**4**

**Principles of Solution Architecture Design**

In the previous chapter, you learned about the attributes of solution architecture. Those attributes are essential properties that a solution architect needs to keep in mind while creating a solution design. In this chapter, you will learn about the principles of solution architecture design, which incorporate various attributes during solution design.

This chapter throws light on the most important and common design principles. However, there could be more design aspects based on product complexity and industry domain. As you move forward on your learning path of becoming a solution architect in this book, these design principles and attributes will be further applied in order to create various design patterns in *Chapter 6*, *Solution Architecture Design Patterns*.

You will learn about the following topics in this chapter:

* Scaling workloads
* Building a resilient architecture
* Design for performance
* Using replaceable resources
* Think loose coupling
* Think service not server
* Using the right storage for the right requirements
* Think data-driven design
* Overcoming architectural constraints
* Adding security everywhere
* Applying automation everywhere

In this chapter, you will not only learn about designing scalable, resilient, and performant architecture, but you will also learn how to protect your application by applying security, overcoming architectural constraints, and applying changes along with the test and automation approach. These principles will help you to apply thinking in the right way by using a data-driven approach.

**Scaling workloads**

In the *Scalability and elasticity* section of *Chapter 3*, *Attributes of the Solution Architecture*, you learned about different modes of scaling and how to scale static content, a server fleet, and a database at a high level. Now, let's look at various types of scaling that can be used to handle workload spikes.

Scaling could be predictive if you are aware of your workload, which is often the case; or it could be reactive if you get a sudden spike or if you have never handled that kind of load before.

For example, the following **Auto Scaling group** has a maximum of six instances and a minimum size of three instances. During regular user traffic, three servers will be up and running to handle the workload, but to handle a traffic spike, the number of servers can reach six. Your server fleet will increase based on the scaling policies you define to adjust the number of instances. For example, you can choose to add one server when CPU utilization goes beyond 60% in the existing servers' fleet, but doesn't spin up more than six servers.

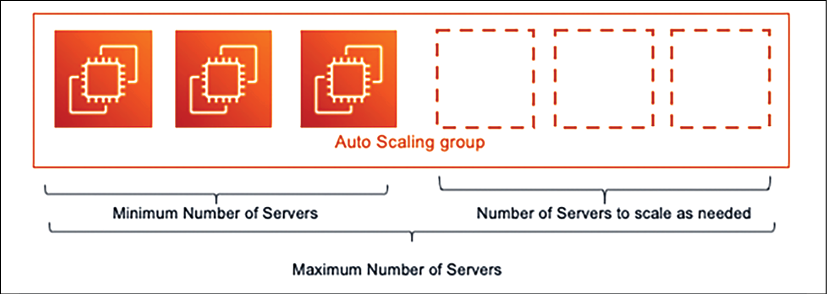


Figure 4.1: Server Auto Scaling

Regardless of scaling being reactive or predictive, you need to monitor the application and collect the data in order to plan your scaling needs. Let's dive deep into these patterns.

**Predictive scaling**

**Predictive scaling** is the best-case approach that any organization wants to take. Often, you can collect historical data of application workloads. For example, an e-commerce website such as Amazon may have a known traffic spike pattern, and you need predictive scaling to avoid any latency issues. Traffic patterns may include the following:

* Weekends have three times more traffic than a weekday
* Daytime has five times more traffic than at night
* Shopping seasons, such as Thanksgiving or Boxing Day, have 20 times more traffic than regular days
* Overall, the holiday season in November and December has 8 to 10 times more traffic than during other months

You may have collected the previous data based on monitoring tools that are in place to intercept the user's traffic, and based on this, you can predict the scaling requirements. Scaling may include planning to add more servers when workload increases, or to add additional caching. The above example of an e-commerce workload tends toward higher complexity and provides lots of data points to help us to understand overall design issues. For such complex workloads, predictive scaling becomes more relevant.

Predictive auto-scaling is a variation on scaling that is becoming very popular, where historical data and trends can be fed to prediction algorithms, and you can predict in advance how much workload is expected at a given time. Using this expected data, you can set up the configuration to scale your application.

To better understand predictive auto-scaling, look at the following metrics dashboard from the AWS predictive auto-scaling feature.

This graph has captured historical CPU utilization data of the server, and based on that, has provided the forecasted CPU utilization:

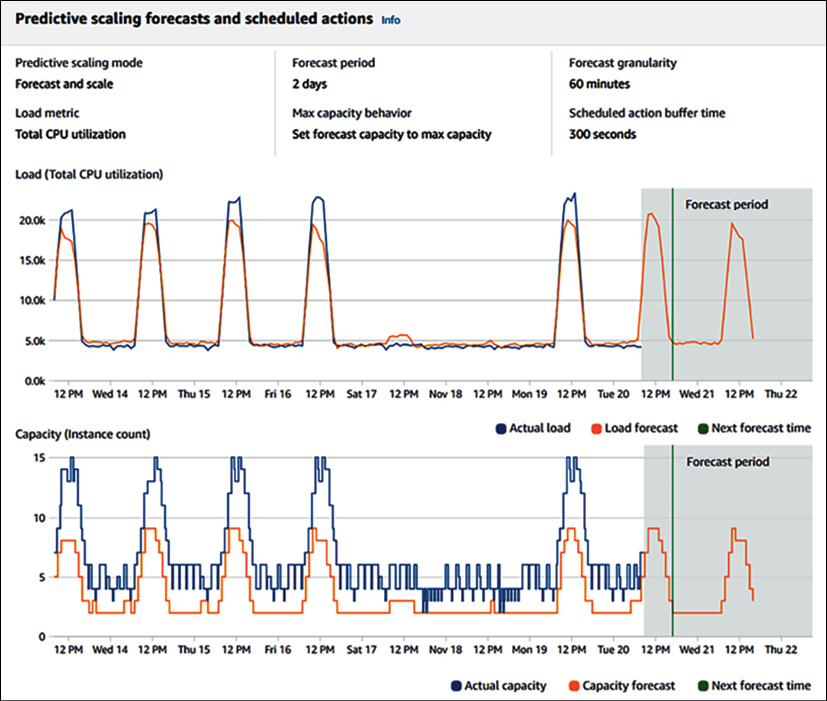


Figure 4.2: Predictive scaling forecast

In the following screenshot, an algorithm is suggesting how much minimum capacity you should plan in order to handle the traffic, based on the forecast:

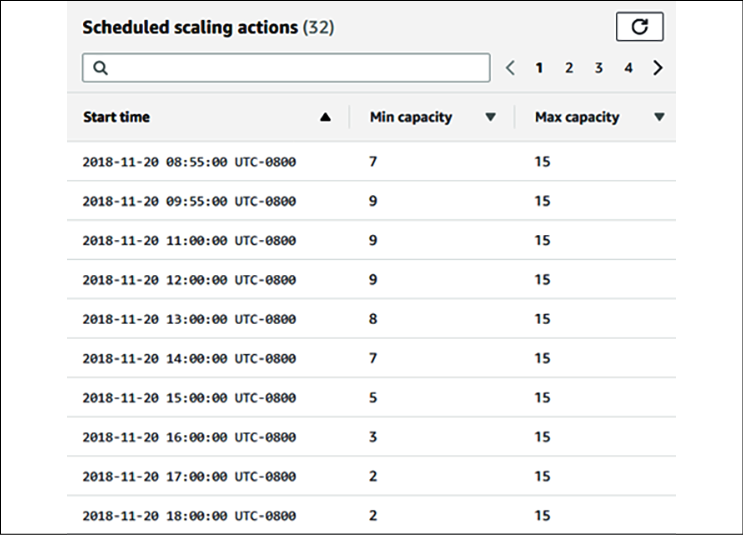


Figure 4.3: Predictive scaling capacity plan

You can see that there is a variation in the minimum capacity at different times of the day. Predictive scaling helps you to best optimize your workload based on predictions, while predictive auto-scaling helps to reduce latency and avoid an outage, as adding new resources may take some time. If there is a delay in adding additional resources to handle website traffic spikes, it may cause a request flood and false high traffic, as users tend to send repeated requests when they encounter slowness or outages.

In this section, you learned about predictive auto-scaling, but sometimes, due to a sudden spike in the workload, you require reactive scaling. We will learn about this in the next section.

**Reactive scaling**

With the use of a machine learning algorithm, predictive scaling is becoming more accurate, but sometimes you may have to deal with sudden traffic spikes, and will therefore depend upon reactive scaling. This unexpected traffic that may arrive could be even 10 times the volume of regular traffic; this usually happens due to a sudden demand or, for example, a first attempt to run sales events, where we're not sure about the level of incoming traffic.

Let's take an example where you are launching a flash deal on your e-commerce website. You will have a large amount of traffic on your home page, and from there, the user will go to the flash deal product-specific page. Some users may want to buy the product; therefore, they will go to the add to cart page.

In this scenario, each page will have a different traffic pattern, and you will need to understand your existing architecture and traffic patterns, along with an estimate of the desired traffic. You also need to understand the navigation path of the website. For example, the user has to log in to buy a product, which can lead to more traffic on the login page.

To plan for the scaling of your server resources for traffic handling, you need to determine the following patterns:

* Determine web pages, which are read-only and can be cached.
* Which user queries need to read just that data, rather than write or update anything in the database?
* Does a user query frequently, requesting the same or repeated data, such as their own user profile?

Once you understand these patterns, you can plan to offload your architecture in order to handle excessive traffic. To offload your web-layer traffic, you can move static content, such as images and videos, to content distribution networks from your web server. You will learn more about the *Cache Distribution* pattern in *Chapter 6*, *Solution Architecture Design Patterns*.

At the server fleet level, you need to use a load balancer to distribute traffic, and you need to use auto-scaling to increase or shrink several servers in order to apply horizontal scaling.

To reduce the database load, use the right database for the right requirements—a NoSQL database for storing user sessions and reviewing comments, a relational database for the transaction, and the application of caching to store frequent queries.

In this section, you learned about the scaling patterns and methods that are used to handle the scaling needs of your application in the form of *predictive scaling* and *reactive scaling*. In *Chapter 6*, *Solution Architecture Design Patterns*, you will learn about the details of the different types of design patterns, and how to apply them in order to be able to scale your architecture.

**Building a resilient architecture**

Design for failure and nothing will fail. Having a resilient architecture means that your application should be available for customers while also recovering from failure. Making your architecture resilient includes applying best practices to recover your application from increased loads due to more user requests, malicious attacks, and architectural component failure. Resiliency needs to be used in all architectural layers, including infrastructure, application, database, security, and networking. A resilient architecture should recover within the desired amount of time.

To make your architecture resilient, you need to define the time of recovery and consider the following points:

* Identify and implement redundant architectural components wherever required.
* Identify and implement backup and disaster recovery plans within a defined **Recovery Time Objective** (**RTO**) and **Recovery Point Objective** (**RPO**)
* Understand when to fix versus when to replace architectural components. For example, fixing a server issue might take longer than replacing it with the same machine image.

Security is one of the most important aspects of application resiliency. From the security perspective, the **Distributed Denial of Service** (**DDoS**) attack has the potential to impact the availability of services and applications. The DDoS attack usually puts fake traffic in your server and makes it busy, meaning that legitimate users are unable to access your application. This can happen at the network layer or the application layer. You will learn more about DDoS attacks and mitigation in *Chapter 8*, *Security Considerations*.

It's essential to take a proactive approach to prevent DDoS attacks. The first rule is to keep as much of the application workload as possible in the private network and not expose your application endpoints to the internet wherever possible.

To take early action, it is essential to know your regular traffic and have a mechanism in place to determine substantial suspicious traffic at the application and network packet levels.

Exposing your application through the **content distribution network** (**CDN**) will provide the inbuilt capability, and adding the **Web Application Firewall** (**WAF**) rule can help to prevent unwanted traffic. During a DDoS attack, Scaling should be your last resort, but be ready with an auto-scaling mechanism to enable you to scale your server in the case of such an event.

To achieve resiliency at the application level, the first thing that comes to mind is redundancy, which leads to making your application highly available by spreading the workload across geographic locations. To achieve redundancy, you can have a redundant server fleet at a different rack in the same data center and in a different region. If servers are spread across different physical locations, the first level of traffic routing can be handled using the **Domain Name System** (**DNS**) server before it reaches the load balancer:

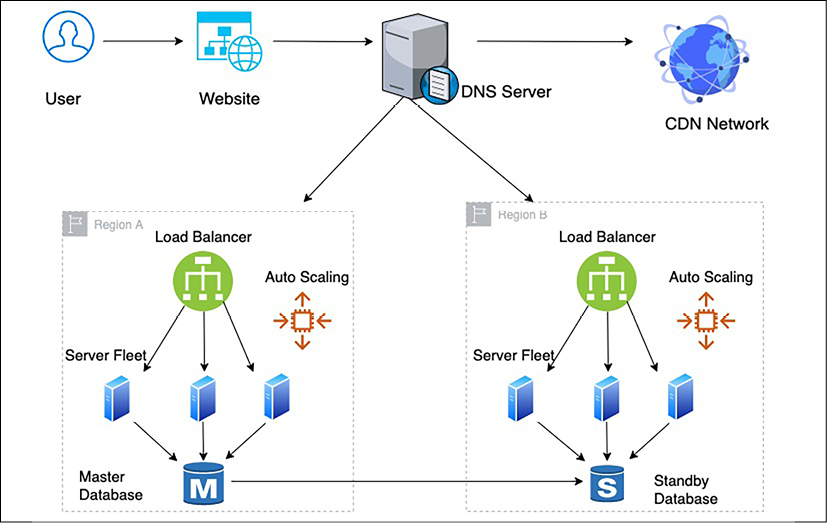


Figure 4.4: Application architecture resiliency

As you can see in the preceding architecture, resiliency needs to be applied in all the critical layers that affect the application's availability to implement the design of failure. To achieve resiliency, the following best practices need to be applied in order to create a redundant environment:

* Use the DNS server to route traffic between different physical locations so that your application will still be able to run in the case of an entire region failure.
* Use the CDN to distribute and cache static content such as videos, images, and static web pages near the user location so that your application will still be available in case of a DDoS attack or local **point of presence** (**PoP**) location failure.
* Once traffic reaches a region, use a load balancer to route traffic to a fleet of servers so that your application should still be able to run even if one location fails within your region.
* Use auto-scaling to add or remove servers based on user demand. As a result, your application should not get impacted by individual server failure.
* Create a standby database to ensure the high availability of the database, meaning that your application should be available in the event of a database failure.

In the preceding architecture, if any components fail, you should have a backup to recover it and achieve architecture resiliency. The load balancer and routers at the DNS server perform a health check to make sure that the traffic is only routed to healthy application instances. You can configure this to perform a shallow health check, which monitors local host failures, or deep health checks, which can also take care of dependency failure. However, a deep health check takes more time and is more resource-intensive than a shallow health check. You will learn more about resilient architecture in *Chapter 9*,*Architectural Reliability Considerations*.

At the application level, it is essential to avoid **cascade failure**, where the failure of one component can bring down the entire system. There are different mechanisms available to handle cascading, such as applying timeout, traffic rejection, implementing the **idempotent operation**, and using circuit-breaking patterns. You will learn more about these patterns in *Chapter 6*, *Solution Architecture Design Patterns*.

**Design for performance**

With the availability of fast internet, customers are seeking high-performance applications with minimal load time. Organizations have noticed that a direct revenue impact is proportional to application performance, and slowness in application load time can significantly impact customer engagement. Modern-era companies are setting a high expectation when it comes to performance, which results in high-performance applications becoming a necessity in order to stay in the market.

Like resiliency, the solution architect needs to consider performance at every layer of architecture design. The team needs to put monitoring in place to continue to perform effectively and work to improve upon it continuously. Better performance means increased user engagement and increases in the return on investment—high-performance applications are designed to handle application slowness due to external factors such as a slow internet connection. For example, you may have designed your blog web page to load within 500 milliseconds where there is good internet availability. However, where the internet is slow, you can load text first and engage the user with content while images and videos are still loading.

In an ideal environment, as your application workload increases, automated scaling mechanisms start handling additional requests without impacting application performance. But in the real world, your application latency goes down for a short duration when scaling takes effect. In a real-world situation, it's better to test your application for performance by increasing the load and understand if you can achieve the desired concurrency and user experience.

At the server level, you need to choose the right kind of server depending upon your workload. For example, choose the right amount of memory and compute to handle the workload, as memory congestion can slow down application performance, and eventually, the server may crash. For storage, it is important to choose the right **input/output operations per second** (**IOPS**). For write-intensive applications, you need high IOPS to reduce latency and to increase disk write speed.

To achieve higher performance, apply caching at every layer of your architecture design. Caching makes your data locally available to users or keeps data in-memory in order to serve an ultra-fast response. The following are the considerations that are required to add caching to various layers of your application design:

* Use the browser cache on the user's system to load frequently requested web pages
* Use the DNS cache for quick website lookup
* Use the CDN cache for high-resolution images and videos that are near the user's location
* At the server level, maximize the memory cache to serve user requests
* Use cache engines such as Redis and Memcached to serve frequent queries from the caching engine
* Use the database cache to serve frequent queries from memory
* Take care of cache expiration and cache eviction at every layer

As you can see, keeping your application performant is one of the essential design aspects and is directly related to organizational profitability. The solution architect needs to think about performance when creating a solution design and should work relentlessly to keep improving the performance of the application. In *Chapter 7*, *Performance Considerations*, you will dive deeper to learn techniques to optimize your application for better performance.

**Using replaceable resources**

Organizations make a significant capital investment in hardware, and they develop the practice of updating them with a new version of the application and configuration. Over time, this leads to different servers running in varied configurations, and troubleshooting them becomes a very tedious task. Sometimes, you have to keep running unnecessary resources when they are not needed, as you are not sure which server to shut down.

The inability to replace servers makes it challenging to roll out and test any new updates in your server fleet. These problems can be solved by treating your server as a replaceable resource, which enables you to move more quickly to accommodate changes such as upgrading applications and underlying software.

That is why, while designing your application, always think of immutable infrastructure.

**Creating immutable infrastructure**

*Immutable* means, during application upgrades, that you will not only replace software but hardware, too. Organizations make a significant capital investment in hardware and develop the practice of updating them with a new version of the application and configuration.

To create replaceable servers, you need to make your application stateless and avoid the hardcoding of any server IP or database DNS name. Basically, you need to apply the idea of treating your infrastructure as software instead of hardware, and not apply updates to the live system. You should always spin up new server instances from the golden machine image, which has all the necessary security and software in place.

Creating immutable infrastructure becomes more viable with the use of a virtual machine, where you can create a golden image of your virtual machine and deploy it with the new version, rather than trying to update an existing version. This deployment strategy is also beneficial for troubleshooting, where you can dispose of the server that has an issue and spin up a new server from a golden image.

You should take a backup of logs for root cause analysis before disposing of the server with issues. This approach also ensures consistency across the environment, as you are using the same baseline server image to create all of your environment.

Canary testing is one of the popular methods for ensuring that all changes are working as intended in the production environment before rolling out to broader users. Let's now learn more about canary testing.

**Canary testing**

**Canary testing** is one of the popular methods that is used to apply rolling deployment with immutable infrastructure. It helps you to ensure that old-version production servers are replaced safely with new servers without impacting end users. In canary testing, you deploy your software update in a new server and route a small amount of traffic to it.

If everything goes well, you will keep increasing traffic by adding more new servers, while disposing of old servers. Canary deployment gives you a safe option to push your changes in the live production environment. If something goes wrong, only small numbers of users are impacted, and you have the option of immediate recovery by routing traffic back to the old servers.

The solution architect needs to think ahead to use replaceable resources for deployment. They need to plan session management and avoid server dependency on hardcoded resources ahead of time. Always treat resources as replaceable and design your applications to support hardware changes.

The solution architect needs to set a standard to use various rolling deployment strategies, such as A/B testing or blue/green deployment. Treat your server like cattle, not like a pet; when this principle is applied to the replacement of problematic IT resources, quick recovery is ensured, and troubleshooting time is reduced.

**Think loose coupling**

A traditional application builds a tightly integrated server where each server has a specific responsibility. Often, applications depend upon other servers for completeness of functionality.

As shown in the following diagram, in a tightly coupled application, the web server fleet has a direct dependency on all application servers and vice versa:

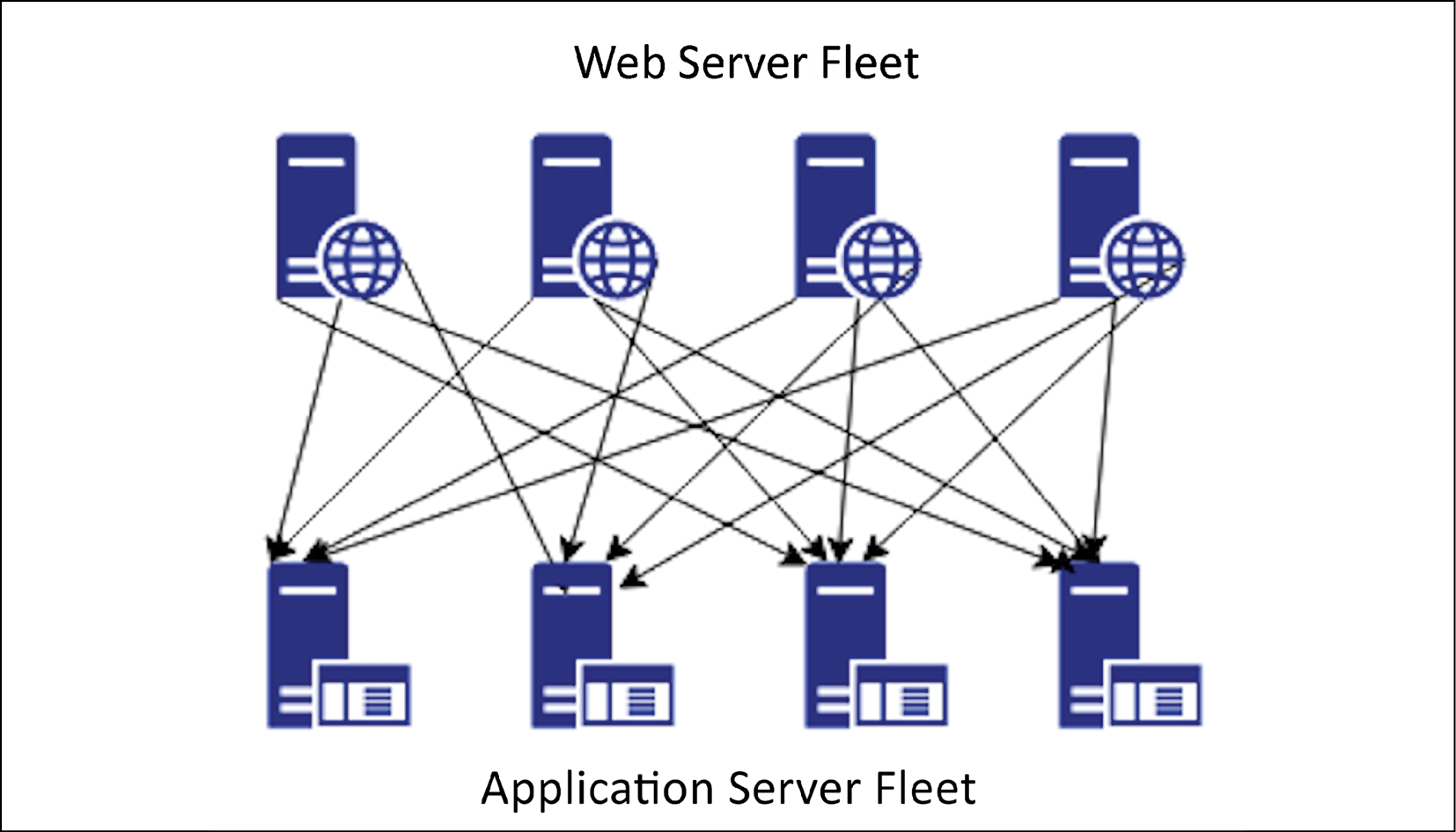


Figure 4.5: Tightly coupled architecture

In the preceding architecture diagram, if one application server goes down, then all web servers will start receiving errors, as the request will route to an unhealthy application server, which may cause a complete system failure. In this case, if you want to scale by adding and removing servers, it requires lots of work, as all connections need to be set up appropriately.

With loose coupling, you can add an intermediate layer such as a load balancer or a queue, which automatically handles failures or scaling for you.

In the following architecture diagram, there is a load balancer between the web server and the application server fleet, which makes sure to always serve user requests from a healthy application server:

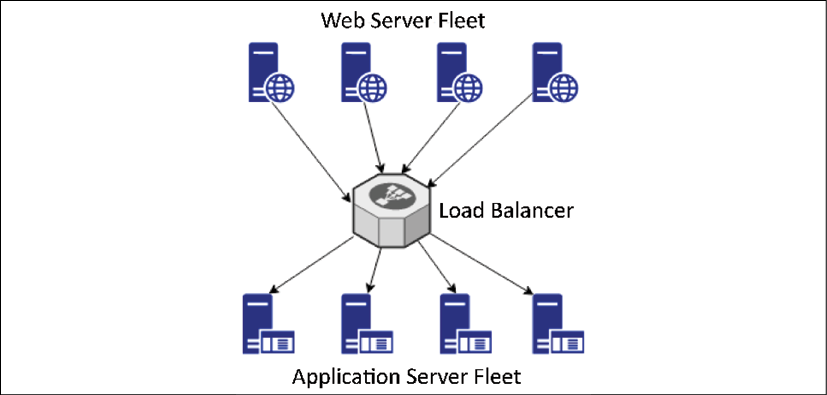


Figure 4.6: Load balancer-based, loosely coupled architecture

If one of the application servers goes down, the load balancer will automatically start directing all the traffic to the other three healthy servers. Loosely coupled architecture also helps you to scale your servers independently and replace unhealthy instances gracefully. It makes your application more fault-tolerant as an error radius is limited to a single instance only.

For a queue-based, loosely coupled architecture, take an example of an image-processing website, where you need to store an image, and then process it for encoding, thumbnails, and copyright. The following architecture diagram has queue-based decoupling. You can achieve loose coupling of systems by using queues between systems and exchanging messages that transfer jobs.

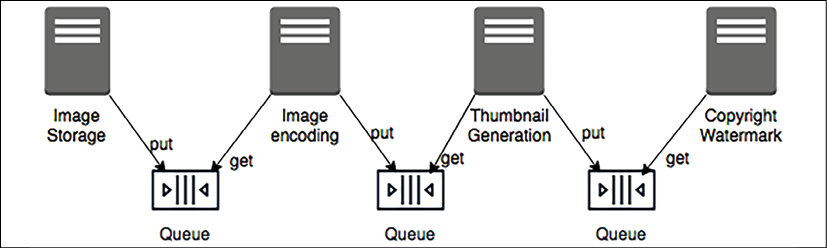


Figure 4.7: Queue-based, loosely coupled architecture

Queue-based decoupling enables the asynchronous linking of systems, where one server is not waiting for a response from another server and is working independently. This method lets you increase the number of virtual servers that receive and process the messages in parallel. If there is no image to process, you can configure auto-scaling to terminate the excess servers.

In a complex system, a loosely coupled architecture is achieved by creating a **service-oriented architecture** (**SOA**), where independent services contain a complete set of functionalities and communicate with each other over a standard protocol. In modern design, microservice architecture is becoming highly popular, which facilitates the decoupling of an application component. The loosely coupled design has many benefits, from providing scalability and high availability to ease of integration.

In the next section, you will learn more about SOA, and you will also dive deep into the details of this topic in *Chapter 6*, *Solution Architecture Design Pattern*.

**Think service not server**

In the previous section, you learned about loose coupling and how important it is for our architecture to be loosely coupled for scalability and fault tolerance. Developing service-oriented thinking will help to achieve a loosely coupled architecture (as opposed to a server-oriented design, which can lead to hardware dependency and a tightly coupled architecture). SOA helps us to achieve ease of deployment and maintenance for your solution design.

When it comes to service-oriented thinking, solution architects always tend toward SOA. The two most popular SOAs are based on **Simple Object Access Protocol** (**SOAP**) services and **Representational State Transfer** **(RESTful)** services. In SOAP-based architecture, you format your message in XML and send it over the internet using the SOAP protocol, which builds on top of the HTTP.

In a RESTful architecture, you can format a message in XML, JSON, or plain text, and send it over a simple HTTP. However, RESTful architecture is comparatively more popular as it is very lightweight and much more straightforward than SOAP.

When talking about SOA nowadays, microservice architecture is increasingly popular. Microservices are independently scalable, which makes it easier to expand or shrink one component of your application without impacting others.

As you can see in the following diagram, in a monolithic architecture, all components are built in a single server and tied up with a single database, which creates a hard dependency, whereas in a microservice architecture, each component is independent, with its own framework and database, which allows them to be scaled independently:

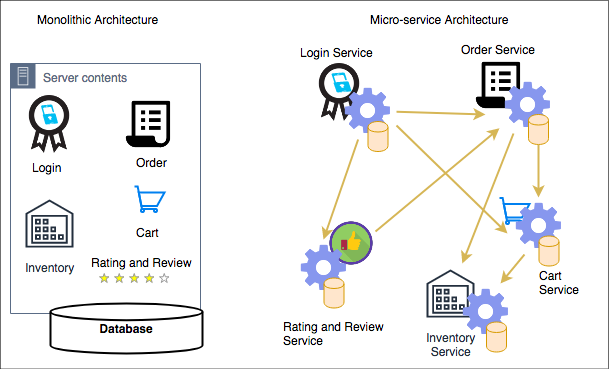


Figure 4.8: Monolithic and microservice architectures

In the preceding diagram, you can see an example of an e-commerce website where customers can log in and place an order, assuming the items they want are available, by adding items to the cart. To convert a monolithic architecture to a microservice-based architecture, you can create applications that are made of small independent components, which constitute smaller parts to iterate.

Taking the *modularization approach* means that the *cost, size, and risk of change* reduces. In the preceding case, each component is created as a service. Here, the **Login**service can independently scale to handle more traffic, as the customer may log in frequently to explore the product catalog and order status, while the **Order**service and the **Cart**service may have less traffic, as a customer may not place the order very often.

Solution architects need to think of microservices while designing a solution. The clear advantage of services is that you have a smaller surface area of code to maintain and services are self-contained. You can build them with no external dependencies. All prerequisites are included in the service, which enables loose coupling and scaling, and reduces the blast radius in case of failure.

**Using the right storage for the right requirements**

For decades, organizations have been using traditional relational databases and trying to fit everything there, whether it is key/value-based user session data, unstructured log data, or analytics data for a data warehouse. However, the truth is, the relational database is meant for transaction data, and it doesn't work very well for other data types—it's like using a Swiss Army knife, which has multiple tools that work but to a limited capacity; if you want to build a house, then the knife's screwdriver will not be able to perform a heavy lift. Similarly, for specific data needs, you should choose the right tool that can do the heavy lifting, and scale without compromising performance.

Solution architects need to consider multiple factors while choosing the data storage to match the right technology. Here are the important ones:

* **Durability requirement**: How should data be stored to prevent data corruption?
* **Data availability**: Which data storage system should be available to deliver data?
* **Latency requirement**: How fast should the data be available?
* **Data throughput**: What is the data read and write need?
* **Data size**: What is the data storage requirement?
* **Data load**: How many concurrent users need to be supported?
* **Data integrity**: How is the accuracy and consistency of data maintained?
* **Data queries**: What will be the nature of the queries?

In the following table, you can see different types of data with examples and appropriate storage types to use. Technology decisions need to be made based on storage type, as shown here:

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Data Example | Storage Type | Storage Example |
| Transactional, structured schema | User order data, financial transaction | Relational database | Amazon RDS, Oracle, MySQL, Amazon Aurora  PostgreSQL, MariaDB,  Microsoft SQL Server |
| Key/value pair, semi-structured, unstructured | User session data, application log, review, comments | NoSQL | Amazon DynamoDB, MongoDB, Apache HBase, Apache Cassandra, Azure Tables |
| Analytics | Sales data, supply chain intelligence, business flow | Data warehouse | IBM Netezza, Amazon Redshift, Teradata, Greenplum, Google BigQuery |
| In-memory | User home page data, common dashboard | Cache | Redis cache, Amazon ElastiCache, Memcached |
| Object | Image, video | File-based | SAN, Amazon S3, Azure Blob Storage, Google Storage |
| Block | Installable software | Block-based | NAS, Amazon EBS, Amazon EFS, Azure Disk Storage |
| Streaming | IoT sensor data, clickstream data | Temporary storage for streaming data | Apache Kafka, Amazon Kinesis, Spark Streaming, Apache Flink |
| Archive | Any kind of data | Archive storage | Amazon Glacier, magnetic tape storage, virtual tape library storage |
| Web storage | Static web contents such as images, videos, and HTML pages | CDN | Amazon CloudFront, Akamai CDN, Azure CDN, Google CDN, Cloudflare |
| Search | Product search, content search | Search index store and query | Amazon Elastic Search, Apache Solr, Apache Lucene |
| Data catalog | Table metadata, data about data | Metadata store | AWS Glue, Hive metastore, Informatica data catalog, Collibra data catalog |
| Monitoring | System log, network log, audit log | Monitor dashboard and alert | Splunk, Amazon CloudWatch, SumoLogic, Loggly |

As you can see in the preceding table, there are various properties of data, such as structured, semi-structured, unstructured, key/value pair, and streaming. Choosing the right storage helps to improve not only the performance of the application but also its scalability. For example, you can store user session data in the NoSQL database, which will allow application servers to scale horizontally and maintain user sessions at the same time.

While choosing storage options, you need to consider the temperature of the data, which could be hot, warm, or cold:

* For hot data, you are looking for sub-millisecond latency and the required cache data storage. Examples of hot data include stock trading and making product recommendations in runtime.
* For warm data, such as financial statement preparation or product performance reporting, acceptable latency might vary from seconds to minutes, and you should use a data warehouse or a relational database.
* For cold data, such as storing 3 years of financial records for audit purposes, you can plan latency in hours, and store it in archive storage.

Choosing appropriate storage as per the data temperature also saves costs in addition to achieving the performance SLA. As any solution design revolves around handling the data, so a solution architect always needs to understand their data thoroughly and then choose the right technology.

In this section, we have covered a high-level view of data to get an idea of using the proper storage according to the nature of the data. You will learn more about data engineering in *Chapter 13*, *Data Engineering* *for Solution Architecture*. Using the right tool for the right job helps to save costs and improve performance, so it's essential to choose the right data storage for the right requirements.

**Think data-driven design**

Any software solution revolves around the collection and management of data. Take the example of an e-commerce website; the software application is built to showcase product data on the website and encourage the customers to buy them. It starts by collecting customer data when they create a login, adding a payment method, storing order transactions, and maintaining inventory data as the product gets sold. Another example is a banking application, which is about storing customer financial information and handling all financial transaction data with integrity and consistency. For any application, the most important thing is to handle, store, and secure data appropriately.

In the previous section, you have learned about different kinds of data types, along with the storage needs, which should help you to apply data thinking in your design. Solution design is heavily influenced by data and enables you to apply the right design-driven solution by keeping data in mind. While designing a solution, if your application needs ultra-low latency, then you need to use cache storage such as **Redis** or **Memcached**. If your website needs to improve its page load time with an attractive high-quality image, then you need to use a content distribution network such as Amazon CloudFront or Akamai to store data near the user location. Similarly, to improve your application performance, you need to understand if your database will be read-heavy (such as a blog website) or write-heavy (such as a survey collection) and plan your design accordingly.

It's not just application design, but operational maintenance and business decisions all revolve around data. You need to add monitoring capabilities to make sure that your application, and in turn, your business, is running without any issues. For application monitoring, you collect log data from the server and create a dashboard to visualize the metrics.

Continuous data monitoring and sending alerts in the case of issues help you to recover quickly from failure by triggering the auto-healing mechanism. From a business perspective, collecting sales data helps you to run a marketing campaign to increase the overall business revenue. Analyzing review sentiment data helps to improve the customer experience and retain more customers, which is critical for any business. Collecting overall order data and feeding it to the machine learning algorithm helps you to forecast future growth and maintain the desired inventory.

As a solution architect, you are not only thinking about application design, but also about the overall business value proposition. It's about other factors around the application, which can help to increase customer satisfaction and maximize the return on your investment. Data is gold and getting insights into data can make a tremendous difference to an organization's profitability.

**Overcoming architectural constraints**

Earlier, in *Chapter 2*, *Solution Architects in an Organization*, you learned about the various constraints that a solution architecture needs to handle and balance. The major limitations are cost, time, budget, scope, schedule, and resources. Overcoming these constraints is one of the significant factors that needs to be considered while designing a solution. You should look at the limitations as challenges that can be overcome rather than obstacles, as challenges always push you to the limit of innovation in a positive way.

A solution architect needs to make suitable trade-offs while considering the constraints. For example, a high-performance application results in more cost when you need to add additional caching in multiple layers of architecture. However, sometimes cost is more important than performance, primarily if a system is used by the internal employees, which doesn't directly impact revenue. Sometimes, the market is more important than launching a fully featured product, and you need to make the trade-off between scope versus speed. In such scenarios, you can take the **minimum viable product** (**MVP**) approach; you will learn more details about this in the next section.

Technology constraints become evident in a large organization, as bringing changes across hundreds of systems will be challenging. When designing applications, you need to use the most common technique that is used across the organization, which will help to remove the everyday challenges. You also need to make sure that the application is upgradable in order to adopt new technology, and be able to plug in components that are built on a different platform.

A RESTful service model is pretty popular when teams are free to use any technology for their development. The only thing they need to provide is a URL with which their services can be accessed. Even legacy systems such as mainframes can be integrated into the new system using an API wrapper around it and overcome technology challenges.

Throughout this book, you will learn more about handling various architectural constraints. Taking an agile approach helps you to overcome constraints and build a customer-centric product. In design principles, take everything as a challenge and not an obstacle. Consider any constraint as a challenge and find a solution to solve it.

**Taking the minimum viable product approach**

For a successful solution, always put the customer first, while also taking care of architectural constraints. Think backward from the customers' needs, determine what is critical for them, and plan to put your solution delivery in an agile way. One popular method of prioritized requirement is **MoSCoW**, where you divide customer requirements into the following categories:

* **Mo (Must have)**: Requirements that are very critical for your customers, without which the product cannot launch
* **S (Should have)**: Requirements that are the most desirable to the customer, once they start utilizing the application
* **Co (Could have)**: Requirements that are nice to have, but their absence will not impact upon the desired functionality of the application
* **W** **(Won't have)**: Requirements that customers may not notice if they are not there

You need to plan an MVP for your customer with must-have requirements and go for the next iteration of delivery with should-have requirements. With this phased delivery approach, you can thoroughly utilize your resources and overcome the challenges of time, budget, scope, and resources. The MVP approach helps you to determine customer needs. You are not trying to build everything without knowing if the features you've built have added value for the customer. This customer-focused approach helps to utilize resources wisely and reduces the waste of resources.

In the following diagram, you can see the evaluation for a truck manufacturing delivery, where the customer wants a delivery truck that gets delivered initially, and you evolve the process based on the customer's requirements:

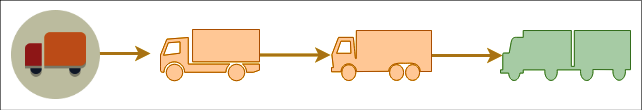


Figure 4.9: MVP approach to building the solution

Here, once a customer gets the first delivery truck, which is fully functioning, they can determine if they need a more significant load to handle, and based on that, the manufacturer can build a 6-wheel, a 10-wheel, and finally, an 18-wheel truck trailer. This stepwise approach provides working products with essential features that the customers can use, and the team can build upon it, as per customer requirements.

You can see how the MVP approach helps to utilize limited resources in an efficient way, which helps to buy more time and clarify the scope, in comparison to an approach where we turn up the first time with an 18-wheel truck, only to find out we only needed a 6-wheeler. In terms of the other factors, when you put the working product in the customer's hands early, it gives you an idea of where to invest. As your application has already started generating revenue, you can present use cases to ask for more resources as required.

**Adding security everywhere**

Security is one of the essential aspects of solution design; any gap in security can have a devastating effect on business and the organization's future.

The security aspect can have a significant impact on solution design, so you need to understand your security needs even before starting the application design. Security needs to be included in platform readiness at the hardware level and in application development at the software level. The following are the security aspects that need to be considered during the design phase:

* **Physical security of data center**: All IT resources in data centers should be secure from unauthorized access
* **Network security**: The network should be secure to prevent any unauthorized server access
* **Identity and Access Management** (**IAM)**: Only authenticated users should have access to the application, and they can do the activity as per their authorization
* **Data security in transit**: Data should be secure while traveling over the network or the internet
* **Data security at rest**: Data should be secure while stored in the database or any other storage
* **Security monitoring**: Any security incident should be captured, and the team alerted to act

Application design needs to balance security requirements such as encryption, and other factors such as performance and latency. Data encryption always has a performance impact as it adds a layer of additional processing because data needs to be decrypted in order to be utilized. Your application needs to accommodate the overhead of additional encryption processing without impacting overall performance. So, while designing your application, think of use cases where encryption is really required. For example, if the data is not confidential, you don't need to encrypt it.

The other aspect of application design to consider is regulatory compliance for adherence to local law. Compliance is essential if your application belongs to a regulated industry such as healthcare, finance, or the federal government. Each compliance has its requirement, which commonly includes the protection of data and the recording of each activity for audit purposes. Your application design should include comprehensive logging and monitoring, which will fulfill the audit requirement.

In this section, you have learned to apply security thinking while designing and keeping any regulatory needs in mind. Security automation is another factor, which you should always implement along with your design, so as to reduce and mitigate any security incidents. However, you have a high-level overview here. You will learn more details in *Chapter 8*, *Security Considerations*.

**Applying automation everywhere**

Most accidents happen due to human error, which can be avoided by **automation**. Automation not only handles jobs efficiently but also increases productivity and saves costs. Anything identified as a repeatable task should be automated to free up valuable human resources so that team members can spend their time on more exciting work and focus on solving a real problem. It also helps to increase team morale.

When designing a solution, think about what can be automated. Think to automate any repeatable task. Consider the following components to be automated in your solution:

* **Application testing**: You need to test your application every time you make any changes to make sure that nothing breaks. Also, manual testing is very time-consuming and requires lots of resources. It's better to think about automating repeatable test cases to speed up deployment and product launch. Automate your testing on a production scale and use rolling deployment techniques, such as canary testing and A/B testing, to release changes.
* **IT infrastructure**: You can automate your infrastructure by using *infrastructure as code* scripting, for example, Ansible, Terraform, and Amazon CloudFormation. The automation of infrastructure allows environments to be created in minutes compared to days. The automation of infrastructure as code helps to avoid configuration errors and creates a replica of the environment.
* **Logging, monitoring, and alerting**: Monitoring is a critical component, and you want to monitor everything every time. Also, based on monitoring, you may want to take automated action such as scaling up your system or alerting your team to act. You can monitor the vast system only by using automation. You need to automate all activity monitoring and logs to make sure that your application is running smoothly, and that it is functioning as desired.
* **Deployment automation**: Deployment is a repeatable task that is very time-consuming and delays the last-minute launch in many real-time scenarios. Automating your deployment pipeline by applying **continuous integration and continuous deployment** (**CI/CD**) helps you to be agile and iterate quickly on product features with a frequent launch. CI/CD helps you to make small incremental changes to your application.
* **Security automation**: While automating everything, don't forget to add automation for security. If someone is trying to hack your application, you want to know immediately and act quickly.

You want to take preventive action by automating any incoming or outgoing traffic in your system boundary and alert any suspicious activity.

Automation provides peace of mind by making sure the product is functioning without a glitch. While designing an application, always makes sure to think from an automation perspective and consider that as a critical component. You will learn more about automation in the coming chapters.

**Summary**

In this chapter, you learned about the various principles of solution architecture design that you need to apply when creating a solution design. These principles help you to take a multi-dimensional look into architecture and consider the important aspects for the success of the application.

You started the chapter with predictive and reactive patterns of scaling, along with the methodology and benefits. You also learned about building a resilient architecture that can withstand failure and recover quickly without impacting the customer experience.

Designing flexible architecture is the core of any design principle, and you learned about how to achieve a loosely coupled design in your architecture. SOA helps to build an architecture that can be easily scaled and integrated. You also learned about the microservice architecture, and how it is different from the traditional monolithic architecture, and its benefits.

You learned about the principle of data-focused design, as pretty much all applications revolve around data. You learned about different data types, with the example of storage and associated technology. Finally, you learned the design principle of security and automation, which applies everywhere and in all components.

As cloud-based services and architecture are becoming standard, in the next chapter, you will learn about cloud-native architecture and develop a cloud-oriented design. You will learn about different cloud migration strategies and how to create an effective hybrid cloud. You will also learn about the popular public cloud providers, with which you can explore cloud technologies further.