Cargo-Cult Containerization: A Critical View of Containers in Modern Software Development

Tommi Mikkonen[1], Cesare Pautasso[2], Kari Systä[3], and Antero Taivalsaari[3,4]
[1]University of Jyväskylä, Jyväskylä, Finland, tommi.j.mikkonen@jyu.fi
[2]USI, Lugano, Swizerland, cesare.pautasso@usi.ch
[3]Tampere University, Tampere, Finland, kari.systa@tuni.fi
[4]Nokia Bell Labs, Tampere, Finland, antero.taivalsaari@nokia-bell-labs.com

***Abstract*—Software is increasingly developed and deployed using containers. While the concept of a container is conceptually straightforward, there are various issues to be considered while using them, ranging from technical details inside containers to the orchestration of containers that jointly form a meaningful application. In recent years, the use of containers has become so prevalent that developers have a tendency to resort to *cargo-cult containerization* – ritual adherence to the use of containers just because so many others are doing the same thing. In this paper, we study advantages and downsides of containers in modern-day software development. We foresee the use of containers to spread into new areas, including IoT systems and embedded devices. At the same time, we caution against indiscriminate use of containers, since excessive containerization can have adverse impacts on software maintenance and overall complexity of a system architecture.**

***Index Terms*—Software design, design principles, continuous software engineering, DevOps, containerization, software containers, container orchestration**

I. Introduction

A famous software engineering aphorism – usually attributed to Butler Lampson or David Wheeler[1] – says that "*all problems in computer science can be solved by adding yet another level of indirection*". While this aphorism is not entirely based in reality – after all, *performance* problems in computing are usually better solved by *removing* layers of indirection – the evolution of computing has undeniably reflected this well known aphorism in the past decades.

After the advent of the World Wide Web in the 1990s and the subsequent introduction of the Software as a Service (SaaS) model, traditional shrink-wrapped software applications have been largely replaced by web-based software services. Such services and applications can be used without conventional installation or manual upgrading, making it possible to deploy software effortlessly faster and on a much larger scale than before. At the same time, computing has become increasingly virtualized, with multiple layers of virtual machines and abstraction levels layered on top of each other. Even though personal computers such as laptops are still in widespread use, they are more and more commonly used only as a host platform for the web browser that allows us to "peer into" those numerous services that we use in our daily lives. Most of those services are nowadays hosted in the cloud.

Application
Programming Language Runtime
Container
Container Engine
Guest Operating System
Virtual Machine
Hypervisor
Host Operating System
Hardware

Fig. 1. Layers of Indirection.

Today, a typical computing environment consists of a remarkably large number of layers of indirection (Figure 1). For instance, even if one is building a relatively simple backend service, the common solution nowadays is to (1) write it in a dynamic language that requires a virtual machine (such as Python, Java or JavaScript/TypeScript when using Node.js). This service is commonly (2) dockerized and then (3) hosted in a container management system such as Kubernetes (K8s). The entire system is then (4) run in virtual machines that are rented from a third-party service provider, such as Amazon Web Services or Microsoft Azure. Moreover, underneath the hood, the third-party service provider may utilize multiple additional levels of virtualization that are invisible to the developers of the service.

In essence, this means that in modern backend services there are *at least four levels of virtualization until the software actually meets bare metal*, discounting the fact that there may be additional virtualization layers also at the hardware

[1]https://quotes.yourdictionary.com/author/quote/585289

level [1]. While there is nothing inherently wrong in this – after all, as long as ample computing power and network capacity is available, those added layers do not have any perceivable impact from the end user's perspective [2] – the developer implications of this approach deserve some additional consideration. These implications have been largely ignored in software engineering research so far.

The multi-layer stack described above – dynamic development language, Docker and Kubernetes running on rented virtual machines – has effectively become the *de facto* model for software development and operations. In many ways, this approach has become so prevalent that it exhibits the signs of *cargo-cult programming* – ritual adherence to a certain way of doing things model just because so many others are doing the same thing, disregarding the original problem and context in which the solution is meant to be applied [3]. According to our industry observations and discussions with various people in the industry and academia, this model is nowadays commonly expected to be used even for simplest applications that might otherwise be run or hosted directly on personal laptop or desktop computers. We refer to such ritual behavior informally as "*cargo-cult containerization*".

In this paper, we take a critical view of containerization. We discuss the emergence of containerization so far, summarize its benefits and take a look at the darker side of containers as well. We caution against indiscriminate use of containers since excessive containerization can have adverse impacts on software architecture, maintenance and to some extent also security and performance. The paper is based on our collective experience of well over a hundred years in the software industry and in the academia, as well as practical hands-on development efforts while creating both academic and industrial applications and services. While we consider ourselves as ardent supporters of container-based software development and deployment, we are increasingly baffled by the fact that this approach is nowadays applied almost blindly to use cases and systems that do not benefit from this approach. To us, this seems like a classic "*if all you have is a hammer, everything looks like a nail*" scenario.

II. The Emergence of the Container Era

A Brief History of Containerization. The early history of containers began with the gradual development and evolution of enabling technologies such as *cgroups* (control groups) in 2006[2] and *namespaces* that originated in the Linux operating system in the 2.4.19 kernel beginning in 2002. These foundational technologies were integrated within the Linux Container runtime project (LXC)[3] originally announced in 2008. LXC version 1.0.0 was released in 2014. LXC provided a complete system for lightweight virtualization.

The introduction of *Docker Containers* (https://docker.com) in 2013 was a cornerstone in the adoption of the container approach [4]. It introduced the integration of prior developments with a set of tools and common packaging format. The popularity of Docker made containers the *de facto* approach for lightweight virtualization [5]. It should be noted that Docker is not the only way to use containerization, though. Alternative technologies include systems such as *LXD* (built on top of LXC), *Podman* (daemon-less), *Containerd/RunC*, and some others.

As the creation of lightweight virtual machines became effortless, developers started to embrace this approach widely to decompose their applications into containers. This led to increasing popularity and extensive use of Docker orchestration systems, including *Docker Compose* and *Docker Swarm*.

The introduction of the large-scale container orchestration tool *Kubernetes* (https://kubernetes.io) in 2014 was another important milestone in the evolution of containerization. The origin of Kubernetes is at Google where it was introduced and used for managing very large container-based systems.

As the popularity of containers increased, many cloud technology providers (such as OpenStack) strengthened their support for lightweight containers. Kubernetes has been included in major cloud provider offerings: Amazon Web Services provides *Amazon Elastic Kubernetes Service* (AWS EKS), Microsoft Azure provides *Azure Kubernetes Service* (AKS), and Google provides *Google Kubernetes Engine* (GKE).

Docker used to be one possible container technology supported by Kubernetes. However, in December 2020 Kubernetes announced the deprecation of the Docker runtime in favor of containers that are compliant with the *Container Runtime Interface* (CRI) (see https://kubernetes.io/blog/2020/12/08/kubernetes-1-20-release-announcement/).

III. Advantages of Containerization

The emergence of the container technologies has brought various advantages. Below, we list the most important benefits categorized under different technical viewpoints.

1) Architectural benefits: From an architectural viewpoint, the key benefits of containerization include the increased modularity and reduced dependency on traditional physical computing architectures. In essence, in container-based systems the traditional computer as a target has disappeared. Instead, the system is composed of a number of containers that can be deployed flexibly into different types of environments. As a result of this evolution, containers themselves have effectively become *the* platform.

More broadly, virtualization provided by the containerization has value of its own. The developers want to control and freeze the used platform components. The runtimes, middleware components and libraries often evolve so rapidly that the developers need means to control the development, deployment and runtime environments. Containers provide a natural boundary for such architectural evolution.

Simply put, container systems allow distribution of the application logic to several containers and provide scalability by introducing concurrent instances of the containers. Container technologies also provide means to organize of the containers to sub-network with controlled interfaces to each other. Furthermore, each of these can evolve and be deployed

[2]https://lwn.net/Articles/236038/

[3]https://linuxcontainers.org/

independently of the other containers, provided that interfaces remain unchanged.

2) Design benefits: In terms of design, containers can serve as the natural scope for designing systems at team level, as well as making it easier to split the development work among different subteams pretty much independently of the final deployment and computing architecture. Inside individual containers, (sub)teams or individual developers can flexibly decide and choose the preferred development languages, frameworks, libraries and tools. More broadly, containerization can reduce the complexity of cloud-native applications and increase portability through well-established abstraction boundaries. This, together with better support for automation, can increase the efficiency of the development organizations and simplify the development of distributed applications.

An additional important design benefit is that containers *serve as a platform for iterative (and portable) design*. Since it is easy to replicate container-based system deployments in different contexts without major dependencies with specific physical computing platforms, it is safer to perform experimentation in the design phase or make experimental versions of system components available to other team members or external partners for further development and experimentation.

3) Benefits for data persistence: In container-based systems, each container is typically responsible for its own data. Hence, it is easier to avoid inter-dependencies related to data in different containers. Among other things, this makes it possible for each container to recover from possible faults more independently. Furthermore, in such a system there is no need to agree on global data formats, apart from those APIs/data formats that are used for interfacing with services in other containers.

4) Benefits for testing: In container-based systems, containers are a natural subject for unit testing. Since the goal of techniques often associated with containers is to produce systems that form meaningful operational entities or subsystems on their own right, they can be tested meaningfully and in most cases independently of each other. Furthermore, such testing can be automated to a large degree.

5) Deployment benefits: Established trends – such as microservices [6] and continuous integration, delivery and deployment [7] – encourage the use of separate containers to remove dependencies between the subteams [8]. In many ways, containers are the natural mechanism to support microservice development, continuous deployment and automation, and DevOps more broadly. The ability of containers to isolate units of deployment from specific physical computing architectures and physical computers plays an essential role in enabling this (Figure 2).

These capabilities have given rise to advanced orchestration systems such as Kubernetes (https://kubernetes.io/). Kubernetes can be seen as a "Swiss army knife" that provides automated policy-based scaling and fault tolerance. It also supports global deployment approaches: scaling-out across different clusters and geographical locations. Hence, automated container orchestration for scaling to unpredictable workloads and fail-over clustering becomes possible. In traditional physical computing platforms, such flexible orchestration would have been virtually (and quite literally) impossible.

Given the ability of containers to serve as a unifying deployment mechanism across different types of physical computing architectures, we foresee containers as an important enabler for *isomorphic IoT systems* [9], i.e., Internet of Things systems in which various elements in the end-to-end system – devices, gateways and various cloud components – can be developed and deployed with a consistent set of technologies. Although containers can still be viewed as an overkill for resource-constrained IoT devices built on low-power MCUs, we anticipate the use of containers to spread also to the IoT domain in the coming years.

6) Runtime benefits: Related to architectural and testing benefits mentioned above, containers can also serve as a meaningful unit for starting, stopping, resetting and restarting various subsystems independently. This can simplify recovery operations when something goes wrong in a particular subsystem, as well as the rejuvenation of long-lived services [10]. In addition, this supports the deployment of new versions of subsystems so that different versions can be tested and run in parallel with different containers in order to check that the new version is behaving correctly [11].

7) Performance benefits: Although containerization itself usually reduces system performance to some extent (see Section IV), containers make it possible to optimize the behavior of the system as a whole in a fashion that would not have been possible in traditional software architectures. Several proposals have been made to optimize the behavior of a container-based system as a whole – for example, monitoring both infrastructure and application-level key performance indicators [12]. Overall, containers (and associated orchestration systems) can make it considerably easier to scale the system as the usage of the system increases. In addition, policy-based optimization allows automigration and automatic configuration changes.

8) Security benefits: Because containers are meant to be self-contained, security and governance of cloud-native applications is increased when using containers. However, the use of containers still requires careful consideration and awareness of risks and certain best practices.

Ultimately, the popularity of containerization and container-based software development is driven largely by *developer convenience*. Containers make it easier for individual software developers and small teams to create, deploy and maintain large-scale software systems with considerably less manpower than was required in traditional projects. Effectively, containerization is part of a larger phenomenon in which *increased virtualization has resulted in a considerably lesser role for traditional IT departments*, empowering individual developers to accomplish tasks which traditionally required an entire IT department.

IV. Downsides of Containerization

So far, the march towards containerization has progressed without much criticism. However, over-reaching use of con-

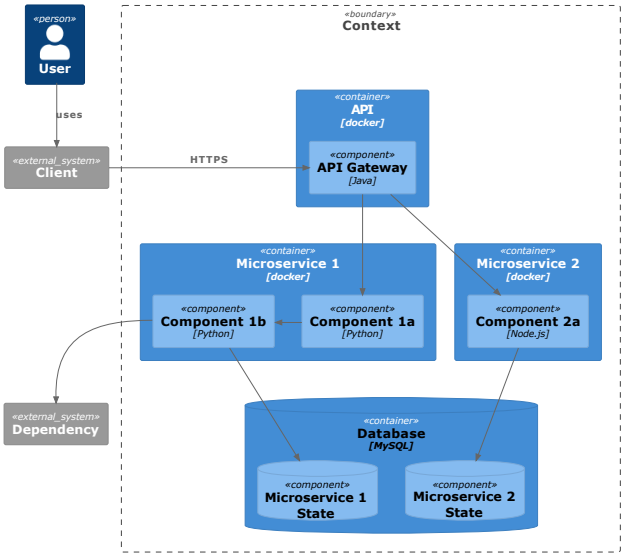

Fig. 2. C4: Context, Containers, Components (and Classes) [13].

tainers can have negative effects on software development and design. Below, we address these topics using the same categorization as before.

1) Architectural disadvantages: Somewhat ironically, the most significant architectural disadvantages of containerization are not related to container *technologies* at all. Rather, containerization has become so successful and prevalent that it has effectively become a *substitute (or synonym) for software architecture and design*. According to our observations, many developers (and especially managers) today assume that as long as they use Docker and some orchestration solution such as Kubernetes, they already have ”software architecture” in place. However, in practice containers are only a lightweight virtualization technology that by no means itself guarantees a meaningful software architecture or design.

In general, only those applications that are designed to run as a set of discrete microservices – with meaningful and well-defined interfaces – stand to gain from containerization. Otherwise, the only real benefit of Docker and similar tools is that they can simplify application delivery and deployment by providing an easy packaging mechanism and virtualization from physical computing architectures. Furthermore, as summarized in the design benefits paragraph earlier, one can easily keep expanding one's own container, using it as developer team's sandbox in which it is easy to experiment and add things without negotiating with other teams or considering the big picture of the architecture – in other words, which logical components belong to which container.

2) Design disadvantages: The general design disadvantages of containerization are closely aligned with the architectural observations above. It is important not to naively equate the use of containers with good design, or assume that containerization alone would yield great system designs.

At first sight the promise of containers providing design encapsulation is indeed attractive – the development team can find a solution of its own, and they are not bound by the decisions of any other teams. However, this flexibility can have its downsides, too. In particular, not having to know the environment in which an individual subsystem runs or its internals can spur opportunistic reuse [14]: an approach in which software components are picked for reuse without any deeper understanding of the components or the context to which they were originally designed – just blindly pick something seemingly applicable and amalgamate that into a larger whole disregarding its origins and the possible dependencies and legacy that this might introduce to the system. While conflicts between dependencies do not have to be solved globally, since each container isolates the necessary add-ons within its boundaries, there is a lack for a global container in which to place shared functionality that is common to the entire system.

As teams gain increased control over their own containers, in the long run such evolution can lead to design conventions that are only associated with a particular team and the containers they work on. In the worst case, some of these conventions and practices will drift apart and are no longer in line with the design ideology of the rest of the system.

3) Disadvantages for data persistence: Because each container is an independent subsystem, handling of permanent storage requires special attention. By design, all of the data inside an individual container will disappear forever when the container shuts down unless the data have been saved externally somewhere. If an application is designed with containers, container systems require the use of solutions such as Docker volumes. However, Docker volumes can lead to complexities when they are run in cloud environments, requiring the developers to worry about consistency/availability/performance tradeoffs that are characteristic of distributed applications [15].

Additionally, partitioning the state of an application across multiple containers can make it impossible to recover a consistent snapshot of data unless all the containers are backed up at the same time [16].

4) Disadvantages for testing: Because each container is a distinct entity that can evolve independently of other containers, the eventual system that is constituted by all the latest but compatible versions of its containers at a given time can become very expensive to test in a dedicated staging context. Instead, the most natural way to test end-to-end system integration is to go live with the production system. This requires that the system operates according to chaos engineering principles [17] and that it can run a rapid fallback operation to recover from the introduction of a failing container.

Testing of containerized systems is further hampered by the fact that logs and system consoles can be much more tedious to locate. This is especially true of those containers that are run inside orchestration systems such as Kubernetes. In systems that are deployed onto physical servers or traditional virtual machines, system consoles and logs are almost trivial to access in comparison.

5) Deployment disadvantages: One of the key benefits of containers is the dramatically reduced dependence on specific physical target platforms. With containerization, it is also easier to automate deployment onto different types of target environments. This has led to the demise of traditional IT departments that were necessary for managing data centers and physical computing infrastructure. The downside of this phenomenon is the increased burden on developers themselves [18]. Containers are typically tied to build and deployment pipelines and managed by the development team instead of the operations or IT department. This increases competency requirements for the development team significantly [19].

Another complexity associated with container deployment pertains to their orchestration. There are various orchestration tools such as Docker Swarm, Kubernetes and Mesos; deciding which is the best suited alternative for a particular use case is not a straightforward selection [20].

While orchestration tools can help tame the complexity of continuously evolving software systems, they can be too powerful for slow-moving software developed and maintained by small teams. In this context – due to lack of knowledge and skills – a temptation emerges to pick some orchestration strategy and settings in an opportunistic, cargo cult fashion, without really understanding the impact and limitations on the chosen deployment strategy.

6) Runtime disadvantages: To tame the complexity of containers, it is crucial to monitor them for performance, availability and security issues. A variety of monitoring tools and external monitoring services and analytics can help address this challenge. Considering the complex nature of the cloud environments, in-depth monitoring of security issues is important.

It should be noted that in the presence of orchestration systems such as Kubernetes, mere knowledge of base operating system level performance and monitoring tools does not suffice, since in orchestrated systems runtime components and processes cannot be accessed directly. Rather, developers must be familiar with tools that are specific to the orchestration system. This further increases the educational needs for developers.

7) Performance disadvantages: Since containerized applications use underlying hardware directly, they are often assumed not to use extra resources. This assumption has been confirmed to be false, e.g., by measurements presented in [21], [22] and [23]. These studies indicate that even lightweight virtualization can have a noticeable effect on performance.

As such, container technologies do not increase memory footprint. However, when the deployed containers introduce different versions of the runtime and middleware from the host, they do increase memory footprint rather significantly. In case the systems are based on multiple containers, this problem is multiplied.

8) Security disadvantages: Compared to traditional operating system virtualization, containers provide less security – but still more security than with no virtualization [24]. Compared to a traditional stack, containers require multi-level security as they consist of multiple layers. In addition to the security of the containerized app, the container registry, the Docker daemon and the host operating system need to be secured as well [25]. Moreover, the container image must be trusted, i.e., there must be assurances in place to guarantee that it has not been subjected to a supply-chain attack.

9) Disadvantages on graphical applications: It should be noted additionally that Docker was designed as a solution for deploying server applications that do not require a graphical interface. While there are some creative strategies (such as X11 video forwarding or VNC [26]) to run GUI applications inside a container, these solutions are clunky at best. For UI-intensive applications, containerization does not generally work well.

We have summarized the container advantages and downsides in Table I that presents our observations and key points in condensed form.

V. Conclusions

In recent years, containers have emerged as a practical solution for modular software design, development, deployment and operations. In fact, containerization has become so successful and prevalent that it has effectively become a substitute or synonym for good software architecture and design. Today, many developers and managers tend to assume that as long as their projects use Docker and a popular container orchestration system such as Kubernetes, they already have the necessary "software architecture" in place. The assumption that containerization almost automatically provides good software architecture has resulted in rather indiscriminate use of containers and orchestration technologies even in contexts where they really are not needed. We refer to this phenomenon as "*cargo-cult containerization*" – ritual adherence to a certain way of developing software just because so many others are doing the same thing.

In this short paper, we have taken a critical look at containerization. We provided a brief history of containerization and container technologies, followed by an analysis of their benefits and drawbacks. A forthcoming, more comprehensive journal version of this short paper will provide a deeper analysis of the tradeoff between the flexibility afforded by containers against the complexity that they inevitably introduce. We are currently working on gathering concrete guidelines and recommendations for distinguishing between merely ritual vs. properly motivated use of containerization.

References

[1] J. Frazelle, "Chip measuring contest: The benefits of purpose-built chips," *Queue*, vol. 19, no. 5, p. 5–21, oct 2021. [Online]. Available: https://doi.org/10.1145/3494834.3501254

[2] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 17, p. e5668, 2020.

[3] R. P. Feynman, E. Hutchings, and R. Leighton, *Surely you're joking, Mr. Feynman! - Adventures of a curious character*. W. W. Norton, 1997.

[4] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

Concern	Advantages	Disadvantages
Architecture	Provides architecturally meaningful ways to delimit development and deployment boundaries.	There is a requirement to determine which logical software components are packaged and deployed in which container
Design	Provides a natural scope for designing systems at team level.	Can result in tangled designs, where the containers get bloated because of all the tasks allocated to a container.
	The team has all the control in its own hands inside each container. Being the owner of the container, the team can usually decide about best possible tools to use.	Same functionality can be implemented differently in different containers. The team may create code that is not well in line with the container ideology, just because they follow their own guidelines and practices.
Data Persistency	Containers isolate data with high granularity.	Storing persistent data requires additional means, usually copying data across containers.
Testing	Containers are a natural subject for unit testing, and this can be automated to a large degree.	System-wide end-to-end integration testing can become problematic, if and when containers change constantly in a parallel fashion. Logs and system consoles not always easy to access.
Deployment	Containers are the mainstream deployment mechanism for continuous software engineering and DevOps. Additionally, they can provide isomorphism to systems with heterogeneous computing devices.	The use of containers can introduce delays in build and deployment phases.
	Orchestration simplifies deployment at container level, and scales well to different use cases.	Orchestration tools are often so complex that developers resort to cargo-cult copying of parameters and tools that are often too complex for the task at hand.
Runtime	Containers can be restarted to reset certain subsystems.	Containers need continuous monitoring for possible problems.
Performance	Faster initialization and shutdown, w.r.t. virtual machines	Increased memory consumption and non-negligible inter-container communication overheads
Security	Provides a basic isolation sandbox	Not as isolated as other virtualization techniques

TABLE I
ADVANTAGES AND DISADVANTAGES OF CONTAINERS.

[5] K. Matthias and S. P. Kane, *Docker: Up & Running: Shipping Reliable Containers in Production*. O'Reilly, 2015.

[6] S. Newman, *Building microservices*. O'Reilly, 2021.

[7] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[8] C. Pahl, P. Jamshidi, and O. Zimmermann, "Microservices and containers," in *Software Engineering 2020*, M. Felderer, W. Hasselbring, R. Rabiser, and R. Jung, Eds. Bonn: Gesellschaft für Informatik e.V., 2020, pp. 115–116.

[9] T. Mikkonen, C. Pautasso, and A. Taivalsaari, "Isomorphic iot architectures with web technologies," *IEEE Computer*, vol. 54, no. 7, pp. 69–78, 2020.

[10] M. Torquato and M. Vieira, "An experimental study of software aging and rejuvenation in dockerd," in *2019 15th European Dependable Computing Conference (EDCC)*, 2019, pp. 1–6.

[11] D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun, and M. Stocker, "Interface evolution patterns: balancing compatibility and extensibility across service life cycles," in *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2019, pp. 1–24.

[12] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," *The Computer Journal*, vol. 62, no. 2, pp. 174–197, 2019.

[13] S. Brown, *Software architecture for developers*. LeanPub, 2013.

[14] N. Mäkitalo, A. Taivalsaari, A. Kiviluoto, T. Mikkonen, and R. Capilla, "On opportunistic software reuse," *Computing*, vol. 102, no. 11, pp. 2385–2408, 2020.

[15] N. G. Bachiega, P. S. L. de Souza, S. M. Bruschi, and S. d. R. S. de Souza, "Performance evaluation of container's shared volumes," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020, pp. 114–123.

[16] G. Pardon, C. Pautasso, and O. Zimmermann, "Consistent disaster recovery for microservices: the BAC theorem," *IEEE Cloud Computing*, vol. 5, no. 1, pp. 49–59, 2018.

[17] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[18] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[19] A. Taivalsaari, T. Mikkonen, C. Pautasso, and K. Systä, "Full stack is not what it used to be," in *Proc. of the 21st International Conference on Web Engineering (ICWE 2021)*, ser. LNCS, M. Brambilla, R. Chbeir, F. Frasincar, and I. Manolescu, Eds. Springer, 2021, pp. 363–371.

[20] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *Proc. of the 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–6.

[21] S. Shirinbab, L. Lundberg, and E. Casalicchio, "Performance evaluation of container and virtual machine running cassandra workload," in *2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*, 2017, pp. 1–8.

[22] D. Beserra, E. D. Moreno, P. T. Endo, and J. Barreto, "Performance evaluation of a lightweight virtualization solution for hpc i/o scenarios," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2016, pp. 004 681–004 686.

[23] A. Kovács, "Comparison of different Linux containers," in *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*, 2017, pp. 47–51.

[24] X. Wang, J. Du, and H. Liu, "Performance and isolation analysis of runc, gvisor and kata containers runtimes," *Cluster Computing*, pp. 1–17, 2022.

[25] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem–vulnerability analysis," *Computer Communications*, vol. 122, pp. 30–43, 2018.

[26] V. Mittal, L.-H. Hung, J. Keswani, D. Kristiyanto, S. B. Lee, and K. Y. Yeung, "GUIdock-VNC: using a graphical desktop sharing system to provide a browser-based interface for containerized software," *GigaScience*, vol. 6, no. 4, 02 2017, giw013. [Online]. Available: https://doi.org/10.1093/gigascience/giw013