docker | CARIAD
A VOLKSWAGEN GROUP COMPANY

TECHNICAL WHITE PAPER

DOCKER DESKTOP · ENTERPRISE ARCHITECTURE

How Docker Desktop works under the hood

A comprehensive guide to how Docker Desktop operates on a Windows workstation — its deployment model, security boundaries, and networking stack.

Valentin Pravtchev, Sr DevOps Engineer
Julius Pravtchev, Sr. DevOps Engineer

2026

Executive summary

As the container runtime layer on Windows becomes part of the operating system itself, enterprises face a sharper question than they did a few years ago. The question is no longer how to run Linux containers on a Windows workstation, but how to securely scale container use for enterprises.

This paper answers that by documenting Docker Desktop's architecture: its deployment and policy model, its security boundaries, and its networking stack. The short version: the OS handles the virtualization layer and runtime; Docker Desktop handles enterprise scaling and compliance. Centralized policy through the Admin Console, SSO and SCIM, Organizational Access Tokens, Enhanced Container Isolation, and VPN-transparent networking have no equivalent in a bare runtime. For any organization, that security and control are the reasons to keep Docker Desktop. The chapters that follow are the evidence behind that conclusion, written so security, IT, and the development teams can evaluate Docker Desktop together.

WHY THIS MATTERS NOW

The container runtime layer on Windows is evolving. As the operating system absorbs more of what once required third-party tooling, enterprises face a question: what does Docker Desktop provide that a built-in runtime does not? This paper gives you the technical basis to make that assessment.

Introduction

Most software professionals still remember Steve Ballmer's infamous 2001 declaration that "Linux is cancer." A recent statement of equal magnitude, but maybe registered by much fewer folks, came from Jensen Huang at CES 2025, when he hailed the combination of Windows and the Windows Subsystem for Linux (WSL) as a "world-class AI PC."

One might wonder why we are starting this whitepaper with this story, but during our research, it became clear that containerization and streamlined multi-component cross-platform tools like Docker Desktop - and the engineering excellence behind them - have played and are still playing

a crucial role in this paradigm shift. Yet, beyond the code and architecture, one must not overlook the massive emotional component these technologies carry.

Introduced by Solomon Hykes in 2013, Docker built on existing Linux container technologies but was the first to package them into a clean CLI and a component-based ecosystem that made containers accessible for everyday development. It lowered the barrier to entry so effectively that containers moved from niche infrastructure tricks to a mainstream development standard almost overnight, and while the technical improvements were significant, the emotional impact was just as powerful. Docker gave development teams a simple way to break free from IT micromanagement, endless ticket requests, and dependency negotiations.

Since those early days, the role of container tools has steadily evolved into a crucial cornerstone of the modern software development landscape. Today, they are deeply embedded in enterprise production platforms, where Kubernetes has become the standard for orchestrating workloads. At the same time, OCI compliant tools such as Buildah, Skopeo, and Podman are widely used in security sensitive environments and CI/CD pipelines due to their rootless capabilities. These platforms are typically built and maintained by specialized engineering teams who not only master cloud native technologies but also actively drive the adoption of new container solutions, often acting as internal evangelists who push innovation across the organization.

On the development side, containers play an equally transformative role. DevContainers have emerged as a modern standard, with many projects shipping fully defined development environments that go far beyond traditional virtual environments in consistency and reproducibility. Docker Desktop has become the easiest and most accessible way to run containers on local workstations, offering strong developer experience on top of the runtime and the ability to scale container development and manage microservice ecosystems, while seamlessly handling corporate constraints such as VPNs, proxies, firewalls, and restricted permissions. However, the IT departments responsible for these workstations often lack deep container expertise, which leads either to the rise of shadow IT solutions or to organizations remaining stuck in micromanagement patterns.

For most of containerization's history, the hard part of running Linux containers on Windows was something Docker Desktop quietly handled for you. That is no longer a safe assumption. At Build 2026, Microsoft announced WSL Containers — a container runtime built directly into WSL 2, reachable through `wslc.exe`, supporting both containerd and Docker. The operating system is moving into the layer Docker Desktop has owned for a decade. This whitepaper documents what

Docker Desktop actually does underneath that layer, so developers, IT, and security teams can answer the question that follows from the launch: when the OS runs Linux containers on its own, what are we still relying on Docker Desktop for?

Agenda

01

Introducing Docker Desktop to your enterprise

This paper is written for the platform, security, and procurement teams who will be asked, sooner than they expect, why the organization pays for Docker Desktop when Windows now runs containers natively. The chapters that follow are the evidence for that conversation.

Docker Desktop is covered by the Docker Subscription Service Agreement. It is free for personal use, education, non-commercial open-source projects, and small businesses with fewer than 250 employees and under $10 million in annual revenue. Larger organizations, government entities, and commercial users outside these limits need a paid Docker Pro, Team, or Business subscription. Enterprises should review the available subscription tiers on the Docker Pricing page. The plans differ not only in their monthly fees but also in the range of enterprise-grade features they include, as summarized in Table 1.

Capability	Pro	Team	Business
Kubernetes	✓	✓	✓
Commercial support	Add-on	Add-on	✓
Single Sign-On (SSO)	—	—	✓
Cross-domain identity management (SCIM)	—	—	✓
Hardened Docker Desktop	—	—	✓
Image / Registry Access Management	—	—	✓
Unlimited Docker Hub pulls	✓	✓	✓
Organization Access Tokens	—	Add-on	✓

Table 1 — Paid subscriptions and their included functionalities.

As a side note, enterprises, regardless of whether they meet the criteria for a paid subscription, may alternatively use Docker Community Edition (CE) within virtualized environments such as WSL, Hyper-V, or other hypervisors (see Figure 1). These options remain freely available but

do not include the advanced management and security capabilities provided under Docker's paid plans.

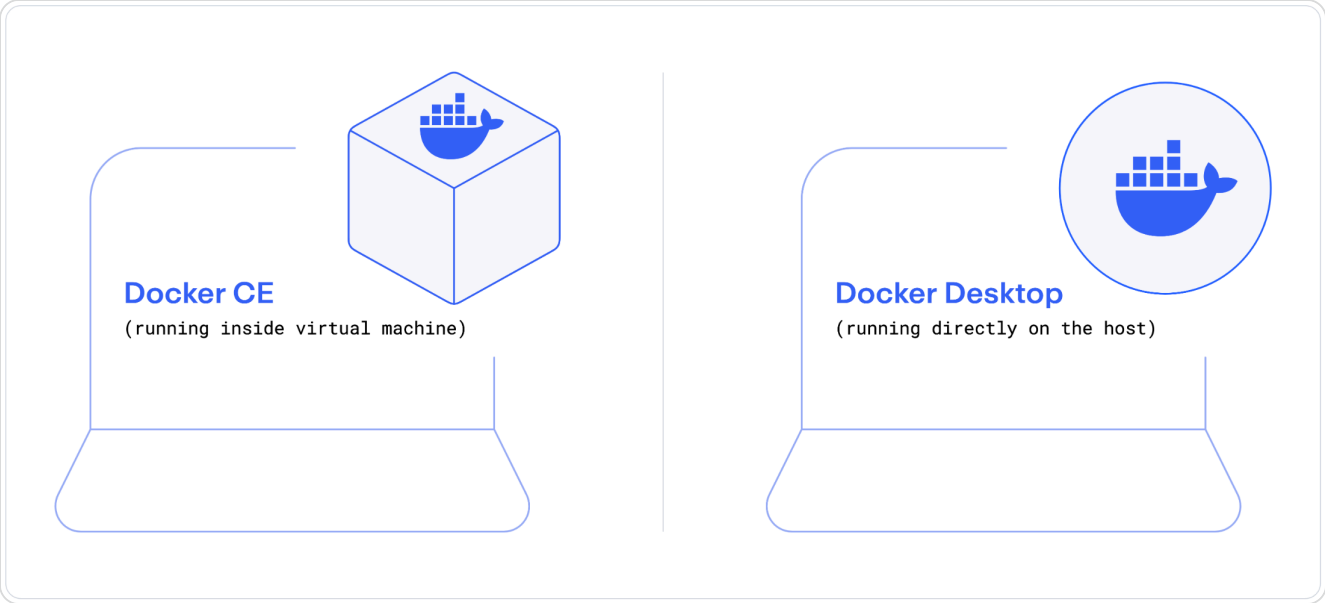

Figure 1 — Deploying Docker Desktop on the host or using Docker Community Edition (CE) inside a virtual environment.

02

Manage and secure your Docker organization

Running containers is only part of the challenge. Managing them at enterprise scale, enforcing consistent policy, and giving security teams the visibility they require demands a different class of tooling than a bare container runtime provides. Docker Desktop, specifically a Business subscription, addresses this directly.

Since we published *Using Docker Desktop at Large-Scale Enterprises*, which outlined how to set up and manage a Docker Hub organization, Docker Inc. has introduced several new enterprise features that can be configured at the organization level to further enhance company security. It is therefore worth providing a brief overview of the most important of these features.

Docker Inc. recently introduced the Admin Console, a centralized, web-based dashboard that allows administrators to manage organizations and teams within Docker Hub. The Admin Console serves as a single point of control for configuring company-wide Docker settings and enforcing consistent policies across all developer environments.

Docker provides centralized remote configuration management through the Admin Console, enabling administrators to enforce policies remotely without deploying a local `admin-settings.json` file on each workstation. Configuration updates now propagate automatically across all connected machines. Furthermore, administrators can define multiple configuration profiles and assign users to specific ones such as granting early access to experimental features for advanced users while keeping them disabled for others.

Recently, Company Management was introduced to allow enterprises to group multiple organizations under one company entity. This enables administrators to centrally configure company-wide settings, manage access, and enforce security policies such as Single Sign-On (SSO) and SCIM across all nested organizations.

Furthermore, enterprises can now generate Organization Access Tokens (OATs). These tokens are secure, programmatic credentials for CI/CD pipelines and automated systems. Unlike personal tokens, OATs are organization-owned, shared among multiple owners, and governed by dedicated usage limits. They also offer granular permission control, improved audit visibility, and the ability to track activity for enhanced security oversight.

Managing the organization is one half of the picture; the other is securing how developers access, build, and consume containerized software at scale. A Docker Business subscription enforces centralized governance over a large development environment and strengthens the software supply chain.

The foundation is identity. Integrating Docker with an enterprise identity provider through Single Sign-On (SSO) and SCIM-based user provisioning — the organization-level controls outlined above — ensures that only authorized employees can access Docker Desktop and related resources using corporate credentials. This applies consistent security policy across thousands of developers while simplifying user lifecycle management.

To reduce supply chain risk from untrusted components, Docker provides Image Access Management (IAM) and Registry Access Management (RAM). These controls restrict developers to approved container images and trusted registries, preventing the use of unknown community images or unauthorized external repositories that could introduce vulnerabilities or malicious code. By improving visibility and governance over software dependencies, they help mitigate supply chain threats similar to Log4Shell.

At the workstation, Docker Desktop's enterprise controls — Enforced Sign-In, centralized Settings Management, and Enhanced Container Isolation (ECI), examined in detail later in this paper — let security teams apply consistent policy at scale while reducing the attack surface of local development environments. The result is a more secure and auditable software supply chain: by standardizing development environments, controlling software provenance, and ensuring only trusted images and registries are used, organizations can accelerate delivery while improving security, governance, and developer productivity — a balance of developer self-service and enterprise-grade control.

A natural next step in this DevSecOps journey is adopting Docker Hardened Images (DHI) to further strengthen the supply chain. DHI provides secure, enterprise-maintained base images that integrate with existing development workflows, and can be served transparently through a registry pull-through cache so developers keep familiar workflows while automatically consuming more secure base images. Because DHI integrates cleanly with existing security analysis and vulnerability scanning tools, the improvement in image security can be measured directly — supporting the long-term goal of reducing vulnerability exposure and standardizing secure base images across teams without adding complexity for developers.

04

Deploying Docker Desktop across the enterprise

Streamlining the installation routine

Standard system-wide administration mode

On Windows 11, Docker Desktop supports two backends: a Linux backend, which integrates with WSL 2 or Hyper-V, either directly on the host (process-isolated) or within a lightweight Hyper-V virtual machine (hyperv-isolated). For both backends, specific Windows NT kernel features must be enabled. These can be activated by running following PowerShell routine from an elevated terminal.

```
# required optional Windows NT kernel features
$features = @("Containers", "Microsoft-Hyper-V-All", "VirtualMachinePlatform")
foreach ($feature in $features) {
  Enable-WindowsOptionalFeature -Online -FeatureName $feature -NoRestart
}
# restart system to activate features
Restart-Computer
```

Figure 2 — PowerShell routine for enabling the Windows NT kernel features required by Docker Desktop.

With the required Windows features enabled, Docker Desktop can be installed. Since version 4.32 Docker Inc. provides an MSI installer specifically tailored for large-scale rollouts through the organization's Mobile Device Management (MDM) system. Furthermore, Docker Desktop can easily be installed via the Microsoft Store. However, because many enterprises restrict access to the Microsoft Store under Zero Trust policies, and because MDM ecosystems differ significantly, this paper does not cover individual MDM integration procedures. Instead, it demonstrates a repeatable, vendor-agnostic deployment method using the MSI installer from an elevated PowerShell session.

The MSI package can be downloaded from the Enterprise Deployment section of organization's Admin Console and executed using `msiexec.exe` from an elevated PowerShell prompt. The MSI installer exposes several installation arguments that affect Docker Desktop's runtime behavior and its attack surface, so choosing appropriate options matters as shown in Figure 3.

```
msiexec /i "DockerDesktop.msi" /L*V ".msi.log" /quiet /norestart `
  ALLOWEDORG="your-organization" ALWAYSRUNSERVICE=1 DISABLEWINDOWSCONTAINERS=1
```

Figure 3 — Installing Docker Desktop from an elevated PowerShell using the MSI installer.

The `ALLOWEDORG="your-organization"` flag should be mandatory for large-scale rollouts. It enforces that every user signs in with their company Docker Hub organization identity, preventing the use of personal or unauthorized accounts. When applied, this flag creates a registry entry named `allowedOrgs` under `HKLM\Software\Policies\Docker\Docker Desktop`. This registry key ensures that security policies from the organization are enforced on the local workstation, safeguarding the entire setup and ensuring consistent authentication across all managed company devices.

The `ALWAYSRUNSERVICE=1` ensures that the privileged helper process `com.docker.service.exe` starts automatically as a Windows service during system boot. Without this flag, users would be prompted for elevated permission when starting Docker Desktop, particularly when using Windows containers or the Linux Hyper-V backend. Automating this startup improves the user experience and reduces administrative overhead.

If your organization does not require Windows containers, the `DISABLEWINDOWSCONTAINERS=1` flag will disable Windows containers entirely. This enforces Docker Desktop to run exclusively with Linux-based containers, which is a recommended best practice in secure enterprise environments as it significantly reduces the attack surface.

These flags, together with the Admin Console's centralized configuration, are the management layer the native runtime does not have. `ALLOWEDORG` ties every login to the company's Docker Hub identity, `ALWAYSRUNSERVICE=1` keeps the privileged helper available without UAC prompts, and `DISABLEWINDOWSCONTAINERS=1` removes an entire attack surface. A bare `wslc.exe` deployment gives an administrator none of this.

At large-scale companies, the deployment is usually performed by a technical system account. Thus, it is necessary to include following command to the installation routine that adds the user to the local Windows group `docker-users`. Membership to that group grants access to the protected Named Pipe `//./pipe/dockerBackendV2`, which is used for inter-process communication between the privileged helper process `com.docker.service.exe` and further Docker Desktop processes.

```
net localgroup docker-users <user> /add
```

Figure 4 — Adding the user to the local Windows group `docker-users` to grant access to the protected named pipe `dockerBackendV2`.

Enterprises should ensure that firewall rules do not block Docker Desktop's required outbound HTTPS connections listed in Table 2. These connections are necessary for registry integration, authentication, UI rendering, and receiving product updates. While it is technically possible to block these endpoints, doing so is not recommended, as it would disrupt essential functionality such as authentication, pulling / pushing images, and keeping the application up to date. To guarantee smooth operation, the relevant domains should be explicitly whitelisted. In later chapters, we will explore Docker Desktop's networking behavior in more detail to provide a deeper understanding of these communications.

URL	Purpose
`registry-1.docker.io`	Required to integrate with Docker Hub services, enabling developers to manage Container images via Docker Desktop UI.
`production.cloudflare.docker.com`	Required to push and pull images to/from Docker Hub for paid plans.
`hub.docker.com`	Required to push and pull images to/from Docker Hub for free plans.
`prod-publishers-content.s3.us-east-1.amazonaws.com`	Required to retrieve resources such as images displayed in the Docker Desktop UI.
`docker-pinata-support.s3.amazonaws.com`	Required to upload support Zips generated on the host to the remote storage system.
`auth.docker.io`	Required to authenticate against Docker Hub.
`login.docker.com`	Required to authenticate against Docker Hub.
`cdn.auth0.com`	Required to authenticate against Docker Hub.
`desktop.docker.com`	Required to fetch any update of Docker Desktop.
`api.dso.docker.com`	Required for usage of Docker Scout Services.
`dhi.io`	Required to pull Docker Hardened Images (DHI).

Table 2 — List of URLs Docker Desktop performs for outbound calls.

Privilege-free per-user mode

Introduced in version 4.72.0, the per-user mode eliminates the need for administrative privileges during both the installation and daily operation of Docker Desktop. Administrative rights are strictly restricted when enabling the WSL 2 system requirement. Following Figure 5 shows how to install WSL2; enterprises should consider specifying the `--no-distribution` flag. It ensures that only the required WSL2 components are deployed without downloading an unvetted Linux distribution, thereby preserving corporate compliance and minimizing the endpoint's attack surface.

```
wsl --install --no-distribution
```

Figure 5 — PowerShell command for installing and setting up WSL2 without any Linux distribution.

Once WSL 2 is in place, the executable installer can be downloaded from the official Docker Desktop Release Notes page. As the installer exposes several command-line arguments, selecting the appropriate flags is critical as shown in Figure 6.

```
'Docker Desktop Installer.exe' install --user --accept-license --quiet --allowed-org=
```

Figure 6 — Installing Docker Desktop in per-user mode from a non-elevated PowerShell.

As a result, the Docker Desktop resources are entirely deployed within the respective user-space under `%LOCALAPPDATA%\Programs\DockerDesktop` without any privileged helper daemon. While the per-user installer significantly reduces the endpoint's attack surface, it lacks support for the Linux Hyper-V backend and native Windows Containers. As the per-user installation mode is relatively new and currently in beta, the following examination focuses entirely on the well-established all-users mode to provide a standardized, proven baseline for traditional enterprise deployments.

05

How Docker Desktop integrates securely into Windows

After installation, it becomes clear that Docker Desktop is a multi-component application, relying on numerous executables, libraries, and supporting components distributed across the workstation for proper operation. The user-facing components, including both the graphical user interface (GUI) and command-line interface (CLI), are located under `C:\Program Files\Docker\Docker`. This directory contains the main GUI launcher, `Docker Desktop.exe`, along with its core libraries and shared assets, while additional GUI modules are organized in the resources and frontend subfolders. The CLI tools can be found in `C:\Program Files\Docker\resources\bin`, including the primary executables `docker.exe` and `kubectl.exe`, with additional plugins stored in `C:\Program Files\Docker\cli-plugins`. These binaries are invoked whenever a user runs a `docker <command>` from a Windows terminal or through the GUI.

Docker Desktop relies on several background processes that are deployed under `C:\Program Files\Docker\Docker`. Those binaries handle core functions like inter-process communication, networking setup, resource management between Windows host and Docker Desktop et cetera. Nearly all processes run with standard user rights, except for the Windows container engine, `dockerd.exe`, and `com.docker.service.exe`, then run under the elevated `NT AUTHORITY\SYSTEM` account.

According to Docker's official documentation, `com.docker.service.exe` follows the principle of least privilege. Although it runs with full SYSTEM permissions, it performs only the minimal set of elevated tasks required for Docker Desktop to operate. These include maintaining DNS entries in the Windows hosts file at `C:\Windows\System32\drivers\etc\hosts`, managing the lifecycle of the `DockerDesktopVM` used by the Hyper-V Linux backend, securely caching security-relevant policies, and a few other tightly scoped administrative operations. Thanks to this carefully engineered privilege model, and the installation decisions described in Streamlining the installation routine including the `ALWAYSRUNSERVICE=1` flag and the use of the local `docker-users` group, the day-to-day work with Docker Desktop can be performed entirely with standard user rights.

Furthermore, Docker Desktop installs several additional assets that may seem insignificant at first but become essential once we examine the bootstrapping process and overall system architecture. These include the `docker-desktop.iso` image located in `C:\Program Files\Docker\Docker\resources`, the `docker-wsl-cli.iso` image as well as the TAR archives `wsl-bootstrap.tar` and `wsl-data.tar` stored under `C:\Program Files\Docker\Docker\resources\wsl`. The ISO serves as the base image for the Hyper-V-based Linux backend via the `DockerDesktopVM` and is also used during initialization of the WSL-based Linux backend. During installation, `wsl-bootstrap.tar` is imported as the Alpine-based `docker-desktop` WSL2 distribution and is immediately converted into an `ext4.vhdx` virtual disk. The `wsl-data.tar` archive creates the `docker-desktop-data` distribution, which stores container images and runtime data. Beginning with Docker Desktop v4.30, `docker-desktop-data` has been replaced by a dedicated virtual hard disk located at `%APPDATA%\Local\Docker\wsl\disk`, which Docker Desktop now manages internally. Systems configured before this change continue to use `docker-desktop-data` for backward compatibility.

Process	Description
`com.docker.admin.exe`	Non-elevated helper that triggers the execution of elevated tasks of Docker Desktop such as starting `com.docker.service.exe`, downloading policies, etc.
`com.docker.backend.exe`	Main background service that coordinates and manages Docker Desktop's core backend operations services
`com.docker.build.exe`	Build manager, handling image builds and supporting targeted builders, adjustable logging, and optional timestamp control.
`com.docker.diagnose.exe`	Manage diagnostics, allowing users to generate, store, and optionally upload diagnostic bundles for troubleshooting.
`com.docker.llama-server.exe`	Docker Desktop's LLM inference engine, built on llama.cpp, that runs and serves AI models locally as the so-called Model Runner.
`com.docker.service.exe`	Elevated service that handles and executed elevated tasks like starting/stopping the Windows engine, managing Hyper-V VM, updating DNS entries, etc.
`com.docker.vmm.exe`	Manages the Virtual Machine Monitor (VMM) service, supporting installation and uninstallation of the VMM required for Hyper-V–based Linux container backends.
`DockerCLI.exe`	CLI tool for managing Docker Desktop, enabling engine switching, shutdown, shared drive listing, and verbose logging.
`docker-credential-wincred.exe`	Helper tool for Docker Desktop to securely handle credential management in the Windows Credential Manager.
`dockerd.exe`	Windows container engine

Table 3 — Excerpt of important backend services and utilities of Docker Desktop.

Named pipes everywhere: how Docker Desktop facilitates IPC on Windows

To establish fast and secure inter-process communication (IPC), Docker Desktop utilizes Windows Named Pipes. Named Pipes act as either one-way or bidirectional channels between one server and one or many clients. Unlike HTTP or other network protocols, data sent through a named pipe is not routed via the network stack, making them less receptive from inferences like firewalls or network policies. Access to a named pipe is governed by Windows' Access-Control

List (ACL) security model, allowing fine-grained control over which users or processes may connect to it. Docker Desktop utilizes named pipes so extensively that CyberArk even implemented PipeViewer inspection tool to enumerate and analyze them.

To aid understanding, we have attached A Lightweight Windows Named Pipe Example within the Appendix that implements Python-based Windows Named Pipe sender and receiver component.

06

A guided look into Docker Desktop's system architecture

Behind the scenes of the Windows container backend

The Windows backend bootstrapping process made simple

The bootstrapping process of Docker Desktop's Windows container backend involves a series of coordinated steps. Docker Desktop can be started either by selecting the application from the Windows Start menu or by running the following command from a Windows terminal.

```
docker desktop start
```

Figure 7 — Starting Docker Desktop from a Windows terminal.

When Docker Desktop is launched from the terminal, `docker.exe` executes the CLI plugin `docker-desktop.exe`, which starts `com.docker.backend.exe`. This backend process immediately spawns a new instance of itself using the `run` command, initiating a multi-layer bootstrapping sequence that brings all required backend services online. During this sequence, the backend creates multiple Windows Named Pipes, accesses the necessary Windows Registry keys, and launches several long-lived services. Among these long-live processes, `Docker Desktop.exe` prompts the user to authenticate with enterprise-linked Docker Hub organization and then spawns four additional instances of itself.

Additionally, the Windows Service Manager (`services.exe`) is responsible for launching the two essential background services, the Docker Engine Service and the Docker Desktop Service. When Docker Desktop is installed with the `ALWAYSRUNSERVICE=1` flag, these services start automatically at system boot, making them immediately available. Without this flag, the services

launch only when Docker Desktop is opened, triggering a UAC elevation prompt and negatively affecting the user experience in large-scale deployments, where users typically lack local admin rights.

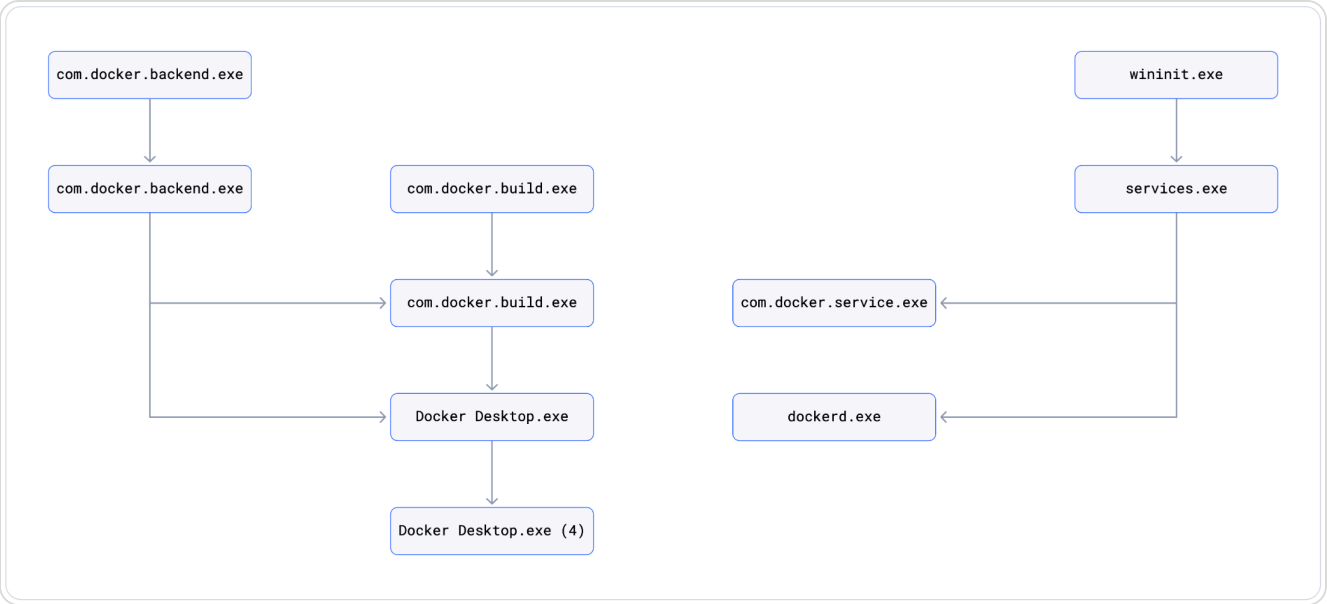

Figure 8 — The bootstrapping process of Docker Desktop's Windows container.

06

How Docker Desktop keeps Windows containers in line

In 2016, Microsoft Corp. introduced Windows containers and has continuously evolved this technology ever since. Windows containers also package applications in consistent and isolated environments but due to the architectural differences between the Windows NT kernel and the Linux kernel, they work differently behind the scenes. Without going too much into detail, Windows containers can operate in two different isolation modes, the so-called Hyper-V Isolation as well as Process Isolation.

On Windows 11, Docker Desktop uses Hyper-V isolation by default. In this mode, each container runs inside its own lightweight virtual machine that comes along with its own instance of the NT kernel. Hyper-V Isolation provides a strong security boundary as containers and host are isolated at hardware level. Figure 9 illustrates how to launch such a hyperv–isolated container using PowerShell.

```
docker run -it mcr.microsoft.com/windows/servercore:ltsc2022 powershell
```

Figure 9 — Running the standard Windows Server 2022 container in hyperv-isolation mode.

Behind the scenes, each interaction with hyperv-isolated containers, whether initiated via the Docker Desktop GUI or from a Windows terminal, ultimately results in a call to `docker.exe`. Depending on the command, `docker.exe` either directly forwards the request to `//./pipe/dockerBackendV2`, or uses a dedicated CLI-plugin, like `docker-compose.exe`, which, in turn, sends the request to the same Named Pipe. The elevated process `com.docker.service.exe` then relays the request to the Windows Engine, `dockerd.exe`, via `//./pipe/dockerDesktopWindowsEngine`. The Windows Engine then utilizes the packaged Golang interface `hcsshim` to interact with the Windows low-level Host Compute Service (HCS) API. It sends HCS-specific Remote Procedure Calls (RPC) that instructs the main HCS component `vmcompute.exe` to create and run a Hyper-V container using the Virtual Machine Worker Process, `vmwp.exe`. As shown in Figure 10, each Hyper-V container includes typical Windows system processes and a dedicated agent, `CExecSvc.exe`, responsible for host-to-container communication and launching user applications inside the container session.

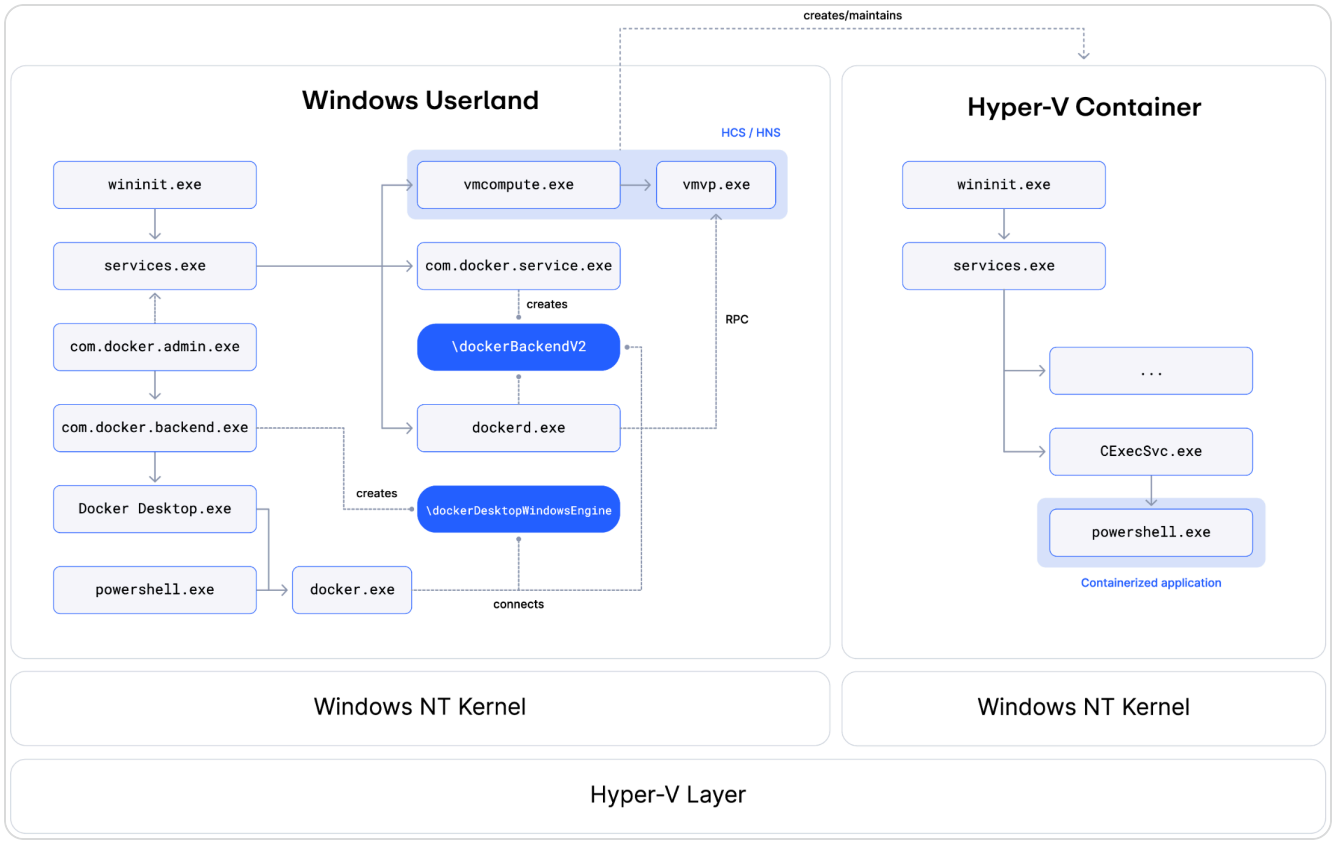

Figure 10 — System architecture when running a hyperv-isolated container on the Windows host.

Processes running inside a hyperv-isolated Windows container are not visible from the Windows host, thus, any exploit that occurs inside the container remains there. However, this also implies that host-based monitoring and security tools cannot inspect or detect what happens inside the container. Moreover, `dockerd.exe` operates under the LocalSystem account, which has extensive privileges on the Windows host. Any container started by the Windows Engine effectively inherits these elevated permissions. In consequence, any low-privilege user that is member of the `docker-users` group can still use bind-mounts to sensitive directories like `C:\Windows`, modify system-critical files from within the container and ultimately can elevate the permissions.

```
docker run -it -v C:\Windows:C:\Temp mcr.microsoft.com/windows/servercore:ltsc2022 pov
```

Figure 11 — Mounting the sensitive `C:\Windows` directory into a hyperv-isolated Windows container.

This risk is inherent to the Windows container engine and is not removed by switching to a native runtime; the value of Docker Desktop here is the policy controls that let you disable Windows containers entirely (`DISABLEWINDOWSCONTAINERS=1`) across managed devices.

In contrast, process-isolated Windows containers rely on traditional OS-level mechanisms, including namespaces, control groups, and other Windows kernel features, to separate containerized workloads from the host. These namespaces isolate key resources including the file system, registry, network ports, process and thread ID space, and the Windows Object Manager namespace. A process-isolated container can be launched using the `--isolation=process` flag, as illustrated in Figure 12.

```
docker run -it --isolation=process mcr.microsoft.com/windows/servercore:ltsc2022 power
```

Figure 12 — Launching a process-isolated Windows container from PowerShell.

Just like hyperv-isolated Windows containers, the `dockerd.exe` uses the Go-based `hcsshim` library to communicate with the HCS. The difference is that, in process isolation, HCS integrates directly with the Windows Session Manager, `smss.exe`, to register and configure the container's components on the host.

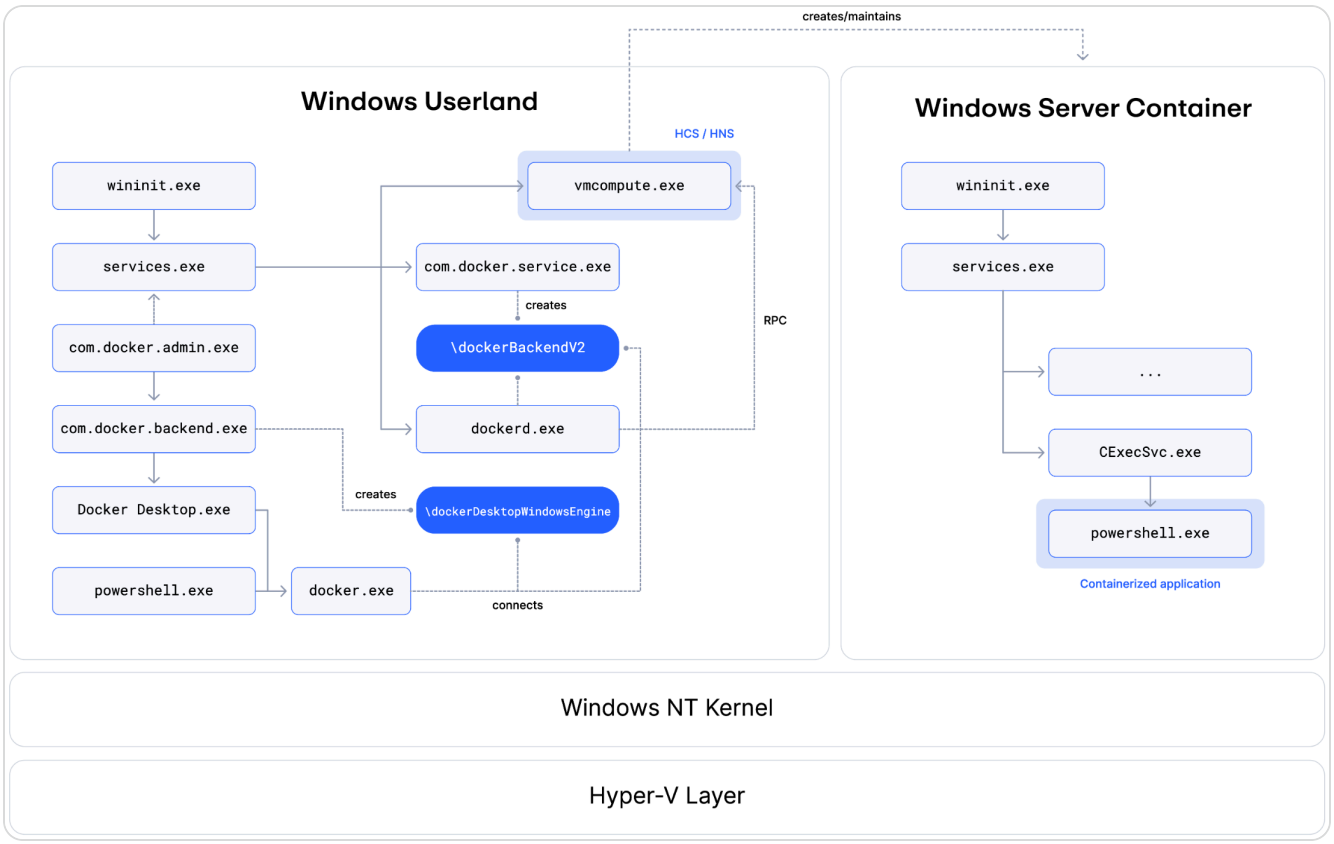

Figure 13 — Simplified system architecture of process-isolated Windows containers.

Since process-isolated Windows containers run directly on the host, the startup is significantly faster but less isolated. Consequently, weakly designed containers or any misconfiguration like bind-mounts of sensitive directives, extended container capabilities, or exposure of the host network stack, can create unintended entry points for attackers. If a container is compromised, it may allow access to the host system and increase the overall attack surface.

Both types of isolation modes support two predefined user contexts, which can be specified using the `-u <user>` flag when launching a container. The `ContainerAdministrator` is the default user context, providing elevated privileges suitable for tasks like installing software or modifying system settings, while the `ContainerUser` is a restricted, non-privileged context that enforces stronger security boundaries.

```
docker run -it -u ContainerUser mcr.microsoft.com/windows/servercore:ltsc2022 powershe
```

Figure 14 — Running a hyperv-isolated Windows container as the `ContainerUser`.

Figure 15 illustrates how the user context affects operations inside a container. Attempts to modify a sensitive Win32 host file fail for the standard `ContainerUser`, while they succeed under the context of `ContainerAdministrator`. However, if a sensitive host directory such as `C:\Windows` is mounted into the container, even `ContainerUser` can modify files within that mounted directory.

```
Add-Content -Path C:\Windows\System32\drivers\etc\hosts -Value "Foo"
```

Figure 15 — Attempts to modify the sensitive Win32 host file within a container.

For security best practices, it is recommended to run Windows containers under the `ContainerUser` context and avoid mounting sensitive host directories. If Windows containers are not required, you can further reduce risk by completely disabling them using the `DISABLEWINDOWSCONTAINERS=1` installer flag.

How networking works with Windows containers

In contrast to Linux containers, where low-level tools like containerd, runc, etc. and the Docker Engine, `dockerd`, itself configure and manage the network stack, Windows Containers utilize low-level Windows APIs, including the Host Compute Service (HCS) and Host Network Service (HNS). When the Windows Engine, `dockerd.exe`, starts for the first time, it automatically creates a NAT-based network called `nat`, which is linked to a Hyper-V virtual switch (vSwitch). All kinds of Windows containers attach to this network by default. HNS is responsible for provisioning the dedicated vSwitch, setting up NAT rules, managing IP address pools, and generating the individual network configurations for each container.

Process-isolated Windows containers inherit the host's firewall rules, which are enhanced with network namespaces, and additional filtering through Azure Virtual Filtering Platform (VFP). However, by default, all outbound traffic is allowed, while inbound traffic is permitted only for specific protocols. Other inbound traffic is blocked unless explicitly allowed. Hyperv–isolated containers, on other hand, are setup very differently. Each one runs its own independent Windows Firewall instance, meaning none of the host's firewall rules apply. As a result, hyperv-isolated containers allow all inbound and outbound traffic. To demonstrate this behavior, the PowerShell snippet shown in Figure 16 must be run with elevated privileges on the host to create a firewall rule that blocks outbound traffic to the domain example.com.

```
# Resolve the IP address
$ipAddress = [System.Net.Dns]::GetHostAddresses("example.com")[0].IPAddressToString
# DENY outbound HTTP/HTTPS to "example.com"
New-NetFirewallRule -DisplayName "Block HTTP/HTTPS to example.com" `
    -Direction Outbound -Protocol TCP -RemotePort 80,443 `
    -RemoteAddress $ipAddress -Action Block
```

Figure 16 — Firewall rule that blocks connections to the domain example.com.

Any process-isolated container started afterward will inherit this rule, while hyperv-isolated containers can still access it. As a result, the following command will return HTTP 200 OK inside a hyperv-isolated container but will be blocked on the host and in any process-isolated container.

```
curl.exe -s -o NUL -w "%{http_code}" https://example.com
```

Figure 17 — Requesting example.com to demonstrate firewall configuration of Windows containers.

When we first worked with Windows containers, we exhibited significant networking issues as soon as connected to our enterprise Virtual Private Network (VPN). The same networking issues appeared to other in-Windows virtualization techniques like Windows Subsystem for Linux 2 (WSL2) or Windows Sandbox, and these issues had been difficult to diagnose. By moving our network infrastructure to a modern approach that provides secure access to private applications without relying on traditional VPNs, these networking problems disappeared.

06

Behind the scenes of the Linux container backend

The Linux backend bootstrapping process made simple

The bootstrapping process of the WSL-based Linux backend largely equals that of the Windows backend, with `com.docker.backend.exe` coordinating and managing the overall startup sequence. In addition, Docker Desktop runs a series of `wsl.exe` commands to initialize and launch the Docker Engine and its supporting components inside both the `docker-desktop` WSL2 distribution and any integrated WSL2 distribution. It is important to note that `docker-desktop` is not intended for general development as its root file system is read-only and updated only with Docker Desktop itself. Any changes inside `docker-desktop` are non-

persistent, so projects must reside on the Windows file system and be bind-mounted into containers or persisted in named volumes. These file system translation tasks add performance overhead, making `docker-desktop` unsuitable for large-scale Linux development.

Following Table 4 lists all processes executed by `com.docker.backend.exe` that remain active until Docker Desktop is shut down. Since the binaries involved offer little usage documentation, their roles can only be inferred from their naming and observed behavior.

Process	Description
`wsl -d docker-desktop -u root -e wsl-bootstrap run --base-image "/path/to/docker-desktop.iso" --cli-iso "/path/to/docker-wsl-cli.iso" --data-disk <ID>`	Initiates the bootstrap process inside docker-desktop WSL2 distribution. The Docker engine, dockerd, and its associated resources like containerd, runc, etc. are loaded from docker-desktop.iso, while the Docker CLI, docker, and its plugins are loaded from docker-wsl-cli.iso
`wsl -d docker-desktop -e /usr/bin/vpnkit-bridge --pid-file=/run/vpnkit-bridge.pid guest`	Runs the vpnkit-bridge background bridge process that enables network communication between containers running in WSL2 and the host
`unshare -muinpf --propagation=unchanged --kill-child /usr/local/bin/wsl-bootstrap jump`	Executed inside docker-desktop to launch wsl-bootstrap jump in a new, isolated Linux environment by creating separate mount, UTS, IPC, network, and PID namespaces, effectively simulating a lightweight container.
`wsl -d <DISTRO> sh`	Runs a shell session inside the default WSL2 distribution.

Table 4 — Sequence of commands to initiate the Linux container backend.

The Hyper-V-based backend employs a different initialization process. In this mode, the privileged helper service `com.docker.service.exe` communicates with the Host Compute Service (HCS) and Host Network Service (HNS) APIs to start the `docker-desktop.iso` disk image as a dedicated Hyper-V virtual machine named `DockerDesktopVM`. This VM hosts the full Linux container engine stack, including `dockerd` and all supporting components.

How engineering excellence delivers seamless Linux containers on Windows

When we started to work with Docker Desktop first, it felt somehow mysterious that running a Windows terminal allows us to work with Linux containers as if they were running natively. But the era in which Linux was dismissed as "cancer" is long behind us. Today, Microsoft Corp. ships a fully featured Linux kernel through the open-source project Windows Subsystem for Linux 2

(WSL2), a remarkable piece of engineering recently praised by Jensen Huang as a "world-class AI PC". Together with modern techniques like ephemeral containers, lightweight utility virtual machines, Windows Named Pipes, Address Family Virtual Socket communication channels, etc., Docker Desktop turns this cross-platform complexity into a smooth, unified developer experience, working with the capabilities of the Windows platform.

Figure 18 provides a top-level view of the WSL-based Linux backend. The Windows components Linux Subsystem Service Manager (Lxss), HCS, and HNS launch the Hyper-V Lightweight Utility VM (LUVM) named WSL, which hosts the Linux kernel under `C:\Program Files\WSL\tools` and runs side-by-side with the Windows host without adding another virtualization layer. Inside this LUVM, the special-purpose Alpine-based WSL2 distribution docker-desktop starts, containing the Go-based executables that orchestrate the startup sequence. The process begins with `wsl-bootstrap run`, which uses the docker-desktop.iso and docker-wsl-cli.iso images. The former provides backend components like dockerd, while the latter includes Docker CLI and plugins. To ensure isolation, Docker Desktop runs `unshare -muinpf`, creating separate mount, process, network, IPC, and UTS namespaces, effectively spinning up a lightweight, ephemeral container with a fully isolated view of system resources.

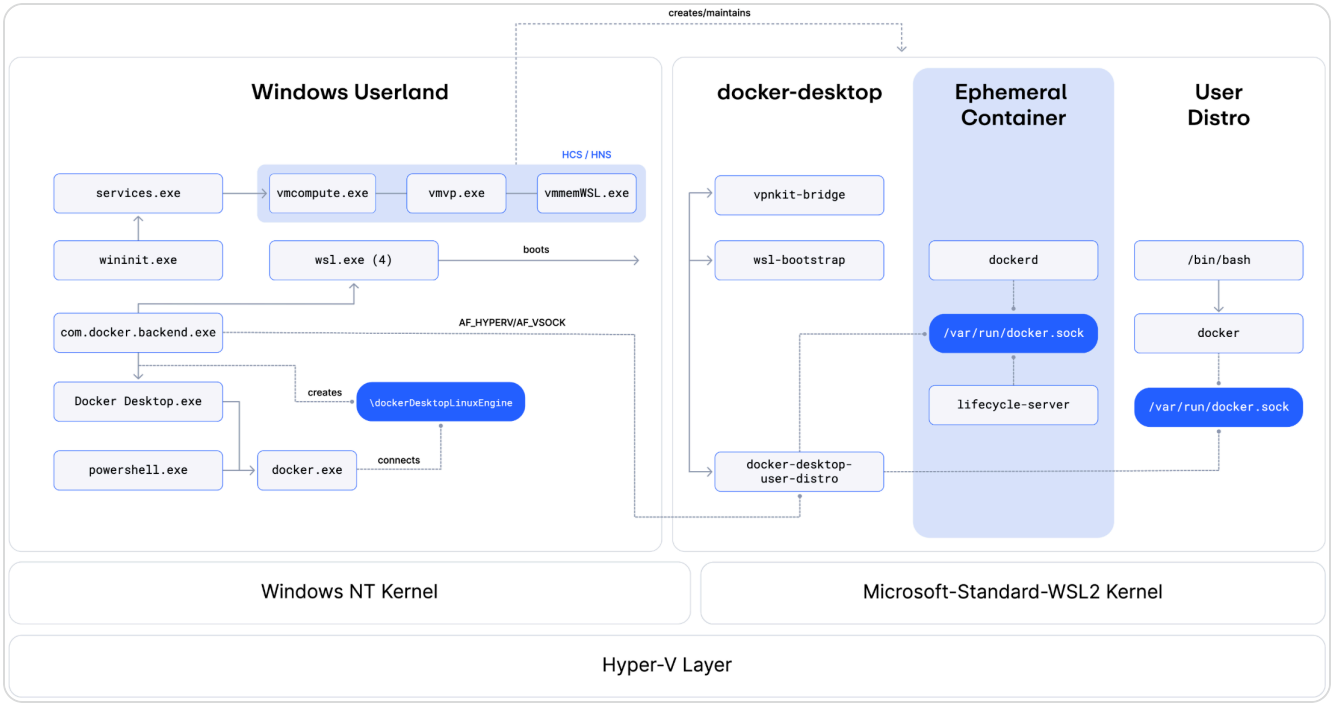

Figure 18 — Simplified top-level system architecture of Docker Desktops WSL-based backend.

When a user runs a Docker command either through the Docker Desktop GUI or a Windows terminal the request is first sent to the Windows named pipe `//./pipe/dockerDesktopLinuxEngine`. The almighty background service

`com.docker.backend.exe` listens on this pipe and processes the request through the internal component Docker API Proxy. From there, the request is forwarded into the docker-desktop WSL2 distribution using AF_VSOCK, a bi-directional communication channel that lets virtual machines and the Windows host exchange data without relying on the host's regular network stack. To aid understanding, Appendix A - A Lightweight AF_Vsock Example implements a Python-based example of a lightweight AF_VSOCK receiver and sender module based on the excellent Medium article Understand Vsock by contributor FrancoisD. Inside the docker-desktop WSL2 distribution, the process docker-desktop-user-distro hands the request to the init process over a Unix domain socket. The lifecycle-server in turn acts as a privileged controller that owns `/var/run/docker.sock` and passes the socket's file descriptor to the Docker daemon, dockerd, ensuring controlled access and secure lifecycle management of the Docker engine. In recent Docker Desktop versions it was merged into the init process; the responsibilities — owning the socket and handing its file descriptor to dockerd — are unchanged.

Furthermore, when a user executes a docker command inside any integrated WSL2 distribution, the standard Unix domain socket `/var/run/docker.sock` is used to forward the request. This socket is a bind-mount from the docker-desktop-user-distro, which forwards requests to the real dockerd running inside the ephemeral container. This arrangement enables seamless communication across different WSL 2 distributions within the WSL2 VM.

It is important to understand that every container, whether launched from Windows or inside a WSL2 distribution, runs inside the LUVM. Since WSL inherits the permissions of the Windows user, even the root capabilities of the Docker Engine cannot escalate privileges on the Windows host unless a vulnerability is present in WSL or Docker Desktop. Docker Desktop strengthens the guest-host security boundary with Enhanced Container Isolation (ECI), which uses the security-focused `sysbox-runc` OCI runtime. ECI runs each container in a Linux user namespace, mapping container-root to an unprivileged user in the LUVM. It also blocks sensitive actions, such as mounting restricted directories or executing certain system calls, providing stronger isolation and protection within the LUVM. Enhanced Container Isolation has no equivalent in the native WSL Containers runtime. An organization relying on the OS runtime alone takes on this isolation gap itself.

06

On the other hand, the top-level system architecture of the Hyper-V-based backend differs as shown in Figure 19. When Docker Desktop was installed without the `ALWAYSRUNSERVICE=1` flag, Windows prompts for elevation since `com.docker.backend.exe` must invoke

`com.docker.admin.exe`, which in turn uses the Windows Service Manager to start the privileged `com.docker.service.exe` process. Once running, this service creates and listens on the named pipe `//./pipe/dockerBackendV2`, which acts as the secure control channel through which Docker Desktop performs privileged operations.

The `DockerDesktopVM` is then started from the `docker-desktop.iso` disk image. This ISO is built entirely using the LinuxKit toolkit and provides a minimal Linux environment containing `dockerd` and all required subsystems. LinuxKit services inside the VM handle Docker's lifecycle management, health monitoring, and diagnostic collection, ensuring that the Docker Engine can be started, supervised, and recovered reliably.

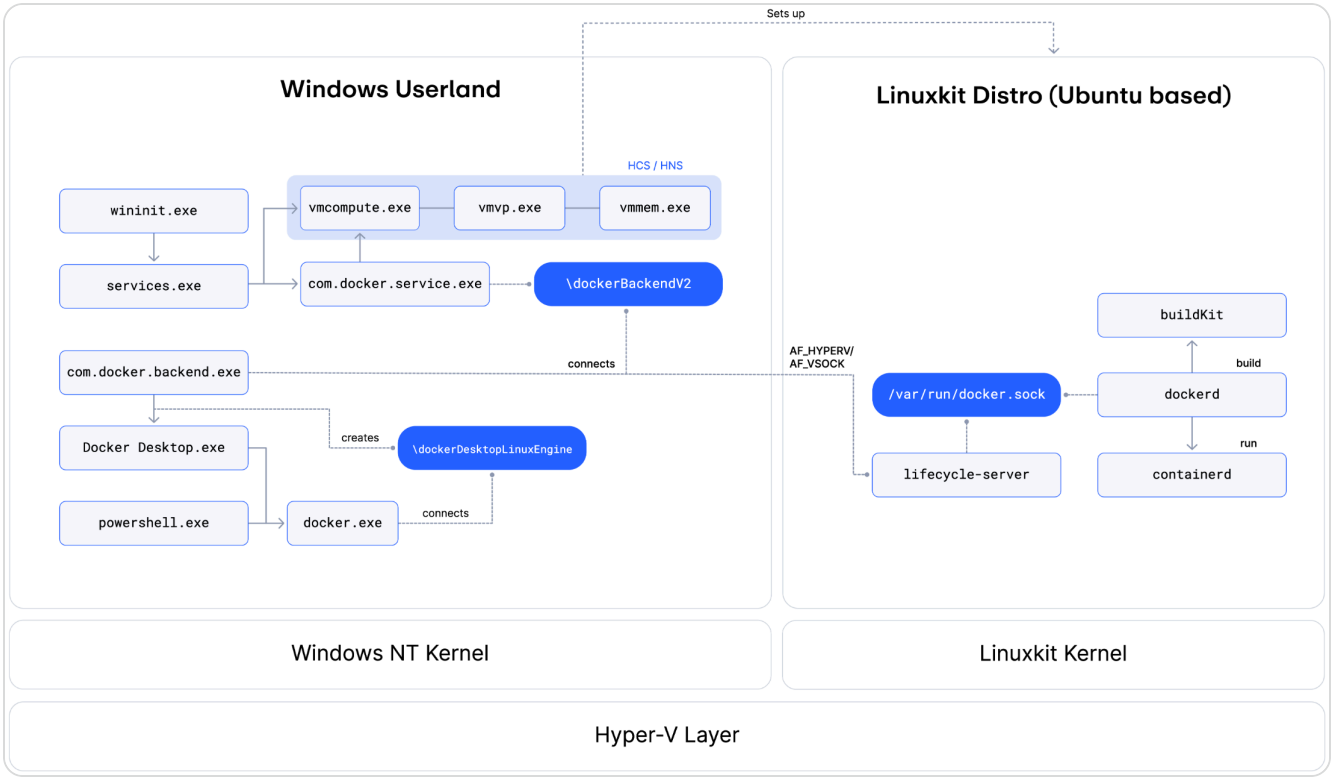

Figure 19 — Simplified top-level system architecture of the Hyper-V based Linux backend of Docker Desktop.

When a user runs a Docker command on the Windows host via the GUI or any terminal, the request is first sent to the named pipe `//./pipe/dockerDesktopLinuxEngine`, where it is processed by internal components of `com.docker.backend.exe`. The process then forwards the request to the DockerDesktopVM using an Address Family Hyper-V Socket, AF_HYPERV, a high-performance communication channel that allows the Windows host to interact with the Docker Engine inside the virtual machine without relying on traditional networking. To foster a deeper understanding about this virtual communication channel we have implemented a lightweight hello world example.

In contrast to the WSL-based backend, Hyper-V provides a stronger security boundary because `DockerDesktopVM` and Windows host are isolated on hardware-level with only a few integration points. The one exception is the central role of privileged helper process `com.docker.service.exe` for the Hyper-V-based backend. This makes it a potential escalation vector if the service or its communication channel were ever compromised. Docker Inc. has designed this component with security in mind and remains committed to promptly addressing any vulnerabilities that may arise. This is a shared consideration: any privileged helper, Docker's or the operating system's, concentrates risk. The relevant question is which vendor commits to addressing vulnerabilities under a support agreement.

06

How Docker Desktop facilitates networking with minimal environmental interference

If you want a single reason Docker Desktop earns its place over the raw WSL stack, it is here. Docker Desktop does not use the Windows or WSL networking stack at all. It ships its own networking components, with `com.docker.backend.exe` at the center, which is why container traffic keeps flowing behind a corporate VPN, proxy, or firewall while plain curl or wget inside WSL2 stalls. The rest of this section explains how.

It felt strange that we were struggling with such massive networking issues with Windows containers and even basic tools like curl or wget inside WSL2, while all networking operations with Docker Desktop, even those running inside containers, worked flawlessly. When we delved deeper into Docker Desktop's system architecture, and especially when we read David Scott's excellent article How Docker Desktop Networking Works Under the Hood, the picture suddenly became clear. Docker Desktop does not rely on the Windows or WSL networking stack at all. Instead, Docker Desktop ships its own carefully engineered networking components, with the almighty `com.docker.backend.exe` at the center, or simplifies operations such as hardcoding certain DNS inside the Win32 host file. In David Scott's article this transport is described as a separate vpnkit-bridge process. In recent Docker Desktop versions — following the networking stack overhaul that replaced vpnkit with the gVisor netstack in Docker Desktop 4.19 — that functionality was merged into the init process. The data flow is unchanged; only the process boundary moved.

So, how does TCP/IP traffic from a container reach the outside world? Under the hood, the Docker Engine first creates a bridge network called `docker0`, inside the WSL2 distribution respectively within the DockerDesktopVM. Every Linux container launched in this environment connects its `eth0` interface to this bridge and receives an internal IP address from the docker0 subnet. The bridge interface is linked to a TAP device named eth0, which usually has the internal IP address 192.168.65.3. The TAP device is managed by the docker-desktop distribution, respectively DockerDesktopVM. As David points out, workstations in enterprise environments often face restrictions that can prevent network traffic from a Linux VM from being routed successfully. To address this, packets are transmitted to the Windows host at user-level using shared memory communication over the AF_VSOCK protocol by the init process, which is a component of the `com.docker.backend.exe`. It provides key networking services, including low-level TCP/IP handling, a high-level DNS server, and an HTTP proxy. Then, com.docker.backend.exe decodes and forwards network traffic to the Windows networking stack using Win32 APIs. Packets addressed to internal destinations are sent directly to the target server, while external traffic is routed through a system proxy. By default, Docker Desktop automatically detects this proxy, but users can manually configure it via Settings > Resources > Proxies.

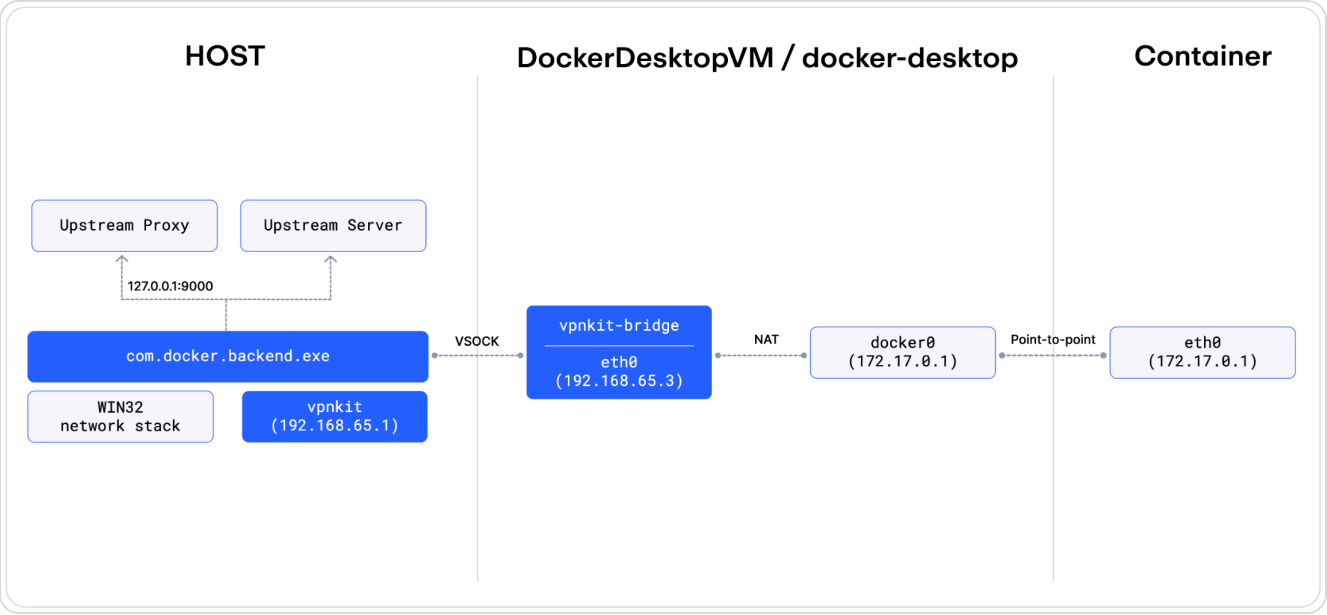

Figure 20 — Simplified TCP/IP flow with Docker Desktop.

As a result, from the Windows host's perspective, all container traffic appears to come from `com.docker.backend.exe`. If the host firewall permits this process to access the network, containers can seamlessly communicate with external networks, including the internet.

DNS requests from containers are handled through two complementary mechanisms. First, the Docker Engine provides a built-in DNS server that resolves the names of containers within the same compose network. This enables services to communicate using human-readable service names such as `nginx`, `mongo`, or `postgres`, instead of relying on dynamically assigned container IP addresses.

All other DNS queries are passed on to a custom DNS server, that selects between two DNS servers running on the host, depending on the domain being queried. The special domain `docker.internal` includes the hostname `host.docker.internal`, which is hardcoded in the Win32 hosts file and always resolves to the correct IP address of the Windows host. Hence, the special host.docker.internal lets containers reach host services reliably without hard-coding host IP addresses.

All remaining queries are resolved using the host's standard OS libraries. This ensures that any domain reachable in the developer's web browser, whether behind a corporate VPN or on the public Internet, resolves identically inside containers. This design provides predictable, consistent name resolution across both internal enterprise networks and external systems.

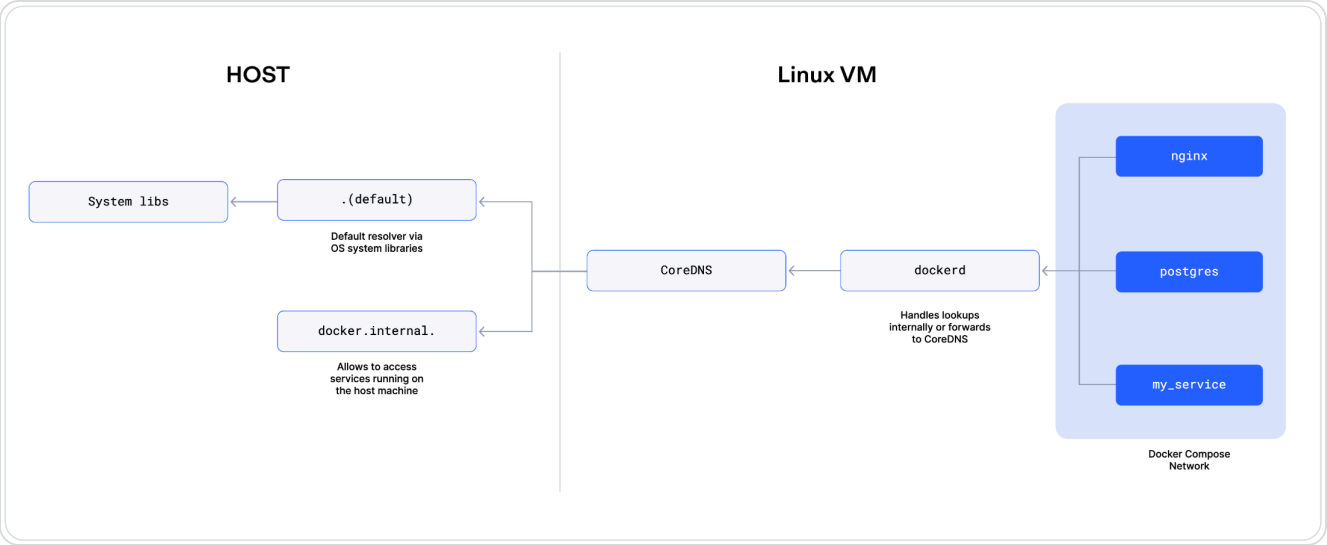

Figure 21 — DNS flow originating from containers running inside the Linux VM.

Docker Desktop vs. native WSL Containers comparison table

What the operating system now provides: It is worth being precise about what WSL Containers actually delivers, because conceding the overlap is the only honest way to argue the rest. Microsoft's runtime runs Linux containers alongside Windows containers in a shared lightweight VM, with startup times measured in fractions of a second, a unified network namespace that lets containers reach `localhost` without port-forwarding workarounds, and pass-through for GPU and NPU hardware. For a developer with no compliance obligations, that covers a real share of day-to-day work. The question this paper answers is what sits above that runtime, not whether the runtime is good.

Build 2026's WSL Containers handle the runtime. They do not address the management, identity, and isolation controls that enterprises depend on. The table below maps where the two overlap and where Docker Desktop remains the only option.

Capability	**Native WSL Containers** wslc.exe / containerd	**Docker Desktop** Business
Linux containers on Windows	✓ Yes	✓ Yes
Sub-second container startup	✓ Yes	✓ Yes (WSL 2 backend)
VPN / proxy-transparent networking	Inherits WSL stack	✓ Own engineered stack
Centralized policy enforcement (Admin Console)	— No	✓ Yes
Single Sign-On (SSO) / SCIM	— No	✓ Yes
Organization Access Tokens	— No	✓ Yes
Enhanced Container Isolation (sysbox-runc)	— No	✓ Yes
Image / Registry Access Management	— No	✓ Yes
Hardened Docker Desktop mode	— No	✓ Yes
Commercial support	— No	✓ 1 business day, 24×5 (Premium: faster SLAs, 24×7)

Table 5 — Docker Desktop (Business) versus native WSL Containers.

08

Conclusion

Running containers in an enterprise, where compliance and security are not optional, takes specialized expertise and coordination across teams. Docker Desktop's paid tiers lower those barriers with controls that the operating system's native runtime does not provide: centralized policy enforcement, SSO and SCIM, Organization Access Tokens, Enhanced Container Isolation, and networking that survives corporate VPNs and proxies without manual configuration.

Now that Windows ships its own container runtime, the decision comes down to what an organization needs above the runtime. For a regulated enterprise that has to enforce identity, restrict registries, isolate workloads, and call someone when it breaks, Docker Desktop is the

clear choice. For a solo developer with no compliance requirements, the native WSL runtime may be enough. This paper exists so that call is made on the architecture, not on price alone.

Working with containers in enterprise environments, where compliance and security are not negotiable, can be challenging, often requiring specialized expertise and close collaboration across multiple departments. From our experience, Docker Desktop, along with its paid subscriptions, provides a range of enterprise-grade security controls that significantly lower these barriers, making it easier to run container workloads on local developer workstations. Beyond security, Docker Desktop integrates smoothly into enterprise infrastructure, offering seamless developer experience without the need for manual local configuration.

With this whitepaper, we aim to simplify discussions with internal security teams and provide a clear understanding of the underlying system architecture. It also provides a solid foundation with technical facts for conversations with developers who usually prefer to use Docker CE inside WSL instead of Docker Desktop. We hope you share our view that Docker Desktop reflects true engineering excellence in both design and security.

Our internal developer community shows strong interest in the recent AI features of Docker, and we aim to support this momentum with practical, hands-on documentation. So, looking ahead, we plan to publish the next whitepaper on a clear guidance for running AI-based workloads on the "world-class AI PC", leveraging Model Runner, MCP Toolkit, and the capabilities of Windows via the Windows Subsystem for Linux 2 (WSL2).

APPENDIX A

A lightweight Windows named pipe example

Figure 22 implements a lightweight pipe server that creates and maintains a Windows named pipe accessible only to members of the local `PIPE-USERS` group.

```
import win32pipe, win32file, win32security
PIPE_NAME = r"\\.\pipe\MySecurePipe"

def create_secure_sd():
    sd = win32security.SECURITY_DESCRIPTOR()
    sd.Initialize()
    # Look up SID for local group "PIPE-USERS"
    pipe_users_sid, _, _ = win32security.LookupAccountName(None, "PIPE-USERS")
    # Grant PIPE-USERS read/write
    acl = win32security.ACL()
    acl.AddAccessAllowedAce(win32security.ACL_REVISION,
        win32file.FILE_GENERIC_READ | win32file.FILE_GENERIC_WRITE, pipe_users_sid)
    sd.SetSecurityDescriptorDacl(True, acl, False)
    return sd

def pipe_server():
    sa = win32security.SECURITY_ATTRIBUTES()
    sa.SetSecurityDescriptor(create_secure_sd())
    pipe = win32pipe.CreateNamedPipe(PIPE_NAME, win32pipe.PIPE_ACCESS_DUPLEX,
        win32pipe.PIPE_TYPE_MESSAGE | win32pipe.PIPE_READMODE_MESSAGE | win32pipe.PIPE
        1, 65536, 65536, 0, sa)
    win32pipe.ConnectNamedPipe(pipe, None)
    _, data = win32file.ReadFile(pipe, 64 * 1024)
    print("[SERVER] Received:", data.decode())
    win32file.CloseHandle(pipe)

if __name__ == "__main__":
    pipe_server()
```

Figure 22 — A lightweight pipe server that creates and maintains a protected Windows named pipe.

Figure 23 implements a simple client that sends messages through this pipe. This works only if the local user is a member of `PIPE-USERS`; otherwise a permission error occurs.

```
import win32file, time
PIPE_NAME = r"\\.\pipe\MySecurePipe"

def client():
    while True:
        try:
            h = win32file.CreateFile(PIPE_NAME, win32file.GENERIC_WRITE, 0, None,
                                     win32file.OPEN_EXISTING, 0, None)
            break
        except:
            print("[CLIENT] Waiting for server...")
            time.sleep(1)
    while True:
        msg = input("Message ('exit' quits): ")
        if msg.lower() == "exit": break
        win32file.WriteFile(h, msg.encode())
    win32file.CloseHandle(h)

if __name__ == "__main__":
    client()
```

Figure 23 — A lightweight pipe client that sends data through the named pipe.

To run the sample, ensure Python 3.x is installed and that `pywin32` can be installed via pip (Figure 24), then create the local group that controls access (Figure 25).

```
pip install pywin32
```

Figure 24 — Installing the `pywin32` module.

```
net localgroup PIPE-USERS /add
```

Figure 25 — Creating the local `PIPE-USERS` group.

APPENDIX B

A lightweight AF_VSOCK example

Following sample implements a lightweight AF_VSOCK sender and receiver pattern using Python3 and QEMU on an Ubuntu 24.04 flavor based on the Medium article *Understand Vsock* by FrancoisD.

First, ensure that the underlying kernel is capable of both Nested Virtualization and that it supports AF_VSock.

```
kvm-ok && zcat /proc/config.gz | grep CONFIG_VSOCKETS=y
```

Figure 26 — Verifying if the Linux kernel provides support for Nested Virtualization and AF_VSock.

Next, install QEMU via Ubuntu's Advanced Package Tool (APT).

```
sudo apt update && sudo apt install -y qemu-system-x64 qemu-utils
```

Figure 27 — Installing QEMU and its associated utilities via apt.

Next, download the "netinst" Debian image that is the most reliable version for kernel-level testing like AF_VSOCK:

```
wget https://cdimage.debian.org/cdimage/daily-builds/daily/current/amd64/iso-cd/debian
```

Figure 28 — Downloading the netinst Debian ISO image.

Next, create a hard blank Debian-based disk image.

```
qemu-img create -f qcow2 debian-vsock.qcow2 10G
```

Figure 29 — Creating a blank Debian based ISO image.

Next, use this command to start the initial Debian installation with VSOCK support enabled:

```
sudo qemu-system-x86_64 -enable-kvm   -m 2G   -vga virtio -display gtk   -drive file=d
  -device vhost-vsock-pci,guest-cid=123
  -nic user,model=virtio-net-pci
  -boot d
```

Figure 30 — Initialize the Debian installation.

Log into the Debian VM using the following command and ensure to use 123 as Context Identifier (CID):

```
sudo qemu-system-x86_64 -enable-kvm   -m 1G -vga virtio -display gtk   -drive file=deb
```

Figure 31 — Booting the Debian VM with a dedicated vsock device and identifier.

At following, a lightweight AF_VSock receiver component that must run inside the Debian VM — and the sender component that must run on the host.

```
import socket
CID, PORT = socket.VMADDR_CID_HOST, 9999
s = socket.socket(socket.AF_VSOCK, socket.SOCK_STREAM)
s.bind((CID, PORT))
s.listen()
conn, (rcid, rport) = s.accept()
print(f"Connection opened by cid={rcid} port={rport}")
while True:
    buf = conn.recv(64)
    if not buf: break
    print(f"Received bytes: {buf}")
```

Figure 32 — Lightweight Python sample that implements an AF_VSOCK receiver.

```
import socket
CID = socket.VMADDR_CID_HOST
PORT = 9999

def vsock_cli():
    with socket.socket(socket.AF_VSOCK,
                       socket.SOCK_STREAM) as s:
        s.connect((CID, PORT))
        while True:
            msg = input("> ")
            if msg.lower() == "exit": break
            s.sendall(msg.encode())
            print(s.recv(1024).decode())

vsock_cli()
```

Figure 33 — Lightweight Python sample that implements an AF_VSOCK sender.

References

1. Huang, Jensen (2025). *NVIDIA CEO Jensen Huang Keynote at CES 2025*.
2. CyberArk Software Ltd. (2022). *Understanding Windows Containers Communication*.
3. Scott, David (2022). *How Docker Desktop Networking Works Under the Hood*. Docker.
4. Madhavapeddy, Anil; Scott, David J.; Cormack, Justin. *A Decade of Docker Containers*. Communications of the ACM.
5. FrancoisD (2023). *Understanding Vsock*. Medium.
6. CyberArk Software Ltd. (2023). *Breaking Docker Named Pipes: Docker Desktop Privilege Escalation — Part 1*.
7. Pravtchev, Valentin; Pravtchev, Julius; et al. (2024). *Using Docker Desktop in Large-Scale Enterprises*.
8. Pravtchev, Valentin; Pravtchev, Julius (2026). *Hello-world AF_HYPERV sample*. GitHub.