Environment Code-First Framework: Provisioning Scientific Computational Environments Using the Infrastructure-as-Code Approach

Daniel Adorno Gomes
University of Trás-os-Montes and Alto Douro
Vila Real, Portugal
www.utad.pt
e-mail: adornogomes@gmail.com

Pedro Mestre and Carlos Serôdio
Centro Algoritmi
University of Minho, Guimarães, Portugal and
CITAB - Centre for the Research and Technology of Agro-Environmental and Biological Sciences
University of Trás-os-Montes e Alto Douro
www.utad.pt
e-mail: pmestre@utad.pt, cserodio@utad.pt

Abstract— **Nowadays, computational resources are vital practically in all areas of science. At the same time, science's dependence on computing is pointed to by experts as one of the main causes of the reproducibility crisis. Many factors have contributed to the low reproducibility of scientific research. They are related to the cultural aspects of the scientific software's development, the behavior of the scientist-developer, and technical issues. Based on these factors, the authors presented the Environment Code-First (ECF) framework to guide researchers on increasing the reproducibility of their works by developing computational environments that can be easily recreated without manual intervention. The framework's foundation is the Infrastructure-as-Code approach, and it intends to permit other researchers to recreate an environment only by executing a script. A real case is presented, demonstrating the provision of a bioinformatics environment by using the Prokaryotic Genomics and Comparative Genomics Analysis Pipeline (PGCGAP) protocol, and the ECF framework. The paper shows a comparison between these two methods in terms of time-consumption, manual intervention, platform-agnosticism, and portability. The tests perfomed on three different machines demonstrated that there are many benefits on using the ECF framework such as independency of platform, total portability, and practically any manual intervention. Of course, there is a cost, and it is related to the hard work on developing the code that generates the environment. Another point that needs to be highlighted is the time spent and efforts on achieving the necessary knowledge to create those programs.**

Keywords-computational environment; infrastructure-as-code; open science; computer programming; containerization; virtualization; reproducible research.

I. INTRODUCTION

The use of computing is essential in all sciences. Many areas such as biology, physics, and chemistry are dependent on simulations to perform experiments in silico. It is faster and cheaper to execute simulations than conduct actual experiments. In other situations, it is impossible to conduct an experiment without computational resources, for example, when it is necessary to process and analyze a large amount of data in a short space of time. Over the past 70 years, research methods have become more and more sustained by computational means. Nowadays, this dependency is pointed by specialists as one of the main factors responsible for the crisis of scientific reproducibility [1][2].

Reproducibility is one of the most important pillars of science, along with transparency and openness [3]. It is a crucial element to recreate and extend the research work developed by others. And this practice is critical to keep science evolving.

From an economic point of view, many losses on investment have been reported in the last few years related to the reproducibility of scientific research. In [4], authors report that, in the United States, around 50% of the total amount invested every year in biomedical research, is supporting scientific studies that other researchers cannot reproduce. Recently the European Commission reported that the losses related to low reproducibility on clinical trials are estimated at 28 billion USD per year [5].

Besides the crisis in science related to reproducibility [6], there is pressure from funding institutions and publishers on authors to adopt open science principles like Open Reproducible Research (ORR) [7]. This means to provide access to the resources (e.g., data, source code, documentation) used to generate the results reported in their publications [8].

These factors increase the need to improve reproducibility and transparency in science, especially in research where computation has an important role [9]. In a survey published by Nature with 1576 researchers, Baker asked them which kind of factors contribute to the irreproducibility. More than 40% of respondents reported computational issues, such as unavailability of computational methods, code, and data [10].

Despite the absence of code and data being the most common issues on reproducibility, it is essential to highlight the problems related to the computational environments that support the research. Incompatibility of operating systems, different versions of the same compiler or interpreter, lack of software packages and libraries, the dependency of libraries of a specific software platform, are all issues related to the environment used to develop scientific applications. Accordingly with Boettiger in [11], these kind of computational issues related to the environment can be classified as dependency hell or code rot.

It is critical for the reproducibility of any research work that made use of computational resources to provide the

environment besides the code, the data, and the documentation [12]. As Donoho wrote in [13]: “An article about computational results is advertising, not scholarship. The actual scholarship is the full software environment, code and data, that produced the result.”

As the way to share scientific knowledge is changing from traditional articles to this new concept based on Open Reproducible Research, it is necessary to change the way scientific applications are developed, focusing not only on generating results but on being reproducible and useful to the wider scientific community, as well [14].

Today, we can count on technologies (e.g., virtual machines, containers and cloud computing), and new technical approaches that permit the treatment of the computational infrastructure that supports scientific research as a software system, a method also known as Infrastructure-as-Code (IaC). Using Infrastructure-as-Code, we can provide the infrastructure programmatically, having many benefits such as treating the infrastructure like a computer system, versioning it, and avoiding typical issues like dependency hell [15]. However, there are many cultural barriers related to scientific software development that contributes to irreproducibility, despite the technical and technological resources available. They are related to the purpose of the scientific applications, the behavior of scientists when developing software, the lack of use appropriate software engineering practices, among other factors.

Considering the issues, particularities and characteristics related to the development of scientific applications, and having the IaC approach as a technical foundation, the authors present in this paper the Environment Code-First Framework. The goal of the framework is to help to increase reproducibility by reducing the time and the efforts when reproducing specific research, mitigating issues related to the availability of the computational environment created and used by the researchers. It guides the scientists on developing the infrastructure's code to make the computational environment available before they start to develop the scientific application in itself. In the end, the environment will be a deliverable as the others objects used and produced by the research, being stored and accessible in the same repository (e.g., Git or Github) with them. That means other researchers will have access to environment code, application code, data, and documentation, all together, avoiding reproducible issues.

The framework defines an architecture based on virtual machines and containers, the two technologies most used by the scientific community in the last fifteen years to create self-contained computational environments. The innovation, in this case, is in the fact that the framework combines the two technologies instead of using them in an isolated way. This approach permits developing more homogeneous environments independent of hardware and software platforms.

Besides increasing reproducibility and transparency of new scientific research works, this proposal can be helpful in other aspects like education and dissemination of the open science principles. It can help experient and future researchers understand and prepare to produce executable papers and migrate old research to this new approach. Also, it can help bring down cultural barriers that impede the advance of the open science philosophy, such as authors' hesitation to share their work to avoid publishing erroneous papers.

The rest of the paper is organized as follows: Section 2 presents some typical technical issues on reproducible research, related to scientific computational environments. Section 3 presents some cultural aspects of the scientific community that difficults the increasement of reproducibility. Section 4 presents some related work on IaC. Section 5 makes an explanation about what is this new approach called Infrastructure-as-Code. Section 6 presents the Environment Code-First framework. Section 7 describes a case study of a recent bioinformatics pipeline implementation, and compares it to the ECF framework implementation. Section 8 presents the discussion. Finally, in Section 9 it is presented the conclusion.

II. TECHNICAL ISSUES

In this section, the authors present some typical issues on reproducible research, related to computational resources, which have been faced by researchers.

Collberg et al. show in [16] the technical issues that researchers have to deal when trying to reproduce results published by others. In general, considerable effort is needed to recreate the original computational environment and achieve similar results. The authors analyzed 613 executable papers. Only 123 applications were correctly compiled. Of them, 102 ran with success. Unavailability of a specific version of a software component and missing third-party packages or libraries are the reasons that caused 36% of the failed builds.

In [17], Glatard et al. expose how complex pipelines that are composed by several parts of software from different sources, can have unexpected outputs or a failed execution of the entire workflow, due minor changes introduced in the computational environments, for example, when a updated is applied in the operating system.

In [11] the author describes a typical issue called "code rot". It is a kind of issue that affects the results of the original code due to updates applied in the software environment to fix bugs, add new features, or deprecate old ones. All the software that composes the environment like the operating system, the development language, and the libraries, can be affected by an update generating different results from the original. Also, the author describes another problem known as dependency hell. It occurs when installed software packages have dependencies on specific versions of other software packages. It also includes platform-specific dependencies that are related to a software development platform. The most common types of dependency hell are DLL hell, JAR hell [18].

Ince et al. reported in [19] problems related to differences between the published and the reproduced results using the same source code of computer programs that were implemented and executed with success by non-original researchers. The issue, in this case, occurred when the programs were deployed using hardware and software

configurations that diverged from the original. As a solution, the authors suggested that the source code of the programs should be made available along with documentation describing the hardware and software environment on which the program was developed and should be executed.

In [20] Ben Marwick highliths how important is, in archaeology, the simulation studies executed by computationally-intensive analysis that use mathematical functions based on single-precision floating-point arithmetic. In this case, the issues are related to the variation of the results when executed accross different opearting systems. Also, the author describes the high difficulty in maintaining an environment reproducibly, even when using only one machine, due to automatic updates that software components sofer considering an extended period.

III. Scientific Applications and Cultural Barriers

Besides the technical issues reported in the last topic, there are other factors related to scientific applications' purposes and the behaviors of the scientists that have impacted the reproducibility of scientific works.

Scientific applications are a special category of software generally developed to support comprehending a particular scientific domain that would be impossible to perform without computational resources. Also, the scientific application development process differs significantly from the development of traditional information systems.

As shown in [21], most of the habits and behaviors of the scientists, when the subject is software development, had been adopted during more than last 60 years. Over this period, a culture had emerged and disseminated, creating the role of the scientist end-user developer, as defined by the authors. In this role, the scientist is responsible for planning, designing, developing, testing, and using the results generated by the application. Most of the time, the scientist works alone, using a PC, focusing entirely on the scientific problem.

In [20], Ben Marwick highlights most scientists' primary computational environment is a PC (i.e., desktop or laptop) using one of these three operating systems, Microsoft Windows, MacOS, or Linux.

Hannay et al. performed a survey with almost 2000 respondents to investigate how scientists develop and use scientific software. On the computational environment, 48.5% of the scientists reported that they use exclusively a desktop or a laptop when working on their scientific applications [22].

As reported in [23], a survey performed with 60 scientific software developers, around 80% of the respondents develop their applications alone.

Related to the software engineering principles and practices, generally, most researchers do not test, document, or release their applications [24]. In [25], a systematic mapping study on using software engineering practices for scientific application development, those facts are reinforced. The study points to that self-education is the most common way adopted by researchers when learning about software development. Also, it highlights that the absence of training is one reason for the low level of knowledge of the researchers on software engineering practices and their benefits.

IV. What Kind of Solutions Has Been Reported?

In this section, the authors present a set of works related to IaC that had applied this practice in different areas of the software industry and scientific applications.

Boettiger in [11] discusses the issues of reproducible research with a focus on the computational environment that supported the research. He describes the main issues that impede the successful execution and extension of the code by other researchers. Besides, he makes a review of some approaches used in IaC such as containerization and virtualization. The author analyzes in-depth containerization based on Docker technology, showing the advantages of this approach, such as portability, reusability, versioning, and cross-platform, and how it can help provision computational environments for scientific research.

In [26] the authors present a study on the benefits that could be brought by the adoption of cloud computing and virtualization techniques in scientific applications. They discuss a cultural problem that avoids using virtualization and cloud resources due to the idea that virtualization techniques hurt the results of the scientific applications in comparison with the execution of physical machines. Also, they explore the feasibility of the IaC approach to meet the requirements of computer science and the main issues that need to be addressed by cloud actors to provide the conditions necessary to obtain the maximum benefit from this type of infrastructure.

In [27], the authors present the main characteristics of cloud computing technology, highlighting those that can help in the development of more robust applications based on aspects such as scalability, resilience, fault tolerance, and security. Besides, they discuss the low cost of adopting cloud computing and show how to transition from traditional biomedical computing workflows to cloud computing environments.

In [28], the authors discuss the complexity on creating scientific computing environments. They discuss common issues in the scientific community like the inability of the scientists on setting up isolated and uniform computational environments. Abscense of best practices, software redundance problems, platform dependency are some of the situations described by the authors. To address these kind of issues, they present means to use DevOps concepts, practices and tools to improve the provision of computational environments and reduce the complexity. The use of virtualization, containerization and configuration management are some of the resources used by DevOps engineers suggested in this paper.

Howe discusses in [29] the challenges of provisioning computational environments for scientific research projects through virtualization on cloud computing platforms. The author presents how a complete working environment of specific research containing dataset, software, notes, logs, and scripts, can be included in a virtual machine. Also, he presents a discussion on the advantages and disadvantages of

the adoption of this approach, and how it can help to increase reproducibility.

Cacho and Taghva [30] present the main difficulties researchers face in reproducing research in the Computer Science field. Some of the problems they highlighted include missing original data, issues with the version of the data, deprecated dependencies, unavailable source code, and missing documentation. They provide a solution for reproducible research based on containerization. A real experiment is used to demonstrate the solution using the application OCRSpell. The authors make the provision of a Docker container that embeds the application by creating an image, uploading it, and making it available to other researchers that want to run the OCRSpell. From this image a researcher just needs to download the image and run the container.

Ben Marwick in [20] demonstrates in a practical way how to produce an executable paper relating principles of reproducible research to DevOps practices and tools. The author describes the efforts to create a publication of archaeological research using resources like Docker, R programming language, Git, and Linux operating system. Also, the author explains each tool used to develop the paper and exposes the reasons that motivate the use of each resource.

V. What Is Infrastructure-as-Code?

Infrastructure-as-Code (IaC) is the management and provisioning of IT infrastructure using source code rather than manual processes. It automates the provisioning of infrastructure and eliminates the need to provision and manage servers, operating systems, database connections, storage, and other infrastructure elements manually, avoiding mistakes.

The infrastructure is treated like a software system, which means development tools and agile practices such as Test-Driven Development (TDD), Continuous Integration (CI), and Continuous Delivery (CD) can be used to improve the quality [31]. Programmatically defining our infrastructure means that our environments will be more consistent and reliable, and identical every time. Manual provisioning generally has diverse interpretations of the same instructions, resulting in different configurations [32].

IaC is based on a few practices as follows [15]:

- All the provisions and configurations related to the infrastructure are defined in executable files, such as shell scripts. The actions that need to be applied in the infrastructure like installing a database, increasing the memory of a server, and even creating a new server, are executed from these files.
- The scripts contain the commands that make the maintenance of the infrastructure, and the documentation of the systems and processes.
- The scripts are the source-code that represents the infrastructure. They need to be kept in a version control system like Git or Github.
- Even in infrastructure source-code, tests are critical to finding errors. Continuous integration pipelines can be set up to test and guarantee the quality of the code, supporting practices like continuous delivery and deployment, which can help decrease the downtime of the systems on upgrades or fixes.

There are many benefits to adopting IaC due to the following characteristics of this approach [33][34]:

- Repeatability: having the infrastructure defined as code ensures that we can recreate it as many times as needed, getting the same result.
- Automation: creating and configuring the infrastructure from executable scripts are tasks that can be automated in addition to mitigating manual intervention and avoiding human mistakes.
- Agility: using resources such as source code management systems and version control systems to store the infrastructure code permits us to apply changes anytime, responding to defects and business demands faster because we can always backward the infrastructure to a known state.
- Scalability: the combination of repeatability and automation allows us to increase our infrastructure in an easy and fast way.
- Consistency: repeatability and automation also guarantee that we will always have the same environment, as defined in the source code.
- Disaster recovery: as we have all the infrastructure defined as code, in case of a catastrophic event where we lose all the environment, it will be easy to recover and recreate it from our source code repository.

The most common issues that IaC addresses in the software industry are related to environment similarity and scalability. Regarding environment similarity, the IaC is helping companies increase the similarities between development, testing, and production environments and ensuring applications have the same behavior in any of them. It also avoids the differences that generally are present when creating the infrastructure by manual intervention. In terms of scalability, the approach is being used by companies that need a high level of dynamism in their infrastructure, like e-commerce. For example, IaC permits rapidly increasing the number of servers when the sales volume is growing and reducing them when the sales are decreasing, which is essential to control the costs [34].

The IaC approach appeared due to the evolution and growth of cloud computing demands. In general, this new approach is related to cloud-based environments, but, it can be used in on-premises infrastructure, as well. Even, it can be applied to isolated machines [35].

There are different ways and tools to implement IaC, depending on the need. The most common tools used in scientific environments are that related to the provision of virtual machines or containers, which embed the software environment, the dataset, and the source-code that composes a specific experiment [11][29].

Using IaC, researchers can write and execute code to define, deploy, update, and destroy the necessary infrastructure for their experiments. This code will be stored

in a version control system, making the experiment reproducible as many times as needed, by the originals and other researchers [10] [11].

VI. The Environment Code-First Framework (ECF)

Based on the findings presented before, there are three main issues related to software environments that support scientific research and directly impact on their irreproducibility. The first is the absence of proper documentation providing a step-by-step on how to reproduce the environment, including all software like libraries, packages, compilers, interpreters, databases, their versions, and how to install and configure them. The second is the dependency of a specific platform of software or hardware due to the use of the researchers made of their personal computers. Creating a computational environment in a specific machine with a specific operating system forces us to use the same platform to reproduce the same environment. The last one is the dependency hell. When a computational environment is built on a specific machine by a researcher in a manual way, it is being created with many dependencies on libraries, packages, and software versions that will exist only on that machine. It is almost impossible to reproduce the same environment on other machines, especially by other researchers different from the original. Producing unique computational environments in software and configuration is an issue called snowflake servers [15] or snowflake systems [34].

As discussed earlier, these issues have their origins in the following root causes arising from the cultural and behavioral aspects of the scientist developer:

- The use of the researchers made of their personal computers;
- Most of the time, the scientist end-user developer works alone or in small teams;
- The scientist end-user developer does not produce documentation;
- Researchers are not interested in software engineering best practices because their focus is on the research. The applications developed by the scientist end-user developer are just tools that help him obtain and process the data they need to analyze.

Also, some of the root causes identified have characteristics that contribute to the manual intervention of the researchers when creating the computational environments that support their research work. The practice of the installation and configuration of the environment in a manual way can be considered another root cause that aggravates the second and the third issues mentioned before.

In this paper, a framework is being proposed to mitigate these issues and to conduct the researchers to create self-contained computational environments more reproducible, isolated, portable, and independent of any platform of software and hardware.

The framework shall drive the development of an environment that is:

- Independent of hardware and software platform, regarding operating systems;
- Ready to run on-premise, on a PC or more powerful servers, or even in the cloud;
- Fully provisioned programmatically, with no installations and configurations manually performed, using a repository (e.g., Github or GitLab) to store the source code of the environment;
- Dynamic in terms of software resources, allowing the addition or remotion of them from the environment's source code at any time, and in a fast way;
- Storable in small files, in Megabyte order, not Gigabyte order, without the need for exhaustive downloads;
- Quickly reproducible and ready to use in the order of minutes.

The framework has two parts. The first part determines the architecture of the computational environment, and the second is a guide that defines a step-by-step procedure that must be followed by the researcher for the development and maintenance of the infrastructure.

A. The ECF Architecture

The main goal of the architecture defined by the ECF is to create a homogeneous environment independent of the hardware and software platforms used by the researcher. For example, a team of five researchers using various types of PCs (e.g., notebooks and desktops) with different operating systems would still have the same environment in all machines when following the ECF framework.

In the last fifteen years, two main technical approaches had been used by the scientific community to create self-contained computational environments proper to reproducibility. The first one are the virtual machines (VMs) and the other are the containers. Both permit us to create packaged computing environments composed of many IT elements (e.g., CPU, memory, and storage) available in file format. When executed inside the host machine, both isolate their environments from the rest of the system. But, while the VMs offer complete isolation from the host operating system, the containers offer lightweight isolation. It occurs due to the use of containers made of the host machine's resources, while VMs have their operating system, CPU, memory, network interface, and storage. This is an advantage of the VMs in terms of security and a barrier in terms of availability and portability, because of the size of their files. The size of a container image file is generally measured in MB, while that of a VM can take several GB [36]-[38].

Nowadays, there is a growing adoption and use of Linux container technology (e.g., Docker, LXC) compared to virtual machines by the software industry and the scientific community. Many scientific papers have been published presenting executable paper solutions based on containers to help grow reproducibility and transparency in science [11][20][30][39][40]. However, the proposal presented by the most of papers related to this subject is usually to use the containers directly on the host machine. This can be a

challenge in terms of platform because the containers have to be compatible with the operating system [41]. Currently, Docker is considered the de facto standard for containerization [42]. The Linux containers based on this standard, only can run directly on machines that have a Linux distribution as the operating system. To run Docker Linux containers on other operating systems such as Microsoft Windows and MacOS, it is necessary to install and configure a set of software that will adapt them to support Linux containers, usually in a manual way [43].

The ECF defines an architecture composed of two modules to permit researchers to obtain precisely the same computational environment when reproducing a research work. The first, called Container Module (CM), is a Linux container with all the software, libraries, and packages needed to develop and run the scientific application. The second, called Virtual Machine Module (VMM), comprehends a hypervisor, a lightweight virtual machine based on Linux distribution, and a container engine (e.g., Docker). In the development phase, the modules will be developed separately. But, in the execution phase the CM will run inside the VMM, on top of the container engine layer. Practically, the CM will work as another layer of the CMM. As shown in Fig. 1, the layers in green represent the physical machine and the operating system installed on it. The other layers, that appears involved by a dotted line, are all part of the architecture defined by the ECF framework. Both modules have to be provisioned programmatically using IaC resources such as ad-hoc scripts, configuration management tools, server templating tools, orchestration tools, and provisioning tools.

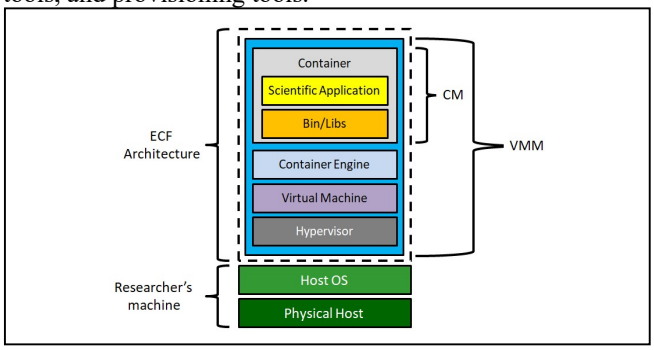

Figure 1. The ECF architecture.

Provisioning the computational environment based on the ECF architecture guarantees the container will always run on the same operating system, independent of the software platform used by the researchers on their physical machine, as shown in the example of Fig. 2.

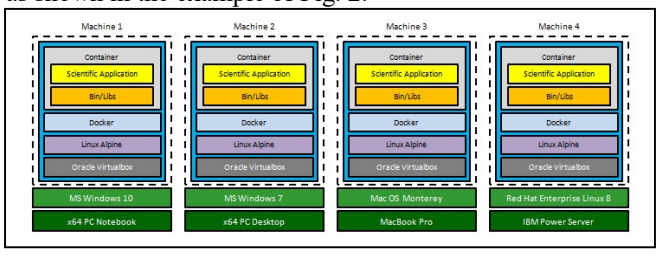

Figure 2. Example of the ECF architecture running on different platforms.

B. The ECF Guidence

The goal of this part of the framework is to guide the researchers on implementing the two modules defined by the ECF architecture, the CM and the VMM. The CM have to be implemented first, as it is one of the four layers that compose the VMM. For both modules, the framework establishes a series of steps that have to be systematically followed by the researchers to get each of them implemented.

1) The CM implementation guide

The implementation of the CM can be summarized in five high-level guidelines:

- Requirements identification;
- Development of the CM source code;
- Source code storage;
- Container image generation;
- Container image storage.

Fig. 3 shows the guidelines in a diagram where we can see in which order researchers have to perform such actions. It is essential to notice the researcher will not perform these actions only once but in a cycled way as many times as needed due to the maintenance of the environment. For example, after the environment is ready and running, if a researcher identify a need to add a new library, it will necessary execute all the steps to get the new version of the container image stored and available for download.

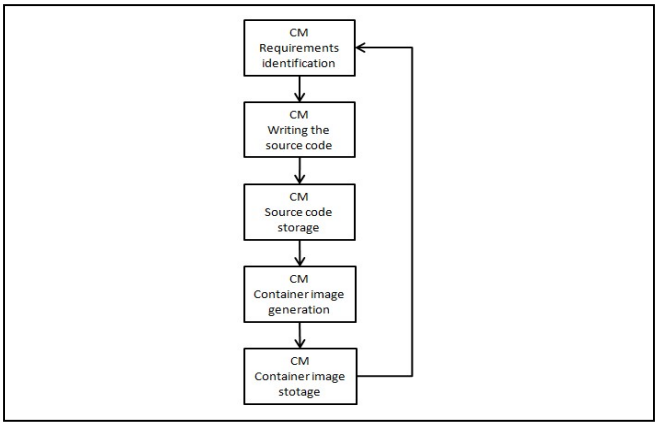

Figure 3. Steps involved in the CM development.

In the requirements identification step, the researcher will identify all the software, libraries, and packages necessary to develop and run the scientific application. The first requirement that has to be defined in this step is the container engine. It is mandatory to know what container engine will be used to build the environment before all the other requirements. The source code the researcher will write to define the installations and configurations to create the environment depends on this definition. It has to follow the patterns and syntax required by the chosen engine. The other requirements can vary from one environment to another, depending on the experiment's needs. The source code files must contain all the instructions and explanations needed to document the commands and configurations specified. The ECF framework defines a form model with a set of questions based on the most common types of software components

used in scientific environments that have to be answered by the researchers when analyzing the requirements to build the CM. Of course, this model must be adapted for each case, including or removing requisites according to the situation. Table I shows the form with the questions defined by the ECF. Besides the column with the question, there are two more columns in the form. One for the answer in itself, and the other two specify the software's version.

TABLE I. CONTAINER MODULE DEVELOPING FORM

Question	Answer	Version
Which container engine will be used?		
Which will be the base image of the containers?		
Which programming languages and compilers will be used?		
Which libraries, packages and third part software have to be installed?		
Which databases will be installed?		
Is it necessary to perform any configuration? In which files?		
Is it necessary to copy any files into the container? Which files?		

The next step consists of developing the container image's source code based on the requisites identified before. After writing the container image's source code, the researcher needs to store it in a version control system like Github and Gitlab. In the following step, it is necessary to compile the source code to generate the image used to create the container that supports the scientific application's development and execution. Also, it is necessary to perform some tests with the image. The last step is to store the image in a container repository (e.g., Docker Hub).

2) The VMM implementation guide

Similarly to the CM, the implementation of the VMM is composed by three high-level guidelines:

- Requirements identification;
- Development of the VMM source code;
- Source code storage.

The sequence the steps have to be executed is shown in Fig. 4. Although the diagram demonstrates that the steps can be performed cyclically, it is not typical for the VMM because it will not change with as much frequency as the CM can change. For example, it will be necessary to change the VMM when we have to increase the quantity of memory or the number of CPUs.

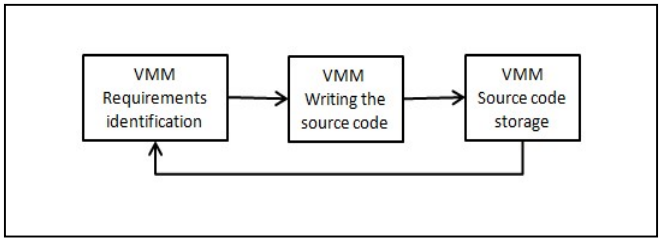

Figure 4. Steps needed to create the VMM.

The VMM is limited to four essential layers: the hypervisor, the virtual machine, the container engine and, the CM in itself. That means there is no need to add or remove any layer. In the requirements identification step, the researcher will have to define which hypervisor will support the virtual machine, how much memory it will use, how many CPUs it will have, and which operating system will be installed. At this point, the container engine is already known, and the CM is ready to use. The other definitions are related with the IaC tools the researcher wants to use to develop the VMM. Also, the ECF defines a form with the main questions to guide the researcher in this phase. It can be visualized in Table II.

TABLE II. VIRTUAL MACHINE MODULE DEVELOPING FORM

Question	Answer	Version
Which hypervisor will be used?		
How much memory will be allocated for the virtual machine?		
How much CPUs will be dedicated to the virtual machine?		
Which operating system will be installed on the virtual machine?		
Which IaC tools will be used to automate the provisioning of the environment?		

At this point, the researcher has all the elements that is necessary to develop the source code of the VMM. The source code must guarantee the hypervisor installation on the physical machine, the provisioning of the virtual machine with the container engine and the CM inside it. The documentation about the actions performed to create the VMM have to be included in the source code files. The source code produced in this phase must be stored in a version control system, but the virtual machine image does not. It will be used only to create an instance of the container image that represents the scientific environment. In this way, the virtual machine image only has to keep stored locally, and it can be destroyed and recreated as many times as needed. During the initialization, the VMM has to check if a new version of the CM is on the image repository. In a positive case, it needs to download the new container image before instantiating it. Once the VMM source code covers all these actions and is already available in a version control system, any researcher can use the produced scripts to recreate an environment.

In Fig. 5, a flowchart shows all the steps involved in running a VMM script on provisioning an entire scientific computational environment.

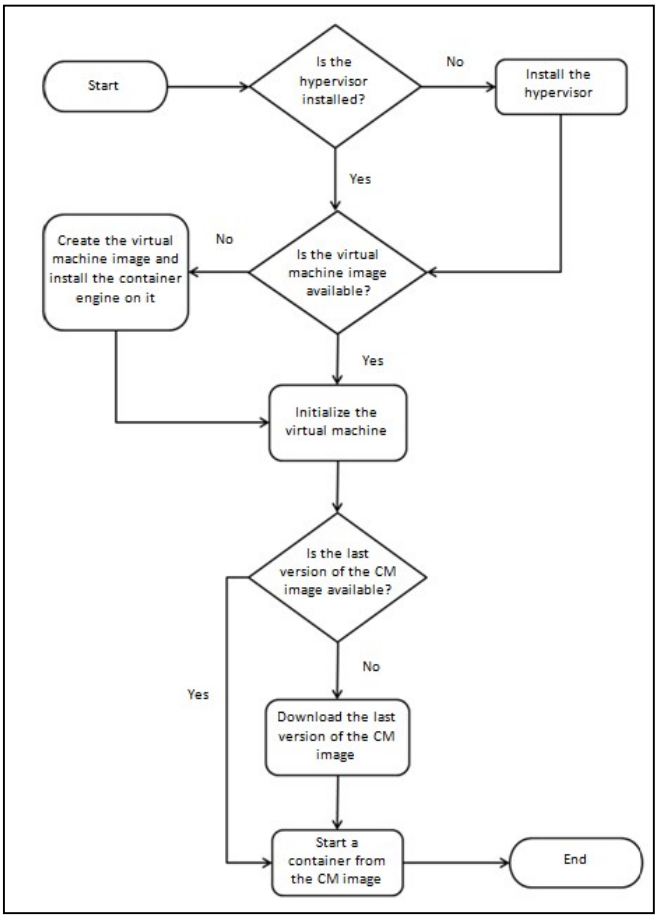

Figure 5. Steps performed by the VMM scripts.

VII. Case Study: The Prokaryotic Genomics and Comparative Genomics Analysis Pipeline (PGCGAP)

In this section, we describe our experience in recreating a computational environment called PGCGAP, the Prokaryotic Genomics and Comparative Genomics Analysis Pipeline, following the guide presented by H. Liu et al. in [44]. Also, we describe how we recreate the same environment following the guidelines defined by the ECF framework.

A. Material and Methods

We performed both implementations on three different physical machines. The machine one (M1) is a PC notebook configured with Microsoft Windows 10 Professional Edition 64-bit operating system, an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz processor, 16 GB of RAM and a hard disk 512 GB SSD. The machine two (M2) is a PC notebook configured with Microsoft Windows 10 Home Edition 64-bit operating system, an Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz processor, 16 GB of RAM and a hard disk 512 GB SSD. The last one, machine three (M3), is a PC notebook configured with Linux Fedora v34 64-bit operating system, an Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz processor, 16 GB of RAM and a hard disk 512 GB SSD.

In [44], the authors split the paper that presents the PGCGAP into two parts. The first part is related to the step-by-step defined by the protocol to provisioning the pipeline. The second part uses the pipeline provisioned to execute a bioinformatics application, processing a significant amount of data and generating results. As our focus is on providing the computational environment, we considered only the first part of the paper in our analysis and comparison.

The comparison of the two implementations considered time consumption, efforts, manual intervention, platform-agnosticism, and portability.

B. Implementing the Original PGCGAP

The PGCGAP is a protocol that guides researchers on implementing a scientific computational environment that supports applications related to genomics and comparative genomics analyses of microbes. This protocol comprises a set of genomic analysis software packages, scripts developed by the PGCGAP's creators, and a guide that specifies configurations and software installations that have to be performed by the researchers in a correct sequence to reproduce the environment. The PGCGAP is free and open-source software licensed under GPLv3. All the third-party software packages used to create the protocol are open source. The source code is available on Github [45].

In the paper, the authors demonstrate the protocol's applicability on a Linux Ubuntu 18.04 operating system. But, they used a machine configured with Microsoft Windows 10 and a feature called Windows Subsystem for Linux (WSL) [46] to create the virtual machine that supported the PGCGAP implementation. The Linux Ubuntu 18.04 OS was installed from the Microsoft Store.

As mentioned before, we reproduced the steps of the paper on three different physical machines. The M1 PC notebook is configured with Microsoft Windows 10 x64 OS, and it has the version 1 of the WSL installed. The M2 is configured with Microsoft Windows 10 x64 OS and WSL version 2. On both PCs were created virtual machines with Linux Ubuntu 18.04 OS installed from the Microsoft Store. Even though the authors did not specify if they used version 1 or 2 of the WSL, we decided to test the PGCGAP on both to verify if it can work adequately on any version. Regarding the M3 PC notebook, it is configured with Linux Fedora v34 x64 OS. We considered testing the protocol on the M3 machine to extend the research and check if it can work correctly on other Linux distributions that are not running on top of the Microsoft Windows Subsystem for Linux.

The PGCGAP protocol defines twelve steps that have to be executed by a researcher when provisioning the environment. These steps include installations of the WSL, the Linux OS, and third-party package software like Miniconda and the PGCGAP in itself. Also, it includes configurations that have to be included in files and performed in a command-line terminal. The authors reported that all the steps were performed in around sixty minutes.

On M1 PC, the provisioning of the PGCGAP environment failed. It presented a corruption error during the eleventh step that corresponds to the setup of the COG database.

All the steps to provision the PGCGAP environment were successful on M2 PC. It was necessary 80 minutes to perform the entire procedure. But, it is essential to highlight that it was needed 30 minutes more to install and configure version 2 of the WSL before executing the PGCGAP steps [47]. This was not an action foreseen by the protocol, but if we consider it, the whole process took 110 minutes in total. In terms of portability, we exported an image of the virtual machine created by the WSL to a zip file around 13.60 GB. Using this file, we could import the virtual machine on a fourth PC notebook (M4), used in this part of the experiment only for the portability test. It was configured with Microsoft Windows 10 operating system, WSL version 2, an Intel(R) Core(TM) i7-9850H CPU @ 2.60GHz processor, 16 GB of RAM and a hard disk 512 GB SSD. After to import the virtual machine from the zip file, it was possible to initialize the PGCGAP environment successfully.

The provision of the PGCGAP on M3 PC was performed successfully, as well. To execute all the steps on this machine it was needed 81 minutes. This is a very close result that we obtained on M2 machine for this part of the provision. In this case, there was no extra installations and configurations to be considered. The portability of the PGCGAP from this machine is not possible, once the environment was provisioned directly on a physical machine.

The implementation of the PGCGAP protocol on the three machines mentioned before was performed manually, according to the steps presented in [44].

C. Implementing the PGCGAP Based on the ECF

For the development of the CM and VMM that will be described in this topic, we used the M4 PC notebook, already mentioned and detailed before.

As defined by the ECF framework, the first step on creating a computational environment is to develop the container module following the CM implementation guide.

For the analysis phase, it was necessary 30 minutes to review and fill the form with the requirements of the environment once most of them are described in [44]. Table III shows the form with the questions and answers used to develop the container module for the PGCGAP environment.

TABLE III. PGCGAP'S CM FORM

Question	Answer	Version
Which container engine will be used?	Docker and dependencies	20.10
Which will be the base image of the containers?	Ubuntu Linux	18.04
Which programming languages and compilers will be used?	Python and dependencies	3.7.6
Which libraries, packages and third part software have to be installed?	Miniconda and dependencies	3
Is it necessary to perform any configuration? In which files?	Add into .bashrc: export OMPI_MCA_opal_cuda_support=true	N/A
Is it necessary to copy any files into the container? Which files?	pgcgap_latest_env.yml	N/A

Based on the form shown in Table III, we could write the source code that defines the infrastructure needed to develop and run the scientific application. Practically, we developed the Dockerfile, a prerequisite necessary to generate the Docker container that will embed all the PGCGAP computational environment. Also, the Dockerfile works as part of the documentation. For programming and documenting the environment, it was necessary around 60 minutes.

The next step consisted of generating the container image based on the definitions of the Dockerfile. Docker performed this operation in 49 minutes, and the final base image file had a size of 8.14 GB. With the container's image ready to be used, it was necessary to execute a set of tests to ensure that a properly PGCGAP environment was being provisioned. We had to test the upload of the image to the Docker Hub, its download from the Docker Hub, and the instantiation of containers from the downloaded base image. Also, we ran some commands on the PGCGAP environment to verify that all the components were adequately installed. The image test operation was performed in 85 minutes.

Considering all the phases executed in the CM development, achieving a successful result took around 224 minutes.

With the container module working correctly, we started to work on the virtual machine module following the VMM guide. Initially, we analyzed the requirements needed to create a virtual machine capable of supporting the PGCGAP container, considering the bioinformatics profile of the applications that will run on this environment. Despite the ECF framework being tool-agnostic, it was designed for Linux containers. In this context, we opted to use only free and open-source software tools commonly used by the scientific community, as shown in Table IV. This step was performed in 30 minutes.

TABLE IV. PGCGAP'S VMM FORM

Question	Answer	Version
Which hypervisor will be used?	Virtualbox	6.1.32
How much memory will be allocated for the virtual machine?	8 GB	N/A
How much CPUs will be dedicated to the virtual machine?	2 CPUs	N/A
Which operating system will be installed on the virtual machine?	Ubuntu Linux	18.04
Which IaC tools will be used to automate the provisioning of the environment?	Vagrant and dependencies	2.2.19
	Ansible and dependencies	2.12.2
Other resources	Shell-scripts Linux (main script)	N/A
	Shell-scripts Windows (main script)	N/A

For this module, the first step was to implement a script used to start the PGPGAP environment, the main script. As our intention was to perform tests on Linux and MS-

Windows machines, we had to develop the main script for both operating systems. The script verifies if the PC has the Virtualbox installed. If not, the hypervisor is downloaded and installed on the machine. After, it verifies if Vagrant and Ansible are installed. In the negative case, the installation is performed, followed by the provisioning of the virtual machine configured with Ubuntu Linux. Otherwise, Vagrant only will start the virtual machine. In order to permit Vagrant to create the virtual machine, we had to specify the configurations and installations required in a file called Vagrantfile, as shown in Fig. 6. In this file, we defined the amount of RAM and the number of CPUs must be allocated for the VM. Also, we requested the installation of the Docker engine using Ansible. Having the VM running up, the script downloads the CM from the Docker Hub, if needed, and starts a container that embeds the PGCGAP environment.

```
##########################################################
# This Vagrantfile defines the configuration that will  #
# be used by the virtual machine module (VMM) to        #
# demonstrate the provision of the PGCGAP Environment   #
# through the implementation based on the               #
# Environment Code-First Framework                      #
##########################################################
Vagrant.configure("2") do |config|

  # Define the virtual machine image
  # Ubuntu Bionic 18.04 64 bits
  config.vm.box = "hashicorp/bionic64"

  # Define the hostname and the network
  config.vm.define :ecf do |ecf_config|
    ecf_config.vm.hostname = "ecf"
    ecf_config.vm.network :private_network,
                          :ip => "192.168.33.10"

    # Define the Ansible configuration
    ecf_config.vm.provision "ansible_local" do |ansible|
        ansible.playbook = "bio_ecf.yml"
        ansible.verbose = "vvv"
    end

    # Define the provider (virtualbox), quantity of memory,
    # and how many CPUs will be used on the VM
    config.vm.provider "virtualbox" do |domain|
        domain.memory = 8192
        domain.cpus = 2
    end
  end
end
```

Figure 6. Vagrantfile implemented for the virtul machine module.

The codification of all parts that compose the VMM, considering the scripts for Linux and Windows and the configuration files for Vagrant and Ansible, took around 280 minutes to be concluded. The scripts developed in this phase had to be tested individually and together, consuming around 350 minutes. This time includes the tests performed with the CM and the VMM together. Considering the time needed to implement both modules, CM and VMM, the total time was 884 minutes.

After the implementation of the CM and the VMM, we were ready to start the tests on the three PC notebooks described before: M1, M2 and M3. We started downloading the main script from the repository for both operating systems, MS-Windows and Linux. Actually, it is the only file that has to be downloaded manually by a researcher that wants to recreate the environment. The PGCGAP environment was provisioned automatically and succesfully by running this script on the three machines. Considering a scenario where all the software components (e.g., Virtualbox, Ansible, Vagrant, CM) had to be downloaded to make the environment available, this operation took 92 minutes on M1, 89 minutes on M2, and 95 minutes on M3. It is essential to highlight that these times can vary depending on the download capacity of the internet connection used to obtain the CM and the VMM. Both of them have to be downloaded, and, in this case, it was used an internet connection with 27 Mbps of download speed.

VIII. Discussion

First of all, it is essential to clarify that the ECF framework's primary goal is to enhance the reproducibility using the IaC approach. In this way, it focuses on guiding original researchers in providing a computational environment that is easily reproducible by them and other researchers. By them, when new members have to be integrated into a research team, for example. And by other researchers when they want to reproduce published results of a research papers. Of course, to enhance reproducibility and develop means that allow to recreate computational environments efficiently, a great effort from the researchers responsible for the provision in terms of learning and programming IaC tools will be necessary. The comparison presented in this topic can not be interpreted literally, but from two points of view, one from the original researcher that is creating the environment by programmatic ways, and another from the researcher that is reproducing it. The work presented in [44], only shows the second perspective.

In terms of time consumption, from the point of view of the original researcher, it was necessary around fifteen hours to build the entire computational environment that supports the PGCGAP usinf the ECF framework. It is essential to highlight that it was designed to assist those directly involved in research development and anyone who wants only to run an application and verify published results. From this second point of view, the effort necessary would be simply downloading and executing only one script to get the computational environment ready to use. Our practical test on this operation consumed 92 minutes, on average, considering the three machines used in the tests (M1, M2, and M3). By following the method presented in [44], our provisioning of the environment took on average 80.5 minutes, remembering that we did not have success in implementing it on the M1 PC notebook. The authors reported in [44] a total time of 94 minutes to provisioning it.

Another concern of the ECF framework is reducing the manual intervention when provisioning computational environments. Our experience in reproducing the original PGCGAP environment proved that, by following this method, all the steps must be performed manually. From enabling the WSL resource on MS-Windows operating system, installing the Ubuntu operating system, downloading and installing the Miniconda platform, adding configuration on some specific Linux files until the installation of the PGCGAP, all of them were performed in a manual way. And, this is not a good practice when trying to produce reproducible environments. On the contrary, manual intervention is one of the leading causes of issues like the

snowflake servers and snowflake systems mentioned earlier. Implementing the PGCGAP based on the guidelines of the ECF framework showed us that it is possible to provide an entire computational environment automatically and programmatically, reducing the manual intervention to a download and execution of only one script.

The platform-agnosticism is another relevant topic that is covered by the ECF framework. The framework focuses mainly on the three most used operating systems by scientific researchers: Linux, MS-Windows and MacOS. But, it does not exclude other platforms like HPC and cloud computing. Once the framework defines a standard layer represented by a virtual machine, the only condition to run an ECF environment is to support virtualization. This abstraction turns different platforms in one common platform using the same distribution of the Linux operating system. The Linux container representing the computational environment can be instantiated from this point, permitting that the applications consistently produce the same results. The PGCGAP environment implemented by the ECF framework could be provisioned successfully on the three machines used in our experiments. Differently, the original PGCGAP could be provisioned only on two of the three machines due to an issue related to version 1 of the WSL resource, which is part of the MS-Windows and is a vital tool of this proposal.

In terms of portability, as the ECF framework was designed to be tool-agnostic, one implementation can be more portable than another depending on the way they were implemented. For example, using Python instead of operating system scripts to develop the VMM produces a more portable environment. In our case, we decided to use shell scripts to implement the main script of the VMM due to our high knowledge of this subject. Choosing another programming language like Python would require more time to learn, implement and test this deliverable. In this way, we had to create one main script for Linux and another for the MS-Windows platform because the hypervisor (Virtualbox) installation is different on both operating systems. All the source code produced is common for any platform from this point ahead. The configuration files created for Vagrant, Ansible, and Docker, for example, will be the same on provisioning the environment for any operating system. The main script is the only deliverable that needs to be download and executed manually by a researcher that wants to reacrete the environment. When it is executed, all the softwares that compose the infrastructure (e.g., Virtualbox, Vagrant, and Ansible) are downloaded and installed automatically from official repositories. The CM is downloaded from the Docker Hub. That means the environment is always provisioned with the same components and versions obtained from the same sources. It can be installed on physical machines or infrastructures supported by cloud computing. Based on what we produced in our experiment, it is limited to Linux and MS-Windows. But, we can extend this coverage only by creating the main script for other platforms (e.g., MacOS, Solaris). Once the original researchers provided access to the source code of the environment, it is possible yet, for other researchers to reproduce it by compiling this code. This practice is another advantage of the ECF framework. There is no need to perform the portability manually, using traditional means like copying a file from one computer to another. On the other hand, this is the only way presented by the original PGCGAP, considering that both machines have MS-Windows 10 operating system and WSL version 2 installed. In this case, it is important to remember that, in our experiment, WSL generated a file with 13.60 of size in the export operation. The container image file that represents the CM had 8.14 GB. A difference of 40.14% between them.

The documentation is another positive point of using the ECF framework to develop the PGCGAP environment. As the framework uses the IaC approach, all the components of the environment are based on code. In this way, we described and explained the environment, installations, and configurations, into the source-code files we had created for Vagrant, Docker, Ansible, and the shell scripts. The documentation is fundamental to guide those who need to recreate the environment, but also for anyone that wants to understand how the environment was developed.

Of course, we can not show only the benefits of the ECF framework. There are costs on adopting it. First of all, it is important to highlight the time that the original researchers must dedicate to programming the components of the environment. As mentioned earlier, it took about fifteen hours to develop and test the source code, given our high level of expertise in the programming languages and tools used in the experiment and the Infrastructure-as-Code approach. The development and test phases tend to be higher when researchers are not information technology specialists (e.g., biologists, chemists, and archeologists). This implies the factor that we consider to be the second higher cost: the time and effort needed to learn about the programming languages, IaC tools and software engineering practices.

IX. CONCLUSION

In [44], the authors presented a step-by-step on recreating an entire computational environment that supports scientific research. That is what we expect from an executable paper in terms of documentation. However, it is based on a manual intervention approach which is not good reproducibility practice. When recreating an environment, increasing reproducibility implies substituting the actions manually performed by automatic means.

The framework proposed in this paper has as primary intention to help researchers enhance the reproducibility of their work regarding the provision of the computational environment. Our study contributes by mitigating typical issues such as the absence of documentation, platform dependency, and manual intervention. The central idea is provisioning the environment based on the Infrastructure-as-Code approach instead of using traditional means. The pre-defined homogeneous architecture permits us to have a big picture of the environment. And the practical guidelines conduct researchers on developing environments that are more reproducible, isolated, portable, and independent of any software and hardware platform.

Our goal in proposing the ECF framework is to support the development of new research works and help researchers

migrate papers already published based on traditional provisioning approaches to this new way easily reproducible.

In the case study presented, the ECF framework proved in practice that it is possible to provide an entire computational environment programmatically with minimal intervention of the researcher who wants to recreate it. In the results, we can testify its benefits, especially regarding the behavior of the environment that was the same on the three machines used in our experiment. This kind of evidence allows us to understand how valuable it is to mitigate manual intervention because we will always have the same environment many times as we reproduce it.

The adoption of the ECF framework has costs. Firstly, we have to highlight the time and effort from the scientist-developer side when producing and testing the code that will generate the environment. These activities require them to dedicate a lot of time and attention. Besides, it is necessary a continuously learn behavior from the scientist-developer, which we consider the second cost when adopting the framework. Developing competence in software engineering practices, programming languages, and IaC tools is challenging and time-consuming. But, it is also a gain for researchers once they are preparing themselves for a new era based on open science principles.

We recommend using open source software for researchers who desire to implement their computational environments following the ECF guidelines. Besides being aligned with the open science principles, open source tools in the general count with large communities supporting the users, accelerating the learning process, and helping them when they face technical problems. The use of mature tools already approved by the scientific community (e.g., Docker, Virtualbox, and Python) is also recommended. Compared with more recent tools, they tend to have fewer technical issues, and there is more documentation and forums to guide new users.

We consider a significant contribution of the ECF framework the educational role that it can have in helping to prepare future generations of researchers on creating more transparent and reproducible research that aggregates value and benefits the scientific community.

As future work, we propose implementing computational environments for different domains of science to help to improve the ECF framework.

REFERENCES

[1] D. A. Gomes, P. Mestre, and C. Serôdio, “Infrastructure-as-Code for Scientific Computing Environments,” CENTRIC 2019: The Twelfth International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services, Nov. 2019.

[2] J. A. Papin, F. Mac Gabhann, H. M. Sauro, D. Nickerson, A. Rampadarath, “Improving reproducibility in computational biology research,” PLoS Comput Biol 16(5): e1007881, 2020, https://doi.org/10.1371/journal.pcbi.1007881.

[3] B. A. Nosek et al., “Promoting an Open Research Culture,” Science, New York, N.Y., 348, 2015.

[4] L. P. Freedman, I. M. Cockburn, T. S. Simcoe, “The Economics of Reproducibility in Preclinical Research,” PLOS Biology 13(6): e1002165, 2015, https://doi.org/10.1371/journal.pbio.1002165.

[5] European Commission. Reproducibility of scientifc results in the EU. [Online] Available from: https://op.europa.eu/en/publication-detail/-/publication/6bc538ad-344f11eb-b27b-01aa75ed71a1. [Accessed: 2022.05.31].

[6] J. R. F. Cacho and K. Taghva, “The State of Reproducible Research,” in Computer Science, 17th International Conference on Information Technology – New Generations (ITNG 2020), Advances in Intelligent Systems and Computing, Volume 1134, Springer, 2020.

[7] M. Munafò et al., “A manifesto for reproducible science,” Nat Hum Behav 1, 0021, 2017.

[8] J. C. Burgelman et al., “Open Science, Open Data, and Open Scholarship: European Policies to Make Science Fit for the Twenty-First Century,” Frontiers in Big Data, Volume 2, 2019.

[9] J. R. F. Cacho and K. Taghva, “Reproducible research in document analysis and recognition,” in Information Technology-New Generations, Springer, Berlin, pp. 389–395, 2018.

[10] M. Baker, “1500 scientists lift the lid on reproducibility,” Nature News 533 (7604), 452, 2016.

[11] C. Boettiger, “An introduction to Docker for reproducible research, with examples from the R environment,” ACM SIGOPS Oper. Syst. Rev., 49, 2014.

[12] S. M. Powers and S. E. Hampton, “Open science, reproducibility, and transparency in ecology,” Ecological Applications 29(1):e01822, 2019.

[13] D. L. Donoho, “An invitation to reproducible computational research,” Biostatistics (Oxford, England), 11, 2010, pp. 385-8.

[14] A. Brinckman et al., “Computing environments for reproducibility: Capturing the “Whole Tale”,” Future Generation Computer Systems, Volume 94, pp. 854-867, 2019.

[15] K. Morris, “Infrastructure as Code: Managing Servers in the Cloud,” 1st ed. O’Reilly Media, Inc., 2016.

[16] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, “Measuring Reproducibility in Computer Systems Research,” Department of Computer Science, University of Arizona, Technical Report. [Online]. Available from: http://reproducibility.cs.arizona.edu/tr.pdf [Accessed: 2022.05.31]

[17] T. Glatard, L. B. Lewis, R. Ferreira da Silva, R. Adalat, N. Beck, C. Lepage, and A. C. Evans, “Reproducibility of neuroimaging analyses across operating systems,” Frontiers in Neuroinformatics, 9, 12, 2015, doi:10.3389/fninf.2015.00012.

[18] C. Riccomini, D. Ryaboy. “The Missing README: A Guide for the New Software Engineer,” 1st ed. No Starch Press, 2021.

[19] D. C. Ince, L. Hatton, and J. Graham-Cumming, “The case for open computer programs,” Nature. 2012; 482(7386):485-488. Published 2012 Feb 22.

[20] B. Marwick, “Computational Reproducibility in Archaeological Research: Basic Principles and a Case Study of Their Implementation,” J Archaeol Method Theory 24, 424–450, 2017, https://doi.org/10.1007/s10816-015-9272-9.

[21] J. Segal and C. Morris, "Developing Scientific Software" in IEEE Software, vol. 25, no. 04, pp. 18-20, 2008, doi:10.1109/MS.2008.85.

[22] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl and G. Wilson, "How do scientists develop and use scientific software?," 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering, 2009, pp. 1-8, doi:10.1109/SECSE.2009.5069155.

[23] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana, “A survey of scientific software development,” In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10). Association for Computing Machinery, New York, NY, USA, Article 12, 1–10, 2010, doi:https://doi.org/10.1145/1852786.1852802.

[24] Z. Merali, “Computational science: Error, why scientific programming does not compute,” Nature 467, 7317, 2010, https://doi.org/10.1038/467775a.

[25] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, J. C. Carver, "Software engineering practices for scientific software development: A systematic mapping study," Journal of Systems and Software, Volume 172, 2021, 110848, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2020.110848.

[26] Á. L. García and E. F. del Castillo, "Analysis of scientific cloud computing requirements," in Proceedings of the 7th Iberian Grid Infrastructure Conference, Madrid, Spain, Sep. 2013, pp. 147-158.

[27] B. S. Cole and J. H. Moore, "Eleven quick tips for architecting biomedical informatics workflows with cloud computing," PLOS Computational Biology, vol. 14, issue 3, Mar. 2018.

[28] D. Clark, A. Culich, B. Hamlin, and R. Lovett, "BCE: Berkeley's Common Scientific Compute Environment for Research and Education," in Proceedings of the 13th Python in Science Conference, Austin, USA, Jul. 2014, pp. 5–12.

[29] B. Howe, "Virtual Appliances, Cloud Computing, and Reproducible Research," Computing in Science & Engineering, vol. 14, no. 4, pp. 36-41, Jul.-Aug. 2012.

[30] J. R. Fonseca and K. Taghva. "Reproducible Research in Document Analysis and Recognition," Advances in Intelligent Systems and Computing: Information Technology - New Generations, Springer, 738 389-395, 2018

[31] Amazon Web Services. Introduction to DevOps on AWS, White Paper, 2014. [Online]. Available from: https://d1.awsstatic.com/whitepapers/DevOps/infrastructure-as-code.pdf [Accessed: 2022.05.31]

[32] Amazon Web Services. Infrastructure as Code, White Paper, 2017. [Online] Available from: https://d1.awsstatic.com/whitepapers/DevOps/infrastructure-as-code.pdf [Accessed: 2022.05.31]

[33] S. Nelson-Smith, "Test-Driven Infrastructure with Chef," 2nd ed. O'Reilly Media, Inc., 2013.

[34] K. Morris, "Infrastructure as Code: Dynamic Systems for the Cloud Age," 2nd ed. O'Reilly Media, Inc., 2020.

[35] IBM Cloud Education. Infrastructure as Code, Blog, 2019. [Online] Available from: https://www.ibm.com/cloud/learn/infrastructure-as-code [Accessed: 2022.05.31]

[36] IBM Cloud Education. Containers vs. Virtual Machines (VMs): What's the Difference?, Blog, 2021. [Online] Available from: https://www.ibm.com/cloud/blog/containers-vs-vms [Accessed: 2022.05.31]

[37] Red Hat Topics Website. Containers vs VMs, 2020. [Online] Available from: https://www.redhat.com/en/topics/containers/containers-vs-vms [Accessed: 2022.05.31]

[38] Microsoft Technical Documentation Website. Containers vs. virtual machines, 2021. [Online] Available from: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm [Accessed: 2022.05.31]

[39] K. Wiebels and D. Moreau, "Leveraging Containers for Reproducible Psychological Research." Advances in Methods and Practices in Psychological Science, Apr. 2021, doi:10.1177/25152459211017853.

[40] X. Qiao, Z. Li, F. Zhang, D. P. Ames, M. Chen, E. J. Nelson and R. Khattar, "A container-based approach for sharing environmental models as web services," International Journal of Digital Earth, 14:8, 2021, 1067-1086, doi:10.1080/17538947.2021.1925758.

[41] Red Hat Topics Website. What's a Linux container?, 2018. [Online] Available from: https://www.redhat.com/en/topics/containers/whats-a-linux-container [Accessed: 2022.05.31]

[42] J. Sparks, "Enabling Docker for HPC," Concurrency and Computation: Practice and Experience, Dec. 2018, https://doi.org/10.1002/cpe.5018.

[43] Docker Documentation Website. Docker Desktop overview. [Online] Available from: https://docs.docker.com/desktop/ [Accessed: 2022.05.31]

[44] H. Liu et al., "Build a bioinformatics analysis platform and apply it to routine analysis of microbial genomics and comparative genomics," Protocol Exchange, 2020, https://doi.org/10.21203/rs.2.21224/v5.

[45] Github of the Project PGCGAP. [Online] Available from: https://github.com/liaochenlanruo/pgcgap [Accessed: 2022.05.31]

[46] Official documentation of the Windows Subsystem Linux. [Online] Available from: https://docs.microsoft.com/en-us/windows/wsl/ [Accessed: 2022.05.31]

[47] Windows Subsystem Linux's documentation on how to migrate from version 1 to version 2. [Online] Available from: https://docs.microsoft.com/en-us/windows/wsl/install-manual/ [Accessed: 2022.05.31]