**Scalability and Load Balancing**  
Scalability refers to a system’s ability to handle increased load by adding resources. Scalability ensures that the system can grow and continue functioning efficiently.

Types of scalability

* Vertical scalability - This involves adding more resources (CPU, RAM, etc.) to a single machine to handle increased load.
* Horizontal scalability - This involves adding more machines (nodes/servers) to distribute the load.

Load balancing refers to distributing incoming traffic across multiple servers so no single server is overwhelmed. It plays a key role in horizontal scalability by ensuring that your system can handle more users while maintaining performance and avoiding downtime

Types of load balancers

* Layer 4 load balancers - Operates at the transport layer, directing traffic based on IP addresses and TCP/UDP ports.
* Layer 7 load balancers- Operates at the application layer, directing traffic based on HTTP headers, URLs, or even cookies.

A load balancing algorithm is the logic that a load balancer uses to distribute network traffic between servers (an algorithm is a set of predefined rules).

Two primary approaches to load balancing.

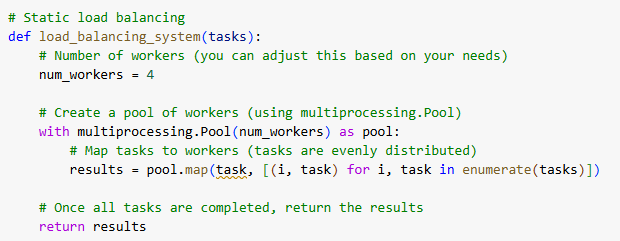
* Static load balancing distributes traffic without making these adjustments. Some static algorithms send an equal amount of traffic to each server in a group, either in a specified order or at random.
* Dynamic load balancing uses algorithms that take into account the current state of each server and distribute traffic accordingly.

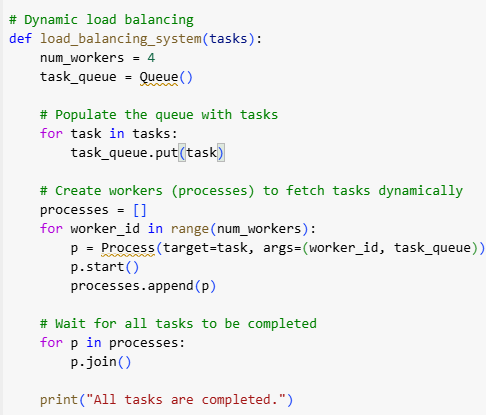
**Static load balancing algorithms**

* Round robin - distributes traffic to a list of servers in rotation using the Domain Name System (DNS). An authoritative nameserver will have a list of different records for a domain and provides a different one in response to each DNS query.
* Weighted round robin - allows an administrator to assign different weights to each server. Servers deemed able to handle more traffic will receive slightly more.
* IP hash - Combines incoming traffic's source and destination IP address and uses a mathematical function to convert it into a hash. Based on the hash, the connection is assigned to a specific server.

**Dynamic load balancing algorithms**

* Least connection- Checks which servers have the fewest connections open at the time and sends traffic to those servers.
* Weighted least connection- Gives administrators the ability to assign different weights to each server, assuming that some servers can handle more connections than others.





**Advanced Distributed Architectures**

**Client-Server Architecture**

Client-server model is one of the oldest and most widely used distributed architectures

* Clients request services
* Servers respond to these requests by providing the requested resources
* Communication is usually over a network like the internet

Examples: Web applications, email systems

**Strengths:** Centralized control and management, easier to secure and maintain

**Limitations:** Single point of failure, may not scale well without load balancers

**Peer-to-Peer Architecture**

In a Peer-to-Peer model, all nodes are equal

* Each node can act as both a client and a server
* Resources are shared directly between peers

Example: File-sharing networks

**Strengths:** High scalability, No central server bottlenecks

**Limitations:** Harder to manage and secure, potential for inconsistent performance

**Cloud-Based Architecture**

Cloud-based systems distribute computation and storage across a network of data centers

* Typically built on virtualization and containerization (e.g., using AWS, Azure)
* Can include microservices, serverless functions, and distributed databases

Example: Scalable web apps, big data processing

**Strengths:** Scalability and flexibility, pay-as-you-go pricing

**Limitations:** Dependence on cloud provider, latency and data privacy concerns

**Fault Tolerance and Reliability in Distributed Systems**

**Fault Tolerance**

Fault Tolerance is the ability of a distributed system to continue operating correctly even when some components fail. It’s a key requirement for building reliable and robust systems

Key fault tolerance techniques

* Replication – involves creating multiple copies of data or services across different nodes. If one node fails, others can take over. Challenge: maintaining consistency among replicas
* Recovery – ensures a system resumes normal operation after a fault. Techniques include restarting failed components, rolling back to previous state
* Redundancy – adding extra hardware/software resources that can take over in case of failure

**Checkpointing**

Checkpointing is a specific fault-tolerant method where a system periodically saves its state. If a crash occurs, the system can roll back to the last checkpoint rather than starting over

A diagram of a diagram

AI-generated content may be incorrect.

**Cloud and Edge Computing in Distributed Environments**

**Cloud Computing vs Edge Computing**

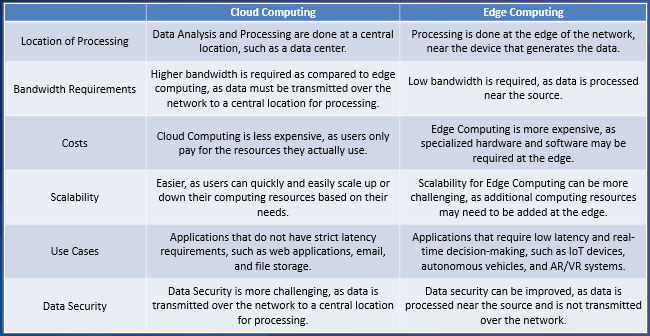
Cloud Computing is a model for delivering information technology services over the internet

* On-demand scalability and flexibility
* Major providers: AWS, Google Cloud, Azure
* Example: running virtual machines in AWS EC2

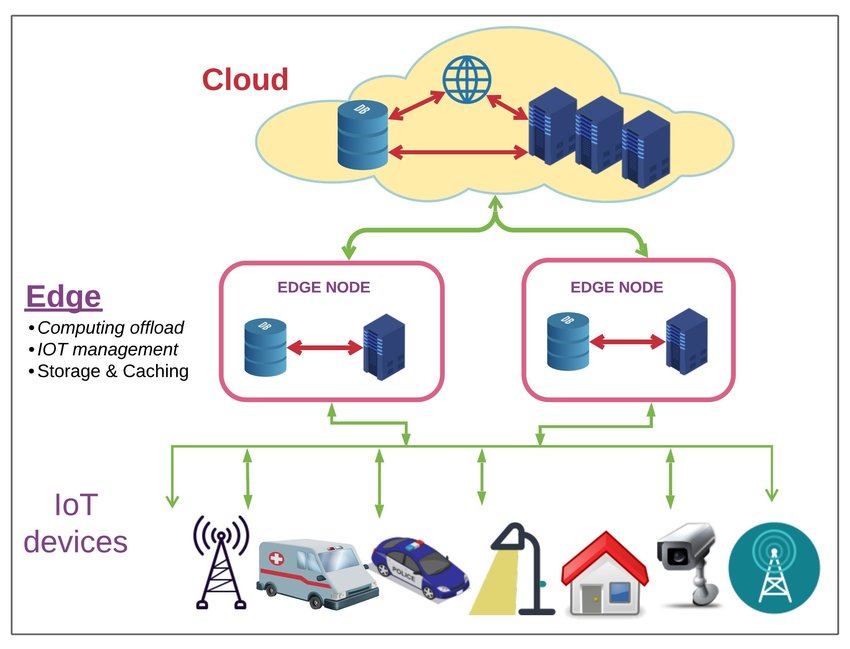
Edge Computing is a distributed computing architecture that brings computing and data storage closer to the source of data. Data processing takes place at the network’s edge, adjacent to the device that generated the data

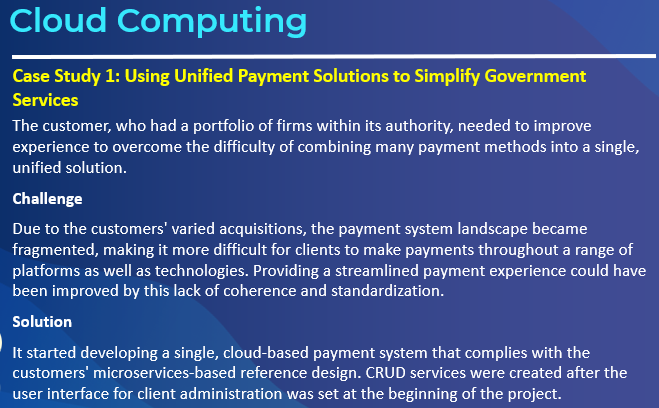
* Reduces latency, saves bandwidth, and enhances real-time performance
* Example: smart cameras processing video at the edge

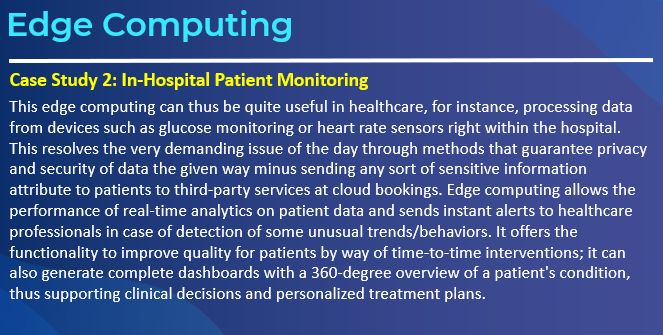
**Cloud Computing vs Edge Computing**



**Cloud Computing vs Edge Computing**







**GPU Programming and Hardware Acceleration**

**GPU Programming**

GPU Programming is a method of running highly parallel general-purpose computations on GPU accelerators

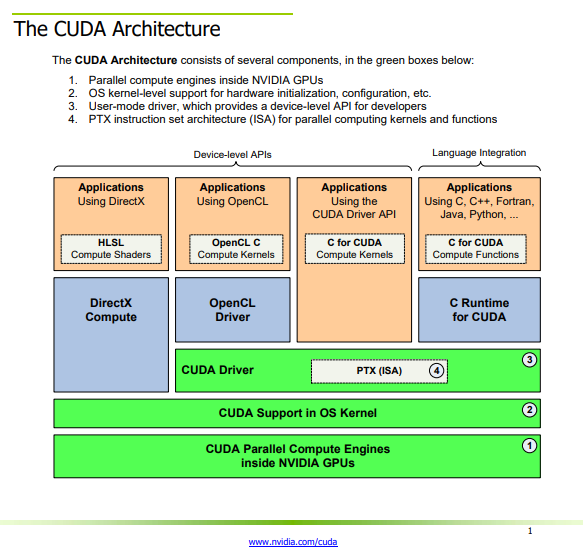
While the past GPUs were designed exclusively for computer graphics, today they are being used extensively for general-purpose computing

CPU vs GPU Computing Differences

* A Central Processing Unit (CPU) is a latency-optimized general-purpose processor that is designed to handle a wide range of distinct tasks sequentially, while a Graphics Processing Unit (GPU) is a throughput-optimized specialized processor designed for high-end parallel computing.

**GPU Programming APIs**

* CUDA - Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming interface (API) created by Nvidia in 2006, that gives direct access to the GPU’s virtual instruction set for the execution of compute kernels.
* OpenCL - OpenCL is an open standard for parallel programming across heterogeneous platforms created by the Khronos Group. OpenCL works with central processing units (CPU), graphics processing units (GPU), digital signal processors, field-programmable gate arrays (FPGA) and other processors or hardware accelerators.



**CUDA Programming Model**

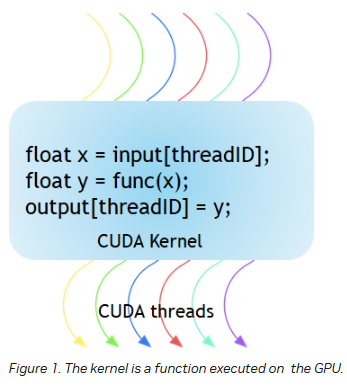
The CUDA programming model provides an abstraction of GPU architecture that acts as a bridge between an application and its possible implementation on GPU hardware.

The host is the CPU available in the system. The system memory associated with the CPU is called host memory. The GPU is called a device and GPU memory likewise called device memory.

To execute any CUDA program, there are three main steps:

* Copy the input data from host memory to device memory, also known as host-to-device transfer.
* Load the GPU program and execute, caching data on-chip for performance.
* Copy the results from device memory to host memory, also called device-to-host transfer.

Figure 1 shows that the CUDA kernel is a function that gets executed on GPU. The parallel portion of your applications is executed K times in parallel by K different CUDA threads



* A group of threads is called a CUDA block. CUDA blocks are grouped into a grid.
* A kernel is executed as a grid of blocks of threads (Figure 2)
* Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism).
* One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks.
* Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time. Figure 3 shows the kernel execution and mapping on hardware resources available in GPU.

**CUDA in Python**

* PyCUDA is a Python interface for CUDA that provides access to the CUDA API from Python. With PyCUDA, you can write CUDA programs in Python, which can be more convenient and easier to read than traditional CUDA C programs.

