**6**

**Solution Architecture Design Patterns**

Have you ever wondered how large enterprises design scalable systems? Before starting application development, solution architects worked across organizations and weighed multiple options to develop architecture design to handle the business need. There are multiple ways to design a solution. A solution architect needs to take the right approach based on user requirements along with the architecture constraints of cost, performance, scalability, and availability. In this chapter, you will learn about various solution architecture patterns along with reference architectures and how to apply them in real-world scenarios.

In the previous chapters, you learned about the attributes and principles of solution architecture design. This chapter is both exciting and essential as you will be able to apply your learning to various architecture design patterns. In this chapter, you will gain an understanding of some of the significant solution architecture patterns, such as layered, event-driven, microservice, loosely coupled, service-oriented, and RESTful architectures.

You will learn the advantages of various architectural designs and examples that demonstrate when to utilize them. You will also gain an understanding of architecture design anti-patterns in addition to the following architecture design patterns:

* Building an *n*-tier layered architecture
* Creating a multi-tenant SaaS-based architecture
* Building stateless and stateful architecture designs
* Understanding service-oriented architecture
* Building a serverless architecture
* Creating a microservice architecture
* Building a queue-based architecture
* Creating an event-driven architecture
* Building a cache-based architecture
* Understanding the circuit breaker pattern
* Implementing the bulkheads pattern
* Creating a floating IP pattern
* Deploying an application with a container
* Database handling in application architecture
* Avoiding anti-patterns in solution architecture

By the end of the chapter, you will know how to optimize your solution architecture design and apply best practices, making this chapter the center point and core of your learning.

**Building an n-tier layered architecture**

In *n*-tier architecture (also known as **multitier architecture**), you need to apply the principle of loosely coupled design (refer to *Chapter 4*, *Principles of Solution Architecture Design*) and attributes of scalability and elasticity (refer to *Chapter 3*, *Attributes of the Solution Architecture*). In multilayer architecture, you divide your product functions into multiple layers, such as presentation, business, database, and services, so that each layer can be implemented and scaled independently.

With multitier architecture, it is easy to adopt new technologies and make development more efficient. This layered architecture provides the flexibility to add new features in each layer without disturbing the features of other layers. In terms of security, you can keep each layer secure and isolated from the others, so if one layer gets compromised, the other layers won't be impacted. Application troubleshooting and management also become manageable as you can quickly pinpoint where the issue is coming from and which part of the application needs to be troubleshot.

The most common architecture in multilayer design is **three-tier architecture**, so let's learn more about it. The following diagram shows an architecture that allows you to interact with a web application from the browser and perform the required functions, for example, ordering your favorite T-shirt or reading a blog and leaving a comment:

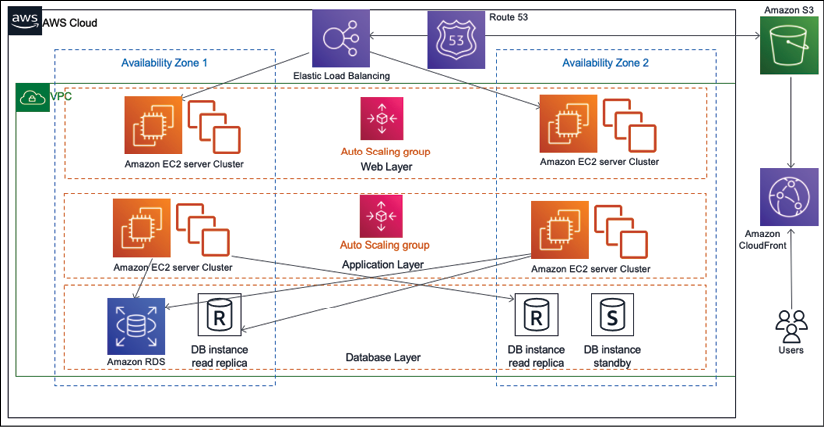


Figure 6.1: Three-tier website architecture

In the preceding architecture, you have the following three layers:

* **Web Layer**: The web layer is the user-facing part of the application. End users interact with the web layer to collect or provide information.
* **Application Layer**: The application layer mostly contains business logic and acts upon information received from the web layer.
* **Database Layer**: All kinds of user data and application data are stored in the database layer.

Let's take a look at these layers in more detail.

**The web layer**

The web layer is also known as the **presentation tier**. The web layer provides a user interface that helps the end user to interact with the application. The web layer is your user interface (in this case, the website page), where the user enters information or browses for it. Web developers may build a presentation tier user interface in technologies such as HTML, CSS, AngularJS, ReactJS, **JavaServer Pages** (**JSP**), and **Active Server Pages** (**ASP**). This tier collects the information from the user and passes it to the application layer.

The web layer is user-facing, so organizations spend most of their time improving the user experience. Many organizations have dedicated **User Experience** (**UX**) teams that conduct research in various areas to understand how users interact with applications.

Also, the solution architect needs to make sure architecture design includes UX input and page load performance. There should be seamless information flow between the web layer and application layer to return the correct information to users within the expected timeframe, such as user login, profile loading, and so on. Let look at more details on the application layer.

**The application layer**

The application layer is also known as the **logic tier**, as this is the core of the product where all the business logic resides. The presentation tier collects the information from the user and passes it to the logic tier to process it and get a result. For example, on an e-commerce website such as [www.amazon.com](http://www.amazon.com/), users can enter a date range on the order page of the website to find their order summary. In return, the web layer passes the data range information to the application layer. The application layer processes the user input to perform business logic such as the count of orders, the sum of amounts, and the number of items purchased. This returns information to the web layer to render it for the user.

Generally, in three-tier architecture, all algorithms and complex logic live in the application tier, including creating a recommendation engine or showing personalized pages to the user as per their browsing history. You may add layers such as a domain layer, data access layer, or presentation layer to make a 4- or 5-tier architecture. Developers may choose to implement this layer in a server-side programming language, for example, C++, Java, .NET, or Node.js. The application layer is the center of system design and requires most of the design effort. Most of the application features depend on logic built at the application layer. The application layer performs logic on the data, which is stored in the database layer. Let's look at the database layer in more detail.

**The database layer**

The database layer, which is also known as the **data tier**, stores all the information related to user profiles and transactions. Essentially, it contains any data that needs to persist in being stored in the data tier. This information is sent back to the application layer for logic processing and then, eventually, it renders to the user in the web layer. For example, if the user is logged in to a website with their user ID and password, then the application layer verifies the user credentials with information stored in the database. If the credentials match the stored information, the user is allowed to log in and access the authorized area of the website.

The architect may choose to build a data tier in relational databases, for example, PostgreSQL, MariaDB, Oracle Database, MySQL, Microsoft SQL Server, Amazon Aurora, or Amazon RDS. The architect may also add a NoSQL database such as Amazon DynamoDB, MongoDB, or Apache Cassandra.

The data tier is not only used to store transaction information but also to hold user session information and application configuration. To meet performance needs, an architect might decide to add caching databases such as Memcached and Redis. You will learn more about various databases in *Chapter 13*, *Data Engineering for Solution Architecture*.

The data tier needs special attention in terms of security. You need to make sure to protect user information by applying data encryption at rest and in transit. In the *n*-tier layered architecture diagram, you will notice that each layer has its own auto scaling configuration, which means it can be scaled independently. Also, each layer has a network boundary, which means having access to one layer doesn't allow access to other layers. You will learn more about security considerations in *Chapter 8*, *Security Considerations*.

When designing multitier architecture, you need to consider how many layers should be added to your design. For example, the solution architect may decide to break down the application layer into a business layer, service layer, and persistent layer. However, each layer requires its fleet of servers and network configurations. So, adding more layers means increasing the cost and management overhead, whereas keeping fewer layers means creating a tightly coupled architecture. The architect needs to decide on the number of tiers based on application complexity and user requirement. For example, you may want to add additional tiers such as a data access layer for database access logic and keep the data storage layer for the database engine. You can add more layers to reduce complexity by defining logical separation, which can help to increase the maintainability of the general application and the ability to scale and achieve performance.

**Creating a multi-tenant SaaS-based architecture**

In the previous section, you learned about multitier architecture, which is also called a **single tenancy** when built for a single organization. Multi-tenant architecture is becoming more popular as organizations adopt the digital revolution while keeping the overall application and operational cost low. The **Software-as-a-Service** (**SaaS**) model is built on a multi-tenant architecture, where a single instance of the software and the supporting infrastructure serve multiple customers. In this design, each customer shares the application and database, with each tenant isolated by their unique configuration, identity, and data. They remain invisible to each other while sharing the same product.

As multi-tenant SaaS providers own everything from the hardware to the software, SaaS-based products offload an organization's responsibilities to the maintenance and updates of the application, as this is taken care of by the SaaS provider.

Each customer (tenant) can customize their user interface using a configuration without any code changes. As multiple customers share a common infrastructure, they get the benefit of scale, which lowers the cost further. Some of the most popular SaaS providers are Salesforce CRM, the Jira tool, and Amazon QuickSight.

As shown in the following architecture diagram, there are two organizations (tenants) using the same software and infrastructure. The SaaS vendor provides access to the application layer by allocating a unique tenant ID to each organization. Each tenant can customize their user interface as per their business needs using a simple configuration:

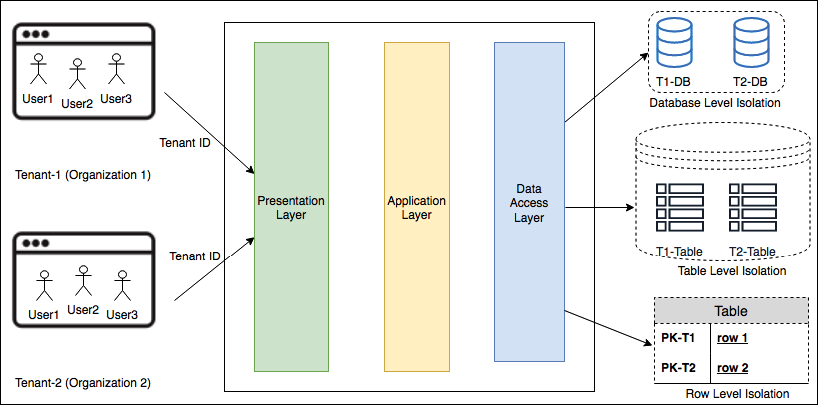


Figure 6.2: Multi-tenant SaaS architecture

As shown in the preceding architecture design, the presentation layer provides a user interface and the application layer holds the business logic. At the data access layer, each tenant will have data-level isolation with one of the following methods:

* **Database Level Isolation**: In this model, each tenant has its database associated with its tenant ID. When each tenant queries data from the user interface, they are redirected to their database. This model is required if the customer doesn't want a single shared database for compliance and security reasons.
* **Table Level Isolation**: This isolation level can be achieved by providing a separate table for each tenant. In this model, tables need to be uniquely assigned to each tenant, for example, with the tenant ID prefix. When each tenant queries data from the user interface, they are redirected to their tables as per their unique identifier.
* **Row Level Isolation**: In this isolation level, all tenants share the same table in a database. There is an additional column in a table where a unique tenant ID is stored against each row. When an individual tenant wants to access their data from the user interface, the data access layer of the application formulates a query based on the tenant ID to the shared table. Each tenant gets a row that belongs to their users only.

For enterprise customers, a careful assessment should be carried out, to understand whether a SaaS solution is the right fit for them based on their unique features' requirements. This is because often, a SaaS model has limited customization capabilities. Additionally, we need to find the cost value proposition if a large number of users need to subscribe. The cost comparison should be calculated based on the total cost of ownership when making a *build versus buy* decision. This is because building software is not the primary business of most organizations, so the SaaS model is becoming highly popular as organizations can focus on their business and let the experts handle the IT side of it.

**Building stateless and stateful architecture designs**

While designing a complex application such as an e-commerce website, you need to handle the user state to maintain activity flow, where users may be performing a chain of activities such as adding to the cart, placing an order, selecting a shipping method, and making a payment. Currently, users can use various channels to access an application, so there is a high possibility that they will be switching between devices; for example, adding items to the cart from their mobile and then completing checkout and payment from a laptop. In this situation, you would want to persist user activity across devices and maintain their state until the transaction is complete. Therefore, your architecture design and application implementation need to plan for user session management in order to fulfill this requirement.

To persist user states and make applications stateless, user session information needs to be stored in persistent database layers such as the NoSQL database. This state can be shared between multiple web servers or microservices. Traditionally, a monolithic application uses stateful architecture, where user session information is stored in the server itself rather than via any external persistence database storage.

The session storage mechanism is the main difference between stateless and stateful application designs. Since session information in a stateful application is local to the server, it cannot be shared between other servers and also doesn't support modern microservice architecture. You will learn more about microservice-based architecture in the *Creating a microservice architecture* section.

Often, a stateful application doesn't support horizontal scaling very well, as the application state persists in the server, which cannot be replaced. The stateful application works well early on when the user base was not very huge. However, as the internet becomes more popular, it is reasonable to assume that you will have millions of users active on a web application. Therefore, efficient horizontal scaling is important for handling such a large user base and achieving low application latency.

In a stateful application, state information is handled by the server, so once users establish a connection with a particular server, they have to stick with it until the transaction completes. You can put a load balancer in front of the stateful application, but to do that, you have to enable sticky sessions in a load balancer. A sticky session routes requests for a particular user session to the same physical machine that serviced the first request to ensure that the user session is not lost due to requests being routed to different servers. The load balancer has to route user requests to one server, where session information has been established. Enabling sticky sessions violates the load balancer's default round-robin request for the distribution method. Other issues may include lots of open connections to the server as you need to implement a session timeout for the client.

Your design approach should focus more on the shared session state using the stateless method, as it allows horizontal scaling. The following diagram shows an architecture that depicts a stateless application for a web application:

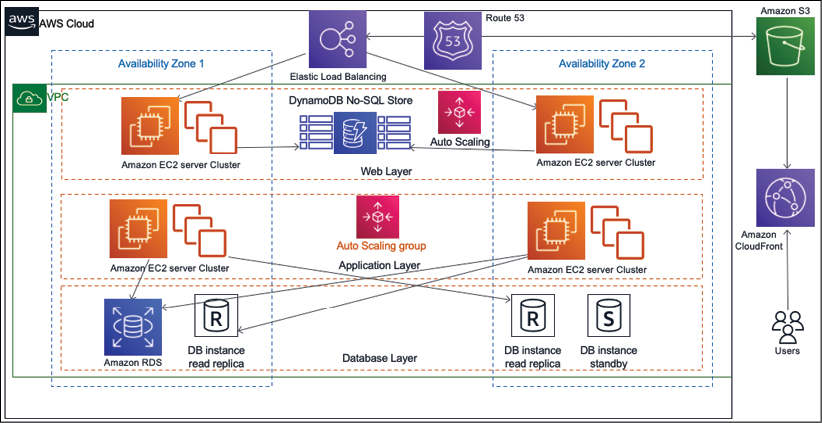


Figure 6.3: A stateless application architecture

The preceding architecture diagram is a three-tier architecture with a web, application, and database layer. To make applications loosely coupled and scalable, all user sessions are stored persistently in the NoSQL database, for example, Amazon DynamoDB.

You should use client-side storage, such as cookies, for the session ID. This architecture lets you use the scale-out pattern without having to worry about a loss of user state information. A stateless architecture removes the overhead to create and maintain user sessions and allows consistency across the application's modules. A stateless application has performance benefits too, as it reduces memory usage from the server side and eliminates the session timeout issue.

Adopting a stateless pattern can complicate tasks; however, with the right approach, you can achieve a rewarding experience for your user base. You can develop applications using the microservice approach with REST design patterns and deploy them in containers. For this, use authentication and authorization to connect users to the server.

You will learn more about the REST design pattern and microservices in the next section. As access to state information from multiple web servers focuses on a single location, you must use caution to prevent the performance of the data store from becoming a bottleneck.

**Understanding service-oriented architecture**

In **service-oriented architecture** (**SOA**) patterns, different application components interact with each other using a communication protocol over the network. Each service provides end-to-end functionality, for example, *fetching an order history*. SOA is widely adopted by large systems to integrate business processes, for example, taking your payment service from the main application and putting it as a separate solution.

In a general sense, SOAs take monolithic applications and spread some of those operations out into individual *services* that operate independently of each other. The goal of using an SOA is to loosen the coupling of your application's services. Sometimes, an SOA includes not just splitting services apart from one another but splitting resources into separate instances of that service. For instance, while some choose to store all of their company's data in a single database split by tables, an SOA would consider modularizing the application by function into separate databases altogether. This allows you to scale and manage throughput based on the individual needs of tables for each database.

SOA has multiple benefits, for example, the parallelization of development, deployment, and operation. It decouples the service so that you can optimize and scale each service individually.

However, it also requires more robust governance to ensure work performed by each service's team meets the same standard. With SOA, the solution could become complex enough to increase the overhead to balance that, so you need to make the right choice of tools and automation of service monitoring, deployment, and scaling.

There are multiple ways in which to implement SOA. Here, you will learn about the **Simple Object Access Protocol** (**SOAP**) web service architecture and **Representational State Transfer** (**REST**) web service architecture.

Originally, SOAP was the most popular messaging protocol, but it is a bit heavyweight as it entirely relies on XML for data interchange. Now, REST architecture is becoming more popular as developers need to build more lightweight mobile and web applications. Let's learn about both architectures and their differences in more detail.

**SOAP web service architecture**

**SOAP** is a messaging protocol that is used to exchange data in a distributed environment in XML format. SOAP is a standard XML where data is transported in an envelope format called a **SOAP envelope**, as shown in the following diagram:

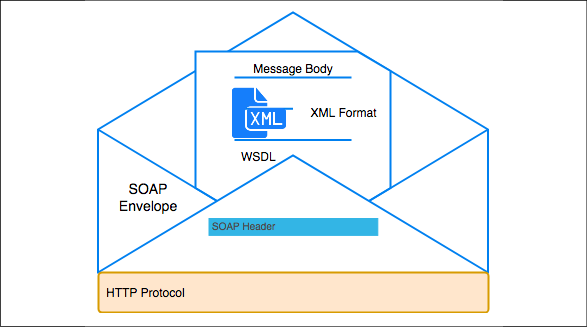


Figure 6.4: SOAP envelope for web service data exchange

As shown in the preceding diagram, a **SOAP envelope** contains two parts and these SOAP messages are formatted in XML and are typically sent using **HyperText Transfer Protocol** (**HTTP**):

* **SOAP Header**: The SOAP header provides information on how a recipient of a SOAP message should process it. It contains authorization information to deliver the message to the right recipient and for data encoding.
* **Message Body**: The message body contains the actual message in the **Web Services Description Language** (**WSDL**) specification. WSDL is an XML format file that describes the **Application Programming Interface** (**API**) contract with the message structure, API operations, and the server's **Unique Resource Locator** (**URL**) address. Using a WSDL service, a client application can determine where a service is being hosted and what functionality it can perform.

The following code shows an example of a SOAP envelope XML. Here, you can see both the header and message wrapped up under the SOAP envelope:

<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">

<env:Header>

<n:orderinfo xmlns:n="http://exampleorder.org/orderinfo">

<n:priority>1</n:priority>

<n:expires>2019-06-30T16:00:00-09:00</n:expires>

</n:orderinfo>

</env:Header>

<env:Body>

<m:order xmlns:m="http://exampleorder.org/orderinfo">

<m:getorderinfo>

<m:orderno>12345</m:oderno>

</m:getorderinfo>

</m:order>

</env:Body>

SOAP commonly uses HTTP, but other protocols such as SMTP can be used.

In a SOAP-based web service, the service provider creates an API contract in the form of WSDL. WSDL lists all of the operations that web services can perform, such as providing order information, updating orders, deleting orders, and more. The service provider shares WSDL with the web service client team, using which the client generates an acceptable message format, sends data to the service provider, and gets the desired response. The web service client fills the values in the generated XML message and sends it across to the service provider with authentication details for processing. Let's look at a WSDL example:

<?xml version="1.0"?>

<definitions name="Order"

targetNamespace="http://example.com/order.wsdl"

xmlns:tns="http://example.com/ order.wsdl"

xmlns:xsd1="http://example.com/ order.xsd"

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>

<schema targetNamespace="http://example.com/ order.xsd"

xmlns="http://www.w3.org/2000/10/XMLSchema">

<element name="PlaceOrder">

<complexType>

<all>

<element name="itemID" type="string"/>

</all>

</complexType>

</element>

<element name="ItemPrice">

<complexType>

<all>

<element name="price" type="float"/>

</all>

</complexType>

</element>

</schema>

</types>

<message name="GetOrderInfo">

<part name="body" element="xsd1:GetOrderRequest"/>

</message>

<message name="GetItemInfo">

<part name="body" element="xsd1:ItemPrice"/>

</message>

<portType name="OrderPortType">

<operation name="GetOrderInfo">

<input message="tns: GetOrderInfoInput "/>

<output message="tns: GetOrderInfoOutput"/>

</operation>

</portType>

<binding name="OrderSoapBinding" type="tns:OrderPortType">

<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

<operation name="GetOrderInfo">

<soap:operation soapAction="http://example.com/GetOrderInfo "/>

<input>

<soap:body use="literal"/>

</input>

<output>

<soap:body use="literal"/>

</output>

</operation>

</binding>

<service name="OrderService">

<documentation>My first Order</documentation>

<port name="OrderPort" binding="tns:OrderBinding">

<soap:address location="http://example.com/order"/>

</port>

</service>

</definitions>

At a high level, you can see six major elements are used to define an SOA contract in WSDL:

* types: It gives data type definition to explain messages exchanged.
* message: Message represents a definition of the data being transmitted. A message has logical parts, which are associated with a definition.
* portType: It is a set of operations and contains input message and output messages.
* binding: Binding defines a protocol and data format specifications for the operations and messages defined by a particular port type.
* port: It provides an address for binding by defining a single communication endpoint.
* service: It is used to aggregate a set of related ports.

A software architect defines the WSDL and message schema, which the development team uses to generate code for client and server skeleton in the programming language of their choice for business logic implementation. The overall intent of this section is to give an overview of SOAP-based architecture.

There are various resources available on the internet, such as W3Schools tutorials to dive deep into SOAP-based service implementation for the development team, that you can explore.

The following diagram shows details about a message exchange in a web service using SOAP. Here, the web service client sends the request to the service provider who hosts the web service, and receives the response with the desired result:

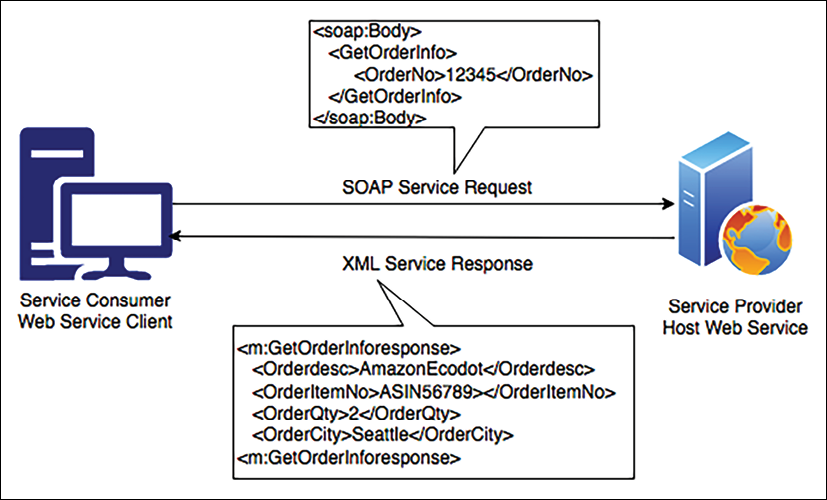


Figure 6.5: SOAP-based web service

In the preceding diagram, the client is an e-commerce website user interface. The user wants their order information and hence sends the SOAP message in XML format to the application server with the order number. The application server hosts the order service, which then responds with the customer's order details.

The implementation of a SOAP-based web service has high complexity and requires a lot of bandwidth, which can impact web application performance, such as page loading time, and any major changes in the server logic require all clients to update their code. REST was created to address SOAP-based web service problems and provide a more flexible architecture. Let's learn more about RESTful architecture and why it is becoming popular.

**RESTful web service architecture**

A **REST** or RESTful web service offers better performance due to its lightweight architecture. It allows different messaging formats such as JSON, plaintext, HTML, and XML, compared to SOAP, which only allows XML. REST is an architecture style that defines the standard for loosely coupled application design using the HTTP protocol for data transmission.

**JavaScript Object Notation** (**JSON**) is a more accessible format for data exchange in REST architecture. JSON is also lightweight and language-independent. It contains a simple key-value pair that makes it compatible with data structures defined in most programming languages.

REST focuses on the design principle for creating a stateless service. Like SOAP-based services, the web service client doesn't need to generate a complex client skeleton, but it can access the web server resources using the unique **Uniform Resource Identifier** (**URI**). The client can access RESTful resources with the HTTP protocol and perform standard operations such as GET, PUT, DELETE, and POST on the resources. Let's take a look at the differences between REST and SOAP:

|  |  |  |
| --- | --- | --- |
| Attributes | REST | SOAP |
| Design | Architectural style with an informal guideline | Predefined rules with a standard protocol |
| Message Format | JSON, YAML, XML, HTML, plaintext, and CSV | XML |
| Protocol | HTTP | HTTP, SMTP, and UPD |
| Session State | Default stateless | Default stateful |
| Security | HTTPS and SSL | Web Services Security and ACID compliance |
| Cache | Cached API calls | Cannot cache API calls |
| Performance | Fast with fewer resources | Needs more bandwidth and compute power |

Your choice of architecture design between REST and SOAP depends upon your organization's needs. The REST service offers an effective way to integrate with lightweight clients such as smartphones, while SOAP provides high security and is suitable for complex transactions. Let's learn about a reference architecture based on service-oriented design.

**Building an SOA-based e-commerce website architecture**

An e-commerce website such as [www.amazon.com](http://www.amazon.com/) has users from all parts of the world and a huge catalog with millions of products. Each product has multiple images, reviews, and videos. Maintaining such a big catalog for a global user base is a very challenging task.

This reference architecture follows SOA principles. The services are operating as independently as possible from each other. This architecture can be implemented using either SOAP-based or RESTful web architecture:

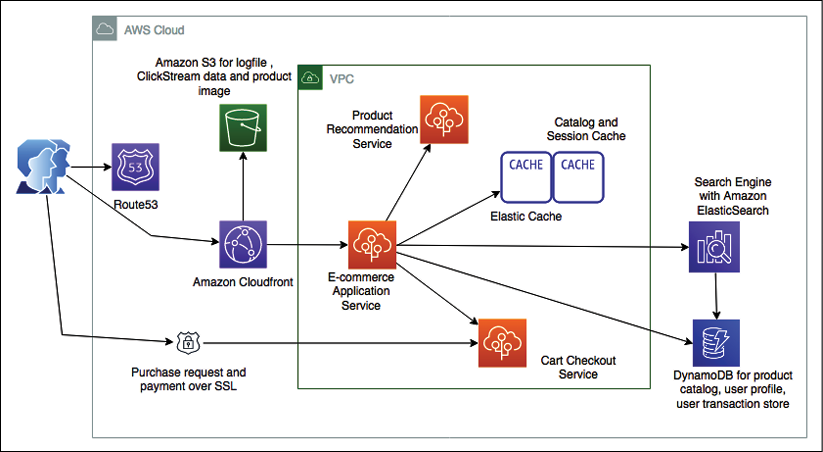


Figure 6.6: E-commerce website SOA

As shown in the preceding architecture diagram, we can take note of the following:

* When a user types a website address into the browser, the user request reaches out to the DNS server to load the website. The DNS requests for the website are routed by Amazon Route 53 to the server where the web applications are being hosted.
* The user base is global, and users continue browsing for products to purchase as the website has a large product catalog with static images and videos. A content distribution network such as Amazon CloudFront caches and delivers static assets to users.
* The catalog contents, such as static product images and videos, along with other application data, such as log files, are stored in Amazon S3.
* Users will browse the website from multiple devices, for example, they will add items in a cart from their mobile and then make a payment on a desktop. To handle user sessions, a persistent session store is required such as DynamoDB. In fact, DynamoDB is a NoSQL database where you don't need to provide a fixed schema, so it is a great storage option for product catalogs and attributes.
* To provide high performance and reduce latency, Amazon ElastiCache is used as a caching layer for the product to reduce read and write operations on the database.
* A convenient search feature is key for product sales and business success. Amazon CloudSearch helps to build scalable search capability by loading the product catalog from DynamoDB.
* A recommendation can encourage a user to buy additional products based on their browsing history and past purchases. A separate recommendation service can consume the log data stored on Amazon S3 and provide potential product recommendations to the user.
* The e-commerce application can also have multiple layers and components that require frequent deployment. AWS Elastic Beanstalk handles the auto-provisioning of the infrastructure, deploys the application, handles the load by applying auto scaling, and monitors the application.

In this section, you learned about SOA along with an architecture overview. Let's learn more about the critical aspect of modern architecture design with a serverless architecture.

**Building a serverless architecture**

In a traditional scenario, if you want to develop an application, you need to have a server where your desired operating system and required software can be installed. While you are writing your code, you need to make sure that your server is up and running. During deployment, you need to add more servers to keep up with user demand and add scaling mechanisms such as **auto scaling** to manage the desired number of servers to fulfill users' requests. In this entire situation, a lot of effort goes into infrastructure management and maintenance, which has nothing to do with your business problem.

Going serverless gives you the ability to focus on your application and write code for feature implementation without worrying about underlying infrastructure maintenance.

Serverless means there is no server required to host your code, which frees you up from auto scaling and decoupling overheads while providing a low-cost model. All heavy lifting of server management and scaling is taken care of by cloud providers.

A public cloud, such as AWS, provides several serverless services in the area of computer and data storage, which makes it easier to develop an end-to-end serverless application. When you talk about serverless, the first thing that comes to mind is AWS Lambda functions, which is a **Function as a Service** (**FaaS**) and is provided by the AWS cloud. To make your application service-oriented, Amazon API Gateway offers you the ability to put RESTful endpoints in front of your AWS Lambda functions and helps you to expose them as microservices. Amazon DynamoDB provides a highly scalable NoSQL database, which is an entirely serverless NoSQL data store, and Amazon **Simple Storage Service** (**S3**) provides serverless object data storage.

Let's take a look at an example of a reference serverless architecture in the following architecture diagram for the delivery of a secure survey:

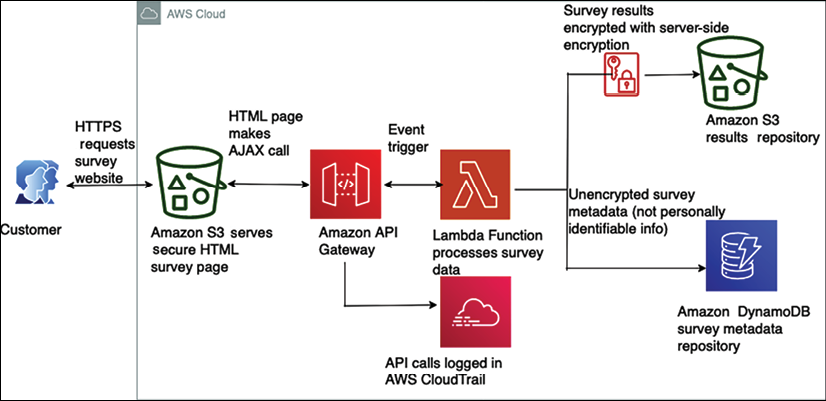


Figure 6.7: Serverless architecture for a secure survey delivery

In this example, a serverless architecture serves, delivers, and processes secure surveys, all on managed services:

1. First, a customer requests the website over HTTPS. The web page is served directly from Amazon S3.
2. The customer's survey is submitted via an AJAX call to Amazon API Gateway.
3. Amazon API Gateway logs this to Amazon CloudTrail. If a survey's results are lost, or if one of the AJAX calls includes malicious activity of some sort, these logs may be helpful in identifying and fixing the problem.
4. Amazon API Gateway then turns the AJAX call into an event trigger for an AWS Lambda function, which pulls the survey data and processes it.
5. The results of the survey are sent by the AWS Lambda function to an Amazon S3 bucket, where they are secured with *server-side encryption*.
6. Metadata from the survey, which does not include any personally identifiable information, is then written and stored in a DynamoDB table. This could be used for later queries and analysis.

Due to the increasing popularity of serverless architecture, you will see more reference architectures using serverless services as we move forward with this book. Also, now more frameworks become available to build and manage serverless applications, such as the AWS **Serverless Application Model** (**SAM**). SAM is an open-source framework for building serverless applications that provides easy syntax to create functions, APIs, and databases for serverless applications.

You can define the application model using **YAML** (**Yet Another Markup Language**). YAML is becoming highly popular and replacing JSON in many places due to its syntactical simplicity, easiness to learn, and being lightweight. During deployment, SAM transforms YAML configuration file syntax into AWS CloudFormation syntax, enabling you to build serverless applications faster.

The concept of microservices is also becoming popular with the adoption of RESTful-style architectures. Let's learn more about REST architectures and microservices in the next sections.

**Creating a microservice architecture**

Often, microservices are architected in REST-style web services and are independently scalable. This makes it easier to expand or shrink the relevant components of your system while leaving the rest untouched. A system that employs microservices can more easily withstand incidents where application availability can degrade gracefully to avoid any cascading failures. Your system becomes fault-tolerant, that is, built with failure in mind.

The clear advantage of microservices is that you have to maintain a smaller surface area of code. Microservices should always be independent. You can build each service with no external dependencies where all prerequisites are included, which reduces the inter-dependency between application modules and enables loose coupling.

The other overarching concept of microservices is **bounded contexts**, which are the blocks that combine together to make a single business domain. A business domain could be something like car manufacturing, bookselling, or social network interactions that involve a complete business process. An individual microservice defines boundaries in which all the details are encapsulated.

Scaling each service is essential while dealing with the large-scale access of applications, where different workloads have different scaling demands. Let's learn about some best practices for designing microservice architecture:

* **Create a separate data store**: Adopting a separate data store for each microservice allows the individual team to choose a database that works best for their service. For example, the team that handles website traffic can use a very scalable NoSQL database to store semi-structured data. The team handling order services can use a relational database to ensure data integrity and the consistency of transactions. This also helps to achieve loose coupling where changes in one database do not impact other services.
* **Keep servers stateless**: As you learned in the previous section, *Building stateless and stateful architecture designs*, keeping your server stateless helps in scaling. Servers should be able to go down and be replaced easily, with minimal or no need for storing state on the servers.
* **Create a separate build**: Creating a separate build for each microservice makes it easier for the development team to introduce new changes and improve the agility of the new feature release. This helps to make sure that the development team is only building code that is required for a particular microservice and not impacting other services.
* **Deploy in a container**: Deploying in a container gives you the tool to deploy everything in the same standard way. You can choose to deploy all microservices in the same way regardless of their nature using containers. You will learn more about container deployment in the *Deploying an application with a container* section.
* **Go serverless**: Try to use a serverless platform or a leveraging function with service capability, such as AWS Lambda, when your microservices are not too complex. Serverless architecture helps you to avoid infrastructure management overhead.
* **Blue-green deployment**: The better approach is to create a copy of the production environment. Deploy the new feature and route a small percentage of the user traffic to make sure the new feature is working as per the expectation in a new environment. After that, increase the traffic in the new environment until the entire user base is able to see the new feature. You will learn more about blue-green deployment in *Chapter 12*, *DevOps and Solution Architecture Framework*.
* **Monitor your environment**: Good monitoring is the difference between reacting to an outage and proactively preventing an outage with proper rerouting, scaling, and managed degradation. To prevent any application downtime, you want services to offer and push their health status to the monitoring layer, because what knows more about status than the service itself? Monitoring can be done in many ways, such as with plugins, or by writing to a monitoring API.

While microservice architectures have various advantages, a modular approach comes with the overhead of managing more infrastructure. You need to carefully choose the tools to help you manage and scale multiple modules in parallel. While designing microservice architecture, wherever possible, try to use serverless platforms, which will help mitigate infrastructure and operation overhead. Let's take a look at a microservice-based reference architecture for a real-time voting application.

**Real-time voting application reference architecture**

A microservice-based architecture is illustrated in the following diagram, representing a real-time voting application, where small microservices process and consolidate user votes. The voting application collects individual user votes from each mobile device and stores all the votes in a NoSQL Amazon DynamoDB database.

Finally, there is application logic in the AWS Lambda function, which aggregates all of the voting data cast by users to their favorite actor and returns the final results:

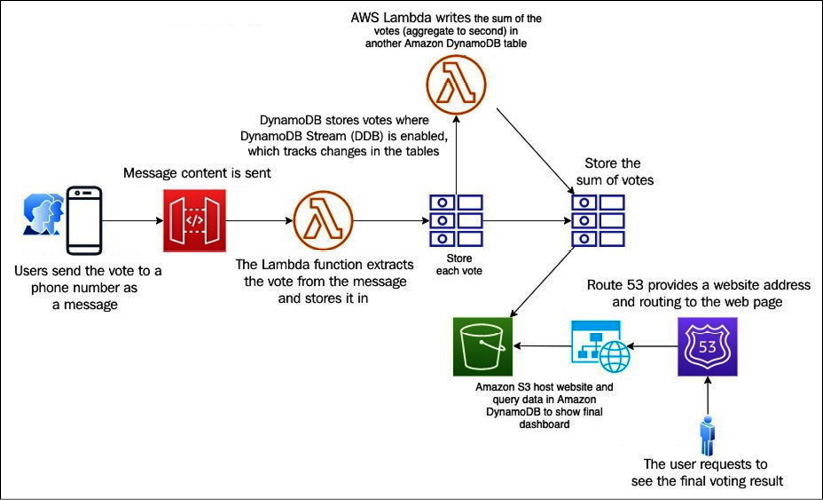


Figure 6.8: Microservice-based real-time voting application architecture

In the preceding architecture, the following things are happening:

1. Users text a vote to a phone number or a short code provided by a third party such as *Twilio*.
2. The third party is configured to send the content of the message to an endpoint created by Amazon API Gateway, which then forwards the response to a function built in AWS Lambda.
3. This function extracts the vote from the message content and writes the result and any metadata into a table in Amazon DynamoDB.
4. This table has DynamoDB Streams enabled, which allows you to track changes to your tables on a rolling basis.
5. After the update, DynamoDB Streams notifies a second AWS Lambda function, which has the application logic to aggregate the votes (to every second) and writes them back to another DynamoDB table. The second table only stores the sum of the votes for each category.
6. A dashboard to display a summary of votes is created using HTML and JavaScript and hosted as a static website in Amazon S3. This page uses the AWS JavaScript SDK to query the aggregate Amazon DynamoDB table and display the voting results in real time.
7. Finally, Amazon Route 53 is used as a DNS provider to create a hosted zone pointing to a custom domain name in the Amazon S3 bucket.

This architecture is not only microservice-based but also serverless. Using microservices, you can create applications made of small independent components, which constitute smaller parts to iterate. Microservice-based architecture means that the cost, size, and risk of change reduces, increasing the rate of change.

Message queues play a vital role in achieving accurate loose coupling and help to avoid application throttling. A queue allows secure and reliable communication between components. Let's learn more about queue-based architecture in the next section.

**Building a queue-based architecture**

In the previous section, you learned about microservice design using RESTful architecture. The RESTful architecture helps your microservice to be easily discoverable, but what happens if your service goes down? This is a contemporary architecture, where your client service waits for a response from the host service, which means that the HTTP request blocks the API. Sometimes, your information may be lost due to the unavailability of a downstream service. In such cases, you must implement some retry logic in order to retain your information.

A queue-based architecture provides a solution to this problem by adding message queues between services, which holds information on behalf of services. The queue-based architecture provides fully asynchronous communication and a loosely coupled architecture. In a queue-based architecture, your information is still available in the message. If a service crashes, the message can get the process as soon as the service becomes available. Let's learn some of the terminology of a queue-based architecture:

* **Message**: A message has two parts—the header and the body. The header contains metadata about the message, while the body contains the actual message.
* **Queue**: The queue holds the messages that can be used when required.
* **Producer**: A service that produces a message and publishes it to the queue.
* **Consumer**: A service that consumes and utilizes the message.
* **Message broker**: Helps to gather, route, and distribute messages between the producer and consumer.

Let's learn about some typical queue-based architecture patterns to get an idea of how they work.

**Queuing chain pattern**

A queuing chain pattern is applied when sequential processing needs to run on multiple systems that are linked together. Let's understand the *queuing chain pattern* using the example of an image-processing application. In an image-processing pipeline, sequential operations of capturing the image and storing it on a server, running a job to create different-resolution copies of the image, watermarking the image, and thumbnail generation are tightly linked to each other. A failure in one part can cause the entire operation to be disrupted.

You can use queues between various systems and jobs to remove a single point of failure and design true loosely coupled systems. The queuing chain pattern helps you to link different systems together and increase the number of servers that can process the messages in parallel. If there is no image to process, you can configure **auto scaling** to terminate the excess servers.

The following diagram shows a queuing chain pattern architecture. Here, the queue provided by AWS is called Amazon **Simple Queue Service** (**SQS**):

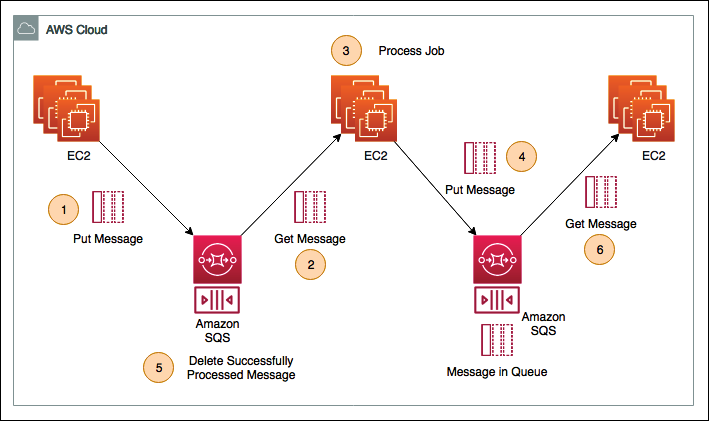


Figure 6.9: Queuing chain pattern architecture

The preceding architecture has the following steps:

1. As soon as the raw image is uploaded to the server, the application needs to watermark all of the images with the company's logo. Here, a fleet of Amazon EC2 servers is running batch jobs to watermark all the images and push the processed image into the Amazon SQS queue.
2. The second fleet of Amazon EC2 servers pulls the watermarked images from the Amazon SQS queue.
3. The second fleet of EC2 workers processes the image and creates multiple variations with different resolutions.
4. After encoding the images, the EC2 workers push the message into another Amazon SQS queue.
5. As the image is processed, the job deletes the message from the previous queue to make space.
6. The final fleet of EC2 servers gets encoded messages from the queue and creates thumbnails along with the copyright.

The benefits of this architecture are as follows:

* You can use loosely coupled asynchronous processing to return responses quickly without waiting for another service acknowledgment.
* You can structure the system through the loose coupling of Amazon EC2 instances or containers using Amazon SQS.
* Even if the Amazon EC2 instance fails, a message remains in the queue service. This enables processing to be continued upon recovery of the server and creates a system that is robust to failure.

You may get fluctuations in application demand that can cause unexpected message loads. Automating your workload as per the queue message load will help you to handle any fluctuations. Let's learn more about using the *job observer pattern* to handle such automation next.

**Job observer pattern**

Queuing chain patterns help you design a loosely coupled architecture, but how will you handle workload spike? In case of request fluctuation, you need to adjust your processing power based on user demand, which can be addressed by the job observer pattern.

In the job observer pattern, you can create an auto scaling group, based upon the number of messages in the queue to process. The job observer pattern helps you to maintain performance through increasing or decreasing the number of server instances used in job processing.

The following diagram depicts the job observer pattern:

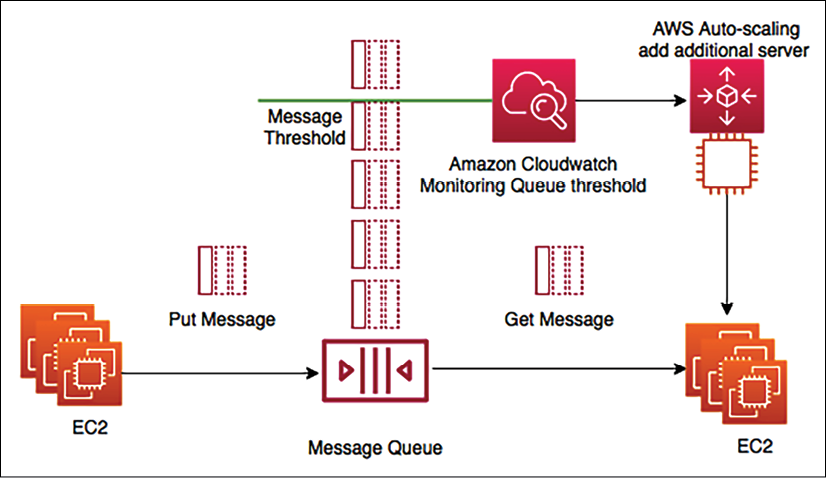


Figure 6.10: Job observer pattern architecture

In the preceding architecture, the first fleet of Amazon EC2 servers is on the left-hand side, running batch jobs and putting messages in the queue, for example, image metadata. The second fleet of EC2 servers on the right-hand side is consuming and processing those messages, for example, image encoding. As the message reaches a certain threshold, Amazon CloudWatch triggers auto scaling to add the additional server in the consumer fleet to speed up the job processing. Auto scaling also removes additional servers when the queue depth goes below the threshold.

The job observer pattern computes scale with job size, providing efficiency and cost savings. The job observer pattern architecture allows the job to be completed in a shorter timeframe. The process is resilient, which means job processing doesn't stop if a server fails.

While queue-based architecture provides loose coupling, it works mostly on the **Asynchronous Pull** method, where the consumer can pull messages from the queue when they are available.

Often, you need to drive communication between various architecture components where one event should trigger other events. Let's learn more about event-driven architecture in the next section.

**Creating an event-driven architecture**

Event-driven architecture helps you to chain a series of events together to complete a functional flow. For example, when you are making a payment to buy something on a website, you are expecting to get your order invoice generated and to get an email as soon as the payment is complete. Event-driven architecture helps to rope in all of these events so that making a payment can trigger another task to complete the order flow. Often, you will see message queues, which you learned about in the previous section, as the central point while talking about event-driven architecture. Event-driven architecture can also be based on the publisher/subscriber model or the event stream model.

**Publisher/subscriber model**

In the **publisher**/**subscriber** (**pub**/**sub**) model, when an event is published, a notification is sent to all subscribers, and each subscriber can take the necessary action as per their requirements of data processing. Let's take an example of a photo studio application, which enriches a photo with different filters and sends a notification to the user. The following architecture depicts a pub/sub model:

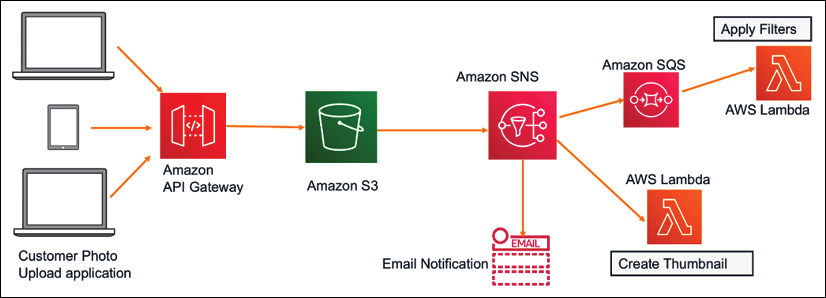


Figure 6.11: Photo studio application pub/sub event-driven architecture

In the preceding diagram, you will notice the following things:

1. The user first uploads the picture to an **Amazon S3** bucket using a web/mobile application.
2. The **Amazon S3** bucket then sends a notification to Amazon **Simple Notification Service** (**SNS**). **Amazon SNS** is a message topic with the following subscribers:
   * Here, the first subscriber is using the email service, and as soon as the photo upload is complete, an email is sent to the user
   * The second subscriber is using an **Amazon SQS** queue, which gets the message from the **Amazon SNS** topic and applies various filters in code written in AWS Lambda to improve the image quality
   * The third subscriber is using the direct **AWS Lambda** function, which creates the image thumbnail

In this architecture, Amazon S3 publishes the message to the SNS topic as a producer, which is consumed by multiple subscribers. Additionally, as soon as the message comes to SQS, it triggers an event for the Lambda function to process images.

**Event stream model**

In the event stream model, the consumer can read from the continuous flow of events coming from the producer. For example, you can use the event stream to capture the continuous flow of a clickstream log and also send an alert if there are any anomalies detected, as shown in the following architecture diagram:

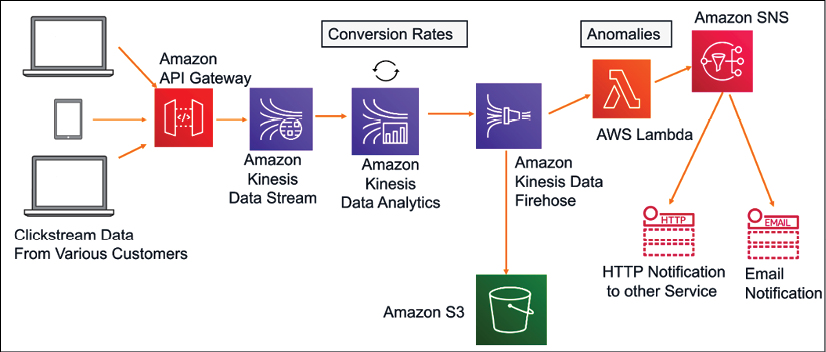


Figure 6.12: Clickstream analysis event stream architecture

Amazon Kinesis is a service that is used to ingest, process, and store continuous streaming data. In the preceding diagram, various customers clicking on e-commerce applications from web and mobile applications produce a stream of click events.

These clickstreams are sent to analytics applications using **Amazon API Gateway** for real-time analytics. In this analytics application, **Kinesis Data Analytics** calculates **Conversion Rates** over a certain period of time, for example, the number of people that ended up making a purchase in the last five minutes. After aggregating data in real time, **Amazon Kinesis Data Analytics** sends the results to **Amazon Kinesis Data Firehose**, which stores all the data files in **Amazon S3** storage for further processing as needed.

A Lambda function reads from the event stream and starts examining the data for **Anomalies**. As anomalies in the conversion rates are detected, the **AWS Lambda** function sends a notification on email for the campaign team to be notified. In this architecture, the event stream is occurring continuously, and **AWS Lambda** is reading from the stream for a specific event.

You should use event-driven architecture to decouple the producer and consumer and keep the architecture extendable so that a new consumer can be integrated at any time. This provides a highly scalable and distributed system with each subsystem having an independent view of events. However, you need to apply a mechanism to avoid duplicate processing and error message handling.

To achieve good application performance, caching is an important factor and it can be applied at every architecture layer and in pretty much any architecture component. Let's learn more about cache-based architecture in the next section.

**Building a cache-based architecture**

Caching is the process of temporarily storing data or files in an intermediary location between the requester and the permanent storage, for the purpose of making future requests faster and reducing network throughput. Caching increases the application speed and lowers the cost. It allows you to reuse previously retrieved data. To increase application performance, caching can be applied at various layers of the architecture, such as the web layer, application layer, data layer, and network layer.

Normally, the server's **random access memory** (**RAM**) and in-memory cache engines are utilized to support application caching. However, if caching is coupled to a local server, then the cache will not be persisting data, in case of a server crash. Now, most of the applications are in a distributed environment, so it's better to have a dedicated caching layer that should be independent of the application life cycle. If you applied horizontal scaling to your application, all servers should be able to access the centralized caching layer to achieve the best performance.

The following diagram depicts the mechanism of caching in various layers of solution architecture:

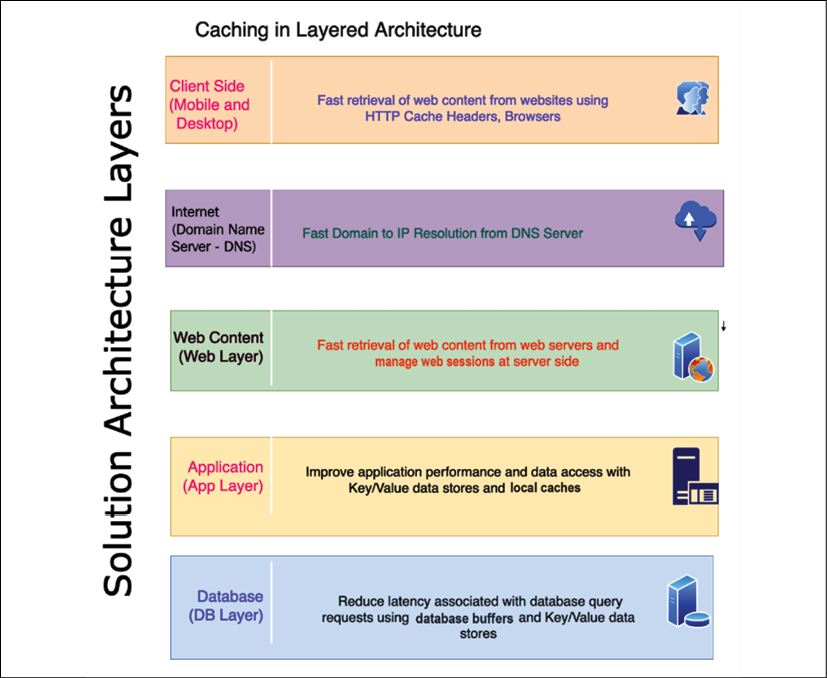


Figure 6.13: Caching at the architecture layers

As shown in the preceding diagram, the following are the caching mechanisms at each layer of architecture:

* **Client side**: Client-side caching is applied to user devices such as mobile and desktop. This caches the previously visited web content to respond faster to a subsequent request. Each browser has its own caching mechanism. HTTP caching makes the application faster by caching content at the local browser. The cache-control HTTP header defines browser caching policies for both the client request and server response. These policies define where the content should be cached and how long it will persist, which is known as **time to live** (**TTL**). Cookies are another method used to store information at the client machine in order to respond to the browser faster.
* **DNS cache**: When a user types the website address over the internet, the public **Domain Name System** (**DNS**) server looks up the IP address. Caching this DNS resolution information will reduce the website's load time. DNS information can be cached to a local server or browser after the first request and any further requests to that website will be faster.
* **Web caching**: Much of the request involves retrieving web content such as images, video, and HTML pages. Caching these assets near to the user's location can provide a much faster response for a page load. This also eliminates disk read and server load time. A **content distribution network** (**CDN**) provides a network of edge locations where static content such as high-resolution images and videos can be cached. It's very useful for reading heavy applications such as games, blogs, e-commerce product catalog pages, and more. The user session contains lots of information regarding user preference and their state. It provides a great user experience to store the user's session in its own key-value store for quick user response.
* **Application caching**: At the application layer, caching can be applied to store the result of a complex repeated request to avoid business logic calculations and database hits. Overall, it improves application performance and reduces the load on the database and infrastructure.
* **Database caching**: Application performance highly depends upon speed and throughput provided by the database. Database caching allows you to increase database throughput significantly and lower data retrieval latency. A database cache can be applied in front of any kind of relational or non-relational database. Some database providers integrate caching, while applications handle local caching.

**Redis** and **Memcached** are the most popular caching engines. While Memcached is faster (it is good for low-structure data and stores data in a key-value format), Redis is a more persistent caching engine and is capable of handling complex data structures required for an application such as a gaming leaderboard; you will learn more details in the section *Memcached versus Redis* in this chapter. Let's learn about a few more caching design patterns.

**Cache distribution pattern in a three-tier web architecture**

Traditional web hosting architecture implements a standard three-tier web application model that separates the architecture into the presentation, application, and persistence layers.

As shown in the following architecture diagram, caching is applied at the web, application, and database layers:

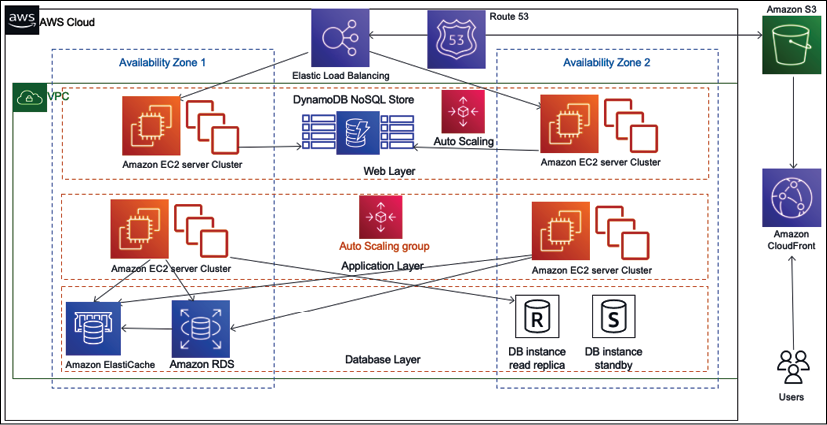


Figure 6.14: Cache distribution pattern architecture

One of the ways you can offload your web page is through caching. In caching patterns, your goal is to try to hit the backend as little as possible. You can write an application where you can cache images, JavaScript, or even full pages to provide a better experience for your users. As shown in the preceding diagram, caching is applied to the various layers of architecture:

* **Amazon Route 53** provides DNS services to simplify domain management and to help cache DNS-to-IP mapping.
* **Amazon S3** stores all static content such as high-resolution images and videos.
* **Amazon CloudFront** provides edge caching for high-volume content. It also uses these cache-control headers to determine how frequently it needs to check the origin for an updated version of that file.
* **Amazon DynamoDB** is used for session stores in which web applications cache to handle user sessions.
* **Elastic Load Balancing** spreads traffic to web server **Auto Scaling** groups in this diagram.
* **Amazon ElastiCache** provides caching services for the app, which removes the load from the database tier.

In general, you only cache static content; however, dynamic or unique content affects the performance of your application. Depending on the demand, you might still get some performance gain by caching the dynamic or unique content. Let's take a look at a more specific pattern.

**Rename distribution pattern**

When using a **CDN** such as Amazon CloudFront, you store frequently used data in an edge location near to the user for fast performance. Often, you set up **TTL** (**Time To Live**) in the CDN for your data, which means the edge location will not query back to the server for updated data until the TTL expires. TTL is the time that an object is stored in a caching system before it's deleted or refreshed. You may have situations where you need to update CDN cached content immediately, for example, if you need to correct the wrong product description.

In such a situation, you can't wait for the file's TTL to expire. The rename distribution pattern helps you to update the cache as soon as new changes are published so that the user can get updated information immediately. The following diagram shows the *rename distribution pattern*:

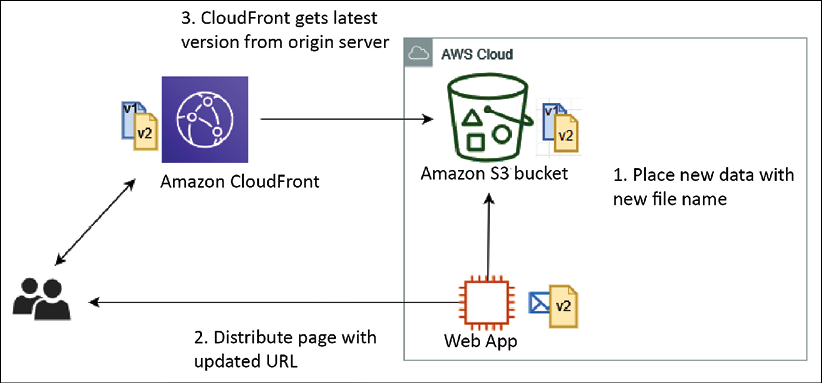


Figure 6.15: Rename distribution pattern architecture

As shown in the preceding diagram, using the rename distribution pattern with the cache distribution pattern helps to solve the update issue. With this pattern, instead of overwriting the file in the origin server and waiting for the TTL in CloudFront to expire, the server uploads the updated file with a new filename and then updates the web page with the new URL. When the user requests original content, CloudFront has to fetch it from the origin and can't serve the obsolete file that's already cached.

However, you have the option to invalidate the old file immediately, but that will cost more, so it's better to put a new version of the file for the CDN to pick immediately. Again, you have to update the URL in the application to pick up a new file, adding some overhead compared to the invalidation option. It would be best if you make a decision based on your business requirement and budget.

If you don't want to use a CDN for a user base distributed across a country, instead, you can use the proxy cache server. Let's learn more about it in the next section.

**Cache proxy pattern**

You can increase your application performance significantly by adding a cache layer. In a cache proxy pattern, static content or dynamic content is cached upstream of the web app server. As shown in the following architectural diagram, you have a caching layer in front of the web application cluster:

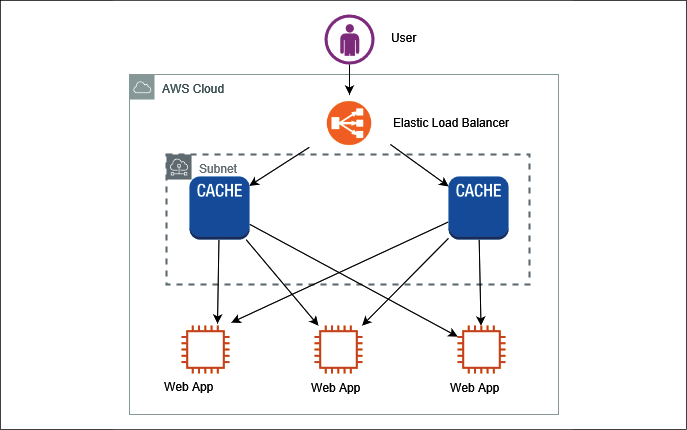


Figure 6.16: Cache proxy pattern architecture

In the preceding diagram, for high-performance delivery, cache content is delivered by the cache server. A few benefits of cache proxy patterns are as follows:

* Cache proxy patterns help you to deliver content using the cache, which means no modification is needed at the web server or application server level.
* They reduce the load of content generation for dynamic content in particular.
* You have the flexibility to set up a cache at the browser level such as in HTTP headers, URLs, cookies, and more. Alternatively, you can cache information in the cache layer if you don't want to store it at the browser level.

In the cache proxy pattern, you need to make sure that you maintain multiple copies of the cache to avoid the single point of failure. Sometimes, you may want to serve your static content from both the server and CDN, each of which requires a different approach. Let's deep dive into this hybrid situation in the next section.

**Rewrite proxy pattern**

Sometimes, you want to change the access destinations of static website content such as images and videos, but don't want to make changes to the existing systems. You can achieve this by providing a proxy server using rewrite proxy patterns. To change the destination of static content to other storage such as a content service or internet storage, you can use a proxy server in front of the web server fleet. As shown in the following architecture diagram, you have a proxy server in front of your application layer, which helps to change the content delivery destination without modifying the actual application:

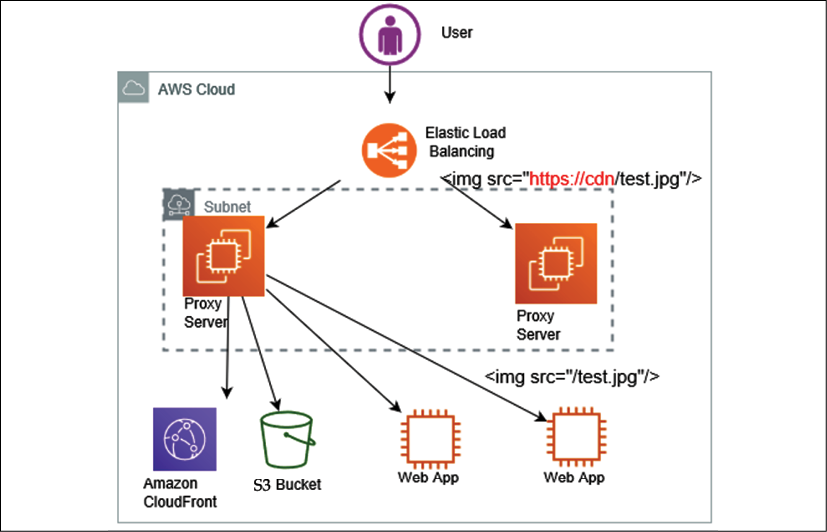


Figure 6.17: Rewrite proxy pattern architecture

As shown in the preceding diagram, to accomplish a rewrite proxy pattern, place the proxy server in front of the currently running system. You can construct a proxy server using software such as **Apache** or **NGINX**. The following are the steps to build a rewrite proxy pattern:

1. Put a running proxy server on an EC2 instance, which is able to overwrite the content between the **load balancer** and the storage service such as **Amazon S3**, which stores the static content
2. Add to the proxy server rules for overwriting URLs within the content. These rules will help **Elastic Load Balancing** (**ELB**) to point to a new location, as shown in the preceding diagram, which redirects the proxy server rule from https://cdn/test.jpg to /test.jpg
3. As required, apply auto scaling to the proxy servers by configuring a number of minimum and maximum proxy servers as per the application load

In this section, you learned about various ways to handle caching for static content distribution over the network. However, caching at the application layer is very important for improving application performance for the overall user experience. Let's learn more about the app caching pattern to handle dynamic user data delivery performance.

**App caching pattern**

When it comes to applying caching to applications, you want to add a cache engine layer between your application servers and the database. The app caching pattern allows you to reduce the load on the database as the most frequent query is served from the caching layer. The *app caching pattern* improves overall application and database performance. As shown in the following diagram, you can see the caching layer applied between the application layer and the database layer:

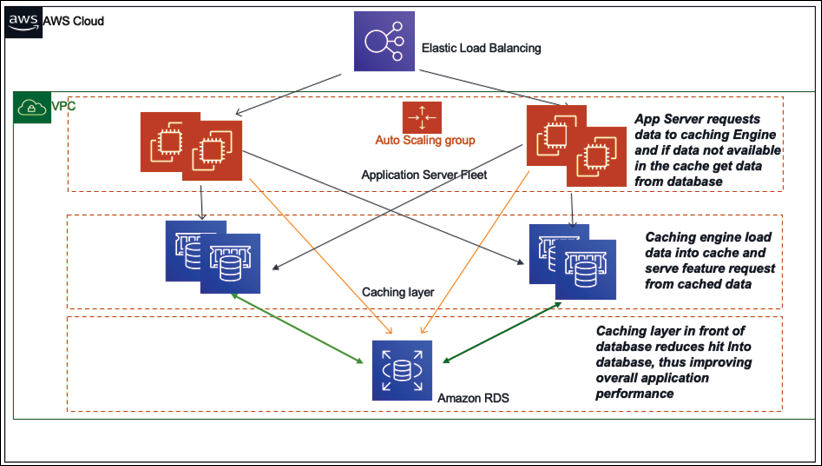


Figure 6.18: Application caching pattern architecture

As shown in the preceding diagram, based on your *data access pattern*, you can use either **lazy caching** or **write-through**. In lazy caching, the cache engine checks whether the data is in the cache and, if not, gets it from the database and keeps it in the cache to serve future requests. Lazy caching is also called the **cache aside pattern**.

In the **write-through** method, data is written in the cache and in the data store at the same time. If the data gets lost from the cache, then it can get it again from the database. Write-through is used mainly in application-to-application situations where users are writing a product review (which always needs to load on the product page). Let's learn more about the popular caching engines *Redis* and *Memcached*.

**Memcached versus Redis**

Redis and Memcached are two popular caching engines used in application design. Often, the Redis cache engine is required for more complex application caching needs such as creating a leaderboard for a game. However, Memcached is more high-performing and is helpful for handling heavy application loads. Each caching engine has its own pros and cons. Let's take a look at the major differences between them, which will help you to make a decision of which to use:

|  |  |
| --- | --- |
| Memcached | Redis |
| Offers multithreading | Single-threaded |
| Able to use more CPU cores for faster processing | Unable to utilize multi-core processor, which results in comparatively slow performance |
| Supports key-value style data | Supports complex and advanced data structures |
| Lacks data persistence; loses the data stored in cache memory in the event of a crash | Data can persist using built-in read replicas with failover |
| Easy maintenance | More complexity involved owing to the need to maintain the cluster |
| Good to cache flat strings such as flat HTML pages, serialized JSON, and more | Good to create a cache for a gaming leaderboard, a live voting app, and more |

Overall, if you need to decide which engine to use, base it on a use case that can justify using Redis or Memcached. Memcached is simple and has lower maintenance, and it is typically preferred when your cache doesn't need the advanced features that Redis offers. However, if you need the advantage of data persistence, advanced data types, or any of the other features listed, then Redis is the best solution.

When implementing caching, it's essential to understand the validity of data that needs to be cached. If the cache hit rate is high, that means the data is available in the cache when required. For a higher cache hit ratio, offload the database by reducing direct queries; this also improves the overall application performance. A cache miss occurs when data is not present in the cache, which increases the load in the database. The cache is not a large data store, so you need to set the TTL and evict the cache as per your application needs.

As you have seen in this section, there are multiple benefits of applying caches, including application performance improvement, the ability to provide predictable performance, and the reduction in database cost.

Let's learn about some more application-based architecture that demonstrates the principle of loose coupling and constraint handling.

**Understanding the circuit breaker pattern**

It's common for a distributed system to make a call to other downstream services, and the call could fail or hang without response. You will often see code that retries the failed call several times. The problem with a remote service is that it could take minutes or even hours to correct, and an immediate retry might end up in another failure. As a result, end users wait longer to get an error response while your code retries several times. This retry function would consume the threads, and it could potentially induce a cascading failure.

The circuit breaker pattern is about understanding the health of downstream dependencies. It detects when those dependencies are unhealthy and implements logic to gracefully fail requests until it detects that they are healthy again. The circuit breaker can be implemented using a persistence layer to monitor healthy and unhealthy requests over the past request interval.

If a defined percentage of requests observe an unhealthy behavior over the past interval or over a total count of exceptions, regardless of percentage, the circuit is marked as open. In such a situation, all requests throw exceptions rather than integrate with the dependency for a defined timeout period. Once the timeout period has subsided, a small percentage of requests try to integrate with the downstream dependency, to detect when health has returned. Once a sufficient percentage of requests are healthy again over an interval, or no errors are observed, the circuit closes again, and all the requests are allowed to integrate as they usually would thoroughly.

The implementation decisions involve the state machine to track/share the healthy/unhealthy request counts. The states of services can be maintained in DynamoDB, Redis/Memcached, or another low-latency persistence store.

**Implementing the bulkheads pattern**

Bulkheads are used in ships to create separate watertight compartments that serve to limit the effect of failure, ideally preventing the ship from sinking. If water breaks through the hull in one compartment, the bulkheads prevent it from flowing into other compartments, limiting the scope of the failure.

The same concept is useful to limit the scope of failure in the architecture of large systems, where you want to partition your system to decouple dependencies between services. The idea is that one failure should not cause the entire system to fail, as shown in the following diagram:

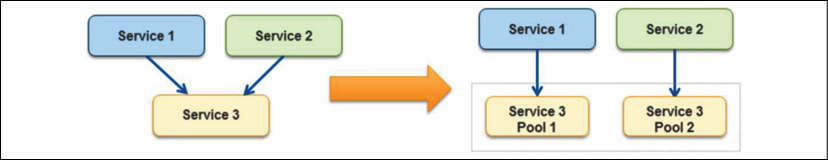


Figure 6.19: Bulkhead pattern

In the bulkhead pattern, it's better to isolate the element of the application into the pool for service, which has a high dependency; so, if one fails, others continue to serve upstream services. In the preceding diagram, **Service 3** is partitioned into two pools from a single service. Here, if **Service 3** fails, then the impact of either **Service 1** or **Service 2** depends on their dependency on the pool, but the entire system does not go down. The following are the major points to consider when introducing the bulkhead pattern in your design, especially for the shared service model:

* Save part of the ship, which means your application should not shut down due to the failure of one service.
* Decide whether less-efficient use of resources is okay. Performance issues in one partition should not impact the overall application.
* Pick a useful granularity. Don't make the service pools too small; make sure they are able to handle application load.
* Monitor each service partition performance and adhere to the SLA. Make sure all of the moving parts are working together and test the overall application when one service pool is down.

You should define a service partition for each business or technical requirement. You should use this pattern to prevent the application from cascading failure and isolating critical consumers from the standard consumer.

Often, legacy application servers have a configuration with hardcoded **Internet Protocol** (**IP**) addresses or **Domain Name Server** (**DNS**) names. Making any server change for modernization and upgrade requires making changes in the application and revalidating it. In these cases, you don't want to change the server address. Let's learn how to handle such a situation with a floating IP in the next section.

**Creating a floating IP pattern**

It's common that monolithic applications have lots of dependencies on the server where they are deployed. Often, application configuration and code have hardcoded parameters based on the server DNS name and IP address. Hardcoded IP configuration creates challenges if you want to bring up a new server in case of an issue with the original server. Additionally, you don't want to bring down the entire application for the upgrade, which may cause significant downtime.

To handle such a situation, you need to create a new server keeping the same server IP address and DNS name. This can be achieved by moving the network interface from a problematic instance to the new server. The network interface is generally based on a **Network Interface Card** (**NIC**), which facilitates communication between servers over a network. It can be in the form of hardware or software. Moving the network interface means that now your new server assumes the identity of the old server. With that, your application can live with the same DNS and IP address. It also allows easy rollback by moving the network interface to the original instance.

The public cloud (for example, AWS) made it easy by providing an **Elastic IP** (**EIP**) and **Elastic Network Interface** (**ENI**). If your instance fails and you need to push traffic to another instance with the same public IP address, then you can move the EIP address from one server to another, as shown in the following architecture diagram:

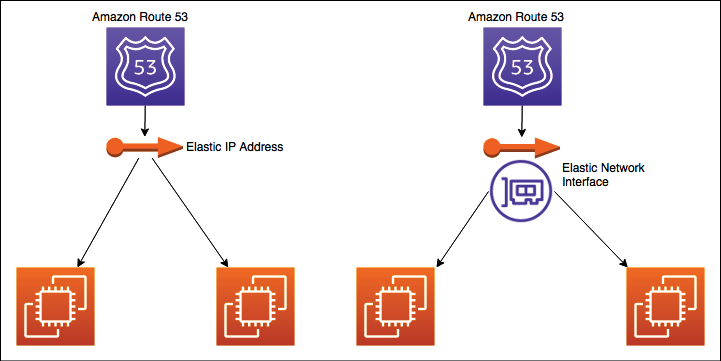


Figure 6.20: Floating IP and interface pattern

Since you are moving EIP, the DNS may not need to update. EIP can move your server's public IP across instances. If you need to move both public and private IP addresses, then use a more flexible approach such as ENI, as shown on the right of the preceding diagram. ENI can move across instances, and you can use the same public and private address for traffic routing or application upgrades.

So far, you have learned about multiple architecture patterns where applications are deployed in the virtual machine. However, in many cases, you may not be able to utilize the virtual machine fully. To optimize your utilization further, you can choose to deploy your application in containers. Containers are most suitable for microservice deployment. Let's learn more about container-based deployment in the next section.

**Deploying an application with a container**

As many programming languages are invented and technologies evolve, this creates new challenges. There are different application stacks that require different hardware and software deployment environments. Often, there is a need to run applications across different platforms and migrate from one to another platform. Solutions require something that can run anything everywhere and is consistent, lightweight, and portable.

Just as shipping containers standardized the transport of freight goods, software containers standardize the transport of applications. Docker creates a container that contains everything a software application would need to be able to run all of its files, such as filesystem structure, daemons, libraries, and application dependencies. Containers isolate software from its surrounding development and staging environments. This helps to reduce conflicts between teams running different software on the same infrastructure.

VMs isolate at the operating system level, and containers isolate at the kernel level. This isolation allows several applications to run on a single-host operating system, and yet still have their filesystem, storage, RAM, libraries, and, mostly, their own *view* of the system:

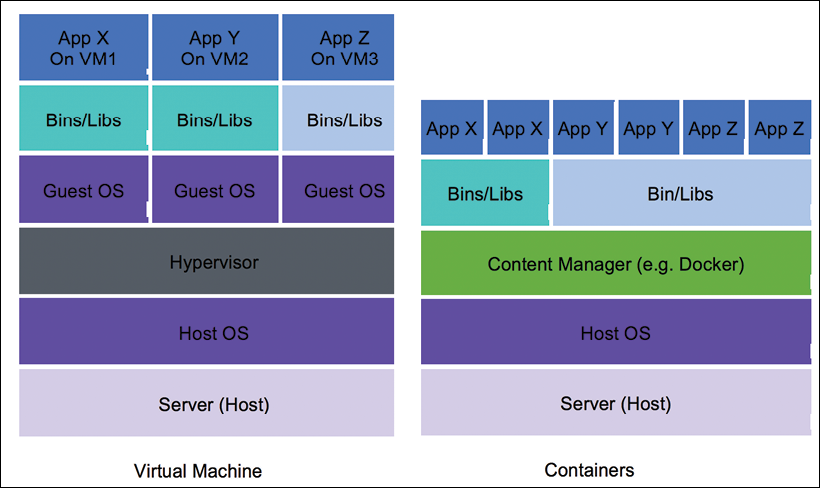


Figure 6.21: Virtual machines and containers for application deployment

As shown in the preceding diagram, multiple applications are deployed in a single virtual machine using containers. Each application has its runtime environment, so you can run many individual applications while keeping the same number of servers. Containers share a machine's operating system kernel. They start instantly and use less computing time and RAM. Container images are constructed from the filesystem layers and share standard files. Shared resourcing minimizes disk usage, and container image downloads are much faster. Let's take a look at why containers are becoming more popular, along with their benefits.

**The benefit of containers**

Customers often ask these questions when it comes to containers:

* Why do we need containers when we have instances?
* Don't instances already provide us with a level of isolation from the underlying hardware?

While the preceding questions are valid, several benefits accrue from using a system such as **Docker**. One of the key benefits of Docker is that it allows you to fully utilize your virtual machine resources by hosting multiple applications (on distinct ports) in the same instance.

Docker uses certain features of the Linux kernel, namely kernel namespaces and groups, to achieve complete isolation between each Docker process, as indicated in the following architecture diagram:

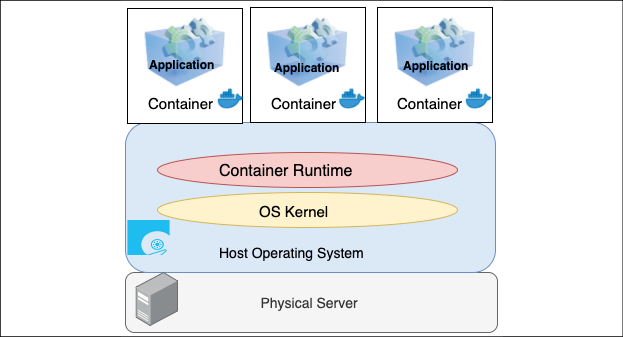


Figure 6.22: Container layer in application infrastructure

As shown in the preceding diagram, it's possible to run two or more applications that require different versions of the Java runtime on the same machine, as each Docker container has its version of Java and the associated libraries installed. In turn, the container layer in the application infrastructure makes it easier to decompose your applications into microservices that can run side by side on the same instance. Containers have the following benefits:

* **Portable runtime application environment**: Containers provide platform-independent capabilities, where you build your application once and deploy it anywhere regardless of the underlying operating system
* **Faster development and deployment cycles**: Modify the application and run it anywhere with quick boot time, typically within seconds
* **Package dependencies and application in a single artifact**: Package the code, library, and dependencies together to run the application in any operating system
* **Run different application versions**: Applications with different dependencies run simultaneously in a single server
* **Everything can be automated**: Container management and deployment are done through scripting, which helps to save cost and human error
* **Better resource utilization**: Containers provide efficient scaling and high availability and multiple copies of the same microservice container can be deployed across servers for your application
* **Easy to manage the security aspect**: Containers are platform-specific rather than application-specific

Container deployment is becoming very popular due to its benefits. There are multiple ways to orchestrate containers. Let's look at container deployment in more detail next.

**Container deployment**

Complex applications with multiple microservices can be quickly deployed using container deployment. The container makes it easier to build and deploy the application more quickly as the environment is the same. Build the container in development mode, push to test, and then release to production. For hybrid cloud environments, container deployment is very useful. Containers make it easier to keep environments consistent across microservices. As microservices aren't always very resource-consuming, they can be placed together in a single instance to reduce cost.

Sometimes, customers have short workflows that require a temporary environment setup. Those environments may be queue systems or continuous integration jobs, which don't always utilize server resources efficiently. Container orchestration services such as Docker and Kubernetes can be a workaround, allowing them to push and pop containers onto the instance.

Docker's lightweight container virtualization platform provides tools to manage your applications. Its standalone application can be installed on any computer to run containers. Kubernetes is a container orchestration service that works with Docker and another container platform. Kubernetes allows automated container provisioning and handles security, networking, and scaling aspects diligently.

Containers help the enterprise to create more cloud native workloads, and public cloud providers such as AWS extend services to manage Docker containers and Kubernetes.

The following diagram shows Docker's container management using Amazon **Elastic Container Service** (**ECS**), providing a fully managed elastic service to automate the scaling and orchestration of Docker containers:

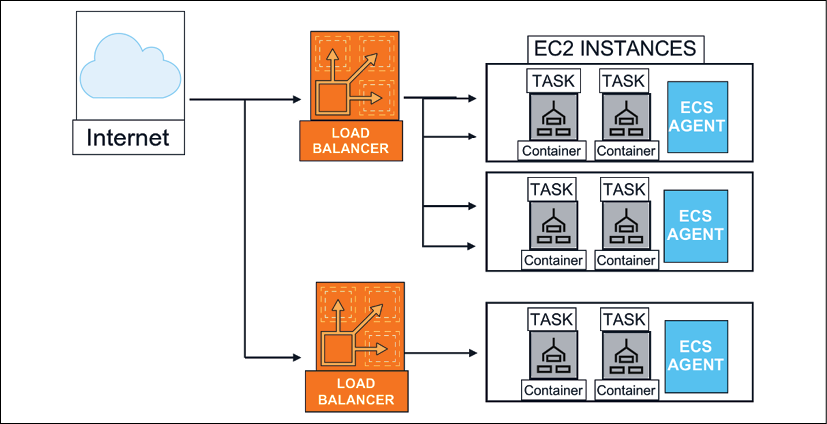


Figure 6.23: Container deployment architecture

In the preceding diagram, multiple containers are deployed in a single Amazon EC2 virtual machine, which is managed through Amazon ECS and facilitates the agent communication service and cluster management. All user requests are distributed using a load balancer among the containers. Similarly, AWS provides Amazon **Elastic Kubernetes Service** (**EKS**) to manage containers using Kubernetes.

Containers are a broad topic, and, as a solution architect, you need to be familiar with all of the available options. This section provides an overview of containers. However, you will need to deep dive further if you choose to utilize containers for your microservice deployment. Let's look at a container-based architecture in the next section.

**Building container-based architecture**

As you learned in the previous section, containerization helps create environments for repeatable and scalable applications. To start container adoption, you need to identify a pilot workload managed through container orchestrations. You can take existing microservice components and deploy them in containers. After identifying gaps and operational needs, you can define a migration strategy to move your workload to containers.

Like any other changes, container migrations come with challenges if your applications are not designed to run in a container environment. As applications often persist files to local storage and make stateful sessions, container migration needs to address these requirements.

For container platforms, you can make choices; you can choose Docker, OpenShift, Kubernetes, and so on. However, Kubernetes is becoming an increasingly popular open-source container orchestrator. Public cloud vendors such as AWS provide a platform to manage containers such as Amazon ECS for Docker and Amazon EKS for Kubernetes. These cloud services provide a control plane to choose various compute options to select self-managed nodes, managed nodes, or serverless options with AWS Fargate. The following architecture diagram shows running a stateful service on Amazon EKS in your programming languages of choice, such as Java or .NET. Given the architecture, you can manage the session state in a Redis database.

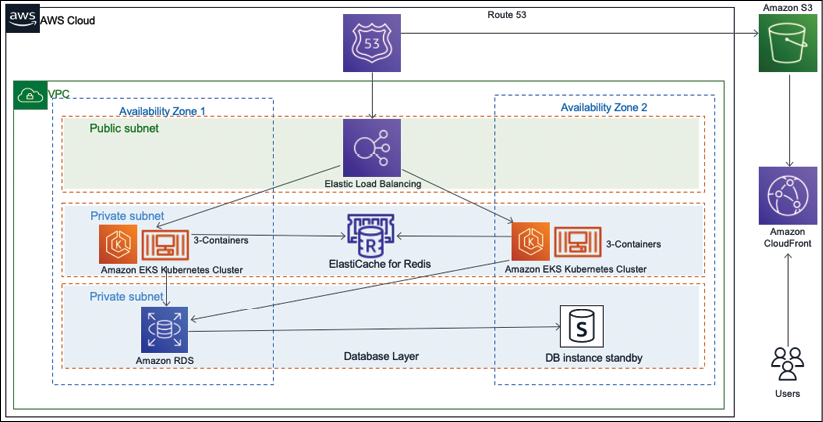


Figure 6.24: Deploying a stateful application on a container

As you can see in the preceding diagram, the container-based architecture includes the following components:

* An Amazon **virtual private cloud** (**VPC**) with one public subnet for the load balancer, and two private subnets for the application and database deployment
* An application load balancer to access the website that is running inside the containers
* An Amazon EKS cluster with a managed node group in Kubernetes. Those nodes run multiple application containers
* An Amazon ElastiCache Redis database to stores the user sessions state

The above architecture helps you to scale the application by saving the user sessions in a Redis database. This solution requires changing the application code, and there are situations where this is not an option.

As of now, you have learned about various architecture patterns focusing on application development. Everyone has to agree that data is an integral part of any architecture design, and most of the architecture revolves around collecting, storing, and processing the visualization of data. Let's learn more about handling data in application architecture in the next section.

**Database handling in application architecture**

Data is always at the center of any application development, and scaling data has always been challenging. Handling data efficiently improves application latency and performance. In the previous section, *Building cache-based architecture*, you learned how to handle frequently queried data by putting a cache in front of your database under the app caching pattern. You can put either a Memcached or Redis cache in front of your database, which reduces the many hits on the database and results in improving database latency.

In application deployment, as the user base of your application grows, you need to handle more data with your relational database. You need to add more storage or vertically scale the database server by adding more memory and CPU power. Often, horizontal scaling is not very straightforward when it comes to scaling relational databases. If your application is read-heavy, you can achieve horizontal scaling by creating a read replica. Route all read requests to database read replicas, while keeping the master database node to serve write and update requests. As a read replica has asynchronous replication, it can add some lag time. You should choose the read replica option if your application can tolerate some milliseconds of latency. You can use read replicas to offload reporting.

You can use database sharding to create a multi-master for your relational database and inject the concept of horizontal scaling. The sharding technique is used to improve writing performance with multiple database servers. Essentially, databases are prepared and divided with identical structures using appropriate table columns as *keys* to distribute the writing processes. As demonstrated in the following architecture diagram, the customer database can be divided into multiple shards:

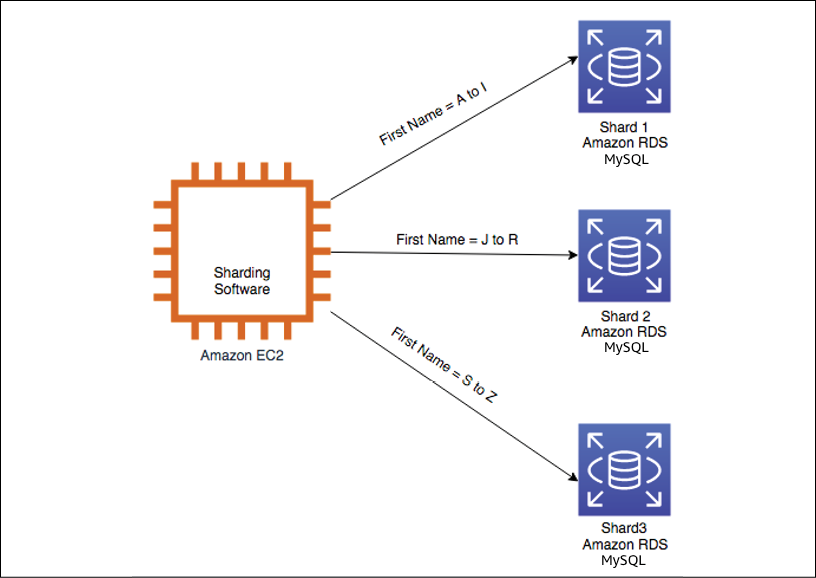


Figure 6.25: Relational database sharding

As shown in the preceding diagram, without *shards*, all data resides in one partition; for example, the user's first name starts with A to Z in one database. With sharding, data is split into large chunks called shards. For example, the users' first names beginning with A to I are in one database, J to R in another database, and S to Z in a third database. In many circumstances, sharding gives you higher performance and better operating efficiency.

You can use Amazon RDS in sharding backend databases. Install sharding software such as MySQL combined with a Spider storage engine on an Amazon EC2 instance. Then, first, prepare multiple RDS databases and use them as the sharding backend databases.

However, what if your master database instance goes down? In that case, you need to maintain high availability for your database. Let's take a closer look at database failover.

**High-availability database pattern**

For the high availability of your application, it is critical to keep your database up and running all of the time. As horizontal scaling is not a straightforward option in the relational database, it creates additional challenges. To achieve high database availability, you can have a standby replica of the master database instance, as shown in the following diagram:

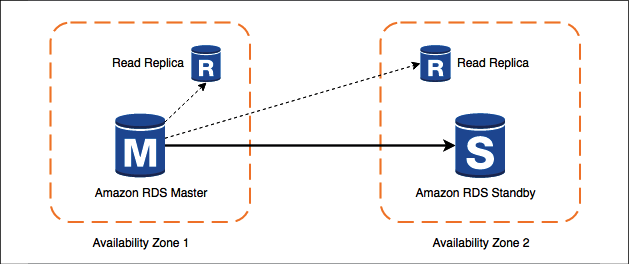


Figure 6.26: High-availability database pattern

As shown in the preceding diagram, if the master instance goes down, your application server switches over to the standby instance. A read replica takes the load off of the master instance to handle latency. Master and standby are located in different **availability zones**, so your application will still be up even when an entire availability zone is down. This architecture also helps to achieve zero downtime, which may be caused during the database maintenance window. When a master instance is down for maintenance, the application can failover to a secondary standby instance and continue serving user requests.

For the purpose of disaster recovery, you will want to define the database backup and archival strategy, depending on your application's **recovery point objective** (**RPO**) of how frequently you want to take backups. If your RPO is 30 minutes, it means your organization can only tolerate 30 minutes worth of data loss. In that case, you should take a backup every half an hour. While storing the backup, you need to determine how long the data can be stored for customer query purposes. You may want to store data for six months as an active backup and then in an archival store as per the compliance requirement.

Consider how quickly you might need to access your backup and determine the type of network connection needed to meet your backup and recovery requirements as per the company **recovery time objective** (**RTO**).

For example, if your company's RTO is 60 minutes, it means you should have enough network bandwidth to retrieve and restore your backup within an hour. Also, define whether you are backing up snapshots of complete systems or volumes attached to systems.

You may also need to classify your data, for example, if it has customer-sensitive information such as email, addresses, personally identifiable information, and more. You need to define the data encryption strategy accordingly. You will learn more about data security in *Chapter 8*, *Security Considerations*.

You can also consider migrating from an RDBMS to a NoSQL database, depending upon your application's growth and complexity. NoSQL can provide you with greater scalability, management, performance, and reliability than most relational databases. However, the process of migrating to NoSQL from an RDBMS can be time-consuming and labor-intensive.

There is lots of data to process in any application, for example, clickstream data, application log data, rating and review data, social media data, and more. Analyzing these datasets and getting insight can help you to grow your organization exponentially. You will learn more about these use cases and patterns in *Chapter 13*, *Data Engineering for Solution Architecture*. As of now, you have learned about the best practices to design a solution architecture. Let's learn about some anti-patterns, which should be avoided, in the next section.

**Avoiding anti-patterns in solution architecture**

In this chapter, you have learned about a different way of designing solution architecture with various design patterns. Often, teams can drift away from best practices due to timeline pressure or the unavailability of resources. You always need to give special attention to the following architecture design anti-patterns:

* In an anti-pattern (an example of a poorly designed system), scaling is done reactively and manually. When application servers reach their full capacity with no more room, users are prevented from accessing the application. On user complaints, the admin finds out that the servers are at their full capacity and starts launching a new instance to take some of the load off. Unfortunately, there is always a few minutes' lag between the instance launch and its availability. During this period, users are not able to access the application. You should take a proactive approach and use auto scaling to add additional processing power when servers reach a certain threshold like 60% CPU utilization or 60% of memory utilization.
* With anti-patterns, automation is missing. When application servers crash, the admin manually launches and configures the new server and notifies the users manually. Detecting unhealthy resources and launching replacement resources can be automated, and you can even notify when resources are changed.
* With anti-patterns, the server is kept for a long time with hardcoded IP addresses, which prevents flexibility. Over time, different servers end up in different configurations and resources are running when they are not needed. You should keep all of the servers identical and should have the ability to switch to a new IP address. You should automatically terminate any unused resources.
* With anti-patterns, an application is built in a monolithic way, where all layers of the architecture including web, application, and data layers are tightly coupled and server dependent. If one server crashes, it brings down the entire application. You should keep the application and web layers independent by adding a load balancer in between. If one of the app servers goes down, the load balancer automatically starts directing all of the traffic to the other healthy servers.
* With anti-patterns, the application is server bound, and the servers communicate directly with each other. User authentication and sessions are stored in the server locally and all static files are served from the local server. You should choose to create a service-oriented RESTful architecture, where the services talk to each other using a standard protocol such as HTTP. User authentication and sessions should be stored in low-latency distributed storage so that the application can be scaled horizontally. The static asset should be stored in centralized object storage that is decoupled from the server.
* With anti-patterns, a single type of database is used for all kinds of needs. You are using a relational database for all needs, which introduces performance and latency issues. You should use the right storage for the right need, such as the following:
  + NoSQL to store the user session
  + Cache data store for low-latency data availability
  + Data warehouse for reporting needs
  + Relational database for transactional data
* With anti-patterns, you will find a single point of failure by having a single database instance to serve the application. Wherever possible, eliminate single points of failure from your architectures. Create a secondary server (standby) and replicate the data. If the primary database server goes offline, the secondary server can pick up the load.
* With anti-patterns, static content such as high-resolution images and videos are served directly from the server without any caching. You should consider using a CDN to cache heavy content near the user location, which helps to improve page latency and reduce page load time.
* With anti-patterns, you can find security loopholes that open server access without a fine-grained security policy. You should always apply the principle of least privilege, which means starting with no access and only giving access to the required user group.

The preceding points provide some of the most common anti-patterns. Throughout this book, you will learn the best practices of how to adopt them in solution design.

**Summary**

In this chapter, you learned about various design patterns by applying the techniques from *Chapter 3*, *Attributes of the Solution Architecture*, and *Chapter 4*, *Principles of Solution Architecture Design*. First, you built the architecture design foundation from a multilayer architecture with a reference architecture from three-tier web application architecture. You learned how to design a multi-tenant architecture on top of a three-tier architecture, which can provide a SaaS kind of offering. You learned how to isolate multi-tenant architecture at the database label, schema level, and table level as per customer and organization needs.

User state management is very critical for complex applications such as finance, e-commerce, travel booking, and more. You learned about the difference between stateful and stateless applications and their benefits. You also learned how to create a stateless application with a persistent layer of the database for session management. You learned about the two most popular SOA patterns, SOAP-based and RESTful-based patterns, along with their benefits. You looked at a reference architecture of an e-commerce website based on SOA and learned how to apply the principles of loose coupling and scaling.

You learned about serverless architecture and how to design a secure survey delivery architecture that is entirely serverless. You also learned about microservice architecture using the example of a serverless real-time voting application, which builds on the microservice pattern. For more loose coupling designs, you learned about the queuing chain and job observer patterns, which provide loosely coupled pipelines to process messages in parallel. You learned about the pub/sub and event stream models to design event-driven architecture.

It's not possible to achieve your desired performance without applying caching. You learned about various cache patterns to apply to caches at the client side, content distribution, web layer, application layer, and database layer.

You learned about architecture patterns to handle failure such as a circuit breaker to handle the downstream service failure scenario and the bulkhead pattern to handle complete service failure. You learned about floating IP patterns to change servers without changing their address in failure situations to minimize downtime.

You learned about the various techniques of handling data in an application and how to make sure your database is highly available to serve your application. Finally, you learned about various architecture anti-patterns and how to replace them using best practices.

While in this chapter you learned about various architecture patterns, in the next chapter, you will learn about architecture design principles for performance optimization. Additionally, you will deep dive into technology selection in the areas of computing, storage, databases, and networking, which can help to improve your application's performance.