied:

|  |  |  |
| --- | --- | --- |
| **NAME** | **Abbr.** | **Comments** |
| Web application | WEB |  |
| Content and media serving | CM |  |
| Batch processing | BCH |  |
| Fault tolerance and high availability | FT |  |
| Large-scale computing | LRG |  |
| Ad serving | AD |  |
| Disaster recovery for local applications | DIS |  |
| File synchronization | FILE |  |
| Media sharing | MEDI |  |
| Online Games | GAME |  |
| Web log analysis | LOG |  |
| Financial services grids | FNCE |  |
| E-Commerce website | ECOM | AWS has three diagrams which interlink |
| Time Series processing | TIME |  |
| Image moderation chatbot | IMGM | Serverless chatbot that removes offensive images; different than the others |
| Microsoft SQL Server reference architecture | SQLS | Architecture that uses SQL Server |
| Drupal Hosting | DRU | Hosting a Drupal site on AWS |
| WordPress Hosting | WRDP |  |
| Varnish Hosting | VARH |  |
| Citrix virtual apps and desktops service | CTX | Hosting Citrix apps |
| MicroFocus Enterprise Service | MFOC | Hosting MicroFocus |
| TIBCO Data Science | TIBC | Hosting TIBCO |

Because this work is based on AWS, it is specific to their systems. The architectures are necessarily strongly shaped by the components provided. Other web services providers, such as Microsoft Azure, appear to have similar general architectures, but different types of components, which may result in somewhat different architectures (see <https://azure.microsoft.com/en-us/solutions/architecture/>.) A general study of major web services providers is therefore desirable.

## General Results

The AWS architectures contained eight different possible architecture patterns. Virtually all of the patterns are very similar to existing architecture patterns. In fact, they might be considered as either applications (in some cases) or variants of the legacy architecture patterns. A summary description of each pattern, with commentary about its relationship to legacy architecture patterns, follows:

Broker-Server. This consists of a Broker component that distributes requests to a set of servers. It is a special application of the original Broker pattern, in that the set of servers is auto-scaling: when load increases, additional servers are brought into service, and when load decreases, servers are taken out of service. It isn’t specified where the control of the auto-scaling happens; however, the AWS component used most commonly as the broker gives a clue: it is referred to as “Elastic Load Balancing”. This appellation may imply that the auto-scaling is controlled by the broker component. Note that the servers provide a hosting site for user application code, so they function as virtual computers upon which the users run their own applications.

Front-End Broker. Another implementation of the original Broker sits at the entry point to several systems. It distributes requests, but in some configurations requests may be directed to a cache to enhance performance (see the Content and Media Serving architecture, for example.) Its main function is low-latency con-tent delivery, but it may also provide a security layer.

Advanced Repository. This is an application of the original pattern Shared Repository, with implementation of advanced features. It includes explicit support for reliability, availability, and scalability. It implements most of these quality attributes through replication and auto-scaling.

High-Availability 3-Tier. This pattern is an application of or extension to the original 3-Tier pattern. The first tier is a Broker-Server (see above) with a set of web servers. The second tier is another Broker-Server, this time with a set of application servers. The third tier is an Advanced Repository. Availability is supported throughout all the layers by having each set of servers implement replication. The documentation shows two “availability zones”, which cut through all three tiers. It does not indicate how the zones are implemented. It may be that the Broker components in the Broker-Servers might have some responsibility for managing the replication. Some of the architectures that use this pattern also show a Front-End Broker in front of everything. It could be considered to be part of this pattern, extending it to four tiers.

Map-Reduce. Map-Reduce is a large-scale component that plays a prominent role in several of the architectures. It is a component provided by AWS. It is called “elastic”, and as such, has the ability to replicate and scale for reliability, cost, and scalability. Internally, a map-reduce component follows the original Pipes and Filters architecture pattern.

Streaming Analysis. Two architectures are concerned with analysis of streaming data. Initially, data comes to a collection of processors, such as servers. The data goes to an Advanced Repository for storage, then to a Map-Reduce component. The reduce data is stored in another Advanced Repository, where it can be subjected to further analysis. This is also a use of the Pipes and Filters architecture pattern.

Notifier. A few systems notify people for various reasons. AWS has two similar notification services, Simple Email Service (SES), and Simple Notification Service (SNS). This is essentially an application of the publish-subscribe pattern. It appears that the user of the architecture should provide code to specify under what conditions data is pushed to the subscribers. The notification services may include support for scaling and possibly availability.

Application Platform. The platforms that host various commercial products (some of which are themselves application platforms) have similar patterns of structures and connections. The pattern approximately follows a 3-Tier or 4-Tier architecture. The structure of this pattern is as follows: there is a virtual private cloud which spans two “availability zones”, and contains two public and private subnets. There is one public and one private subnet in each availability zone. Within the public subnets, there is an auto-scaling group of gateways or hosts. Within the private sub-nets, an auto-scaling group hosts the application. These auto-scaling groups are man-aged by load balancing, and appear to be instances of the Broker-Server proposed pattern. There are slight variations: the Citrix Virtual Apps architecture includes four private subnets, and the Varnish architecture does not include any private subnets; the public subnets host the Varnish application.

A point of discussion is whether these are patterns in their own right. Because these potential patterns are firmly in the context of web services, and implement the quality attributes of security, scalability, performance, and reliability, it may be worthwhile to consider them as new patterns. On the other hand, if they are simply the legacy patterns, they still provide modern uses of them. And there is the philosophical question of how much variation of a pattern is necessary to constitute a new pattern. These are as yet unresolved questions which may generate insightful discussions.

## The Patterns

Note: this is an incomplete work. The following three patterns are a start, to be followed in the future by the other patterns.

The general context is web services applications built on AWS components. The web service provider (in this case AWS) provides a platform upon which user applications can be deployed.

## Broker-Server (aka Broker-Dynamic Load Balancer)

### Context

The application to be deployed on AWS has an uneven pattern of usage. Over time, the usage will probably increase. Clients of the application depend on it to be available all the time. (Exact availability expectations are not given, but an estimate would be in the range of 99.99 percent availability.)

### Problem

A web services application needs to handle dynamic loads and be highly available, but keep ongoing operating costs low.

### Forces

As a user considers using web services (in this case AWS), there are several different forces which one must carefully consider.

* Operating cost: users are charged fees for hosting their application. These fees generally include fees based on the usage of computing resources. For example, they might be charged a based on the CPU-minutes used in a month. Naturally, users wish to minimize their ongoing costs. In particular, users don’t want to pay for unused capacity.
* Dynamic loads: For many, if not most, applications, usage is uneven. For example, if an e-commerce company launches an advertising campaign, clients may flood the application with requests. Users want to handle all client requests, and in a timely manner. Therefore, it may be desirable to have high service capacity. But when demand slackens, the user may be stuck with idle capacity.
* Scalability: Users expect their business to grow. There should be a way to scale up computing capacity as needed. Starting out with high capacity is also not financially wise.
* Availability: Clients expect applications to be available at all times. The typical approach to increasing availability is to have redundant systems, so if one fails, the other can take over. However, this increases costs, and often forces unused excess capacity.
* Development cost: Users need to keep development costs low. However, implementing quality attributes (in this case cost, scalability, and availability) it typically difficult and expensive.

### Solution

Use a broker that takes the role of a server in a client-server paradigm. In particular, it dynamically manages the computing resources, adding or removing resources as needed.

Amazon refers to these as elastic load balancers. A load balancer serves as the single point of contact for the clients. It distributes incoming traffic across multiple targets. When incoming traffic increases, the load balancer can increase the number of targets to handle the additional requests for service. When the traffic decreases, the load balancer removes excess targets from service.

The load balancer also monitors the health of its registered targets and ensures that it routes traffic only to healthy targets. When the load balancer detects an unhealthy target, it stops routing traffic to that target, and then resumes routing traffic to that target when it detects that the target is healthy again.

The user of an AWS load balancer can specify the minimum and maximum number of targets. A minimum of at least two helps achieve high availability. A maximum number of targets creates an upper bound for cost of services.

### Consequences

Elastic scaling managed by a load balancer helps minimize user costs. Because additional targets are employed only when needed, and freed when no longer needed, the user pays only for the computing resources that are actually needed. The user does not need to pay for extra capacity that is mostly idle.

Because of the ability to set a maximum number of targets to be used, the user can control the maximum cost of the system. However, setting a maximum creates a risk that under very heavy load, clients may not get optimal service if the load exceeds the capacity of the maximum targets. Users should monitor usage to determine whether maximum limits are appropriate.

The load balancer increases system availability by checking the health of the targets, and not routing requests to unhealthy targets. A minimum of two targets increases availability through redundancy. Of course, high availability through redundancy increases the minimum cost of the service. Potential users should weigh the benefits and the costs of high availability.

Although availability is increased through redundant targets, the load balancer itself is a single point of failure. To my knowledge, AWS does not offer architectures with redundant load balancers. Perhaps there is redundancy or other availability mechanisms in the load balancers themselves.

An important benefit is that redundancy, scalability, and cost management are all implemented in the elastic load balancing components and architecture provided by AWS. Such code is generally very difficult to write, but the users of the AWS architectures and components don’t have to write it themselves.

Note that the AWS load balancing components are not to be used by themselves. They require use of other AWS components to complete the system (i.e., they interface with each other.) This should cause no difficulty for users, as there is a large suite of AWS and AWS-compatible components, as well as numerous AWS reference architectures.

## Advanced Repository

### Context

Most web services applications need persistent storage of data.

### Problem

In a web services application, data storage must be reliable, available, scalable, and secure.

### Forces

Persistent storage raises the following forces:

* Deployment: As the computing service is hosted in the cloud, it would be more convenient for the data storage to also be hosted in the cloud.
* Operating cost: To keep operating costs low, the data storage should not have higher capacity than is needed.
* Scalability: The data storage must be able to increase as more data is stored. Data storage will likely increase even if the application’s business doesn’t expand.
* Availability: It must meet the availability of the entire system. Availability can be enhanced through redundancy. This increases operating costs.
* Reliability: The integrity of the data must not be compromised. The reliability of the data can be improved also through redundancy.
* Security: the data must be held secure from attacks including theft and data corruption.

### Solution

Build an advanced repository as a single component in the architecture. The Advanced Repository has the following capabilities within itself:

* It can dynamically grow to handle increased storage requirements. It can be initially configured to be the appropriate size for the application (it doesn’t need to start out huge.)
* It guarantees availability. This is typically done through redundancy, but it is transparent to users.
* It guarantees reliability. This may be done through redundancy and automated backups, but it is also transparent to users.
* The data may be encrypted, and access may be controlled (e.g., through a system of authentication.) This is implemented within the Advanced Repository, although access control is of necessity not transparent to users.

An Advanced Repository may be shared (see Shared Repository.)

### Consequences

Implementation of reliability and availability may require redundant storage; thus the ongoing cost will likely increase to handle the redundancy.

The implementation hides details of availability, reliability, and scaling, and most of the details of security.

Because of differences in users, a single type of Advanced Repository may not be optimal for all. In fact, AWS has three different repository components.

## Front-End Broker

### Context

There are two possible contexts; both are possible.

An application might have different actions for different requests. These actions are such that they are best handled by different types of targets. However, clients access the different services through the same portal.

Alternatively, the application may be such that similar requests are often made, and can be serviced with the same results. It is desirable to optimize performance.

### Problem

A single entry point for clients may go to different applications.

In addition, performance and security can be important issues.

### Forces

* Different actions should be taken, depending on the input. These different actions require routing to different types of targets to service them.
* In order to optimize response time, and save processing costs, results of certain common queries can be cached. This is complementary to the previous force, in that a cache becomes a different destination of routing. Thus, both can be satisfied by the same approach.
* Scalability, extensibility: It should be possible to add capacity to handle increased load. Similarly, it should be possible to extend the capabilities of the application by adding new types of targets.
* Security of transactions is important

### Solution

Use a Broker that is a single entry point and front-end processor for all requests. The Broker routes requests to different services depending on the nature of the request. For example, the AWS Content and Media Serving reference architecture shows requests being routed to either a content retrieval service or a streaming service.

The Broker may also manage a cache. For example the AWS Content and Media Serving reference architecture shows that content requests are first routed to a cache. When the cache misses, the system retrieves the data from the data storage system.

The Broker may also implement security measures as desired.

### Consequences

The Broker can help provide a unified interface to the client. For example, the client does not have to care whether requested content is streamed live or retrieved from an archive; the interface can be the same.

The fact that the Broker manages the cache not only improves performance, it can help minimize operating costs. Content found in the cache may reduce the demand on the compute servers, and thus reduce the computing time charged to users.

Users of this pattern (and the components themselves) may not have to concern themselves much with implementation of security measures. However, it does not necessarily protect against all security vulnerabilities. For example, users must employ secure coding practices in their application code.

A point of discussion is whether this is a separate pattern from the original Broker architectural pattern. The structure of the components is basically the same for both.