**7**

**Performance Considerations**

In this era of fast internet, users expect very high-performance applications. There have been experiments that show that every second of application load delay causes a significant loss in an organization's revenue. Therefore, the application's performance is one of the most critical attributes of solution design that can impact your product adoption growth.

In the previous chapter, you learned about various solution architecture design patterns that can be used to solve a complex business problem. In this chapter, you will understand the best practices to optimize your application for optimal performance. You will learn various design principles that you can use to optimize the solution architecture's performance. Here, performance needs to be optimized at every layer and in every component of the architecture.

You will understand how to choose the right technology at various layers of your architecture to improve your application's performance continuously. You will learn how to follow the best practices of performance optimization in this chapter. We will focus on the following topics in particular:

* Design principles for architecture performance
* Technology selection for performance optimization
* Performance monitoring

By the end of the chapter, you will understand important attributes of performance improvement, such as latency, throughput, and concurrency. You will be able to make better decisions regarding your choice of technology, which can help you to improve performance at the various layers of architecture, such as compute, storage, database, and networking.

**Design principles for architecture performance**

Architectural performance efficiency focuses on using application infrastructure and resources to meet increasing demand and technology evaluation. Technology vendors and open-source communities continuously work to improve the performance of applications. Often, large enterprises continue to work on legacy programming languages and technologies because of fear of changing and taking risks. As technology evolves, it often addresses critical performance issues, and the advancement of technology in your application helps improve application performance.

Many large public cloud providers, such as **Amazon Web Services** (**AWS**), Microsoft Azure, and **Google Cloud Platform** (**GCP**), offer technology as a service. This makes it easier to adopt complex technologies more efficiently with minimal effort—for example, you might use storage as a service to manage a massive amount of data or a NoSQL database as a managed service to provide high-performance scalability to your application.

Now organizations can utilize a **content distribution network** (**CDN**) to store heavy image and video data near user locations to reduce network latency and improve performance. With Edge locations, it becomes easier to deploy workloads closer to your user base, which helps to optimize application performance by reducing latency over the network.

As servers virtualize, you can be more agile and experiment with your application, and you can apply a high degree of automation. Agility helps you experiment and determine what technology and method are best suited for your application workload. For example, you can choose if your server deployment should go for virtual machines, containers, or use serverless with AWS Lambda, a **Function as a Service** (**FaaS**). Let's look at some vital design principles to consider for your workload performance optimization.

**Reducing latency**

Latency can be a significant factor in your product adoption, as users are looking for faster applications. It doesn't matter where your users are located, you need to provide a reliable service for your product to grow. You may not achieve zero latency, but the goal should be to reduce the response time to within the user tolerance limit.

Latency is the time delay between the user sending a request and receiving the desired response.

The following diagram shows an example where it takes 600ms for a client to send a request to the server and 900ms for the server to respond, which introduces a total latency of 1.5 seconds (1500ms):

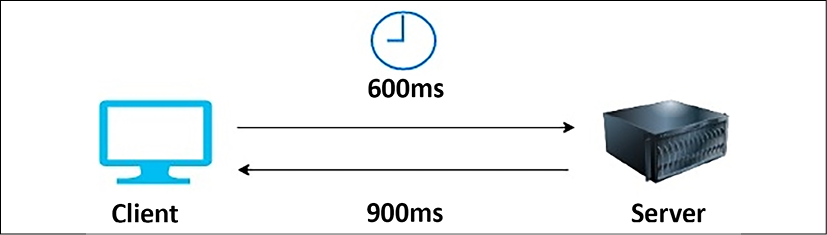


Figure 7.1: Request-response latency in a client-server model

Now, any application needs to access the internet to have a diverse set of global users. These users expect consistency in performance, regardless of their geographical location. It is sometimes challenging, as it takes time to move data over the network from one part of the world to another.

Network latency can be caused by various factors, such as the **network transmission medium**, **router hops**, and **network propagation**. Often, a request that is sent over the internet hops over multiple routers, which adds latency. Enterprises commonly use their fiber-optic line to set up connectivity between their corporate network and cloud, which helps avoid inconsistency.

In addition to the problems caused by the network, latency can occur in various components of the architecture. Your compute server can have latency issues at the infrastructure level due to memory and processor problems, where the data transfer between the CPU and RAM is *slow*. The disk can have latency due to slow read and write processes. Latency in a **hard disk drive** (**HDD**) is dependent on the time it takes to select a disk memory sector to come around and position itself under the head for reading and writing.

The disk memory sector is the physical location of data in the memory disk. In an HDD, data is distributed in memory sectors during write operations, as the disk is continuously rotating, so data can be written randomly. During the read operation, the head needs to wait for the rotation to bring it to the disk memory sector.

At the database level, latency can be caused by slow data reads and writes from the database due to hardware bottlenecks or slow query processing. Taking the database load off by distributing the data with partitioning and sharding can help to reduce latency.

There could be an issue with transaction processing from code that needs to be handled using garbage collection and multithreading at the application level. Achieving low latency means *higher throughput*, as latency and throughput are directly related, so let's learn more about throughput.

**Improving throughput**

Network throughput is the quantity of data sent and received at a given time. At the same time, latency is defined when the user initiates a request in the application and gets the response. When it comes to networks, *bandwidth* plays an important role.

Bandwidth determines the maximum amount of data that can get transferred over the network.

Throughput and latency have a direct relationship as they work together. Lower latency means high throughput as more data can transfer in less time. To understand this better, let's take the analogy of a country's transportation infrastructure.

Let's say that highways with lanes are network pipelines and cars are data packets. Suppose a given highway has 16 lanes between 2 cities. Not all vehicles can reach the destination at the desired time; they may get delayed because of traffic congestion, lanes closing, or accidents. Here, latency determines how fast a car can travel from one city to another, while throughput tells us how many cars can reach their destinations. For a network, using full bandwidth is challenging because of errors and traffic congestion.

Network throughput is measured by the amount of data sent over the network in **bits per second** (**bps**). Network bandwidth is the maximum size of the network pipeline that it can process. The following diagram illustrates the amount of data transferred between the client and the server:

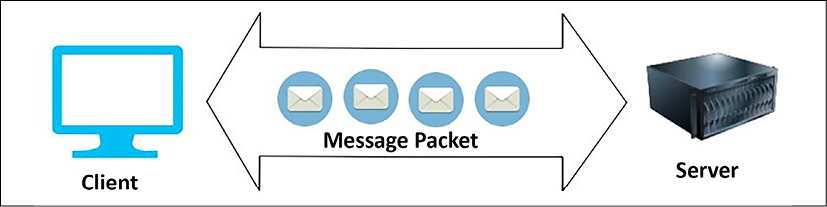


Figure 7.2: Throughput in a network

In addition to the network, throughput is applicable at the disk level. Disk throughput is determined by a factor of **input/output operations per second** (**IOPS**) and the amount of data requested (I/O size). Disk throughput is determined in **megabytes per second** (**Mbps**) using the following formula:

Average I/O size \* I/OPS = Throughput in MB/s

So, if your disk IOPS is 20,000 and the I/O size is 4 KB (4,096 bytes), then the throughput will be 81.9 MB/s (20,000 x 4,096 and converted from bytes to megabytes).

I/O requests and disk latency have a direct relationship. **I/O** means write and read, respectively, while **disk latency** is the time taken by each I/O request to receive the response from the disk. Latency is measured in milliseconds and should be minimal. It is impacted by disk **revolutions per minute** (**RPM**). **IOPS** is the number of operations that the disk can serve per second.

At the operating system level, throughput is determined by the amount of data transfer between the CPU and RAM per second. At the database level, throughput is determined by the number of transactions a database can process per second. At the application level, your code needs to handle transactions that can be processed every second by managing the application memory with the help of garbage collection handling and efficient use of the memory cache.

When you look at latency, throughput, and bandwidth, there is another factor called concurrency, which applies to the various components of architecture and helps to improve application performance. Let's learn more about concurrency.

**Handling concurrency**

**Concurrency** is a critical factor for solution design as you want your application to process multiple tasks at a time. For example, your application needs to handle multiple users simultaneously and process their requests in the background. Another example is when your web user interface needs to collect and process web cookie data to understand user interaction with the product while showing users their profile information and the product catalog. Concurrency is about doing multiple tasks at the same time.

People often get confused between parallelism and concurrency by thinking they are both the same thing; however, concurrency is different from parallelism. In parallelism, your application divides an enormous task into smaller subtasks, which it can process in parallel with a dedicated resource for each subtask. In concurrency, however, an application processes multiple tasks simultaneously by utilizing shared resources among the threads.

The application can switch from one task to another during processing, which means that the critical section of code needs to be managed using **locks** and **semaphores**.

As illustrated in the following diagram, concurrency is like a traffic light signal where the traffic flow switches between all four lanes to keep traffic going. As there is a single thread along which you should pass all traffic, processing in other lanes has to stop while traffic in one lane is in the *clearing process*. In the case of parallelism, there is a parallel lane available, and all cars can run in parallel without interrupting each other, as shown in the following diagram:

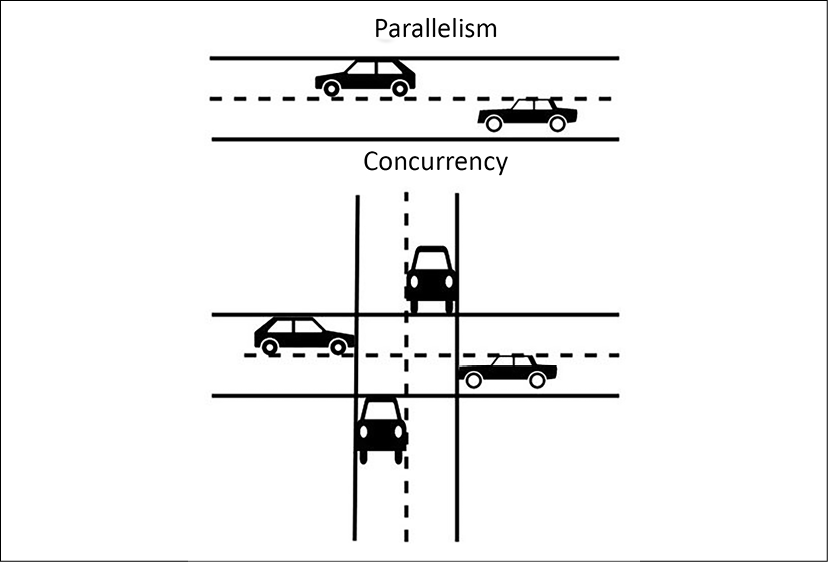


Figure 7.3: Concurrency versus parallelism

In addition to transaction processing at the application level, concurrency needs to apply at the network level where multiple servers share the same network resources. There is a need to handle many network connections for a web server when users try to connect to it over the network. It needs to process the active request and close the connection for the completed or timeout request. At the server level, you will see multiple CPUs assigned or a multicore processor. These help in handling concurrency as the server can handle more threads to complete various tasks simultaneously.

At the memory level, the shared memory concurrency model helps to achieve concurrency. In this model, the concurrent modules interact with each other using shared memory. It could be two programs running on the same server and sharing filesystems to read and write. Also, there could be two processors or processor cores sharing the same memory. The disk in your server can encounter concurrency situations where two programs try to write to the same memory block. Concurrent I/O helps to improve disk concurrency by allowing the disk to read and write a file simultaneously.

The database is always a central point of architecture design. Concurrency plays an essential role in data handling as the database should have the ability to respond to multiple requests simultaneously. Database concurrency is more complicated as one user might be trying to read a record while another user is updating it simultaneously. The database should only allow data viewing when it gets fully saved. Make sure that the data is completely committed before another user tries to update it. Caching can help to improve performance significantly; let's learn about some different cache types in architecture.

**Applying caching**

In *Chapter 6*, *Solution Architecture Design Patterns*, you learned how to apply caching at various levels of architecture in the *Cache-based architecture* section. Caching helps to improve application performance significantly. Although you learned the different design patterns to apply to the cache by adding an external caching engine and technology such as a CDN, it's essential to understand that almost every application component and infrastructure has a cache mechanism. Utilizing the caching mechanism at each layer can help reduce latency and improve the application's performance.

The CPU has its hardware cache at the server level, which reduces the latency when accessing data from the main memory. The CPU cache includes the instruction and data cache; the data cache stores copies of frequently used data. The cache is also applied at the disk level, but it is managed by operating system software (known as the **page cache**); however, the CPU cache is entirely managed by hardware. The disk cache originates from secondary storage, such as the HDD or **solid-state drive** (**SSD**). Frequently used data is stored in an unused portion of the main memory (that is, the RAM as the page cache, which results in quicker access to content).

Often, the database has a cache mechanism that saves the results from the database to respond faster. The database has an internal cache that gets data ready in the cache based on the pattern of your use. They also have a query cache that saves data in the main server memory (RAM) if you make a query more than once. The query cache gets cleared in case of any changes in the data inside the table. If the server runs out of memory, the oldest query result gets deleted to make space.

You have a DNS cache at the network level, which stores the web domain name and corresponding IP address local to the server. DNS caching allows a quick DNS lookup if you revisit the same website domain name. The DNS cache is managed by the operating system and contains all recent visits to websites. You learned about client-side cache mechanisms such as the **browser cache** and various caching engines like **Memcached** and **Redis** in *Chapter 6*, *Solution Architecture Design Patterns*.

In this section, you learned about the original design factors, such as latency, throughput, concurrency, and caching, which need to be addressed for architecture performance optimization. Each component of the architecture (whether it is a network at the server level or an application at the database level) has a certain degree of latency and a concurrency issue that needs to be handled.

You should design your application for the desired performance, as improving performance comes with a cost. The specifics of performance optimization may differ from application to application. Solution architecture needs to direct the effort accordingly—for example, a stock-trading application cannot tolerate even sub-millisecond latency. On the other hand, an e-commerce website can live with a couple of seconds' latency. Let's learn about selecting technology for various architecture levels to overcome performance challenges.

**Technology selection for performance optimization**

In *Chapter 6*, *Solution Architecture Design Patterns*, you learned about various design patterns, including microservice, event-driven, cached-based, and stateless. An organization may choose a combination of these design patterns depending on their solution's design needs. You can have multiple approaches to architecture design depending on your workload. Once you finalize your strategy and start solution implementation, the next step is to optimize your application. To optimize your application, you need to collect data by performing load testing and defining benchmarking as per your application's performance requirements.

Performance optimization is a continuous improvement process, one in which you need to take cognizance of optimal resource utilization from the beginning of solution design to after the application's launch. You need to choose the right resources as per the workload or tweak the application and infrastructure configuration. For example, you may want to select a NoSQL database to store the session state for your application and store transactions in the relational database.

For analytics and reporting purposes, you can offload your production database by loading data from the application database to data warehousing solutions and create reports from there.

In the case of servers, you may want to choose a virtual machine or containers. You can take an entirely serverless approach to build and deploy your application code. Regardless of your approach and application workload, you need to choose the primary resource type: computing, storage, database, and network. Let's look at more details on how to select these resource types for performance optimization.

**Making a computational choice**

In this section, you will see the use of the term *compute* instead of *the server*, as nowadays software deployments are not limited to servers. A public cloud provider such as AWS has serverless offerings, where you don't need a server to run your application. One of the most popular FaaS offerings is AWS Lambda. Like AWS Lambda, other popular public cloud providers extend their offerings in the FaaS space—for example, Microsoft Azure has Azure Functions, and GCP offers Google Cloud Functions.

However, organizations still make the default choice to go for servers with virtual machines. Now, containers are also becoming popular as the need for automation and resource utilization is increased. Containers are becoming the preferred choice, especially in the area of microservice application deployment. The optimal choice of computing—whether you want to choose server instances, containers, or go for serverless—depends upon application use cases. Let's look at the various compute choices available.

**Selecting the server instance**

Nowadays, the term **instance** is getting more popular as virtual servers become the norm. These virtual servers provide flexibility and better use of resources. Particularly for cloud offerings, all cloud providers offer virtual servers, which can be provisioned with a mere click on a web console or API call. The server instance helps in automation and provides *infrastructure as code*, where everything can be automated everywhere.

As your workload varies, you might prefer one of the different kinds of processing unit choices available. Let's look at some of the most popular options for processing power:

* **The central processing unit** (**CPU**): The CPU is one of the most popular computing processing choices. CPUs are easy to program, enable multitasking, and, most importantly, are versatile enough to fit anywhere, making them the preferred choice for general-purpose applications. The CPU's function is measured in GHz (gigahertz), which indicates that the clock rate of the CPU speed is in billions of cycles per second.

CPUs are available at a low cost; however, they cannot perform well for parallel processing as CPUs have the primary capabilities of sequential processing.

* **The graphical processing unit** (**GPU**): As the name suggests, the GPU was designed initially to process graphics applications and provide massive processing power. As the volume of data grows, it needs to process data by utilizing **massively parallel processing** (**MPP**). For large data processing use cases, such as machine learning, GPUs have become the obvious choice and are used in many compute-intensive applications. You may have heard of the **tera floating-point operation** (**TFLOP**) as a unit of computation power for GPUs. A teraflop refers to the processor's capability to calculate one trillion floating-point operations per second.

GPUs consist of thousands of small cores, compared to CPUs, which have very few large cores. GPUs have a mechanism to create thousands of threads using CUDA programming, and each thread can process data in parallel, which makes processing super fast. GPUs are a bit costlier than CPUs. When it comes to processing capabilities, you will find that GPUs are in the sweet spot of cost and performance for an application that requires image analysis, video processing, and signal processing. However, they consume lots of power and may not work with a specific algorithm where more customized processors are required.

* **Field-programmable gate array** (**FPGA**): FPGAs are very different from CPUs or GPUs. They are programmable hardware with a flexible collection of logic elements that can be reconfigured for the specific application, which can be changed after installation. FPGAs consume much less power than GPUs but are also less flexible. They can accommodate MPP and also provide a feature to configure them as CPUs. Overall, the FPGA cost is higher, as they need to be customized for each application and require a longer development cycle. FPGAs may perform poorly for sequential operations and are not very good for flops (floating-point operations).
* **Application-specific integrated circuit** (**ASIC**): ASICs are purpose-built custom integrated circuit optimization for a specific application—for example, specific to the deep learning TensorFlow package, which Google provides as a **tensor processing unit** (**TPU**). They can be custom designed for the applications to achieve an optimum combination of power consumption and performance. ASICs incur high costs because of the most extended development cycle, and you have to perform a hardware-level redesign for any changes.

The following diagram shows a comparison between the types of processing mentioned in the preceding list. Here, the ASIC is most efficient but takes a longer development cycle to implement. ASICs provide the most optimal performance but have the least flexibility to reutilize, while CPUs are very flexible and can fit many use cases:

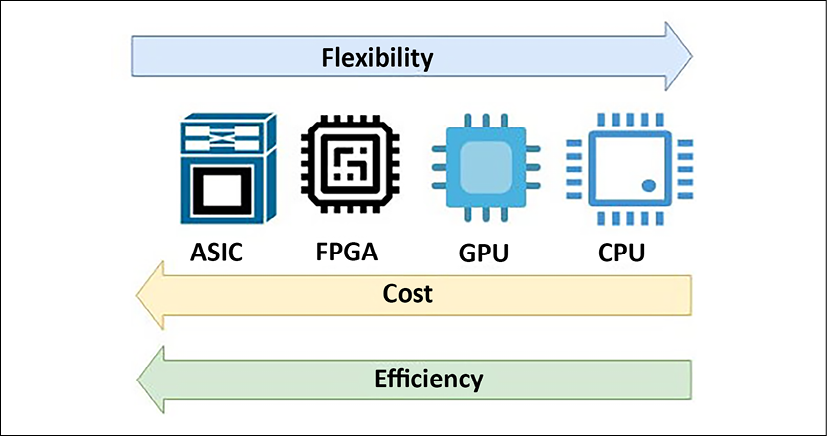


Figure 7.4: Comparison between CPUs, GPUs, FPGAs, and ASICs

As shown in the preceding diagram, from the cost perspective, CPUs are the cheapest, and ASICs are the costliest. Today, the CPU has become a commodity and is used everywhere for general-purpose devices to keep costs lower. The GPU has become famous for compute-intensive applications, and the FPGA has become the first choice where more customized performance is required. You will see these processing choices available from public cloud providers, such as AWS. In addition to CPUs, the **Amazon Elastic Cloud Compute** (**EC2**) offering provides a P-series instance that makes heavy use of the GPU. The F-series instance provides FPGAs for custom hardware acceleration.

In this section, you learned about the most popular computing choices. You may hear about other types of processors, such as the **accelerated processing unit** (**APU**). The APU combines the CPU, GPU, and **digital signal processor** (**DSP**), which is optimized to analyze analog signals and then requires high-speed data processing in real time. Let's learn more about other popular compute-type containers that are gaining popularity rapidly because of their capability to optimize the use of resources within the virtual machine.

**Working with containers**

In *Chapter 6*, *Solution Architecture Design Patterns*, you learned about container deployment and its benefits in the section titled *Deploying an application with a container*. The use of containers is becoming the norm for deploying complex microservice applications because of the ease of automation and resource utilization efficiency. There are various platforms available for container deployment.

Because of their popularity and platform-independent capabilities, containers become the first choice to build a cloud-agnostic platform. You can deploy containers in your on-premise data center and manage them through your cloud. Also, you can take a relocate approach to move a container from on-prem to the cloud without any changes.

You can build a multi-cloud platform with a container, and now each major public cloud vendor provides tools to manage a container environment spread over multiple platforms. For example, AWS provides **ECS Anywhere**, which enables you to run and manage container workloads on customer-managed infrastructure easily. Similarly, GCP provides **Google Anthos**, which gives you container management across on-premise and other cloud platforms. Let's learn about some of the most popular choices in the container area, their differences, and how they work together.

**Docker**

Docker is one of the most in-demand technologies. It allows you to package an application and its related dependencies together as a container and deploy it to any operating system platform. Docker provides platform-independent capabilities to a software application, making the overall software development, testing, and deployment process simplified and more accessible.

Docker container images are portable from one system to another over a local network or across the internet using Docker Hub. You can manage and distribute your container using a Docker Hub container repository. If you make any changes in the Docker image that cause issues in your environment, it's easy to revert to the working version of the container image, making overall troubleshooting easier.

Docker containers help you to build a more complex multilayer application. For example, suppose you need to run the application server, database, and message queue together.

In that case, you can run them side by side using a different Docker image and then establish communication between them. Each of these layers may have a modified version of libraries, and Docker allows them to run on the same computing machine without conflict.

When using Docker, the development team builds an application and packages it with required dependencies into a container image. This application image is run in a container on the Docker host. Like you manage code in a code repository such as GitHub, in the same way, a Docker image should be stored in a registry. Docker Hub is a public registry, and other public cloud vendors provide their own registries, such as **AWS ECR** (**Elastic Container Registry**) and **Azure Container Registry**. In addition, you can have a private registry on-premises for your own Docker images.

Public cloud providers, such as AWS, provide container management platforms, such as **Amazon Elastic Container Service** (**ECS**). Container management helps to manage Docker containers on top of the cloud virtual machine, Amazon EC2. AWS also provides the serverless option of container deployment using Amazon Fargate, where you can deploy containers without provisioning virtual machines.

Complex enterprise applications are built based on microservices that may span across multiple containers. Managing various Docker containers as a part of one application can be pretty complicated. Kubernetes helps to solve the challenges of the multi-container environment; let's learn more about Kubernetes.

**Kubernetes**

Kubernetes can manage and control multiple containers in production environments with ease. You can consider Kubernetes as a container orchestration system. You can host a Docker container in bare metal (physical server) or a virtual machine node called a Docker host, and Kubernetes can co-ordinate across a cluster of these nodes.

Kubernetes makes your application self-healing by replacing unresponsive containers in the case of any application error. It also provides horizontal scaling capabilities and a blue-green deployment ability to prevent any downtime. Kubernetes distributes incoming user traffic load between the container and manages the storage shared by various containers.

As shown in the following diagram, Kubernetes and Docker work well together to orchestrate your software application. Kubernetes handles network communication between Docker nodes and Docker containers:

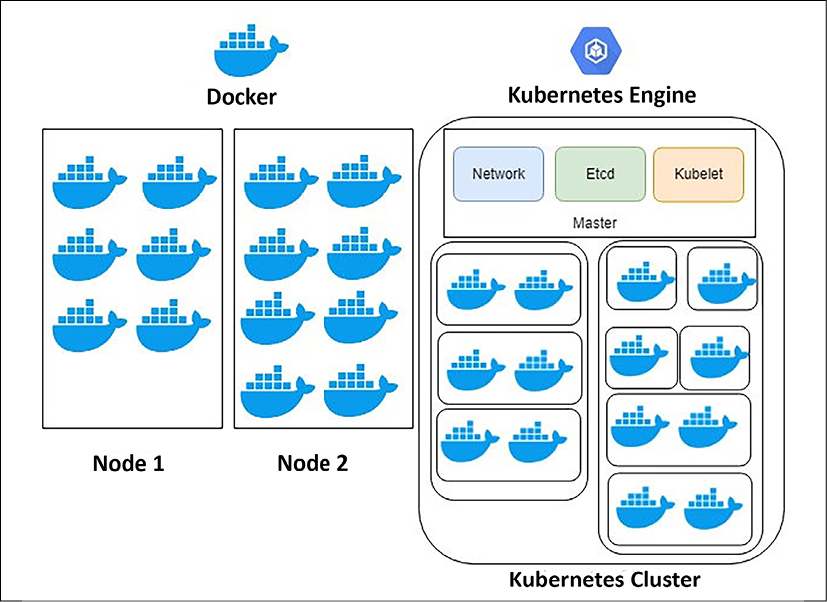


Figure 7.5: Docker and Kubernetes

Docker works as an individual piece of the application, and Kubernetes takes care of the orchestration to make sure all these pieces work together as designed. It's easier to automate overall application deployment and scaling with Kubernetes. In Docker, containers are hosted in nodes, and each Docker container in a single node shares the same IP space. In Docker, you need to manage the connections between containers by taking care of any IP conflict. Kubernetes solves this problem by having a master instance that keeps track of all nodes hosting containers.

Kubernetes's master node is responsible for assigning an IP address and hosting a key-value store for container configuration and a **kubelet** to manage the containers. The kubelet is the primary "node agent" that runs on each node and ensures that the containers defined in the pods are started and continue running. Docker containers are grouped into **pods**, where they share the same IP address. This entire setup is called a **Kubernetes cluster**.

While Kubernetes is quickly becoming popular, other options are available, such as **Docker Swarm**, which Docker itself builds. Docker Swarm is a **container orchestration tool**, which allows the user to manage multiple containers deployed across multiple host machines. However, Swarm doesn't have a web-based interface like Kubernetes and does not provide auto scaling and external load balancing.

Kubernetes is complex to learn. A public cloud provider, such as AWS, provides Amazon **Elastic Kubernetes Service** (**EKS**) to simplify the management of the Kubernetes cluster. OpenShift is another Kubernetes distribution managed by Red Hat and is offered as a **Platform as a Service** (**PaaS**). Similarly, Microsoft Azure provides **Azure Kubernetes Service** (**AKS**) and GCP provides **Google Kubernetes Engine** (**GKE**), offering a simple way to automatically deploy, scale, and manage Kubernetes.

Overall, containers add a layer of virtualization to the whole application infrastructure. While they are helpful in resource utilization, you may want to choose a bare-metal physical machine for your application deployment if it requires ultra-low latency.

**Going serverless**

In recent years, serverless computing has become possible because of the popularity of public cloud offerings by cloud providers such as Amazon, Google, and Microsoft. Serverless computing allows developers to focus on their code and application development without worrying about underlying infrastructure provisioning, configuration, and scaling. Serverless offerings abstract server management and infrastructure decisions from developers, and let them focus on their area of expertise and the business problem they are trying to solve. Serverless computing brings the relatively new concept of **Function as a Service** (**FaaS**).

FaaS offerings are available using AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions. You can write your code in the cloud editor, and AWS Lambda handles the computing infrastructure underneath to run and scale your function. You can design event-based architecture or RESTful microservices by adding an API endpoint using Amazon API Gateway and AWS Lambda functions. Amazon API Gateway is a managed cloud service that adds RESTful APIs and WebSocket APIs as frontends for the Lambda functions and enables real-time communication between applications. You can further break your microservice into small tasks that can be scaled automatically and independently.

In addition to focusing on your code, you never have to pay for idle resources in the FaaS model. Rather than scaling your entire service, you can scale the required functions independently with built-in availability and fault tolerance.

However, it could be a pretty daunting task if you have thousands of features to orchestrate, and predicting the auto scaling cost can be tricky. It is perfect for scheduling jobs, processing web requests, or queuing messages.

In this section, you learned about the various computing choices, looking at server instances, serverless options, and containers. You need to select these compute services based on your application's requirements. No rule forces you to choose a particular type of computing; it is all about your organization's choice of technology, the pace of innovation, and the nature of the software application.

However, in general, you can stick to a virtual or bare-metal machine for the monolithic application, and for complex microservices, you can choose containers. For simple task scheduling or events-based applications, you can go for serverless functions as an obvious choice. Many organizations have built complex applications entirely on serverless, which helped them reduce costs and achieve high availability without managing any infrastructure.

Let's learn about another important aspect of your infrastructure and how it can help you to optimize performance.

**Choosing storage**

Storage is one of the critical factors for your application's performance. Any software application needs to interact with storage for installation, logging, and accessing files. The optimal solution for your storage will differ based on the following factors:

|  |  |
| --- | --- |
| Access methods | Block, file, or object |
| Access patterns | Sequential or random |
| Access frequency | Online (hot), offline (warm), or archival (cold) |
| Update frequency | **Write once read many** (**WORM**) or dynamic |
| Access availability | Availability of storage when required |
| Access durability | Reliability of data store to minimize any data loss |
| Access throughput | IOPS and data read/write per second in MBs. |

These depend upon your data format and scalability needs. You first need to decide whether your data will be stored in block, file, or object storage. These are storage formats that store and present data in a different way. Let's look at this in more detail.

**Working with block storage and storage area network**

Block storage divides data into blocks and stores them as chunks of data. Each block has a unique ID that allows the system to place data wherever it is most easily accessible as blocks don't store any metadata about files. Hence, a server-based operating system manages and uses these blocks in the hard drive. Whenever the system requests data, the storage system collects the blocks and gives the result back to the user. Block storage deployed in a **storage area network** (**SAN**) stores data efficiently and reliably. It works well when a large amount of data needs to be stored and accessed frequently—for example, database deployment, email servers, application deployment, and virtual machines.

SAN storage is sophisticated and supports *complex, mission-critical applications*. It is a high-performance storage system that communicates block-level data between the server and storage; however, SAN is significantly costly and should be used for large-scale enterprise applications where low latency is required.

To configure your block-based storage, you must choose between an SSD and an HDD. HDDs are the legacy data storage system for servers and enterprise storage arrays. HDDs are cheaper, but they are slower and need more power and cooling. SSDs use semiconductor chips and are faster than HDDs. They are much more costly; however, as technology evolves, SSDs have become more affordable and gain popularity because of their efficiency and lower power and cooling requirements.

**Working with file storage and network area storage**

File storage has been around for a long time and is widely used. In file storage, data is stored as a single piece of information and is organized inside folders. When you need to access the data, you provide the file path and get the data files; however, a file path can grow complicated as files become nested under multiple folder hierarchies. Each record has limited metadata, including the filename, time of creation, and updated timestamps. You can take the analogy of a book library where you store books in drawers and keep a note of the location where each book is stored.

**Network area storage** (**NAS**) is a file storage system that is attached to the network and displays to the user where they can store and access their files. NAS also manages user privilege, file locking, and other security mechanisms that protect the data. NAS works well for file-sharing systems and local archives. When it comes to storing billions of files, NAS might not be the right solution, given that it has limited metadata information and a complex folder hierarchy. To store billions of files, you need to use object storage. Let's learn more about object storage and its benefits over file storage.

**Working with object storage and cloud data storage**

Object storage bundles data with a unique identifier and metadata that is customizable. Object storage uses a flat address space compared to the hierarchical addresses in file storage or addresses distributed over a chunk of blocks in block storage. Flat address space makes it easier to locate data and retrieve it faster regardless of the data storage's location. Object storage also helps the user to achieve unlimited scalability of storage.

Object storage metadata can have lots of details such as object name, size, timestamp, and so on, and users can customize it to add more information than tagging in file storage. Data can be accessed by a simple API call and is very cost-effective to store. Object storage performs best for high-volume, unstructured data; however, objects cannot be modified but only replaced, making it not a good use case for a database.

Cloud data storage, such as **Amazon Simple Storage Service** (**S3**), provides an unlimited scalable object data store with high availability and durability. You can access data with a unique global identifier and metadata file prefix. The following diagram shows all three storage systems in a nutshell:

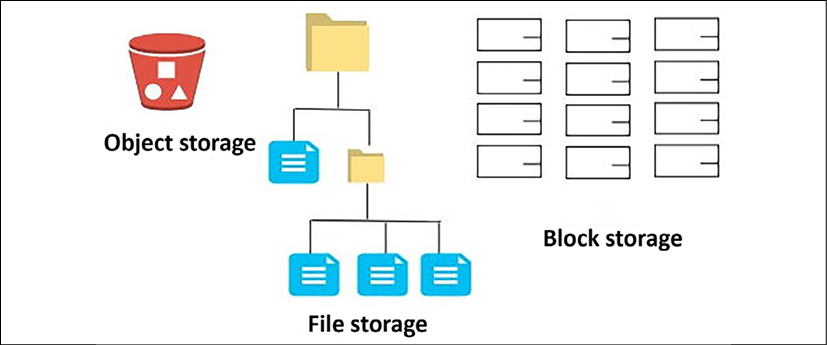


Figure 7.6: Data storage systems

As shown in the preceding diagram, block storage stores data in blocks. You should use block storage when your application needs very low latency and data storage access by a single instance. File storage stores data in a hierarchical folder structure and has little latency overhead. You should use the file storage system when a separate room needs to access multiple instances. Object storage stores data in buckets with a unique identifier for the object. It provides access over the web to reduce latency and increase throughput. You should use object storage to store and access static content, such as images and videos. You can store a high volume of data in the object store and perform big data processing and analysis.

**Direct-attached storage** (**DAS**) is another kind of data storage that is directly attached to the host server; however, it has very limited scalability and storage capacity. The **magnetic tape drive** is another popular storage system for backing up and archiving. Because of its low cost and high availability, magnetic tape drives are used for archival purposes but have high latency, making them unsuitable for use in direct applications.

Often, you will need to increase throughput and data protection for a mission-critical application, such as a transactional database, where data is stored in SAN storage; however, individual SAN storage may have limited volume and throughput. You can overcome this situation using a **redundant array of independent disks** (**RAID**) configuration. RAID is a way of storing data on multiple disks. It protects data loss from drive failure and increases disk throughput by striping various disks together.

RAID uses the technique of disk mirroring or disk striping, but for the operating system, RAID is a single logical disk. RAID has different levels to distinguish the configuration type. For example, RAID 0 uses disk striping and provides the best performance but has no fault tolerance, whereas RAID 1 is **disk mirroring**. It duplicates the data storage and provides no performance improvement for write operations but doubles the read performance. You can combine RAID 0 and RAID 1 to form RAID 10, also known as RAID 1+0. It helps to achieve the best of both with high throughput and fault tolerance. It requires a minimum of four disks and stripes data across mirrored pairs.

Select the storage solution that matches your access pattern to maximize performance. There are various options available with a cloud offering to choose from for your block, file, and object storage method. For example, the public cloud AWS provides **Amazon Elastic Block Store** (**EBS**) as a SAN type of storage in the cloud and **Amazon Elastic File System** (**EFS**) as a NAS type of storage in the cloud. Amazon S3 is very popular for object storage. Similarly, Microsoft Azure provides Azure Disk Storage for SAN, Azure Files for NAS, and Azure Blob for block storage. Different storage solutions allow you to choose your storage methods based on the access pattern, whether working in an on-premise environment or going cloud native.

Now that you have learned about the compute and storage choices necessary to achieve optimal performance, let's look at the next critical component of application development: the database. Choosing the right database for the right need will help you maximize your application performance and lower overall application latency. There are different types of databases available, and choosing the correct database is critical.

**Choosing the database**

Often, you will want to standardize a common platform and use a database for ease of management; however, you should consider using a different database solution as per your data requirements. Selecting the incorrect database solution can impact system latency and performance. The choice of a database can vary based on your application's requirements for availability, scalability, data structure, throughput, and durability. There are multiple factors to consider when choosing to use a database. For example, the access pattern can significantly impact the selection of database technology. It would be best if you optimized your database based on the access pattern.

Databases generally have a configuration option for workload optimization. You should consider the configuration for memory, cache, storage optimization, and so on. You should also explore the operational aspect of database technologies regarding scalability, backup, recovery, and maintenance. Let's look at the different database technologies that can be used to fulfill the database requirements of applications.

**Online transactional processing**

Most of the traditional relational databases are considered to use **online transactional processing** (**OLTP**). The transactional database is the oldest and most popular method of storing and handling application data. Some examples of relational OLTP databases are Oracle, Microsoft SQL Server, MySQL, PostgreSQL, Amazon RDS, and others. The data access pattern for OLTP involves fetching a small dataset by looking up its ID. A database transaction means that either all related database updates were completed or none of them were.

The relational model allows processing complex business transactions in an application, such as banking, trading, and e-commerce. It will enable you to aggregate data and create complex queries using multiple joins across tables. While optimizing your relational database, you need to consider including the following optimizations:

* A database server that includes computing, memory, storage, and networking
* Operating system-level settings, such as a RAID configuration of the storage volume, volume management, and block size
* Database engine configuration and partition as required
* Database-related options, such as schema, index, and view

Scaling can be tricky for the relational database as it can scale vertically and hit the upper limit of system capacity. For horizontal scaling, you have to read the replica for *read scaling* and partition for *write scaling*. In the previous chapter, you learned how to scale a relational database in the section titled *Database handling in the application architecture*.

OLTP databases are suitable for large and complex transactional applications; however, they don't scale well where a massive amount of data needs to aggregate and be queried. Also, with the internet boom, there is a lot of unstructured data coming from everywhere, and relational databases cannot handle unstructured data efficiently out of the box. In this case, the NoSQL database comes to the rescue. Let's learn more about how to handle a nonrelational database.

**Nonrelational databases**

There is a lot of unstructured and semistructured data produced by applications such as social media programs, the **Internet of Things** (**IoT**), clickstream data, and logs, where you have a very dynamic schema. These data types may have different schemas for each set of records. Storing this data in a relational database could be a very tedious task. Everything has to be filed in a fixed schema, which can either cause lots of null values or data loss. The nonrelational or **NoSQL** database provides you with the flexibility to store such data without worrying about a fixed schema. Each record can have a variable number of columns and can be stored in the same table.

NoSQL databases can store a large amount of data and provide *low-access latency*. They are easy to scale by adding more nodes when required and can support horizontal scaling out of the box. They can be an excellent choice to store user session data and make your application stateless to achieve horizontal scaling without compromising user experience. You can develop a distributed application on top of the NoSQL database, which provides good latency and scaling, but query joining must be handled at the application layer. NoSQL databases don't support complex queries such as joining tables and entities.

There are various choices available for the NoSQL database, for example, Cassandra, HBase, and MongoDB, which you can install in a cluster of virtual machines; however, the public cloud-like AWS provides a managed NoSQL database called **Amazon DynamoDB**,which offers high throughput sub-millisecond latency with unlimited scaling.

You can use OLTP for a relational database, but it has limited storage capacity. It doesn't respond well to queries for large amounts of data and performs aggregations as required for data warehouses. Data warehousing needs are more analytical than transactional. The **online analytical processing** (**OLAP**) database satisfies the gap of the OLTP database to query a large dataset. Let's learn more about the OLAP database.

**Online analytical processing**

OLTP and NoSQL databases are helpful for application deployment but have limited capabilities for large-scale analysis. A query for a large volume of structured data for analytics purposes is better served by a data warehouse platform designed for faster access to structured data. Modern data warehouse technologies adopt the columnar format and use MPP, which helps to fetch and analyze data faster.

The columnar format avoids the need to scan the entire table when you need to aggregate only one column for data—for example, if you want to determine your inventory sales in a given month. There may be hundreds of columns in the order table, but you need to aggregate data from the purchase column only. With a columnar format, you will only scan the purchase column, which reduces the amount of data scanned compared to the row format and increases the query performance.

With MPP, you store data in a distributed manner between child nodes and submit a query to the leader nodes. Based on your partition key, the leader node will distribute queries to the child nodes, where each node picks up part of a query to perform parallel processing. The leader node then collects the subquery result from each child node and returns your aggregated result. This parallel processing helps you to execute the query faster and process a large amount of data quicker.

You can use this kind of processing by installing software such as IBM Netezza or Microsoft SQL Server on a virtual machine, or you can go for a more cloud-native solution, such as Snowflake. A public cloud, such as AWS, provides the petabyte-scale data warehousing solution Amazon Redshift, which uses the columnar format and MPP. You will learn more about data processing and analytics in *Chapter 13*, *Data Engineering for Solution Architecture*.

You need to store and search a large amount of data, especially when you want to find a specific error in your logs or build a document search engine. For this kind of capability, your application needs to create a data search functionality. Let's learn more about data search.

**Building a data search functionality**

Often, you will need to search a large volume of data to solve issues quickly or get business insights. Searching your application data will help you access detailed information and analyze it from different views. To search for data with low latency and high throughput, you need to have search engines as your technology choice.

Elasticsearch is one of the most popular search engine platforms and is built on top of the **Apache Lucene** library. Apache Lucene is a free and open-source software library that is the foundation of many popular search engines. The **ELK** (short for **Elasticsearch**, **Logstash**, **and Kibana**) Stack is easy to use to discover large-scale data and index it for searching automatically. Because of its properties, multiple tools have been developed around Elasticsearch for visualization and analysis. For example, **Logstash** works with Elasticsearch to collect, transform, and analyze a large amount of an application's log data. **Kibana** has an in-built connector with Elasticsearch that provides a simple solution for creating dashboards and analyzing indexed data.

Elasticsearch can be deployed in virtual machines and scale horizontally to increase capacity by adding new nodes to the cluster. The public cloud AWS provides the managed service **Amazon OpenSearch Service**, making it cost-effective and simple to scale and manage the Elasticsearch cluster in the cloud.

In this section, you learned about the various database technologies and where they are used. Your applications can use a combination of all database technologies for their different components to achieve optimal performance. For complex transactions, you need to use a relational OLTP database, and to store and process unstructured or semistructured data, you need to use a nonrelational NoSQL database. You should use a NoSQL database where very low latency is required over multiple geographical regions and where you need to handle complex queries at the application layer, such as in a gaming application. If you need to perform any large-scale analytics on structured data, use a data warehouse OLAP database. You can use a cache database to improve the performance efficiency of a database. You learned about Redis and Memcached in *Chapter 6*, *Solution Architecture Design Patterns*, in the section *Memcached versus Redis.*

Let's look at another critical component of your architecture, which is **networking**. Networking is the backbone of the entire application and establishes communication between the servers and the outside world. Let's learn about networking as regards application performance.

**Improving network performance**

In this era of fast internet availability in almost every corner of the world, it is expected that applications will have a global user reach. Any delay in the system's response time depends upon the request load and the distance of the end-user from the server. If the system is not able to respond to user requests promptly, it can have a ripple effect by continuing to engage all system resources and pile up a considerable request backlog, which will degrade the overall system performance.

To reduce latency, you should simulate the user's location and environment to identify any gaps. As per your findings, you should design the server's physical location and caching mechanism to reduce network latency; however, the network solution choice for an application depends upon the networking speed, throughput, and network latency requirements. An application to handle a global user base needs to have fast connectivity with its customers, and location plays an important role. Edge locations provided by the CDN help to localize the rich content and reduce overall latency.

In *Chapter 6*, *Solution Architecture Design Patterns*, you learned how to use a CDN to put data near your user's location in the section titled *Cache-based architecture*. There are various CDN solutions available with an extensive network of edge locations. You can use a CDN if your application is static-content-heavy, where you need to deliver large image and video content to your end-user. Some of the more popular CDN solutions are Akamai, Cloudflare, and Amazon CloudFront (provided by the AWS cloud). Let's look at some DNS routing strategies to achieve low latency if your application is deployed globally.

**Defining a DNS routing strategy**

To have global reach, you may be deploying your application in multiple geographical regions. When it comes to user request routing, you want to route their requests to the nearest and fastest available server for a quick response from your application. The DNS router provides the mapping between the domain names and the IP addresses. It ensures that the requests are served by the correct server when the user types in the domain name—for example, when you type amazon.com in your browser to do some shopping, your request is always routed to the Amazon application server DNS service.

The public cloud-like AWS provides a DNS service called **Amazon Route 53**, where you can define a different kind of routing policy as per your application's needs. Amazon Route 53 provides DNS services to simplify domain management and zone APEX support. The following are the most used routing policies:

* **Simple routing policy**: As the name suggests, this is the most straightforward routing policy and doesn't involve any complications. It is helpful to route traffic to a single resource—for example, a web server that serves content for a particular website.
* **Failover routing policy**: This routing policy requires you to achieve high availability by configuring active-passive failover. If your application goes down in one region, then all the traffic can be routed to another region automatically.
* **Geolocation routing policy**: If the user belongs to a particular location, you can use a geolocation policy. A geolocation routing policy helps to route traffic to a specific region.
* **Geoproximity routing policy**: This is like a geolocation policy, but you have the option to shift traffic to other nearby locations when needed.
* **Latency routing policy**: If your application runs in multiple regions, you can use a latency policy to serve traffic from the region where the lowest latency can be achieved.
* **Weighted routing policy**: A weighted routing policy is used for A/B testing, where you want to send a certain amount of traffic to one region and increase this traffic as your trial proves more and more successful.

Additionally, Amazon Route 53 can detect anomalies in the source and volume of DNS queries and prioritize requests from users that are known to be *reliable*. It also protects your application from a DDoS attack. Once traffic passes through the DNS server, in most cases, the next stop will be a load balancer, which will distribute traffic among a cluster of servers. Let's learn some more details regarding the load balancer.

**Implementing a load balancer**

The load balancer distributes network traffic across the servers to improve concurrency, reliability, and application latency. The load balancer can be *physical* or *virtual*. It would be best if you chose a load balancer based on your application's needs. Commonly, two types of load balancer can be utilized by the application:

* **Layer 4 or network load balancer**: Layer 4 load balancing routes packets based on information in the packet header—for example, source/destination IP addresses and ports. Layer 4 load balancing does not inspect the contents of a packet, which makes it less compute-intensive and therefore faster. A network load balancer can handle millions of requests per second.
* **Layer 7 or application load balancer**: Layer 7 load balancing inspects and routes packets based on the full contents of the packet. Layer 7 is used in conjunction with HTTP requests. The materials that inform routing decisions are factors such as HTTP headers, URI path, and content type. It allows for more robust routing rules but requires more compute time to route packets. The application load balancer can route the request to containers in your cluster based on their distinctive port number.

Depending on the environment, you can choose hardware-based load balancers, such as an F5 load balancer or a Cisco load balancer. You can also select a software-based load balancer, such as **Nginx**.

The public cloud provider AWS facilitates a managed virtual load balancer called **Amazon** **Elastic Load Balancing** (**ELB**). ELB can be applied at layer 7 as an application load balancer and layer 4 as a network load balancer.

A load balancer is an excellent way of securing your application, making it highly available by sending a request to healthy instances. It works together with auto-scaling to add or remove instances as required. Let's look at auto-scaling and learn how it helps to improve overall performance and the high availability of your application.

**Applying auto-scaling**

You learned about auto-scaling in *Chapter 4*, *Principles of Solution Architecture Design*. You learned about predictive auto-scaling and reactive auto-scaling in the section titled *Design for scale*. The concept of auto-scaling became popular with the agility provided by the cloud computing platform. Cloud infrastructure allows you to quickly scale up or scale down your server fleet based on user or resource demand.

With a public cloud platform such as AWS, you can apply auto-scaling at every layer of your architecture. You can scale the web server fleet based on your requests in the presentation layer and at the application layer based on the server's memory and CPU utilization. You can also perform scheduled scaling if you know the traffic pattern when the server load increases. At the database level, auto-scaling is available for relational databases such as Amazon Aurora Serverless and Microsoft Azure SQL database. A NoSQL Database such as Amazon DynamoDB can be auto-scaled based on throughput capacity.

When auto-scaling, you need to define the number of desired server instances. You need to determine the maximum and minimum server capacity as per your application's scaling needs. The following screenshot illustrates the auto-scaling configuration from the AWS cloud:

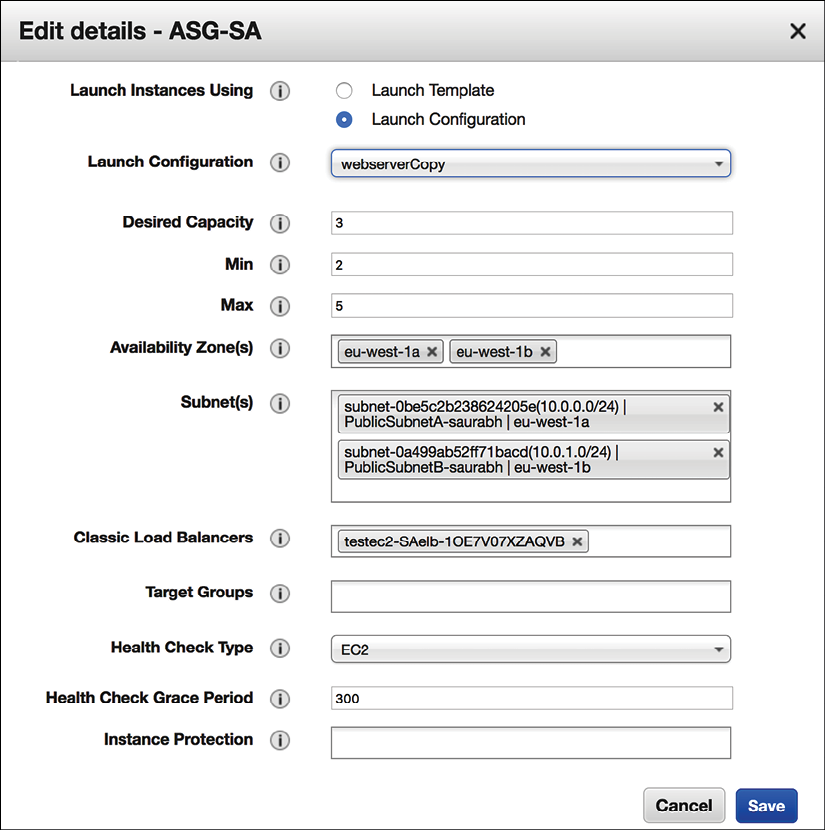


Figure 7.7: Auto-scaling configuration

In the preceding auto-scaling configuration setting, if three web server instances are running, it can scale up to 5 instances if the CPU utilization of servers goes above 50% and scale down to 2 instances if the CPU utilization goes below 20%. In an unhealthy instance, the count will go below the desired capacity in a standard scenario. In such a case, the load balancer will monitor the instance health and use auto-scaling to provide new instances. The load balancer monitors instance health and will trigger auto-scaling to provision new instances as required.

Auto-scaling is a good feature to have, but make sure you set up your desired configurations to limit the cost of a change in CPU usage. In the case of unforeseen traffic due to a **distributed denial of service** (**DDoS**) attack, auto-scaling can significantly increase costs. It would help if you planned to protect your system from such kinds of events. You will learn more about this in *Chapter 8*, *Security Considerations*.

You need **high-performance computing** (**HPC**) to perform manufacturing simulation or human DNA analysis at the instance level. HPC performs well when you put all instances in the same network close to each other for low latency of data transfer between a cluster node. Between your data centers or the cloud, you can choose to use your private network, which can provide an additional performance benefit. For example, to connect your data center to the AWS cloud, you can use Amazon Direct Connect. Direct Connect provides 10 Gbps private fiber-optic lines, where network latency is much lower than sending data over the internet.

In this section, you have learned about various networking components that can help to improve application performance. You can optimize your application network traffic according to your user location and application demand. Performance monitoring is an essential part of your application, and you should do proactive monitoring to improve the customer experience. Let's learn more about performance monitoring.

**Managing performance monitoring**

Performance monitoring is essential when you are trying to understand any performance issue and reduce end-user impact proactively. You should define your performance baseline and raise the alarm to the team in the case of a threshold breach—for example, an application's mobile app load time should not be more than three seconds. Your alarm should be able to trigger an automated action to handle poorly performing components—for example, adding more nodes in a web application cluster to reduce the request load.

There are multiple monitoring tools available to measure application performance and overall infrastructure.

You can use a third-party tool, such as Splunk or the AWS-provided Amazon CloudWatch, to monitor any application. Monitoring solutions can be categorized into **active monitoring** and **passive monitoring** solutions:

* With active monitoring, you need to simulate user activity and identify any performance gap upfront. Application data and workload situations are constantly changing, which requires continuous proactive monitoring. Active monitoring works alongside passive monitoring as you run the known possible scenarios to replicate the user experience. You should run active monitoring across all dev, test, and prod environments to catch any issue before reaching the user.
* Passive monitoring tries to identify an unknown pattern in real time. For a web-based application, passive monitoring needs to collect essential metrics from the browser that can cause performance issues. You can gather metrics from users regarding their geolocation, browser types, and device types to understand the user experience and the geographic performance of your application. Monitoring is all about data, and it includes the ingestion, processing, and visualization of lots of data.

Performance always comes with a cost, and, as a solution architect, you need to think about the trade-offs to take the right approach. For example, an organization's internal applications, such as the timesheet and HR programs, may not need as high performance as external products, such as e-commerce applications. An application that deals with trading (for example) needs very high performance, which requires more investment. As per your application's needs, you can balance durability, consistency, cost, and performance. You will continue to learn about the various monitoring methods and tools in upcoming chapters and dive deep into monitoring and alerts in *Chapter 9*, *Architectural Reliability Considerations*.

Tracking and improving performance are complex tasks where you need to collect lots of data and analyze patterns. An access pattern helps you to make the right choice for performance optimization. Load testing is one method that allows you to tweak your application configuration by simulating user load and provides you with data to make the right decisions for your application architecture. Applying continuous active monitoring in combination with passive monitoring helps you to maintain consistent performance for your application.

**Summary**

In this chapter, you learned about the various architecture design principles that impact the performance of applications. You learned about latency and throughput at different layers of architecture and how they relate to each other.

For highly performant applications, you need to have low latency and high throughput at every architecture layer. Concurrency helps to process a large number of requests. You also learned the difference between parallelism and concurrency and gained insight into how caching can help to improve overall application performance.

Then you learned about choosing your technology and their working models, which can help achieve your desired application performance. While looking at the compute option, you learned about the various processor types and their differences to help you make the right choice when selecting server instances. You learned about containers and how they can help you to utilize the resources efficiently and at the same time help to improve performance. You also learned how Docker and Kubernetes work well with each other and fit into your architecture.

In the section on choosing storage, you learned about different kinds of storage, such as block, file, and object storage, and their differences. You also learned about the available storage choices in on-premise and cloud environments. Storage choice depends on multiple factors. You can enhance disk storage durability and throughput by putting multiple volumes in a RAID configuration.

In the section on choosing a database, you learned about the various database types, including relational, nonrelational, data warehouse, and data search. While looking at choosing your network, you learned about the different request routing strategies that can help you to improve network latency for your globally distributed user. You learned how load balancers and auto-scaling could help you manage many user requests without compromising application performance.

In the next chapter, you will learn how to secure your application by applying authentication and authorization. It will ensure that your data at rest and in transit and your application are protected from various kinds of threats and attacks. You will also learn about compliance requirements and how to satisfy them when designing your application. You will learn the details about security audits, alerts, monitoring, and automation.

**Join our book's Discord space**

Join the book's Discord workspace to ask questions and interact with the authors and other solutions architecture professionals: <https://packt.link/SAHandbook>