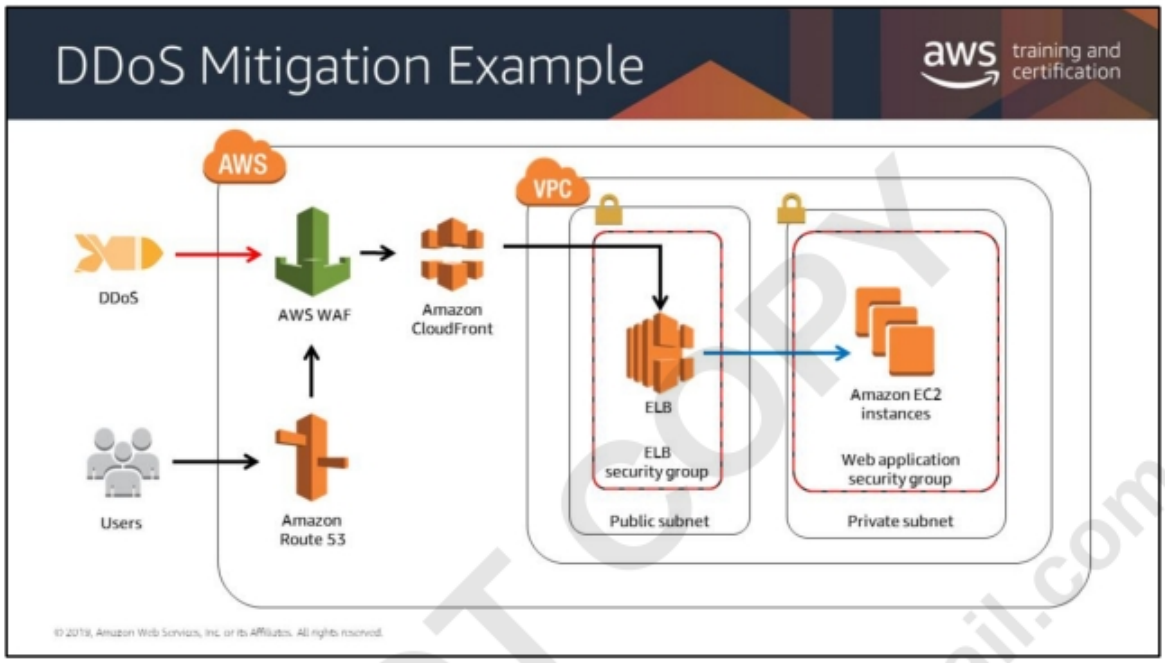


This is an example of a resilient architecture that can help to prevent or mitigate DDoS attacks.

The strategy to minimize the attack surface area is to (a) reduce the number of necessary Internet entry points, (b) eliminate non-critical Internet entry points, (c) separate end user traffic from management traffic, (d) obfuscate necessary Internet entry points to the level that untrusted end users cannot access them, and (e) decouple Internet entry points to minimize the effects of attacks. This strategy can be accomplished with Amazon Virtual Private Cloud.

Within AWS, you can take advantage of two forms of scaling: horizontal and vertical scaling. In terms of DDoS, there are three ways to take advantage of scaling in AWS: (1) select the appropriate instance types for your application, (2) configure services such as Elastic Load Balancing and Auto Scaling to automatically scale, and (3) use the inherent scale built into the AWS global services like Amazon CloudFront and Amazon Route 53.

Because ELB only supports valid TCP requests, DDoS attacks such as UDP and SYN floods are not able to reach your instances.
You can set a condition to incrementally add new instances to the Auto Scaling group when network traffic is high (typical of DDoS attacks).

Services that are available within AWS regions, like Elastic Load Balancing and Amazon EC2, allow you to build DDoS resiliency and scale to handle unexpected volumes of traffic within a given region. Services that are available in AWS edge locations, like Amazon CloudFront, AWS WAF, Amazon Route 53, and Amazon API Gateway, allow you to take advantage of a global network of edge locations that can provide your application with greater fault tolerance and increased scale for managing larger volumes of traffic.

The benefits of using each these services to build resiliency against infrastructure layer and application layer DDoS attacks are discussed in the following sections.

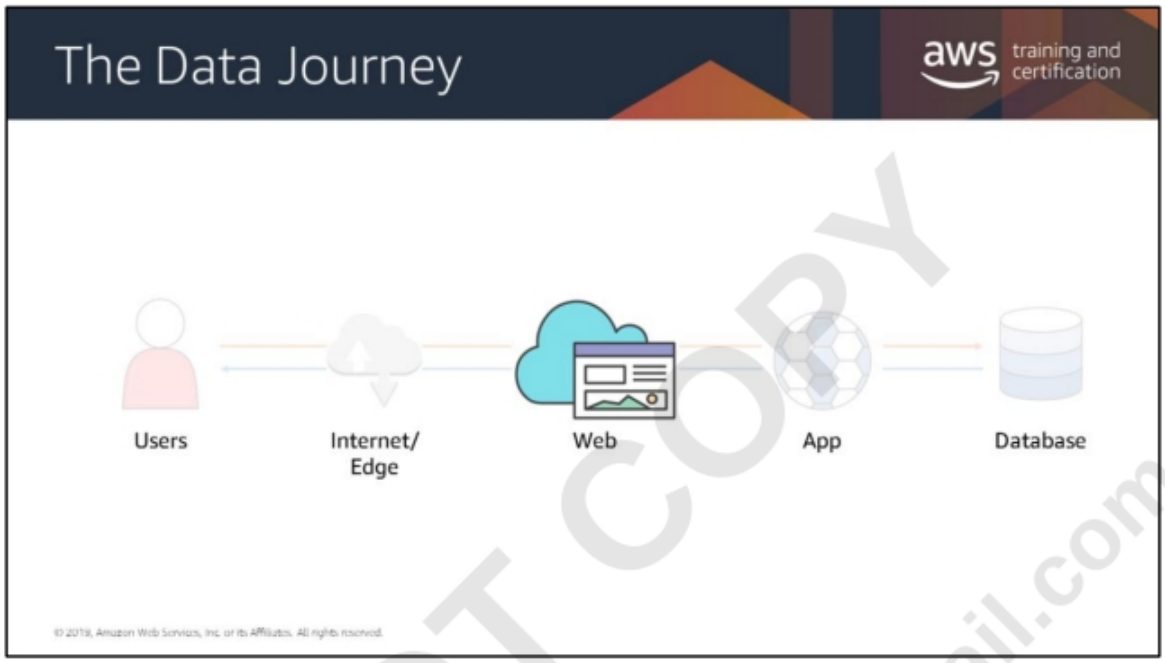
Amazon CloudFront also has filtering capabilities to ensure that only valid TCP connections and HTTP requests are made while dropping invalid requests.

A WAF (web application firewall) is a tool that applies a set of rules to HTTP traffic, in order to filter web requests based on data such as IP addresses, HTTP headers, HTTP body, or URI strings. They can be useful for mitigating DDoS attacks by offloading illegitimate traffic.

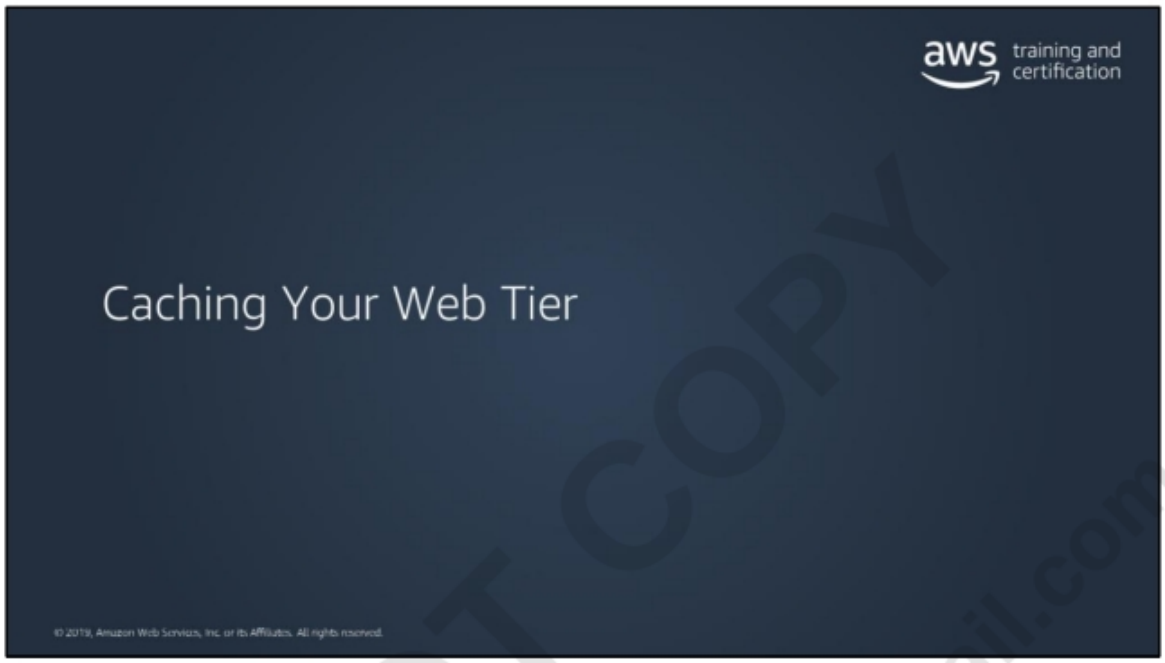
AWS now offers a managed WAF service. For more on AWS WAF, see:
http://docs.aws.amazon.com/waf/latest/developerguide/what-is-aws-waf.html

Whitepaper: AWS Best Practices for DDoS Resiliency:
https://d0.awsstatic.com/whitepapers/Security/DDoS_White_Paper.pdf

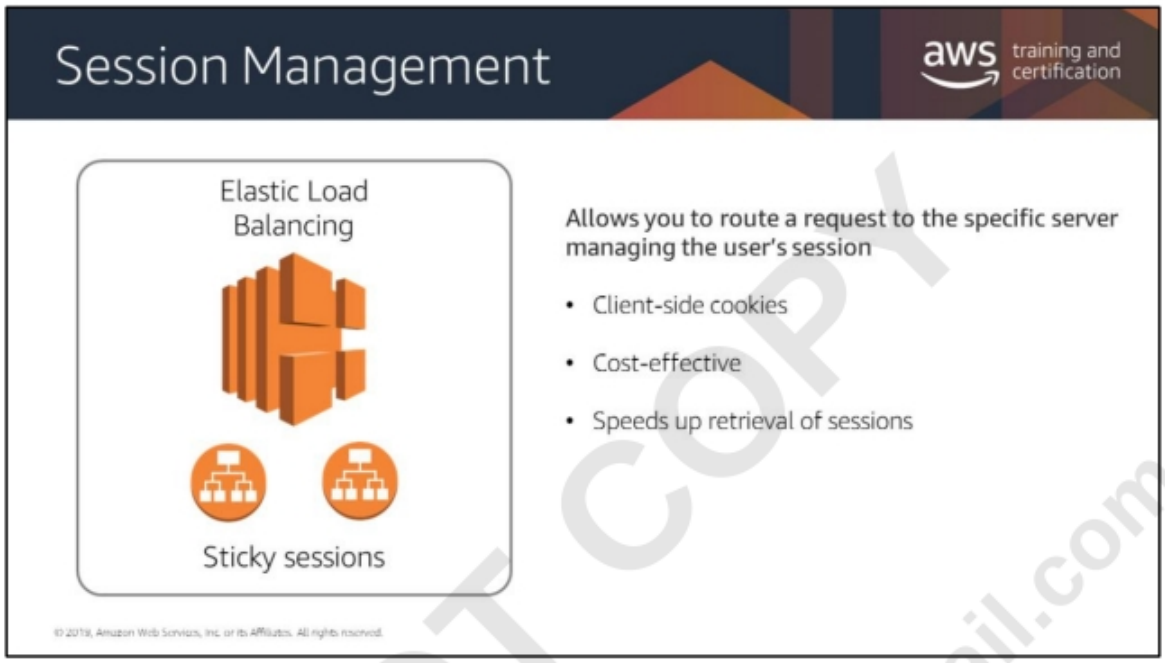
/



/



/



It might not seem as if session management is related to caching but it is.

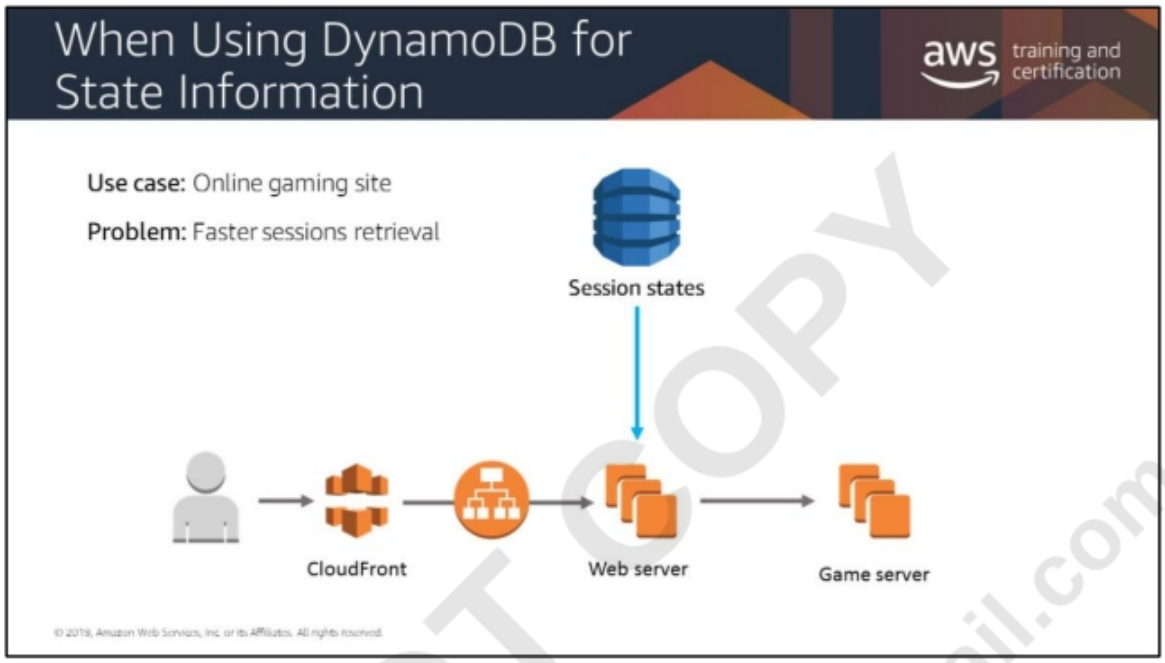
Basically, a web session is a sequence of HTTP transactions from the same user into an environment. HTTP was designed to transfer documents. It does not manage state. Every request is independent of previous transactions. Do you really want people to send credentials for every request made to your server? Does your server have the network and compute power needed to scale as users and their requirements increase?

Session management is authentication and access control. Common approaches to managing state include using sticky sessions or a distributed cache.

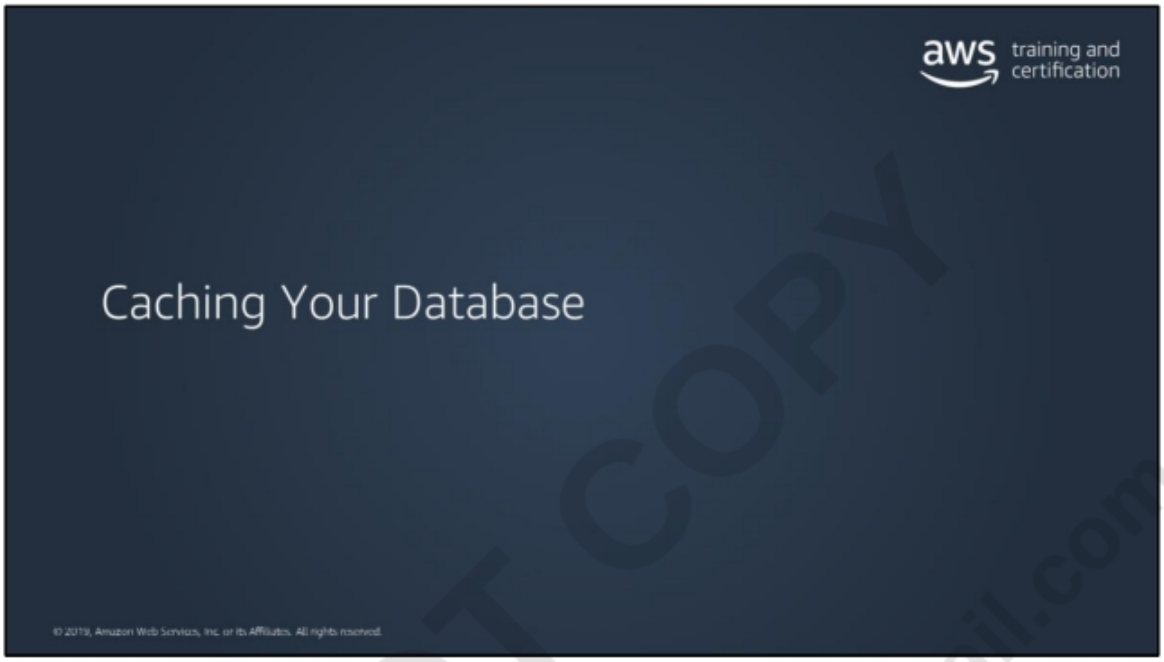
Sticky sessions, also called *session affinity*, allow you to route a request to the specific server managing the user's session. A session's validity can be determined by a number of methods; client-side cookies or parameters set at a load balancer.

Sticky sessions are cost-effective because you are storing sessions on the web servers running your applications. This eliminates network latency and speeds up retrieval of those sessions. However, in the event of failure, you are likely to lose the sessions stored on a node.

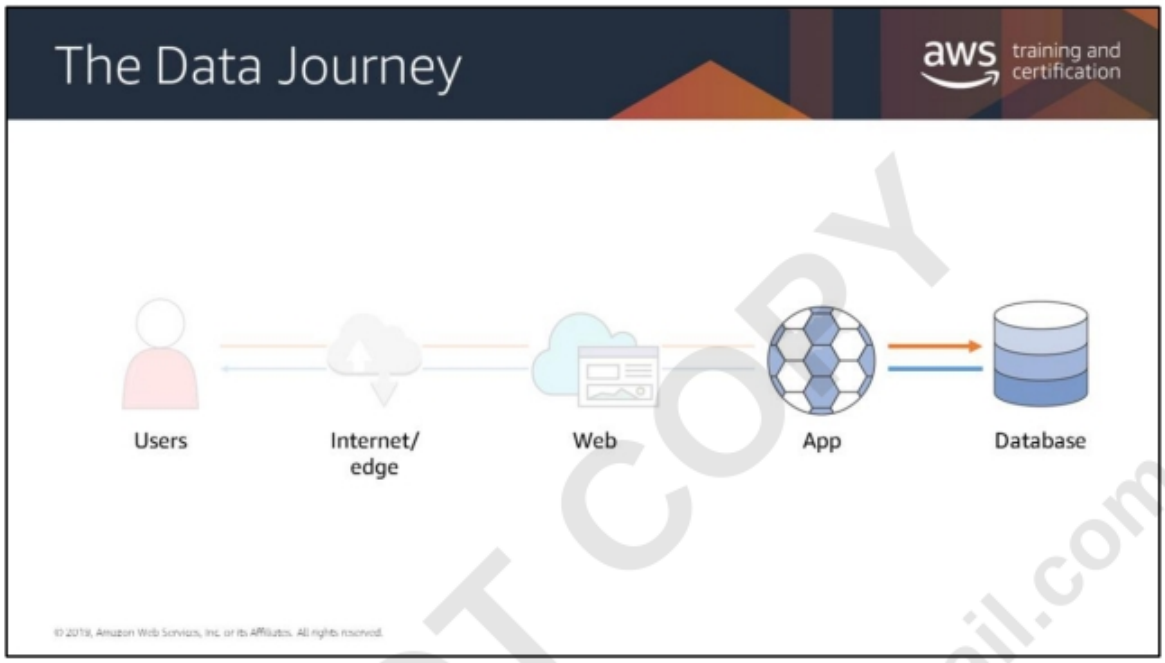
When scaling, it's possible traffic might be unequally spread across servers as active sessions prevent routing traffic to the increased capacity.

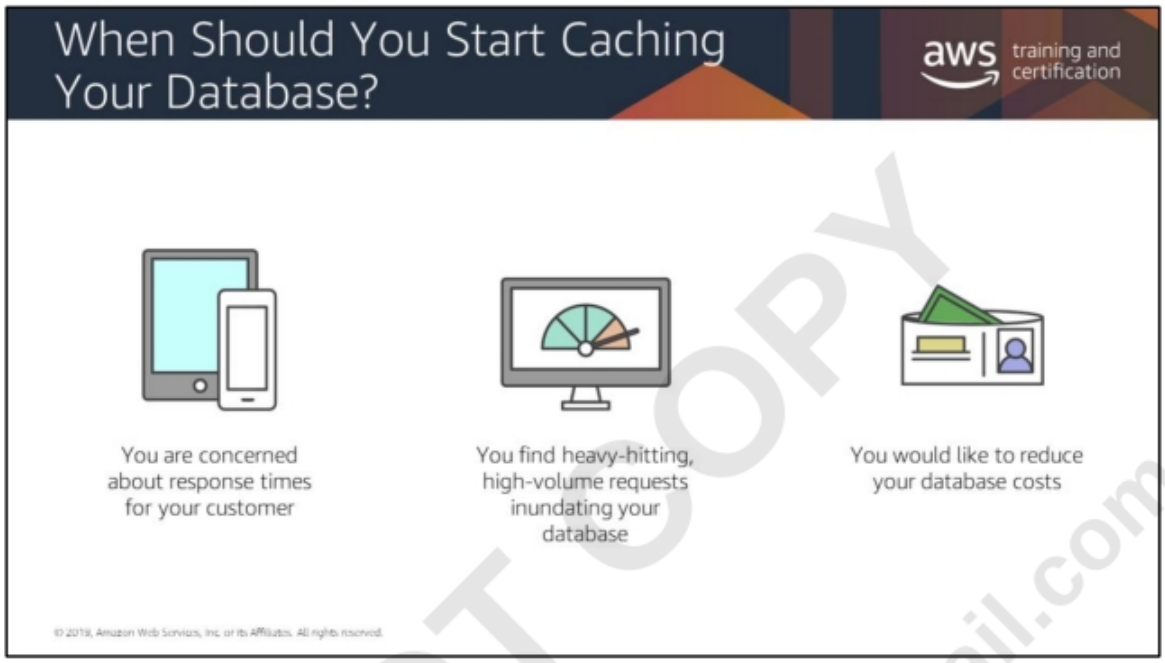


/

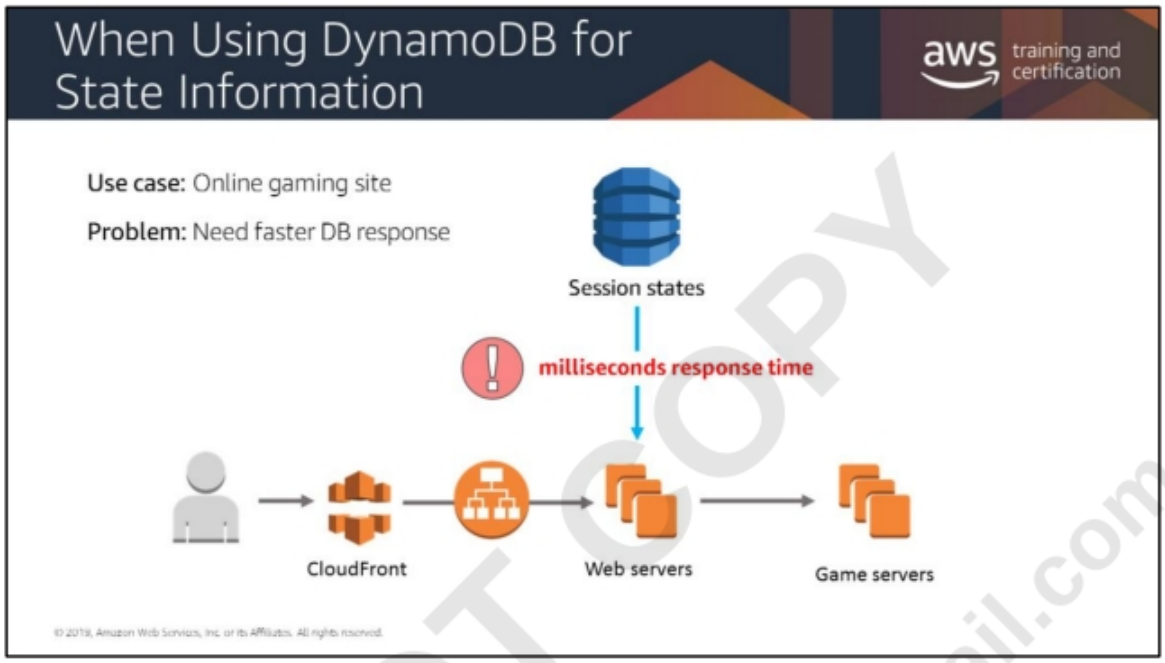


/

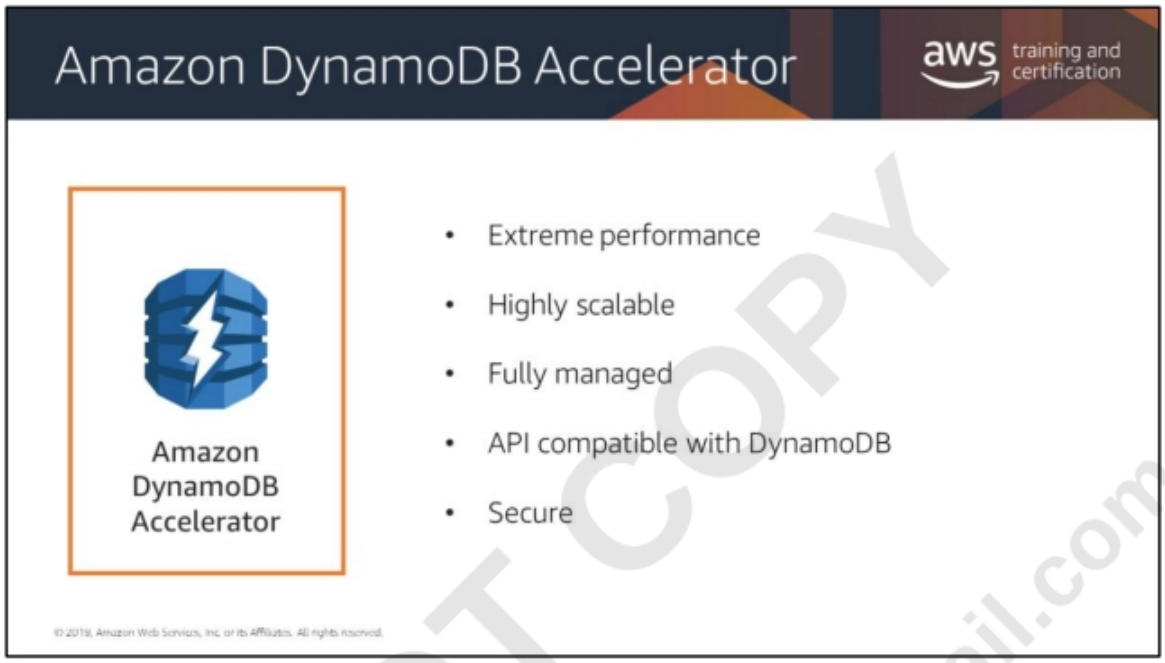




/



/



Extreme Performance
DynamoDB offers consistent single-digit millisecond latency. DynamoDB plus DAX have response times in microseconds for millions of requests per second for read-heavy workloads.

Highly Scalable
DAX has on-demand scaling. You can start with a three-node DAX cluster and add capacity by as required up to a ten-node cluster.

Fully Managed
Like DynamoDB, DAX is fully managed. DAX takes care of management tasks including provisioning, setup and configuration, software patching, and replicating data over nodes during scaling operations. DAX automates common administrative tasks such as failure detection, failure recovery, and software patching.

API Compatible with DynamoDB
DAX is API-compatible with DynamoDB and there is no need to make any functional application code changes. Provision a DAX cluster, use the DAX client SDK to point existing DynamoDB API calls at the DAX cluster, and DAX handles the rest.

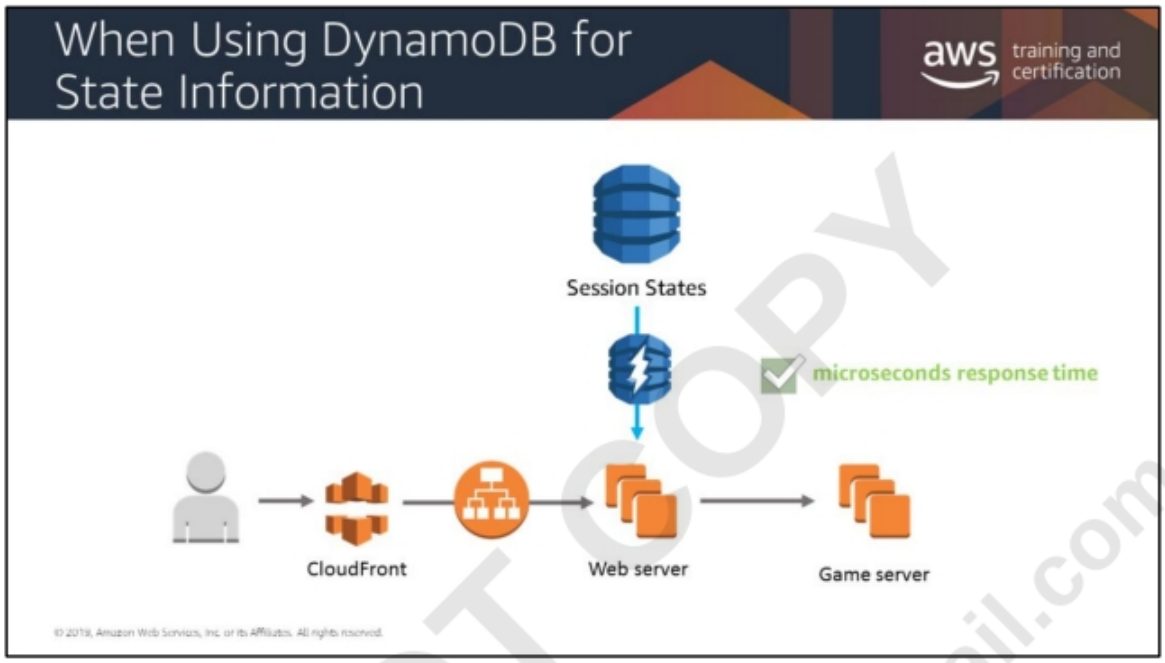
Flexible
You can provision one DAX cluster for multiple DynamoDB tables, multiple DAX clusters for a single DynamoDB table or a combination of both.

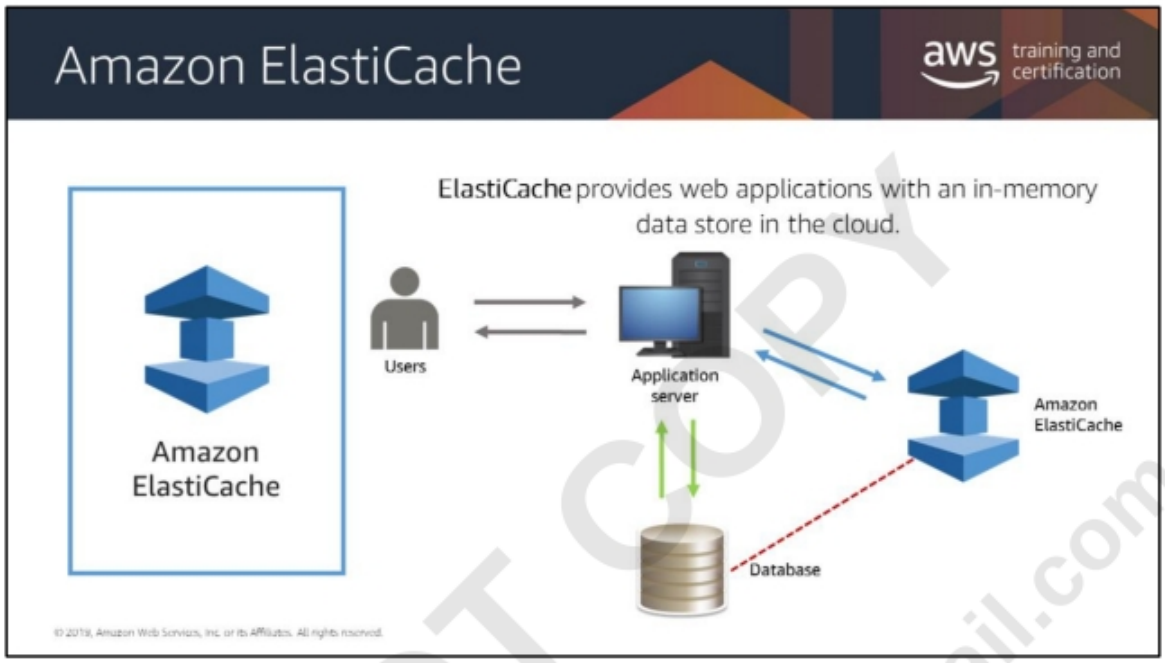
Secure
DAX fully integrates with AWS services to enhance security. You can use AWS Identity and Access Management (IAM) to assign unique security credentials to each user and control each user's access to services and resources. Amazon CloudWatch enables you to gain system-wide visibility into resource utilization, application performance, and operational health. Integration with AWS CloudTrail enables you to easily log and audit changes to your cluster configuration. DAX supports Amazon Virtual Private Cloud (VPC) for secure and easy access from your existing applications. Tagging provides you additional visibility to help you manage your DAX clusters.

The retrieval of cached data reduces the read load on existing DynamoDB tables. This means it may reduce provisioned read capacity and lower overall operational costs.

For more information see: https://aws.amazon.com/blogs/database/amazon-dynamodb-accelerator-dax-a-read-throughwrite-through-cache-for-dynamodb/

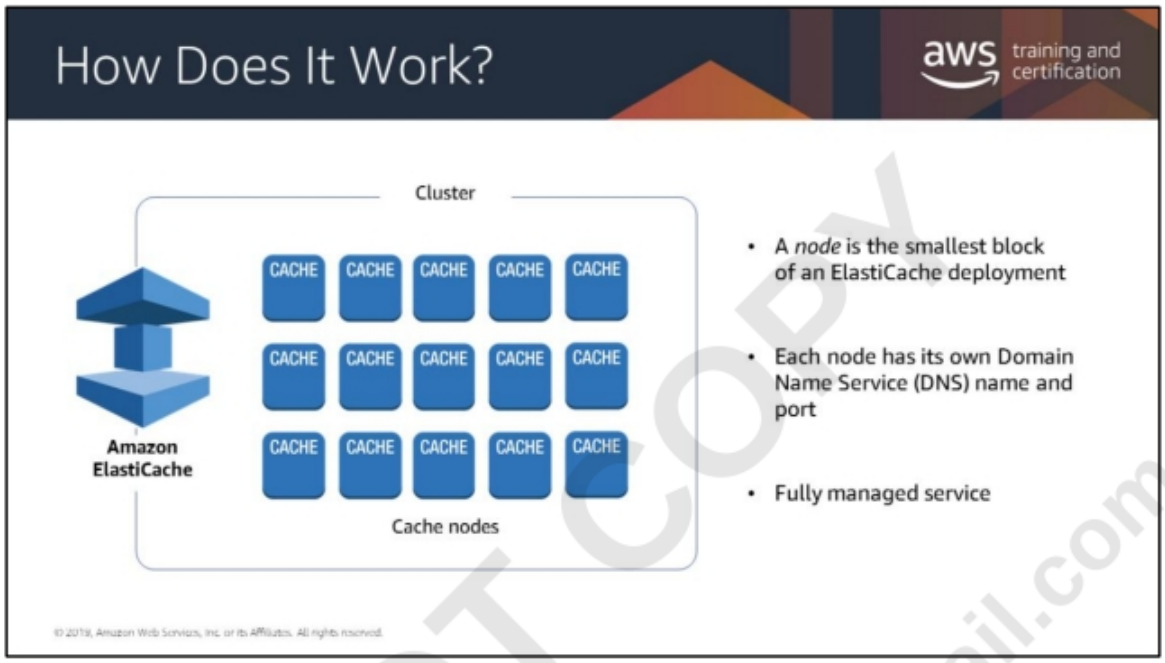
/





Amazon ElastiCache is a web service used to deploy, operate, and scale an in-memory cache in the cloud. ElastiCache improves the performance of web applications by allowing you to retrieve information from fast, managed, in-memory data stores, instead of relying on slower disk-based databases. Whenever possible, applications will retrieve data from ElastiCache and defer to databases when data isn't found in cache.

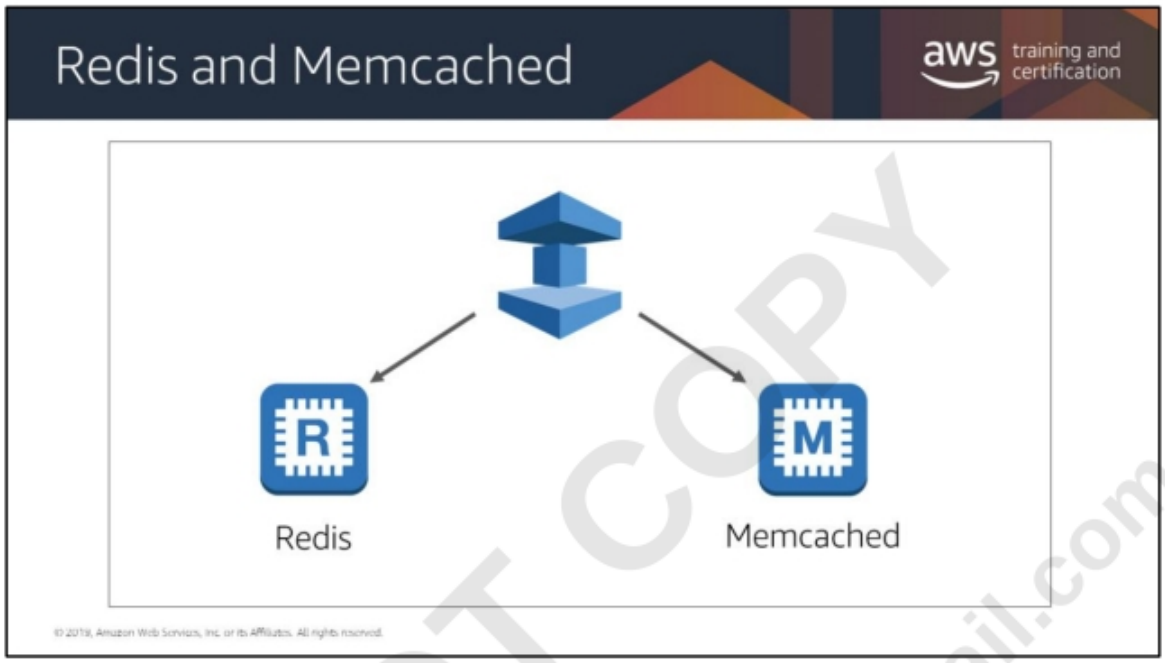
/



A *cache node* is the smallest building block of an ElastiCache deployment. It can exist in isolation from other nodes, or in some relationship to other nodes, which is also known as a *cluster*.

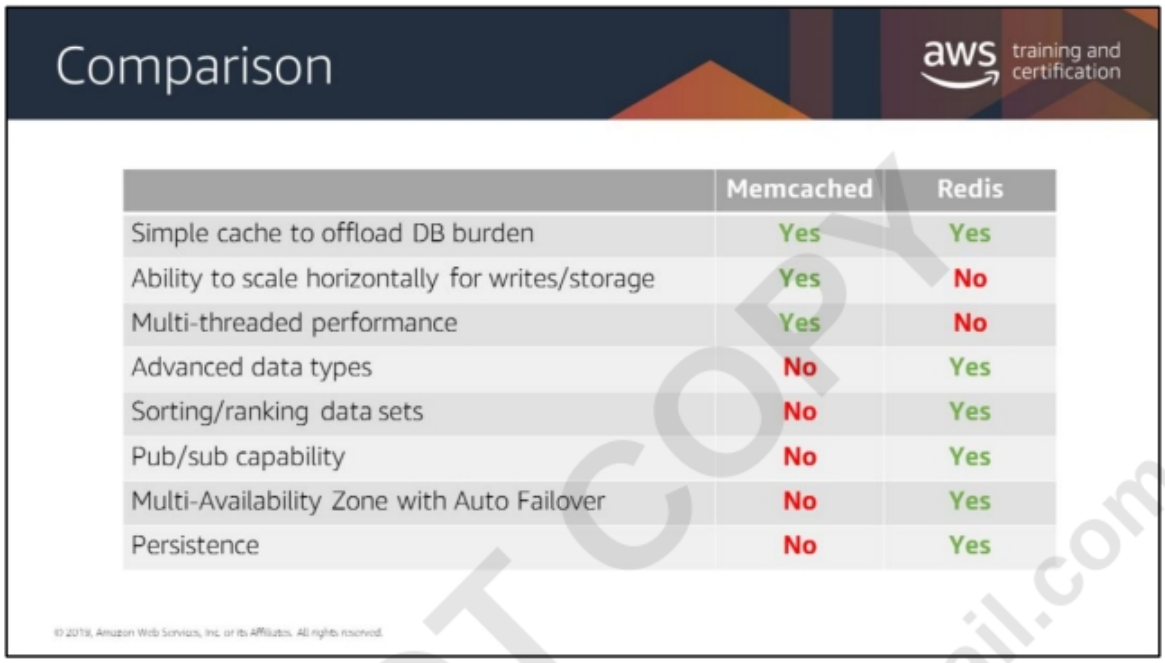
Amazon ElastiCache is fully managed, so you're not stuck running cache nodes you're not using, and if you need more, your cluster can scale out to accommodate your capacity needs.

/

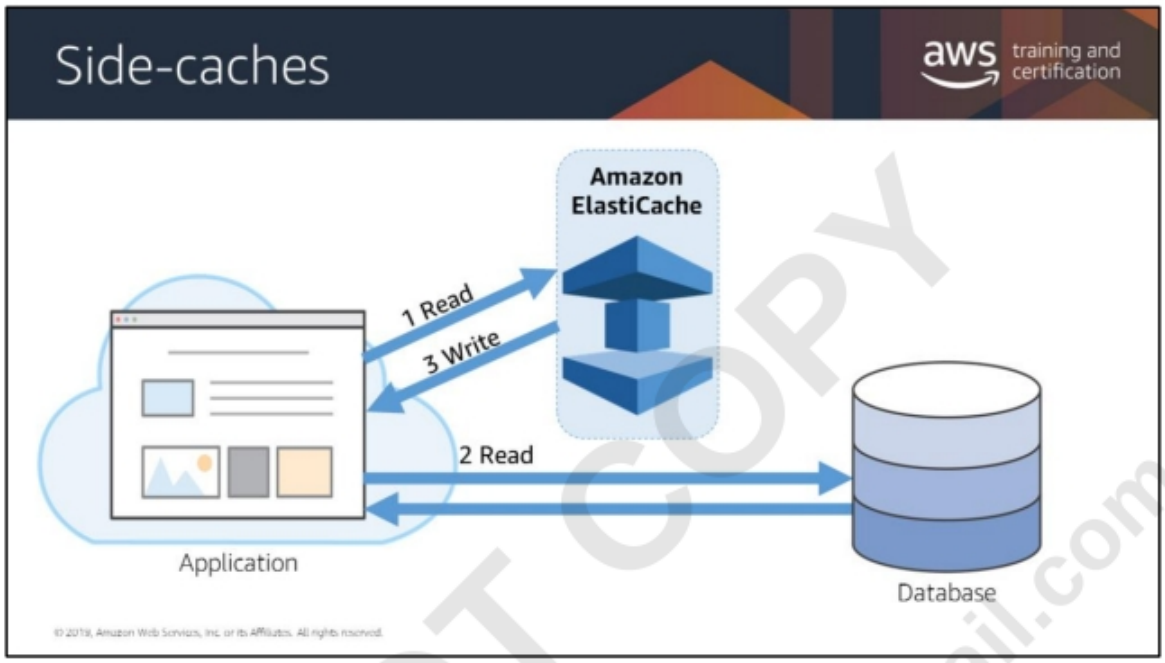


ElastiCache with Memcached can scale up to 20 nodes per cluster, while ElastiCache for Redis can scale up to 90 nodes for increased data access performance. ElastiCache supports Amazon VPC, enabling you to isolate your cluster to the IP ranges you choose for your nodes.

ElastiCache runs on the same highly reliable infrastructure used by other AWS services. For Redis workloads, ElastiCache provides high-availability through Multi-AZ with automatic failover. For Memcached workloads, data is partitioned across all nodes in the cluster, allowing you to scale out to better handle more data when demand grows. With ElastiCache, you no longer need to perform management tasks such as hardware provisioning, software patching, monitoring, failure recovery, and backups. ElastiCache continuously monitors your clusters to keep your workloads up and running so that you can focus on application development.



/

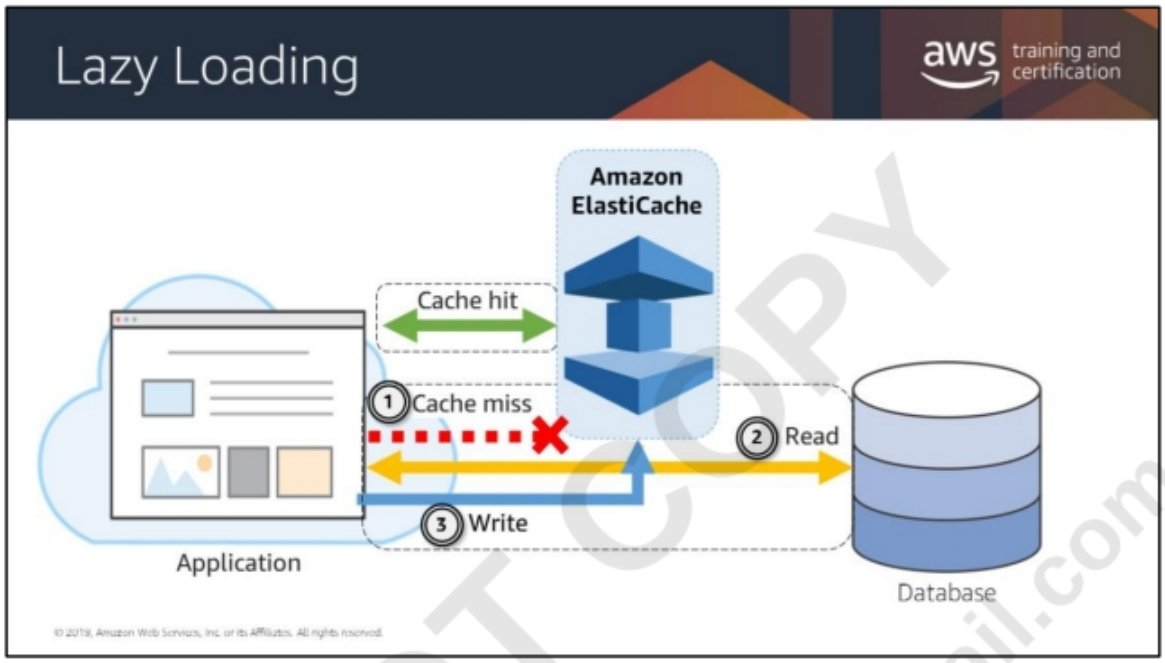


When you're using a cache for a backend data store, a side-cache is perhaps the most commonly known approach. Canonical examples include both Redis and Memcached. These are general-purpose caches that are decoupled from the underlying data store and can help with both read and write throughput, depending on the workload and durability requirements.

For read-heavy workloads, side-caches are typically used as follows:

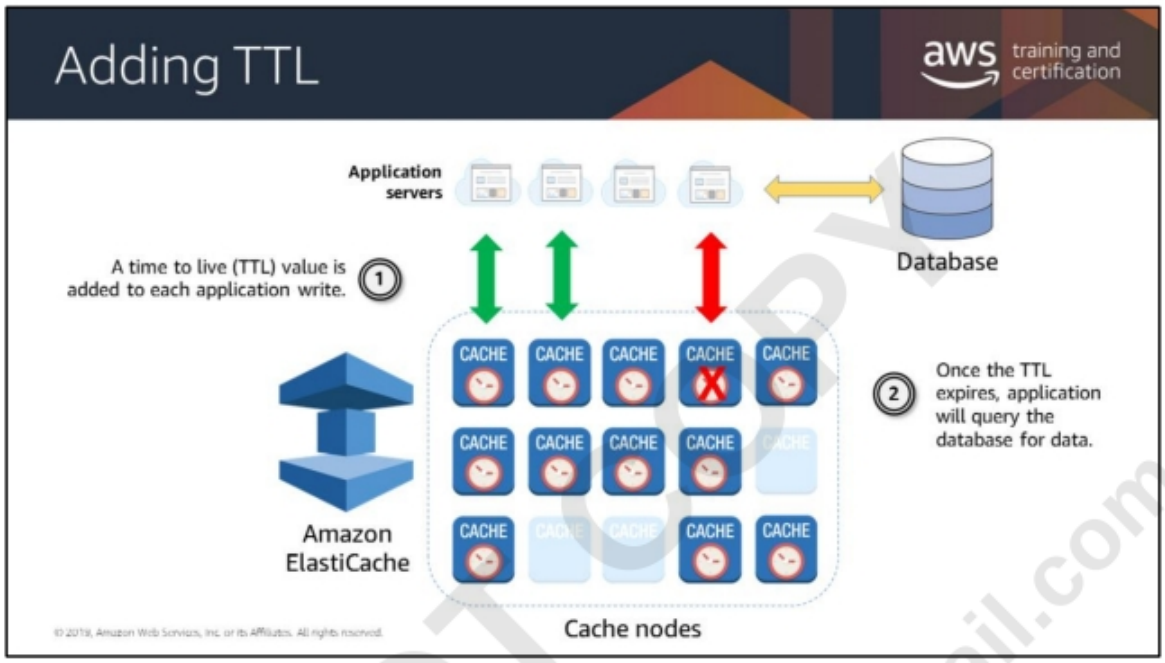
1. For a given key-value pair, the application first tries to read the data from the cache. If the cache is populated with the data (cache hit), the value is returned. If not, on to step 2.
2. Because the desired key-value pair was not found in the cache, the application then fetches the data from the underlying data store.
3. To ensure that the data is present when the application needs to fetch the data again, the key-value pair from step 2 is then written to the cache.

For more information, see: https://aws.amazon.com/blogs/database/amazon-dynamodb-accelerator-dax-a-read-throughwrite-through-cache-for-dynamodb/

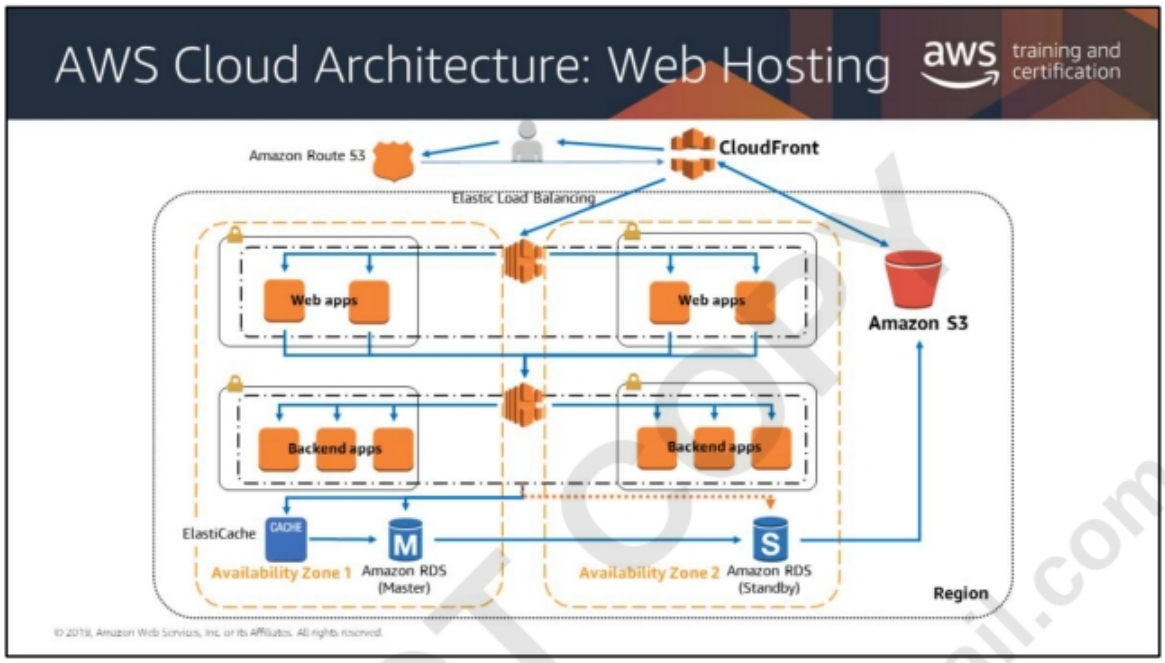


Lazy loading is a caching strategy that loads data into the cache only when necessary. In this deployment, ElastiCache sits between your application and the data store, or database, that it accesses. Whenever your application requests data, it first makes the request to the ElastiCache cache. If the data exists in the cache and is current, a cache hit occurs and ElastiCache returns the data to your application. Otherwise, your application requests the data from your data store which returns the data to your application. Your application then writes the data received to the cache so it can be retrieved more quickly the next time it is requested.

With lazy loading, only requested data is cached. Since most data is never requested, lazy loading avoids filling up the cache with unnecessary data. However, there is a cache miss penalty. Each cache miss results in three trips, which can cause a noticeable delay in data getting to the application. Also, if data is only written to the cache when there is a cache miss, data in the cache can become stale since there are no updates to the cache when data is changed in the database. The Write Through and Adding TTL strategies address this issue, and we'll cover those next.



Lazy loading allows for stale data, while Write Through ensures that data is always fresh, but may populate the cache with unnecessary data. By adding a time to live, or TTL, value to each write, you can enjoy the advantages of each strategy and largely avoid cluttering up the cache with data. TTL is an integer value or key that specifies the number of seconds or milliseconds, depending on the in-memory engine, until the key expires. When an application attempts to read an expired key, it is treated as though the data is not found in cache, meaning that the database is queried and the cache is updated. This keeps data from getting too stale and requires that values in the cache are occasionally refreshed from the database.

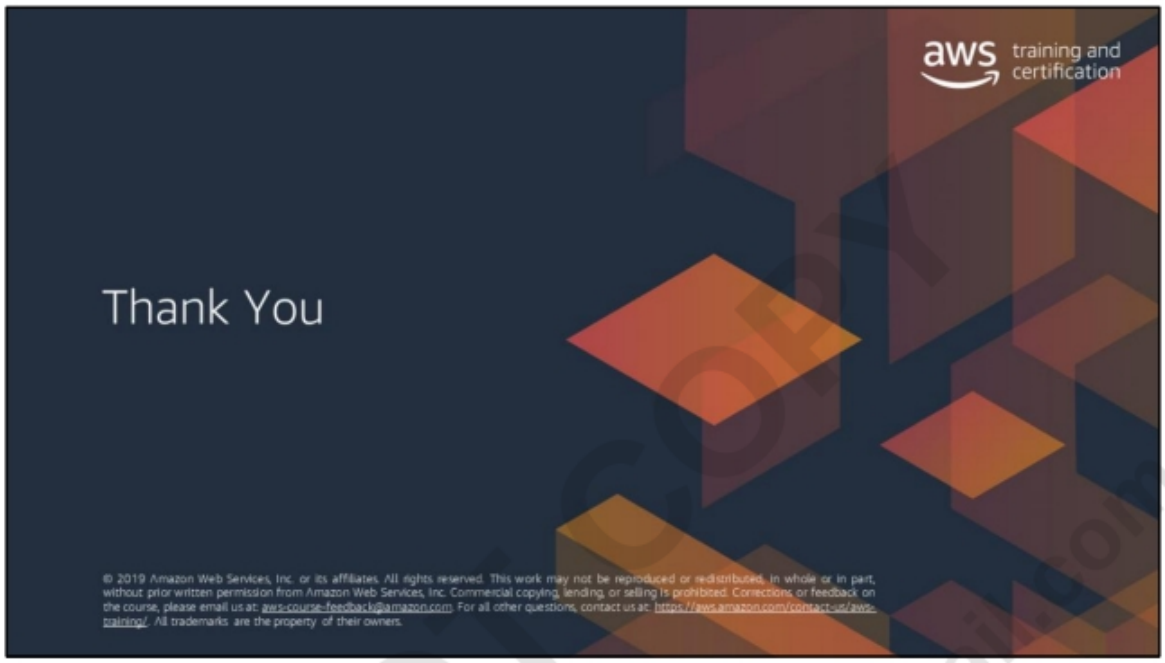


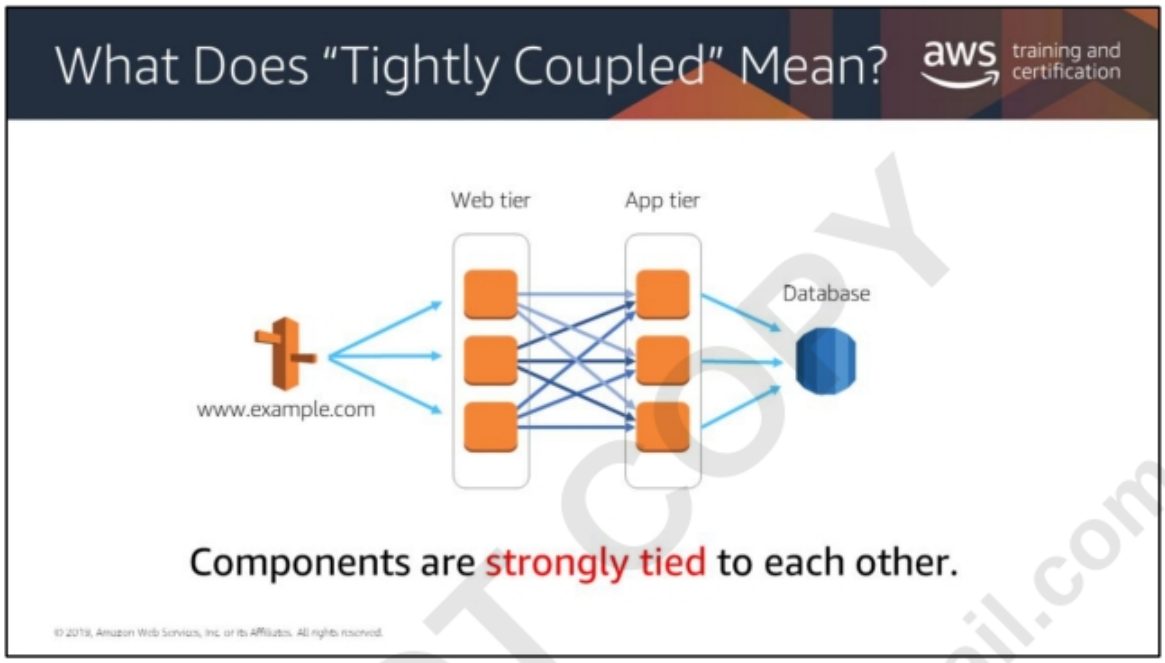
Traditional web hosting architecture implements a common three-tier web application model that separates the architecture into presentation, application, and persistence layers. Scalability is provided by adding hosts at the presentation, persistence, or application layers.

Web application hosting in the AWS Cloud

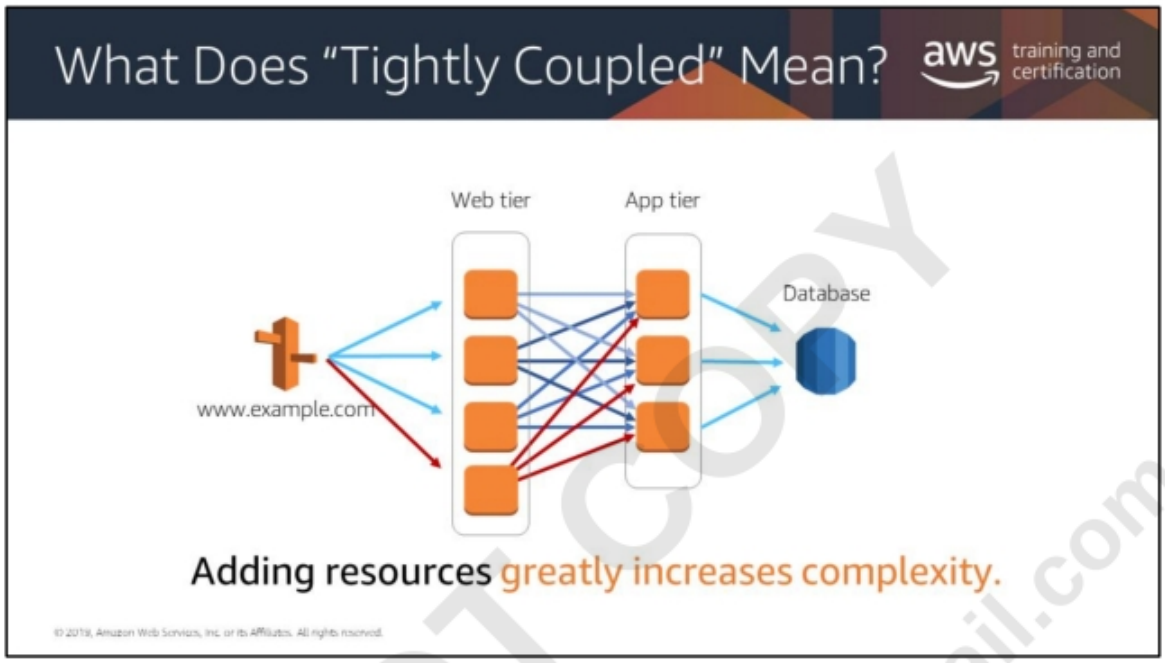
The traditional web hosting architecture is easily portable to the cloud services provided by AWS products with only a small number of modifications, but the first question that should be asked concerns the value of moving a classic web application hosting solution into the AWS cloud. If you decide that the cloud is right for you, you'll need a suitable architecture.

- *Amazon Route 53* provides DNS services to simplify domain management and zone APEX support.
- *Amazon CloudFront* provides edge caching for high-volume contents.
- *Elastic Load Balancing* spreads traffic to web server Auto Scaling groups in this diagram.
- *Exterior Firewall* is moved to every web server instance via security groups.
- *Backend Firewall* is moved to every back-end instance.
- *App server load balancer* on Amazon EC2 instances spreads traffic over the app server cluster.
- *Amazon ElastiCache* provides caching services for the app, which removes load from the database tier.

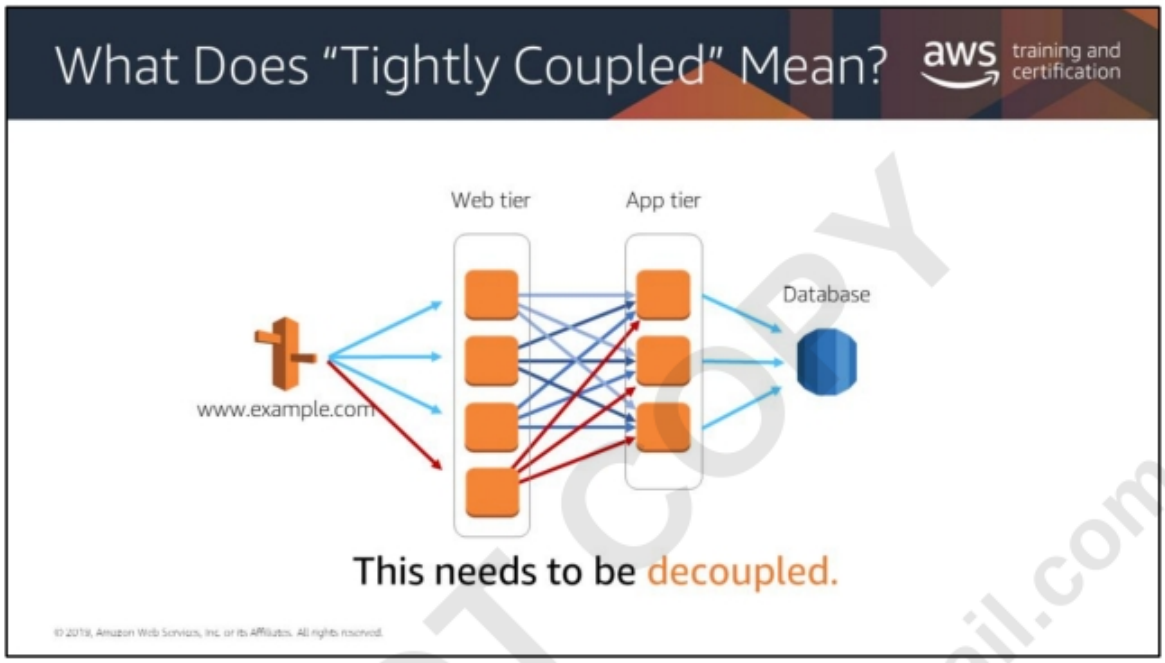


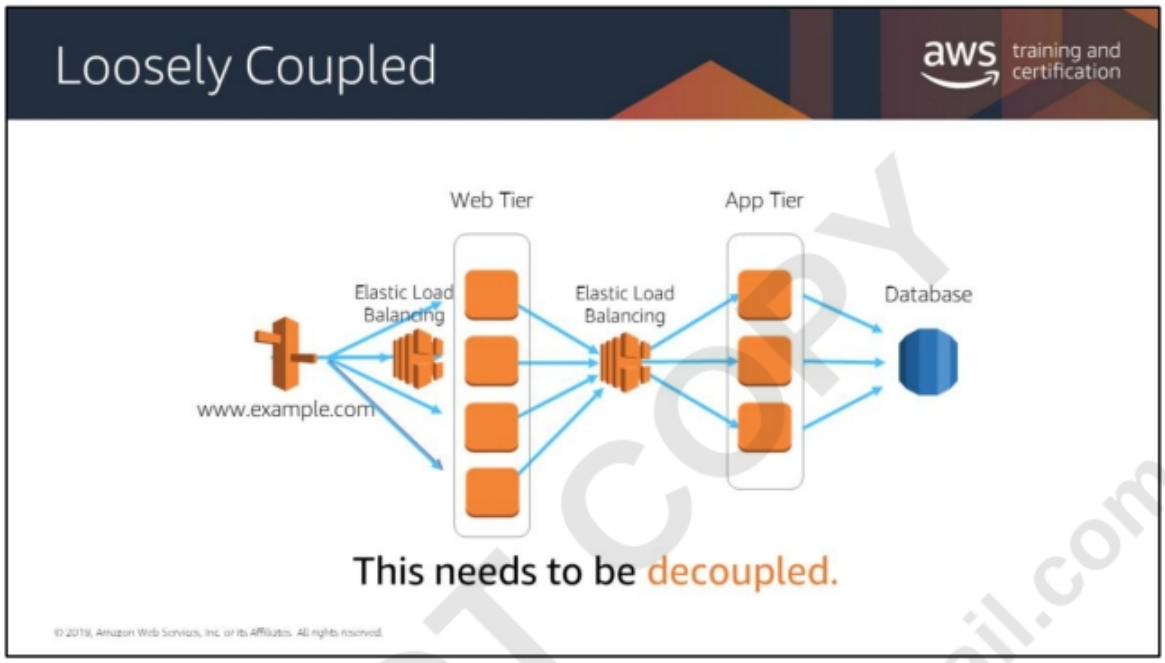


Traditional infrastructures revolve around chains of tightly integrated servers, each with a specific purpose. When one of those components/layers goes down, however, the disruption to the system can ultimately be fatal. Additionally, it impedes scaling; if you add or remove servers at one layer, every server on every connecting layer has to be connected appropriately also.

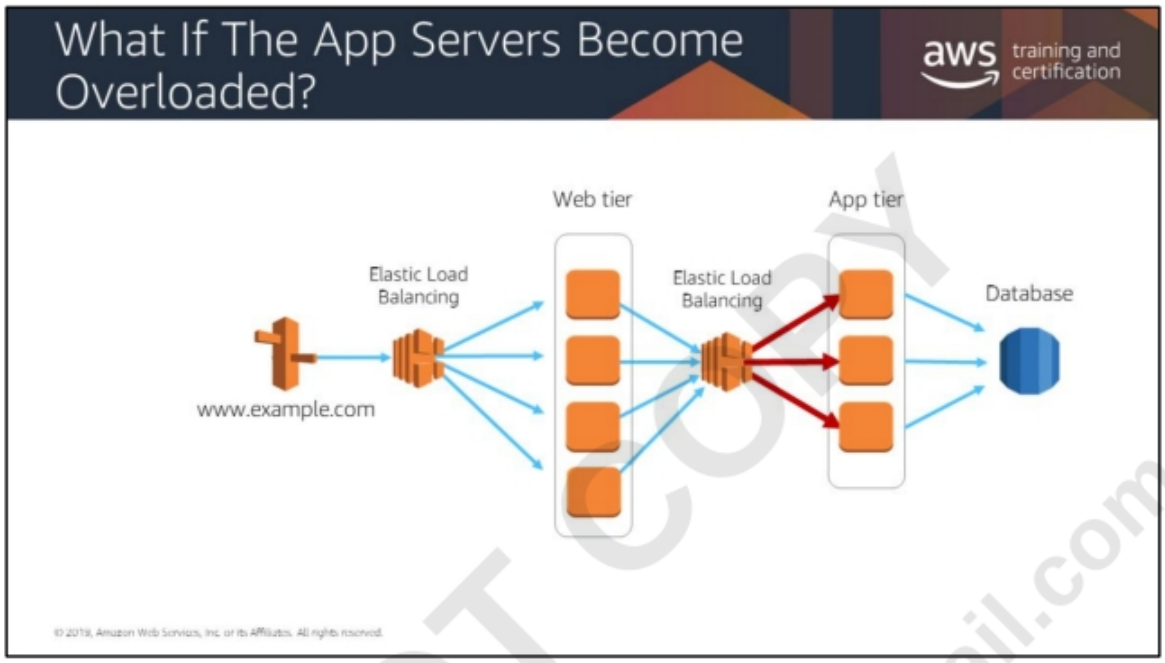


You need to reduce interdependencies so that the change or failure of one component does not affect other components. With loose coupling, you leverage managed solutions as intermediaries between layers of your system. This way, failures and scaling of a component or a layer are automatically handled by the intermediary.

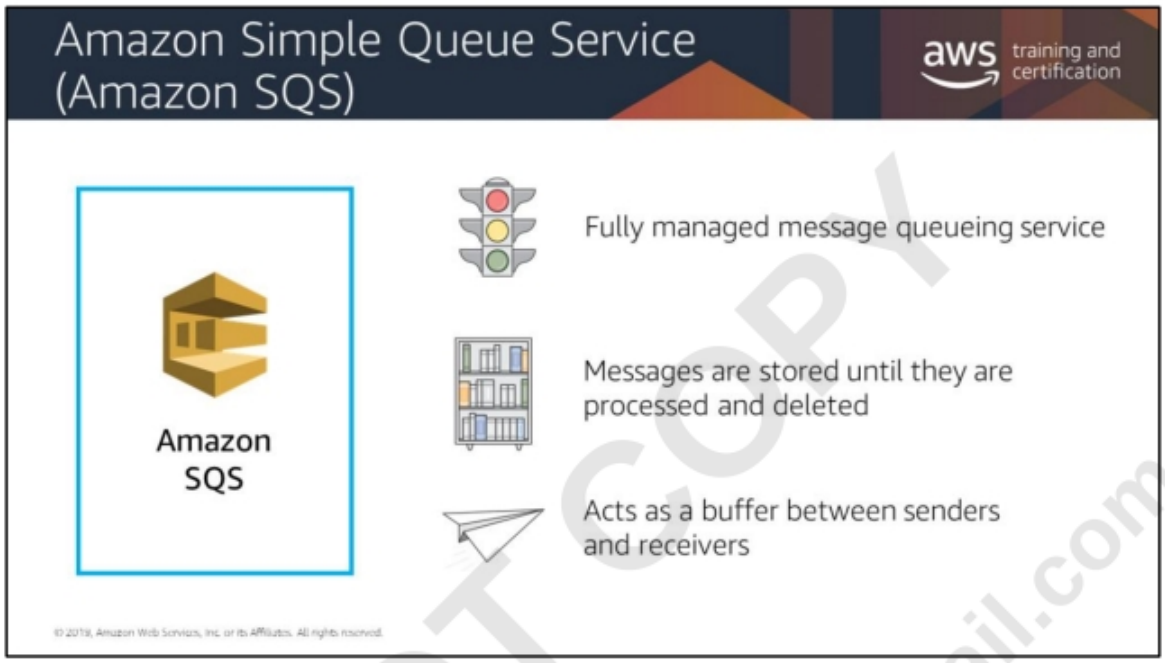




/



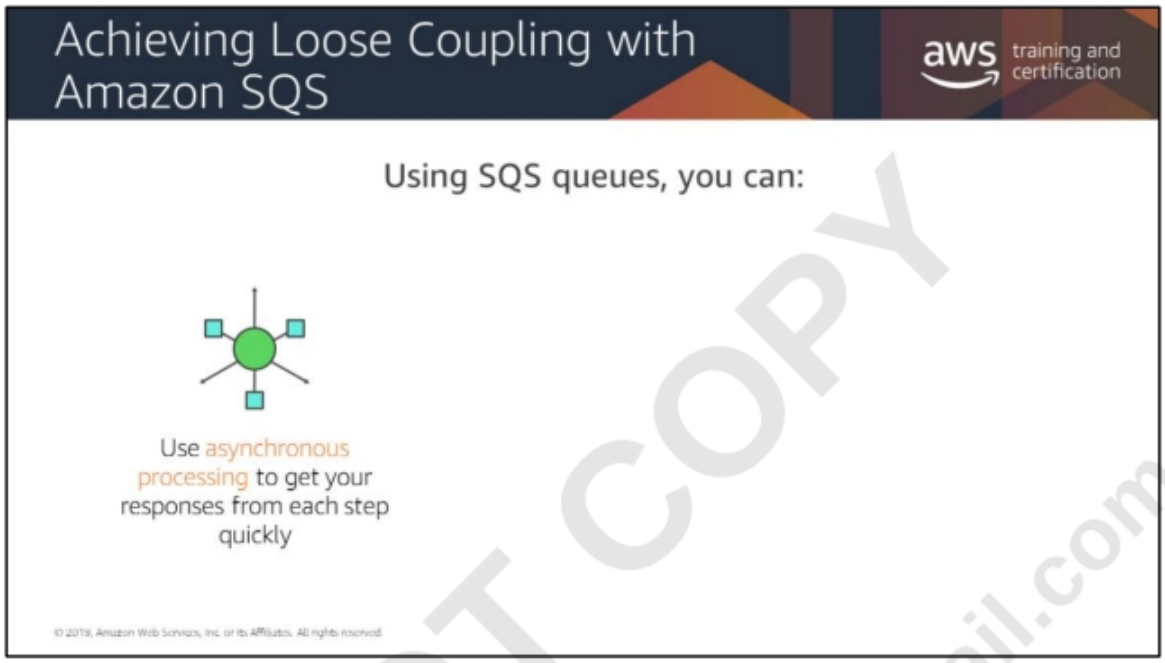
Consider a web application that processes customer orders. One potential point of vulnerability in the order processing workflow is in saving the order in the database. As a new requirement, the business expects that every order has been persisted into the database. However, any potential deadlock, race condition, or network issue could cause the persistence of the order to fail. Then, the order is lost with no recourse to restore the order.

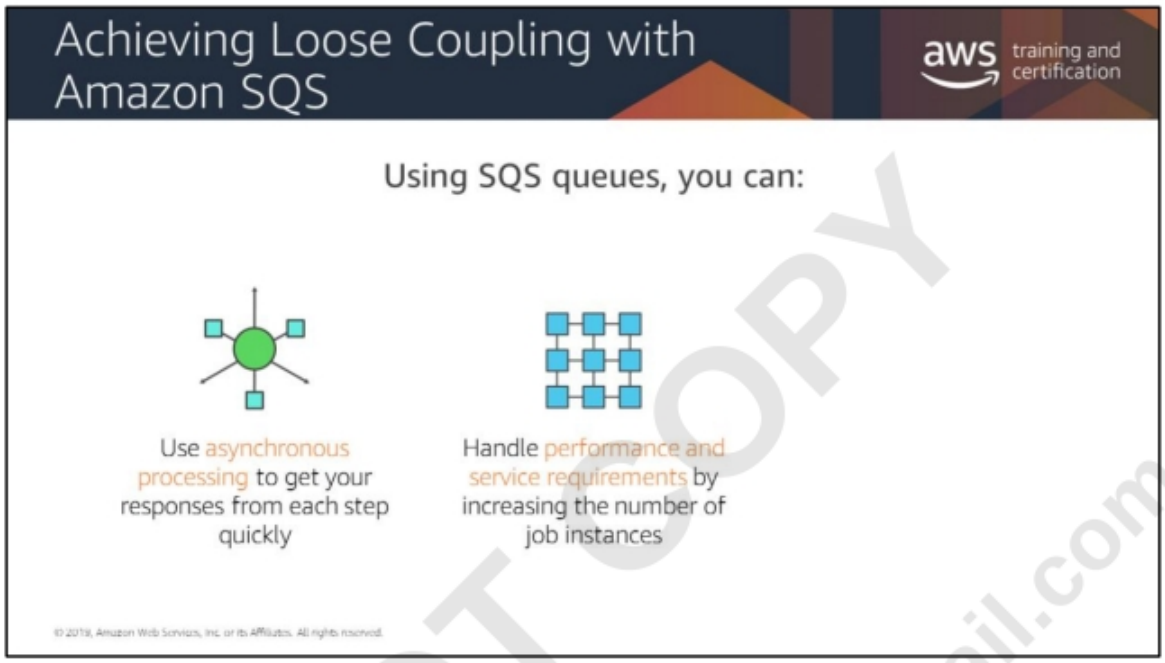


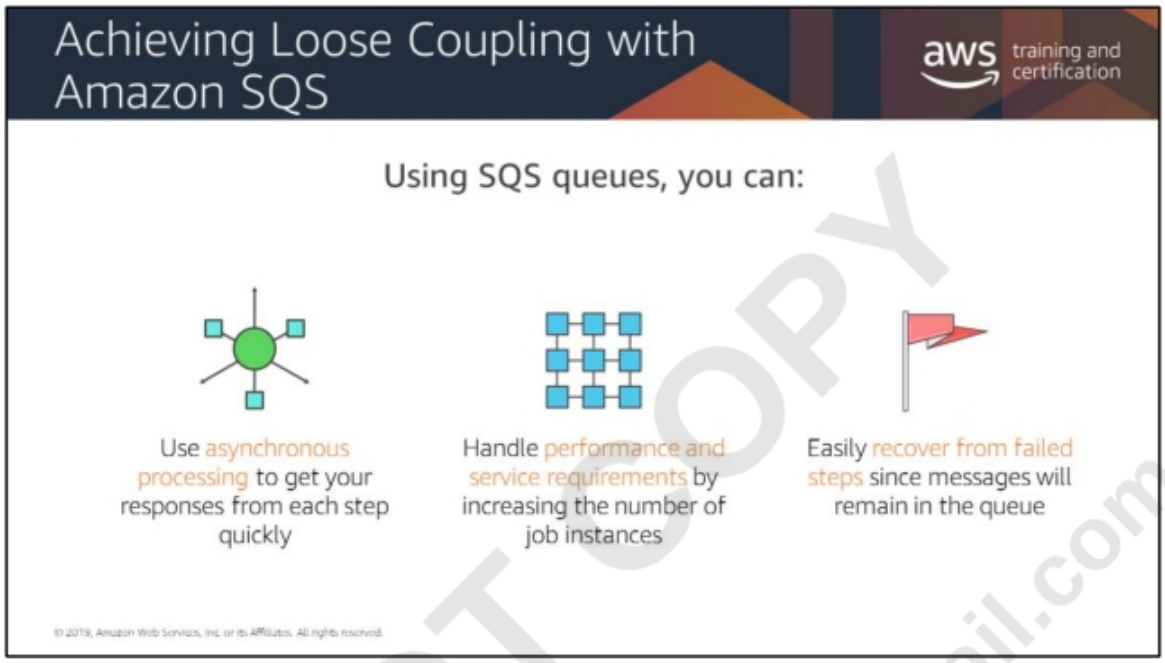
Amazon SQS is a fully-managed service that requires no administrative overhead and little configuration. The service works on a massive scale, processing billions of messages per day. It stores all message queues and messages within a single, highly-available AWS Region with multiple redundant Availability Zones, so that no single computer, network, or Availability Zone failure can make messages inaccessible. Messages can be sent and read simultaneously.

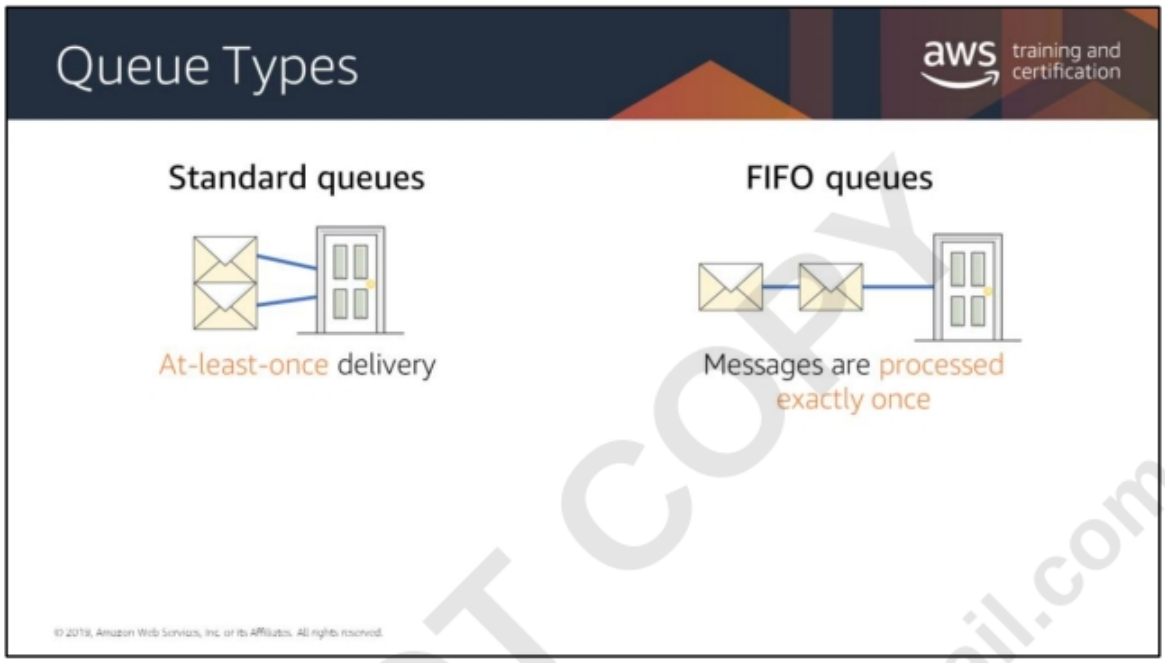
Developers can securely share Amazon SQS queues anonymously or with specific AWS accounts. Queue sharing can also be restricted by IP address and time-of-day. Server-side encryption (SSE) protects the contents of messages in Amazon SQS queues using keys managed in the AWS Key Management Service (AWS KMS). SSE encrypts messages as soon as Amazon SQS receives them. The messages are stored in encrypted form and Amazon SQS decrypts messages only when they are sent to an authorized consumer.

/







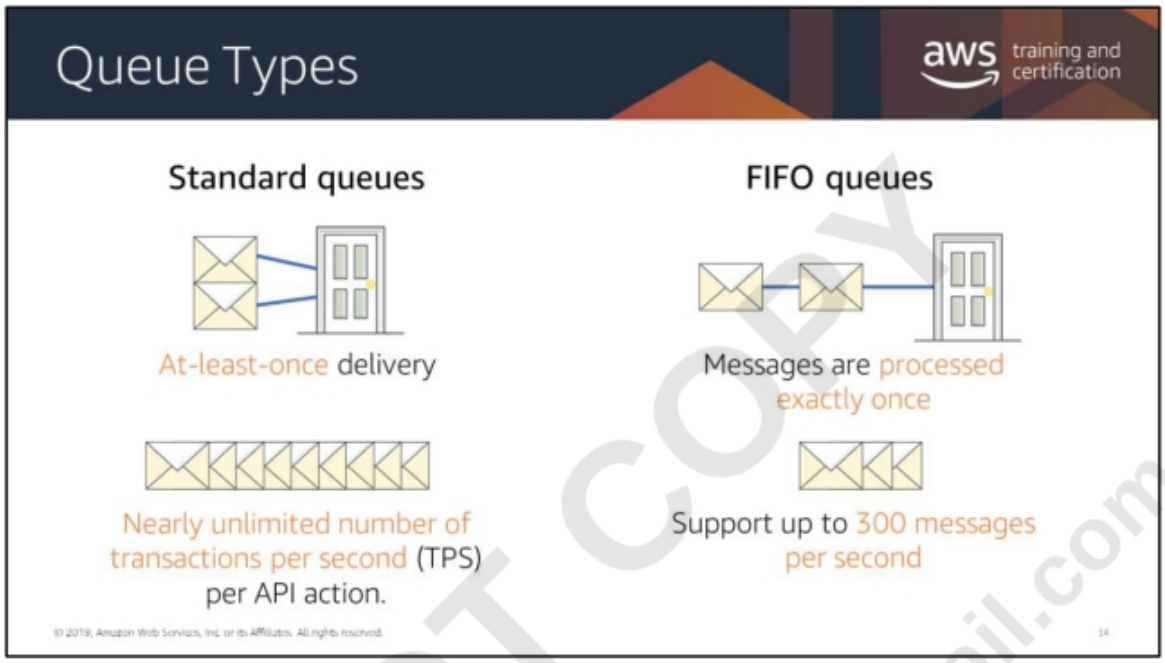


There are two types of SQS Queues: Standard and FIFO.

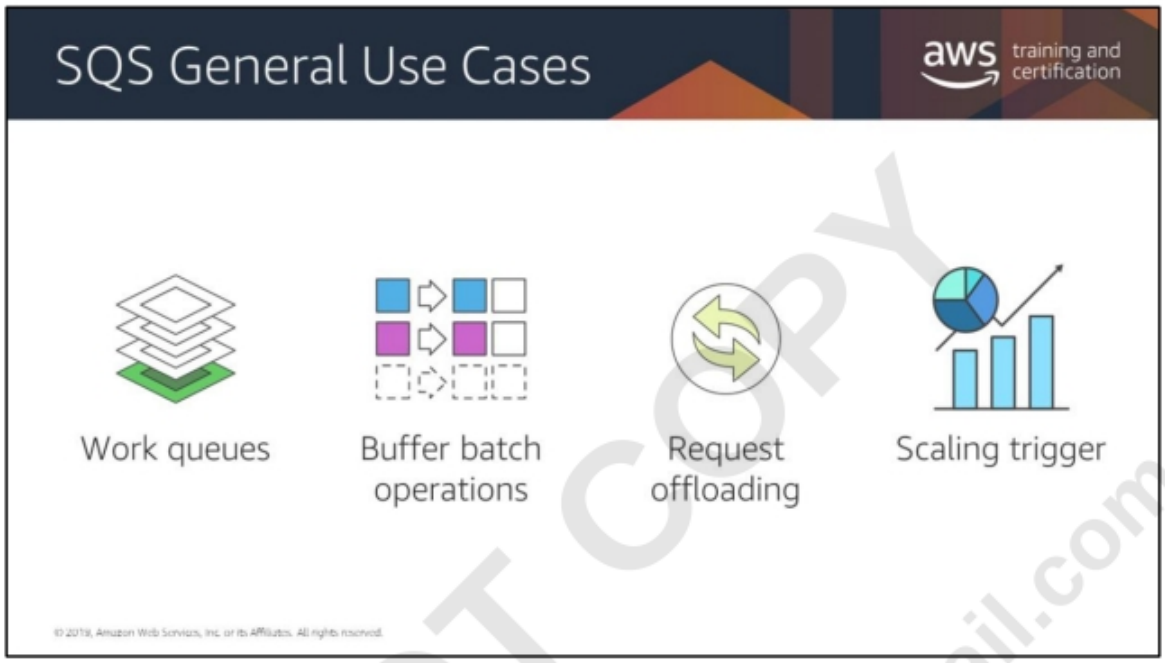
Standard queues offer at-least-once delivery and best-effort ordering.
At-Least-Once Delivery means that occasionally more than one copy of a message is delivered.
Best-Effort Ordering means occasionally messages might be delivered in an order different from which they were sent.

FIFO (*first in, first out*) queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent.

/



However, standard queues offer maximum throughput, while FIFO queues support up to 300 messages per second (300 send, receive, or delete operations per second). When you batch 10 messages per operation (maximum), FIFO queues can support up to 3,000 messages per second.



There are many different ways to use SQS queues.

Work queues: Decouple components of a distributed application that may not all process the same amount of work simultaneously.

Buffering batch operations: Add scalability and reliability to your architecture and smooth out temporary volume spikes without losing messages or increasing latency.

Request offloading: Move slow operations off of interactive request paths by enqueueing the request.

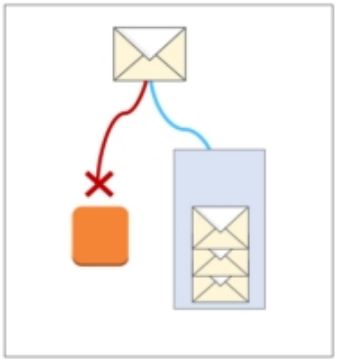
Auto Scaling: Use Amazon SQS queues to help determine the load on an application, and when combined with Auto Scaling, you can scale the number of Amazon EC2 instances out or in, depending on the volume of traffic.

/

Amazon SQS Features

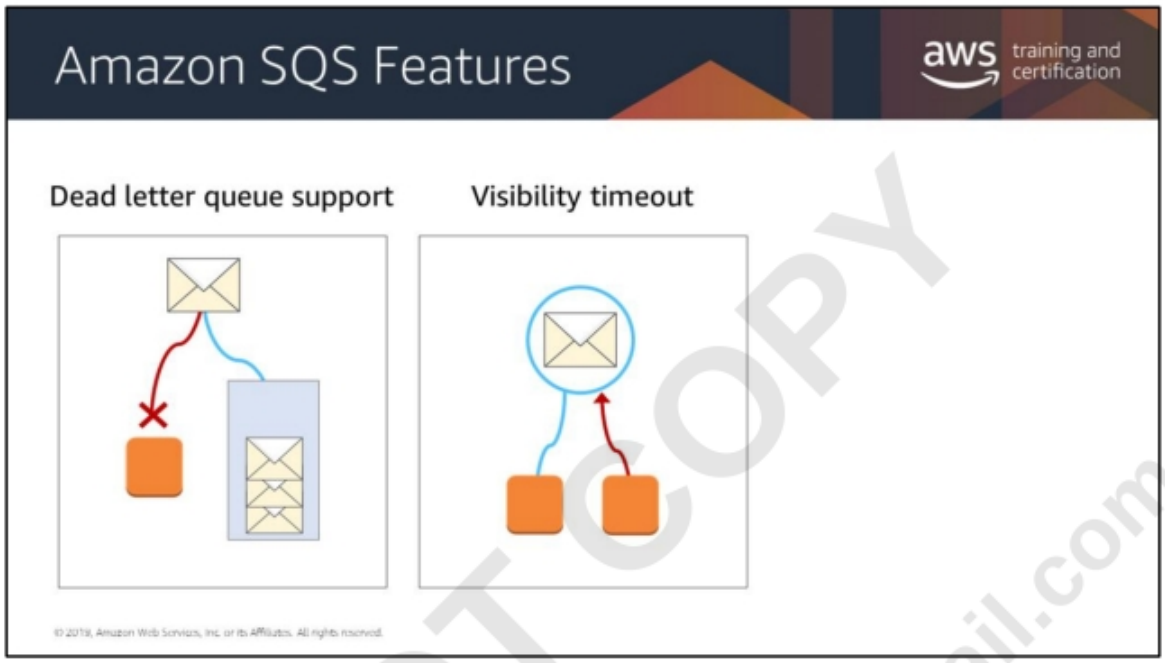
aws training and certification

Dead letter queue support

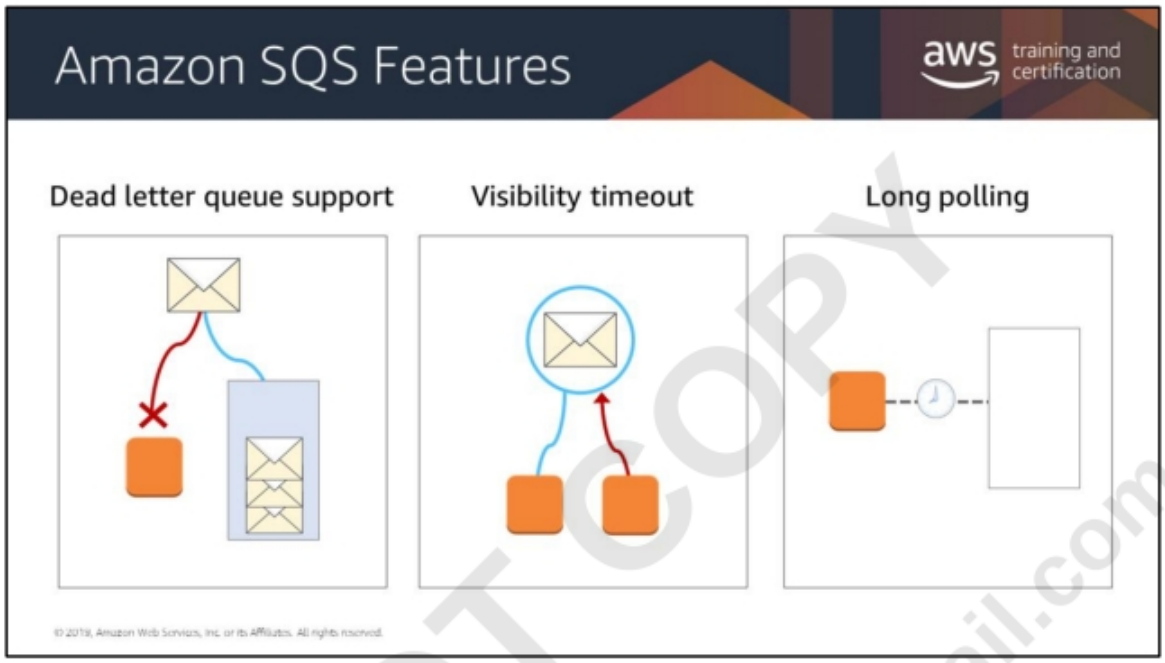


A *dead letter queue* (DLQ) is a queue of messages that could not be processed. It receives messages after a maximum number of processing attempts has been reached. A DLQ is just like any other Amazon SQS queue. Messages can be sent to it and received from it like any other SQS queue. You can create a DLQ from the SQS API and the SQS console.

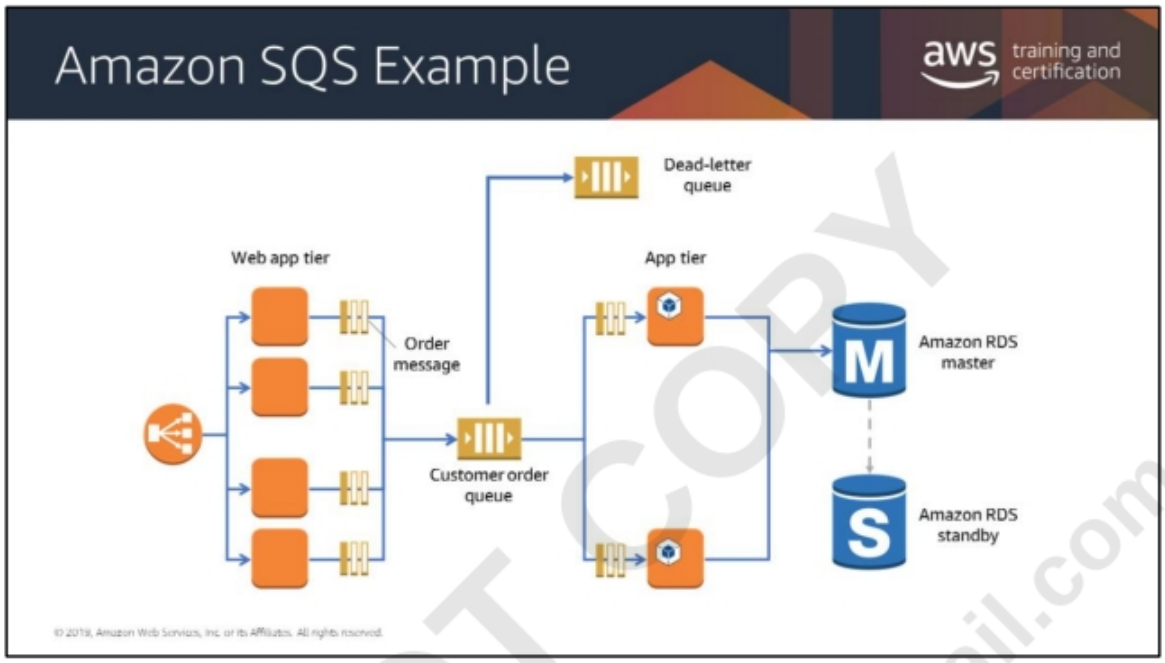
/



Visibility timeout is the period of time that a message is "invisible" to the rest of your application after an application component gets it from the queue. During the visibility timeout, the component that received the message usually processes it and then deletes it from the queue. This prevents multiple components from processing the same message. When the application needs more time for processing, the "invisible" timeout can be modified.



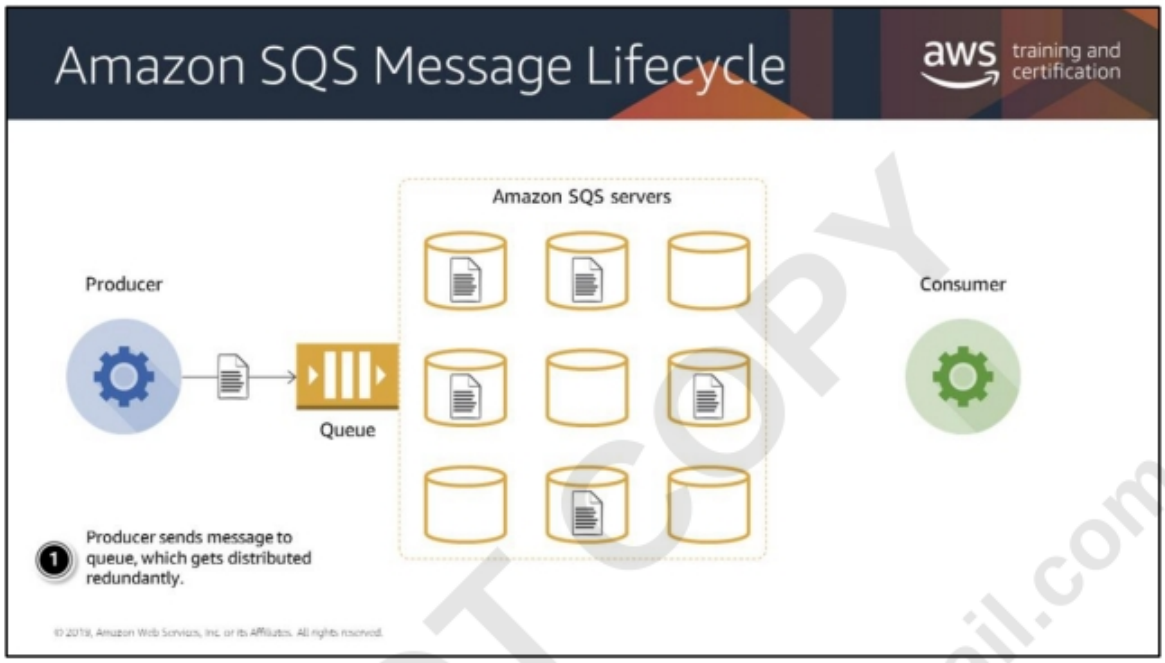
Long polling is a way to retrieve messages from your Amazon SQS queues. The default of short polling returns immediately, even if the message queue being polled is empty. However, long polling doesn't return a response until a message arrives in the message queue, or the long poll times out. Long polling makes it inexpensive to retrieve messages from your Amazon SQS queue as soon as the messages are available.



Introducing an SQS queue helps improve your ordering application. Using the queue isolates the processing logic into its own component and runs it in a separate process from the web application. This, in turn, allows the system to be more resilient to spikes in traffic, while allowing work to be performed only as fast as necessary in order to manage costs. In addition, you now have a mechanism for persisting orders as messages (with the queue acting as a temporary database), and have moved the scope of your transaction with your database further down the stack. In the event of an application exception or transaction failure, this ensures that the order processing can be retired or redirected to the Amazon SQS Dead Letter Queue (DLQ), for re-processing at a later stage.

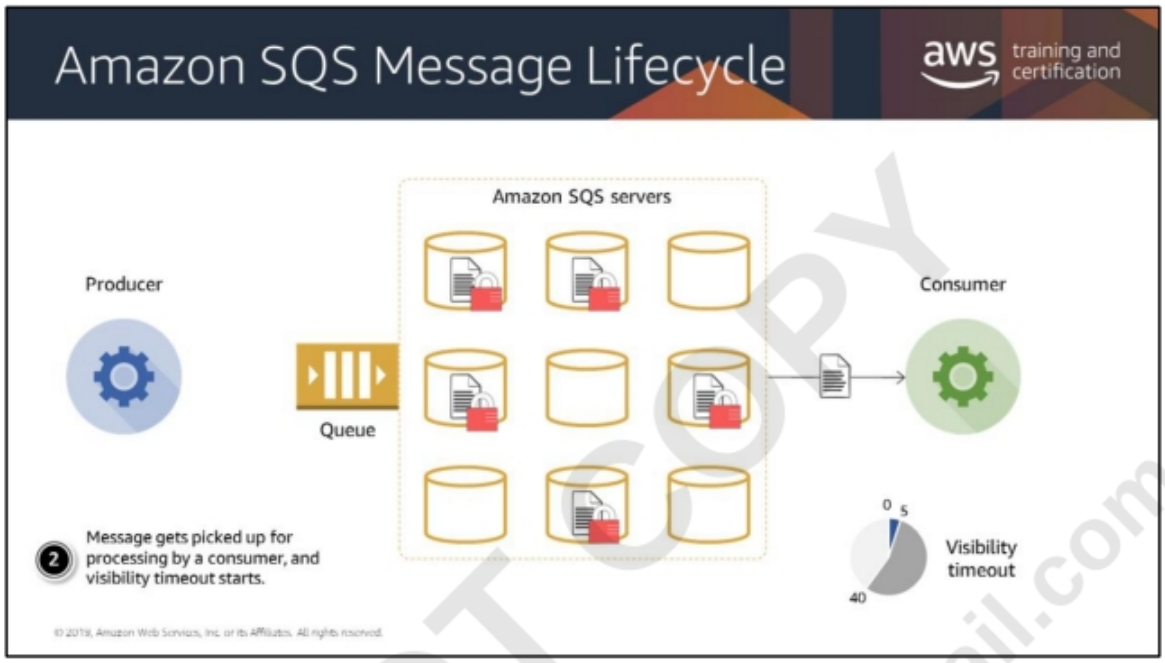
For more information regarding this use case, see https://aws.amazon.com/blogs/compute/building-loosely-coupled-scalable-c-applications-with-amazon-sqs-and-amazon-sns/

/

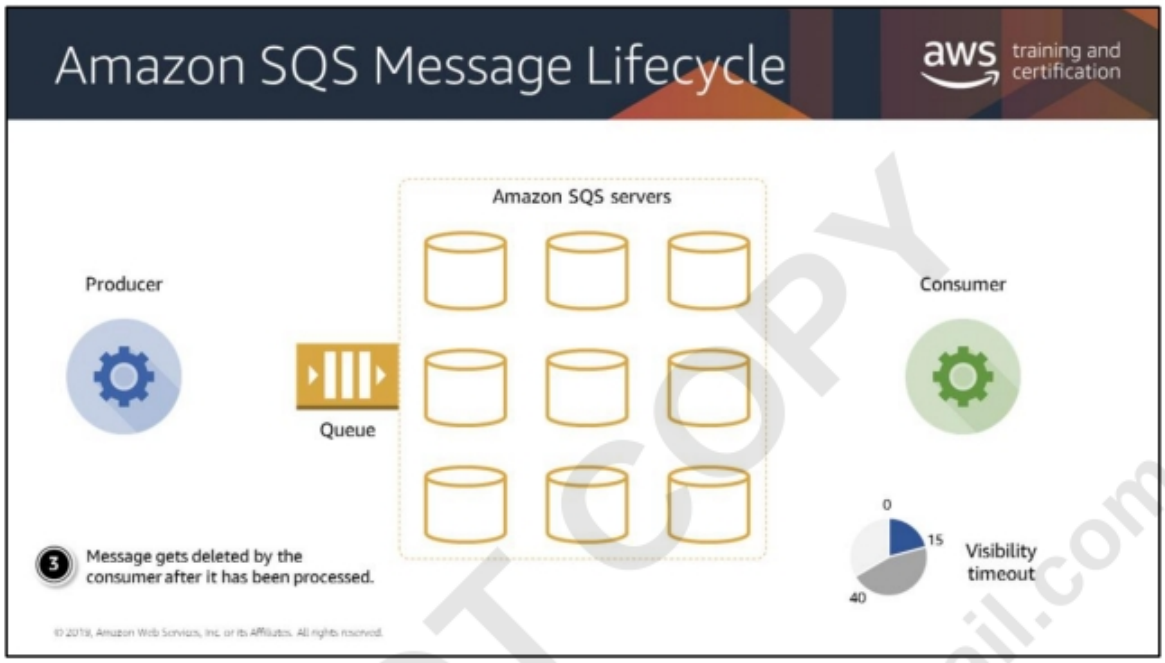


Amazon Simple Queue Service (Amazon SQS) is a distributed queue system that enables web service applications to queue messages that one component in the application generates to be consumed by another component. A *queue* is a temporary repository for messages that are waiting to be processed, keeping messages from 1 to 14 days (default is 4 days). Using Amazon SQS, you can decouple the components of an application so they run independently. Messages can contain up to 256 KB of text in any format. Amazon SQS supports multiple producers and consumers interacting with the same queue. Amazon SQS can be used with several AWS services including: Amazon EC2, Amazon S3, Amazon ECS, AWS Lambda, and Amazon DynamoDB.

Amazon SQS offers two types of message queues. Standard queues offer maximum throughput, best-effort ordering, and at-least-once delivery. Amazon SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent, with limited throughput. The following scenario describes the lifecycle of an Amazon SQS message in a queue, from creation to deletion. Here we have a producer sending a message to a queue, and the message is distributed across the Amazon SQS servers redundantly.

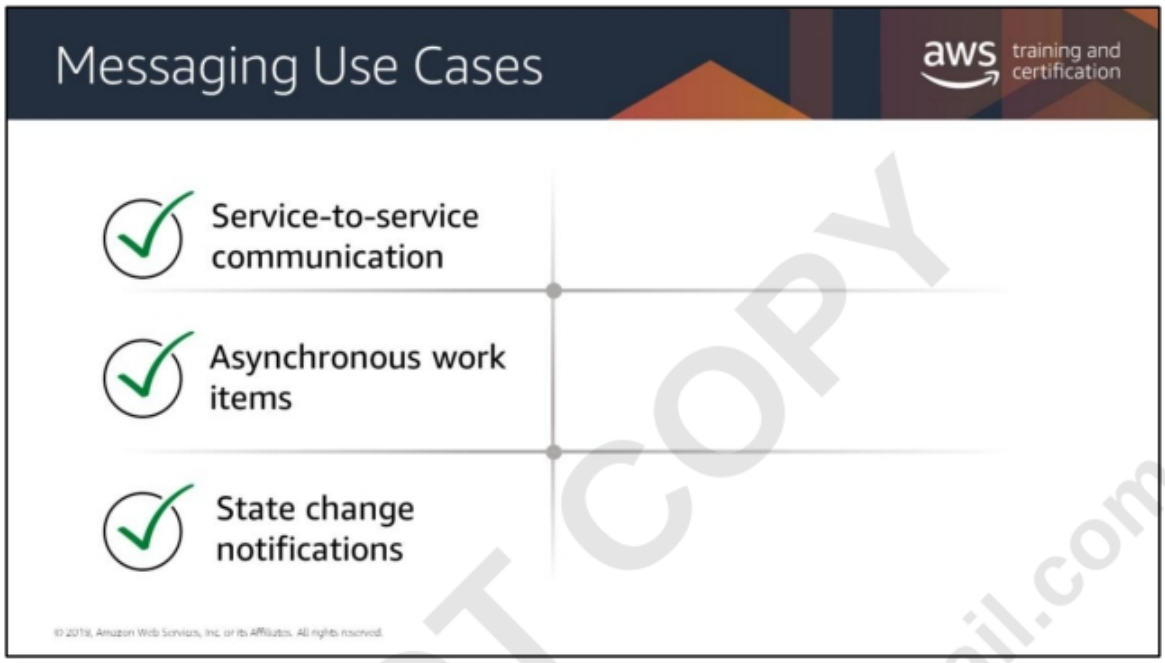


When a consumer is ready to process messages, it retrieves the message from the queue. While the message is being processed, it remains in the queue. To prevent other consumers from processing the message again, Amazon SQS sets a *visibility timeout*, a period of time during which Amazon SQS prevents other consumers from receiving and processing the message. The default visibility timeout for a message is 30 seconds. In this case, we have it set up for 40 seconds. The maximum is 12 hours. If the consumer fails before deleting the message before the visibility timeout expires, the message becomes visible to other consumers and the message may be processed again. Typically, you should set the visibility timeout to the maximum time that it takes your application to process and delete a message from the queue.



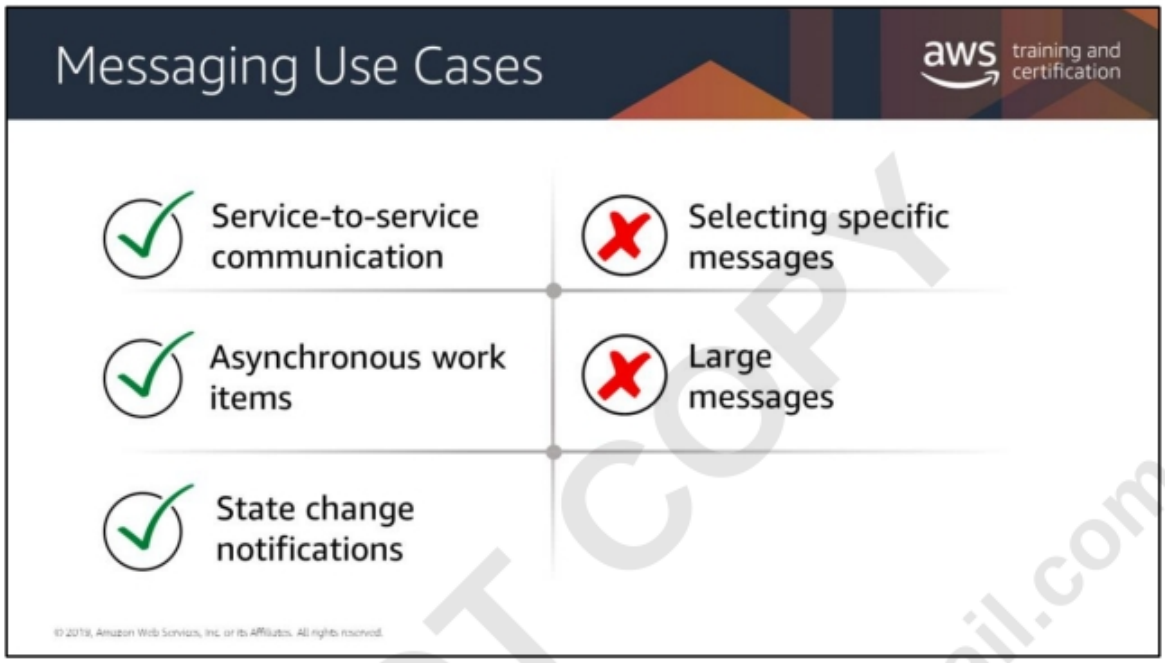
Amazon SQS doesn't automatically delete the message. Because Amazon SQS is a distributed system, there's no guarantee that the consumer actually receives the message (for example, due to a connectivity issue, or due to an issue in the consumer application). Thus, the consumer must delete the message from the queue after receiving and processing it.

/

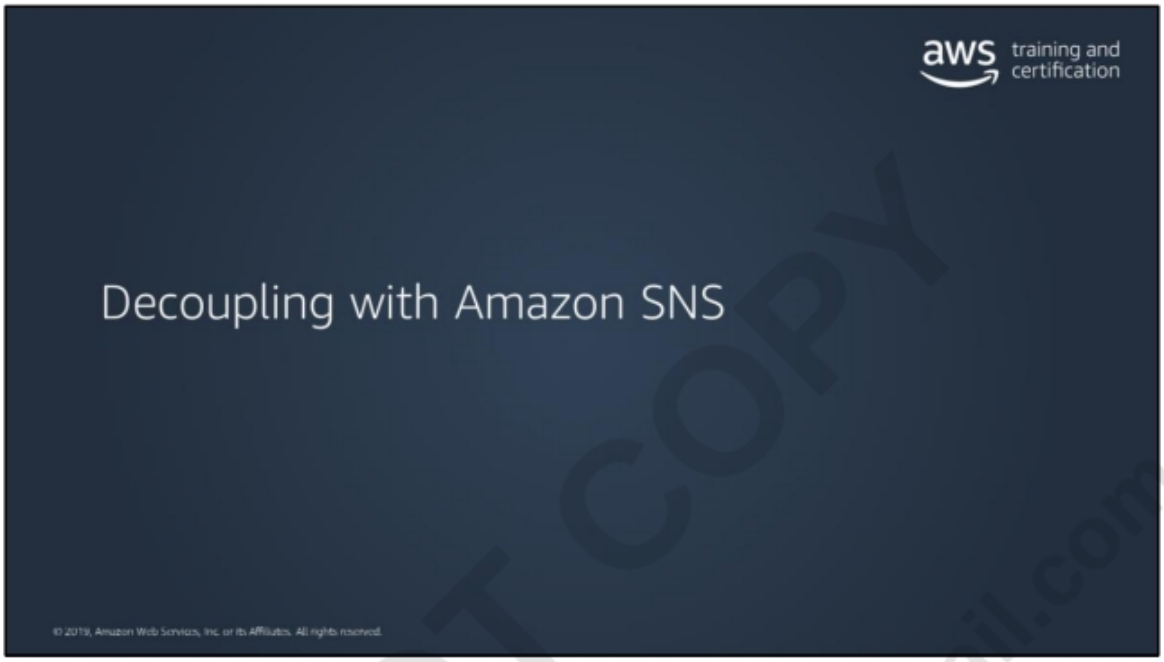


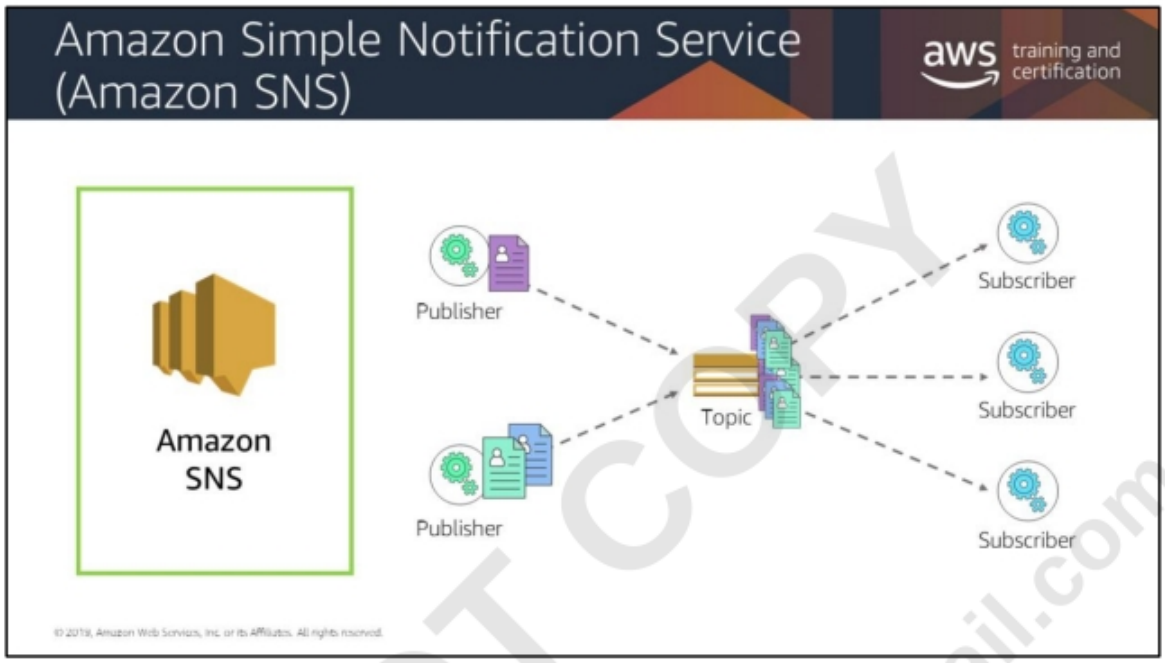
There are some common use cases where messaging services are a great fit. When there are two services or systems that need to communicate with each other. Let's say a website (the frontend) has to update customer's delivery address in a customer relationship management (CRM) system (the backend). Alternatively, you can also set up a queue and have the frontend website code send messages to the queue and have the backend CRM service to consume them. Or let's say a hotel booking system needs to cancel a booking and this process takes a long time. Alternatively, you can put a message into a queue and have the same hotel booking system consume messages from that queue and perform asynchronous cancellations. Also, messaging services are great for change notifications. You have a service that manages some resource and other services that receive updates about changes to those resources. For example, an inventory system may publish notifications when a certain item is low and needs ordering.

/



It's also important to know when a particular technology won't fit well with your use case. Messaging has its own set of commonly encountered anti-patterns. It's tempting to have the ability to receive messages selectively from a queue that match a particular set of attributes, or even match an ad-hoc logical query. **For example, a service requests a message with a particular attribute because it contains a response to another message that the service sent out. This can lead to a scenario where there are messages in the queue that no one is polling for and are never consumed.** Most messaging protocols and implementations work best with reasonably sized messages (in the tens or hundreds of KBs). As message sizes grow, it's best to use a dedicated storage system, such as Amazon S3, and pass a reference to an object in that store in the message itself.





Amazon Simple Notification Service (Amazon SNS) is a web service that makes it easy to set up, operate, and send notifications from the cloud. The service follows the "publish-subscribe" (pub-sub) messaging paradigm, with notifications being delivered to clients using a "push" mechanism.

You create a topic and control access to it by defining policies that determine which publishers and subscribers can communicate with the topic. A publisher sends messages to topics they have created or to topics they have permission to publish to.

Instead of including a specific destination address in each message, a publisher sends a message to the topic. Amazon SNS matches the topic to a list of subscribers who have subscribed to that topic and delivers the message to each of those subscribers.

Each topic has a unique name that identifies the Amazon SNS endpoint for publishers to post messages and subscribers to register for notifications. Subscribers receive all messages published to the topics to which they subscribe, and all subscribers to a topic receive the same messages.

/

Amazon SNS supports encrypted topics. When you publish messages to encrypted topics, Amazon SNS uses customer master keys (CMK), powered by AWS KMS (https://aws.amazon.com/kms/) , to encrypt your messages.

Amazon SNS supports customer-managed as well as AWS-managed CMKs. As soon as Amazon SNS receives your messages, the encryption takes place on the server, using a 256-bit AES-GCM algorithm. The messages are stored in encrypted form across multiple Availability Zones (AZs) for durability and are decrypted just before being delivered to subscribed endpoints, such as Amazon Simple Queue Service (Amazon SQS) queues, AWS Lambda functions, and HTTP and HTTPS webhooks.

https://aws.amazon.com/blogs/compute/encrypting-messages-published-to-amazon-sns-with-aws-kms/

/