2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

How Docker containers can make chip development more productive

Philipp Wagner, Martijn Berkers, Holger Horbach, Johannes Kösters, Udo Krautz
IBM, Böblingen, Germany

***Abstract*—Chip design is, in a way, the art of managing complexity. Much of this complexity is handled by the tools employed during the design process, which in itself need to be combined and configured properly. Docker (OCI) containers can help deal with this complexity by bundling all dependencies and configuration together in a shareable form. We show two use cases where containerization makes chip development more productive: chip development on a local machine with VS Code Dev Containers, and container checkpointing as a way to skip redundant computation in functional verification. Based on a user survey and telemetry data, we show that developers enjoy the productivity wins from the Dev Container environment.**

***Keywords*—front-end development; productivity; containers; Docker; checkpointing**

I. Introduction

Many of us think and talk about developer and development productivity, for a good reason: increasing productivity means getting a product into the market faster or cheaper. Yet, "increasing productivity" isn't a simple lever to turn. Productivity results from getting many of the things right that are involved in making a product, or in our case, a chip. But it often boils down to a simple recipe: do more of what you want to do (i.e., create value), and less of what you don't want to do.

This advice is valid, but not actionable. In the ever-lasting effort to become even more productive in chip development, we at IBM found Docker containerization to be a key enabler. We focus on the front-end development flow, including RTL design and functional verification. We start the discussion by introducing the reader to containers in Section II; Readers already familiar with the topic can skip ahead to Section III, where we discuss how VS Code Dev Containers makes our front-end developers more productive. A second use case for containerization is presented in Section IV: the ability to checkpoint and restore a complex simulation environment to increase simulation efficiency. Section V concludes the paper by summarizing the results and providing an outlook on future developments.

II. Why containers are great in chip development, and how they work

To understand how containers lead the way to increased developer productivity, we need to take a closer look at what a container actually is and what it is valued for.

A. A first look at Docker containers

At its core, a container can be seen as a way to bundle dependencies of a complex "application" in an executable form, from the operating system up to all binaries, libraries, and configuration files needed to fulfill a certain task (like running a simulation of a circuit). This makes containers portable between machines and its execution reproducible.

Behind the scenes, containers build on top of a Linux kernel feature called "namespaces" [1]. Tools like Docker [2] or Podman [3] simplify the interaction with containers. Initiatives like Open Container Initiative (OCI) specify how container images are structured to enable sharing them between runtime environments.

Containers can run on any Linux machine; users of other operating systems like Windows or macOS can run containers by creating a Linux virtual machine first, or by using a tool that does so under the covers, like Docker Desktop [4].

Individual containers can be conveniently managed on the command-line (e.g., by calling `docker run`). For dozens of even thousands of containers orchestration tools allocate resources, start and stop containers as needed, group related services, etc. The most common orchestration tool is Kubernetes, or a variant of it, like Red Hat OpenShift.

Using Docker or Podman (a drop-in replacement in this case) is simple, as the following complete walkthrough shows. Typically the contents of a container image are described in a `Dockerfile`, which looks a bit like a shell script.

```
# Start with a minimal Red Hat Enterprise Linux (RHEL) 8.
FROM redhat/ubi8
# Install system tools.
RUN dnf install -y make git
# ...
```

A call to `docker build` then consumes the `Dockerfile` and produce a container image.

```
$ docker build -f Dockerfile -t my-namespace/my-image:latest .
```

After a few seconds, the container image is ready to be instantiated into a container with `docker run`, followed by the command to execute inside the container. Our host runs Fedora Linux. To show that we are actually executing the command inside the container with Red Hat Enterprise Linux (RHEL) 8 we take a look at `/etc/os-release`.

```
$ cat /etc/os-release | grep PRETTY_NAME
PRETTY_NAME="Fedora Linux 42 (Workstation Edition)"
$ docker run -it my-namespace/my-image:latest cat /etc/os-release | grep PRETTY_NAME
PRETTY_NAME="Red Hat Enterprise Linux 8.10 (Ootpa)"
```

After the build, the container image is only available on the machine where it was built. To share the image with others it is pushed to a remote location called “registry,” Docker Hub being the default.

```
$ docker push my-namespace/my-image:latest
```

Now anybody can get an identical environment by executing the same `docker run` command as used above; the image is pulled automatically from the registry if it does not exist locally.

This short walkthrough already highlights why containers have become so ubiquitous: based on robust technology in the Linux kernel, Docker and others have built very convenient tooling and an extensive ecosystem around it, where many things “just work.”

The complexity of a container solution can increase, of course, as one adds requirements (be it “enterprise-grade” authentication, or sophisticated deployments in Kubernetes clusters). Also, the use of the Docker command-line tool is by far not the only way to build containers or interact with them. Still, the journey into containerization can start small and expand as additional requirements come up. Given how vast the container ecosystem is these days it’s unlikely that the journey hits a roadblock any time soon.

Equipped with a basic understanding of containers we are ready to explore how development on a local computer with VS Code Dev Containers makes developers more productive.

III. Local development with VS Code Dev Containers

Visual Studio Code (VS Code) Dev Containers seamlessly integrate VS Code [5], today’s most popular integrated development environment (IDE) [6], with a containerized development environment in which all chip development tools run. This combination creates a development environment that is easy to setup, fully runs on a local computer, and is customizable to the individual developer personality.

We first take a closer look at why this setup makes developers productive, followed by a discussion of the implementation, and finally results from a user survey.

A. Dev Containers from a user’s point of view

VS Code Dev Containers offer the user a fully integrated, local development environment for chip development, as if they were running VS Code on a Linux machine with all development tools installed.

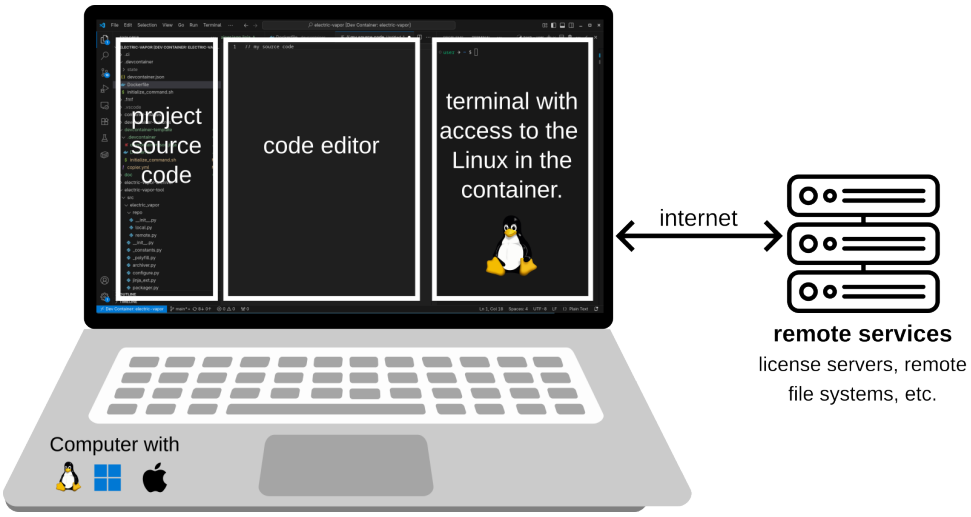

Figure 1: An illustration showing VS Code Dev Containers in action.

Figure 1 shows the main work panes. Users can browse through the project's files, open them in the integrated editor with full syntax highlighting and state-of-the-art semantic autocomplete, use the AI integration, and do all the other things one expects from an advanced editor. Additionally, users can open a terminal pane. This terminal is the window into the underlying containerized Linux where all chip development tools are present. To build the project, synthesize it, or run a simulation, users just type whatever commands they are used to (or create shortcuts in the user interface for it).

Before development can start, users have to first perform a one-time setup of a small set of software components.

- VS Code (free of charge)
- Dev Containers VS Code extension (free of charge)
- Docker[1]. On Windows and macOS, use either Docker Desktop (proprietary) or Rancher Desktop (open source)
- On macOS to display graphical applications: XQuartz (optional, open source)

All software components are available for Windows, macOS, and Linux. On Macs with an ARM CPU the project can either provide a ARM-based development environment (if all tools are available for ARM), or use a x86_64 container (utilizing the built-in Rosetta binary translation).

Next developers obtain the project source code or directly specify a Git repository URL, and ask VS Code to open the project in a Dev Container. Under the hood, VS Code reads the project configuration from dedicated files in the project's source code, instructs Docker to pull a container image suitable for the project, starts it, places the source code in it, and finally presents the user with an editing window.

The whole one-time setup takes around 15 minutes in our experience. After that time, users are able to continue coding in a highly productive, state-of-the-art development environment.

How does all of this work? The next section takes a look behind the scenes.

B. How to configure VS Code Dev Containers

Before developers can make use of a Dev Container, the project needs to set one up by adding at least one file to its source code repository. In the following, we describe a setup that goes beyond the bare minium and matches the production setup we are currently using at IBM. Dev Containers are by no means restricted to this setup; refer to the official documentation for details.

[1]It is possible to use Podman and Podman Desktop instead of Docker, but Podman is not yet well supported by VS Code. We repeatedly found edge cases that led us to using Docker for the time being.

```
{
  "name": "my-chip",
  // The Docker image to build this container on.
  "image": "icr.io/chip-projects/images/my-chip:v1",
  // Initialization steps on the host, e.g. obtaining credentials.
  "initializeCommand": "${localWorkspaceFolder}/.devcontainer/initialize_command.sh",
  // Install Dev Container Features.
  "features": {
    "ghcr.io/devcontainers/features/common-utils:2": {},
    "icr.io/chip-projects/features/vscode-developer-general:1": {},
    "icr.io/chip-projects/features/vscode-developer-chip:1": {}
  },
  // Customizations like VS Code settings; we place them in reusable Features instead.
  "customizations": {}
}
```

Listing 1: `devcontainer.json`, the configuration file describing the Dev Container.

The entry point is the configuration file is `.devcontainer/devcontainer.json` [7]. This file is read by the Dev Containers extension when users open a project in a Dev Container. Listing 1 shows a slightly simplified configuration file that describes a Dev Container project named “my-chip”.

At first, the Bash script `initialize_command.sh` is called *on the host* (or the Linux VM on the host in case of Windows or macOS). We use this script to prompt the user for credentials to authenticate against the IBM Cloud Container Registry (icr.io), from where the container image specified in the `image` key is downloaded. The container image is described in the `.container/Dockerfile` file and provides an environment tailored to the project’s needs.

For each project, we build a Docker image that contains everything to run batch jobs, e.g., simulations, synthesis runs, etc. Re-use between projects is achieved by layering the project image on top of a more generic “chip development” image.

To keep the image size small, it does not contain any tools needed for interactive (human) use, like debuggers, documentation, etc. This functionality is added through Dev Container Features [8]. Features are essentially a combination of a shell script and VS Code configuration snippets that are applied on top of the container image. Just like for container images, they can be layered from from generic ones to project-specific ones.

Further customization can be done directly in the `customizations` object, where VS Code extensions can be pre-installed, or settings configured.

C. Why VS Code Dev Containers make developers more productive

Developer productivity is a vast field of research with surprisingly little hard facts [9]. Much of the uncertainty can be attributed to us being being individuals, each with unique preferences and habits. But even with that in mind, we are able to identify properties of a development environment which influence developer productivity. We look at three properties: the value of a ready-to-use environment, the ability to customize the development environment, and finally, the value of iteration speed.

1) The value of ready-to-use environments: A “chip development environment” is rarely a single tool; it is a combination in-house and third-party tools, scripts, configuration files, and source code from different origins. Since the development environment itself is not the product, it tends to be developed only to the extent that it can get the chip done, resulting in an environment that is sometimes fragile to setup. Documentation to guide new developers through the process helps, but is prone to errors and time-consuming to follow. Especially for short-term assignments (e.g., internships, helping out with a problem on another project, etc.), the hurdle to get a working development setup can consume a significant portion of time spent on the project.

With containers, we describe in executable form what an optimal project environment looks like, starting with a bare operating system installation. We test these instructions as part of our continuous integration pipelines to ensure they remain valid.

Additionally, the Dev Container ships with pre-configured editor settings and recommended extensions that the development project as a whole has found to be helping productivity – a way of sharing best practices in executable form. Taken together, developers can get “real” work done faster – a win for productivity.

2) Customize the development environment: Developers are human, and development is a creative process. A development environment must be able to adapt to the individual user.

With Dev Containers, users get an abundance of customization options, if they wish to do so. With over 75,000 extensions[2] and thousands of configuration options, users are very likely to find the customization they are looking for. Furthermore, developers have root permissions in their local development environment and are able to install any utility that makes them productive, without impacting others.

3) Local development iterations are probably faster: The development process (no matter if it's software or hardware) is done in iterations: code, build, debug, publish, and start over. Some parts of the iteration create value, most notably the coding and debugging parts. Other parts of the iteration have to be performed, but don't add value. Productive developers spend less time on those steps.

In our environment at IBM, we have experienced that local development in a Dev Container reduces the amount of time spent waiting for commands to complete. Even though this observation is not universally true, it matches anecdotal evidence from other companies.

The time to build and commit code is often dominated by file access latencies. During a development iteration, developers tend to modify only a handful of files; all the others are unchanged and the build outputs should be re-used from previous runs. Still, tools like Git or GNU Make have to access all files to determine which files actually changed. Even though accessing a single file does not take long, the effect is multiplied thousandfold in large repositories. In those, it is not uncommon to see commands like `git status` take minutes instead of fractions of a second when a large source code repository is stored on a network file system as opposed to a local SSD. Additionally, file access latencies heavily influence the user experience in interactive editing.

Somewhat counterintuitively, we also observe that simulations for functional verification often run faster in local containers. The single-thread performance of a CPU in a modern developer laptop can, in many cases, beat server processors which are optimized for lower frequencies and higher core counts.

Even though there's no single knob to influence productivity, we have seen a rise in perceived productivity from our developers. Three of the main contributing aspects are pre-configured environments, the ability to fully customize the development environment, and faster development iterations.

D. User reception

Does the Dev Container increase productivity? We think so, even though measuring productivity is almost impossible. What we can provide instead are hints towards that, or proxy metrics.

A key metric for us is developer adoption. Developers want to be productive, they want to "get things done." Today, our primary development environment is a remote desktop environment. In early 2025, we added the Dev Container to one of our projects and allowed developers to opt into this way of doing development. After only three months, and with no requirement to do so, around thirty percent of developers use the Dev Container daily, and more than half of the users have tried it at least once.

To better understand the motivations of users, we conducted a user survey. Overall, 88 percent of respondents felt the Dev Container made them more productive. They attributed that largely to the ability to work locally and the speed and especially latency advantages seen there. As many users have not used VS Code before, they also observed that this editor supports them well.

E. Limits of local development

We found that Dev Containers are a great solution for front-end chip design, i.e., development tasks that are close to software development. But Dev Containers are neither the only solution to tackle the challenges we have described, nor are they applicable in all scenarios.

First and foremost, a Dev Container cannot use more resources than the local computer provides. Memory-intensive workloads, especially back-end tasks like the synthesis of large chip designs, can easily exceed the

[2]On June 30, 2025, 76,783 extensions were listed at https://marketplace.visualstudio.com/search?target=VSCode.

memory limits of a developer laptop with 16 or 32 GB of memory. Large-scale regression runs in functional verification that benefit from parallelization are also a bad fit for a laptop CPU.

Sharing of simulation results and collaborative debugging also requires rethinking. With our remote desktop setup, all users can access the same network file system and collaborate to debug simulation failures by looking at the same output files. For our container users, we have explored ways to share simulation outputs, but continue to iterate on tools and processes in that area.

For us, Dev Containers for chip development are a success story. Our second use case for containers promises to become a similar win for productivity.

IV. Container checkpointing

A. The motivation for checkpointing

Some computation tasks in a chip design flow can take hours or even days. For those jobs, checkpointing can increase development productivity in a number of ways.

- Skip repeated computation, e.g., long-running initialization sequences at the beginning of a simulation or an analysis task.
- Add resiliency to the computation by taking periodic checkpoints to recover from infrastructure failure, or to move workload from one machine to another for maintenance work ("live migration").
- Share an application in a specific state with a team of engineers. For example, a simulation run in a failure state can be shared with a team of engineers. All engineers can now in parallel debug the failure in a fully interactive, yet independent environment.

When a checkpoint is taken, the state of the running application, e.g., the state of the simulator including the simulated model and the testbench state, are written to disk. This checkpoint can be restored at a later time on the same machine or on another machine. The application will then resume its computation as if nothing happened. Over the years, multiple checkpointing mechanisms have been developed.

B. Related work

Many applications such as our in-house simulator have supported checkpoints for a long time through custom code that writes all relevant data structures to disk and restores them as needed. However, with the growing complexity of chip design environments, we are looking for checkpointing solutions that go beyond a single application (or process) and checkpoint all (relevant) running processes, including their libraries, open files, etc.

A commonly used, mature solution is DMTCP [10]. DMTCP is able to capture the full state, as stored in memory, of a set of running applications without the need to modify the application(s). It operates in user-space and relies on a preloaded library that intercepts certain system calls. DMTCP incurs a small runtime overhead (typically below 10 percent [11, 12]). The filesystem changes made by the application (such as log files written) are not part of the checkpoint created by DMTCP, but can be included manually.

A level below DMTCP operates Checkpoint/Restore In Userspace (CRIU) [13]. Despite the name, it relies heavily on Linux kernel functionality to capture the state of a running application. It has been part of Linux for over a decade now[3], and is hence available by default even in enterprise distributions like Red Hat Enterprise Linux 8. With kernel support CRIU promises to be more robust than DMTCP. Just like DMTCP, CRIU only concerns itself with checkpointing the memory state.

Container checkpointing combines CRIU to checkpoint the in-memory application state with filesystem snapshots, as they are readily available in the way filesystem layering works in containers. Both Docker [14] and Podman [15] have their own implementations of container checkpointing, as does Kubernetes [16]. Even though the checkpointing implementations of Docker and Podman look similar at first glance, they differ widely. In contrast to DMTCP, root permissions are required to create checkpoints in both Docker and Podman (due to the way CRIU operates).

[3]CRIU requires a Linux kernel in version 3.11 (or newer), which was first released in September 2013.

C. Container checkpointing for functional verification

We have implemented container checkpointing for our functional verification environment. Our implementation consists of three parts:

- A simulation for functional verification running in a container. The container contains the source code and all tools to run the simulation. We use an in-house simulator and a testbench written in C++, but the approach is not limited to this setup.
- Podman.
- A custom "checkpoint service" daemon (discussed below).

The creation of a checkpoint can either be initiated on the host (*host-triggered checkpoints*), or by the application in the container itself (*application-triggered checkpoints*). Application-triggered checkpoints are interesting because in many cases, the testbench knows best when a good moment for a checkpoint has come: after the initialization sequence is done, after 1000 cycles of simulated time have elapsed, when a data miscompare is observed, etc.

Our checkpoint service bridges the gap between the verification environment running in the container, and the container engine (Docker or Podman). It supports both modes of operation. For host-triggered checkpoints, the checkpoint service can be instructed to create a checkpoint through a command-line interface. For application-triggered checkpoints the checkpoint service provides a HTTP-based REST API, exposed over a Unix socket mounted into the container. (This method of communication matches the way Docker communicates between its server and client components.) The application inside the container can then connect to this socket to talk to the checkpoint service and issue requests.

Upon receiving a checkpoint request, the checkpoint service calls `podman container checkpoint`, which causes Podman to create a checkpoint and write it to file.

After an application requests a checkpoint, it goes into to a polling loop. Soon after that, Podman suspends the execution of the application. When the container is then restored later, it still finds itself in the polling loop. The checkpoint service can then instruct the application to make certain changes (e.g., enable more verbose debug output, or reseed the testbench) before it resumes running.

D. Checkpointing performance

We found container checkpoints to work very well for our functional simulation setup. Both the time to create and restore a checkpoint, as well as the file size of the resulting checkpoint, are a linear function of the working set in memory and on disk.

In our testing[4], the time to create a checkpoint increased linearly from around 1 second for a 200 MiB working set to around 10 seconds for a 4 GiB working set when compressing them with the default zstandard compression. Without compression, checkpoint times were very similar. Restore times with zstandard compression were between 1 and 11 seconds, closely matching the time required to take a checkpoint. Without compression, restores were twice as fast. Even for hard-to-compress fully random data the checkpoint file size did not exceed the size of the working set plus a fixed overhead of around 200 KiB.

E. We prefer Podman for checkpointing

Even though the checkpointing functionality in Docker and Podman look similar at first glance, they differ greatly. In Docker, checkpointing is marked experimental and needs to be enabled explicitly. The checkpoint only contains the memory state; the filesystem state has to be obtained separately (e.g., by committing the container to an image). In our testing, we have observed multiple cases where Docker was not able to create a checkpoint or restore it. (We did not debug those fully.)

In contrast, the checkpointing functionality in Podman was very reliable in our testing.[5] Additionally, Podman offers attractive features over Docker: checkpoints cover both the filesystem and the memory state, the ability to write an OCI image of a checkpoint, and much more, as documented at [17].

[4]Measurements taken on an Intel Core i9-13900H laptop with 32 GiB memory and an SSD.
[5]We tested various versions of Podman over the last year, currently we are running Podman 5.5.2.

V. Summary and Outlook

In our search for a more productive chip development environment at IBM, we found containerization helpful. After only three months of being offered, around 30 percent of our front-end engineers in a large chip project have moved from a remote desktop environment to using VS Code Dev Containers as their day-to-day development environment, because they feel it makes them more productive. Container checkpointing is a technology we are currently exploring to speed up simulations in functional verification by skipping repeated initialization steps. Initial results are very promising, and we are now looking at ways to deploy this approach more widely in our compute clusters. One thing is for sure: our journey into containerization won't stop here – the productivity wins are just too promising.

Acknowledgments

The authors thank Matthis Roppel for his implementation work on container checkpointing and all Dev Container users at IBM for their valuable feedback.

References

[1] *Namespaces(7) - Linux Manual Page*. v6.10. Sept. 1, 2024. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 06/28/2025).

[2] *Docker: Accelerated Container Application Development*. Apr. 9, 2025. URL: https://www.docker.com/ (visited on 06/28/2025).

[3] *Podman*. URL: https://podman.io/ (visited on 06/28/2025).

[4] *Docker Desktop: The #1 Containerization Tool for Developers — Docker*. Apr. 7, 2025. URL: https://www.docker.com/products/docker-desktop/ (visited on 06/28/2025).

[5] *Visual Studio Code - Code Editing. Redefined*. URL: https://code.visualstudio.com/ (visited on 04/22/2025).

[6] *Technology — 2024 Stack Overflow Developer Survey*. URL: https://survey.stackoverflow.co/2024/technology#1-other-tools (visited on 04/22/2025).

[7] *Development Container Specification*. URL: https://containers.dev/implementors/spec/ (visited on 06/29/2025).

[8] *Dev Container Features Reference*. URL: https://containers.dev/implementors/features/ (visited on 06/29/2025).

[9] Ciera Jaspan and Collin Green. "A Human-Centered Approach to Developer Productivity". In: *IEEE Software* 40.1 (Jan. 2023), pp. 23–28. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2022.3212165. URL: https://ieeexplore.ieee.org/document/9994260/ (visited on 10/04/2023).

[10] Jason Ansel, Kapil Arya, and Gene Cooperman. "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. 2009 IEEE International Symposium on Parallel & Distributed Processing. May 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5161063. URL: https://ieeexplore.ieee.org/document/5161063 (visited on 07/03/2025).

[11] Igor Ljubuncic et al. "Be Kind, Rewind: Checkpoint & Restore Capability for Improving Reliability of Large-Scale Semiconductor Design". In: *2014 International Conference on Intelligent Networking and Collaborative Systems*. 2014 International Conference on Intelligent Networking and Collaborative Systems. Sept. 2014, pp. 622–627. DOI: 10.1109/INCoS.2014.90. URL: https://ieeexplore.ieee.org/abstract/document/7057160 (visited on 07/03/2025).

[12] *DMTCP (FAQ): Distributed MultiThreaded Checkpointing*. URL: https://dmtcp.sourceforge.io/FAQ.html (visited on 07/03/2025).

[13] *CRIU*. CRIU. URL: https://criu.org (visited on 07/03/2025).

[14] *Docker Checkpoint*. Docker Documentation. URL: https://docs.docker.com/reference/cli/docker/checkpoint/ (visited on 04/22/2025).

[15] *Podman Checkpoint — Podman*. URL: https://podman.io/docs/checkpoint (visited on 07/04/2025).

[16] *Kubelet Checkpoint API*. Kubernetes. URL: https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api/ (visited on 07/04/2025).

[17] *Podman-Container-Checkpoint — Podman Documentation*. URL: https://docs.podman.io/en/stable/markdown/podman-container-checkpoint.1.html (visited on 07/04/2025).