UNIVERSITÀ DEGLI STUDI DI TRIESTE

Scalability

Machine Learning Operations

Roberta Lamberti

Università degli Studi di Trieste - a.a. 2025-2026

30 October 2025

Whoami

Roberta Lamberti

- BSc in Statistics at University of Salerno
- MSc in Scientific & Data-Intensive Computing, curriculum Data Engineering and High Performance Computing at University of Trieste
- Master in Data Management and Curation at SISSA & Area Science Park
- Intern at LADE (Laboratory of Data Engineering) at Area Science Park

Contact: roberta.lamberti@studenti.units.it

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Agenda

1. Scalability
2. Scalability measures
3. Scalability in MLOps context
4. Vertical Scalability
5. Horizontal Scalability

Scalability

What is scalability?

Scalability can refer to many application contexts.

In a broader point of view, scalability is the capacity of a certain process, system or organization to handle a growing amount of workload, yet maintaining operational efficiency.

Practical example:

Suppose to be an HR specialist at a small company, that has only 5 employees. Each day, each employee needs to register the entrance and the exit into the office by signing a form in the HR specialist office.

Now, suppose that the month after the company decides to hire 10 more people, and the month after 20 more people, and so on. So, each month, the company employee size increases by a factor of n.

In this way, each day the HR specialist will spends her entire day just making sure all the n employees are going to the office and work. Is this scalable? NO!

In order to make everyone's life easier, we must implement an efficient strategy. What do we do? We distribute badges to each employee, and give them the directive to swipe the badge at specific readers located at the office entrance.

In computing and software, scalability refers to handle increasing amounts of work, users, or data without performance loss or the need for major redesign. A scalable system can grow in two main ways: horizontal and vertical scaling.

Why do we need it?

There are several reasons that can justify this question.

1. **Growth**: As user numbers or data volumes increase, scalable systems can expand to maintain speed and reliability.
2. **Cost efficiency**: Scalable solutions allow organizations to start small and add resources only when needed, avoiding unnecessary upfront costs.
3. **Performance**: Scalability ensures that applications remain responsive and stable even under heavy load.
4. **Business continuity**: It helps prevent downtime or service interruptions when demand spikes, supporting customer satisfaction and trust (we don't our application to stay down for too long or too often, otherwise the users might start using competitors applications)
5. **Future-proofing**: Scalable architectures are easier to adapt to new requirements, technologies, or markets (for example, you might need to upgrade the current framework that you have, such as TensorFlow and Pythorch, in order to make sure your application still work, as sometimes the framework maintainers upgrade their packages)

UNIVERSITÀ DEGLI STUDI DI TRIESTE

How can we state if an application is actually scaling?

We use some measures:

- Latency;
- Overhead;
- Throughput;
- Performance

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Latency

The latency is a fundamental KPI of an application.

It refers to the time needed by a system to respond to a user request.

Example:

Nowadays if we want to communicate, we just use whatsapp, telegram or e-mail, but how people were used to communicate before? For example, in the XX century, telegrams were quite used. But before? People were used to write letters.

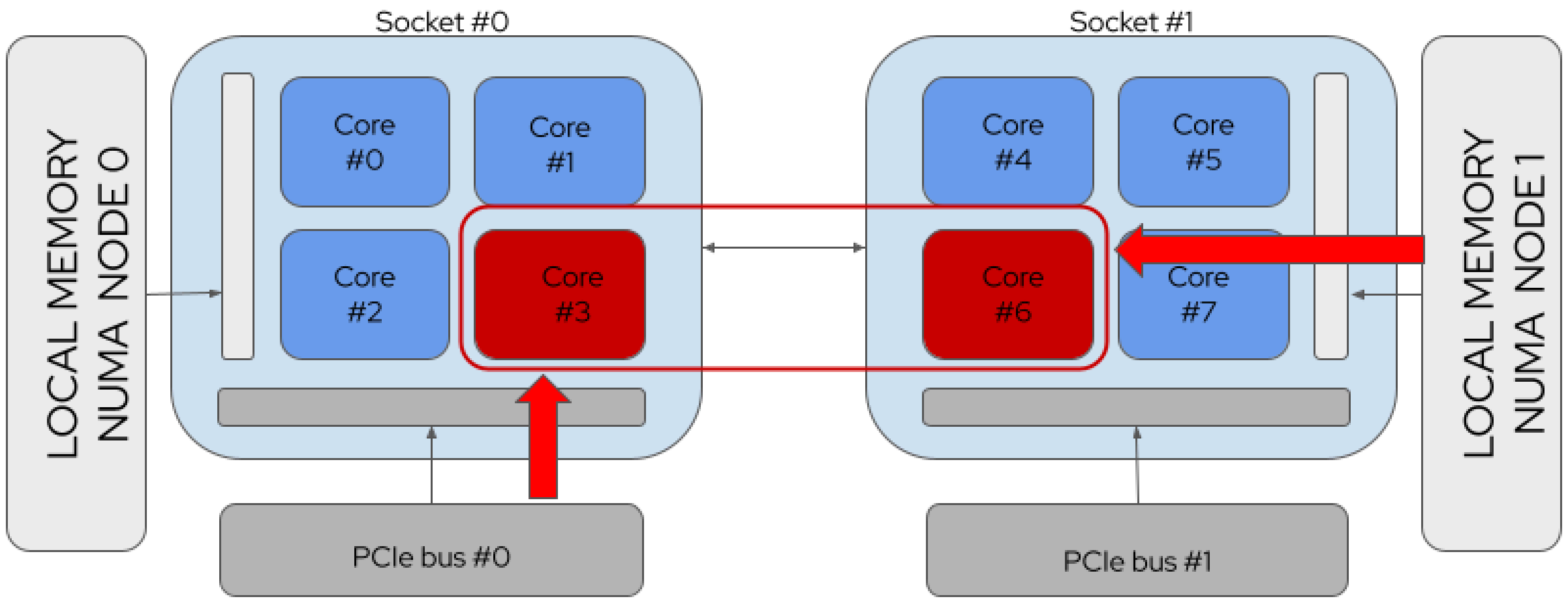
Now, suppose you want to send a letter from Venice to Naples. The latency in this context is the time spent by the postman (that was travelling on horses) to reach Naples from Venice.

And what if you want to send another letter to Trieste? Which of the two postmen is going to have lower latency? Of course the one travelling from Venice to Trieste.

This is the same logic we follow in HPC when we want to send a message from one core of a processing unit to a core of another processing unit.
Hardware context example: a message from one core to another needs time to travel from point A to point B. Further away it has to travel, higher might be the latency.

In general, we want low latency, as we want our application to be fast and not spend to much time to complete tasks.

Shared memory: CPU topology and NUMA (Non uniform memory access)



In the previous image, there is the topological structure of a CPU. The earliest computers had CPUs with a single processing core. Nowadays, only multiprocessor CPUs are used.
So, in this image we can see two physical processors (in the image they are called socket, as the socket is the slot that contains the processor of the CPU). Each processor contains cores (that is the part of the CPUs doing all the computation). These cores are directly attached to local memory (NUMA nodes).

A message going from core 0 to core 1 within socket 0 experiences low latency, since they share the same memory controller.

However, if the message needs to go from core 3 within Socket 0 to core 6 within Socket 1 the latency increases, because the data must travel across the NUMA interconnect (which is the arror in between the two sockets) between the two areas of the CPUs (the processors, that in the image are called Socket #0 and Socket #1).

This is just the same logic behind the letters example, but with more technical terms.

N.B. These concepts will be explored in further detail in HPC course with Luca Tornatore.

Latency in MLOps

Developed application context: on the other hand, when developing an application, latency is the time it takes for the application to process a user request.

- **SOLUTION**: in order to reduce application latency, stress and load tests can be run to identify bottlenecks that slow down the system.

Additional measures can also be taken to optimize code, improve hardware and software infrastructure (vertical scaling), and implement caching (horizontal scaling) and other techniques.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Overhead

Overhead refers to the extra resources (time, memory, CPU) consumed by managing parallel or distributed operations, rather than doing the actual computation.

It manifests as slower processing, less memory, less storage capacity, less bandwidth and longer latency.

Example

In simple terms, imagine you have to follow lessons everyday at the university in Trieste, but you live in the Maggiore Hospital area.

The work is represented by going to the lesson, but in order to complete this task we need to consider additional work.

The additional work consists of:

- Wearing clothes, shoes
- Get the keys before going out
- Go to the univerisity and then coming back

The overhead in this case is all the additional work we need to just to complete the task "go to lesson".

Practical example

Examples:

- Communication overhead between nodes in a distributed system. In a distributed system, multiple computer work together over a network. In order to coordinate, they need to send messages, share data and synchronize state. Every message introduces **communication overhead**, that is represented by extra time and resources (such as RAM, in terms of buffers, queues, caches etc.) spent in transmitting, waiting for a responde and handling eventual errors.
- Synchronization overhead in multi-threaded applications. When multiple threads share data or resources, they must synchronize using some tools like locks or barries to avoid race conditions.
  A race condition is a situation that occurs when multiple processes or threads access and modify the data they share. How do they share data? Based on the NUMA or UMA. These topics will be seen better in HPC course. This is just a broad overview.
  Anyway, this syncronization introduces overhead, and threads have to wait for one another, even if the application is parallel.

Goal: Minimize overhead to maximize efficiency.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Overhead in MLOps context:

By overhead, we refer to the costs and complexities of managing the machine learning lifecycle at scale, such as infrastructure, maintenance, and manual tasks. While MLOps aims to reduce this overhead through automation, continuous integration/continuous deployment (CI/CD), and cloud-native solutions, the main challenges of this areas are:

- infrastructure costs
- achieving fast iteration speeds (it represents the overhead linked to the dead times of inefficiences in the development cycle. For example it might occur we have to wait hours and hours for training our model, because we can't parallelize the application)
- integrating disparate tools (in an ML cycle we have different stages, like data preparation, training, serving and monitoring, and all these stages use different tools, that not always are compatible)

Throughput

Throughput measures the amount of work completed per unit of time. It's often expressed as requests per second, transactions per minute, or data processed per hour.

High throughput = system can handle many operations concurrently.

In a broader view, it can be represented as it follows: throughput = Amount of work / Time
In MLOps: predictions per second, training samples per batch (it's a group of jobs or collection of data processed by a computer in an automated way), API requests per second.

It's the concept on which a computing paradigm is based: it's called *High Throughput Computing (HTC)*.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Performance

Performance is a composite metric that includes latency, throughput, and resource utilization. It measures the rate at which a system can process a large number of tasks over a given period.

In throughput we want to:

- complete many tasks in parallel, which is suitable for problems that require repeated calculations over a long time of period;

A high-performance system:

- Responds quickly (low latency)
- Handles many requests in (high throughput)
- Uses resources efficiently (low overhead)

Performance trade-off

The performance trade-off involves balancing different goals:

- model accuracy versus computational cost
- speed versus explainability

For example, complex models may offer higher accuracy but require more resources and be harder to understand, while simpler models are faster but less accurate.
MLOps strategies are used to manage these trade-offs, from initial model design and feature engineering to deployment and monitoring in production environment

Scalability in MLOps Context

In Machine Learning Operations, scalability is critical across three main stages:

1. **Training**: scaling model training across multiple GPUs or nodes
2. **Serving**: scaling inference to handle many prediction requests
3. **Monitoring**: scaling observability infrastructure to track models in production

Training at Scale

Challenges:

- Large datasets (terabytes or more)
- Complex models (billions of parameters)
- Long training times (days or weeks)

Solutions:

- Distributed training (data parallelism, model parallelism)
- Gradient accumulation (Gradient accumulation is a training technique used in deep learning to effectively simulate larger batch sizes without increasing memory usage. This is a common trick when your GPU can’t fit a big batch into memory.)
- Mixed precision training
- Tools: PyTorch DDP, TensorFlow Strategy, Horovod, DeepSpeed

Challenges and solutions

Large Datasets

Nowadays one of the main challenges is handling the data coming from several sources. In fact, we get into the so called ***curse of dimensionality***. It refers to the exponential growth in data volume and the computational cost, as the number of features (dimensions) increases. One of the possible approaches to solve this problem is to apply dimensionality reduction algorithms, that are part of the unsupervised learning ML approach.

Complex models (billions of parameters)

An example of a complex model is a large language model, which is a specific kind of neural network. This kind of architecture has many parameters, and it's not easy to handle all the computational power of all the interactions between the various neurons within each hidden layer.

Challenges and solutions

Long training times (days or weeks)

As we want to stay ahead within the ML world, we can't afford to spend too much time in training a model. If we do it locally, there is a risk the laptop won't be able to handle the computational power required in order to deliver results.
So, to cut the timings, we need to have access to distrubuted clusters in order to use more powerful resources such as GPUs and CPUs.

Serving at Scale

Challenges:

- High request volumes (thousands of requests per second)
- Low latency requirements (< 100ms)
- Model versioning and A/B testing

Solutions:

- Model servers: TensorFlow Serving, TorchServe, NVIDIA Triton
- Caching and batching
- Load balancing
- Auto-scaling infrastructure

Challenges and solutions

High request volumes (thousands of requests per second)

- Handling thousands of concurrent requests requires horizontally scalable systems and efficient load balancing. The load balancer has a key role, as it ensures incoming traffic is evenly distributed across model replicas (across many servers). Can be implemented with Kubernetes Services, NGINX, Envoy, or cloud-native load balancers (AWS ELB, GCP Load Balancer).

Another fundamental aspect to handle is the need to minimize I/O overhead by using asynchronous services or queue-based systems (e.g., Kafka, RabbitMQ).

The model servers like TensorFlow Serving are high performance inference servers designed to manage model lifecycles, batching, and hardware acceleration.

Another solution is using autoscaling infrastructures. This enables the system to scale up or down based on the incoming workload, preventing both overload and resource waste. In Kubernetes, usually the Horizontal Pod Autoscaler (HPA) is used in order to serve this purpose.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Challenges and solutions

Low latency requirements (< 100ms)

- Techniques include model optimization (quantization, pruning, TensorRT conversion), hardware acceleration (GPU/TPU), and intelligent caching (e.g., precomputing frequent embeddings).
- Architectural optimizations also help — reducing network calls, using lightweight serialization formats (Protobuf instead of JSON), or co-locating services to minimize latency.

Model versioning and A/B testing

- A/B testing (or canary/shadow deployments) allows you to expose only a portion of traffic to a new model and compare performance metrics before full rollout.
- It requires integration with a model registry (like MLflow or SageMaker Model Registry) and a deployment orchestrator that supports multiple model versions.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Challenges and Solutions

Monitoring at Scale

Challenges:

- Tracking performance metrics across many models (latency, throughput, resource usage)
- Detecting model drift and data drift
- Logging predictions and ground truth

Solutions:

- Centralized logging: Prometheus, Grafana, ELK stack
- ML-specific monitoring: MLflow, Weights & Biases, Neptune.ai
- Alerting and anomaly detection

Challenges and solutions

Centralized logging and metrics platforms

Tools like Prometheus + Grafana can collect system and application metrics in one place.

The perks are High scalability, integration with alert and customizable dashboards.

The use case is to monitor latency, throughput, GPU/CPU usage, and basic model metrics in real-time.

ML-specific monitoring platforms

Platforms like MLflow, Weights & Biases, Neptune.ai provide model-focused metrics tracking, experiment logging, and versioning.

They can track:

- Training/validation metrics
- Deployment metrics
- Feature distributions to detect data drift

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Challenges and solutions

Alerting and anomaly detection

Alerts notify engineers when metrics deviate from expected ranges (e.g., sudden drop in accuracy, high latency, or drift detected).

Anomaly detection can be statistical (control charts) or ML-based (autoencoders, monitoring models).

Integrating alerts with Slack, email, telegram bots, PagerDuty ensures rapid response to production issues.

Distributed Cluster

A **distributed cluster** is a group of interconnected computers (nodes) working together as a single system.

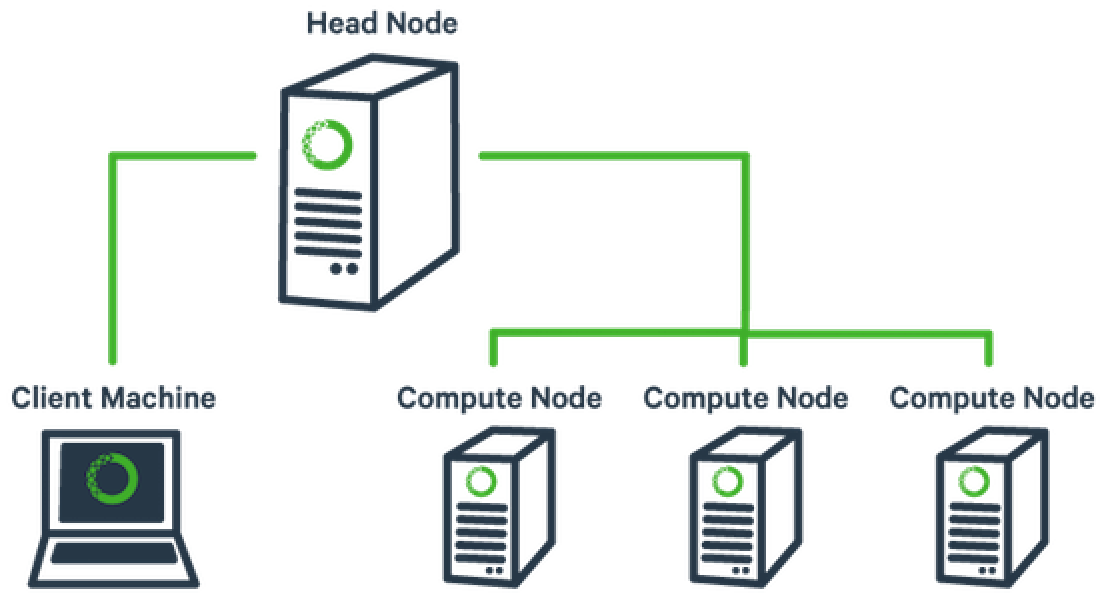
Key components:

- **Compute nodes**: performs computations
- **Storage nodes**: usually some server nodes are reserved to store data. These data will be organized through distributed file systems (e.g. CephFS)
- **Network**: high-speed interconnect (InfiniBand, Ethernet)
- **Scheduler**: manages job allocation (SLURM - to manage the queues based on priority and other factors). In the figure in the next slide, Slurm services are usually installed in the head node, and they act as the central controller for the cluster.

Use cases: large-scale ML training, big data processing, scientific simulations.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Distributed Cluster



Why do we need to know how to use a cluster?

As said in the previous slides, whenever we need to train a model, the computational cost would be too high for our personal machine.
So, what needs to be done in order to get some results from our model is to use distributed system resources (such as GPUs and CPUs).

More details about the usage of a cluster will be exposed in detail in next semester course High Performance Computing. In case you are still interested to have a glimpse and read in your freetime, you can consult the following references:

1. HPC course repository

2. ORFEO DOC

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Vertical Scalability

Vertical scaling (scaling up) = adding more resources to a single machine.

Examples:

- Adding more RAM
- Upgrading to a faster CPU
- Adding more GPUs to a server

Pro & Cons

Pros:

- Simpler to implement. Vertical scaling (scaling up) means upgrading the resources of a single machine — adding more CPU cores, RAM, or faster storage. It’s straightforward to implement because you don’t need to redesign your system architecture or manage multiple nodes. Often, you can scale up by simply replacing hardware or upgrading a virtual machine.
- No need for distributed algorithms. Since everything runs on a single machine, there’s no need for consensus protocols, distributed locks, or synchronization mechanisms. This makes the software logic simpler and reduces the risk of inconsistencies that can occur in distributed systems.
- Lower communication overhead. All processes share the same memory and bus, so communication between components is extremely fast. There’s no network latency, no inter-node data transfer, and no need for load balancing. This makes vertical scaling ideal for workloads that require tight coupling or high-speed data access.

Pro & Cons

Cons:

- Physical limits. You can only scale a single machine so far. At some point, you hit hardware boundaries — there’s a maximum amount of RAM you can install, and CPU cores can’t grow indefinitely. Beyond that, vertical scaling is no longer possible.
- Single point of failure. Since the entire system runs on one machine, if that machine fails, everything goes down. There’s no redundancy or fault tolerance unless you implement backup or replication strategies separately.
- Can be **very expensive**. High-end hardware can be prohibitively costly. For example, DGX GPU, is for 400k. While these machines deliver outstanding performance, the cost-to-benefit ratio often becomes unsustainable compared to scaling out with multiple cheaper nodes.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Horizontal Scalability

Horizontal scaling (scaling out) = adding more machines to the system.

Examples:

- Adding more web servers behind a load balancer
- Adding more worker nodes to a Kubernetes cluster
- Distributing data across multiple database shards

Pro & Cons

Pros:

- Nearly unlimited scalability
- Fault tolerance (redundancy). Fault tolerance (redundancy). Suppose you have deployed several applications inside containers. You can think of containers as isolated boxes that communicate with each other through APIs. If one box fails, it doesn’t necessarily affect the others, since each container runs in its own environment. This isolation improves reliability and prevents a single failure from taking down the entire system.
- Cost-effective (use commodity hardware). Instead of relying on one powerful and expensive machine, you can scale out using multiple inexpensive nodes. However, this approach isn’t endlessly efficient — older or cheaper nodes may eventually struggle to keep up with modern frameworks and workloads. So, not feasible in the long run

Pro & Cons

Cons:

- More complex architecture. Horizontal scaling introduces architectural complexity. Managing multiple servers or nodes requires careful coordination, monitoring, and deployment strategies. You also need to handle service discovery, health checks, and version consistency across the cluster. This adds overhead compared to a single-node system.
- Requires distributed algorithms. When data and computation are spread across several nodes, you can’t rely on simple sequential logic anymore. You need distributed algorithms for synchronization, consensus (e.g., Raft, Paxos), and data consistency. Designing and debugging these systems can be challenging, as failures and communication delays are common in distributed environments.
- Network communication overhead. Scaling out increases the number of network hops between services and nodes. Each request or data transfer introduces latency and potential bottlenecks. Even small network delays can accumulate, especially in high-throughput systems. As a result, network optimization and load balancing become critical to maintain performance.

Vertical vs Horizontal scaling

Aspect	Vertical Scaling (Scale Up)	Horizontal Scaling (Scale Out)
Approach	Add more power (CPU, RAM, GPU) to a single machine	Add more machines or nodes to the system
Complexity	Simple to implement	Requires distributed architecture
Performance	Very fast internal communication, low latency	May introduce network latency
Scalability Limit	Limited by hardware capacity	Nearly unlimited (add more nodes)
Fault Tolerance	Single point of failure	High redundancy and resilience
Cost	Expensive (high-end servers, e.g., DGX ≈ $400K)	Cheaper nodes, but higher operational complexity
Best for	Small to medium workloads, fast prototyping	Large-scale, high-traffic, production systems

Summary

Vertical scaling is great for simplicity and low-latency communication but limited by hardware constraints, cost, and lack of redundancy.

Horizontal scaling provides flexibility and fault tolerance, but it requires more sophisticated architecture and introduces new challenges in coordination, consistency, and communication efficiency.

Benchmarking

A **benchmark** is a standardized test used to measure and compare system performance.

Why benchmark?

- Compare different hardware configurations
- Evaluate software optimizations
- Choose between alternative solutions (e.g., databases, frameworks)

Examples:

- OSU Micro-Benchmarks (MPI communication. MPI - message passing interface - is a framework for High Performance Computing applications. It is used as a communication protocol for programming parallel application. Its goals are high performance, scalability and portability)
- LINPACK (CPU performance)
- MLPerf (ML training and inference)

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Load Balancing

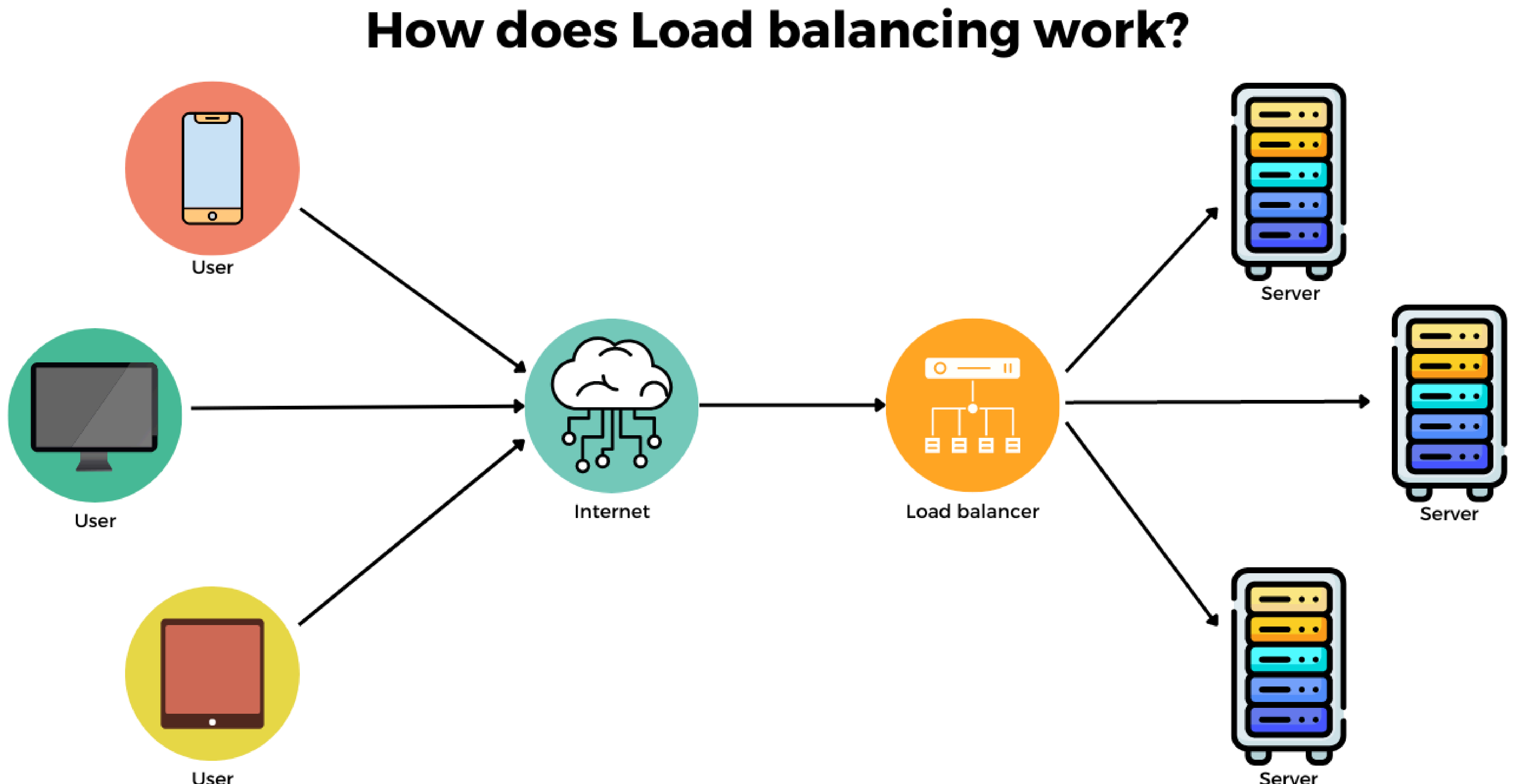
Load balancing distributes incoming requests across multiple servers to ensure no single server is overwhelmed.

Strategies:

- **Round-robin**: requests are distributed evenly in sequence
- **Least connections**: send requests to the server with fewest active connections
- **IP hash**: route requests based on client IP
- **Weighted**: assign more traffic to more powerful servers

Tools: NGINX, HAProxy, AWS ELB, Google Cloud Load Balancing

Load Balancing



Microservices Architecture

Microservices = breaking an application into small, independent services hosted in containers that communicate via APIs (even virtual machines solutions can be implemented).

Characteristics:

- Each service has a single responsibility
- Services can be scaled independently
- Polyglot: each service can use different tech stacks
- Fault isolation: failure in one service doesn't crash the entire system

Example: An e-commerce app might have separate services for user management, product catalog, order processing, and payment.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Cluster and Container Orchestration

Containers (Docker) package applications with their dependencies for consistent deployment.

Container orchestration (Kubernetes) manages deployment, scaling, and networking of containers.

Key features:

- Auto-scaling based on load
- Self-healing (restarts failed containers)
- Service discovery and load balancing
- Rolling updates and rollbacks

Note: In the next tutorial, we'll explore containers in detail, including writing Dockerfiles and .yaml (yet another mark-up language) files.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Scalability in MLOps Pipelines

MLOps pipelines consist of three main stages, each requiring scalability:

1. **Data Preprocessing**: handling large datasets, feature engineering at scale
2. **Model Training**: distributed training, hyperparameter tuning
3. **Model Deployment**: serving predictions at scale, A/B testing

Tools:

- Data: Apache Spark, Dask, Ray
- Training: Kubeflow, MLflow, Weights & Biases
- Deployment: Kubernetes, Docker, cloud platforms (AWS SageMaker, GCP Vertex AI)

Data Preprocessing at Scale

Challenges:

- Processing terabytes of raw data
- Feature engineering on large datasets
- Data validation and quality checks

Solutions:

- Distributed processing: Apache Spark, Dask
- Batch processing: Airflow, Prefect
- Stream processing: Kafka, Flink

Best practices:

- Partition data effectively
- Cache intermediate results
- Use columnar formats (Parquet, ORC) for efficiency

Training at Scale

Challenges:

- Training large models (GPT, BERT, ResNet)
- Hyperparameter tuning (thousands of experiments)
- Managing compute resources

Solutions:

- Data parallelism: split data across GPUs
- Model parallelism: split model across GPUs
- Pipeline parallelism: split layers across stages
- Distributed hyperparameter search: Ray Tune, Optuna

Example frameworks: PyTorch DDP, Horovod, DeepSpeed, Megatron-LM

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Deployment at Scale

Challenges:

- Serving thousands of predictions per second
- Low latency requirements (real-time inference)
- Model versioning and rollback
- A/B testing different models

Solutions:

- Model serving: TensorFlow Serving, TorchServe, Triton
- Batch inference for offline predictions
- Kubernetes for auto-scaling
- Feature stores for consistent features (Feast, Tecton)

UNIVERSITÀ DEGLI STUDI DI TRIESTE

References

- HPC course repository
- ORFEO DOC

UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

Thank You!