IT Project Guidance

Introduction to Containers and Kubernetes

Version:

0.1

## Purpose

This document supports common understanding of container technologies across enterprise IT projects. It provides foundational guidance for teams transitioning from virtual machines to more flexible, scalable deployment models using containers. It aims to assist both technical and non-technical stakeholders in evaluating, adopting, and applying container strategies effectively.

## Synopsis

The document introduces containers as a modern deployment construct, contrasting them with traditional virtual machines and outlining their practical application in development and infrastructure. It includes an example project using .NET Core, explores supporting technologies like Docker and Kubernetes, and addresses deployment options, scaling, and operational concerns. Appendices cover advanced topics to support safe and scalable use in production contexts.

## Contents

[Purpose 1](#_Toc197435968)

[Synopsis 1](#_Toc197435969)

[Contents 2](#_Toc197435970)

[Introduction 4](#_Toc197435971)

[Objective 4](#_Toc197435972)

[Background 4](#_Toc197435973)

[Containers vs Virtual Machines 4](#_Toc197435974)

[Move Away from VMs 5](#_Toc197435975)

[Containers 5](#_Toc197435976)

[Images and Registries 6](#_Toc197435977)

[Advantages and Disadvantages 6](#_Toc197435978)

[Advantages of Containers 6](#_Toc197435979)

[Disadvantages of Containers 6](#_Toc197435980)

[How Docker Serves Applications 7](#_Toc197435981)

[Considerations 7](#_Toc197435982)

[On Server and Desktop Runners 7](#_Toc197435983)

[On Container Layering (Stacking) 7](#_Toc197435984)

[On Container Immutability 8](#_Toc197435985)

[On Using Different Base Images on the Same Host 8](#_Toc197435986)

[On Kubernetes 9](#_Toc197435987)

[Demonstration 9](#_Toc197435988)

[Prerequisites 10](#_Toc197435989)

[Create a web service 10](#_Toc197435990)

[Deploying to Azure 10](#_Toc197435991)

[Next Steps - consider WebAssembly 12](#_Toc197435992)

[How Much the Demonstration Covers 12](#_Toc197435993)

[Conclusion 13](#_Toc197435994)

[Appendices 14](#_Toc197435995)

[Appendix A - Document Information 14](#_Toc197435996)

[Versions 14](#_Toc197435997)

[Images 14](#_Toc197435998)

[Tables 14](#_Toc197435999)

[References 14](#_Toc197436000)

[Review Distribution 14](#_Toc197436001)

[Audience 14](#_Toc197436002)

[Structure 14](#_Toc197436003)

[Diagrams 15](#_Toc197436004)

[Acronyms 15](#_Toc197436005)

[Terms 15](#_Toc197436006)

[Appendix B – Secrets and Configuration Injection 16](#_Toc197436007)

[Appendix C – Persistent Storage Strategies 16](#_Toc197436008)

[Appendix D – Image Security and Optimisation 16](#_Toc197436009)

[Appendix E – Monitoring and Logging 16](#_Toc197436010)

[Appendix F – Autoscaling and Resource Constraints 17](#_Toc197436011)

[Appendix G – Multi-Container Patterns and Networking 17](#_Toc197436012)

# Introduction

This document was developed in response to the increasing interest across programmes and delivery teams in container-based application deployment. The shift away from virtual machines—driven by cost pressures, evolving security models, and the need for greater agility—has left many architects and developers needing a common frame of reference to evaluate and adopt containerisation patterns. This document contributes to that need by consolidating foundational knowledge, presenting practical examples, and demystifying the deployment lifecycle.

It aims to serve both technical specialists and decision-makers by framing containers not merely as a development tool, but as a critical shift in operational thinking. In particular, it clarifies where containerisation overlaps with and diverges from legacy deployment models, and provides a usable starting point for teams assessing how to modernise their infrastructure and delivery practices.

# Objective

This document provides a plain-language introduction to container technologies, focusing on their role in modernising deployment and infrastructure patterns. It is intended for both technical professionals and decision-makers who may be encountering containerisation as part of a shift away from virtual machines, toward more efficient, scalable, and maintainable software architectures.

The content distinguishes containers from earlier deployment approaches, outlines their practical advantages and disadvantages, and walks through a basic example using .NET Core to demonstrate real-world application. It also explains container layering, how containers interact with web servers, and concludes by touching on emerging alternatives such as WebAssembly.

# Background

Before exploring how containers rose to prominence, it's important to understand the limitations of earlier deployment models and the infrastructure context in which containers evolved. This section outlines how containers differ from virtual machines, why those differences matter, and the broader shift in strategy and cost models that has followed.

## Containers vs Virtual Machines

Virtual machines (VMs) and containers both allow applications to run in isolated environments, but they differ fundamentally in how they achieve this. VMs simulate complete hardware systems and include a full operating system for each instance. They are typically managed through hypervisors like VMware ESXi or Microsoft's Hyper-V and are well-suited to legacy systems, monolithic applications, or environments requiring strong separation.

Containers, on the other hand, run at the operating system level. They share the host OS kernel and isolate application processes, making them far lighter and faster to start. A container includes just the application and its dependencies—not an entire OS. This difference enables much higher resource efficiency and density.

VMs may be more appropriate when strong security boundaries between workloads are needed, or when running different operating systems on the same host. Containers shine in scenarios requiring rapid startup, dynamic scaling, high portability across environments, and simplified CI/CD workflows. They are particularly attractive for microservice architectures and modern development practices.

## Move Away from VMs

In recent years, the industry has steadily moved away from reliance on virtual machines for application hosting, driven by both architectural considerations and increasingly unsustainable cost structures. VM-based environments such as those powered by VMware ESXi or Hyper-V require full guest operating systems, leading to significant overhead in terms of memory, storage, and startup times. This inefficiency was often tolerated due to the mature management and clustering features of platforms like VMware vSphere. However, recent substantial price increases—especially following Broadcom's acquisition of VMware—have triggered a sharp industry backlash. Many organisations have responded by accelerating their exit strategies from VMware infrastructure.

## Containers

Containers, by contrast, offer lightweight process-level isolation without the need for duplicating the operating system. This allows for far greater density and efficiency. On top of this, open container orchestration platforms like Kubernetes have matured, providing the high-availability and resource management capabilities once unique to enterprise VM solutions. As a result, containers not only offer a more cost-effective alternative but also avoid vendor lock-in and support modern DevOps pipelines. The confluence of rising VM costs and increased container maturity has made the transition away from traditional VM platforms both attractive and urgent for many enterprises.

Containers emerged as a response to the limitations of virtual machines. VMs package an entire operating system and run atop a hypervisor, which can be resource-heavy and slow to spin up. Containers, by contrast, share the host OS kernel and isolate applications at the process level, significantly reducing overhead. Originally popularised by Docker, containers gained enterprise traction when orchestration platforms like Kubernetes (sometimes colloquially referred to as "Kontainers") made it possible to manage large fleets of them at scale. While Azure PaaS services (App Service, Functions, etc.) provided elegant abstractions for hosting applications, they often obscured control and became rigid when hosting composite, multi-language, or plugin-heavy systems. Containers allowed for more control, consistent builds, and infrastructure flexibility across dev, test, and prod environments.

## Images and Registries

Containers are built into layered filesystems and stored as images. These images are versioned and pushed to container image repositories—also called registries—which function similarly to package managers like NPM or NuGet but for containers. Each image in a repository includes everything needed to run the application: binaries, configuration, environment variables, and runtime dependencies.

Popular public repositories include Docker Hub, GitHub Container Registry, and Azure Container Registry. Organisations can also host private registries to control access and manage versioned deployments securely.

# Advantages and Disadvantages

## Advantages of Containers

Containers provide lightweight, isolated environments that can run anywhere Docker (or compatible runtimes) are supported. They reduce the problem of "it works on my machine" by packaging dependencies with the application. They enable versioned deployments, rollback, and seamless CI/CD pipelines. Containers are ideal for microservices, as each service can run independently and be scaled individually. Developers can use containers to simulate production environments locally. Containers also support immutability and clean startup, reducing configuration drift.

## Disadvantages of Containers

Containerisation adds architectural complexity. Networking, storage, and persistent state management require deliberate design. Orchestration systems like Kubernetes offer power but are difficult to learn and overkill for smaller projects. Container startup time, while faster than VMs, is slower than serverless options. Debugging can be more complicated, particularly when multiple containers interact. Windows support is partial, with Linux containers being dominant. In scenarios where PaaS provides sufficient abstraction, containers may simply reintroduce operational responsibility.

Example: Hosting a Static Site with a DLL Dependency

# How Docker Serves Applications

When running a web application inside a Docker container, Docker itself is not acting as the web server. Instead, it creates an isolated environment in which your application can run just as it would on a physical machine. In the case of a .NET Core application, this typically means Kestrel—the internal web server built into ASP.NET Core—is what actually listens for HTTP requests. Docker simply maps the container's internal port (exposed by Kestrel) to a port on the host machine. This means Docker's role is infrastructure: it provides process isolation, environment consistency, and network routing, but does not replace or inject a web server. If you ran the same application outside of a container, it would still rely on Kestrel unless explicitly fronted by IIS or NGINX. Containers just encapsulate the application along with its dependencies, so you don’t need to manually configure hosting environments.

# Considerations

Below are explored some considerations that help form an understanding of what containers are and provide.

## On Server and Desktop Runners

To run containers, systems need to have a compatible container runtime installed. On servers, the most common setup involves installing Docker Engine directly. This is a lightweight runtime suitable for headless environments, providing command-line tools and services to run, monitor, and network containers efficiently. Alternatively, containerd may be used—this is a lower-level runtime that Docker itself relies on, and is also used by Kubernetes for production-grade orchestration.

For desktop development environments, Docker Desktop is the most widely adopted option. It packages the Docker Engine with a graphical interface, system tray integration, and developer-friendly features. On Windows, it integrates with WSL2 for performance and compatibility. On macOS, it uses a lightweight Linux VM under the hood.

Other options include Podman, which supports rootless operation and can serve as a drop-in replacement for Docker CLI in many contexts. Rancher Desktop offers a GUI-based alternative that includes Kubernetes support, while Minikube simulates a full Kubernetes cluster locally.

Due to container technology being open source and standards-driven (notably under the Open Container Initiative), there is no vendor lock-in in the choice of runtime. This flexibility supports varied security models, resource constraints, and team capabilities.

## On Container Layering (Stacking)

Containers are not class hierarchies. However, images can be layered.

The Dockerfile's FROM directive pulls a base image. You can create a shared base image (e.g. with common DLLs or runtime configurations), then build multiple containers on top of it. For example:

1. Base image includes shared assemblies in /shared
2. App containers copy only their specific assemblies and reference /shared
3. This supports reuse, smaller image sizes, and consistency

## On Container Immutability

However, containers are immutable once built. They do not "inherit" changes dynamically from the base. Any update to the base requires re-building dependent images. This is more akin to inheritance in build-time layering than runtime dependency injection.

## On Using Different Base Images on the Same Host

A server hosting containers can run multiple containers side-by-side, each using a different base image. For example, one container might use mcr.microsoft.com/dotnet/aspnet:6.0, and another might use mcr.microsoft.com/dotnet/aspnet:8.0. This is possible because each container includes its own runtime environment and dependencies. The host OS only needs Docker installed—it does not need to install or maintain the individual runtimes used within each container. This means the ASP.NET Core version inside each container is completely isolated. The trade-off is that duplicated dependencies across containers may consume more disk space and memory, but this is mitigated by Docker's layered caching model, which can share unchanged base layers between containers when possible.

This architecture allows for high flexibility and compatibility, as containerised applications are decoupled from the host and from each other, avoiding version conflicts.

When running a web application inside a Docker container, Docker itself is not acting as the web server. Instead, it creates an isolated environment in which your application can run just as it would on a physical machine. In the case of a .NET Core application, this typically means Kestrel—the internal web server built into ASP.NET Core—is what actually listens for HTTP requests. Docker simply maps the container's internal port (exposed by Kestrel) to a port on the host machine. This means Docker's role is infrastructure: it provides process isolation, environment consistency, and network routing, but does not replace or inject a web server. If you ran the same application outside of a container, it would still rely on Kestrel unless explicitly fronted by IIS or NGINX. Containers just encapsulate the application along with its dependencies, so you don’t need to manually configure hosting environments.

## On Kubernetes

Kubernetes is an open-source orchestration platform designed to manage the deployment, scaling, and operation of containerised applications. While Docker made it easy to run individual containers, Kubernetes emerged to address the complexity of running containers in production at scale—particularly across multiple servers or cloud regions.

At its core, Kubernetes coordinates which containers run on which nodes, ensures they stay running (restarting them if necessary), manages scaling (adding or removing containers based on load), handles service discovery and networking, and supports rolling updates with rollback if needed. This effectively replicates and extends what developers used to get from Azure PaaS platforms like App Services or Functions, however the added complexity provides significantly more flexibility and control.

Kubernetes supports declarative configuration: developers define desired state in a YAML file (e.g. “3 replicas of this app running”), and Kubernetes actively maintains that state. For example, it can be configured to maintain a minimum of 1 running instance of a container, scaling up to 3 replicas in response to CPU usage, memory thresholds, or other load indicators. This enables elastic scaling while enforcing operational constraints.

It can automatically reschedule containers after failure, distribute traffic, and allocate resources across a cluster. This makes it ideal for elastic, highly-available services.

However, Kubernetes introduces substantial operational complexity. It has a steep learning curve, and its configuration model—while powerful—is often verbose and requires a deep understanding of networking, resource constraints, namespaces, and volume management. It is not well-suited for small projects or those without dedicated DevOps capability.

In summary, Kubernetes exists to make large-scale container deployment reliable, repeatable, and observable. It is increasingly the default orchestration platform used in enterprise and cloud-native environments where the benefits outweigh the complexity.

# Demonstration

This section provides a hands-on example to ground the preceding concepts in a tangible development workflow. The example demonstrates how to containerise a basic .NET-based web application that relies on a shared DLL. It walks through the steps of compiling, publishing, and packaging the application into a container image, then running it using Docker. This example is intended to show how containers encapsulate not just executables but also their runtime and dependency context, enabling consistent execution across environments.

## Prerequisites

Install Docker Desktop (or equivalent Docker runtime) on your development machine.

## Create a web service

To demonstrate a basic container use case:

1. Create a .NET Core minimal web API project that serves a webpage or endpoint json.
2. Reference from the page a shared class library (DLL) that provides some dynamic content (e.g. a greeting or date service).
3. Build the project using dotnet build to confirm it compiles.
4. Create a Dockerfile with the following contents:

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base # Use official ASP.NET Core runtime as the base image

WORKDIR /app # Set the working directory inside the container

COPY ./publish . # Copy published files from local folder into container

ENTRYPOINT ["dotnet", "YourApp.dll"] # Run app using the dotnet runtime and compiled DLL

1. publish it using dotnet publish.

dotnet publish -c Release -o publish # Compile and publish your .NET app to the 'publish' folder

docker build -t static-site . # Build the Docker image and tag it as 'static-site'

docker run -p 5000:80 static-site # Run the img as a container, mapping local port 5000 to container port 80

1. Visit http://localhost:5000 to view the webpage served from the container, using your referenced DLL.

## Deploying to Azure

To run the same container in Azure, several hosting options are available. If this is your first time deploying containers to the cloud, **Azure Web App for Containers** is the most straightforward choice. It offers predictable hosting with familiar App Service mechanisms and integrates easily with source control or container registries.

Once you are comfortable with container workflows, or if your workload benefits from event-driven autoscaling, **Azure Container Apps** provides a more flexible and cost-efficient environment. It supports serverless-style deployment with built-in scaling and workload isolation.

For small-scale or demonstration purposes:

* **Azure Container Apps (ACA)** is recommended. It supports autoscaling, includes HTTPS endpoints, and charges based on execution and resource usage, not uptime.
* **Azure Web App for Containers** incurs fixed monthly costs but simplifies deployment via familiar App Service mechanisms.

Cost considerations:

* ACA can operate with near-zero cost for demo or development workloads if traffic is low and compute resources are minimal.
* Avoid using Azure Kubernetes Service (AKS) unless deploying at scale, due to the operational overhead and baseline costs.

To deploy to Azure Web App for Containers (simplest method) via command line:

1. Ensure your container image is published to a registry (e.g. Docker Hub or Azure Container Registry):
2. docker tag static-site <registry-name>.azurecr.io/static-site:v1

docker push <registry-name>.azurecr.io/static-site:v1

1. Create a resource group and Azure App Service Plan:
2. az group create --name container-demo-rg --location australiaeast

az appservice plan create --name container-demo-plan --resource-group container-demo-rg --is-linux --sku B1

1. Create the Web App pointing to the container image:
2. az webapp create --resource-group container-demo-rg --plan container-demo-plan --name container-demo-app \

--deployment-container-image-name <registry-name>.azurecr.io/static-site:v1

1. Configure container registry authentication if using a private registry:
2. az webapp config container set --name container-demo-app --resource-group container-demo-rg \
3. --docker-custom-image-name <registry-name>.azurecr.io/static-site:v1 \
4. --docker-registry-server-url https://<registry-name>.azurecr.io \

--docker-registry-server-user <username> --docker-registry-server-password <password>

1. Browse to https://container-demo-app.azurewebsites.net to verify the deployment.

This sequence can be integrated into automated build pipelines, enabling consistent container delivery to Azure as part of your CI/CD process.

Azure CLI and portal walkthroughs are available for both methods.

## Next Steps - consider WebAssembly

While Containers are robust and mainstream today, their limitations are leading to other directions being explored.

For example, if starting a new project that may take a year before being made available for use, consider WebAssembly-based containers may simplify some of the container-based patterns.

## How Much the Demonstration Covers

The walkthrough above provides approximately 70–75% of what most developers and architects need to become productive with containers.

It demonstrates:

* How to package an application and its dependencies into a container
* Docker image structure and layering
* Running containers locally and deploying to Azure
* Use of shared libraries and runtime environments
* Basic understanding of image registries and host isolation

Topics not covered here, and only alluded too in the appendices —but worth pursuing to be productive —include:

* Secrets and configuration injection
* Persistent storage strategies
* Image security scanning and optimisation
* Monitoring and logging strategies in production
* Resource constraints and autoscaling configuration
* Multi-container setups (e.g. Docker Compose)
* Inter-container networking and service discovery

While the example serves as a strong foundation for most use cases, advancing beyond it is essential for teams deploying production systems with complex requirements.

# Conclusion

Containers provide a pragmatic, powerful middle-ground between VMs and serverless/PaaS. They enable controlled, repeatable environments, suited to modular applications and plugin architectures.

While they increase operational complexity compared to PaaS, they offer far greater transparency, flexibility, and control. Layered images support reuse and help manage dependencies, though not as dynamically as shared in-memory references.

Appendices

Appendix A - Document Information

Authors & Collaborators

* Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft

### Images

**No table of figures entries found.**

### Tables

**No table of figures entries found.**

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Acronyms

API

: [Application Programming Interface](#Term_ApplicationProgrammingInterface).

DDD

: Domain Driven Design

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. [IT](#Acronym_IT) is a subset of ICT.

IT

: acronym for Information, using Technology to automate and facilitate its management.

UI

: User Interface. Contrast with [API](#Acronym_API).

### Terms

Refer to the project’s Glossary.

Application Programming Interface

: an Interface provided for other systems to invoke (as opposed to User Interfaces).

Capability

: a capability is what an organisation or system must be able to achieve to meet its goals. Each capability belongs to a domain and is realised through one or more functions that, together, deliver the intended outcome within that area of concern.

Domain

: a domain is a defined area of knowledge, responsibility, or activity within an organisation or system. It groups related capabilities, entities, and functions that collectively serve a common purpose. Each capability belongs to a domain, and each function operates within one.

Entity

: an entity is a core object of interest within a domain, usually representing a person, place, thing, or event that holds information and can change over time, such as a Student, School, or Enrolment.

Function

: a function is a specific task or operation performed by a system, process, or person. Functions work together to enable a capability to be carried out. Each function operates within a domain and supports the delivery of one or more capabilities.

Person

: a physical person, who has one or more Personas. Not necessarily a system User.

Persona

: a facet that a Person presents to a Group of some kind.

Quality

: a quality is a measurable or observable attribute of a system or outcome that indicates how well it meets expectations. Examples include reliability, usability, and performance. Refer to the ISO-25000 SQuaRE series of standards.

User

: a human user of a system via its UIs.

User Interface

: a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces ([GUI](#Acronym_GUI)) or Text/Console User Interfaces (TUI).

Appendix B – Secrets and Configuration Injection

Managing secrets and configuration securely is critical in containerised environments. Avoid baking secrets directly into images. Instead, use environment variables, mounted volumes, or secret management services like Azure Key Vault or Kubernetes Secrets. In development, .env files may be used locally, but these must never be committed to source control. CI/CD pipelines should inject secrets at runtime.

Appendix C – Persistent Storage Strategies

Containers are ephemeral by default. To persist state, mount external storage using volumes. Docker supports named volumes and host-mounts, while Kubernetes provides Persistent Volume Claims. Choose strategies based on performance, durability, and sharing needs. For Azure, use Azure Files for cross-pod access or Azure Disks for high-performance block storage.

Appendix D – Image Security and Optimisation

Minimise attack surface by using slim base images (e.g. Alpine-based variants), remove build tools in final stages, and scan images using tools like Trivy or Microsoft Defender for Containers. Regularly patch base images and avoid unnecessary layers. Implement image signing to ensure provenance.

Appendix E – Monitoring and Logging

Integrate logs using stdout/stderr streams which container runtimes capture. Use logging sidecars or agents for advanced needs. For monitoring, tools like Prometheus and Grafana provide metrics, while services like Azure Monitor offer integrated visibility. Include health checks and readiness probes.

Appendix F – Autoscaling and Resource Constraints

Specify CPU/memory limits in your container definitions to avoid resource contention. Kubernetes and Azure Container Apps support autoscaling based on CPU, memory, or HTTP traffic. Define minimum/maximum replica settings, and monitor scaling behaviour in test environments to refine thresholds.

Appendix G – Multi-Container Patterns and Networking

Many real-world systems consist of multiple services that must run in coordination. Multi-container setups allow developers to compose such systems and define how containers interact.

Tools like Docker Compose simplify this by letting users define multiple containers and their interdependencies in a single YAML file. This enables:

* Coordinated startup and shutdown of containers
* Network isolation via Docker-defined bridge networks
* Named services for built-in DNS-based discovery

In Kubernetes, multi-container architectures can be modelled via Pods (which share network and storage context) or separate Deployments communicating via Services. Kubernetes supports service discovery, load balancing, and DNS resolution for internal communication.

For production environments, secure service-to-service communication, fault isolation, and scaling independence are key concerns. Consider:

* Using internal-only service definitions (non-public endpoints)
* Employing sidecar containers for logging, monitoring, or proxying
* Managing cross-service traffic using service meshes like Istio or Linkerd

In both Docker and Kubernetes, understanding network namespaces, port mapping, and container naming conventions is critical to successfully deploying interdependent services.