A comparison of hosting techniques for online cybersecurity competitions

Niccolò Maggioni[0009−0009−3901−8842] and Letterio Galletta[0000−0003−0351−9169]

IMT School for Advanced Studies Lucca, Lucca, Italy
{niccolo.maggioni, letterio.galletta}@imtlucca.it

Abstract. Online cybersecurity competitions have gained significant traction in the community for educational and evaluative purposes because they offer dynamic environments for learning intricate technical concepts in an engaging, non-traditional, and interactive manner. While such competitions are increasingly popular and frequently held, organizing them is not straightforward. Developers must design challenges that are both innovative and balanced in difficulty, ensuring an enjoyable learning experience. Concurrently, effective hosting, configuring, and managing the online infrastructure that underpins these games are technical challenges demanding informed decisions. Unfortunately, no comprehensive resource exists to guide organizers in choosing suitable hosting architecture and tools based on their technical proficiency and event scale. This paper contributes to addressing this gap by examining prevalent architectures for hosting Capture The Flag (CTF) competitions and evaluating them using criteria encompassing several factors. These factors include environment setup, deployment, configuration, and updates of vulnerable services, system maintenance, and security mechanisms. Each criterion is qualitatively assessed with an associated numeric score. Finally, the paper puts forward recommendations on architecture and tools based on event size and organizer skills.

Keywords: Cybersecurity games · Capture the Flag · Jeopardy CTF competitions · Hosting architectures · Hosting recommendations.

1 Introduction

Recently, cybersecurity games, especially Capture The Flag (CTF) competitions, have gained significant popularity within the computer security community for educational purposes and evaluation objectives. These competitions are widely considered excellent methods for learning deeply technical concepts in a fun, non-traditional, and interactive environment. They cater to a diverse range of skill levels, accommodating beginners, practitioners, and even experts. Participants are usually organized into teams that compete for a prize or recognition. In a typical competition, players face a series of hacking challenges presented as vulnerable remote services. To solve a challenge, players must find an attack to reveal a secret piece of information, which is then submitted to a scoring server.

These competitions are predominantly hosted online, last a limited period of time, and attract thousands of participants worldwide. The player community is remarkably active, with new competitions occurring on a weekly basis, as documented by the reference portal [6].

However, despite the weekly frequency of new competitions, organizing them is far from a straightforward task. On the one hand, organisers should prioritise participants' game experience: they should design challenges that are both innovative and enjoyable, striking a balance between difficulty and accessibility, and that facilitate players to learn new cybersecurity concepts or subjects. Achieving such an engaging game experience demands considerable dedication from challenge developers. On the other hand, organisers must meticulously handle the design and implementation of the online infrastructure that provides the game environment to participants. A minor mistake in its setup, configuration, or management can lead to malfunctions during the competition, detrimentally affecting players' enjoyment and overall perception of the event. While crafting engaging challenges remains an art honed by experienced developers, effectively hosting, configuring, and managing the game infrastructure is a technical matter that necessitates well-informed decisions during competition design. Although the common choice is to opt for a public cloud environment to host the CTF event, organisers often find themselves overwhelmed by the numerous architectures and tools available. Unfortunately, these options are frequently inadequately documented or are only available as informal articles scattered across the web [23,14,45,49,13]. To the best of our knowledge, a comprehensive literature that assists organizers in choosing the appropriate architecture and tools according to their technical expertise and the scale of the event is currently missing.

This paper contributes to tackling this issue by examining the prevalent architectures for hosting CTF competitions and conducting a qualitative assessment based on well-defined criteria. These criteria draw inspiration from those typically employed for assessing cloud platforms and encompass diverse facets, including environment setup, packaging, deployment, configuration, and update of vulnerable services, system maintenance, as well as security mechanisms for isolating processes and limiting their resource usage. We furnish a qualitative judgment that elucidates to which extent the architecture meets each criterion alongside a numeric score that synthesizes our evaluation. Additionally, we assign an aggregate score to each architecture, computed as the cumulative sum of criterion scores. The analysis of these scores shows that the architectures providing a certain level of automation in deploying and managing vulnerable services are the most suitable for adoption in a context where CTF organisers have no prior experience or infrastructural preferences. Subsequently, we offer a set of recommendations about the architecture based on the event size and organizers' technical skills. We believe that the outcomes of our analysis can assist CTF organizers in selecting the most appropriate architecture.

In summary, the main contributions of this paper are:

- We examine the prevalent architectures used to host vulnerable applications in the context of CTF competitions.

- We present an evaluation methodology and a set of criteria for assessing different architectures. These criteria encompass various factors, including environment setup, packaging, deployment, configuration, and updating of vulnerable services, system maintenance, as well as security considerations such as isolation capabilities and resource usage limitations.
- We assess each architecture according to the introduced methodology, providing a qualitative judgment alongside a numeric score for each criterion. Our analysis indicates that the architectures providing a certain level of automation in deploying and managing vulnerable services are the most suitable for adoption.
- We provide a list of recommendations to CTF organisers on the architectures to use based on the scale of the event and their technical skills.

The rest of the paper is organised as follows. Section 2 introduces the world of CTF competitions. We present the prevalent architectures to host such competitions in Section 3. Section 4 introduces our methodology and the criteria we adopt for the assessment, which is carried out in Section 5. We discuss the results of our assessment in Section 6, and we furnish recommendations to CTF organisers in Section 7. Section 8 compares our work with the relevant literature, while Section 9 draws some conclusion and discuss future work.

2 Background

This section introduces CTF competitions and presents the requirements organisers need to meet to host them.

2.1 What CTF competitions are

CTF events are computer security competitions. Participants engage in a series of hacking challenges consisting of remote services that are designed to contain security flaws and vulnerabilities, e.g., buffer overflow, Cross-Site Scripting (XSS), with the goal of uncovering a concealed piece of information known as the "flag." Flags consist of a string of characters with a known syntax, serve as evidence of successful exploitation, and allow earning a certain amount of points. Typically, participants can either work alone or in teams, and the competition is won by those who accumulate the largest amount of points. At the start of the competition, each challenge is worth a predetermined amount of points by the event organizers. In most competitions, this value decreases exponentially as more and more participants solve the same challenge: this ensures that the most difficult challenges retain a high points value and thus enable stronger players to climb the leaderboards.

There are two types of widely known CTF events: *jeopardy* and *attack-defence*. The first format exposes a list of challenges to the players, who solve them at their own pace and usually without interacting with other players - at least not in a real-time manner. Some challenges may require to be solved in

a specific order, but players are generally free to approach them in any order they prefer and to concentrate on those that they feel are the most interesting. Solving all the challenges is considered a noteworthy achievement but not a requirement. Concerning the kind of challenges, jeopardy CTFs follow a standard categorization where the prominent categories include: Binary Exploitation, Cryptography, Forensics, Reverse Engineering, and Web Security. However, the organizers of each event are free to tweak them to their needs and add new - or remove existing - ones if needed. For example, many events propose challenges concerning smart contract technology or hardware. Jeopardy CTFs typically run for two to three days, with some special events lasting up to a few weeks.

The attack-defence format is instead oriented on real-time interactions between teams of players across the network. In this type of competition, each team is either granted access to a remote machine or is assigned the task of hosting it on their premises. These hosts, referred to as "vulnboxes," are connected to the game infrastructure, and operate using pregenerated OS template images that the CTF organizers have equipped with vulnerable software. These applications are the same for all teams and typically range in number from three to six. The competition runs over a series of rounds called "ticks" that typically last from 60 to 180 seconds. At the start of each round, a central component of the game infrastructure managed by the organizers, the *game server*, interacts with each team's services and verifies that they are working as expected. If the check succeeds, the game server adds new flags to the service. In the meantime, players must exploit the opponent's services to extract the flags using a combination of manual proof-of-concept exploits and, subsequently, automated attacks. The network structure is designed to make it impossible for players to determine whether the network traffic is coming from the game server or another player, forcing them to accurately inspect each packet to identify, block, and eventually replicate malicious exploits.

This paper focuses on requirements for jeopardy CTFs since they are the most common type of event on the scene [6]. Organizers who host attack-defence competitions usually have extensive domain-specific knowledge and build their infrastructure according to their needs, technical preferences, and availability of computational resources.

2.2 Requirements for jeopardy CTFs

A typical jeopardy CTF needs the following software components to be functional and properly organized:

- A public website that allows players or teams to register, access the various challenges when the competition is running, and see the scoreboard with the scores of the various teams. Currently, CTFd [5] is the most popular choice, followed by the relatively newer rCTF [44]; however, teams are free to adopt custom solutions.
- A real-time messaging platform to let the participants communicate with the organizers; currently, Discord is the most common solution, but other approaches that allow direct communication are fine.

- One or more remote hosts serving the competition website and the challenges. Hosting both on the same host is typically not recommended since possible problems with the challenges could damage the website availability or even the scoring data.
- A set of challenges organized into various categories: each challenge can be either static, such as a file to be downloaded or other static assets served through the Internet, or dynamic, such as an entire application responding to the users' inputs remotely.

There are also other factors to consider when organising a CTF, for example:

- Organisers should adequately balance the difficulty of the challenges. As a practical rule, every participant should solve at least one challenge, but only a small portion of teams should solve the most difficult ones.
- Challenge developers must have a clear and precise understanding of the flaws they introduce and the consequences of possible exploits; failing to do so could pave the way for malicious exploitation of the platform. Examples of issues may include unsafe Inter-Process Communication (IPC), the lack of isolation for expected Remote Command Execution (RCE) attacks, and the absence of Denial of Service (DoS) and scraping filters.
- The available budget for organising the event is usually limited: bigger events can obtain sponsorships from cloud providers or companies active in the cybersecurity field, but most of the time, the events are self-funded by the organizing team. The estimation of the necessary budget must take into account that the environment must be set up ahead of time to allow the participants to register on the website (one month is enough for most events), and that it is customary to leave the website with the scoreboard and challenges online for some time after the end of the competition (typically from one to three weeks) to allow participants to test and document their solutions and improve their skills properly.

3 Hosting architectures for CTFs

Here, we briefly survey the most common approaches to hosting vulnerable applications in the context of a CTF. We list below the architectures ordered by their level of complexity and required administrative specialization. Their adoption is documented in the literature [22,40] or in articles that CTF organizers have informally published throughout the years [23,14,45,3,49,13].

Dedicated server (legacy deployments) We have a bare metal server or a virtual machine fully dedicated to hosting the CTF challenges. The server must be equipped with the required computational resources from the beginning and must be placed in a securely manageable environment. Services are deployed through a standard pipeline: they are developed on a different machine, and then their code (source or binary), together with the required dependencies, are manually copied into the server; setting the execution

environment with all the required configuration parameters is performed manually, as well as restarting the necessary processes. Due to the nature of CTF competitions, the operating system running on this architecture is expected to be a Linux distribution.

Dedicated server (containerized deployments) This is the same physical architecture as above ("Dedicated server (legacy deployments") but with the key difference that vulnerable services and their dependencies are deployed using application containers. These containers typically run on runtimes and toolchains supported by Docker [10], Podman [36], or equivalent software. A more detailed explanation of this approach has been described by Merkel [31]. Due to the nature of CTF competitions and container runtimes, the operating system running on this architecture is expected to be a Linux distribution.

Hyper-converged infrastructure In this case, the infrastructure is made of a group of (physical or virtualized) servers that feature virtualization, storage, and advanced networking capabilities and is managed from a single interface. The concept of individual servers is abstracted as much as possible, giving operators the feeling of managing a single piece of hardware equipped with the union of the resources of the involved machines. Here, we consider Proxmox VE [38] as the reference implementation for this category: it is a browser-based environment that enables users to manage QEMU [39] virtual machines, LXC [27] containers, and various types of virtual networks. This solution is based on Debian GNU/Linux [7].

Simple orchestrator or task runner This architecture consists of software or a combination of applications that assist users in managing the vulnerable services interactively and is characterized by static specifications that describe how to deploy and modify each service. Any changes to the deployment require changes in such specifications. Concretely, these specifications might be configuration files, deployment plans, or other assets similar to human-readable static text files. Here, we consider a combination of HashiCorp Nomad [20] and Consul [17] as a reference implementation for this category: their capabilities allow users to manage and monitor QEMU virtual machines, native Linux containers (LXC and others), application containers (Docker [10], Podman [36]), and standard processes. This architecture is a simple yet powerful starting ground for transitioning to more complex ones.

Complex orchestrator This architecture typically consists of a tightly coupled suite of components that abstracts away the complexities of deploying, scaling, and managing containerized applications. This kind of orchestrator is usually meant to handle large-scale deployments, but its abstraction capabilities can also help in smaller deployments where it is desirable to define precisely the environment in which each vulnerable piece of software must run. In this paper, the reference implementation for this category is Kubernetes [46]. It can orchestrate different loads, but we focus only on application containers here. Moreover, since many features of this approach are only reliable when used in the Cloud, all the observations in the following sections assume that a public cloud deployment has been chosen and that the best practices recommended by the cloud provider have been followed. If a provide-managed

offering is incompatible with the CTF organizers' requirements, it is assumed that they can operate the chosen complex system appropriately.

4 Assessment methodology

We describe our methodology to assess the various architectures to host CTF competitions. In our evaluation, we neglect the hardware or platforms where the deployment is carried out - such as a public or private cloud provider, on-premises or co-located physical servers, or combinations of the two - since these options are numerous and constantly changing. Their management can be considered unavoidable overhead independently of the goal these resources are used for, and thus, it must not affect the analysis of their characteristics.

We extend the methodology by Maiya et al. [28] on classifying the manageability of cloud platforms. More precisely, we revise the proposed use cases and metrics to fit the organization of CTF competitions. Indeed, these events require managing insecure by-design software and executing exploits crafted by players, with relatively few limitations on what can be done once attackers have taken control of the target service or machine. Running vulnerable software inevitably leads to unexpected behavior of challenges during the competition or unplanned and hard-to-diagnose outages of the game platform. Thus, we introduce security-oriented criteria regarding the isolation of running processes and network activities, mechanisms for limiting computational resources, and tools to quickly servicing internal components when they break. Moreover, we simplify the scoring system where each characteristic receives an absolute rating on a scale from 1 to 3: 1 represents the worst value, while 3 is the best one. Thus, the architecture with the higher score will be considered the most convenient in a generic use case, namely when CTF organisers have no particular prior experience or infrastructural preferences. Later in the paper, we provide recommendations for specific use cases and environments that could slightly change these classifications.

Below, we report the criteria we use to score the various architectures:

Initial setup complexity The complexity, in terms of required domain-specific knowledge and experience, of the initial bring-up of the whole architecture. A system with few components and self-explanatory configurations will be considered simpler than one with more components or harder-to-understand configurations.

Initial setup duration An estimate of the time required for the initial bring-up of the whole architecture. Time duration is classified into three segments: *less than an hour*, *multiple hours*, and *multiple days*. A system with a quicker bring-up will be better than one requiring a longer process.

Services packaging The complexity of packaging software for deployment or for making the code running on the developer's machines ready for deployment on the production infrastructure. A specific differentiation must be made between static assets being served directly, either through a persistent

network connection or a single-time download, and dynamic services requiring interaction with players. A system with a streamlined, less error-prone packaging process will be considered better than one with more steps and a higher probability of errors.

Services deployment The complexity of getting packaged software to run on the production infrastructure through specific protocols or definitions. A system with a streamlined, less error-prone deployment process and more readable deployment specifications will be considered better than one with a more involved, less risk-averse deployment process and less readable deployment specifications.

Services updating The complexity of updating and applying patches to either the code of deployed vulnerable services or their configuration deployment. A system with a streamlined, less error-prone updating process will be better than one with more steps and a higher probability of errors.

Services configuration An classification of the capabilities related to adding and editing external configuration parameters for specific services such as environment variables, secret tokens, API keys, passwords, and inter-service dependencies handled separately from the application code. A system that exposes such elements to deployed applications more easily, reliably, and securely will be considered better than one that does so in more complex, unreliable, and insecure ways.

Resources limitation A classification of the capabilities related to placing, monitoring, and enforcing limits on shared hardware resources like CPU, RAM, disk space, and network bandwidth. A system that enables operators to specify more types of limits in more granular ways will be considered better than one that exposes fewer types of limits in coarser ways.

Isolation & security A classification of the security capabilities of the platform and their relative complexity. We consider how the solution allows for the management of exposed interfaces and which mechanisms it provides for services isolation (from each other, from the operating system of the hosting machine, and the network), for network filtering and logging, for protection against scrapers and DoS (*Denial of Service*) attacks, and for rate limiting. A system offering more of these capabilities will be considered better than one offering less.

Changes of state An estimate on the number of ways the solution handles misbehaving services: automatic restarts after crashes, detection and notifications of repeated failures, exponential back-off timers. A system that offers more ways of automatically solving unexpected state discrepancies and more means of notification will be considered better than one that offers less flexibility.

Scaling A classification of the process of improving a service availability or performance through either vertical (adding more resources to a single instance of the service or its hosting node) or horizontal (adding more copies of the service or more hosting nodes) scaling. A more accessible, faster-to-edit, or more flexible and automated scaling system will be considered better than

one with more complex, slower-to-edit, less flexible, and automatic scaling mechanisms.

Introspection and maintenance A classification of the relative complexity of the system (often referred to as "amount of moving parts") and the required competencies for troubleshooting activities. A critical rating factor will be whether operators can use widely known diagnostic tools or more specialized, domain-specific solutions that need to be installed and configured beforehand. A system that allows easy use of commonly available utilities will be considered better than a more complex one that requires custom diagnostics solutions.

5 Evaluation of the proposed solutions

In this section, we provide a qualitative evaluation of the architectures of Section 3 according to the methodology of Section 4. We make a qualitative judgment for each criterion and assign it a numeric score.

5.1 Dedicated server (legacy deployments)

Below, we report the scores together with a brief qualitative judgment. The scores are summarized in Table 1.

Initial setup complexity (Score: 3) Since the physical hardware or a virtual machine is already provisioned, operators need to set up a customized initialization procedure: they need to install an OS, configure the networking, and audit the overall security of the system, e.g., authentication and authorization parameters, remote access and basic firewall rules. While this process can be tedious to execute on multiple machines, it can be easily automated in a modular way through tools like Ansible [2]. This makes the chain of setup operations declarative, repeatable, and possibly idempotent.

Initial setup duration (Score: 2) Independently from the automation of the setup, the time needed to complete these preliminary operations is measurable in a matter of hours: performing the setup manually is time-consuming and error-prone, but writing a good automated solution takes a comparable amount of time.

Services packaging (Score: 1) While most operating systems use their packaging solutions, such as DEB or RPM archives, the time and effort needed to package software in such formats correctly make them unsuitable in a short-lived and highly dynamic environment required by CTF competitions. Thus, software must be distributed through simpler mechanisms, often using bare source code archives. This makes packaging artifacts difficult to version, organize and transfer reliably.

Services deployment (Score: 1) According to the considerations on services packaging above, the deployment process usually relies on a combination of custom scripts, code launchers, and custom decisions about the structure of the filesystem and the management of background processes. This makes deployments unstable, hardly repeatable, and in need of constant manual interventions.

Services updating (Score: 1) In the context of this architecture, updating services is similar to a new deployment. Thus, the considerations above apply.

Services configuration (Score: 1) Organizing a service's configuration files is a task largely left to its developers: as no standards exist in this context, in the best-case scenario, a group of system operators and service developers might agree on a common configuration format (YAML, TOML, INI, CSV, or other custom syntax) and the path in the file system where files are stored (relative to the application directory, or absolute). Moreover, run-time checks that enforce these informal, internal conventions are often missing.

Resources limitation (Score: 1) With no mechanism that supervises the execution of the services, limiting the resources they have access to can be rather difficult and may lead to hard to troubleshooting issues in seemingly unrelated operations inside the service's logic. Relative scheduling priority and coarse CPU usage percentage can be limited using standard Unix and Linux tools like *nice* and *cpulimit*, but limiting other resources such as RAM and network bandwidth cannot be done reliably without recurring to complex features of modern Linux kernels such as *control groups* [30]. Therefore, limiting the consumption of system resources is a complex task.

Isolation & security (Score: 1) Although organizers might choose to dedicate an entire machine or VM to a single service, this usually never happens due to the enormous amount of resources that would be required for running multiple copies of the same operating system and other low-level resources. Similar to what was said above for resource limitation, isolating processes running on the same host is not an easy task: legacy approaches for this task were based on *chroot* in Linux environments and *jails* in FreeBSD [37], but the modern standards rely on Linux cgroups. Note that these approaches have evolved in the containerization techniques and that applying them manually exposes operators to many nuisances and problems already solved in dedicated tooling.

Changes of state (Score: 2) The operating system processes responsible for starting and upkeeping the system itself (called *init systems* and including *sysvinit*, *OpenRC*, *upstart*, *systemd* as the most historically relevant implementations) can be tweaked and extended to take care of custom services as well. They can be configured to start services on boot, restart them in case of crashes, and redirect their logging to appropriate facilities such as system journals or dedicated log files. While these init systems can be extended to various degrees, they are still limited by their designs, and any additional capabilities such as interactive notifications to operators, checking if the service is replying properly to user requests, or handling misconfigurations must still be implemented manually by the service developers.

Scaling (Score: 2) The aforementioned init systems can be configured to run multiple instances of the same service. But this is an immediate solution only for *stateless* services that either do not hold a state in their memory or rely on an external source of data, e.g., a shared database, disk, or memory area, for all their features. *Stateful* services that need keep some context information in the memory, on the other hand, usually need more complex

Table 1. Scores assigned to the "Dedicated server (legacy deployments)" architecture.

Criterion	Score
Initial setup complexity	3
Initial setup duration	2
Services packaging	1
Services deployment	1
Services updating	1
Services configuration	1
Resources limitation	1
Isolation & security	1
Changes of state	2
Scaling	2
Introspection and maintenance	3
Total score	18

mechanisms to be run in parallel: for example, they might need external *load balancers* that multiplex network connections or other similar devices to operate correctly without providing users with incoherent data. In this kind of architecture, the implementation of such mechanisms is entirely up to the operators of the competition and to the developers of the challenges. This is a burden, however, custom implementations have the advantages that they can be fully inspected, controlled, and understood.

Introspection and maintenance (Score: 3) Although most of the aforementioned characteristics require system operators to intervene manually, and challenge developers to know the production system architecture, the bare-bones dedicated-machine architecture has a great advantage in maintenance: since the system is fully custom-built, whoever built it has complete knowledge of its inner workings. Such internal mechanisms can also only reach a certain upper limit of complexity, after which the operators would have already considered adopting other architectural solutions.

5.2 Dedicated server (containerized deployments)

Below, we list the scores (summarized in Table 2) with a brief judgment.

Initial setup complexity (Score: 3) Setting up a container runtime is usually straightforward: installing a package through the system's package manager is typically enough. For the rest, the same observations of the "legacy" version also apply here.

Initial setup duration (Score: 2) Container runtimes do not usually need extensive configuration processes. For the rest, the same observations of the "legacy" version also apply here.

Services packaging (Score: 3) Each container runtime has its own packaging mechanism, but the most common one is using *Dockerfile* [9]. This declarative format originates from Docker but is supported by Podman as well.

Independently of the used runtime, the container image built from *Dockerfile* is a portable archive that can be easily versioned, transferred, and archived. Depending on the underlying runtime, each newly built image may reuse cached parts of the previous builds and images so as to reduce disk usage.

Services deployment (Score: 3) Deploying a container image is rarely more complex than transferring the image archive to the target machine - usually either manually or through services called *registries* - and issuing the appropriate shell command to run the image with the needed options (environment variables, network port bindings, storage volumes). These options can be tuned during the first deployment and saved for future reference. If the deployment involves multiple services that interact with each other, tools like Docker Compose [11] can help the operators coordinate multiple containers.

Services updating (Score: 2) Updating an existing containerized deployment consists of transferring the new image, stopping the old container, and starting a new one with the same options. This can lead to short periods of time where services are not reachable or are restarting, but they can usually be timed accurately to minimize the impact on users. However, short and infrequent unavailability bursts are usually not a major concern in CTF competitions. The downside of this approach is that if the new deployment silently fails, e.g., the application runs but cannot interact with users, there are no automated systems to revert the changes, potentially leaving the service in a broken state until the operator realizes the problem.

Services configuration (Score: 2) In this architecture, the configuration files for each service must be either embedded into their respective container images or copied to the host file system and made available inside the containers through the facilities offered by the underlying runtime, such as mount points defined at runtime. Comparing this architecture to the aforementioned legacy one, the usage of containers brings forward a common standard regarding the organization of the configuration files: such files can still end up scattered through the filesystem, but it is always possible to infer their location from the mounts and the other options for running a container. However, this convention is not enforceable, and there could be cases where application configuration files are split into two groups: embedded inside the container and mounted from the outside. This approach improves the coherence, but the problem is not solved.

Resources limitation (Score: 2) Typically, container runtimes allow operators to set specific resource usage limits for core resources: CPU, RAM, and sometimes GPU. These options are exposed to operators as simpler abstractions over the mechanisms provided by the Linux kernel - mainly *cgroups* - and, hence, they inherit the same limitations and complexities. A container runtime is not guaranteed to provide all the options (and their combinations) supported by the kernel, but the main use cases are typically always covered.

Isolation & security (Score: 3) Containerized services run as isolated processes by default: they cannot connect to each other, and their memory and disk spaces are kept separate. Each container can be tied to a specific network interface or virtual bridge if needed, reducing its access to local or remote

resources. If a participant of the CTF manages to exploit a containerized service causing an RCE (Remote Command Execution), the scope of that exploit can be limited to the single service's environment. If the targeted service was properly packaged and deployed, the security breach would not easily spread to other components of the competition's architecture.

Changes of state (Score: 1) Container runtimes by themselves have limited capabilities when it comes to detecting faults, restarting crashed applications, and keeping track of these events in general. A typical pitfall of these systems is the handling of fast-failing containers: if a container crashes or stops, the runtime will immediately purge it and start a new one; if this happens immediately after it first gets started, a vicious cycle may start in which the operator does not have enough time to troubleshoot the failing image properly. The operator has then to resort to patching the source code or editing the container's options to override the startup command, usually replacing it with an infinite delay or some sort of *no-op* operation to gain enough time to manually start the service inside the container and investigate the causes of the crash. Some additional tools like Docker Compose introduce health checks, but these mechanisms are often rather limited in their modularity and have their own drawbacks when handling crashes and restarts.

Scaling (Score: 2) Containerizing services make running multiple instances straightforward, but additional tooling is needed to take advantage of the improved load capacity correctly since it still does not follow a standard structure. For the rest, the same observations of the "legacy" version above still apply.

Introspection and maintenance (Score: 2) The complexity of managing a container runtime is usually very low, and standard tools can be installed inside application containers to support troubleshooting. Some peculiarities of the chosen runtime could make troubleshooting certain issues more difficult, typically those related to network and storage resources. For the rest, the same observations of the "legacy" version above still apply.

5.3 Hyper-converged infrastructure

Below, we report the scores with a brief qualitative judgment. Our discussion considers Proxmox as the reference implementation of this architecture. The scores are summarized in Table 3.

Initial setup complexity (Score: 3) The setup of a hyper-converged software solution is typically straightforward and not particularly different from a bare operating system installation. The installation process of Proxmox is simpler than a bare Debian Linux distribution, thanks to its graphical installer, the optimized default configurations, and the selection of preinstalled software packages. In most cases, the system is fully operational and ready to support deployments as soon as the installation process is complete.

Initial setup duration (Score: 3) The time required by the initial setup is comparable to the architectures listed in Section 5.1 and Section 5.2. The setup

Table 2. Scores assigned to the "Dedicated server (containerized deployments)" architecture.

Criterion	Score
Initial setup complexity	3
Initial setup duration	2
Services packaging	3
Services deployment	3
Services updating	2
Services configuration	2
Resources limitation	2
Isolation & security	3
Changes of state	1
Scaling	2
Introspection and maintenance	2
Total score	25

can require less than an hour if the CTF organizers have no special requirements regarding network topology, storage technologies, or high-availability capabilities.

Services packaging (Score: 1) Proxmox supports LXC containers and QEMU virtual machines, both of which do not support advanced packaging techniques. Operators can create LXC container templates similar to Docker images: they can start from a blank environment and install all the necessary low-level components of the chosen Linux distribution rather than specify individual libraries and binaries as dependencies required for the services being developed. The same considerations are valid for QEMU virtual machines. These build processes can be automated but are similar to what was reported in Section 5.1, thus, the score assigned to this metric.

Services deployment (Score: 1) As explained for the previous criterion, the environment the services will be deployed into is equivalent to the one described in Section 5.1. Therefore, the score assigned is the same.

Services updating (Score: 1) In the context of this architecture, updating services can be seen as a new deployment. Thus, the discussion of Section 5.1 still applies.

Services configuration (Score: 2) The considerations for this metric presented in Section 5.1 apply here because of the nature of LXC containers to behave like "lightweight virtual machines" and QEMU running actual virtual machines. As mentioned in Section 5.2, the modularity offered by an arbitrary external filesystem mounted in a LXC container can be seen as a way to keep configuration files close together on the main host's filesystem.

Resources limitation (Score: 3) LXC containers and QEMU virtual machines allow operators to easily limit CPU (sockets, cores, units, and priority), RAM (exclusively pre-allocated, reserved, and maximum amounts), and network (bandwidth and priority) resources available to a given entity. In

the case of Proxmox, these limits can be quickly specified through its web interface, and advanced scheduling or limiting algorithms can be chosen. For example, RAM memory can be allocated to virtual machines through *ballooning devices*: these virtual memory spaces can expand and contract at runtime, recovering unused memory from idle VMs and allocating the new ones to those VMs needing resources. Storage limits are set during the creation of the environment and can be extended at will as long as the main host is left with enough storage capacity.

Isolation & security (Score: 3) Since each service runs in its dedicated environment and the resources of the main host are paravirtualized, isolation is guaranteed on every level. Performance is on par with typical virtual machine hypervisors. As reported in Section 5.2, malicious exploits can compromise, at most, the environment of a single service. Lateral movement into other environments and services is typically caused by misconfigurations of the main host or failures to keep proper logical bounds between different services.

Changes of state (Score: 1) This architecture typically does not include reactive monitoring solutions: once LXC containers and virtual machines are started, they can only crash due to serious kernel malfunctions - just like a physical machine would. Under very specific circumstances kernel panics or other irrecoverable errors may be triggered and lock up an individual environment instead of crashing it, but nevertheless no native insights are available on the state of the applications running inside those environments. Moreover, in the case of LXC containers, the kernel is shared with the host operating system, and a fault at that level would probably hang or reboot the whole physical machine.

Scaling (Score: 2) Scaling on hyper-converged architectures like Proxmox can be seen as a combination of the observations reported in Section 5.1 and Section 5.2: running multiple copies of a given service is a simple task, but coordinating them to effectively act as a single unit and share the computational load is a complex effort. The considerations mentioned in the paragraphs referenced above are also valid for this architecture.

Introspection and maintenance (Score: 2) Maintenance of hyper-converged architectures can be quite low, especially given the short period of time that CTF competitions run for. Once the system is set up with the preferred kernel version and system tools, there are few reasons to update or reboot the system if its expected lifetime is less than a month. Introspection, on the other hand, suffers from the multiple layers of abstraction that a single service may be hidden behind: to troubleshoot a problem within an application, an operator might have to delve through all abstraction layers in reverse, starting from the service's shared libraries and ending up tracing the host's kernel behavior.

Table 3. Scores assigned to the "Hyper-converged infrastructure" architecture.

Criterion	Score
Initial setup complexity	3
Initial setup duration	3
Services packaging	1
Services deployment	1
Services updating	1
Services configuration	2
Resources limitation	3
Isolation & security	3
Changes of state	1
Scaling	2
Introspection and maintenance	2
Total score	22

5.4 Simple orchestrator or task runner

Below, we report the scores with a brief qualitative judgment. In our discussion, we consider Nomad and Consul as the reference implementations of this architecture. The scores are summarized in Table 4.

Initial setup complexity (Score: 2) At the time of writing, both Nomad and Consul are distributed as self-contained binaries that run on top of a preinstalled operating system. For this reason, all the assumptions regarding the initial setup reported in Section 5.1 are still valid. The setup process of a single-node Nomad cluster is fairly simple and well-documented in the official developer resources. If additional features, e.g., advanced service discovery and health checks, are required, Nomad can be complemented with a basic single-node Consul deployment on the same host. Again, this procedure is fairly well documented and with some experience, it can be deployed inside Nomad itself instead as a separate service for increased coherence and modularity.

Initial setup duration (Score: 2) Due to their binary distributions, both Nomad and Consul are considerably quick to deploy on top of an existing operating system. The initial setup can be completed and tested in a matter of hours, even by those operators who never worked with them. This is because complex features are often optional and can be enabled incrementally when needed.

Services packaging (Score: 3) Typically, in a CTF competition, developers will probably use a container-based packaging system. What was reported about packaging in Section 5.2 therefore also applies to this architecture. However, the execution of packaging software in Nomad can also be done in other ways: the platform officially supports *task drivers* for single binaries, Docker and Podman containers, Java JAR files, QEMU VMs, and optionally LXC containers and *systemd-nspawn* namespaces. These last two options

are currently maintained by the community but endorsed by HashiCorp. All in all, the reference implementations support enough packaging styles to accommodate the needs of most developers.

Services deployment (Score: 3) Similarly to what is described in Section 5.2 about Dockerfiles, Nomad has its own format for a deployment. This file is called *Job Specification* [19] and is written in *HashiCorp configuration language* [18], or HCL for short, which is a purpose-specific JSON [12] notation. Deploying a *Job* is done through the *application* of this file: the main Nomad server parses its contents and proactively creates the environment needed to run the given software package with the specified requirements and limitations. In the case of containers, this specification includes all the options, storage mounts, and other flags that would have been specified manually through the Docker command line interface. The Job Specification format must cover a wide variety of use cases and is thus intrinsically complex, but in the context of CTFs it can have a quite readable and maintainable text structure due to the low amount of functionalities usually needed: once a container has been defined by its basic characteristics, resource limits have been put in place, healthchecks and other minor management policies have been set, it is rare to see other substantial additions to the specification.

Services updating (Score: 3) Updating a deployment can be done by editing its Job Specification and re-applying it. To assist operators in avoiding critical mistakes, the Nomad command line interface supports a special "plan" mode, which does not directly apply the requested changes, but shows a graphical difference between the current and the new specification, highlighting the sections that would change. The plan is assigned a progressive id, which operators can use during re-deployment to ensure that only the previewed changes will be applied. If the context on the server has changed - for example another operator has modified the Job or the server was forced to reschedule it in favor of higher priority tasks - the new deployment will not be accepted, and the operator will be warned of the discrepancies.

Services configuration (Score: 3) As already mentioned in Section 5.2 and Section 5.3, configuration files for a service can always be mounted from external storage into the target container. The Job Specification format introduces another way of further improving the placement of such files: embedding them directly into the deployment specifications. With some experience, operators can statically define the contents of configuration files and other secret values along the main deployment's options. This enables developers and operators to better use versioning tools without manually cross-referencing different software states with their respective settings.

Resources limitation (Score: 2) While the Nomad orchestrator can technically limit the resources available to a given deployment, it cannot do so through its own mechanisms. It depends on the packaging format used by the developer of a service. The Job Specification file gives operators an abstract way of specifying such limits, which then get re-implemented by the internal driver of each packaging format. For example, raw processes will be directly encapsulated in Linux control groups, containers will have the relevant options

passed through to their runtime, and so on. It is not guaranteed that all the task drivers will be able to enforce all the available resource limits.

Isolation & security (Score: 3) Since the underlying technologies typically used in a CTF competition (containers and VMs) are the same as the ones analyzed in Section 5.2 and Section 5.3, the comments made in those sections are valid for this architecture as well.

Changes of state (Score: 2) Simple orchestrators like Nomad often lack advanced state management capabilities. While they usually implement simple techniques for handling application crashes, such as delayed retries and exponential back-offs, they do not guarantee to react to the changes of the application state themselves. For example, Nomad must be complemented with another HashiCorp product, Consul, to provide proper health checking and interactive service discovery mechanisms. This can greatly enhance the state management capabilities of this orchestrator at the expense of additional architectural complexities and increased maintenance costs (both in terms of time and resources).

Scaling (Score: 2) The number of instances of a single service can be easily adjusted by changing the *count* field of its Job Specification, but no additional networking structures are automatically put in place to distribute the users' requests. The effort required to put these extra components in place is comparable to what was described in Section 5.3. A substantial feature of Nomad over other solutions is the embedded Autoscaler [16] that can handle the automated scaling of deployments based on resource usage.

Introspection and maintenance (Score: 2) The level of introspectability of this architecture is comparable to what was analyzed in Section 5.3 and is subject to the same observations. However, an aspect favoring this solution is the centralized logging of the most relevant low-level events generated by Nomad. Maintainability can be considered straightforward: Nomad and Consul versions can be easily upgraded and rolled back by replacing their main binary and restarting the corresponding process manually or through the operating system's init facilities.

5.5 Complex orchestrator

Below, we report the scores with a brief qualitative judgment. In our discussion, we consider Kubernetes as the reference implementation of this architecture. The scores are summarized in Table 5.

Initial setup complexity (Score: 2) Kubernetes heavily relies on cloud-only technology to operate at maximum efficiency. However, two fundamental functionalities, *Load Balancers*, and dynamic *Persistent Volumes*, that can be replicated on on-premises, self-managed hardware with considerable effort and prior knowledge of the inner workings of this solution include. For completeness, we mention two projects that can be used in self-managed environments to regain some of such functionalities: MetalLB [32] and Rancher

Table 4. Scores assigned to the "Simple orchestrator or task runner" architecture.

Criterion	Score
Initial setup complexity	2
Initial setup duration	2
Services packaging	3
Services deployment	3
Services updating	3
Services configuration	3
Resources limitation	2
Isolation & security	3
Changes of state	2
Scaling	2
Introspection and maintenance	2
Total score	27

Longhorn [43]. Deploying Kubernetes on cloud platforms can be done manually, but in the context of CTF competitions, it is more convenient and less expensive to use a minimal provider-managed cluster. Most medium to large-scale providers offer support for Kubernetes clusters at the moment of writing [1,8,15,33,35], and some of them are available to sponsor CTF events. The initial setup complexity of this architecture is very low, although it does require a considerable amount of preparation and documentation by the operators.

Initial setup duration (Score: 3) The initial setup of a provider-managed Kubernetes cluster can be done in a few minutes. Given that a valid account is already registered with the chosen cloud provider, in some cases, it is even possible to complete the setup process through a web browser.

Services packaging (Score: 3) Kubernetes is an orchestrator explicitly designed to manage application containers, hence, the considerations on packaging done in Section 5.2 still apply. This orchestrator supports different container runtimes [24] - newer versions comply with the *Container Runtime Interface* (CRI) standard - and consequently, it does not enforce a specific packaging process: as long as the final result adheres to the Docker image specifications, the processes or tools employed by developers are immaterial.

Services deployment (Score: 3) In addition to what was reported in Section 5.2 and Section 5.4, Kubernetes relies on a custom set of YAML [48] documents, called *manifests*, to define and organize its resources. Writing and understanding manifests can be quite complex: their syntax is not always obvious, many nested fields have repeated or similar names, and the resources' specifications are subject to frequent changes. However, the advantage of using these documents is that all the specifications for the software components required by a particular service can be contained in a single text file. This satisfies the requirement of keeping logically separate services in different images, files, or assets found in all the previously analyzed architectures.

Services updating (Score: 2) Updating a service managed by Kubernetes works similarly as described in Section 5.4, except for the absence of the "plan" mode: when a manifest file is modified, the new state is applied with no checks. This approach has the disadvantage of not automatically deleting old resources: if a specific resource is removed from the manifest and the new version is applied, it is up to the operator to manually delete the unreferenced elements from the cluster.

Services configuration (Score: 3) Manifests support the definition of textual key-value pairs to be used as environment variables or mounted as raw files inside the application containers via through the usage of entities called *ConfigMaps* and *Secrets*. This feature allows keeping all the resources to run a service close to its source code and packaging assets. Developers do not need to worry about where to store configuration files; operators are only concerned with properly applying a single specification.

Resources limitation (Score: 3) The approach that Kubernetes takes to resource limitation is similar to the one that was described in Section 5.4: CPU, RAM, and storage space limits are defined in the manifest, and the underlying container runtime is in charge of applying the proper *cgroups* configurations to enforce them. Special types of limitations, such as the ones on network bandwidth, can be enforced through purpose-specific overlays and external plugins.

Isolation & security (Score: 2) The perceived security of a container orchestrator is often inversely related to its complexity: due to the large number of separate components and specifications that make up Kubernetes, operators need to pay special attention to each of them to ensure that no misconfigurations occur in the services publicly exposed and their environments. Experienced operators can reach a satisfying level of isolation between services and overall security - both from the outside world and from the inside of the cluster - in a matter of hours, but for CTF events a more thorough, multi-day analysis is suggested. Specialized literature [21,29,34] provides many insights into the key points to be covered during analyses of this kind.

Changes of state (Score: 3) Complex orchestrators, and Kubernetes in particular, handle state changes very well: thanks to their modular nature, it is usually possible to develop custom plugins to react to deployments' state changes in arbitrary ways. Kubernetes itself handles restarts and exponential back-offs natively, as well as proper *healthchecks* for containerized applications: it can interact with them in various ways to determine their status, preventing subtle issues and application lock-ups that simpler orchestrators usually cannot detect. This feature is the cornerstone of the scaling and introspection features analyzed in the following paragraphs.

Scaling (Score: 3) The complexity of scaling workloads is comparable to what was described in Section 5.4, with the addition of more advanced techniques revolving around the management of stateful and stateless applications. Once the state issue is solved, most applications can be scaled automatically through Kubernetes' facilities (*load balancers*, *ingresses*, etc.). These components provide a set of strategies for distributing network traffic to the various

Table 5. Summary of the scores assigned to the "Complex orchestrator" architecture.

Criterion	Score
Initial setup complexity	2
Initial setup duration	3
Services packaging	3
Services deployment	3
Services updating	2
Services configuration	3
Resources limitation	3
Isolation & security	2
Changes of state	3
Scaling	3
Introspection and maintenance	1
Total score	28

instances of an application container, relieving operators from the burden of manually setting up such mechanisms and reconfiguring them every time the scaling needs to be adjusted.

Introspection and maintenance (Score: 1) The complexity of these kinds of orchestrators is both their strength and their weakness: since they consist of many components, operators need a great deal of experience to recognize problematic situations and to determine where to intervene, and which tools fit the situation at hand. For the same reason, it is common for an initially simple problem to spread across multiple subsystems and become more complex, creating deadlock situations in which an operator must identify the first component to troubleshoot and follow the chain of events.

6 Discussion of the results

Figure 1 shows the cumulative scores of the considered architectures in a generic use case when CTF organisers have no particular prior experience or infrastructural preferences. By the discussion of Section 5 and the chart in the figure, we conclude that the architectures providing a certain level of automation in deploying and managing services obtain higher scores. Moreover, we can observe that the complexity of the chosen architecture plays a limited role. This is clear when we compare "Dedicated server (legacy deployments)" and "Dedicated server (containerized deployments)" with "Complex orchestrator". In the first case, there is a 10-point difference between the architectures, which highlights a large margin of operational improvement at the cost of the greater complexity of the chosen solution. In the second case, there is only a 3-point difference between the two architectures, which remarks a less relevant difference between environments based on application containers. Similarly, the single-point difference between "Simple orchestrator or task runner" and "Complex orchestrator" shows that

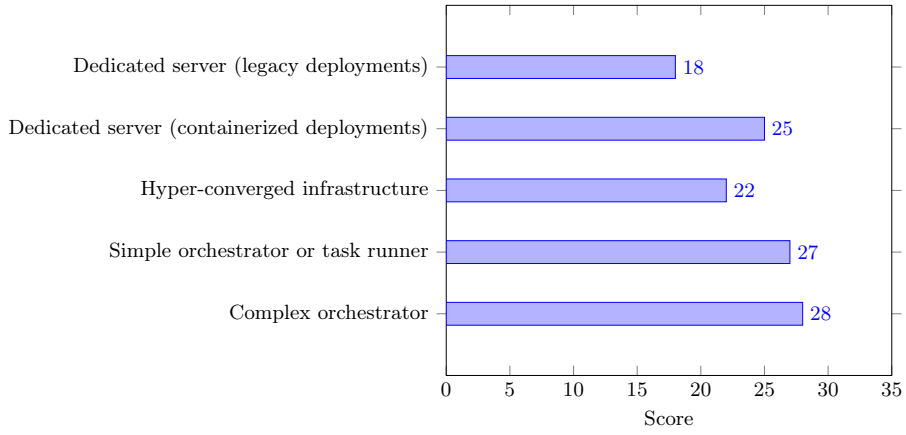

Fig. 1. The cumulative scores of each analyzed architecture.

once a certain level of abstraction over the physical hardware and automation is reached, the exact technology used has a minor impact in organizing CTF events.

The above observations are strengthened by Figure 2 showing the scores of the various criteria of Section 4 grouped by architectural solution. The chart supports the hypothesis that the most complex solutions have similar characteristics because their scores differ little. For example, the score difference between the criteria of the "Simple orchestrator or task runner" and the "Complex orchestrator" architectures is never greater than 1. This difference grows to 2 when considering the "Dedicated server (containerized deployments)" architecture. Again, this small difference can be explained by the fact that it is sufficient to reach a certain level of abstraction and to ensure a certain amount of fundamental functionalities to reduce the operational burden of the platform.

Finally, it is interesting to consider the case of the "Hyper-converged infrastructure". This solution comes out as a mid-range choice that would present no real advantages over the alternatives, encouraging organizers to either lower their operational complexity and go for one of the simpler solutions or raise it by choosing a more complex but more performant architecture.

7 Recommendations for specific use cases

The previous sections analyzed the various architectures in a generic context where CTF organisers have no prior experience or infrastructural preferences. However, these elements are key factors when the organizers are already experienced with automated, large-scale cloud deployments or have essential technical competencies to carry out such tasks. Below, we provide some recommendations on the architectures to use based on the scale of the event and the technical skills of the organisers.

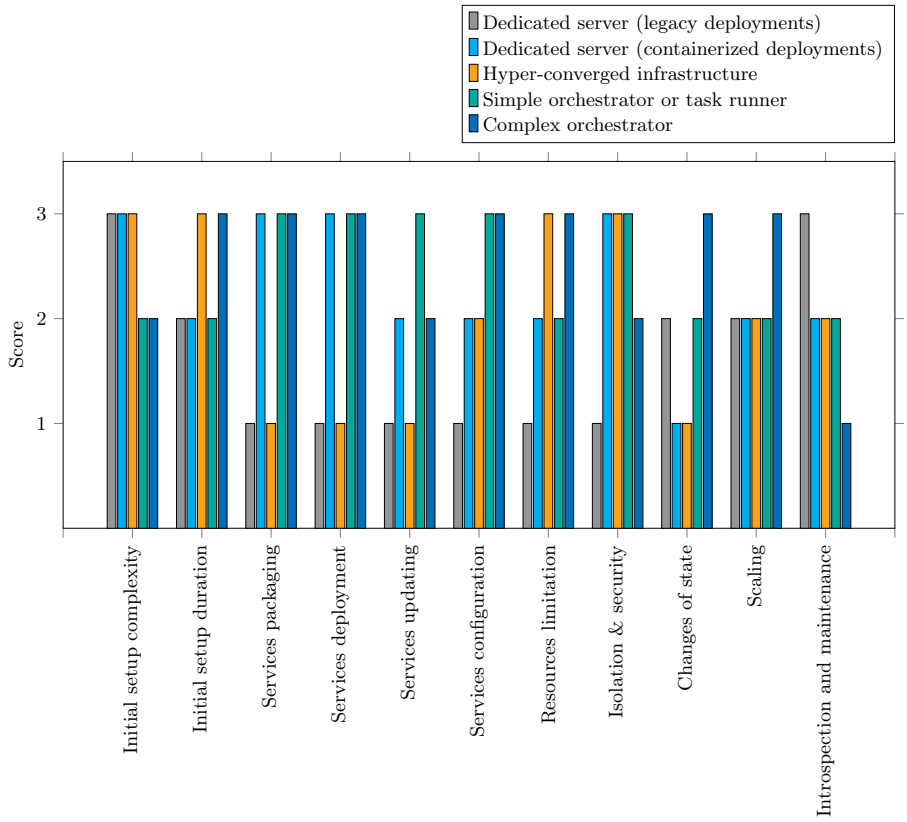

Fig. 2. The cumulative scores of the analyzed architectures, grouped by criterion.

Small scale If an event is restricted to a small (<100), selected group of participants and is intended to run for a very limited time (<24 hours), it may not be worth it to deploy complex technologies if the organizers have no prior experience with them. Such an event would probably run with acceptable results even with the "Dedicated server (legacy deployments)" architecture because the manual burden of setting up and managing the event is tolerable. However, application containers are strongly suggested to ensure a certain isolation level among the challenges and limit the scope of eventual dangerous operations initiated by participants.

Mid-size CTFs When the number of participants (>100, <2000) grows, organizers may face challenges related to, e.g., heavier concurrency and resource allocation. Complex architectures may be justified if the organizers have at least some prior experience with such technologies. Still, the downside of this choice is the organizational burden of correctly setting up a complex environment and the number of people who can work on such a process. Application

containers are usually a hard requirement at this size since the number of developers involved in the event organization needs a robust development and deployment pipeline with as many automated steps and validations as possible. A simple orchestrator or task runner could be useful in managing the deployment process, but it is still possible for competent operators to carefully manage services manually if desired.

Mid-to-Large size CTFs An event with >2000 registered participants is usually well-known in the field and probably not run for the first time. At this size, organizers are expected to be already competent with the process of managing and, more importantly, troubleshooting CTF events: it can be supposed that their past experiences have given them enough confidence in their architectural choice that only minimal improvements would need each time a new event is hosted. When it is not ensured that the team who manages the event is the same each time, a leaner, simpler, well-documented approach could be preferred over a more optimized, complex, and less documented one. Operators should prioritize stability and developer experience when choosing an architectural solution, possibly avoiding approaches that rely on a single team member being able to complete certain tasks at any given time.

Large-size CTFs or teams with multiple experienced operators Advising on how to run large-scale CTFs is beyond the scope of this paper since the organizers of such events already possess enough domain-specific knowledge and have gained enough experience with the past editions of their events to have a clear picture of how the whole infrastructure should look and work. This is especially true for teams that include multiple operators skilled in the same technologies, as this enables them to split their work and avoid cognitive stress in the most intense phases of the event. Such teams would probably favor the progressive introduction of new and potentially unstable technologies not yet widely used in the CTF field to foster innovation and push the limits of the typical challenges offered in smaller competitions.

Finally, note that unforeseen complications are common during CTF events: participants forge more powerful exploits than expected, there is too much computational load - or even not enough, misuse of internal APIs, intentional Denial of Service attacks, etc. Due to this fact, the most important recommendation to CTF organizers can be summarized as *"use a technology that you are familiar with."* Time is essential when players flood services with requests and tentative exploits. When something goes wrong, the time it takes to fix the problem and get the experience back to normal is directly proportional to the participants' opinion of the event.

8 Related work

Most of the literature focuses on using CTF competitions for educational purposes, and only a limited number of papers delve into their organizational aspects and offer guidance for potential organizers. Herein, we survey select papers regarding these two investigation lines.

As regards the organizational aspects, Kucek et al. [25] compare several game portals, such as CTFd, following a methodology similar to ours. Their criteria comprehend several factors that must be considered when organizing a competition regarding customization, installation, setup, and reliability, and assess open-source solutions available at the time of writing according to them. In contrast, our work does not consider game portals but focuses on the hosting architectures that can be used in CTFs. Karagiannis et al. [22] compare the usage of virtual machines and container technologies (both Linux and application types) in the context of CTF competitions. Differently, our work does not only analyse how virtual machines and containers can be used to run vulnerable services in the context of CTFs, but also considers further aspects such as service configuration, deployment, and update. Following a similar line of investigation, Raj at al. [41] explored the usage of application containers for attack-defence CTFs. Although they consider a different kind of competition, some of the hosting techniques we considered could also be used for hosting attack-defence CTFs. Raman at al. [42] proposed a framework to assess the quality of CTF competitions through subjective and objective metrics. Although the world of these competitions has changed since the publication of their work, their framework could prove useful for organisers in evaluating their course of action.

The effectiveness of CTF competitions as introductions to cybersecurity and ethical hacking for high school and university students has been studied in several papers in the literature. Lagorio et al. [26] outlines the advantages and disadvantages of integrating CTFs into higher education courses and the impact on students' performances before and after participating in such events. They point out that this integration requires careful planning for combining theory and practice. Similarly, Cole [4] studied the impact of this kind of competition on students' motivation and exam scores, specifically what happens when classical exam questions are reformulated into more practical and goal-oriented exercises akin to the challenges found in CTFs. The author verified the positive impact of the new learning method on students without prior or advanced technical skills. Vykopal et al. [47] further corroborated these findings of introducing CTF completion inside standard lectures but outlined potential pitfalls that could negatively impact students' learning and evaluation processes.

9 Conclusions and future work

This paper has surveyed the prominent architectures used to host CTF competitions and provided a qualitative assessment according to a set of criteria ranging from the complexity of setting up the infrastructures to the security mechanisms for isolating vulnerable services. Our analysis reveals that the architectures providing a certain level of automation in deploying and managing services are the most suitable to be adopted in a generic context where CTF organisers have no prior experience or infrastructural preferences. Finally, we have provided some recommendations for adopting architectures based on the event's size and the organizers' prior experience. In any case, the organizers' judgment of their skills

and desire to experiment with new architectures always win on the proposed recommendations, even if these must always be evaluated in the ever-changing and surprising context of CTF competitions.

In future work, we plan to implement an automated tool to calculate the best fit between various architectures, depending on the requirements, the size of an event, and the competencies of the organizing team. The tool will supply a questionnaire asking the key requirements and will provide a generic recommendation as output. The questionnaire could also involve other aspects, such as the size of the organizing team, whether or not the event is new or has had past editions, the historical or expected number of participants, the number of operators with respect to challenge developers, the categories of challenges to be offered, and the technologies most of the team is already acquainted with. Moreover, this approach could be further extended to combine the questionnaire results with a static analysis of the challenges' source code to automatically determine the packaging format used for developing and testing the challenges and to guess the standard category they belong to. Combining these two factors could pre-determine some of the questionnaire's answers and guide the users toward solutions that support the detected technologies with as little manual intervention as possible.

References

1. Amazon Web Services: Managed Kubernetes Service – Amazon EKS – Amazon Web Services — aws.amazon.com. https://aws.amazon.com/eks/, [Accessed 20-08-2023]
2. Ansible, R.H.: Ansible is Simple IT Automation — ansible.com. https://www.ansible.com/, [Accessed 15-08-2023]
3. born2scan Team: born2scan – How we hosted DanteCTF 2021: A brief tour of the infrastructure that supported the first edition of DanteCTF — born2scan.run. https://born2scan.run/articles/2021/06/29/How-we-hosted-DanteCTF-2021.html, [Accessed 12-08-2023]
4. Cole, S.V.: Impact of capture the flag (ctf)-style vs. traditional exercises in an introductory computer security class. In: Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1. p. 470–476. Association for Computing Machinery (2022). https://doi.org/10.1145/3502718.3524806
5. CTFd LLC: CTFd : The Easiest Capture The Flag Framework — ctfd.io. https://ctfd.io, [Accessed 08-08-2023]
6. CTFtime team: CTFtime.org / All about CTF (Capture The Flag) — ctftime.org. https://ctftime.org/event/list/past, [Accessed 08-08-2023]
7. Debian: Debian – The Universal Operating System — debian.org. https://www.debian.org/, [Accessed 15-08-2023]
8. Digitalocean: DigitalOcean Managed Kubernetes — digitalocean.com. https://www.digitalocean.com/products/kubernetes, [Accessed 20-08-2023]
9. Docker Inc.: Dockerfile reference — docs.docker.com. https://docs.docker.com/engine/reference/builder/, [Accessed 16-08-2023]
10. Docker Inc.: Home — docker.com. https://www.docker.com/, [Accessed 10-08-2023]
11. Docker Inc.: Overview of Docker Compose — docs.docker.com. https://docs.docker.com/compose/, [Accessed 16-08-2023]

12. ECMA: ECMA-404: The JSON data interchange syntax, 2nd edition. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland (12 2017). https://doi.org/10.13140/RG.2.2.28181.14560
13. Extra Good Labs, Inc: How to Run a CTF — jumpwire.io. https://jumpwire.io/blog/how-to-run-a-ctf, [Accessed 12-08-2023]
14. Goedegebure, C.: Hosting a CTF made easy using Docker and DigitalOcean — coengoedegebure.com. https://www.coengoedegebure.com/hosting-a-ctf-made-easy/, [Accessed 12-08-2023]
15. Google: Google Kubernetes Engine (GKE) | Google Cloud — cloud.google.com. https://cloud.google.com/kubernetes-engine, [Accessed 20-08-2023]
16. HashiCorp: Autoscaling | Nomad | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/nomad/tools/autoscaling, [Accessed 18-08-2023]
17. HashiCorp: Consul by HashiCorp — consul.io. https://www.consul.io/, [Accessed 10-08-2023]
18. HashiCorp: GitHub - hashicorp/hcl: HCL is the HashiCorp configuration language. — github.com. https://github.com/hashicorp/hcl, [Accessed 18-08-2023]
19. HashiCorp: Job Specification | Nomad | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/nomad/docs/job-specification, [Accessed 18-08-2023]
20. HashiCorp: Nomad by HashiCorp — nomadproject.io. https://www.nomadproject.io/, [Accessed 10-08-2023]
21. Hauber, C.: Taking Over A Kubernetes Cluster: Automating An Attack Chain. Bachelor's thesis, Johannes Kepler University Linz (2022), https://www.ssw.uni-linz.ac.at/Teaching/BachelorTheses/2022/Hauber_Carina.pdf
22. Karagiannis, S., Ntantogian, C., Magkos, E., Ribeiro, L.L., Campos, L.: Pocketctf: A fully featured approach for hosting portable attack and defense cybersecurity exercises. Information **12**(8), 318 (Aug 2021). https://doi.org/10.3390/info12080318
23. Kimminich, B.: Hosting a CTF event – Pwning OWASP Juice Shop. https://pwning.owasp-juice.shop/part1/ctf.html, [Accessed 12-08-2023]
24. Kubernetes Documentation: Container Runtimes — kubernetes.io. https://kubernetes.io/docs/setup/production-environment/container-runtimes/, [Accessed 20-08-2023]
25. Kucek, S., Leitner, M.: An empirical survey of functions and configurations of open-source capture the flag (ctf) environments. Journal of Network and Computer Applications **151**, 102470 (2020). https://doi.org/https://doi.org/10.1016/j.jnca.2019.102470
26. Lagorio, G., Ribaudo, M., Armando, A.: Capture the flag competitions for higher education. In: ITASEC. CEUR Workshop Proceedings, vol. 2940, pp. 447–460. CEUR-WS.org (2021), https://ceur-ws.org/Vol-2940/paper38.pdf
27. LinuxContainers: Linux Containers — linuxcontainers.org. https://linuxcontainers.org/, [Accessed 10-08-2023]
28. Maiya, M., Dasari, S., Yadav, R., Shivaprasad, S., Milojicic, D.: Quantifying manageability of cloud platforms. In: 2012 IEEE Fifth International Conference on Cloud Computing. pp. 993–995 (2012). https://doi.org/10.1109/CLOUD.2012.111
29. Martin, A., Hausenblas, M.: Hacking Kubernetes. O'Reilly Media, Incorporated (2021)
30. Menage, P.: Control Groups — The Linux Kernel documentation — docs.kernel.org. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html, [Accessed 16-08-2023]
31. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux Journal **2014** (03 2014)

32. MetalLB: MetalLB, bare metal load-balancer for Kubernetes — metallb.universe.tf. https://metallb.universe.tf/, [Accessed 20-08-2023]
33. Microsoft: Managed Kubernetes Service (AKS) | Microsoft Azure — azure.microsoft.com. https://azure.microsoft.com/en-us/products/kubernetes-service, [Accessed 20-08-2023]
34. Minna, F., Blaise, A., Rebecchi, F., Chandrasekaran, B., Massacci, F.: Understanding the security implications of kubernetes networking. IEEE Security & Privacy **19**(5), 46–56 (2021). https://doi.org/10.1109/MSEC.2021.3094726
35. OVH: Managed Kubernetes Service — ovhcloud.com. https://www.ovhcloud.com/en/public-cloud/kubernetes/, [Accessed 20-08-2023]
36. Podman: Podman — podman.io. https://podman.io/, [Accessed 10-08-2023]
37. Project, T.F.: Chapter 17. Jails — docs.freebsd.org. https://docs.freebsd.org/en/books/handbook/jails/, [Accessed 16-08-2023]
38. Proxmox Server Solutions GmbH: Proxmox - Powerful open-source server solutions — proxmox.com. https://www.proxmox.com/, [Accessed 10-08-2023]
39. QEMU: QEMU — qemu.org. https://www.qemu.org/, [Accessed 10-08-2023]
40. Raj, A.S., Alangot, B., Prabhu, S., Achuthan, K.: Scalable and lightweight CTF infrastructures using application containers (pre-recorded presentation). In: 2016 USENIX Workshop on Advances in Security Education (ASE 16). USENIX Association, Austin, TX (Aug 2016), https://www.usenix.org/conference/ase16/workshop-program/presentation/raj
41. Raj, A.S., Alangot, B., Prabhu, S., Achuthan, K.: Scalable and lightweight CTF infrastructures using application containers (pre-recorded presentation). In: 2016 USENIX Workshop on Advances in Security Education (ASE 16). USENIX Association, Austin, TX (Aug 2016), https://www.usenix.org/conference/ase16/workshop-program/presentation/raj
42. Raman, R., Sunny, S., Pavithran, V., Achuthan, K.: Framework for evaluating capture the flag (ctf) security competitions. In: International Conference for Convergence for Technology-2014. pp. 1–5 (2014). https://doi.org/10.1109/I2CT.2014.7092098
43. Rancher: Longhorn — longhorn.io. https://longhorn.io/, [Accessed 20-08-2023]
44. redpwn: rCTF — rctf.redpwn.net. https://rctf.redpwn.net/, [Accessed 12-08-2023]
45. Rørvik, M.: How to host a CTF? — bekk.christmas. https://www.bekk.christmas/post/2020/18/how-to-host-a-ctf, [Accessed 12-08-2023]
46. The Linux Foundation: Production-Grade Container Orchestration — kubernetes.io. https://kubernetes.io/, [Accessed 10-08-2023]
47. Vykopal, J., Švábenský, V., Chang, E.C.: Benefits and pitfalls of using capture the flag games in university courses. In: Proceedings of the 51st ACM Technical Symposium on Computer Science Education. p. 752–758. Association for Computing Machinery (2020). https://doi.org/10.1145/3328778.3366893
48. YAML: The Official YAML Web Site — yaml.org. https://yaml.org/, [Accessed 20-08-2023]
49. Zeyu's Infosec Blog: Hosting a CTF — SEETF 2022 Organizational and Infrastructure Review — infosec.zeyu2001.com. https://infosec.zeyu2001.com/2022/hosting-a-ctf-seetf-2022-organizational-and-infrastructure-review, [Accessed 12-08-2023]