Introduction to Containers

Prepared by:
Ms. Avita Katal
Assistant Professor (SG)
School of Computer Science
UPES, Dehradun

Objectives

- Identify the traditional computing issues for software development.

- Define a container and describe its characteristics and list container benefits and challenges, and popular container vendors.

Digital Transformation and Containers

- **Cloud-native** is the newest application development approach for building **scalable, dynamic, hybrid cloud-friendly software.**

- Container technology is a powerful part of that approach.

- Let's check out the analogy of a shipping container. The modern shipping industry standardized a set of container sizes, so no matter what item is shipped, the container size remains the same.

    - Standardization significantly improves shipping efficiency.

    - Logistics staff select container transport options such as ships, planes, trains, and trucks, based on the container's size and the client's delivery needs.

- Digital container technology is similar. Containers solve the problem of **making software portable so that applications can run on multiple platforms.**

Digital Transformation and Containers

- A container, powered by the containerization engine, is a standard unit of software that encapsulates the **application code, runtime, system tools, system libraries, and settings necessary for programmers to build, ship, and run applications efficiently**.

- Operations and underlying infrastructure issues are no longer blockers.

- You can quickly move applications from **your laptop to a testing environment**, from a **staging environment to a production environment,** from a **physical machine to a virtual machine**, or a **private cloud or public cloud**, and always know that your application will work correctly.

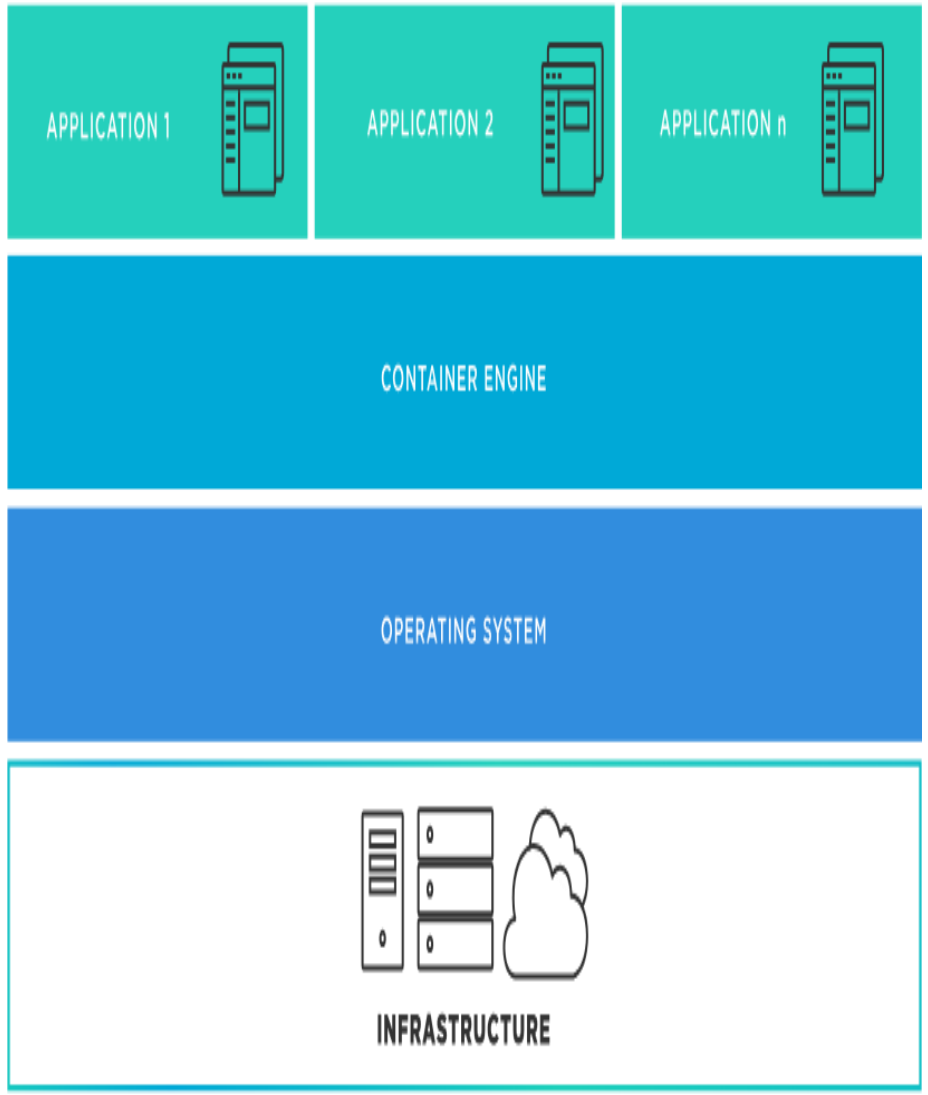
Why use Containers?

- A container can be small, **just tens of megabytes**, and developers can almost instantly start containerized applications. With these capabilities, containers serve as the foundation for **today's development and deployment solutions standards.**

- Let's examine some of the development and deployment challenges organizations encounter with traditional computing environments .Containers enable organizations to overcome these challenges:
  - **ISOLATION AND ALLOCATION:** In traditional environments, developers can't isolate the app and allocate or designate specific storage and memory resources for apps on physical servers.
  - **SERVER UTILIZATION:** Servers are often underutilized or overutilized, leading to poor utilization and a poor return on investment.
  - **PROVISIONING AND COSTS**: Traditional deployments require comprehensive provisioning resources and expensive maintenance costs.
  - **PERFORMANCE:** The limits of physical servers can constrain application performance during peak workloads.
  - **PORTABILITY:** Applications are not portable across multiple environments and operating systems.
  - **RESILIENCY:** Implementing hardware for resiliency is often time-consuming, complex and expensive.
  - **SCALABILITY:** Traditional on premises IT environments have limited scalability
  - **AUTOMATION:** Finally, automation is challenging when distributing software to multiple platforms and resources using traditional environments.

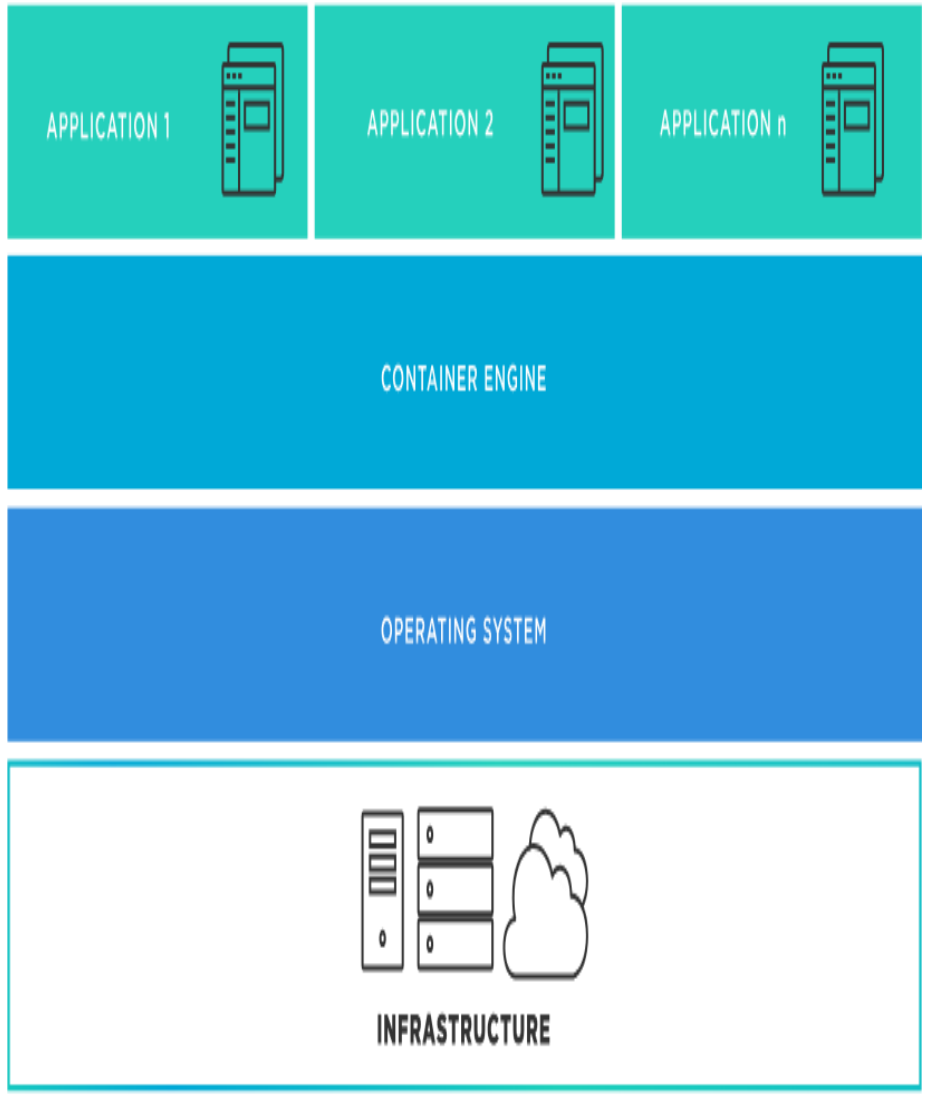
Container Characteristics

- Container engines **virtualize the operating system** and are responsible for running containers.
- **Platform-independent** containers are lightweight, fast, isolated, portable, and secure and often require less memory space.
- Binaries, libraries, and other entities within the container enable apps to run, and one machine can host multiple containers. Containers help programmers **quickly deploy code into applications.**
- Containers are platform-independent and can run on the **cloud, desktop, and on-premises**.



Container Characteristics

- Containers being **operating system-independent**, run on **Windows, Linux, or Mac OS**.
- Containers are **also programming language and IDE independent**—whether you are running Python, Node, Java, or other languages.
- Containers enable organizations to quickly create applications using automation.
- Lower deployment time and costs.
- Improve resource utilization, including CPU and memory.
- Port across different environments, and support next-gen applications, including microservices.



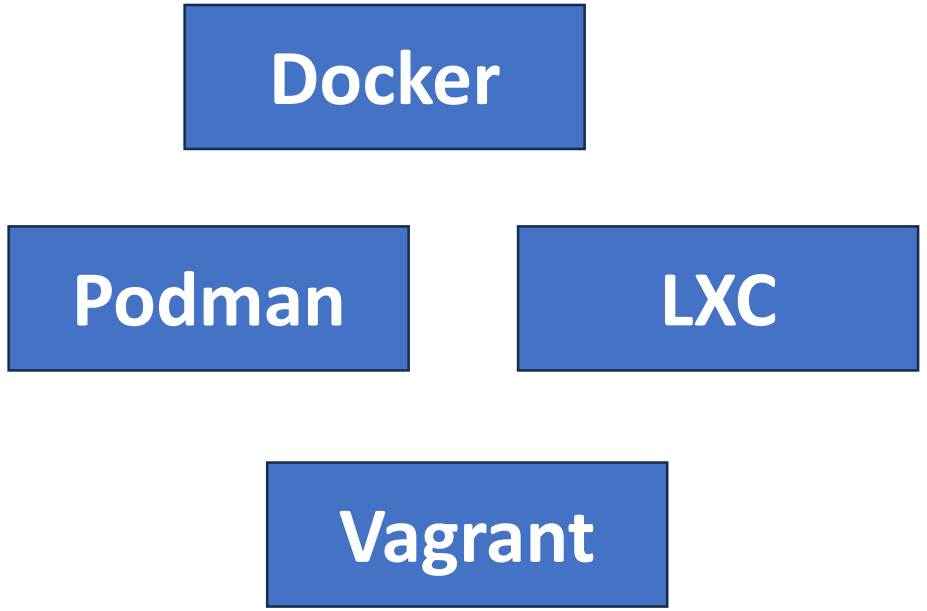
Container Challenges

Using containerization is not without its challenges:

- **Server security** can become an issue if its operating system is affected.
- Developers can become overwhelmed when managing thousands of containers.
- **Converting monolithic legacy applications** can be a complex process, and developers can experience difficulty **right-sizing containers** for specific scenarios.

Container Vendors

- ***Docker*** is a robust platform and the most popular container platform today.
- ***Podman*** is a daemon-less container engine that is more secure than Docker.
- Developers often prefer ***LXC*** for data-intensive applications and operations.
- And, ***Vagrant*** offers the highest levels of isolation on the running physical machine.

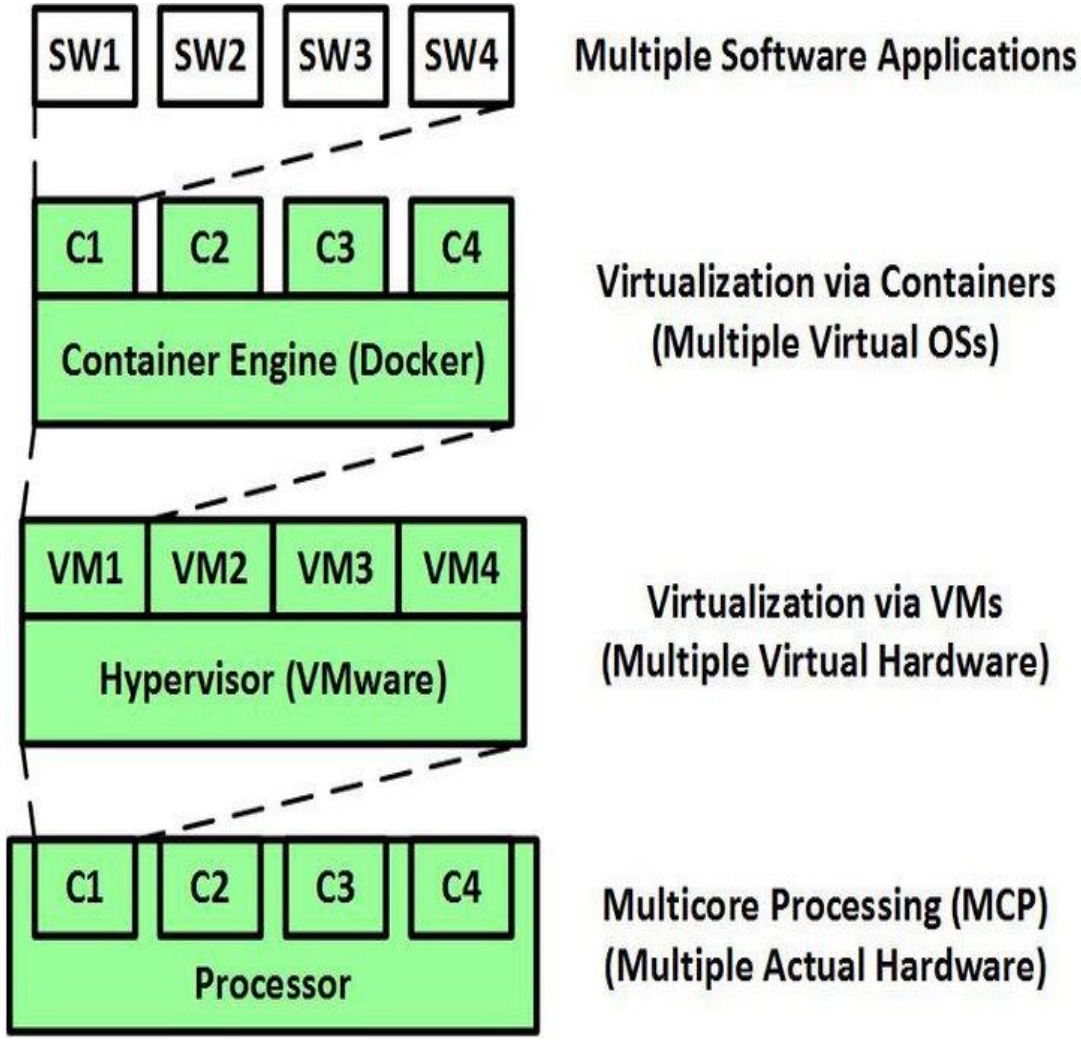


Containerization

A **container** is a virtual runtime environment that runs on top of a single operating system (OS) kernel and emulates an operating system rather than the underlying hardware.

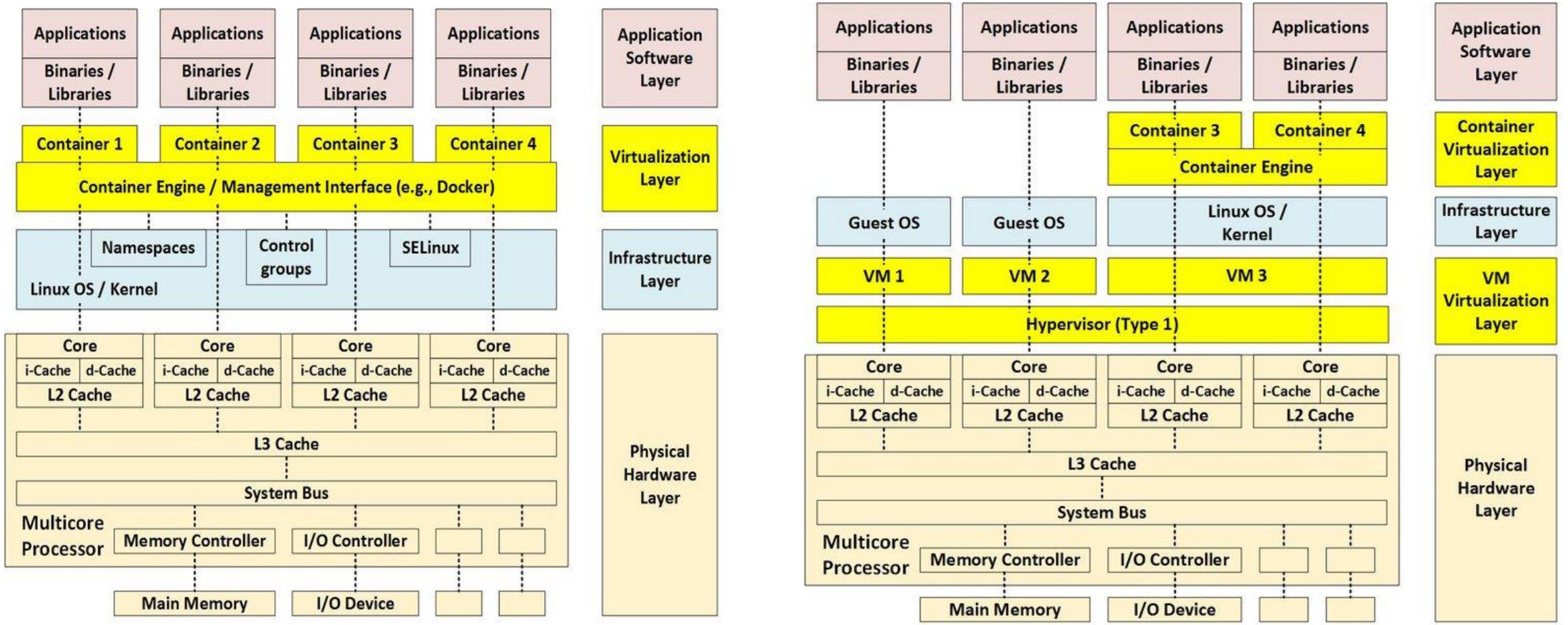
A **container engine** is a **managed environment** for deploying containerized applications. The container engine allocates cores and memory to containers, enforces spatial isolation and security, and provides scalability by enabling the addition of containers.

A **hybrid container architecture** is an architecture combining virtualization by both virtual machines and containers, i.e., the container engine and associated containers execute on top of a virtual machine. Use of a hybrid container architecture is also known as hybrid containerization.



Source: https://insights.sei.cmu.edu/blog/virtualization-via-containers/

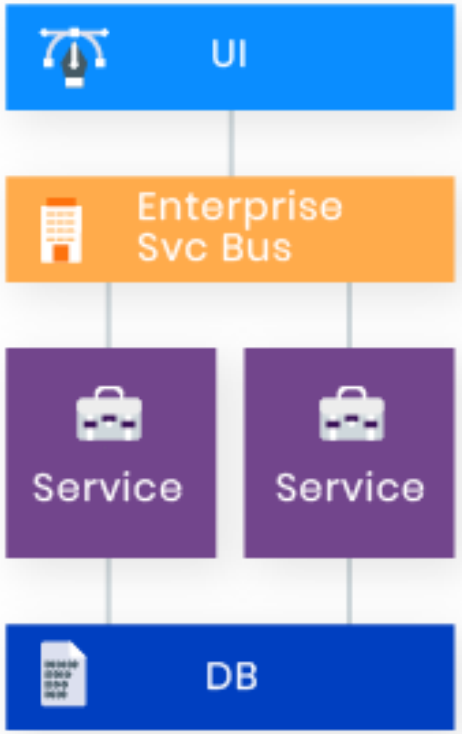
Containerization



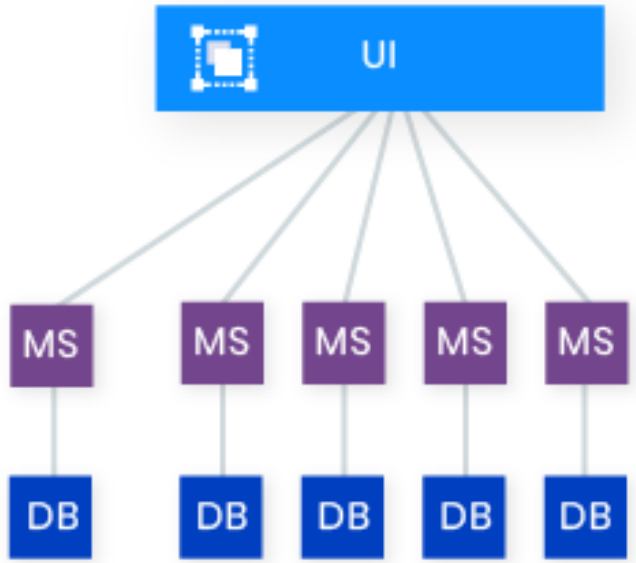
Source: https://insights.sei.cmu.edu/blog/virtualization-via-containers/

SERVER ARCHITECTURE
Monolithic vs SOA vs Microservices

Monolithic



Service - Oriented



Microservices

Monolithic Architecture

In software engineering, a monolithic pattern refers to a **single indivisible unit.** The concept of monolithic software lies in different components of an application being combined into a single program on a single platform. Usually, a monolithic app consists of a **database, client-side user interface, and server-side application**. All the software's parts are unified and all its functions are managed in one place.

Pros of a monolithic architecture

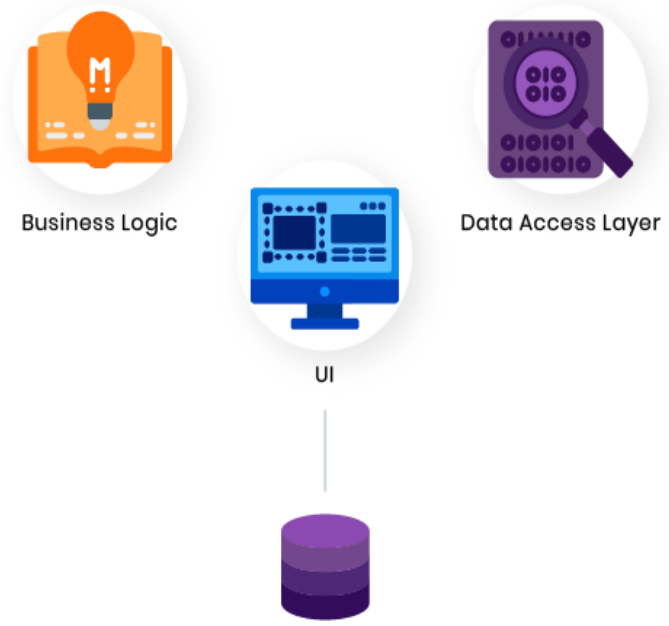
- Simpler development and deployment
- Fewer cross-cutting concerns
- Better performance

Cons of a monolithic architecture

- Codebase gets cumbersome over time
- Difficult to adopt new technologies
- Limited agility

The bottom line

- The monolithic model isn't outdated, and it still works great in some cases. Some giant companies like **Etsy stay monolithic** despite today's popularity of microservices.
- Monolithic software architecture can be beneficial if your team is at the **founding stage**, you're building an unproven product, and you have no experience with microservices.
- Monolithic is perfect for **startups** that need to get a product up and running as soon as possible.



Monolithic Architecture

Service Oriented Architecture

A service-oriented architecture (SOA) is a software architecture style that refers to an **application composed of discrete and loosely coupled software agents** that perform a required function. SOA has two main roles: a **service provider** and a **service consumer.** Both of these roles can be played by a software agent. The concept of SOA lies in the following: an application can be designed and built in a way that its modules are integrated seamlessly and can be easily reused.

Pros of a SOA architecture

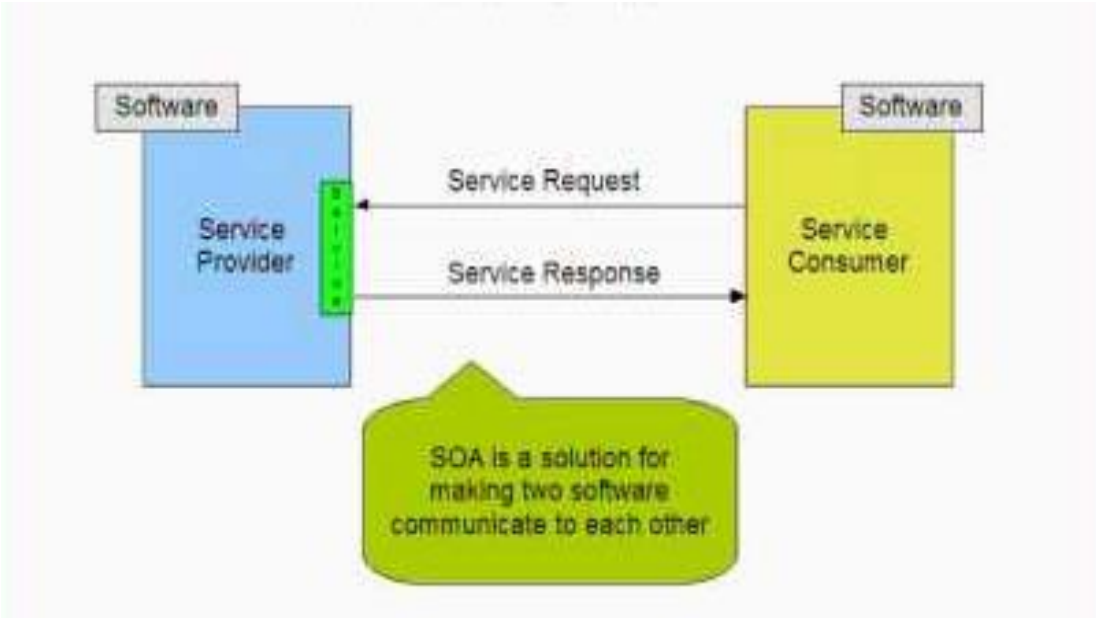
- Reusability of services
- Better maintainability
- Higher reliability
- Parallel development

Cons of a SOA architecture

- Complex management
- High investment costs
- Extra overload

The bottom line

- The SOA approach is best suited for complex enterprise systems such as those for **banks.**
- A banking system is **extremely hard to break into microservices**. But a monolithic approach also isn't good for a banking system as one part could hurt the whole app.
- The best solution is to use the SOA approach and organize complex apps into isolated independent services.



Microservice Architecture

- Microservice is a type of **service-oriented software architecture** that focuses on building a series of **autonomous components that make up an app.**
- Unlike monolithic apps built as a single indivisible unit, microservice apps consist **of multiple independent components** that are glued together with **APIs.**
- The microservices approach focuses mainly on **business priorities and capabilities,** whereas the monolithic approach is organized around technology layers, UIs, and databases.
- The microservices approach has become a trend in recent years as more and more enterprises become agile and move toward **DevOps.**

The bottom line

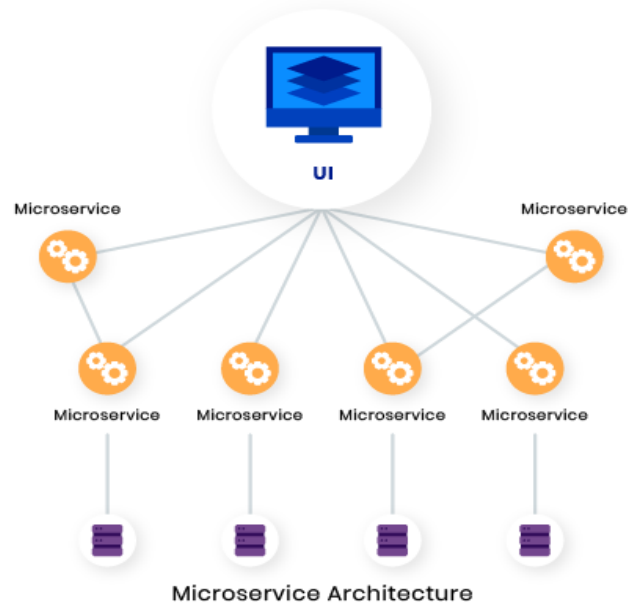
- Microservices are good, but **not for all types of apps**. This pattern works great for evolving applications and complex systems.
- Consider choosing a microservices architecture when you have **multiple experienced teams and when the app is complex enough to break it into services**.
- When an application is large and **needs to be flexible and scalable,** microservices are beneficial.

Pros of a SOA architecture

- Easy to develop, test, and deploy
- Increased agility
- Ability to scale horizontally

Cons of a SOA architecture

- Complexity
- Security concerns
- Different programming languages



SERVERLESS ARCHITECTURE

Serverless architecture is a cloud computing approach to **building and running apps and services without the need for infrastructure management**. In serverless apps, code execution is managed by a server, allowing developers to deploy code **without worrying about server maintenance and provision**. In fact, serverless doesn't mean "no server."

The serverless architecture incorporates two concepts:

- **FaaS (Function as a Service)**
- **BaaS (Backend as a Service)**

The bottom line

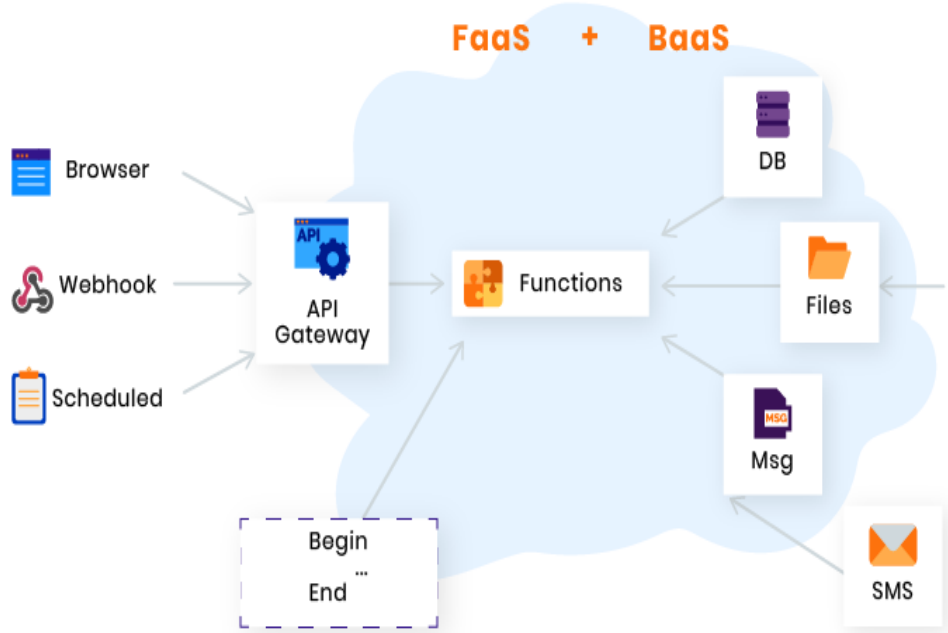
Serverless software architecture is beneficial for accomplishing **one-time tasks and auxiliary processes**. It works great for client-heavy apps and apps that are growing fast and need to scale limitlessly.

Pros of a Serverless architecture

- Enhanced scalability
- Lower costs
- Easy to deploy

Cons of a Serverless architecture

- Not for long-term tasks
- Vendor lock-in



Containerization Related to Microservices?

The main difference between microservices and containers is that microservices are an **architectural paradigm**, while containers are a **means to implement that paradigm**. Containers host the individual microservices that form a microservices application.

However, an enterprise can host and deploy microservices in a variety of other ways:

- **Serverless functions.** These provide **isolated environments** that run code preconfigured to respond to triggers such as a user login request.

- **VMs.** It's uncommon to host microservices inside VMs. Nevertheless, it's technically feasible for developers to deploy a set of microservices inside individual VMs and then connect them together to form a microservices app. This provides even **stricter isolation between microservices than do containers**.

- **Directly on the OS.** There is also **no technical reason why you can't deploy a set of microservices directly on the same OS** and not isolate them inside of a container or VM.

- **Unikernels.** These **lightweight, self-booting environments contain everything required to run a specific application or service.** They can be used to deploy microservices on a server without a conventional OS.

How microservices and containers work together?

Nevertheless, containers are the most popular way to implement a microservices architecture, for several reasons:

- ***Fast start times:*** VMs take up to a few minutes to start, but containers can typically start in just a few seconds. It's easier to maximize the agility (the easiness of a development team to understand and maintain an application) of microservices when they are hosted inside containers.
- ***Security:*** Containers provide isolation for each containerized application or microservice, which reduces the risk that a security vulnerability can spread. Microservices deployed directly on a host OS are less secure in this respect.
- ***Service discovery*****:** It's simpler for microservices to **locate and communicate with each other** when they all run in containers that are managed on the **same platform.** If you deploy microservices in VMs, serverless functions or unikernels, each host may have a **different networking configuration**. This makes it harder to **design a network architecture for reliable service discovery.**
- ***Orchestration:*** Along similar lines, microservices are easier to orchestrate -- **schedule, start, stop and restart -- when they run in containers on a shared platform.**
- ***Tools:*** The tools that support microservices deployment with containers have significantly matured over the past several years. **Orchestration platforms for containers, such as Kubernetes**, are massively popular and well supported in every major public cloud today. By comparison, there are **few tools to orchestrate microservices hosted in unikernels or VMs.**

When to choose containers vs. microservices?

- Put simply, for an enterprise that wants to deploy microservices, ***containers offer the best tradeoff among security, performance and management.***

- In some scenarios, it makes more sense to deploy microservices without containers -- for instance, a workload may require **especially strict isolation between microservices, or several microservices** require different OS environments. VMs might be better suited for these situations.

- Also, keep in mind that containers can host monolithic applications, not just microservices.

Recap

- Organizations are moving to containers to overcome challenges around isolation, utilization, provisioning, performance, and more.
- A container is a standard unit of software that encapsulates everything needed to build, ship, and run applications.
- Containers are operating system, programming language, and platform-independent. They lower deployment time and costs, improve utilization, automate processes, and support next-gen applications (microservices).
- Developers may find that management, legacy project migration, and right-sizing are significant challenges.
- And finally, major container vendors include Docker, Podman, LXC, and Vagrant.