

For the past decade, Docker has provided a robust solution for building, shipping, and sharing applications. But behind its simple “build and run” workflow lie years of complex technical challenges.

BY ANIL MADHAVAPEDDY, DAVID J. SCOTT, AND JUSTIN CORMACK

A Decade of Docker Containers

DOCKER IS A widely used developer tool that first simplifies the assembly of an application stack (`docker build`), then allows for the rapid distribution of the resulting executables and data (`docker push`), and subsequently supports running multiple applications isolated from one another on the same machine (`docker run`). Developers can compile their own Docker images via a single Dockerfile alongside their source code, and reuse other published images to share packaging efforts across the global set of programming languages and application stacks.

Since Docker's first release in 2013,¹⁴ it has seen rapid adoption in diverse sectors, from stellarator simulations at Proxima Fusion to streaming services

at Netflix to deploying software in space with BalenaOS. It is also something developers seem to *enjoy* using, and is consistently at the top of Stack Overflow's community rankings as the “most desired” and “most used” developer tool.^a The Docker Hub, just one of several registries where images can be shared, hosts more than 14 million application images and delivers more than 11 billion image pulls per month.

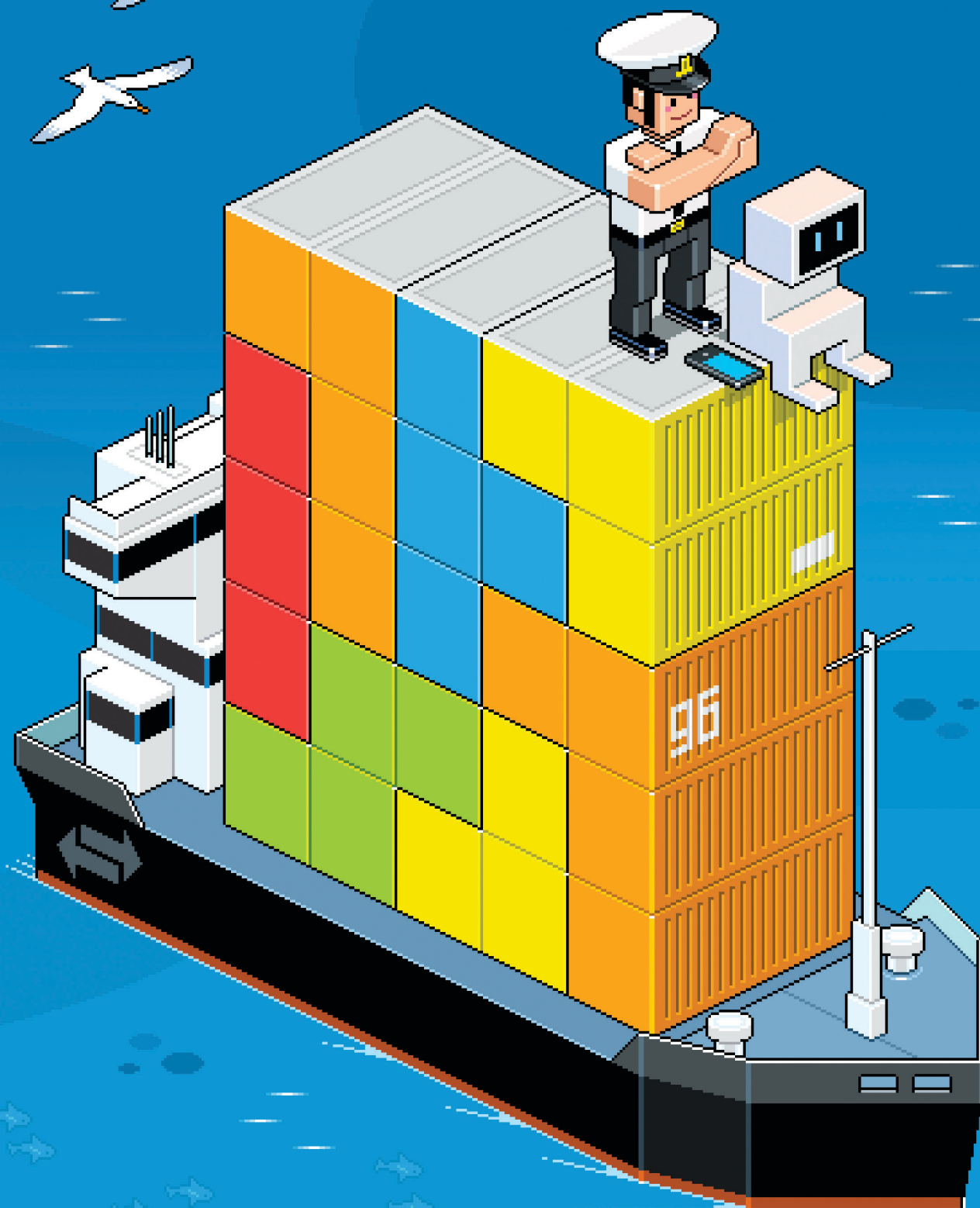
Docker's popularity stems from its tackling a longstanding problem many developers face: how to develop and deploy microservices that are increasingly written in diverse languages.²⁶ It has become the de facto standard for how we manage cloud-native applications on multi-tenant platforms such as Kubernetes,⁵ and has set a higher (but not yet perfect) bar for reproducible scientific research.⁴

But what *is* Docker, when we look behind the seemingly simple command line interface? It is a system that builds on decades of advances across the operating system stack, evolving

a <https://www.docker.com/blog/docker-stack-overflow-survey-thank-you-2023/>

» key insights

- Just as shipping containers standardized how goods are transported, Docker packages applications with everything they need to run (code, data, configuration) into portable “containers” that work anywhere.
- Linux namespaces enable Docker to isolate applications without heavyweight virtual machines by remapping resource access (filesystems, networks, processes) at the kernel level to give each container its own view of system resources while maintaining native performance.
- Docker supported macOS and Windows by embedding Linux within the desktop applications via library virtual machine monitors rather than running in the host operating system.
- Docker repurposed SLIRP, a 1990s dial-up tool originally for Palm Pilots, to avoid triggering corporate firewall restrictions by translating container network traffic through host system calls instead of network bridging.



since its original release to incorporate much systems research in its quest to provide a frictionless developer experience. In this article, we explain the technical foundations of Docker, beginning with its origins on Linux, and subsequently how we rebuilt it to work on macOS and Windows without com-

promising ease of use. Today, developer workflows are evolving rapidly with AI-driven workloads, so we also discuss the future of Docker as it adapts to support heterogeneous hardware such as GPGPUs and FPGAs.

Technical Origins

In the early 2000s, it was common practice to manually install a Linux distribution with myriad dependencies, and hand-compile and configure a batch of software to run on a new machine.¹¹ By 2010, this process had become even more complex with the rise of cloud computing, where applications were expected to run on multiple virtual machines across hosts with varying resource requirements.¹³ Docker simplified this process by empowering developers to package their application and all its dependencies into a series of filesystem images, or “containers,” that could be run on any machine with just Docker installed. And unlike the virtual machine experience (which involved installing an entire operating system), it needed just a few commands to get up and running.

A typical workflow. A developer using Docker writes a Dockerfile that describes how to build their application using a familiar shell syntax that is extended to make it a step-by-step process. For example, a Python-based website might have a Dockerfile that looks like this:

```
FROM python:3
COPY requirements.txt /app/
requirements.txt
WORKDIR /app
RUN pip install -r
requirements.txt
COPY . /app
EXPOSE 80
CMD ["python", "app.py"]
```

The developer then runs `docker build` to create a Docker container image by executing this Dockerfile. This image can then be pushed to Docker Hub, which acts as a central registry of images:

```
$ docker build -t avsm/my-
python-app .
$ docker push avsm/my-python-
app
```

The image can be downloaded and run on any machine with Docker in-

stalled. For instance, to run the image with a local data volume mounted and with a single network port exposed to the host, the developer would tell their users to run:

```
docker run -v data:/app/data -p
80:80 avsm/my-python-app
```

The application is now running in a container, isolated from the host system and any other containers running on the same machine. The developer can iterate on their application, and when they are ready to release a new version, they rebuild the image and push it to Docker Hub. Users can update images they are using independently of one another and without needing to worry about conflicts between different versions of the same software.

The Docker command line interface has evolved over the years to incorporate many more commands, and the back-end systems it uses have been entirely rearchitected, but the original workflow of writing a Dockerfile and using `docker build` and `docker run` has remained consistent since 2013. A search on GitHub finds more than 3.4 million Dockerfiles in the root of public repositories hosted there, showing just how popular this distribution mechanism is across almost any type of software project.¹⁷

Under the hood. Let’s now understand how Docker containers work at a lower level within Linux. Operating system kernels isolate process memory spaces from one another, but deliberately share several other types of system resources.¹² The OS kernel boots from a single shared filesystem that contains configuration files, dynamic libraries, and per-application state. While convenient, this single shared filesystem makes it very difficult to install *multiple* applications at the same time if they have conflicting dynamic library requirements. Processes also need to communicate with each other; for example, a Web front end needs to communicate with a back-end database. Linux supports multiple inter-process communication methods including networking, Unix signals, and Unix domain sockets.³⁴ Although such shared channels are essential for co-operating processes, the sharing can also lead to undesired *interference* if

Glossary

Container: A standalone executable software image that includes everything needed to run an application, including code, libraries, and dependencies.

Image: A read-only filesystem template used to create containers that contains the application code and required dependencies in a standardized (“OCI image”) on-disk format.

Docker engine: The core server-side component of Docker, responsible for building, running, and managing containers on a host system.

Dockerfile: A text file that contains a sequence of instructions to build an OCI image. It defines the steps, such as the base image to be used, and the commands for installing dependencies and copying files into the image.

Registry: A centralized repository used to store and distribute Docker images. The most common public registry is Docker Hub, but the protocol is standardized and most cloud operators run their own.

Volume: A mechanism for persisting data used by containers. Volumes are stored outside the container filesystem to ensure data is not lost when a container is removed.

Orchestration: The automated management of containerized applications, including scaling, deployment, and networking. Tools like Kubernetes or Nomad are often used for this purpose.

Namespace: A feature in the Linux kernel that provides isolated environments for processes at level of named resources, providing lightweight isolation while sharing the same underlying kernel.

Layer: A component of a Docker image that represents a change made to the filesystem. Images are built as a composition of filesystem layers to enable reusability and reduce storage duplication.

Bridge network: The default network type in Docker that allows containers on the same host to communicate with related services, while isolating them from external networks.

there are clashes *between* applications, for example, in the choice of network ports.

One approach to solving these clashes is to run each application in its own individual virtual machine (VM), using a separate guest kernel, userspace, and filesystem. Hypervisors multiplex many VMs on shared hardware.¹ This is effective but heavyweight: It requires multiple kernels, duplicate filesystems, duplicate caches, and bridged network interfaces and introduces significant complexity if the user just wants to run a couple of applications quickly. It is also hard to de-duplicate storage and memory efficiently because each guest OS acts independently by assuming it is the hardware's only user.³⁶

These challenges raised a question: Could we use OS primitives rather than heavyweight VMs? In 1978, Unix v7 added `chroot()` to allow processes to use entirely separate root filesystems, but did not support composing multiple filesystems arising from different applications. Systems such as Nix⁸ and Guix⁶ require software to be repackaged into distinct per-application directories, and use dynamic linking to resolve the right library versions. While effective, this requires all software packaging to be modified, which is not always possible for proprietary software. It is also only a partial solution, as it does not solve the problem of network port clashes between applications.

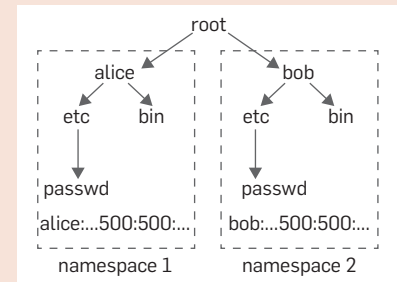
Docker instead opted to use a feature of Linux called *namespaces*,³⁸ which wasn't available when Nix was first created. Namespaces give each process more control over how to access shared resources such as files and directories. For example, in Figure 1, in a root filesystem containing `/alice/etc/passwd` and `/bob/etc/passwd`, two processes under different namespaces, could see `/etc/passwd` differently and resolve to the version under either `/alice` or `/bob`. The process itself has no idea that its requests are being remapped into the wider root filesystem, and it can never “see” files outside its scope. Crucially, the namespacing applies only when *opening* a resource, and the resulting file descriptor operates as a normal kernel resource for subsequent operations, such as reading or writing, without fur-

ther overhead. This allows the Linux kernel to manage the shared resources efficiently, while still providing the level of isolation that the application needs from the underlying filesystem. Once opened, the file descriptors can also be passed across processes in the usual way, ensuring compatibility with Unix programming norms.

Namespacing is not a modern feature in Linux, having been added incrementally over the years. Filesystem (or “mount”) namespacing was first added in 2001 to the Linux 2.5.2 kernel,³⁸ followed by interprocess communication namespacing in 2006 in Linux 2.6.19,¹⁶ and then the network stack in 2007 in Linux 2.6.24.³ Over the years, Linux has accumulated support for seven distinct types of namespaces,²⁷ which when used together permit tremendous flexibility in how a single kernel can assign resources to processes with minimal overhead. However, as they were introduced piecemeal rather than the OS being designed for them—as with Plan 9²⁸—these namespaces were low-level and difficult to use. Variations on the theme in other operating systems such as FreeBSD¹⁵ and Solaris³⁰ also never broke through to popular use. The major advance that Docker made in 2013 was therefore to use namespaces to find the pragmatic balance between heavyweight isolation, as offered by VMs, and ease of use and compatibility with existing software, as offered by OS primitives. Next, we will look at how this works.

How Docker runs Linux containers. Docker is a client-server application, with a server daemon (`dockerd`)

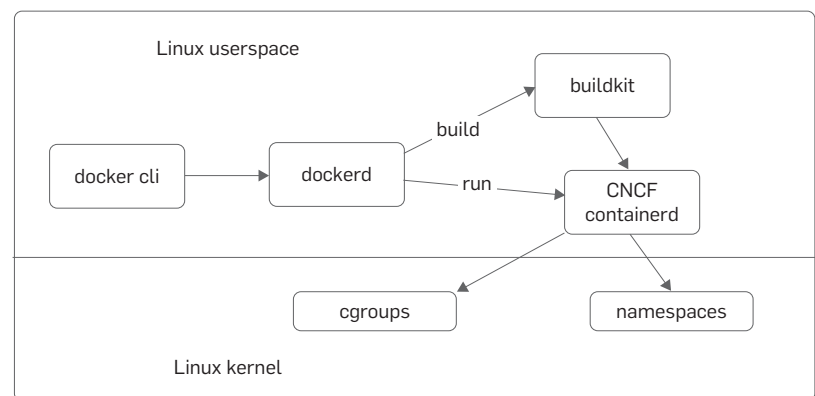
Figure 1. Linux mount namespaces allow processes to control how filenames are resolved.



that runs on the host machine and a docker CLI client that sends requests via a RESTful Docker API. The daemon creates and manages all the system resources, such as containers, images, networks, and volumes. When the developer invokes a docker CLI command, it sends API calls via a well-known Unix domain socket. While the daemon used to be a monolithic program, in around 2015 we split it up into the specialized components⁷ shown in Figure 2. The first component, `buildkit`, assembles filesystem images, and then the `containerd` manages the instantiation of those images into running containers with associated network and storage resources.

Container images. When `docker build` is invoked, Docker builds a filesystem image that represents the executables and data from the input Dockerfile. The container images are stored in a layered filesystem format, where each layer is applied on top of the previous layer. The bottom of these layers is usually bootstrapped from an

Figure 2. The Docker component architecture.



operating system distribution such as Debian or Alpine Linux, but can also be hand-assembled via a simple tar archive. The subsequent layers then correspond to the filesystem differences resulting from the execution of individual commands in a Dockerfile. This is the basis for Docker Hub's ability to share images across the Internet. The image format itself has been standardized since 2016 by a community of users in the Open Container Initiative (OCI),^b with multiple independent implementations now available.

The images themselves are stored in a content-addressable storage system where the hash of the filesystem image is used as the key to manage it. This allows efficient deduplication of storage, and ensures that the image is immutable once it has been pushed to Docker Hub. The image can be pulled by any user and run on any machine, and the hash can be used to verify that the image has not been tampered with. Docker uses modern Linux filesystems such as overlayfs, btrfs, or ZFS to manage the copy-on-write layers directly with efficient snapshotting and cloning. Docker also supports lazy-pulling of images via the stargz storage snapshotter.^c

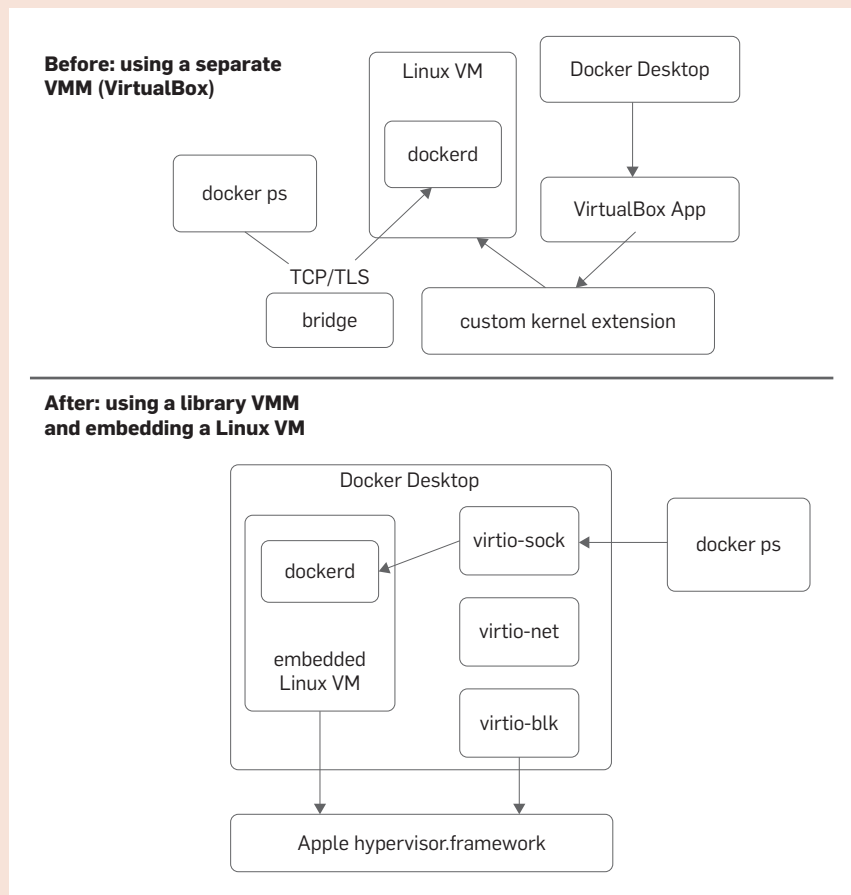
Container instances. Calling `docker run` on an OCI image results in the allocation of system resources to create a namespace-isolated process (or "container") that is bootstrapped from the filesystem images. The `containerd` process is responsible for dynamically configuring the namespaces required for each container, performing tasks such as:

- Defining process "control groups" for resource isolation and I/O rate limits
- Remapping local network ports within the container to those exposed externally on the host interfaces
- Attaching mutable storage volumes from the host filesystem for persistent application state
- Isolating the process tree of the container with PID namespaces
- Mapping the container's local user IDs to different ones on the host with

^b <https://github.com/opencontainers/image-spec>

^c <https://github.com/containerd/stargz-snapshotter>

Figure 3. Using a traditional standalone hypervisor (top) versus the Docker approach of using a library virtual machine monitor and embedding a Linux VM (bottom).



user namespaces so that, for example, the `avsm` user might always consistently appear as UID 1000 within the container but actually be mapped to non-interfering UIDs 12345 or 23456 on different hosts.

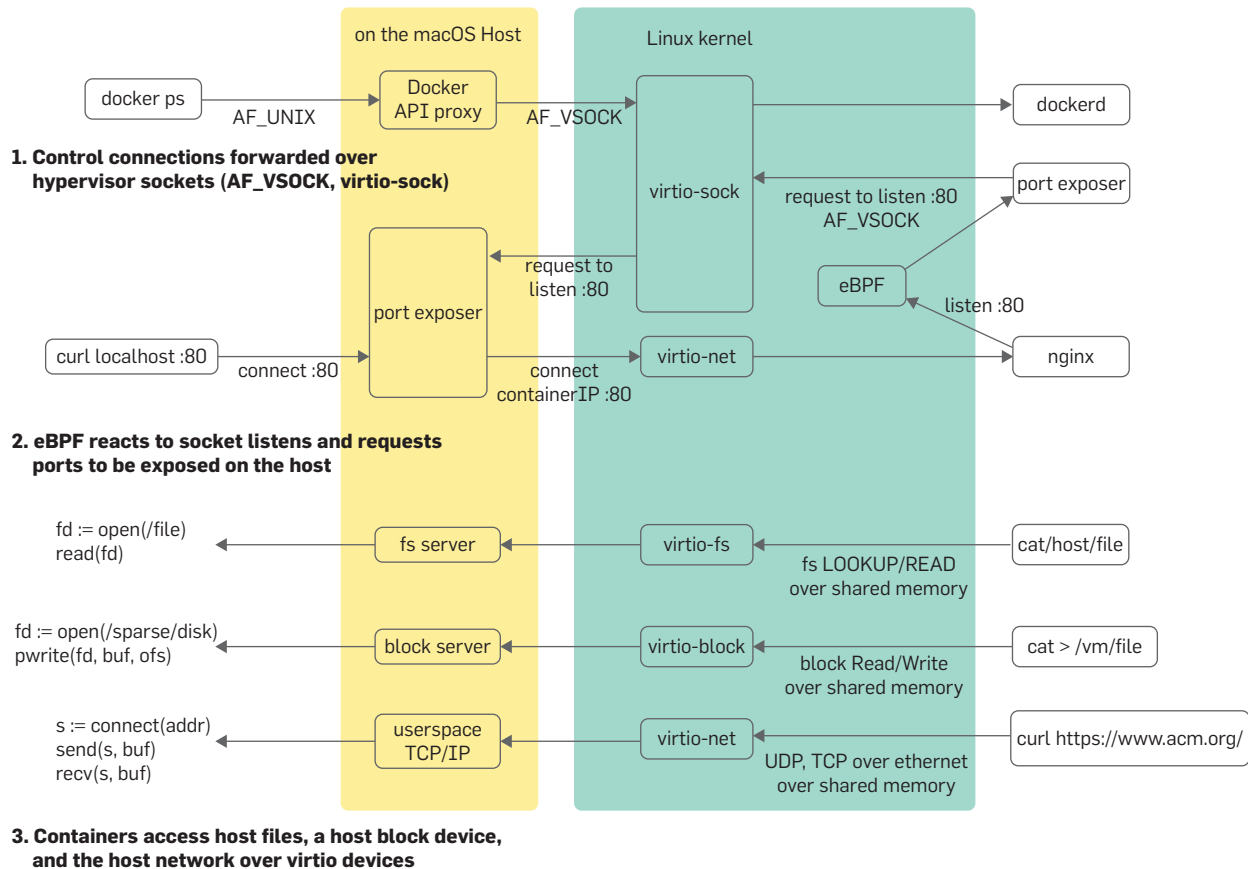
While there is some overhead involved in the construction of these namespaces, it is far lower than the spawning of a full Linux VM²⁰ and can be done in a fraction of a second in most cases. The Linux kernel itself garbage collects containers that have exited, just as with normal processes.

Evolving Beyond Linux

This client-server architecture made it easy to manage remote Docker instances, since the CLI could just be configured to send its commands over a secure network connection, for example, to Docker hosts running on the cloud. In 2015, we took advantage of this flexibility to solve another pressing problem resulting from the growing reach of Docker. In the two years af-

ter its launch, Docker had established itself as a widely adopted tool for Linux development, but it hit a usability wall. The majority of developers were still using macOS or Windows as their primary development environment, but the Docker filesystem images could run only on a Linux kernel. Meanwhile, the rise of the public cloud led Linux to become the preferred choice for deployment. We quickly needed to find a way to make Linux containers run on macOS and Windows to remove a huge barrier to developing cloud services.

Building a seamless Docker for a Mac application. The key constraint when designing Docker for Mac and Windows is that it had to work without additional configuration for developers already familiar with the Linux version, and also be able to run the same Docker images. The answer lay in combining the latest in hypervisor virtualization with the best of Linux namespaces. Instead of the conventional approach of running Linux

Figure 4. The Docker for Mac application architecture.

alongside the desktop OS, we inverted the software architecture by embedding the hypervisor *within* a userspace application running on macOS or Windows, and ran Linux inside that application instead. The inspiration for this approach came from our research into unikernels,²² which had shown that it was possible to flexibly embed operating system components within a larger application.

Embedding Linux in an application. We first designed a library virtual machine monitor (VMM) called HyperKit that used hardware virtualization extensions in Intel CPUs to run a Linux kernel in a normal user process¹⁹ (Figure 3). This embedded Linux kernel would then run the Docker daemon, which in turn would run the containers and act as a normal Docker server endpoint (Figure 4). We hid all the Linux management details within the desktop application, allowing `docker build` and `docker run` running on the desktop to “just work” by forward-

ing the invocations to the embedded Linux instance. This approach has been so successful that it has been adopted by other container systems such as Podman,³⁹ and is now a standard way to run containers on macOS and Windows.

We designed a custom Linux distribution called LinuxKit to reflect that—instead of being a conventional standalone Linux distribution—it is intended to be used as a component and be embedded within a larger application. To minimize application startup time, we built a custom userspace that included only the necessary components to run Docker containers, and ran every single component within a container itself, leaving nothing at all running in the root namespace used at boot time. This allowed us to take advantage of the same copy-on-write filesystems and network namespaces that Docker containers themselves use, and to run the entire system in a highly isolated way. The combination

of LinuxKit and HyperKit could boot up a Linux process almost as quickly as a native macOS process; thus, Docker for Mac and Windows applications were born and released in 2016.

Networking. However, while the Linux containers now ran well in macOS and Windows, plumbing networking through to the embedded Linux container proved surprisingly tricky. The conventional approach of bridging Ethernet network traffic from the desktop to the Linux VM required complex network management. Even worse, the bridging approach also fell afoul of firewalls and virus checkers on corporate desktops that detected this as potentially malicious traffic, resulting in thousands of bug reports from our beta users. Fortunately, an ancient tool called SLIRP³¹ provided a viable solution, with an approach that was first used to connect Palmpilot PDAs to the Internet in the mid-1990s!

Outgoing network traffic would trigger false positives in security scan-

ners, since those are often configured to block all traffic from unknown processes that bypass the host OS network stack—exactly what happens when a Linux VM bridges its traffic directly to the desktop network stack. As a workaround, we turned to the unikernel libraries from MirageOS²¹ to translate between Linux networking requests and macOS and Windows native socket calls. When a container attempts a TCP handshake, an ethernet frame containing the TCP SYN is sent to the host over the virtio protocol³³ using shared memory on the Mac (Figure 5). This is then received by the “library VMM” and sent by `sendmsg` into the userspace TCP/IP stack running on the host OS. This userspace stack, dubbed `vpnkit` and written in OCaml,²³ then invokes the macOS `connect()` syscall and either completes the TCP handshake or signals an error. With this architecture, the outgoing traffic from the Linux container will be perceived by the VPN policy as originating from the Docker application rather than from a separate machine. Deploying `vpnkit` in our beta tests in 2016 reduced bug reports from corporate users by more than 99%, and this approach has been a key component of Docker for Mac and Windows ever since. The SLIRP approach has subsequently seen adoption elsewhere in the serverless cloud world,⁴⁰ reviving an old dial-up networking trick to solve new problems in container management.

Incoming network traffic was also a challenge, but for different reasons. By default, when a Linux container listens on a port, it is not automatically exposed to the Internet unless requested on the CLI (e.g., `docker run -p 80:80 nginx` to expose `nginx` on port 80). The ideal user experience when running a container is that the container port appears directly on the desktop IP address, accessible via the browser on a URL such as `http://localhost:8080`. The conventional approach with desktop virtualization software like VMware Fusion would expose a temporary intermediate IP instead of `localhost`. Our LinuxKit kernel installed a custom eBPF program²⁵ that triggered the creation of a corresponding listening socket on the desktop host, and activated a port forwarder to allow the container to receive connections trans-



The key constraint when designing Docker for Mac and Windows is that it had to work without additional configuration for developers already familiar with the Linux version, and also be able to run the same Docker images.



parently without much overhead. This allowed for the perfect developer experience of running a Linux container on a Mac, and having it immediately be accessible on `localhost`, just as it would be on a native Linux machine.

Storage. A similar problem also existed with file storage, since developers need to edit their code and access their data files locally, while still being able to run the code and tests in a container. This live file access is normally done via a “bind mount” on Linux, expressed as `docker run -v /host:/container`. A bind mount is a non-portable Linux kernel filesystem concept, where part of the filesystem is grafted onto another part of the tree. Since macOS and Windows are different kernels, this will not work, so Docker uses the `virtio-fs` shared memory protocol that originated with the KVM hypervisor to send filesystem operations to the host, formatted as FUSE requests. The host receives these requests and invokes the corresponding `open`, `read`, and `write` syscalls. This also means that the developer’s code and data can stay on the host filesystem, making it available to backup and search tools such as Apple’s Time Capsule or Spotlight, rather than requiring these tools to be integrated within the Linux VM.

Enter Windows Services for Linux.

By 2017, the popularity of Linux deployment on the cloud was becoming clear, and Microsoft released the Windows Services for Linux (WSL) subsystem to allow running Linux applications directly on Windows. The first version of this subsystem did not use virtualization, preferring instead to dynamically translate system calls invoked by Linux binaries into the corresponding Windows system calls via another library operating system.²⁹ This approach was successful for many applications, but Docker containers were a step too far for this to work. The Linux kernel has a large number of system calls, and WSL did not support enough of them to run Docker containers.

In 2018, Microsoft rearchitected WSL, releasing version 2, to adopt a similar approach to Docker for Mac, by running a full Linux VM in the background. At this point, Docker for Windows integration is now seamless; WSL2 Docker runs the daemon and user containers inside a LinuxKit WSL

distribution and takes care of forwarding the Docker API and network ports from both Windows itself and from other Linux distributions (Figure 6).

To recap, the architectural approach that enabled Docker containers to evolve across platforms has been the library OS approach of repurposing traditionally “kernel only code” as user-space libraries that can be embedded within other applications. The success of this architecture is demonstrated in its invisibility and ubiquity—millions of developers use tools such as Docker and its derivatives every day without needing to worry about which operating system they are running on.

Emerging Developer Workflows

Multiple CPU architectures. In the early days of Docker, the majority of workloads in the cloud were based on Intel architectures. This all changed with the release of the Amazon Graviton ARM processor for cloud workloads in 2018 and then the Apple M1 ARM CPU series in 2020. Suddenly, there were both cost savings and performance improvements to be had by running workloads on ARM, and developers wanted to take advantage of this. Today, it is necessary to support multiple CPU architectures within the same Docker image so developers can run their applications on Intel, ARM, POWER, or emerging open source RISC-V CPUs. On the server side, this ability was added to Docker images by extending the OCI image format with support for “multiarch manifests” that record which architectures an image is built for.

This still left us with the problem of how to build these images for multiple CPU architectures from a single host, without introducing the notoriously complex problem of cross compilation. We turned to another relatively obscure feature in Linux known as `binfmt_misc`, which allows for executables to be run through custom userspace applications. QEMU² can translate between multiple CPU architectures, so we installed this within the embedded LinuxKit in Docker for Desktop to transparently translate between ARM and Intel binaries. While this was a significant overhead, it was usually necessary only during the build phase, as the resulting multiarch images could be run natively on any host without modi-

fication. Apple subsequently introduced hardware and software support for CPU instruction set translation via “Rosetta” in their CPU series,²⁴ which was easily integrated into the Docker architecture. Today, running Intel and ARM containers side by side is a common workflow for developers.

Managing secrets with trusted execution environments. Managing secrets such as passwords or API keys has always been a challenge for containerized environments, since they must be injected dynamically into a container rather than baked into a filesystem image. Docker has always supported socket forwarding, so a local domain socket can be mounted into a container, including forwarding that socket

into the Linux VM in the case of Docker for Mac or Windows. This allows users to use key-management systems such as `ssh-agent` inside a container without ever directly revealing the keys. Socket forwarding provides a good first level of protection, but modern environments require more layers of defense against malware lurking in the ever-growing software supply chain.

The first port of call is to use hypervisor protection directly within the container runtime to increase cross-container protection levels.³² Beyond that, Docker has been integrating a hardware feature in modern CPUs that can protect secret data from even the host operating system. Trusted execution environments (TEEs) allow for the creation

Figure 5. Traffic from a traditional bridged network is blocked by local policy, while traffic from local processes and VM traffic indirection over SLIRP is accepted.

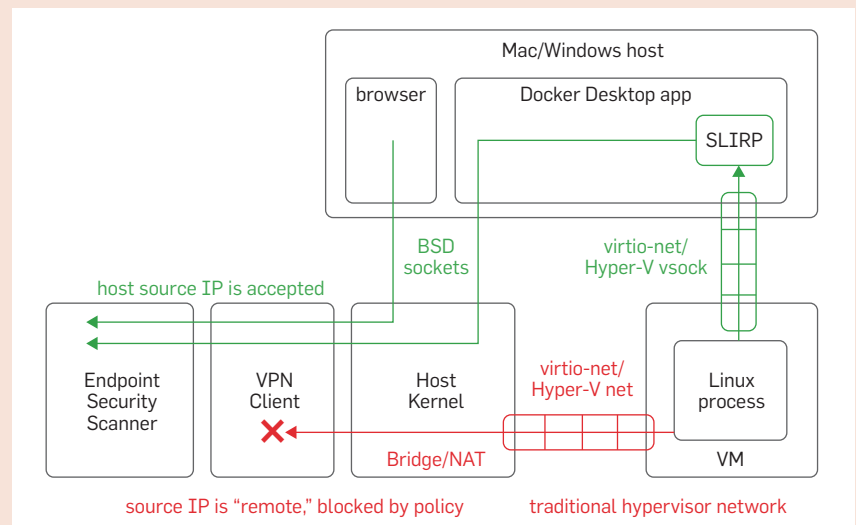


Figure 6. Docker for Windows architecture on WSL2.

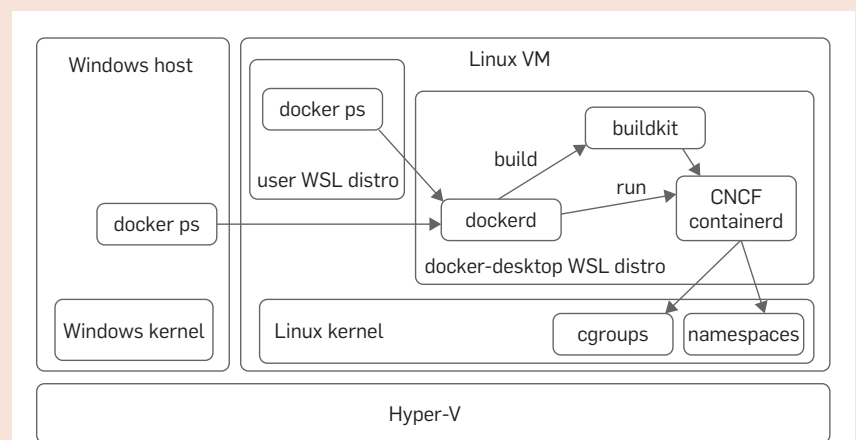
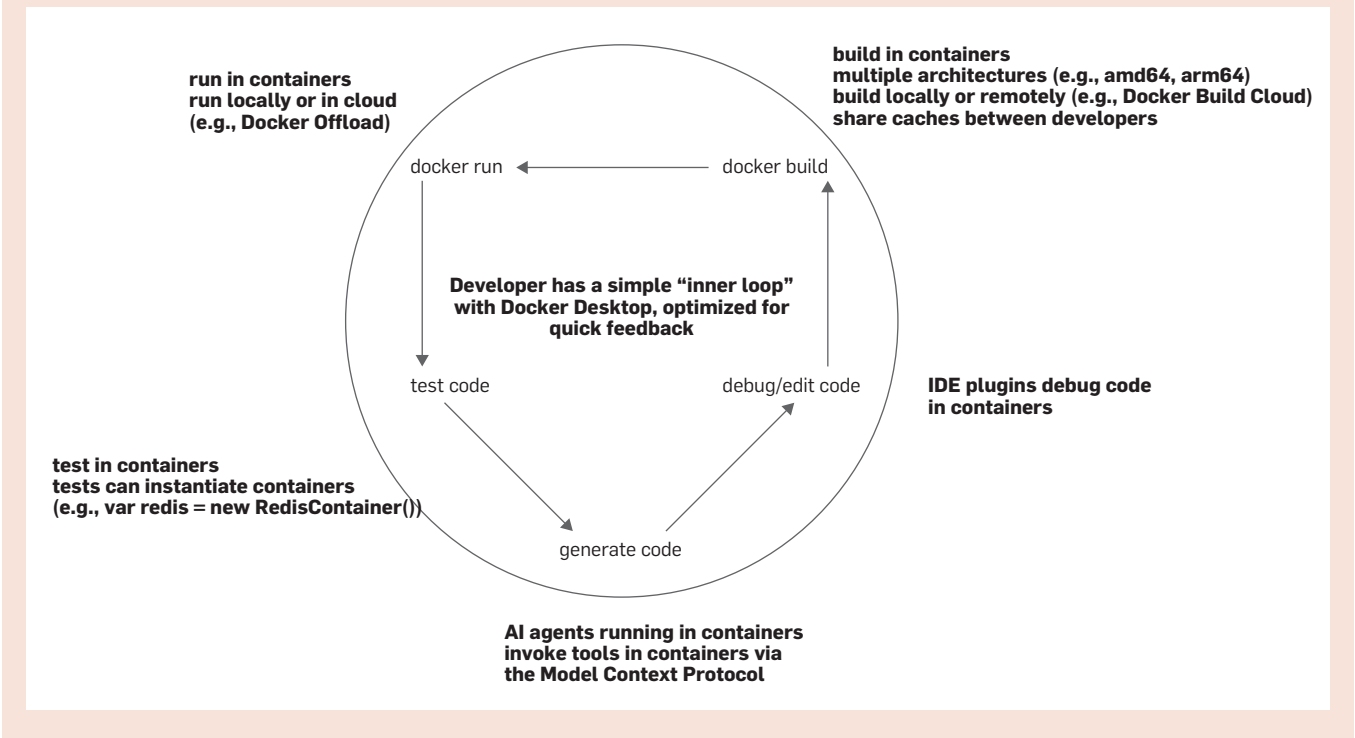


Figure 7. The Docker developer workflow in 2026.

of “confidential VMs” that can enforce data-access restrictions across the application, kernel, and even hypervisor boundaries.³⁵ However, configuring and using TEEs has much of the same management complexity as OS virtualization, since it effectively boots up a mini operating system kernel within the TEE.

A community of users from the Confidential Containers working group has been developing applications that can run within TEEs and be managed via Docker. The client-server architecture of Docker integrates well with these applications, since the Docker CLI running on the desktop can forward encrypted messages from a local TEE through multiple forwarding sockets across the host, all the way through to a remote TEE environment running within a cloud environment.⁹ This allows developers to authenticate to sensitive cloud environments without needing to be on site, and to store their credentials securely within desktop enclaves and retain the convenience of local development.

GPGPU support for AI workloads. We have so far focused on how Docker has evolved to run on different operating systems and CPUs, but the rise of AI workloads has brought entirely new challenges. Machine learning work-

loads mostly run on GPUs, which the Docker ecosystem has had to adapt to support. The core challenge is that GPU workloads require precisely matched kernel GPU drivers and user-space libraries, while multiple containers run on a single shared kernel. This introduces the same basic conflict that Docker was designed to solve in the first place: how to run multiple applications with conflicting dependencies on the same machine. What happens if two applications require different versions of the same kernel GPU driver?

Since March 2023, Docker has supported the container device interface (CDI),¹⁰ which supports customizing the filesystem image at the point of container start, allowing GPU device files and GPU-specific dynamic libraries to be bind mounted, and the `ld.so` cache to be regenerated. While this ensures Docker images are portable across a particular class or vendor of GPUs, it is not entirely seamless across different operating systems and hardware brands. The available dynamic libraries added by CDI effectively define different APIs, so there is nothing comparable to the stable Linux system call ABI that has traditionally been the interface for containers running on CPUs. An application designed for an Nvidia GPU

will still struggle to run on an Apple M-series CPU, since the underlying GPU virtualization support is not yet mature enough to translate the vector instructions across such diverse hardware. We are continuing to work with the wider container community and GPU manufacturers to develop a more flexible and secure way to manage GPU-related dependencies, and are hopeful that initiatives for portable interfaces³⁷ will converge toward a consensus.

Conclusion

Docker began its life in 2013 aiming to help developers more easily build, share, and run any application. It is now integrated deeply into the standard cloud and desktop development workflows, with millions of developers worldwide using it daily and billions of monthly requests. One of our consistent goals has been to maintain a vibrant and diverse open source community that builds standards for interoperation, ensuring there is no lock-in to any single vendor. The Cloud Native Computing Foundation (CNCF) acts as a steward for several core components,⁷ while the Open Container Initiative (part of the Linux Foundation) is the steward of the image format. Today, multiple implementations of

many of these elements are thriving, and we are seeing growing numbers of deployments in the cloud, on the desktop, and in the edge for automotive, mobile, and even spacecraft.¹⁸

Software development moves quickly, so we are constantly evolving Docker under the hood to keep up with the latest developments. Figure 7 shows a typical developer workflow in 2025 that integrates continuous test and deployment, integrated development environment (IDE) language servers, and AI assistance via agentic coding. From a Docker perspective, the core “build and run” workflow remains very similar to the user experience from a decade ago, but with much more systems support²³ to reduce the friction involved with running in diverse environments that all need robust sandboxing.

If you are a developer, our goal is to make Docker an invisible companion that helps you ship your code faster—and for you to enjoy the process. Docker is designed to be extensible enough to evolve with your needs, especially in the face of modern AI coding workflows. We hope you remix it into whatever software environment you find yourself in, and that you share what you learn with our community.

Acknowledgments

We are grateful to Jon Crowcroft, Michael W. Dales, Patrick Ferris, Ryan Gibb, and Hamed Haddadi, who gave us comments about this article. We also thank the Docker community, past and present, for their feedback and contributions to the Docker project including but not limited to Solomon Hykes (the founder of the project), Harald Albers, Kevin Alvarez, Jeff Anderson, Mary Anthony, Gianluca Arbezano, Vincent Batts, Morgan Bauer, Laura Brehm, David Calavera, Michael Crosby, Doug Davis, Stephen Day, Bruno de Sousa, Vincent Demeester, Sven Dowideit, Alex Ellis, Phil Estes, Lorenzo Fontana, Jessie Frazelle, Thomas Gazagnaire, Brian Goff, Tianon Gravi, Paweł Gronowski, Evan Hazlett, Erik Hollensbe, John Howard, Andrew Hsu, Olli Janatuinen, Lei Jitang, Scott Johnston, Vishnu Kannan, Samuel Karp, Albin Kerouanton, Kir Kolyshkin, Kenfe-Mickaël Laventure, Aaron Lehmann, Djordje Lukic, Andrea Luzzardi, Derek McGowan, Alexander Morozov, Richard Mortier,

Antonio Murdaca, Rob Murray, Bjorn Neergaard, Daniel Nephin, Arnaud Porterie, Jana Radhakrishnan, Anusha Ragunathan, David Sheets, Boaz Shuster, Cory Snider, Cristian Staretu, John Stephens, Akihiro Suda, Yong Tang, Sam Thibault, Shaun Thompson, Tõnis Tiigi, James Turnbull, Sebastiaan van Stijn, Tibor Vass, Austin Vazquez, Madhu Venugopal, Victor Vieux, Sam Whited, and Jeremy Yallop. 

References

- Barham, P. et al. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 164–177.
- Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conf., FREENIX Track Vol. 41*, USENIX Association (2005).
- Biederman, E.W. L2 network namespace infrastructure. (2007); <https://lwn.net/Articles/239625/>
- Boettiger, C. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Rev.* 49, 1 (2015), 71–79.
- Burns, B., Beda, J., Hightower, K., and Evenson, L. *Kubernetes: Up and Running*. O'Reilly Media (2022).
- Courtès, L. and Wurmus, R. Reproducible and user-controlled software environments in HPC with Guix. Springer Intern. Publishing (2015) 579–591.
- Crosby, M. Containerd: A daemon to control runC. Docker (2015); <https://www.docker.com/blog/containerd-daemon-to-control-runc/>
- Dijkstra, E. and Löh, A. NixOS: A purely functional Linux distribution. In *Proceedings of the 13th ACM SIGPLAN Intern. Conf. on Functional Programming*. ACM (2008), 367–378.
- Ferguson, C. Securing admin access to Monzo's platform. Monzo (2023); <https://monzo.com/blog/2023/12/14/securing-admin-access-to-monzo-platform>
- Cloud Native Computing Foundation. Container device interface specification v1.0.0 ([n. d.]); <https://github.com/cncf-tags/container-device-interface/blob/main/SPEC.md>
- Glickstein, B. and Hodgson, K. GNU Stow—managing the installation of software packages. GNU (1998); <https://www.gnu.org/software/stow/>
- Harrison, M.A., Ruzzo, W.L., and Ullman, J.D. Protection in operating systems. *Commun. ACM*, 19, 8 (Aug. 1976), 461–471.
- Hayes, B. Cloud computing. *Commun. ACM* 51, 7 (Jul. 2008), 9–11.
- Hykes, S. The future of Linux containers. YouTube (2013); <https://www.youtube.com/watch?v=w9CAH9nSLs>
- Kamp, P.-H. and Watson, R.N.M. Jails: Confining the omnipotent root. In *Proceedings of the 2nd Intern. SANE Conf., Vol. 43*. (2000), 116.
- Korotae, K. IPC namespace core (2006); <https://lwn.net/Articles/187274/>
- Lin, C., Nadi, S., and Khazaee, H. A large-scale data set and an empirical study of docker images hosted on docker hub. In *2020 IEEE Intern. Conf. on Software Maintenance and Evolution (ICSME)*. IEEE (2020), 371–381.
- Maas, N., Willnecker, R., Hemmersbach, R., and Hauslage, J. Apex: A new commercial off-the-shelf on-board computer platform for sounding rockets. *Rev. of Scientific Instruments* 90, 10 (Oct. 2019).
- Madhavapeddy, A. Improving Docker with Unikernels: Introducing HyperKit, VPNKit and DataKit. Docker (2016); <https://www.docker.com/blog/docker-unikernels-open-source/>
- Madhavapeddy, A. et al. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symp. on Networked Systems Design and Implementation*. USENIX Association (2015), 559–573; <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy>
- Madhavapeddy, A. et al. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th Intern. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM (2013), 461–472.
- Madhavapeddy, A. and Scott, D. Unikernels: Rise of the virtual library operating system. *ACM Queue* 11, 11 (Nov. 2013), 30–44.
- Madhavapeddy, A. et al. Functional networking for millions of Docker desktops (experience report). *Proc. ACM Program. Lang.* 9, ICFP (Aug. 2025), 256:597–256:615.
- Mettig, R., Glass, C., Case, A., and Richard, G.G. Assessing the threat of Rosetta 2 on Apple Silicon devices. *Forensic Science Intern.: Digital Investigation* 46(Sep. 2023), 301618.
- Miano, S. et al. Creating complex network services with eBPF: Experience and lessons learned. In *2018 IEEE 19th Intern. Conf. on High Performance Switching and Routing*. IEEE (2018), 1–8.
- Newman, S. *Building Microservices*. O'Reilly Media (2021).
- Ovens, S. The 7 most used Linux namespaces. Redhat (2021); <https://www.redhat.com/en/blog/7-linux-namespaces>
- Pike, R. et al. The use of name spaces in Plan 9. *SIGOPS Oper. Syst. Rev.* 27, 2 (Apr. 1993), 72–76.
- Porter, D.E. et al. Rethinking the library OS from the top down. In *Proceedings of the 16th Intern. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM (2011), 291–304.
- Price, D. and Tucker, A. Solaris zones: Operating system support for consolidating commercial workloads. In *LISA Vol. 4*, USENIX Association (2004), 241–254.
- Price, K., May, T., and Moustakis, C. Slirp, the ppp/slip-on-terminal emulator (1996); <https://slirp.sourceforge.net>
- Randazzo, A. and Tinnirello, I. Kata containers: An emerging architecture for enabling MEC services in fast and secure way. In *2019 6th Intern. Conf. on Internet of Things: Systems, Management and Security*. IEEE (2019), 209–214.
- Russell, R. Virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Rev.* 42, 5 (2008), 95–103.
- Stevens, W.R. *TCP/IP Illustrated (vol. 1): The Protocols*. Addison-Wesley (1993).
- Tarkhani, Z. and Madhavapeddy, A. Information flow tracking for heterogeneous compartmentalized software. In *Proceedings of the 26th Intern. Symp. on Research in Attacks, Intrusions and Defenses*. ACM (2023), 564–579.
- Tennenhouse, D.L. Layered multiplexing considered harmful. In *Protocols for High-Speed Networks* (1989).
- Tillet, P., Kung, H.T., and Cox, D. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN Intern. Workshop on Machine Learning and Programming Languages*. ACM (2019), 10–19.
- Viro, A. Per-process namespaces for Linux (2001); <https://lore.kernel.org/all/Pine.GSO.4.21.0102242253460.24312-100000@weyl.math.psu.edu/>
- Walsh, D. *Podman in Action: Secure, Rootless Containers for Kubernetes, Microservices, and More*. Simon and Schuster (2023).
- Young, E.G. et al. The true cost of containing: A gvisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX Association (2019); <https://www.usenix.org/conference/hotcloud19/presentation/young>

Anil Madhavapeddy (avsm2@cam.ac.uk) is the Professor of Planetary Computing at the University of Cambridge, Cambridge, U.K.

David J. Scott is a principal engineer at Docker, U.K.

Justin Cormack is an independent researcher in the U.K.

 This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2026 Copyright held by the owner/author(s).



Watch the authors discuss this work in the exclusive Communications video. <https://cacm.acm.org/videos/docker-containers>