Architecture Patterns of Web Services Applications

Neil B. Harrison
Department of Computer Science, Utah Valley University, Orem, Utah, USA, neil.harrison@uvu.edu

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 briefly describes the patterns found and gives their frequency. It also describes three of the web services architecture patterns in detail.

KEYWORDS

Software Architecture Patterns, Web Services, Quality Attributes

1 Introduction[1]

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.

Several cloud computing/web services platforms are not available and widely used. An application developer can use these platforms (and the components of a platform) to provide the infrastructure for computing, especially in a client-server paradigm [1], and simply implement the domain-specific application. Popular platforms include Amazon Web Services (AWS) [2], Microsoft Azure [3], and Google Cloud [4].

We wished to study modern architectures built on these platforms. What architecture patterns do they employ? Which existing patterns (see [1] and [5]) are used? Are there new architecture patterns?
We studied multiple reference architectures from AWS. 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.

2 The Study of AWS Architectures

The following AWS architectures were studied:

Table 1: AWS Architectures 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. A general study of major web services providers is therefore desirable.

3 Results

Several patterns appear in the AWS reference architectures. They may be variants of existing architecture patterns, or may be new patterns in their own right. This paper gives short descriptions of the patterns found, and shows their frequency in the architectures studied. It then gives detailed descriptions of three of the patterns.

3.1 Patterns found

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.

3.1.1 Dynamic Broker

This pattern consists of a Broker component that distributes requests to a set of servers. It is a special application of the original Broker pattern [1], 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 implies 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.

3.1.2 Content Broker

Another implementation of the original Broker sits at the entry point to several systems. It distributes requests based on request content. In some configurations it may direct requests to a cache to enhance performance (see the Content and Media Serving architecture, for example.) Its main function is low-latency content delivery, but it may also provide a security layer.

3.1.3 Advanced Repository

This is an application of the original pattern Shared Repository [1], 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.

3.1.4 High-Availability 3-Tier

This pattern is an application of or extension to the original 3-Tier pattern [5]. The first tier is a Dynamic Broker (see above) with a set of web servers. The second tier is another Dynamic Broker, this time with a set of application servers. The third tier is an Advanced Repository. It supports availability 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 Dynamic Brokers might have some responsibility for managing the replication. Some of the architectures that use this pattern also show a Content Broker in front of everything. It could be considered part of this pattern, extending it to four tiers.

3.1.5 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 [1].

3.1.6 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.

3.1.7 Notifier

A few systems notify people for various reasons. AWS has two similar notification services, Simple Email Service (SES), and Simple Notification Service (SNS) [2]. This is essentially an application of the Publish-Subscribe pattern [1]. 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.

3.1.8 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 managed by load balancing, and appear to be instances of the Dynamic Broker 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.

3.2 Pattern distribution

The following table shows the distribution of the patterns within the architectures. Since a pattern can appear more than once in an architecture, the table shows the number of instances. Where a pattern contains other patterns (see descriptions above), both are counted. Uncertain instances of the patterns are indicated in parentheses. The total frequency shows the number of architectures where the pattern appears, not the total number of appearances of the pattern. The total in parentheses shows the total if speculative appearances are included.

Table 2: Patterns found in architectures.

	Dynamic Broker	Content Broker	Advanced Repo	High Avail 3	Map-Reduce	Stream	Notify	App-Plat
WEB	2	1	2	1				
CM		1	1			(1)		
BCH	(1)		1					
FT	1		1	(1)				
LRG			1			(1)		
AD	2		1-2		1		(1)	
DIS	(1)		1					
FILE	1		1-2				1	
MEDI	2		1			1	(1)	
GAME	2	(1)	1	1			1	
LOG		(1)	1		1	(1)		
FNCE			1		1			
ECOM		1	2		1		1	
TIME	1		1		1	1		
IMGM								
SQLS	1	1	1					1
DRU	1-2	1		(1)				1
WRDP	1-2	1		(1)				1
VARH		1	1					(1)
CTX		1						1

MFOC	1-2							1
TIBC	1-2		1					1
Total Freq.	12 (14)	8 (10)	17	2 (5)	5	2	3 (5)	6 (7)

4 Pattern Descriptions

4.1 Dynamic Broker (aka Broker-Dynamic Load Balancer)

4.1.1 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 at least 99.99 percent availability.)

4.1.2 Problem

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

4.1.3 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.

4.1.4 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.

4.1.5 Differences from original

How is this different from the traditional Broker pattern? In the original Broker pattern, the broker component manages interaction with a set of servers. The number of servers may be fixed; see figure 1. Thus, the responsibility of the broker component may be somewhat simple.

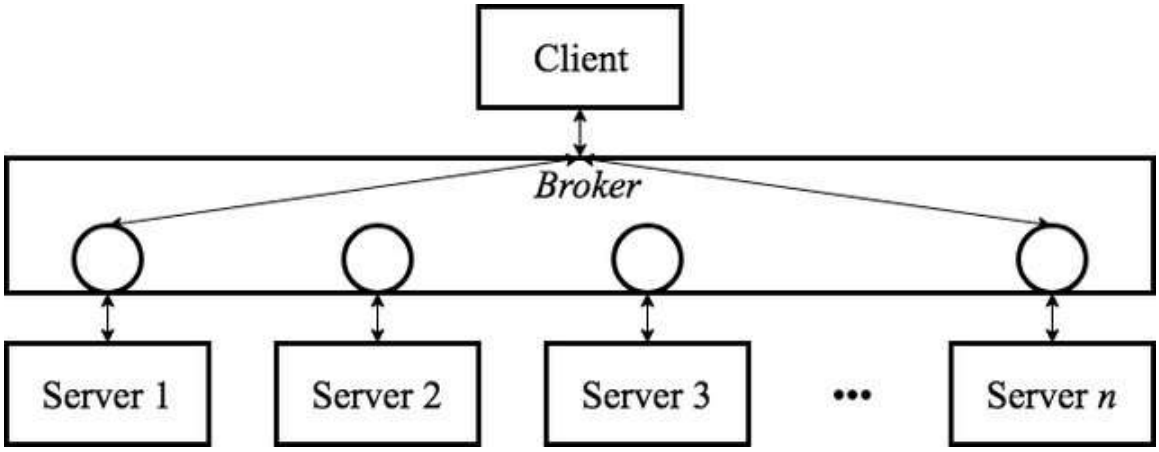

Figure 1: Original Broker Pattern [5].

The Dynamic Broker is a special case of the Broker pattern where the number of servers is dynamic, and not known beforehand. Here, the broker component must be sophisticated. It must monitor the load, and as the load warrants, do the following:

- Obtain additional servers. In cloud services, this involves requesting a server from a server management system. The AWS architectures do not show this interaction, as it is part of the "cloud", and isn't visible to the application developer.
- Release a server back to the cloud when it is no longer needed. This can be important if the application is billed on the basis of server usage.
- Ensure availability of services; this may be as simple as keeping two servers active at all times.

Figure 2 shows an architecture with the Dynamic Broker concerned with load balancing and an elastic set of servers. The diagram shows two "availability zones", as well as auto-scaling groups of servers. There is an "Internal Application Load Balancer" component, which also appears to be a Dynamic Broker.

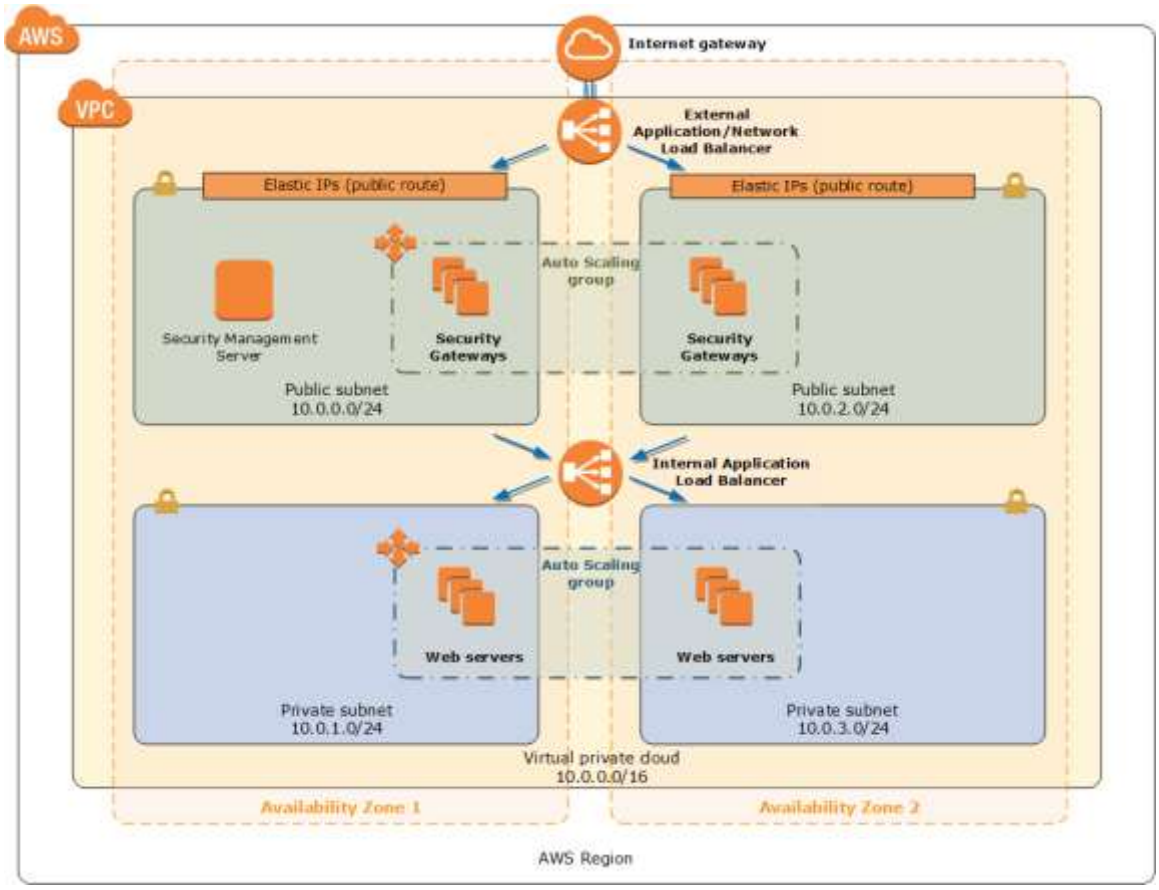

Figure 2: Dynamic Broker Pattern [2].

4.1.6 Consequences

The Dynamic Broker addresses the forces in this manner:

- Operating cost: 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 (simultaneous) targets allowed, the user can control the maximum cost of the system.
- Dynamic loads: Elastic scaling as described above explicitly handles dynamic loads.
- Scalability: As a user's business grows, the maximum number of targets used can be adjusted.
- 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.
- Availability: 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 should be specified to increase availability through redundancy.
- Development cost: Developers do not have to write the components provided by AWS, but do have to learn the associated technology and application programming interfaces (APIs). 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 this is not a benefit of the pattern by itself, but rather a benefit of the AWS components. It does, however, illustrate the reusability of the components of the Dynamic Broker pattern.

The Dynamic Broker pattern also has the following limitations and liabilities:

- Operating cost: high availability through redundancy of targets increases operating costs.
- Availability: if there is a maximum number of targets, and load exceeds the capacity of the targets, then end users will experience delays in service or even lack of availability of the service. One can avoid this by monitoring traffic and increasing maxima as needed.
- Availability: Although availability is increased through redundant targets, the load balancer itself is a single point of failure. The reference architectures studied did not show evidence of redundant brokers anywhere. It is unclear how or if AWS mitigates this risk.
- Availability and Scalability: The Dynamic Broker provides availability and scalability for this part of the system, but do not provide it for the entire system.

Note that the AWS Dynamic Broker components are not to be used by themselves. They are to be used with other AWS components, such as components in the Advanced Repository, 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.

4.2 Advanced Repository

4.2.1 Context

Data and the insights from data have become central to modern web-based applications. It is now possible to store large amounts of data and analyze them.

4.2.2 Problem

In modern web applications, data has become central. While it is necessary to both store and analyze large amounts of data, management, analysis, and manipulation of distributed data is extremely complex, and requires specialized expertise to implement.

4.2.3 Forces

Distributed 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.
- Intelligent data management of large data is necessary, but requires specialized expertise in areas of data science.

4.2.4 Solution

Build a repository as a component in the architecture. Implement additional capabilities above a simple repository. Provide access to the repository and advanced features as services, such that the user need not be concerned with the implementation of the features.

The Advanced Repository may have 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, and should be transparent to users.
- It guarantees reliability. This may be done through redundancy and automated backups.
- 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.
- Data mining and transformation features may be provided.

An Advanced Repository may be shared, and usually is.

An Advanced Repository may also be active (see Active Repository in [ZDUN]).

The original Shared Repository is shown in figure 3:

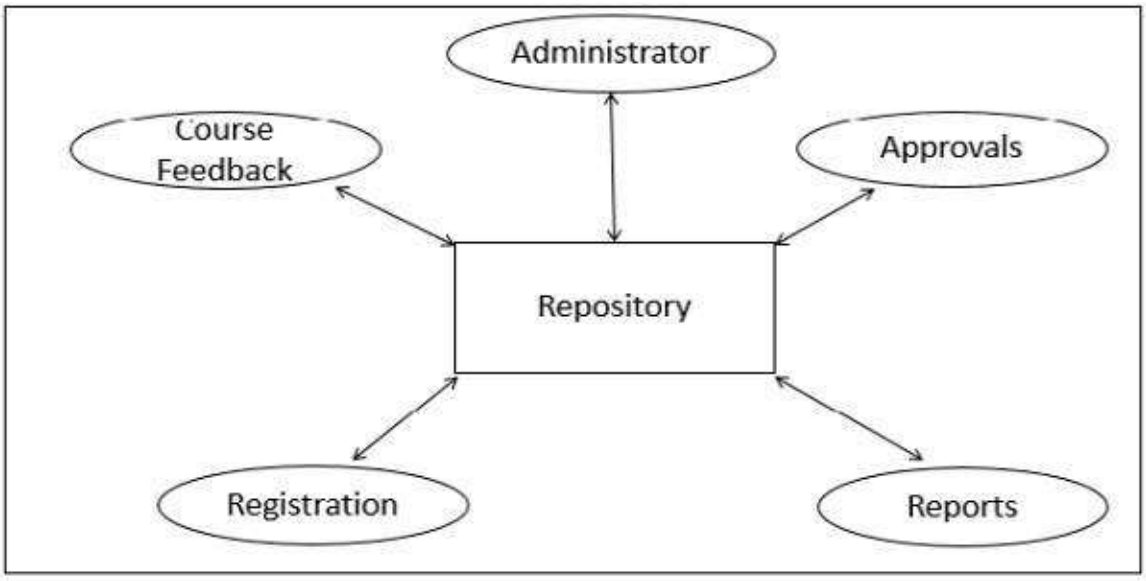

Figure 3: Shared Repository Example [5].

An Advanced Repository provides numerous features for managing, searching, and using data. Figure 4 below shows a reference architecture for a data lake using AWS components. Numerous services may be provided behind an API. Authorization is provided by an AWS Lambda component. The DynamoDB is the database itself. The CloudWatch component monitors the health of the data system, and creates logs. The AWS Glue component simplifies and automates data discovery, conversion, mapping, and job scheduling tasks. It crawls raw data in the buckets and suggests transformations.

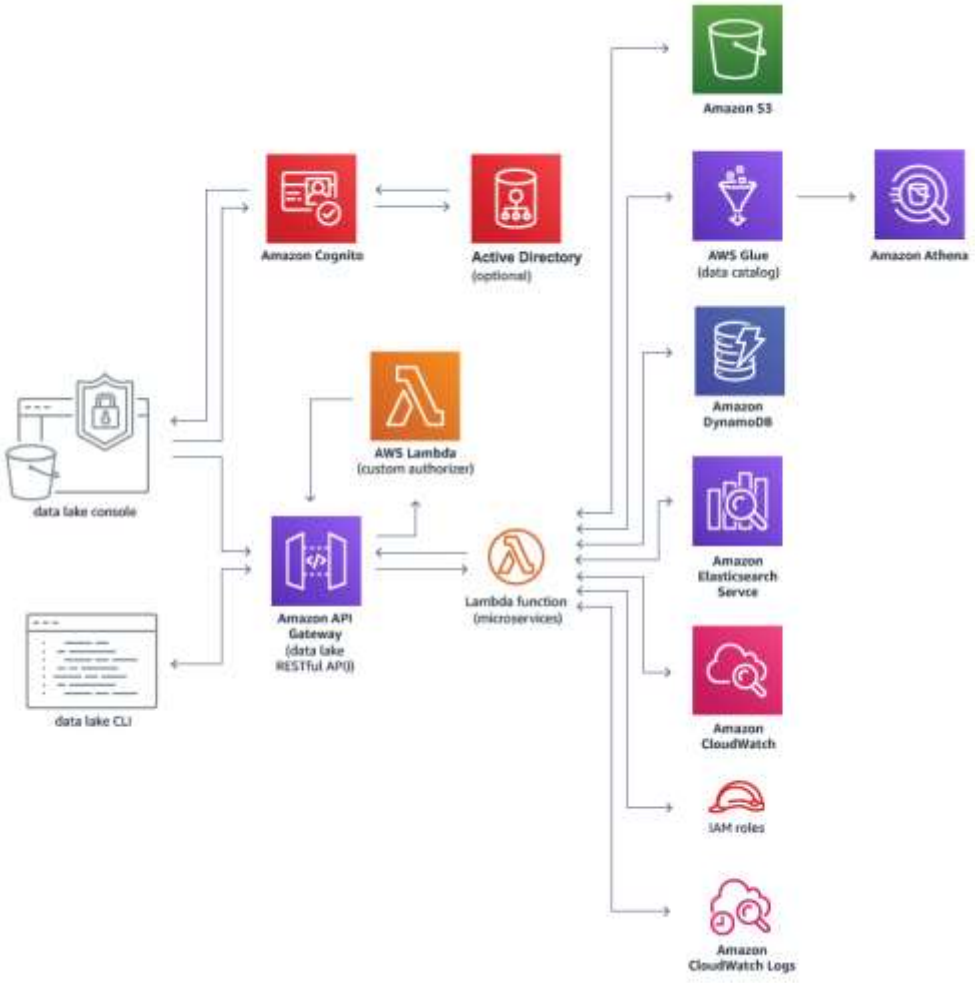

Figure 4: Data Lake architecture in AWS [2].

The application in Figure 5 shows an Advanced Repository with other capabilities. This shows a repository with active-standby for availability, automatic scaling, backup for reliability, and security.

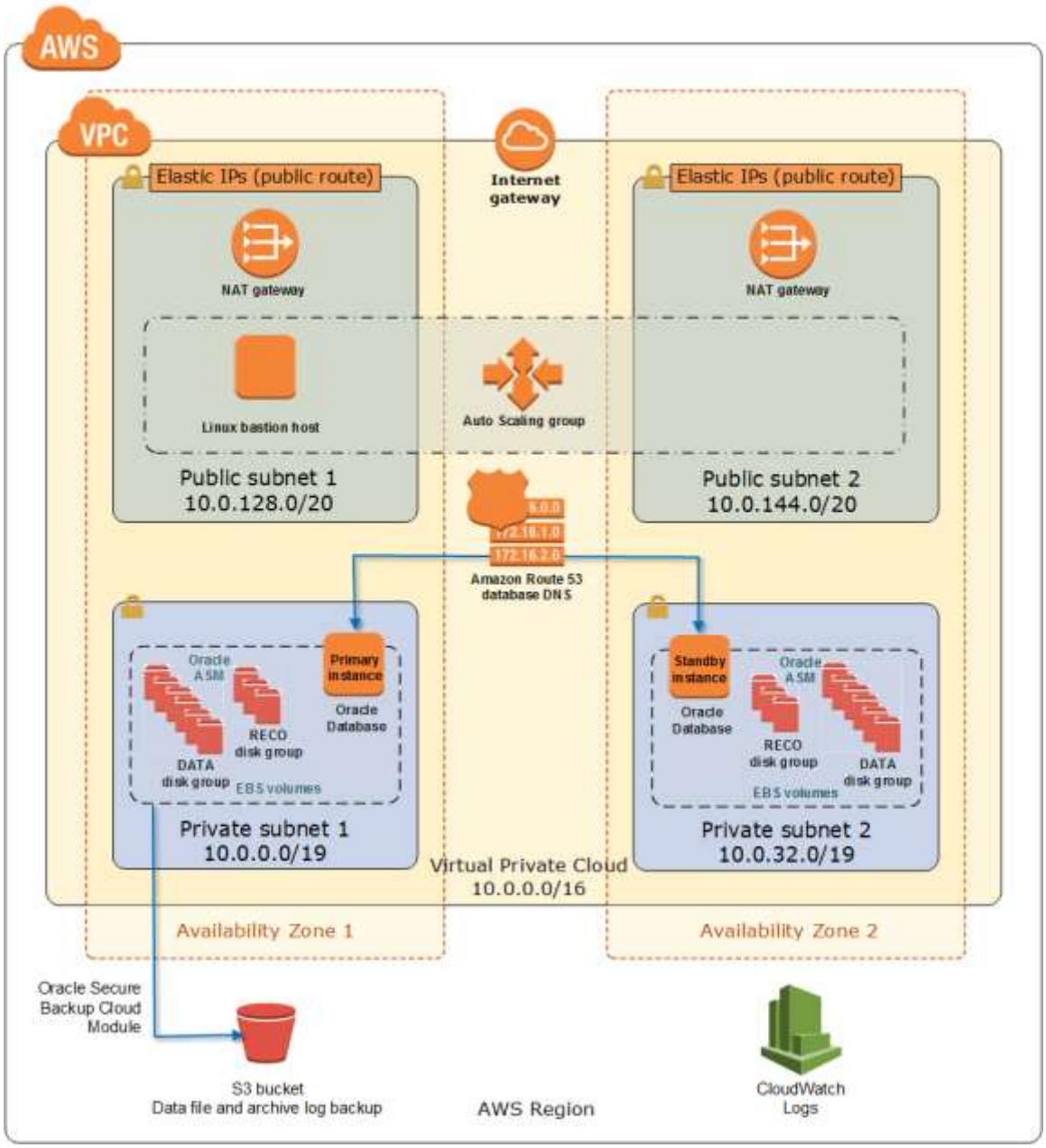

Figure 5: Architecture of a database system in AWS [2].

4.2.5 Consequences

The chief benefits of an Advanced Repository is that features needed in addition to a basic database are provided behind an interface, and that the application developer does not have to implement the features. Specifically, the Advanced Repository addresses the forces as follows:

- Deployment: the repository is deployed on a cloud service such as AWS.
- Operating cost: the application developer does not incur expenses associated with a dedicated server, but instead pays for database services hosted by the cloud service provider.

- Scalability: One of the options provided by Advanced Repositories is scalability.
- Availability: Availability may be provided, and will be transparent to the application.
- Reliability: Cloud databases guarantee reliable storage of data.
- Security: Cloud database providers also provide security of data.

4.3 Content Broker

4.3.1 Context

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.

4.3.2 Problem

A single entry point for clients may go to different applications, based on the type of request.

4.3.3 Forces

Persistent storage raises the following forces:

- Different actions should be done 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.
- Security of transactions is important.

4.3.4 Solution

Use a Broker that is a single entry point to process and distribute multiple different types of requests. It is typically in front of one or more traditional Broker, which handle load balancing and distribution to homogeneous servers. The Content 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.

The key difference between the Content Broker and a traditional Broker is that the traditional Broker is concerned with load distribution and reliability, and distributes to homogeneous servers. Any server can handle any type of message. Thus, a traditional Broker can be somewhat agnostic of application type. On the other hand, the Content Broker routes requests based on the type of request. It routes to different types of services, and must know the role of each. If load balancing or similar capabilities are also needed, traditional Brokers may be installed behind the Content Broker.

Figure 6 shows a Content Broker that routes different types of data and requests to the proper handlers, including anomaly detection, trip data, driver safety score, diagnostic trouble codes, location-based marketing, and registration.

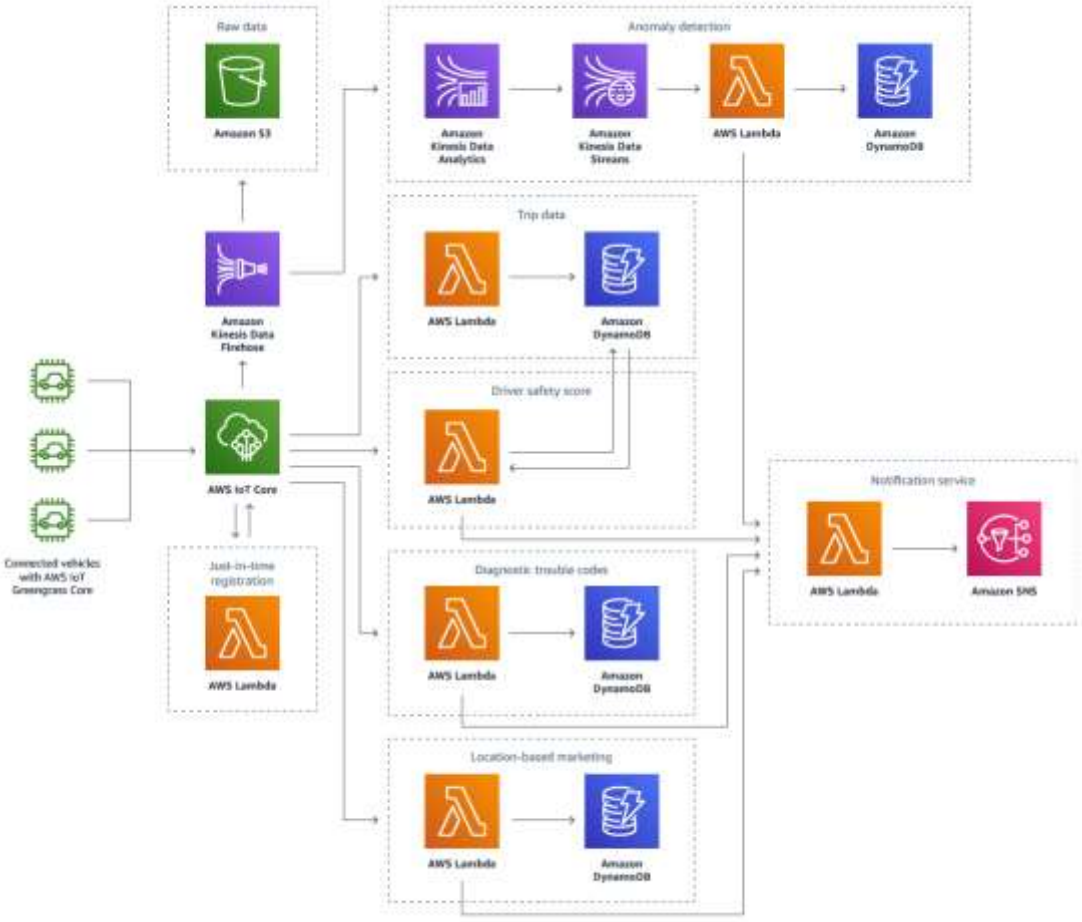

Figure 6: Architecture of a connected vehicle in AWS [2].

Figure 7 shows a Content Broker (AWS WAF) as a security front end. It provides protection against various types of attacks as shown. Requests are not only routed to a traditional Broker (CloudFront plus Application Load Balancer), but log data is sent to an analysis engine for updating the security policies of the Content Broker.

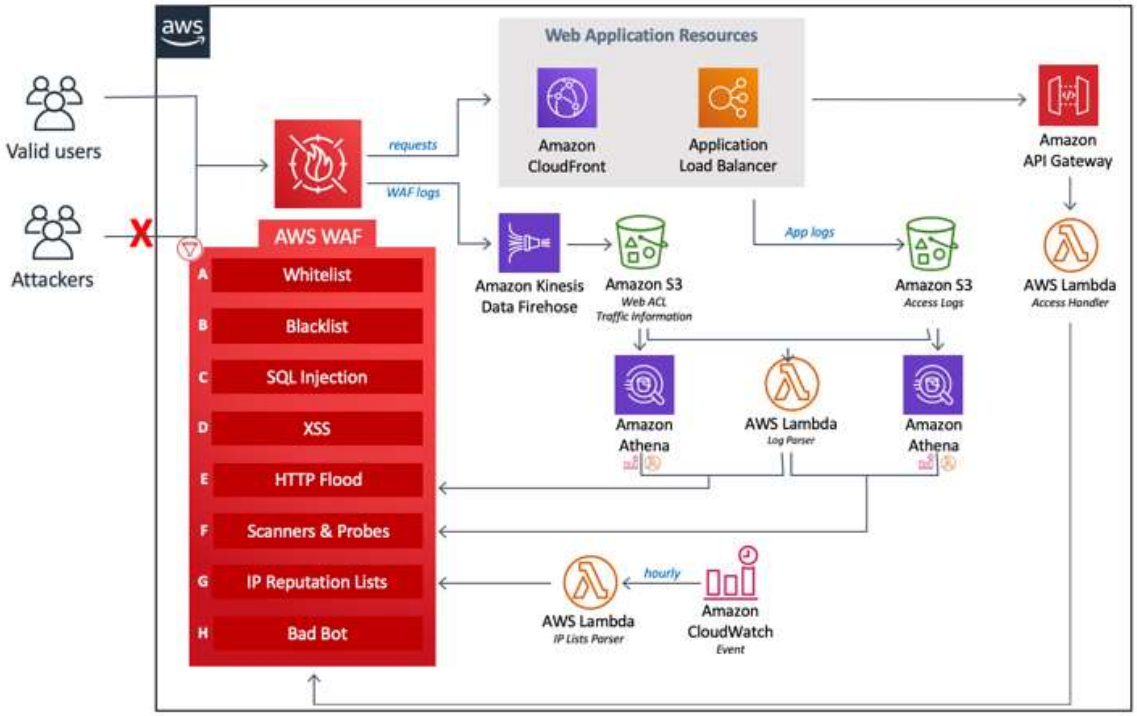

Figure 7: Architecture of a security front end in AWS [2].

4.3.5 Consequences

The Broker provides 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 computing 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.

5 Conclusion

With these three patterns, we can see the foundations of the original architecture patterns, augmented by additional structures that provide additional functionality. It is notable that much of the functionality is in support of various quality attributes, especially security and scalability. Because of this significant contribution, it may be useful to consider these as independent patterns.

There is much room for additional study. Other web services platforms can be studied to determine the generality of these patterns, and to search for other patterns. One might study who these patterns might be applied outside of platforms. The remaining patterns should be written in detail.

Much work has been done with regard to architecture patterns, their interactions with quality attributes, and the tactics used to implement them [7, 8, 9, 10, 11]. The web services platforms implement the tactics in reusable architectures. Further study of the patterns, quality attributes, and tactics in web services architectures will be valuable.

ACKNOWLEDGMENTS

Thanks to Stefan Soberning, who gave valuable comments during shepherding. Thanks also to the writers' workshop at PLoP 2019 for their insights and comments.

REFERENCES

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996). Pattern-Oriented Software Architecture: A System of Patterns. Wiley, Chichester, England.
[2] Amazon Web Services Architecture Homepage, https://aws.amazon.com/architecture, accessed 2019/11/14.
[3] Microsoft Azure Homepage, https://azure.microsoft.com/en-us, accessed 2019/11/11.
[4] Google Cloud Homepage, https://cloud.google.com, accessed 2019/11/11.
[5] P. Avgeriou and U. Zdun (2005). Architectural Patterns Revisited – a Pattern Language, 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005), Irsee, Germany.
[6] N. Harrison and P. Avgeriou (2008). Incorporating Fault Tolerance Techniques in Software Architecture Patterns. In: Proceedings of the International Workshop on Software Engineering for Resilient Systems (SERENE '08), Newcastle upon Tyne (UK), ACM Press.
[7] F. Bachmann, L. Bass, M. Klein, and C. Shelton (2005). Designing software architectures to achieve quality attribute requirements. IEE Proceedings, vol. 152.
[8] N. Harrison and P. Avgeriou (2010). How do Architecture Patterns and Tactics Interact? A Model and Annotation. Journal of Systems and Software, 83 (10), pp. 1735-1758.
[9] N. Harrison, and P. Avgeriou (2007). Leveraging Architecture Patterns to Satisfy Quality Attributes. Proceedings of First European Conference on Software Architecture (ECSA), Madrid, LNCS.
[10] L. Bass, P. Clements, and R. Kazman (2003). Software Architecture in Practice 2nd ed. Addison Wesley, Reading, MA, USA.
[11] N. Harrison, P. Avgeriou, and U. Zdun. (2010). On the Impact of Fault Tolerance Tactics on Architecture Patterns. Proceedings of the International Workshop on Software Engineering for Resilient Systems (SERENE '10), ACM Press.