**3**

**Attributes of the Solution Architecture**

The solution architecture needs to consider multiple attributes and design applications. Solution design may have a broad impact across numerous projects in an organization, which demands a careful evaluation of the various properties of the architecture while also striking a balance between them. This chapter will provide a holistic understanding of each attribute, and how they are interrelated and coexist in solution design.

There may be more attributes than covered here, depending on the solution's complexity, but in this chapter, you will learn about the common characteristics that can be applied to most aspects of solution design. You can also view them as **non-functional requirements** (**NFRs**), which fulfill an essential aspect of design. It is the responsibility of a solution architect to look at all the attributes and make sure that they satisfy the desired requirements and fulfill customer expectations.

In this chapter, we will cover the following topics:

* Scalability and elasticity
* High availability and resiliency
* Fault tolerance and redundancy
* Disaster recovery and business continuity
* Extensibility and reusability
* Usability and accessibility
* Portability and interoperability
* Operational excellence and maintainability
* Security and compliance
* Cost optimization and budgets

**Scalability and elasticity**

Scalability has always been a primary factor while designing a solution. If you ask any enterprise about their existing and new solutions, most of the time they like to plan ahead for scalability. **Scalability** means giving your system the ability to handle growing workloads, and it can apply to multiple layers, such as the application server, web app, and database.

As most applications nowadays are web-based, let's also talk about elasticity. This is not only about growing out your system by adding more capabilities, but also shrinking it to save on unnecessary costs. Especially with the adoption of the public cloud, it has become easy to grow and shrink your workload quickly, with elasticity now replacing scalability. Traditionally, there are two modes of scaling:

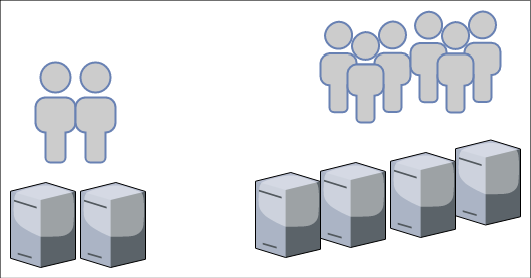
* **Horizontal scaling**: Horizontal scaling is becoming increasingly popular as computing power has become an exponentially cheaper commodity in the last decade. In horizontal scaling, the team adds more servers to handle increasing workloads:

Figure 3.1: Horizontal scaling

As an example, take the diagram shown in *Figure 3.1*; let's say your application is capable of handling *1,000 requests per second* with *two server instances*. As your user base grows, the application starts receiving *2,000 requests per second*, which means you may want to double your application instances to four to handle the increased load.

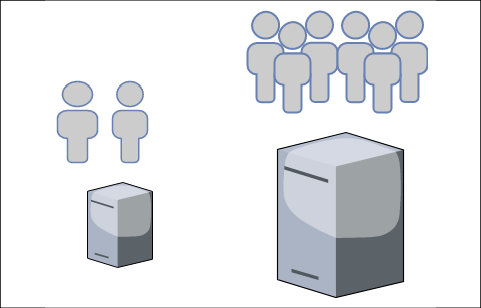
* **Vertical scaling**: This has been around for a long time. It is a practice in which the team adds additional computer storage capacity and memory power to the same instance in order to handle increasing workloads. As shown in *Figure 3.2*, during vertical scaling, you will get a larger instance—rather than adding more new instances—to handle the increased workload:

Figure 3.2: Vertical scaling

The vertical scaling model may not be as cost-effective, however; when you purchase hardware with more computing power and memory capacity, the cost increases exponentially. You want to avoid vertical scaling after a certain threshold unless it is absolutely required to handle increasing workload. Vertical scaling is most commonly used to scale relational database servers. However, you need to think about database sharding here. If your server hits the limits of vertical scaling, a single server cannot grow beyond a certain memory and computing capacity.

**The capacity dilemma in scaling**

Most businesses have a peak season when users are most active and the application has to handle additional load to meet demands. Take the classic example of an e-commerce website selling a variety of products, such as clothes, groceries, electronic items, and merchandise. Such sites have regular traffic throughout the year, but get 10 to 20 times more traffic in the shopping season; for example, Black Friday and Cyber Monday in the US, or Boxing Day in the UK, will see such spikes. This pattern creates an interesting problem for capacity planning, where your workload is going to increase drastically for a couple of months in the year.

In the traditional on-premises data center, additional hardware can take between four and six months before it becomes application-ready, which means a solution architect has to plan for capacity. Excess capacity planning means your IT infrastructure resources will be sitting idle for most of the year, and less capacity means you are going to compromise user experience during significant sales events, thus impacting the overall business significantly. This means a solution architect needs to plan elastic workloads, which can grow and shrink on demand. The public cloud makes capacity planning very easy, where you can get more resources—such as computer storage capacity—instantly, for a limited time period, as per an organization's needs.

**Scaling your architecture**

Let's continue with the e-commerce website example by considering a modern three-tier architecture, and see how we can achieve elasticity at a different layer of the application. Here, we are only targeting the elasticity and scalability aspects of architecture design. You will learn more about this in *Chapter 6*, *Solution Architecture Design Patterns*. *Figure 3.3* shows a three-tier architecture diagram of the AWS cloud tech stack.

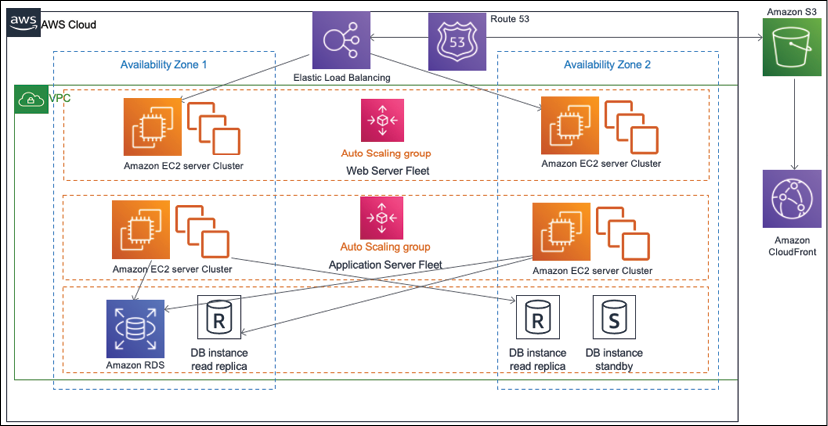


Figure 3.3: Scaling three-tier architecture

You can see a lot of components in this figure, including the following:

* Virtual server (Amazon Elastic Cloud Compute)
* Database (Amazon RDS)
* Load balancer (Amazon Elastic Load Balancer)
* DNS server (Amazon Route53)
* CDN service (Amazon CloudFront)
* Network boundary (VPC) and object store (Amazon S3)

As can be seen in *Figure 3.3*, there is a fleet of web and application servers behind the load balancer. In this architecture, the user sends an application request to the load balancer, which routes traffic to the web server. As user traffic increases, auto-scaling adds more servers in the web and application fleet. When there is low demand, it removes additional servers. Here, auto-scaling can add or remove servers based on the chosen matrix-like CPU utilization and memory utilization; for example, you can configure it such that when CPU utilization goes beyond 60%, add three new servers; if it goes below 30%, you can then remove two existing servers.

In addition to servers, scaling storage is another important aspect due to the growing size of data flow. This is especially the case for static content, such as images and videos, growing rapidly in size; this warrants more focus on storage scaling than has ever been done before. In the next section, you will learn about static content scaling.

**Static content scaling**

The web layer of the architecture is mostly concerned with displaying and collecting data and passing it to the application layer for further processing. In the case of an e-commerce website, each product will have multiple images—and perhaps even videos—to show a product's texture and demos, which means the website will have a great amount of static content with a read-heavy workload since, most of the time, users will be browsing products. In addition to that, users may upload multiple images and videos for product review.

Storing static content in a web server means consuming lots of storage space, and as product listings grow you have to worry about storage scalability. The other problem is that static content (such as high-resolution images and videos) requires large file sizes, which may cause significant load latency on the user's end. The web tier needs to utilize the **Content Distribution Network** (**CDN**) to solve this issue by applying content caching at edge locations.

CDN providers (such as Akamai, Amazon CloudFront, Microsoft Azure CDN, and Google CDN) provide edge locations across the globe where static content can be cached from the web server to available videos and images near the user's location, reducing latency. You will learn more about caching in *Chapter 6*, *Solution Architecture Design Patterns*.

To scale the static content storage, it is recommended to use object storage, such as Amazon S3, or an on-premise custom origin, which can grow independently of memory and computer capabilities. Additionally, scaling storage independently with popular object storage services, such as Amazon S3, saves on cost. These storage solutions can hold static HTML pages to reduce the load of web servers and enhance user experience by reducing latency through the CDN.

**Server fleet elasticity**

The application tier collects user requests from the web tier and performs the heavy lifting of calculating business logic and talking to the database. When user requests increase, the application tier needs to scale to handle them, and then shrink back as demands decrease. In such scenarios, users are tied to the session, where they may be browsing from their mobile and purchasing from their desktop.

Performing horizontal scaling without handling user sessions may cause a bad user experience, as it will reset their shopping progress.

Here, the first step is to take care of user sessions by decoupling them from the application server instance, which means you should consider maintaining the user session in an independent layer, such as a NoSQL database; these databases are *key-value pair stores*, where you can store semi-structured data. NoSQL databases are best suited for semi-structured data where data entries vary in their schema. For example, one user can enter their name and address while setting up a user profile. In contrast, another user can enter more attributes, such as phone number, gender, marital status in addition to name and address. As both users have different sets of attributes, NoSQL data can accommodate them and provide faster search. Key-value databases such as Amazon DynamoDB are highly partitionable and allow horizontal scaling at scales that other types of databases cannot achieve.

Once you start storing your user session in NoSQL databases such as Amazon DynamoDB or MongoDB, your instance can scale horizontally without impacting the user experience. You can add a load balancer in front of a fleet of application servers, which can distribute the load among instances; with the help of auto-scaling, you can automate the addition or removal of instances on demand.

**Database scaling**

Most applications use relational databases to store their transactional data. The main problem with relational databases is that they cannot scale horizontally until you plan for other techniques—such as sharding—and modify your application accordingly. This will be a lot of work.

When it comes to databases, it is better to take preventive care and reduce their load. Using a mix of storage methods, such as storing user sessions in separate NoSQL databases, storing static content in an object store, and applying an external cache, helps to offload the master database. It's better to keep the master database node for writing and updating data and use an additional read replica for all read requests.

The Amazon RDS engine provides up to six read replicas for relational databases, and Oracle plugins can live-sync data between two nodes. Read replicas may have milliseconds of delay while syncing with the master node, and you need to plan for that while designing your application. It is recommended to use a caching engine such as Memcached or Redis to cache frequent queries and thus reduce the load on the master node.

If your database starts growing beyond its current capacity, then you need to redesign and divide the database in to shards by applying partitions.

Here, each shard can grow independently, and the application needs to determine a partition key to store user data in a respective shard. For example, if the partition key is user\_name, then usernames starting from A to E can be stored in one shard, names starting from F to I can be stored in the 2nd partition, and so on. The application needs to direct user records to the correct partition as per the first letter of their name.

So, as you can see, scalability is a significant factor while designing a solution architecture, and it can impact the overall project budget and user experience significantly if it's not planned properly. A solution architect always needs to think in terms of elasticity while designing applications and optimizing workloads for the best performance and least cost.

A solution architect needs to evaluate different options such as CDNs for static content scaling and load balancing, autoscaling options for server scaling, and various data storage options for caching, object stores, NoSQL stores, read replicas, and sharding.

In this section, you have seen discovered the various methods of scaling and how to inject elasticity into the different layers of your architecture. Scalability is an essential factor to ensure that there is high application availability to make your application resilient. We will learn more about high availability and resiliency in the next section.

**High availability and resiliency**

The one thing an organization doesn't want to see is *downtime*. Application downtime can cause a loss of business and user trust, which makes high availability one of the primary factors while designing the solution architecture. The requirement of application uptime varies from application to application.

If you have an external-facing application with a large user base, such as an e-commerce website or social media platform, then 100% uptime becomes critical. In the case of an internal application (accessed by an employee, such as an HR system or an internal company), a blog can tolerate some downtime. Achieving high availability is directly associated with cost, so a solution architect must always plan for high availability, as per the application requirements, to avoid over-architecting.

To achieve a **high availability** architecture, it is better to plan workloads in the isolated physical location of the data center so that, should an outage occur in one place, then your application replica can operate from another location.

As shown in the architecture diagram in *Figure 3.4*, you have a web and application server fleet available in two separate availability zones (which represent the different physical locations of the data centers).

The load balancer helps distribute the workload between two availability zones in case **Availability Zone 1** goes down due to a power or network outage. **Availability Zone 2** can handle user traffic, and your application will be up and running.

In the case of the database, you have a standby instance in **Availability Zone 2**, which will failover and become the primary instance in the event of an issue in **Availability Zone 1**. Both the master and standby instances continuously synchronize data.

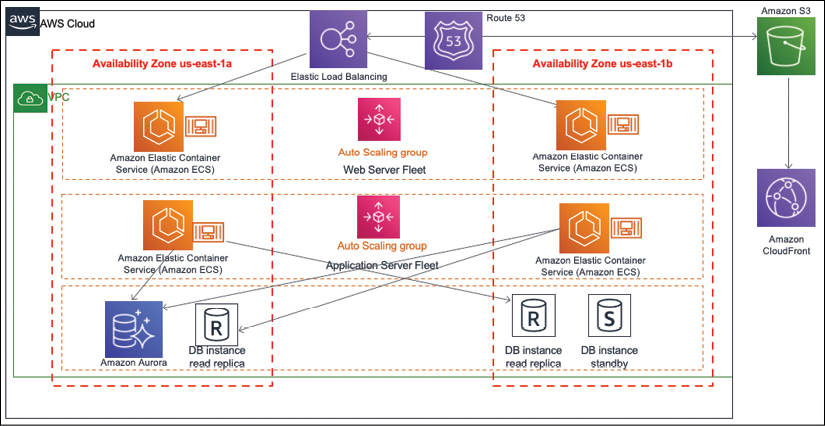


Figure 3.4: High availability and resilience architecture

The other important factor is the architecture's resiliency. When your application is in trouble and you are facing an intermittent issue, then apply the principle of self-healing, which means your application should be able to recover itself without human intervention.

For your architecture, resiliency can be achieved by monitoring the workload and taking proactive action. As shown in *Figure 3.4*, the load balancer will be monitoring the health of instances. If any instance stops receiving the request, the load balancer can take out the bad instances from the server fleet and tell autoscaling to spin up a new server as a replacement. The other proactive approach is to monitor the health of all instances (such as CPU and memory utilization and spinning up new instances as soon as a working instance starts to reach a threshold limit), such as by ensuring CPU utilization is higher than 70%, or that memory utilization is more than 80%.

The attributes of high availability and resiliency can help in terms of cost by achieving elasticity. For example, if server utilization is low, you can take out some servers and save on cost of having this excess capacity.

The high availability architecture goes hand-in-hand with self-healing, where you can make sure that your application is up and running, but you also need to have quick recovery to maintain the desired user experience.

While high availability ensures your system is up and available for users, it is also essential to maintain performance where fault tolerance comes into play. Let us now turn to the subjects of fault tolerance and redundancy.

**Fault tolerance and redundancy**

In the previous section, you learned that fault tolerance and high availability have a close relationship to each other. High availability means your application is available to the user, but perhaps with degraded performance. Suppose you need four servers to handle users' traffic. For this, you put two servers in two different physically isolated data centers. If there is an outage in one data center, then user traffic can be served from another data center. But now you have only two servers, which means only 50% of the original capacity is available, and users may experience performance issues. In this scenario, your application has 100% high availability, but is only 50% fault tolerant.

Fault tolerance is about handling workload capacity if an outage occurs, without compromising system performance. A full fault-tolerant architecture involves high costs due to increased redundancy. Whether your user base can live with degraded performance for the period of application recovery depends on your application's criticality.

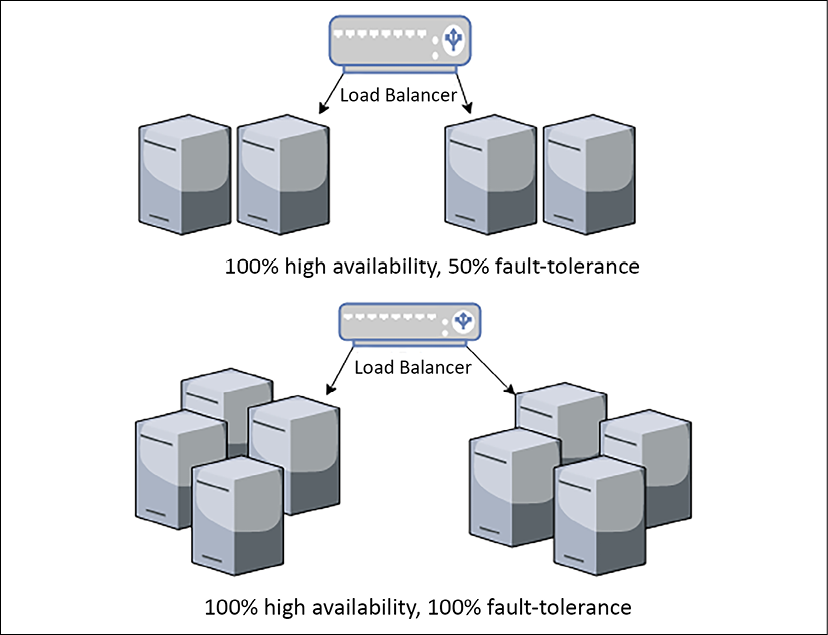


Figure 3.5: Fault-tolerance architecture

As shown in *Figure 3.5*, your application needs four servers to handle the full workload by distributing them into two different zones. In both scenarios, you are maintaining 100% high availability. To achieve 100% fault tolerance, you need full redundancy and have to maintain the double count of the servers so that the user doesn't encounter any performance issues during the outage of one zone. By keeping the same number of servers, a fault tolerance of only 50% is achieved.

While designing the application architecture, a solution architect needs to determine the nature of the application's user and whether a fault tolerance of 100% is required, which will inevitably come with a cost implication. For example, an e-commerce website may need 100% fault tolerance, as degraded performance directly impacts business revenue. At the same time, the internal payroll system, which employees use at the end of the month to check their salary slips, can tolerate reducing performance for a short period.

For business continuity, it is required to plan for uncertainty, which can cause system downtime and hamper overall availability. Disaster recovery helps to mitigate this risk by ensuring the system is available in unforeseen events. Let's learn more about disaster recovery planning in the next section.

**Disaster recovery and business continuity**

In the previous section, you learned about using high availability and fault tolerance to handle application uptime. There may be a situation when the entire region where your data center is located goes down due to massive power grid outages, earthquakes, or floods, but your global business should continue running. In such situations, you must have a disaster recovery plan where you will plan your business continuity by preparing sufficient IT resources in an entirely different region, perhaps even in different continents or countries.

When planning disaster recovery, a solution architect must understand an organization's **Recovery Time Objective** (**RTO**) and **Recovery Point Objective** (**RPO**). RTO is a measure of how much downtime a business can sustain without any significant impact; RPO indicates how much data loss a business can tolerate. Reducing RTO and RPO means incurring greater cost, so it is essential to understand whether the business is mission-critical and needs minimal RTO and RPO. For example, a stock trading application cannot afford to lose a single data point, or a railway signaling application cannot be down, as human life depends on it.

The following architecture diagram in *Figure 3.6* shows a multi-site disaster recovery architecture where the primary data center location is in Ireland, Europe, and the disaster recovery site is in Virginia, USA, hosted on the AWS public cloud. In this case, a business can continue operating, even if something happens to the entire European region or to the public cloud. The fact that the disaster recovery plan is based on a multi-site model to achieve minimal RTO and RPO means minimal to no outage, and no data loss.

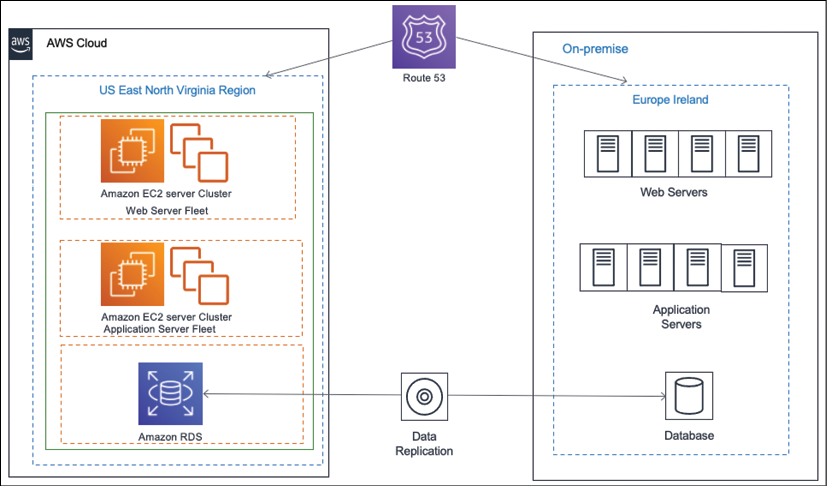


Figure 3.6: Hybrid multi-site disaster recovery architecture

The following are the most common disaster recovery plans, all of which you will learn about in *Chapter 12*, *DevOps and Solution Architecture Framework*:

* **Backup and Store**: This plan is the least costly but has the maximum RTO and RPO. In this plan, all the server's machine images and database snapshots should be stored in the disaster recovery site. In the event of a disaster, the team will try to restore the disaster site from a backup.
* **Pilot Lite**: In this plan, all the server's machine images are stored as a backup, and a small database server is maintained in the disaster recovery site with continual data synchronization from the main site. Other critical services, such as Active Directory, may be running in small instances. In the event of a disaster, the team will try to bring up the server from the machine image and scale up a database. Pilot Lite is a bit more costly but has lower RTO and RPO than Backup and Store.
* **Warm Standby**: In this plan, all the application servers and the database server (running at low capacity) instances in the disaster recovery site and continue to sync up with the leading site. In the event of a disaster, the team will try to scale up all the servers and databases. Warm Standby is costlier than the Pilot Lite option, but has lower RTO and RPO.
* **Multi-Site**: This plan is the most expensive and has a near-zero RTO and RPO. In this plan, a replica of the leading site is maintained in a disaster recovery site with equal capacity and that actively serves user traffic. In the event of a disaster, all traffic will be routed to an alternate location.

Often, organizations choose a less costly option for disaster recovery, but it is essential to perform regular testing to make sure the failover is working. The team should make operational excellence a routine checkpoint to make sure there is business continuity in the event of disaster recovery.

**Extensibility and reusability**

Businesses evolve as they grow, where applications not only scale to handle an increased user base but also keep adding more features to stay ahead and attain a competitive edge. A solution design needs to be extendable and flexible enough to modify an existing feature or add new functionality. To modularize their application, organizations often want to build a platform with a group of features and launch them as separate applications. This is only possible with reusable design.

To achieve solution extensibility, a solution architect needs to use a loosely coupled architecture wherever possible. At a high level, creating a RESTful- or queue-based architecture can help develop loosely coupled communication between different modules or across applications. You will learn more about the other kinds of architecture in *Chapter 6*, *Solution Architecture Design Patterns*. In this section, we will take a simple example to explain the concept of architecture flexibility.

*Figure 3.7* shows an API-based architecture in an e-commerce application. Here, you have independent services, such as product catalog, order, payment, and shipping being utilized by an end user application in a pick-and-choose manner. Mobile and browser applications are used by the customer to place an online order. These applications need a product catalog service to browse the product on the web, an order service to place an order, and a payment service to make a payment.

The product catalog and order service, in turn, communicate with the shipping service to send ordered items to the customer's doorstep. On the other hand, brick-and-mortar stores use Point of Sale systems, where a customer representative scans barcodes, places orders on behalf of the customer, and takes payment. Here, no shipping service is required, as the customer picks up the item in-store.

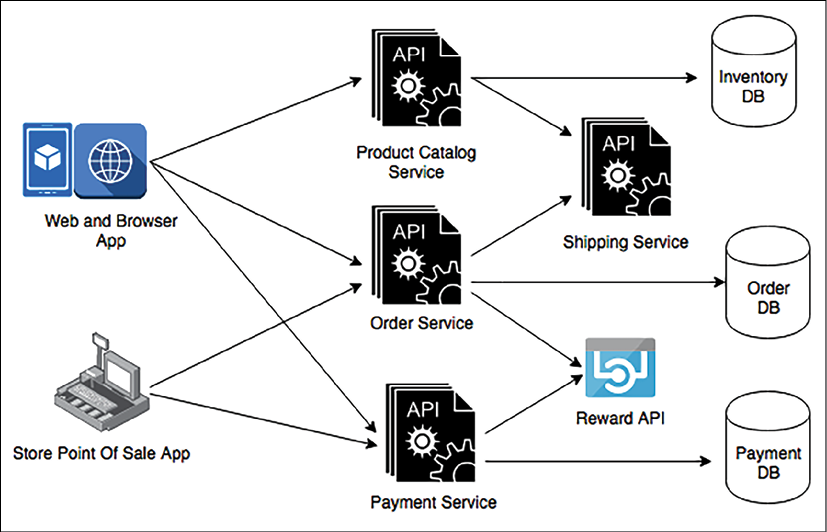


Figure 3.7: Extensible API-based architecture

From *Figure 3.7*, you can see the Reward API, which is used for third-party API integration. This architecture allows you to extend the current design to integrate the Reward API for customer retention, and to attract new customers by providing benefits when they purchase an item. Here, you can see how payment services are reutilized by both online and store ordering. Another service can reuse this if the organization wants to take payments for a gift card service, food services, and so on.

Extensibility and reusability are not limited to the service design level—it goes deep into the actual API framework level, where software architects should use **object-oriented analysis and design** (**OOAD**) concepts, such as inheritance and containership, to create an API framework. This can be extended and reutilized to add more features to the same service.

You may create a very feature-rich product, but it may not find wide appeal with users until they find it easy to navigate and access. Your application's usability and accessibility play a significant role in product success. Let's learn more about this in the next section.

**Usability and accessibility**

You want your users to have a seamless experience when browsing through the application. It should be so smooth that they don't even notice how easily they are able to find things—without any difficulties whatsoever. You can do this by making your application highly usable. User research and testing are essential aspects when it comes to defining usability that can satisfy user experience.

Usability is how quickly the user can learn navigation logic when using your application for the first time. It's about how quickly they can bounce back if they make a mistake and are able to perform the task efficiently. Complex and feature-rich applications have no meaning if they can't be used effectively.

Often, when you are designing your application, you want to target a global audience or significant geographic region. Your user base should be diverse in terms of technical amenities and physical abilities. You want your application to be accessible to everyone, regardless of whether a user has a slow internet connection, uses an old device, or they have physical limitations.

Accessibility is about inclusion, making your application usable by everyone. While designing an application, a solution architect needs to make sure it can be accessed over a slow internet connection and is compatible with a diverse set of devices. Sometimes, they may have to create a different version of the application altogether to achieve that.

Accessibility design should include design components, such as voice recognition and voice-based navigation, screen magnifiers, and an ability to read content aloud. Localization helps the application become available in a language that's specific to a region; for example, Spanish, Mandarin, German, Hindi, or Japanese.

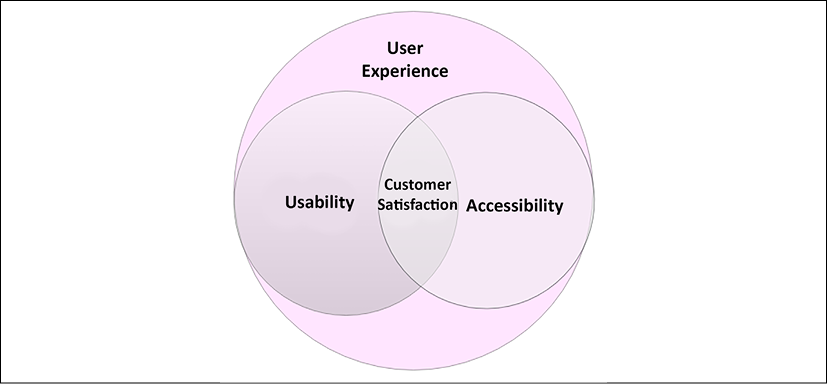


Figure 3.8: Customer satisfaction with usability and accessibility

As shown in *Figure 3.8*, customer satisfaction is a component of both usability and accessibility. You must know your users to achieve usability and accessibility—where accessibility is a component of usability—as they go hand in hand. Before starting the solution design process, a solution architect should work alongside a product owner to research users by conducting interviews, surveys, and gathering feedback on the mock frontend design. You need to understand the users' limitations and empower them with supporting features during application development.

When the product is launched, the team should plan for A/B testing by routing a small portion of user traffic to new features and understanding user reactions. A/B testing is a method of comparing two versions of an application against each other to determine which one performs better. After launch, the application must have a mechanism to collect continuous feedback (by providing a feedback form or by launching customer support) to make the design better.

A system cannot work alone long-term. To make the application feature-rich and simplified for user interactions the solution architect must consider its operability with other applications. Let's look at portability and interoperability in the next section.

**Portability and interoperability**

Interoperability is about the ability of one application to work with others through a standard format or protocol. Often, an application needs to communicate with the various upstream systems to consume data and downstream systems to supply data, so it is essential to establish that communication seamlessly.

For example, an e-commerce application needs to work with other applications in the supply chain management ecosystem. This includes enterprise resource planning applications to keep a record of all transactions, transportation life cycle management, shipping companies, order management, warehouse management, and labor management.

All applications should be able to exchange data seamlessly to achieve an end-to-end feature from customer order to delivery. You will encounter similar use cases everywhere, whether it is a healthcare application, manufacturing application, or telecom application.

A solution architect needs to consider application interoperability during design by identifying and working with various system dependencies. An interoperable application saves a lot in terms of cost, as it depends on systems that can communicate in the same format without any data messaging effort. Each industry has its standard size for data exchange that it needs to be understood and adhered to.

In general, for software design, the architect may choose a popular format, such as JSON or XML for different applications, so that they can communicate with each other. In modern RESTful API design and microservice architecture, both formats are supported out of the box.

System portability allows your application to work across different environments without the need for any changes, or with only minimal changes. Any software application must work across various operating systems and hardware to achieve higher usability. Since technology changes rapidly, you will often see that a new version of a software language, development platform, or operating system is released. Today, mobile applications are an integral part of any system design, and your mobile apps need to be compatible with major mobile operating systems platforms, including iOS, Android, and Windows.

During the design phase, the solution architect needs to choose a technology that can achieve the desired portability of the application. For example, if you are aiming to deploy your application across different operating systems, programming languages such as Java may be a good choice, as it is often supported by all operating systems, and your application will work on a different platform without needing to be ported across. For mobile applications, an architect may choose a JavaScript-based framework such as React Native, which can provide cross-platform mobile app development.

Interoperability enriches system extensibility, and portability increases the usability of an application. Both are critical attributes of architecture design and may add additional exponential costs if they're not addressed during solution design. A solution architect needs to carefully consider both aspects, as per industry requirements and system dependencies.

**Operational excellence and maintainability**

Operational excellence can be a great differentiator for your application by providing an on-par service to customers with minimal outage and high quality. It also helps the support and engineering teams increase productivity by applying proactive operational excellence. Maintainability goes hand-in-hand with operational excellence. Easily maintainable applications help reduce costs, avoid errors, and let you gain a competitive edge.

A solution architect needs to design for operation, which means the design should include how the workload will be deployed, updated, and operated in the long term.

It is essential to plan for logging, monitoring, and alerting to capture all incidents and take quick action for the best user experience. Apply automation wherever possible, whether deploying infrastructures or changing the application code to avoid human error.

Including deployment methods and automation strategy in your design is very important, as this can accelerate the time to market for any new changes without impacting existing operations. Operational excellence planning should consider security and compliance elements, as regulatory requirements may change over time and your application must adhere to them in order to operate.

Maintenance can be proactive or reactive; for example, once a new version of an operating system becomes available, you can modernize your application to switch platforms immediately, or monitor system health and wait until the end of the life of the software before making any changes. In any case, changes should be made in small increments with a rollback strategy. To apply these changes, you can automate the entire process by setting up a **continuous integration** and **continuous deployment** (**CI**/**CD**) pipeline. For the launch, you can plan for A/B or blue-green deployment.

For operational readiness, architecture design should include the appropriate documents and knowledge-sharing mechanisms—for example, creating and maintaining a runbook to document routine activity, and creating a playbook that can guide your system process through issues. This allows you to act quickly in the event of an incident. You should use *root cause analysis* for post-incident reporting to determine why the issue occurred and make sure it doesn't happen again.

Operational excellence and maintenance are an ongoing effort; every operational event and failure is an opportunity to learn and help improve your operation by learning from previous mistakes. You must analyze the operation's activities and failures, do more experimentation, and make improvements. You will learn more about operational excellencein *Chapter 10*, *Operational Excellence Considerations*.

**Security and compliance**

Security is one of the most essential attributes of solution design. Many organizations are compromised by security breaches, which results in a loss of customer trust and damage to your business' reputation. Industry-standard regulations, such as PCI for finance, HIPAA for health care, GDPR for the European Union, and SOC compliance, enforce security safeguards to protect consumer data while providing standard guidance to the organization. Depending on your industry and region, you must comply with local legislation by adhering to compliance needs.

Primarily, application security needs to be applied in the following aspects of solution design:

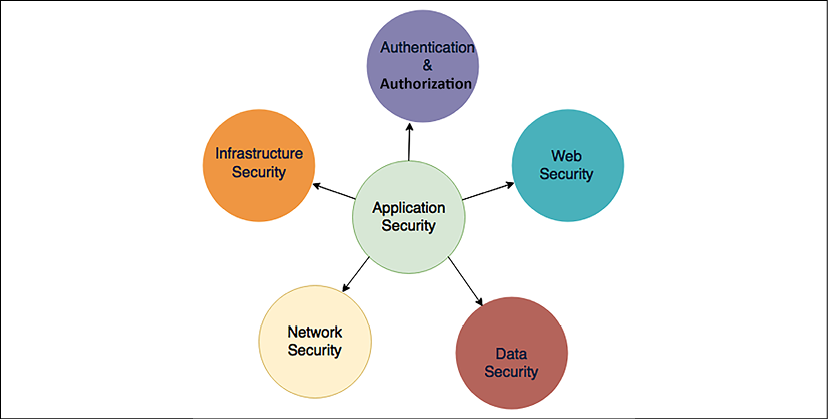


Figure 3.9: Security aspects in solution design

Let's take a look at the different security aspects. You will dive deep into each component in *Chapter 8*, *Security Considerations*.

**Authentication and authorization**

**Authentication** means specifying who can access the system, while **authorization** is applied to activities that a user can perform after getting inside the system or application. Solution architects must consider the appropriate authentication and authorization system while creating a solution design. Always start with the least privileged and provide further access as required by the user role.

If your application is for corporate internal use, you may want to allow access through a federated organizational system, such as Active Directory, SAML 2.0, or LDAP. If your application is targeting mass user bases, such as those that exist on social media websites or gaming apps, then you can allow them to authenticate through OAuth 2.0 and OpenID access, where users can utilize their other IDs, such as Facebook, Google, Amazon, and Twitter.

It is important to identify any unauthorized access and take immediate action to mitigate security threats; this warrants continuously monitoring and auditing the access management system. You will learn about application security in *Chapter 8*, *Security Considerations*.

**Web security**

A web application is often exposed to the internet and is vulnerable to external attacks. Solution design must consider preventing attacks, such as **cross-site scripting** (**XSS**) and **SQL injection**. These days, the **Distributed Denial of Service** (**DDoS**) method of attack is causing trouble for organizations. To prevent this, the appropriate tools are required, and an incident response plan needs to be put in place.

Solution architects should plan to use a **Web Application Firewall** (**WAF**) to block malware and SQL injection attacks. A WAF can be used to prevent traffic from a country where you don't have a user base or to block malicious IP addresses. A WAF, in combination with a CDN, can help to prevent and handle DDoS attacks.

**Network security**

Network security helps prevent overall IT resources inside an organization and application from being open to external users. Solution design must plan to secure the network, which can help prevent unauthorized system access, host vulnerabilities, and port scanning.

Solution architects should plan for minimal system exposure by keeping everything behind a corporate firewall and avoiding internet access wherever possible. For example, the web server shouldn't be exposed to the internet; instead, only the load balancer should be able to talk to the internet. For network security, plan to utilize an **Intrusion Detection System** (**IDS**) and an **Intrusion Prevention System** (**IPS**) and put them in front of network traffic.

**Infrastructure security**

If you are maintaining your own data center, then the physical security of the infrastructure is very important if you wish to block physical access to your server on the part of any unauthorized user. However, if you are leasing a data center or using a private cloud, then this can be handled by a third-party vendor. Logical access to the server must be secured by network security, which is done by configuring the appropriate firewall.

Malicious attacks are common, which is the primary reason for data center security breaches. Security infrastructure becomes very important, managing who can access data, and protect it from any vulnerabilities. From your application hosting data center to the company HR system and global location, you need to ensure every level of IT infrastructure is secure.

**Data security**

Data is one of the most critical components that need to be secured. After all, you are putting layers of security at the access, web, application, and network levels to secure your data. Data can be exchanged between two systems, so it must be *secure in transit*, or it may be *secure at rest* (sitting in a database or some storage unit).

Solution design needs to plan data-in-transit security with **Secure Socket Layer/Transport Layer Security** (**SSL/TLS**) and security certification. Data at rest should be secured using various encryption mechanisms, which may be symmetric or asymmetric. The design should also plan to secure the encryption key with the right key management approach, as per application requirements. Key management can be achieved by using a hardware security module or services provided by cloud vendors. Rules of least privilege using identification and authorization management should be applied to define who can access what data.

While ensuring security, it is essential to have a mechanism to identify any security breach as soon as it occurs to enable a swift response. Adding automation to every layer to monitor, and get an immediate alert for, any violation must be part of the solution design. DevSecOps is becoming a trend in most organizations since it applies best practices to automating security needs and security responses during the software development part of the life cycle. You will learn more about DevSecOps in *Chapter 12*, *DevOps and Solution Architecture Framework*.

To comply with the relevant legislation, solution design needs to include an audit mechanism. For finance, regulatory compliance such as the **Payment Card Industry Data Security Standard** (**PCI DSS**) is strictly required to gain the log trails of every transaction in the system, which means all activity needs to be logged and sent to the auditor when required. Any **Personal Identifiable Information** (**PII**) data, such as customer email IDs, phone numbers, and credit card numbers, needs to be secured by applying encryption and limited access for any application storing PII data.

In on-premise environments, it is the customer organization's responsibility to secure the infrastructure and application, and also to get the appropriate certification for compliance. However, in the public cloud, environments such as AWS ease this burden, as infrastructure security and compliance are taken care of by the cloud vendor. The customer shares responsibility for the security of the application and makes sure it's compliant by completing the required audit.

**Cost optimization and budget**

Every solution is limited by budget, with investors looking for maximal ROI. The solution architect needs to consider cost-saving during architecture design.

Cost should be optimized from pilot creation to solution implementation and launch. Cost optimization is a continuous effort and should be a continuous process. Like any other constraint, cost-saving comes with a trade-off; it should make a point of determining whether other components, such as the speed of delivery and performance, are more critical.

Often, cost increases due to the over-provision of resources and overlooking the cost of procurement. The solution architect needs to plan optimal resources to avoid excessive underutilization. At the organization level, there should be an automated mechanism to detect ghost resources, where team members may create development and test environments that are no longer be in use after the completion of the implementation task. Often, ghost resources go unnoticed and cause costs to overrun. Organizations need to keep a record of their IT inventory by applying automated discovery, which ensures that all IT inventory is tracked and logged into a central database with their current health and operating status.

During technology selection, it is essential to evaluate *build versus source cost*. Sometimes, it's better to use a third-party tool when your organization doesn't have the expertise on hand and the cost of the build will be high, for example, sourcing log analysis and business intelligence tools. Also, we need to determine the ease of learning and the complexity of implementation when selecting a technology for solution implementation. From an IT infrastructure perspective, we need to evaluate capital expenditure versus operation expenditures, as maintaining a data center requires high capital investment upfront to meet unforeseen scaling demands. Since multiple choices are available, solution architects can select options from the following: public, private, and multi-cloud. Alternatively, they can take a hybrid approach.

Like all components, cost needs to be automated, and alerts need to be set up against budget consumption. Cost needs to be planned and divided between the organizational unit and the workload so that responsibilities can be shared with all groups. The team needs to continuously look at cost optimization by optimizing operation support and workload as more historical data is collected. You will learn more about cost optimization in *Chapter 11*, *Cost Consideration*.

**Summary**

In this chapter, you learned about the various solution architecture attributes that need to be considered while creating a solution design. You learned about two modes of scalability, vertical and horizontal, and how to scale various layers of the architecture, including the web layer, application servers, and databases.

You also learned how to apply elasticity to your workload using autoscaling so that it can grow and shrink on demand. This chapter also provided insights into designing a resilient architecture, and the methods used to achieve high availability. Furthermore, this helped you understand fault tolerance and redundancy so that you can make your application performant, as per your user's expectations, and plan for disaster recovery for the continuation of your business in the case of unforeseen events.

You then learned about the importance of making your architecture extendable and accessible and how architecture portability and interoperability help reduce costs and increase the adoption of your application. This chapter ended by explaining methods to apply operational excellence and security, and save on costs at every layer of your architecture, and how those attributes should be considered right from the beginning of the solution design process. You will look at each component in more detail later in this book.

In the next chapter, you will learn about the principle of solution architecture design. We will focus on how to design the solution architecture while bearing in mind various attributes that were explained in this chapter.