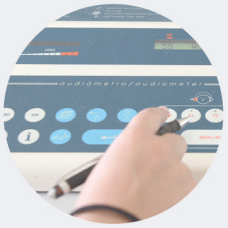
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA BIOMÉDICA






Desenvolvimento de um interface gráfico flexível para o Grupo de Evolução Fenotípico do Pegi3s

DIOGO JOSÉ DE OLIVEIRA ADÃO
julho de 2024

P.PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

peg 3s

Development of an adaptable graphical interface for the Pegi3s Phenotypic Evolution Group

-Diogo José de Oliveira Adão-

Dissertação para obtenção do Grau de Mestre em Engenharia Biomédica

Orientador: Luís Filipe Coelho

Júri:

Presidente:

[Nome do Presidente, Categoria, Escola]

Vogais:

[Nome do Vogal1, Categoria, Escola]

[Nome do Vogal2, Categoria, Escola] (até 4 vogais)

Resumo

Os dados de sequência, que representam genomas completos, exomas ou transcriptomas, são cada vez mais importantes na investigação e prática biomédica. No entanto, a contínua falta de bioinformáticos desde a década de 1990 representa um grande desafio. Consequentemente, aplicações de bioinformática fáceis de usar, adaptadas a investigadores sem formação em informática, são essenciais.

No contexto do software, as tecnologias de contentores Linux, particularmente o Docker, oferecem uma solução devido às suas numerosas vantagens. As imagens Docker fornecem pacotes de software prontos a usar, com todas as dependências necessárias instaladas, facilitando a implementação em qualquer sistema operativo que tenha o Docker instalado. Isto elimina a tarefa muitas vezes difícil e, por vezes, impossível de instalar dependências de software. As imagens Docker são imutáveis, aumentando a replicabilidade, e facilitam o desenvolvimento de pipelines complexos, aplicações autónomas com interfaces gráficas e bases de dados. Estas características ajudam a resolver a crise de replicabilidade nas análises bioinformáticas.

O Pegi3s (Phenotypic Evolution Group) é responsável por manter o Projeto de Imagens Docker de Bioinformática, uma coleção crescente com mais de 100 imagens Docker de softwares tipicamente usados nos campos de genética, transcriptómica, proteómica, filogenética, manipulação de sequências, entre outros.

Durante o desenvolvimento deste projeto, foi criada uma nova máquina virtual com uma versão mais recente do Ubuntu, em relação à máquina virtual anteriormente utilizada. Devido a algumas atualizações de software, tornou-se necessário desenvolver um software que permitisse ao utilizador gerir as suas próprias imagens Docker (Docker Manager).

Todas as imagens do pegi3s foram revistas e foi criado um conjunto padronizado de Dados de Teste para a maioria delas. Com estes Dados de Teste, foram obtidos os resultados da execução da maioria dos programas. Com a informação obtida, foi criado o “DocNRun pegi3s”, um software que permite ao utilizador executar imagens Docker sem utilizar a Linha de Comando.

Atualmente, tanto o Docker Manager como o DocNRun estão nos repositórios do pegi3s disponíveis para qualquer pessoa descarregar.

Palavras-chave: pegi3s, Docker, GUI, Replicabilidade

Abstract

Sequence data, representing full genomes, exomes, or transcriptomes, is increasingly important in biomedical research and practice. However, the ongoing shortfall of bioinformaticians since the 1990s poses a major challenge. Consequently, easy-to-use bioinformatics applications tailored for researchers without an informatics background are essential.

In the software context, Linux container technologies, particularly Docker, offer a solution due to their many advantages. Docker images provide ready-to-use software packages with all necessary dependencies installed, making deployment easy on any operating system with Docker. This eliminates the often difficult and sometimes impossible task of installing software dependencies. Docker images are immutable, enhancing reproducibility, and facilitate the development of complex pipelines, standalone applications with GUIs, and databases. These features help address the reproducibility crisis in bioinformatics analyses.

Pegi3s (Phenotypic Evolution Group) is responsible for maintaining the Bioinformatics Docker Images Project, a growing collection with over 100 Docker images of software typically used in the fields of genomics, transcriptomics, proteomics, phylogenetics, sequence manipulation, among others

During the development of this project, a new virtual machine was developed with a newer version of Ubuntu relative to the previously used virtual machine. Due to some software updates, it became necessary the development of a software that allowed the user to manage his images (Docker Manager).

All images from the pegi3s were reviewed and a standardized set of Test Data was created for most of them. With this Test Data, it was obtained the results for the execution of most programs. With the obtained information, “DocNRun pegi3s” was created, a software that allows the user to run Docker images without using the Command Line.

Currently, both Docker Manager and DocNRun exist in the pegi3s repositories available for anyone to pull.

Keywords: pegi3s, Docker, GUI, Replicability

ACKOWLEDGMENTS

It is with a great sense of satisfaction and fulfilment that I deliver this work. The completion of this dissertation marks the end of an important stage in my life and as such, I would like to thank all those who, in one way or another, made the completion of this report possible.

First of all, I want to thank my parents for all the help they have given me over the past few years and for all the motivation they have provided. Without them, I would never have achieved what I have achieved.

I want to thank my brother for all the times he served as a confidant and with whom I could always speak freely.

I would also like to thank my supervisors from i3s, Jorge Vieira and Cristina Vieira for always making me feel included and motivated. Without their support this project would never come to fruition. This thanks also include all other lab members for all their assistance and input, making this project what it is today.

Next, I would like to extend my thanks to my supervisor Luis Filipe Coelho. Your guidance and expertise were invaluable throughout the process of completing my master's thesis. Thank you for all your support during this period.

Lastly, I want to thank ISEP, for all the memories and for all the people that I am sure I will carry with me for the rest of my life.

Index

List Of Figures

List Of Tables

Acrónimos

Lista de Acrónimos

OS Operating system

VM Virtual Machine

GUI Graphical User Interface

SSL Secure Sockets Layer

WSL Windows Subsystem for Linux

1. Introduction

1.1 Context

The present thesis reports the project developed during the period from September 11 to July 14 as part of the Thesis/Dissertation curricular unit, taught in the second semester of the second year of the master's in biomedical engineering at the Instituto Superior de Engenharia do Porto. This project will be developed with the Phenotypic Evolution Group (Figure 1) at the Instituto de Investigação e Inovação em Saúde (i3s).

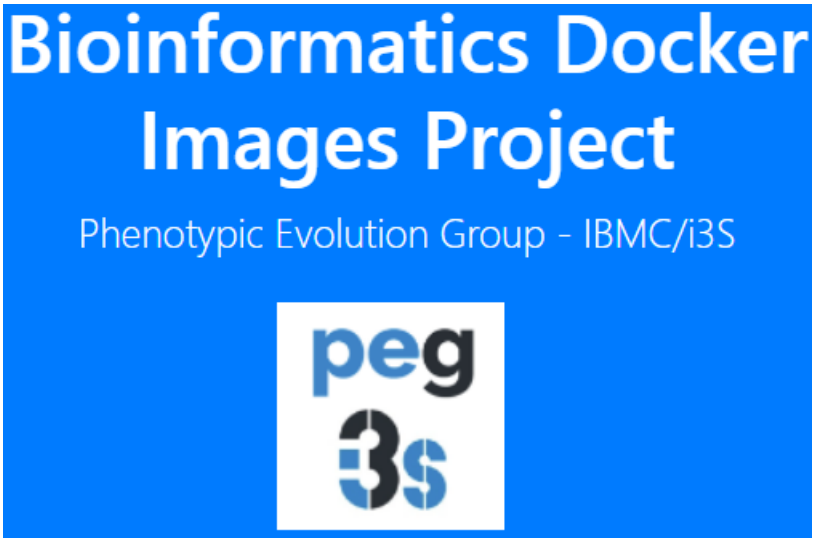


Figure 1 – Projeto de Imagens Docker do i3s (https://pegi3s.github.io/dockerfiles/).

The sequencing of data, whether representing complete genomes, exomes, or transcriptomes, is becoming increasingly important in biomedical research as well as in medical practice. However, there has been a decline in bioinformaticians since 1990, which poses a significant barrier to the use of such data (Petar Brlek, 2024). The ease of use of bioinformatics applications designed for researchers without an informatics background (the majority of referred investigators have formation in biomedical sciences) is, therefore, essential.

Thus, despite an increase in the number of publicly available software tools for studies in this area, there is no consistent increase in the number of people who know how to use them. These software tools generally require learning time, which is unattractive to researchers. Container technologies (Podman, containerd, runc, Buildah and Docker, with the latter being the most popular out of all of them) can help solve this problem due to their multiple advantages. A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another (Docker1, 2024). Firstly, Docker images can be used to provide ready-to-use software packages. Since all dependencies are already installed, and Docker images can be used regardless of the operating system, the user does not need to worry about software installation. This is important as many software applications have difficult-to-install dependencies, and some of them become unavailable after a few years. Docker images are also immutable, contributing to reproducibility. The use of Docker images facilitates the development of

complex pipelines, applications with graphical interfaces that require external software, and even databases. Due to these characteristics, Docker images can help solve the reproducibility crisis in bioinformatics analyses.

With the aim of developing open industry standards for container formats and runtimes, the Linux Foundation established the Open Container Initiative (OCI), an open governance framework. OCI was founded in June 2015 with the goal of creating guidelines that define the packaging and operation of containers in order to maintain the interoperability and flexibility of this technology. OCI contributes to innovation and stability in the container ecosystem by promoting cooperation amongst different stakeholders, such as cloud service providers, enterprise customers, and developers of container runtimes. This ensures that a variety of software tools and environments can coexist peacefully (Open Container Initiative, 2024).

1.2 I3S

I3S results from the merger of three institutes and researchers from various schools of UPorto, thus consolidating an extensive collaboration among all institutions over many years (i3s, 2024).

The long-term collaboration between the Institute of Molecular and Cellular Biology (IBMC), the National Institute of Biomedical Engineering (INEB), and the Institute of Molecular Pathology and Immunology of the University of Porto (IPATIMUP) - the three founding institutes of i3S - encompasses joint projects, co-supervision of PhD students, sharing of large-scale equipment, and employment of research personnel under coordinated policies. Six schools of the University of Porto (FMUP, ICBAS, FMDUP, FCUP, FEUP, and FFUP) and three hospitals (São João Hospital Center, Porto Hospital Center, and the Portuguese Oncology Institute) also contribute to i3S activities. This broad participation of schools, research institutions, and hospitals in a research institute is unique in Portugal and constitutes a valuable asset for scientific and technological development, while creating an environment that fosters groundbreaking research and the translation of discoveries into the clinic. The building where i3S is headquartered, still quite new as it was completed in 2015, is also a key element in creating this vibrant environment (i3s, 2024). The building for this institution can be seen in Figure 2.

Figure 2 – I3S Building, Street view (i3s, 2024).

This intricate network that is i3S focuses on three Integrated Programs: Cancer; Infection, Immunity, and Regeneration; and Neurobiology and Neurological Disorders.

The Phenotypic Evolution Group at i3S combines bioinformatics, biostatistics, 3D structure modeling, genomics, transcriptomics, evolutionary biology, and population genetics to produce predictive models that can help address the aforementioned questions. The group's knowledge in evolutionary biology and population genetics allows for a deeper understanding of the behavior of biological systems, as Theodosius Dobzhansky stated, "Nothing in Biology makes sense except in the light of evolution," and as noted by Michael Lynch, "Nothing in evolutionary biology makes sense except in the light of population genetics." Expertise in bioinformatics, biostatistics, and -omics enables the development of software pipelines that can be used to analyze the ever-growing biological datasets in a relatively short period, thus providing useful answers. Such diverse knowledge results from the integration of researchers from two different research groups at IBMC (Molecular Evolution and Evolutionary Systems Biology).

This group is responsible for the development of the pegi3s Bioinformatics Docker Images Project (Vieira, et al., 2024), a growing collection with over 100 Docker images of software typically used in the fields of genomics, transcriptomics, proteomics, phylogenetics, sequence manipulation, among others. This project is a service offered by the Portuguese community within the context of the European ELIXIR program (Elixir, 2024).

The basic philosophy of the Bioinformatics Docker Images Project is described by López-Fernández et al. (López-Fernández, Ferreira, Reboiro-Jato, Vieira, & Vieira, 2022).The project's rich documentation is the main difference compared to larger projects. For example, hyperlinks to software manuals or GitHub pages are provided for each Docker image. Additionally, clear instructions on how to run the images and, in many cases, test data that can be used to quickly evaluate the Docker image are provided, along with details on how each Docker image was built. Docker files and supporting scripts are freely available to the public. Instructions on how to install Docker in Linux and Windows are also provided.

1.3 Objectives

Currently, the tools are classified into broad categories to facilitate finding the most suitable tool(s) for a specific project. However, as the number of available Docker images increases, new ways of quickly finding relevant images become more important, a crucial aspect in the work being developed. Additionally, the virtual machine provided on the website is already a few years old and needs to be updated.

As such, the internship objectives will be as follows:

O1. Update the virtual machine provided in the i3s website ;
O2. Review all Docker images from pegi3s website to ensure the proper functioning of each one;
O3. Create a standardized set of test data to facilitate providing examples for the development of new Docker images;
O4. Collect data generated by each Docker image, which will be available to users to facilitate understanding of the software;
O5. Develop a graphical interface application that will allow convenient use of all Docker images.

1.4 Document Structure

The present document describes in detail all the experimental work carried out, divided into 6 parts.

Chapter 1 introduces the research work and outlines the defined objectives. Chapter 2 presents similar works to the developed interface, discussing their main benefits and limitations, and compares them at the end of the chapter. Chapter 3 provides a comprehensive overview of existing research and literature relevant to the topic, aiming to contextualize the current study within the broader academic discourse. Chapter 4 describes the materials and methods, offering an overview of the software utilized and how it was used to achieve the defined objectives. Chapter 5 presents the results obtained in this work, along with their discussion. Finally, chapter 6 offers a conclusion taking into account the work developed and the objectives delineated in the beginning of the project.

2. Related Works

This chapter offers a thorough analysis of existing software that offer similar features to those that the developed GUI intends to offer. The most relevant software that offer the similar capabilities of the GUI developed on this project are: DockStation, Docker Desktop and Portainer.io.

2.1 DockStation

DockStation is a developer-focused application designed for managing Docker-based projects. It uses a user-friendly GUI allowing the user to monitor, configure, and manage services and containers efficiently (Kazlouski & Smith, 2024). The developer mentions DockStation came to fruition to avoid “excessively long CLI one-liners” (DockStation, 2017). Which is a similar motivation to the development of the GUI from this thesis. It also mentions that the GUI provided by the Docker group at the time, Kitematic (now known as Docker Desktop), felt lacking and basic (DockStation, 2017).

The product relies on 4 basic principles which are the following:

- Straightforward kickstart, avoiding the need to jump into extensive documentation;
- Native application, ensuring it launches effortlessly with minimal to no initial setup required;
- Backward compatibility. All projects created within DockStation can also be run in the CLI. Additionally, other Docker Compose projects can be imported into the app;
- Extensive GUI functionality, allowing the user to navigate and manage everything through a user-friendly interface.

DockStation offers features such as on the fly creation. In Figure 3, is the menu for creating a new project.

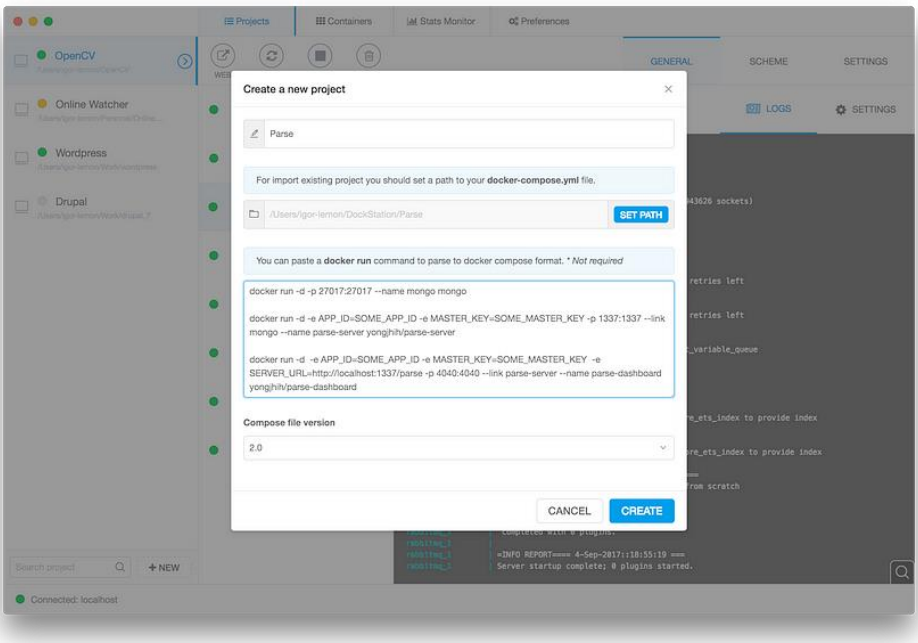


Figure 3 - Menu for image creation in DockStation.

The parse Docker commands is a feature that allows the user to convert the run command directly to a Docker Compose project within the DockStation (DockStation, 2017).

Users are also able to monitor their projects by checking the logs with full text search. It has the basic tools to manage containers such as rapid launch, stop and restart, multiple containers clean up, info about the containers and ability to switch the image version and bind port. The tools can be seen in Figure 4 (DockStation, 2017).

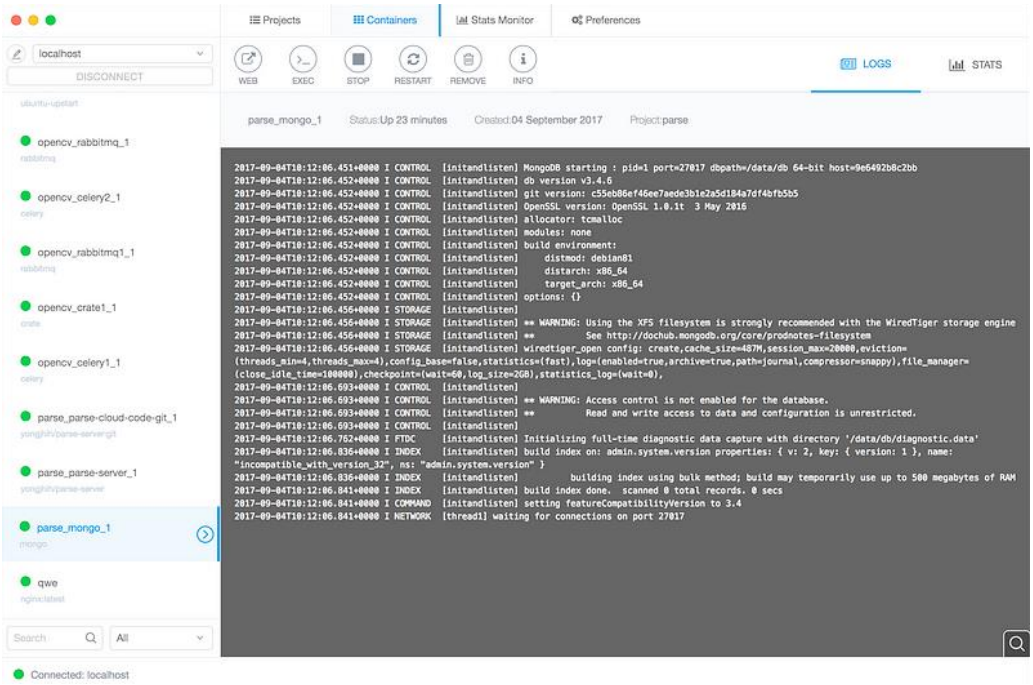


Figure 4 - Layout of an image running and its logs and interaction tools in DockStation.

Other features that are offered by DockStation are Docker Hub support, full featured Docker Machine support, real time resource monitoring and switch the workstation from local Docker to remote Docker. It also offers customization of the GUI (DockStation, 2017).

In conclusion, DockStation is a versatile tool for creating, running and managing Docker images.

2.2 Docker Desktop

Docker Desktop is as an easy to install application for Mac, Linux or Windows environments that allows a user to build, share and run containerized applications and microservices (Docker2, 2024).

It provides a straightforward customizable GUI that allows a user to manage its containers, applications, and images directly from its machine. It works in a way that is similar to Portainer, being one of the upgrades that were later found (Docker2, 2024).

Docker Desktop reduces the time spent on complex setups facilitating the process for the user. It takes care of port mappings, file system concerns, and other default settings, and is regularly updated with bug fixes and security updates.

Docker Desktop is also simple to maintain due to monthly releases by its developers including new features, team lead and business deliver secure and innovative applications. Moreover, it's also regularly maintained with bug fixes and security updates.

Docker Desktop is also very safe, consistently monitoring and managing patches and security fixes as needed.

The main downside to Docker Desktop is that when creating a Virtual Machine, the KVM virtualization should be turned on in a computer that is running the VM, not being related to the VM itself. This is a software application that is used to create Virtual Machines. This option activation was overly complex in a project whose main objective was to facilitate the life of biologist without background experience in the area of bioinformatics. Another huge downside was the inability to run GUI in a windows environment, which was the environment on which it was being run, since it is not possible to activate the "xhost +" command in the console.

Docker Desktop allows the user to create a container from a pulled image. The user just has to go to their image list, select that image and press the run button (these menus will be further explained later). By doing this, it will open the menu in Figure 5.

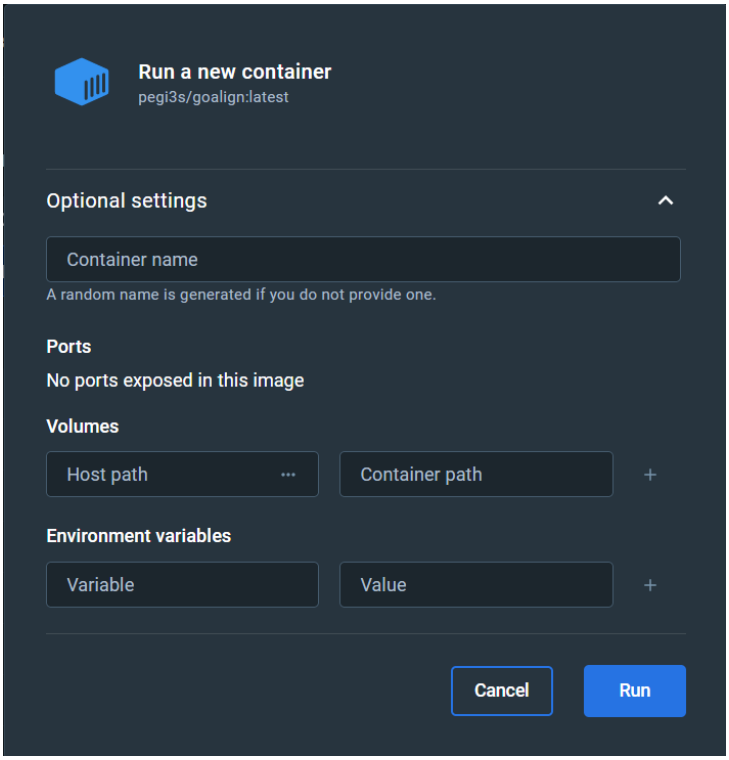


Figure 5 - Menu to run Docker images in Docker Desktop (running a Docker image, it will create a container out of that image).

The user can set the container name and attribute a port if it is exposed (useful in web application development). It can also assign volumes to the container which are ways to store data outside of the container and environment variables which help in the configuration of the container.

In conclusion Docker Desktop is a useful tool for building, sharing and running containerized applications. It offers users with a customizable UI and added security. It can run in multiple os and allows the user to manage his Docker images. The Docker software will be later discussed more in depth with the features that were relevant in this project.

2.3 Portainer.io

Portainer for Docker is a management UI which allows a user to easily utilize different Docker environments (such as Docker hosts or Swarm clusters). Portainer was created with the intent of being as simple to deploy as to use (Docksal, 2024).

Portainer consists of a single container, whether deployed as a Linux container or a Windows native container and supports other platforms as well. Portainer allows management of all Docker resources (containers, images, volumes, networks, etc.) It is compatible with both the standalone Docker engine and Docker Swarm mode (Portainer1, 2024).

Portainer had added security features. During the installation, Portainer will by default create a self-signed SSL (secure sockets layer) certificate that encrypts the traffic between the Portainer Server and the end user and the traffic between the Portainer Server and the Portainer Agent. It also allows the user to replace the self-assigned certificate with theirs (Portainer2, 2024).

Portainer also allows the user to switch the Portainer logo with their own, being the only customization option available (Portainer2, 2024).

To run a Docker image in Portainer, a user has to navigate to the container section of the menu. By doing this, he will be met with menu in Figure 6.

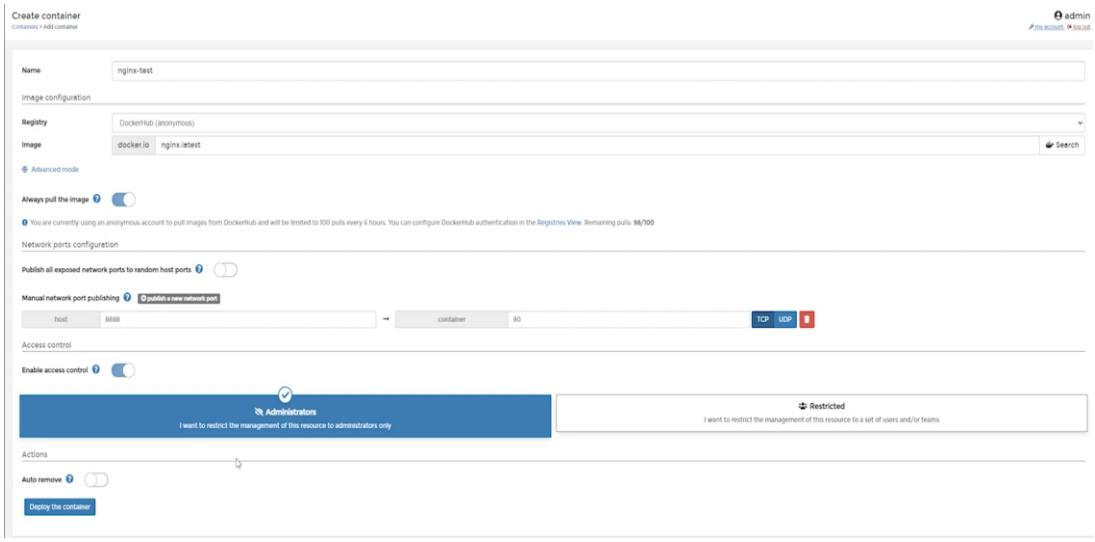


Figure 6 - Menu for running a Docker image in Portainer.io.

In the menu above, the user needs to select the name of the container that the image will be run on. Afterwards, it is necessary to fill the registry (where the image is) and the specific image name. It also has a “Always pull the image” option which should always be on so as to pull the Docker image when running the container. The advanced features menu is displayed in the Figure 7 below (Portainer3, 2024).

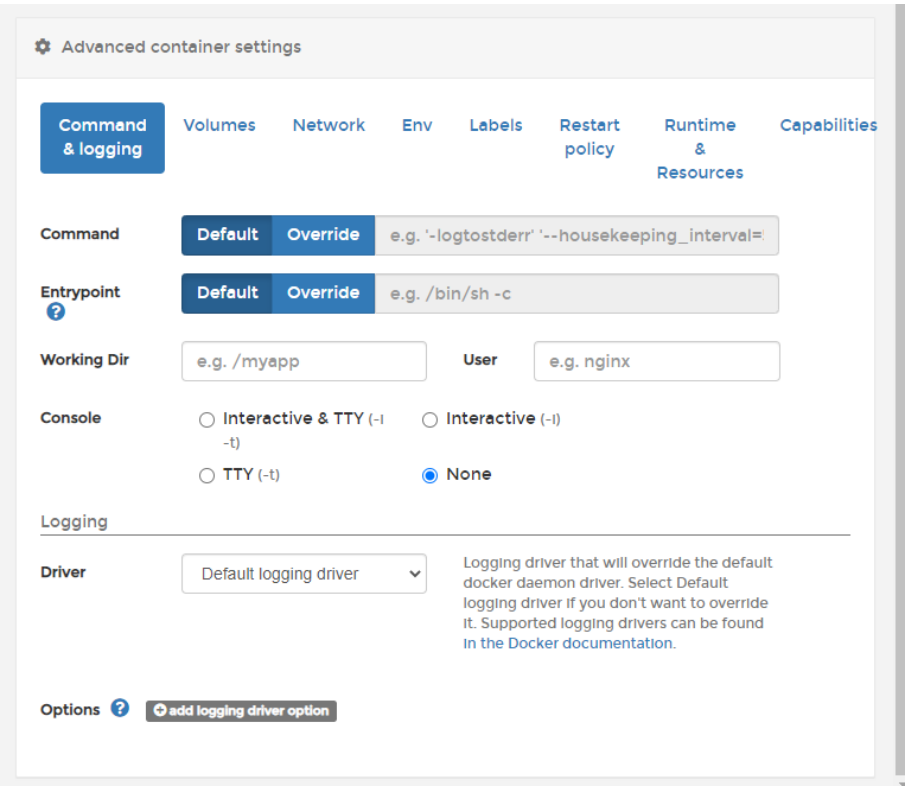


Figure 7 - Advanced features for running a container.

The “Command” section allows the user to set the command that is run when the container starts. The “Default” option allows him to use the default command provided by the container's image and the “Override” option provides a command to override the default value. Similar to the command option, the “Entry Point” allows the user to use either the default entry point defined by the image, or use override it with a different entry point. The “Working Dir” option allows the user to choose the working directory for the container to run (Portainer4, 2024).

Other options are the console (to set the console configuration), the user (specifies the user that the container's command should run as) and the driver (allows the user to select the logging driver to use for the container, and the available options are the logging drivers configured on the Docker host). In the option section the user can set additional options for the logging driver. It is also possible to add a new option by clicking "Add Logging Driver Option" and configuring accordingly (Portainer4, 2024).

In conclusion Portainer.io is a powerful tool to not only manage Docker images but also to utilize multiple Docker environments. It has added security features and also allows the user to run Docker images. However, it is only possible to run it as a Docker image, being its main

limitation. The Portainer.io tool will be later discussed more in depth with the most relevant features to this project.

2.4 Final Remarks

To compare the differences between the GUI for using containers, various parameters will be utilized like:

- Independence, the ability to run Docker images without Docker installed;
- Run images, allows the user to run
- Image management, allows the user to manage the Docker images in the machine;
- Customization of the GUI, allows the user to make changes to the appearance of the GUI;
- Runs in multiples os;
- Added security, which are other security methods beside Docker.

The parameters were chosen in account to them being the most referenced by the companies in the respective websites. Plus, they other parameters where chosen in order to better compare the other programs with the GUI to be developed. All the relevant software is compared in Table 1 using the parameters explained above.

Table 1 - Comparison between the relevant software.

	DockStation	Docker Desktop	Portainer.io
Independence	Yes	Yes	No
Runs Images	Yes	Yes	Yes
Image management	Yes	Yes	Yes
Customization of the GUI	Yes	Yes	Little
Runs in multiple os	Yes	Yes	Yes
Added security	Yes	Yes	Yes
Made for non bioinformatics	No	No	No

DockStation, Docker Desktop, and Portainer.io are all tools designed to facilitate Docker image management with features similar to the GUI to be developed. The strengths and weaknesses for the relevant parameters can be viewed on the table above.

DockStation and Docker Desktop both offer independence, full image management capabilities, customization of the GUI, and added security. They run on multiple operating systems, making them versatile options for various environments.

Portainer.io, while also proficient in running images and providing image management, falls short in customization of the GUI, offering limited options in this regard. It does not operate

independently (needs to be run as a Docker image. Nevertheless, it still supports multiple operating systems and includes added security features.

In summary, all software analyzed provide a comprehensive set of features with high levels of independence and customization, making them suitable for more technical users who need flexibility.

It is important to note that most of the existing Docker software are focused on managing the Docker software and few of them have features like running Docker images. Plus, all of the images compared are developed for people with a background in bioinformatics, not being tailored for biologists to use ("the acceptance and use of these applications by biologists lags behind this proliferation", (Shachak, Shuval, & Fine, 2007)).

To make it easier for people with less qualifications to use, the GUI would have to be easy to achieve basic tasks like running or creating containers. It should also be easy to learn how to use the software and good documentation on how to use it.

3. Repeatability, Replicability and Reproducibility in Bioinformatics

End-to-end automated process

Literate Programing

Code version control and persistent sharing

Computer environment control

Persistent data sharing

Documentation

Continue Validation

Containers in assuring reproducibility

This chapter provides a comprehensive overview of existing research and literature pertinent to the topic under investigation. This chapter aims to contextualize the current study within the broader academic discourse. Here, the importance of bioinformatics will be discussed, highlighting how containers, specifically Docker, support this crucial practice.

On the field of bioinformatic research, where technological advancements and data generation are rapidly evolving, the reliability of obtained results is essential. For this, three main concepts ensure this reliability (Apostal, Apostal, & Marsh, 2018):

- Repeatability refers to the ability to achieve consistent precision measurements by the same team, using the same measurement system, under identical operating conditions, and at the same location, across multiple trials. In the field of bioinformatics, this means that researchers can reliably replicate their own computations;
- Replicability refers to the ability of a different team to achieve the same measurement with the specified precision, under the same operating conditions, whether in the same or a different location, across multiple trials. In bioinformatics, this means that an independent group can obtain the same results using the original researcher's artifacts;
- Reproducibility refers to the ability of a different team to achieve the same measurement with the specified precision using a different measurement system, in a different location, and across multiple trials. In bioinformatics, this means that an independent group can obtain the same results as the original research by using artifacts they have developed independently.

It should be noted, according to the definition presented above, that a studies validity might be compromised by analytical errors despite being repeatable and replicable. Therefore, reproducibility is a must in order to create an ever-growing scientific framework. It should improve computational research's overall reliability in terms of robustness. Robustness refers to the ability to apply the findings to different contexts (Ziemann, Poulain, & Bora, 2023).

In the year of 2009, a systematic evaluation was conducted on the reproducibility of bioinformatic articles produced on the area of microarray gene expression. Ou of the 18 articles analyzed, only 2 of those were able to be reproduced (11%), bringing into question the reliability of these studies (Ioannidis, Allison, Ball, & Coulibaly, 2009). This tendency tends to repeat itself in bioinformatic studies due to the lack of documentation, software and citing missing data. In 2018/2019 an attempt to reproduce five bioinformatic studies by the National Institutes of Health failed to produce any results (Maryam Zaringhalam, 2020). More studies in this area turned out similar results. Only 245 of Jupiter notebooks out of the 4169 in biomedical articles reproduced similar results when compared to the original study (Samuel S, 2022).

In order to fight the lack of reproducibility in Bioinformatics, a number of guideline practices have been developed. The good practices can be divided into 5 pillars of reproducible computational research (Ziemann, Poulain, & Bora, 2023). These five pillars (below) establish clear parallels with the open science framework (open data, code and papers). However, on this

concrete situation, practical reproducibility is prioritized, with a particular emphasis on programming techniques, open reporting, and the function of computational ecosystems (Ziemann, Poulain, & Bora, 2023). This organization can be viewed in Figure 8.

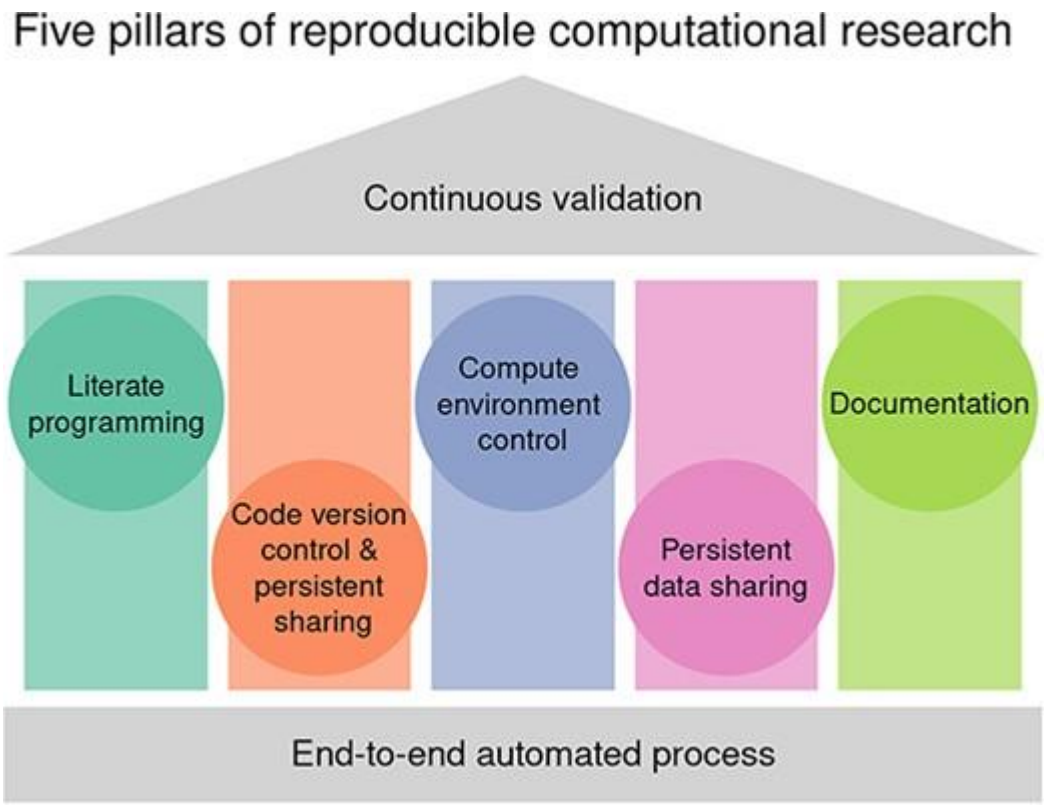


Figure 8 - Five pillars of reproducible computational research (Ziemann, Poulain, & Bora, 2023).

3.1 End-to-end automated process

To ensure reproducibility in bioinformatics, workflows should be formalized in code, automating processes from data inspection to the generation of study conclusions. Automated processes eliminate the need for manual steps, which are time-consuming and error prone. Without end-to-end automation, achieving other best practices is challenging (Ziemann, Poulain, & Bora, 2023). Although not error-free, scripted workflows allow better auditing and easier reproduction compared to graphical tools like spreadsheets, which are susceptible to data entry and manipulation errors. For instance, spreadsheets have caused problems such as gene names being inadvertently converted to dates (Panko, 1996).

To facilitate end-to-end automation, it is crucial that code and data are "linked," allowing the code to automatically fetch and process data from publicly accessible locations. Although manual data cleaning is sometimes necessary, retaining both raw and cleaned data along with a cleaning protocol is recommended to ensure a clear starting point for computational workflows (Ziemann, Poulain, & Bora, 2023).

3.2 Literate programming

Literate programming was first introduced by Donald Knuth in 1984, and it is a paradigm where a computer program is explained in a natural language, such as English, interwoven with snippets of macros and traditional source code. This approach allows for the generation of compilable source code from the combined explanations and code (Knuth, 1992).

This methodology is already being implemented in multiple projects, such as Project Jupyter. It provides a 'notebook'-type interface, incorporating R, Python, and other computer language chunks into documents. A notable effort to enhance reproducibility through literate programming in Jupyter notebooks was demonstrated in the analysis of RNA-seq samples from Zika virus-infected patients. In this study, the authors supplemented their traditional research paper with a Jupyter notebook that documented and executed the entire analysis process (Wang & Ma'ayan, 2016).

The utilization of literate programing offers substantial benefits over alternative approaches, such as (Ziemann, Poulain, & Bora, 2023):

- Provenance of results is demonstrated, with code and results included in the document, avoiding errors from copy-pasting;
- It saves time by requiring minimal changes for routine reports, unlike manual assembly in a word processor;
- It allows extensive documentation, enabling the inclusion of text, links, tables, images, and other objects, facilitating the authoring of entire journal articles;
- Outputs are organized sequentially within the document, making it easier to understand the logical steps of the analysis;
- It is amenable to version control, a best practice in software development;
- Output reports are error-free, as the document is rendered only when the entire script compiles successfully;
- Flexible output formats include PDF, DOC/DOCX, HTML, slideshows, and e-books, supporting rich and interactive content.

Literate programming proves invaluable in scientific communication across various scenarios, whether for sharing data analysis reports, presenting at meetings, writing research articles, or self-publishing e-books. It enables the creation of transparent data narratives with clear origins in a format that is easily shareable (Ziemann, Poulain, & Bora, 2023).

3.3 Code version control and persistent sharing

In bioinformatics, sharing code is becoming a standard practice to ensure reproducibility and transparency, and it is a requirement for many specialized journals. Code sharing has been shown to improve article citation rates. One of the most popular methods for sharing research code is through online software repositories that integrate version control. Version control

systems, also referred to as 'source control,' track changes made to sets of files, typically encompassing source code, scripts, and documentation (Ziemann, Poulain, & Bora, 2023).

3.4 Compute environment control

Regular software updates, while beneficial for fixing bugs and adding features, pose a challenge for future reproducibility. In bioinformatics, these changes can affect results, making it essential to report the exact versions of all programs and packages used in an analysis and to archive them for future reference. Reproducing an older study, in any software currently out to date, may require rolling back to previous versions and potentially the operating system, as system dependencies for low-level routines must also be met (Ziemann, Poulain, & Bora, 2023).

To accomplish this, a virtual machine can be utilized, by installing the desired version of the software without having to change the host computer version. However, and despite this method providing good reproducibility, setting up an environment with dated features, the software and additional packages would take a few hours, which isn't desirable (Ziemann, Poulain, & Bora, 2023). Therefore, containers are used to attempt to revolve the downsides of utilizing VMs, being more lightweights as they are share parts of the host operating system. Therefore, and opposed to VMs, running a container incurs only a small reduction in performance as compared to the host system (Docker2, 2024).

3.5 Persistent data sharing

Computational research wouldn't be reproducible nor auditable without data sharing. Data sharing is a crucial aspect of "data science", which emphasizes collaboration, transparency, accessibility and inclusivity. Sharing data not only facilitates replication and auditing but also allows it to be reused in other contexts, increasing efficiency by preventing duplication of effort and opening up previously unfeasible new research ideas (Ziemann, Poulain, & Bora, 2023).

Examining unprocessed data enhances research rigor by revealing unintentional mistakes and problems with research integrity. As such, a study piece lacking supporting data and code is similar to a press release or advertisement in that the veracity of the claims is unknown (Ziemann, Poulain, & Bora, 2023).

3.6 Documentation

Documentation is the essential element that holds a data science project together. The published article serves as the central artifact, detailing the research project and linking to all supporting materials. From a reproducibility perspective, the methods section is paramount. It must be sufficiently detailed to enable other researchers to understand, replicate the experiments or analysis, and achieve similar results (Ziemann, Poulain, & Bora, 2023).

Unfortunately, in bioinformatics, crucial details in data processing procedures are often omitted, hindering reproducibility. Commonly missing information includes software and package versions, parameter settings, and configuration files. To address this issue, the 'Materials Design Analysis Reporting' (MDAR) checklist has been developed to guide authors in providing comprehensive methodological reporting in the life sciences (Chamber, Collings, & Graf, 2019).

Clear links to software code, datasets, and other resources, such as computing environments, should be included in the article (Ziemann, Poulain, & Bora, 2023).

Reproducibility depends heavily on the presence of a thorough README file in the code repository. It should explain the code's, software's, or project's general goal and how it connects to the article (Ziemann, Poulain, & Bora, 2023).

3.7 Continuous Validation

When the 5 pillars stated before are not integrated correctly and checked, problems tend to emerge. This being the reason why regular code testing after making updates to the codes or data is considered best practice. Testing for bioinformatics software may include an integration test to see if the functions are interacting as intended and a series of unit tests for each function (Ziemann, Poulain, & Bora, 2023).

3.8 Containers in assuring the reproducibility

Nowadays, software containers are being used more and more in many different computer disciplines. Applications running in containers have a lightweight, uniform operating environment that is simple to share with peers, facilitating reproducible outcomes (Apostal, Apostal, & Marsh, 2018).

Containerization helps provide instructions for packaging the building blocks of computer-based research (i.e., code, data, documentation, and the computing environment). In particular, plain text files that serve as a machine- and human-readable recipe for setting up a computing environment and interacting with data are the foundation upon which containers are constructed. Scientific article writers can significantly increase the documentation, transparency, and reusability of their work.

Although numerous tutorials are available on using containers for reproducible research, there is a lack of detailed manuals specifically on writing the instructions to create containers for computational research, aside from general best practice guides. The article “Ten simple rules for writing Dockerfiles for reproducible data science” (Daniel Nüst, 2020) summarizes well the best practices in order to obtain containers that are able to assure the reproducibility of scientific research. The ten rules are shown in Table 2 (Daniel Nüst, 2020).

Table 2 - Ten rules for writing Dockerfiles for reproducible data science.

Rule	Description
Use available tools	Avoid writing a Dockerfile from scratch by looking for tools that help generate a Dockerfile.
Build upon existing images	Build upon existing, well-maintained Docker images using version-specific tags for security, transparency, and reproducibility.
Format for clarity	Use indentation, new lines and comments to make the Dockerfile well documentable and readable prioritizing clear, well-documented instructions over minimizing image size.
Document within the Dockerfile	Document the Dockerfile with clear comments, metadata labels, and usage instructions for transparency and reproducibility.
Specify software versions	Specify software versions explicitly to ensure reproducible builds and stable workflows.
Use version control	Use version control for Dockerfiles and all related files to ensure consistent builds, enable collaboration, and enhance the sustainability and impact of your work.
Mount datasets at run time	Mount datasets at runtime using bind mounts for better manageability, accessibility, and separation of data from the container's software environment.
Make the image one click runnable	Make the Docker image runnable with a single command by setting sensible defaults and providing clear instructions for both interactive and headless use cases.
Order the instructions	Order Dockerfile instructions from least to most likely to change to optimize build caching and efficiency.
Regularly use and rebuild	Regularly use, rebuild, and document containers to ensure reproducibility, security, and up-to-date workflows.

The rules above are utilized in the creation of all images inside the pegi3s repository. These were also the rules followed when creating the Docker images for the Docker Manager GUI and the Docker App GUI that allows the use of all Docker images.

4. Materials

Linux

Virtual Box

Docker

Docker Desktop

Portainer

Python

In this section, the software used to achieve de desired results are going to be outlined.

4.1 Linux

Linux is an operating system that is steadily growing in popularity, gaining traction not only among developers but also among the average consumers. It offers various distributions such as Fedora, Debian, Ubuntu and CentOS, allowing a user to select which one suits him best (Noviantika, 2024). These distributions are compared in Table 3.

Table 3 - Comparison between Linux distributions (Bryan, 2016), the higher the number the better.

	RHEL/CentOS	Debian	Fedora	Ubuntu	FreeBSD
Desktop	3	4	4	5	3
Server	5	5	3	5	5
Package Management	4	5	4	5	4
Number of Pacakges	2	5	3	5	3
Quality of Packages	3	3	3	4	3
3rd Party Repo Support	2	4	3	5	1
FSF philosophy	2	4	3	1	2
Not Too Evil	3	5	4	3	5
Out of Box HW Support	4	3	4	5	2
Automatic Security Updates	3	5	3	5	3
Init System	0	0	0	0	5
Stable and Dependable	3	4	2	4	5
Cutting Edge, Up to Date	2	4	5	5	3
Release Upgrades	1	4	1	4	5
Long Term Support	5	3	1	4	2
Predictible Release Cycle	3	2	2	5	3

As a disclaimer, it is important to notice that the aforementioned table is only a user's own personal experience and is only mentioned to showcase the differences between the different Linux distributions. Nevertheless, the advantages of Ubuntu are likely recognized by many users since Ubuntu ranks first in many evaluations.

The various Linux distributions can also be shortened as "distros". These are operating systems that are based on the Linux kernel (the central component of and operating system that manages operations of computer and hardware (GeeksForGeeks, 2023)). On this project, it was opted to use the Ubuntu operating system.

Ubuntu was first introduced in 2004 by the British company Canonical. It is based on Debian, a popular distro back then. Debian was difficult to install, so Ubuntu was proposed as a more user-friendly alternative.

Today, Canonical, as the manager of Ubuntu, is tasked with releasing a new Ubuntu version every six months. It also provides hosting servers for Ubuntu Community, which allows people worldwide to contribute to testing bugs and give technical support for free (Noviantika, 2024).

Ubuntu is extremely popular, being the world's most widely used Linux workstation platform according to Ubuntu's official website. The reasons it is so popular are (Noviantika, 2024):

- User-friendliness – Ubuntu has a simple and intuitive interface ;
- Strong Security – Along with AppArmor, Ubuntu employs advanced security measures to prevent breaches;
- Software options – Ubuntu offers a vast selection of applications ready to be installed, many of which are exclusive to this operating system;
- Enhanced privacy – Ubuntu guarantees a strict data privacy and provides user with the ability to customize their privacy settings;
- Lightweight performance – The operating system works on low-end devices, since its interface uses less than 1GB of RAM;
- Free of charge – Ubuntu is a free open-source Linux distribution.

Moreover, the choice of working with Ubuntu was also due to the fact that it's the number one platform for containers. From Docker to Kubernetes to LXD, Ubuntu is able to run containers at scale. It is fast, secure and simple, and as of now, Ubuntu powers millions of computers worldwide (Docker3, 2024).

Not only as a mean to run Docker images, but Ubuntu is also used as a base image for building a new container. When selecting an operating system as the right base, we should focus on certain aspects as portrayed in Table 4 (JFrog, 2022).

Table 4 – Aspects to consider when selecting a base operating system.

Aspect	Description
Dependencies	If the dependencies required by an application are already included in the base image, there is less wok when configuring the Docker image by adding custom commands to the Dockerfile.
Extensibility	Having more dependencies and other resources built into the base image simplifies updating overtime, without needing to modify the Dockerfiles.
Security	The fewer components included in the base image, the lower the risk of containing a vulnerability on a library or other resource an attacker could exploit.
Image "bloat"	Using an unnecessarily large image can lead to image bloating, which is a term that describes container images that are larger and take longer to download because they contain components the applications doesn't need.

To select an appropriate base image, it is necessary to find a correct balance between all the aspects portrayed above. Ubuntu was the one used on this project.

Ubuntu base images are easily found on Docker Hub, and they are created using the root file system of Ubuntu Base. This is a variant of the Ubuntu Linux operating system. It is a lighter-weight version of Ubuntu that one would normally install as the desktop version on a computer.

This version doesn't contain the software necessary to run a graphical interface, office software or other utilities that are irrelevant for most container user cases (JFrog, 2022).

The Windows Store offers an Ubuntu application that functions as a Linux terminal within Windows in a Windows Subsystem for Linux (WSL). This allows users to take advantage of the benefits of Linux in a Windows environment.

4.2 VirtualBox

VirtualBox is a general-purpose full virtualizer for x86 hardware, targeted at server, desktop, and embedded use (Oracle, 2023). In other words, it can create virtual machines.

A Virtual Machine (VM) is essentially a digital version of a physical computer, such as a laptop, smartphone, or server. It includes a CPU, memory, storage for your files, and even the ability to connect to the internet if necessary. Unlike the physical components that make up a traditional computer (hardware), VMs are considered virtual or software-defined computers that reside within physical servers, existing solely as code (Microsoft, 2024).

Virtualization is the process of creating a software based or "virtual" version of a computer, with dedicated amounts of CPU, memory, and storage that are "borrowed" from a physical host computer – such as a personal computer- or a remote server, often referred to as an image – such as a server in a cloud's provider's datacenter. A virtual machine (VM) is essentially a computer file, often referred to as an image, that operates like a real computer. It can run in a window as a separate computing environment, allowing users to run a different operating system or even use it as their primary computing experience, which is common on many work computers. The VM is isolated from the rest of the system, ensuring that the software within the VM doesn't interfere with the host computer's main operating system (Microsoft, 2024).

When a user opens the VirtualBox application, he is presented with the menu in Figure 9.

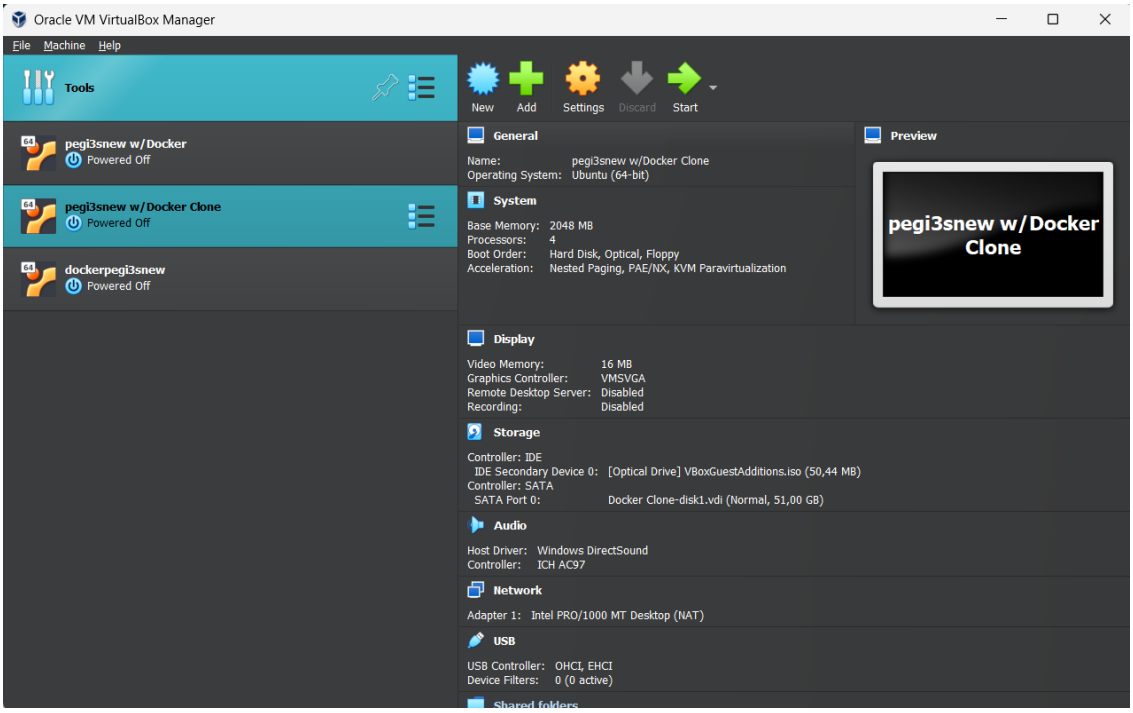


Figure 9 - VirtualBox home page.

As shown in the image above, a user can select an already existing virtual machine, and run it by double clicking it or by pressing the start green arrow button, on the right), create a virtual (on the blue button), add an already existing virtual machine or edit a virtual machine (with settings).

When a user wishes to create a virtual machine, he is met with the menu in Figure 10.

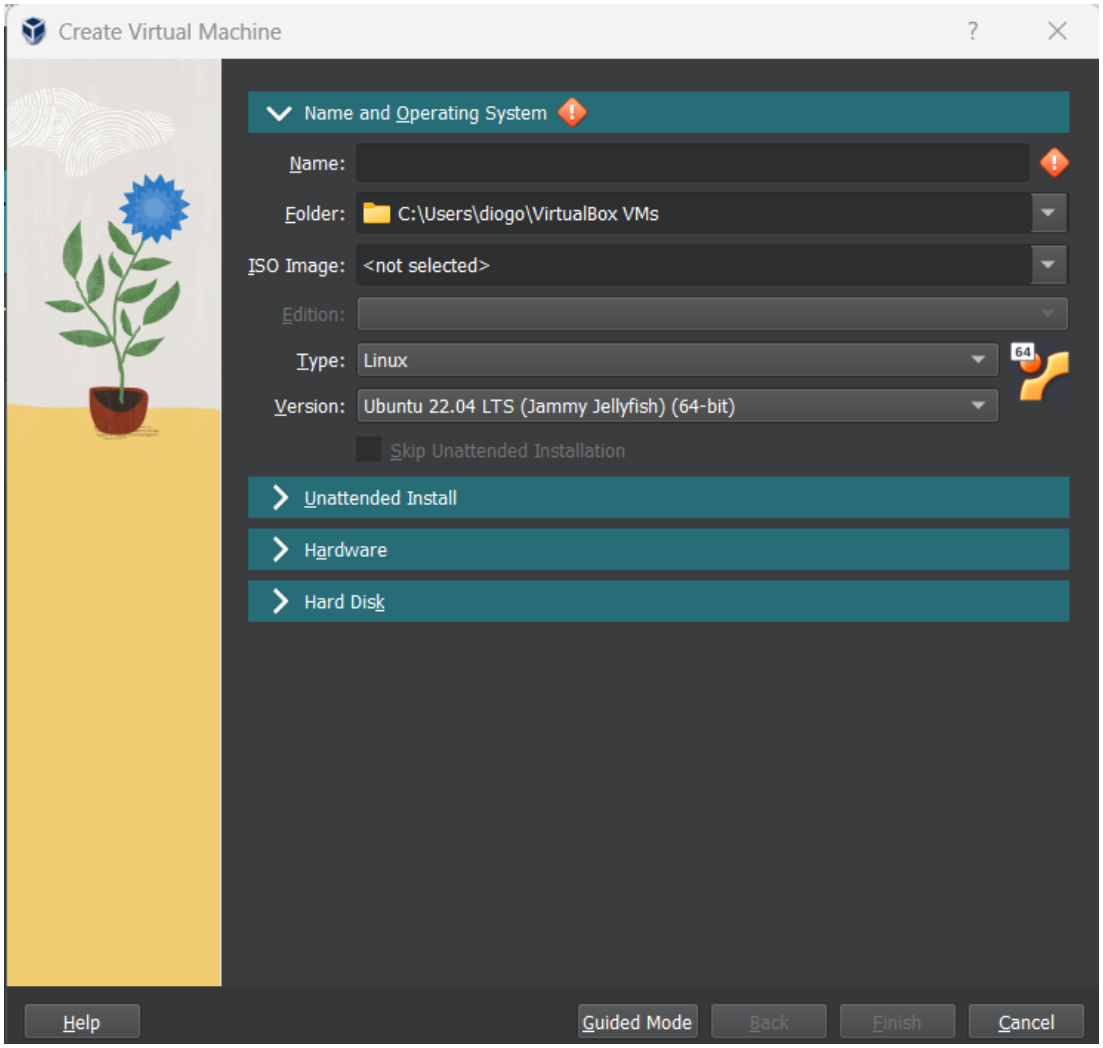


Figure 10 - Menu to add a virtual machine.

On the aforementioned menu, a user needs to select the VM name, the folders it's going to be stored on, the type of VM (if it's Windows, Linux, or Mac – Linux in this case), the version (distro, as it was previously mentioned, Ubuntu was used on this project) and the ISO Image. An ISO Image is an exact copy of an entire optical disk, and it's a smaller sized duplicate of large sets of data. In order to perform its function first needs to be opened and assembled so the data can be viewed. Here, it is used as a reference optical disk from where the VM is going to be created (Naor, 2021).

On the Figure 11's menu, a user can also choose the VM hardware and it's hard disk specifications as shown from the images below.

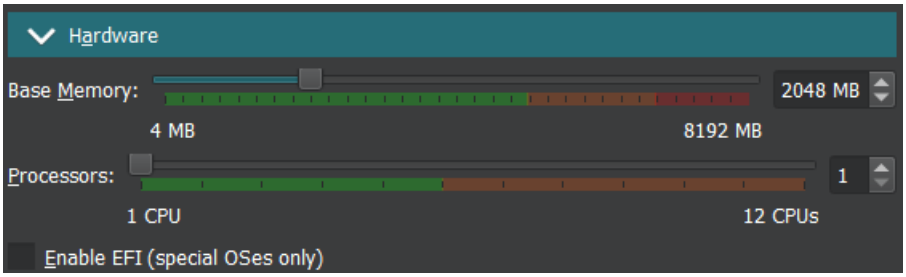


Figure 11 - Hardware specification's menu.

When creating a VM, a user can choose both the base memory (RAM memory) and the number of processors it can utilize from the host computer when running the VM. It should be noticed that these setting are limited to the hosts computer capacities.

When selecting hardware specifications, a balance should be found where the VM is neither so slow that it cannot perform the tasks expected of it, nor so resource-intensive that the host computer freezes and is unable to run it.

Extensible Firmware Interface (EFI) initializes hardware components and operating system image files when a computer starts. EFI supports more modern features and customization options than BIOS, enabling faster boot times (Open Shift, 2024). This feature wasn't used in this project.

On the Hard disk specification menu (Figure 12), a user could either choose to create its own virtual disk, use an existing one or not adding a hard disk. On this project, it was used an already existing virtual hard disk.

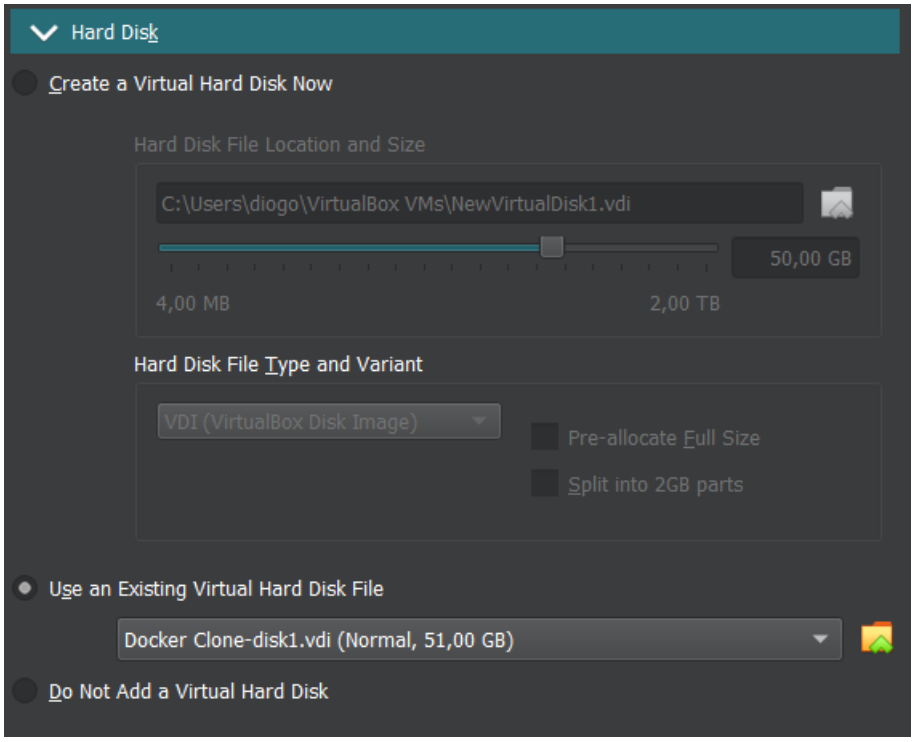


Figure 12 - Hard disk specification's menu.

When a user runs a Virtual Box for the first time, it's the same as booting up a computer that utilizes Linux for the first time. On this project, it was used to update the already existing VM to a newer version of Ubuntu, and it was initially utilized to run Docker images before new solutions were found.

4.3 Docker

Docker is a software platform that allows a user to build, test, and deploy applications quickly and works as the foundation of the Bioinformatic Docker Images Project. Docker packages software into standardized units called containers that have everything the software needs to run including libraries, system tools, code, and runtime. Using Docker, a user can quickly deploy and scale applications into any environment and have the certainty that the code will run (Amazon, 2024).

Docker container technology was first introduced in 2013 as an open-source Docker Engine. It capitalized on established computing concepts surrounding containers and specifically in the Linux world, primitives known as cgroups and namespaces. What sets Docker's technology apart is its emphasis on addressing the needs of developers and system operators to separate application tendencies from infrastructure (Docker4, 2024).

The success of Docker on the Linux world drove a partnership with Microsoft that allowed Docker containers and its functionality to Windows Server.

During run time, container images become containers, the same happens with Docker containers, images become containers when they run on Docker Engine. Compatible with both Linux and Windows-based applications, container software will always run the same, regardless of the infrastructure. Containers have the capability of isolating software from its environment, ensuring that it works uniformly despite differences for instance between development and staging. An important thing to note is; despite having similar resource isolation and beneficial allocation, containers and virtual machines are different in a way that containers virtualize the operating system instead of hardware, making them more portable and efficient. (Docker4, 2024)

Docker containers that run on Docker Engine are (Docker4, 2024):

- Standard → Docker has established the norm for containers within the industry, allowing them to be portable anywhere;
- Lightweight → Containers share the machine's operating system kernel and, for that reason, do not require an OS application, making the server more efficient with the added benefit of reducing server and licensing costs;
- Secure → Docker provides the strongest default isolation capabilities in the industry and applications are safer in containers.

4.4 Docker Desktop

As it was previously explained, Docker Desktop is a GUI application build for Mac, Linux or Windows with the intent of building, sharing and running Docker images. The usage of this software came as a later discovery, allowing for Docker images to be run in a Windows environment facilitating the deployment of almost all Docker images.

Docker Desktop was mainly utilized to manage the Docker images by deleting them, acting as a replacement for Portainer and serving as a model for the creating of the Docker Manager.

When a user opens Docker Desktop, it starts the Docker Engine. Docker Engine is an open-source containerization technology for building and containerizing applications. It serves as a client with a long-running daemon process, an API which specify interfaces that programs can use to talk and instruct the Docker daemon, and a command line interface (CLI) client. Scripting or direct CLI commands are used by the CLI to control or communicate with the Docker daemon via Docker APIs. The underlying API and CLI are used by numerous other Docker apps. Images, containers, networks, and volumes are just a few of the Docker objects that the daemon produces and maintains.

Despite allowing users to run Docker images in the Windows terminal, the Docker Desktop is not compatible with the previously mentioned Ubuntu application. Meaning both cannot run at the same time. Since the Ubuntu application allowed the user to run GUI, this was the preferred option. Also, the Docker images that are run with the help of the Ubuntu application

(in the WSL) cannot be accessed by Docker Desktop meaning that another application would need to be used in order to manage those.

The appearance of the Docker Desktop application is shown in Figure 13.

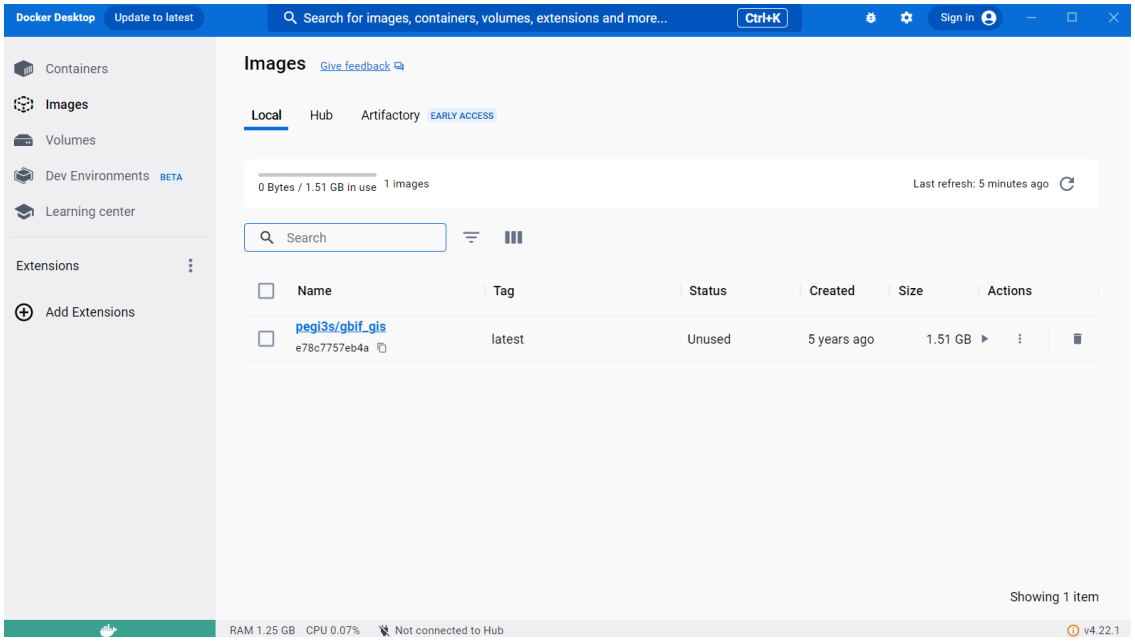


Figure 13 - Docker Desktop Images Menu.

It is an intuitive application, and overall, a more complete application than Portainer, allowing the user to much further interact with the files present in its computer regarding the Docker software. By pressing an image, the user is met with the menu in Figure 14.

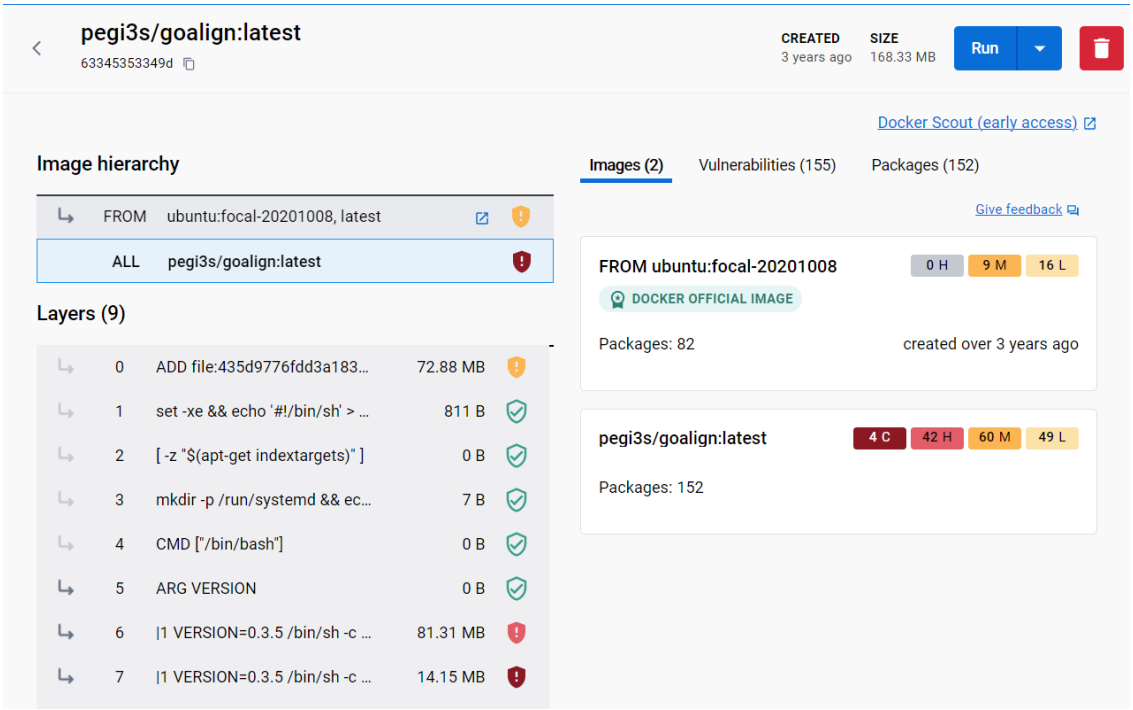


Figure 14 - Image menu in the Docker Desktop application.

The user can check the different layers in the Docker image, check its vulnerabilities, different versions of the image and also the packages. On this menu the user can also run or delete de Docker image.

On this project, Docker Desktop acted as a way to manage the Docker images in the computer and also run the images in a Window's engine.

Docker Desktop also allows the user to directly run Docker images straight from the software, however, it doesn't allow the user to specify parameters.

Another downside to Docker Desktop is despite it allowing the user to run some Docker images in a Windows os, it is unable to run images that require a GUI. This happens because the command xhost + is not available in Windows.

4.5 Portainer

As it was previously explained, Portainer.io is a management UI which allows a user to easily utilize different Docker environments. It consists of a single container that allows for the management of all Docker resources, being compatible with both Linux and Windows.

After Portainer Docker image is correctly installed, set up and running, the Portainer Server is ready to be used. The user just needs to access the open a web browser and go to https://localhost:9443. The last 4 numbers are the port and are part of the program set up. After it runs, the user is met with the page in Figure 15.

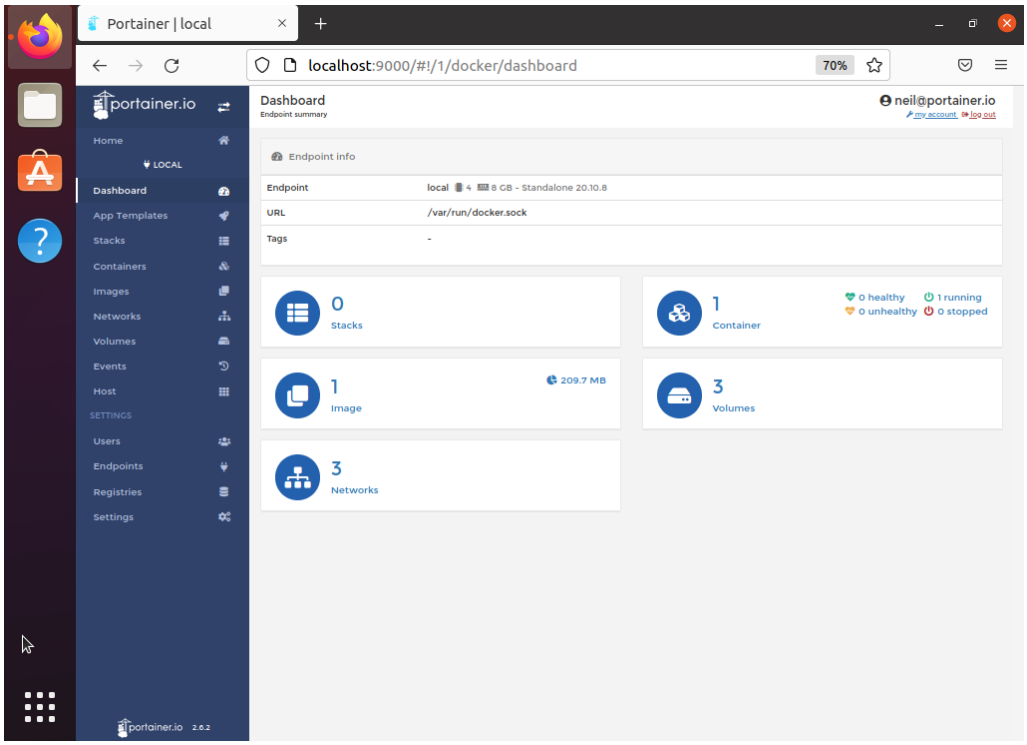


Figure 15 - Portainer main menu.

Here the user can display project stack containers (Figure 16), open a web terminal into any container (Figure 17) and display logs with filtering (Figure 18):

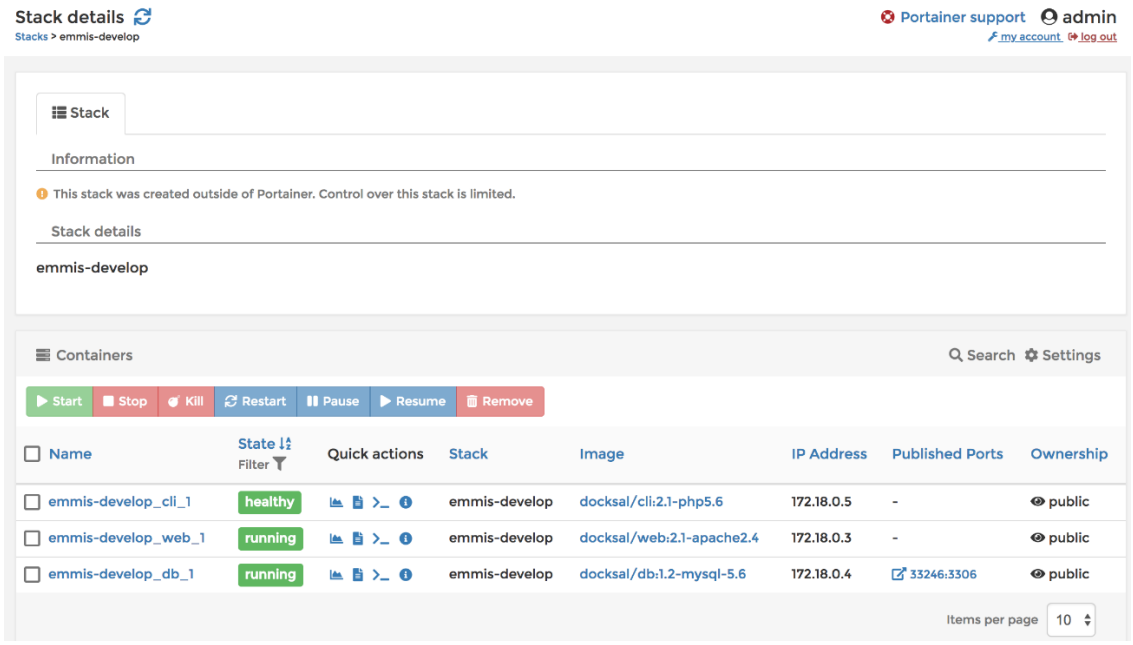


Figure 16 - Menu for project stack containers.

On the menu above, the user can interact with the containers at his disposal.

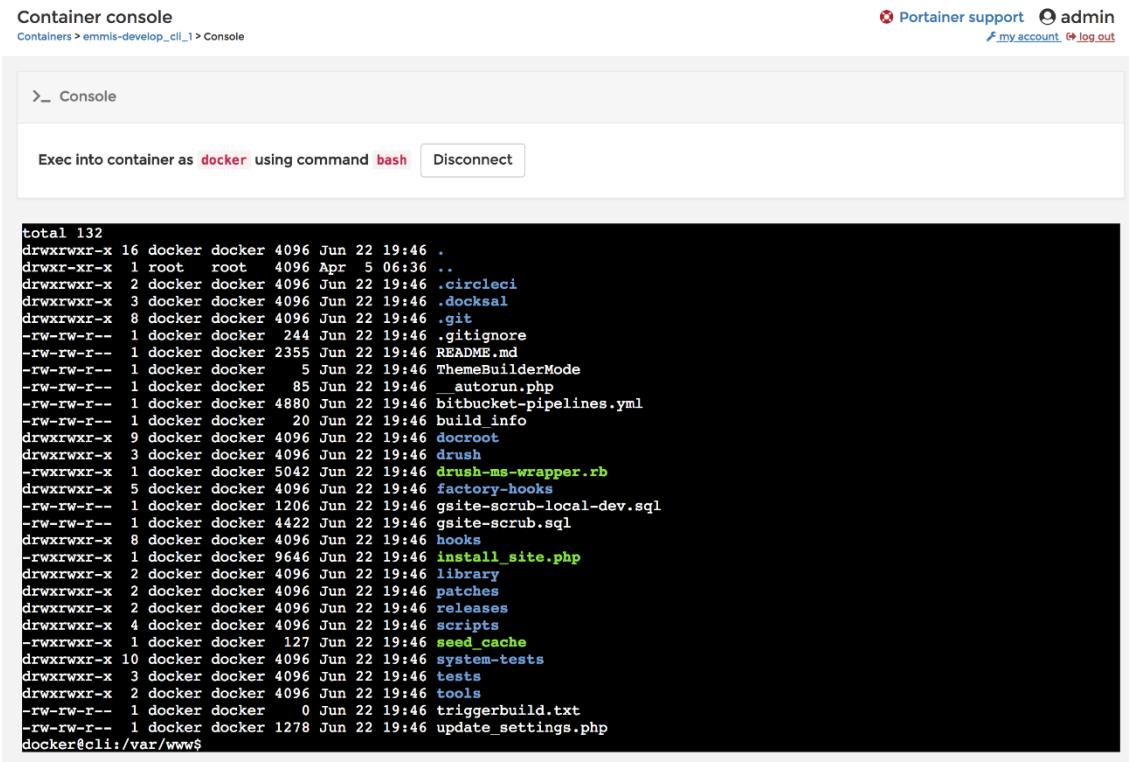


Figure 17 – Portainer's web terminal.

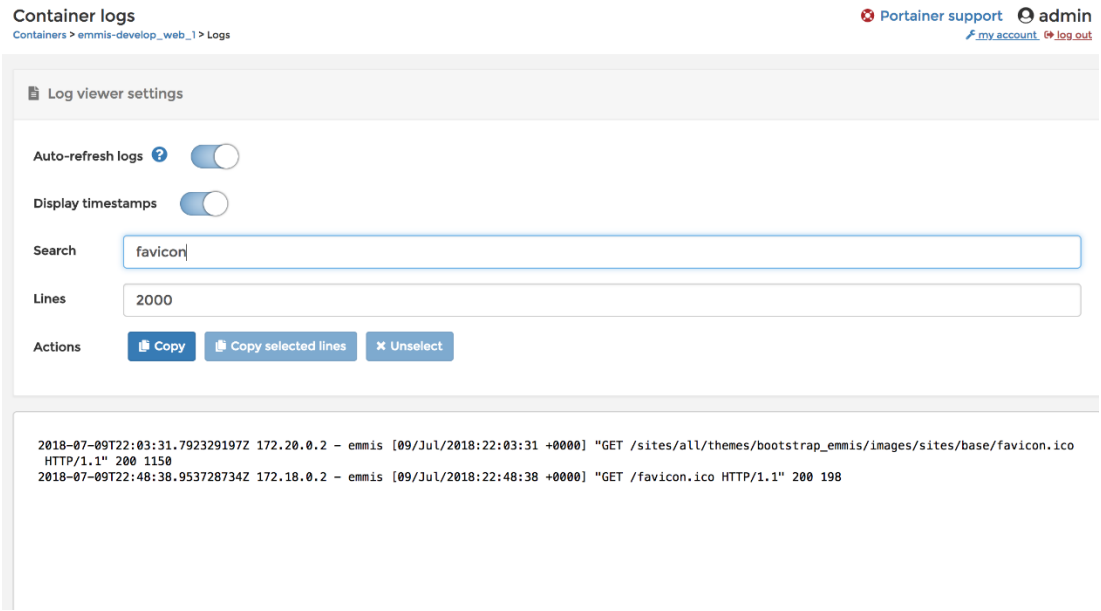


Figure 18 - Portainer's log display.

Portainer was the software utilized to manage the Docker images in the old version of the Virtual Machine. However, currently Portainer for companies only has a paid version so it became an unattractive option. So other software came up as better choices instead of Portainer like Docker Desktop or Docker Manager (a GUI developed during the duration of this project).

4.6 Python

Python was first released in 1991 and it's a general-purpose, high-level programming language. Its design philosophy emphasizes code readability achieving this with the use of significant indentation (Kuhlman, 2012).

Python is an interpreted, object oriented, high level-programing language with dynamic semantics. It's dynamic typing and dynamic binding, along with its high-level built in data structures, make it an appealing language for Rapid Application Development and for usage as scripting of glue language to connect existing components. Due to its straightforward and basic syntax, Python promotes readability, which lowers software maintenance costs. Python's support for packages and modules promotes code reuse and program modularity. The large standard library and the Python interpreter ae freely distributable and accessible for free on all major platforms in source or binary form (Python1, 2024).

On this project, python was the programming language selected to develop the graphical interfaces. Despite its not optimal performance (it's generally slower than languages like C++ or Java), Python still bodes fairly against its competition due to (Cutting & Stephen, 2021):

- Extensive Libraries and frameworks → Python offers a wide range of libraries and frameworks for GUI development (such as Tkinter);
- Cross platform compatibility → Python and its GUI frameworks can support multiple platforms, including Windows, macOS and Linux, which is extremely important for the development of cross-platform applications;
- Community Support → Python has a big and active community, providing extensive documentation, tutorials, and forums for support;
- Scalability → Python allows developers to start with simple applications and gradually add complexity as needed;
- Previous experience → Python was utilized to develop the auto-phylo-pipeliner GUI, a previous project of the Phenotypic Evolution Group.

For the development of the GUI, Tkinter was the primary library used.

The Tkinter package, also known as "Tk interface", is the standard Python interface to the Tcl/Tk GUI toolkit. Tk and Tkinter and both supported on most Unix platforms, macOS and Windows systems (Python2, 2024).

It should be noted that Tcl/Tk is not a single library but rather a collection of different modules, each with separate functionalities and official documentations (Python2, 2024).

Tcl is a dynamic interpreted programming language, just like python and can be used as a standalone programming language. This library has a C interface to create and manage one or more instances of a Tcl interpreter, run Tcl commands and scripts in those instances, and add custom commands implemented in either Tcl or C. Every interpreter is equipped with an event

queue where events can be sent and processed. Tkinter bridges the gap between Python and Tcl, as the latter's execution model is centered around cooperative multitasking (Python2, 2024).

Tk is a Tcl package implemented in C that provides additional custom commands for generating and modifying GUI widgets. Every Tk object loads Tk into its own instance of the Tcl interpreter. Tk's widgets have a lot of customization options, but the result is an outdated look. Tk generates and processes GUI events via Tcl's event queue (Python2, 2024).

Themed Tk (**Ttk**) is a newer family of Tk widgets that improve the appearance on different platforms than many of the classic Tk widgets (Python2, 2024).

Other libraries were utilized to build the GUI, being the most noticeable ones:

- json → this module allows the program to extract the data from the JSON file containing the information of all the Docker images available;
- os → this module provides a portable way of using operating system dependent functionalities, such as Docker commands;
- subprocess → this module allows the user to spawn new processes. This is the main reason why there can be 2 windows from the GUI working at the same time. If it was not for this, the parent windows would be inoperable while the child window is opened;
- threading → this module allows for different activities to run multiple threads in a Python program. This was used for quality-of-life improvements in the user experience;
- webbrowser → this module is a useful web browser controller and allows the program to open a web browser depending on the user's inputs, rounding up to quality-of-life improvements.

Other libraries were utilized but the main ones are listed above.

For this project, Python and Tkinter were the preferred options since this software was used at Pegi3s in previous projects.

5. Development

Updating the virtual machine

Revision of all Docker images

Creation of a standardized set of test data and compilation of the generated data

Development of a graphical interface

In order to facilitate the reading of the current thesis, there is going to be made a separation of the different tasks performed during the internship, such as: update of the virtual machine used, revision of all Docker images in the repository, creation of a standardized set of test data, and collection of the data generated by each Docker image, and finally development of a graphical interface application that will allow the use of all Docker images. The tasks performed are represented on Figure 19.

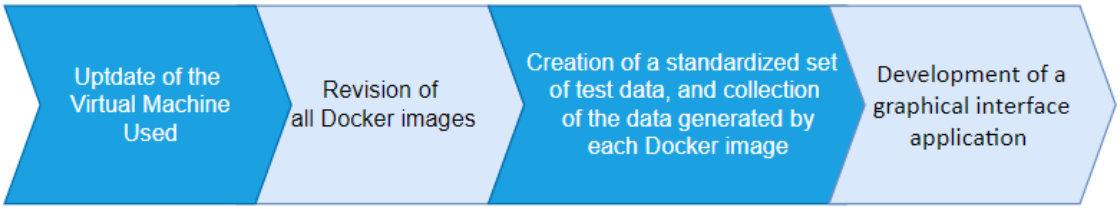


Figure 19 - Work developed in the thesis organized chronologically.

Certain steps could be revisited at any time during the internship. For example, steps two and three might need to be repeated if a new image was added.

5.1 Updating the virtual machine

The first task was to update the virtual machine that was provided on the pegi3s website, as it was outdated and running Linux version 16.04. The OVA file that was to be generated had to run Linux 22.04 (the newest version) and Docker, similar to what happened in the latest version. Also, the log in had to follow the older version, where both the username and the password was pegi3s.

One of the goals to achieve was to first understand the set-up process of a virtual machine, it's capabilities and how a Docker daemon could be run on any machine.

The virtual machine (VM) was created using Oracle VM Virtual Box (Oracle, 2023). The creation process began by clicking "New," as shown in Chapter 2.2. Firstly, it was required to select the name of the VM. The name "pegi3s" was chosen, as it was the previous version's name. The directory where it was stored had no relevance to the VM itself.

The ISO file, a copy of an entire optical disk, was also selected. This optical disk could either be generated from the previous OVA version, which might lead to compatibility issues. Despite both being Linux, the VM's default settings of the previous VM might not be optimal in the newer VM.

The ISO image was obtained in Ubuntu's official website (Figure 20), and it was chosen the desktop version of it in order to create the VM.

	Name	Last modified	Size	Description
	Parent Directory		-	
	SHA256SUMS	2024-02-22 15:31	202	
	SHA256SUMS.gpg	2024-02-22 15:31	833	
	ubuntu-22.04.4-desktop-amd64.iso	2024-02-20 19:39	4.7G	Desktop image for 64-bit PC (AMD64) computers (standard download)
	ubuntu-22.04.4-desktop-amd64.iso.torrent	2024-02-22 15:31	374K	Desktop image for 64-bit PC (AMD64) computers (BitTorrent download)

Figure 20 - ISO image (highlighted in red) from Linux version 22.04 (Ubuntu, 2024).

The highlighted image can be directly imported to the Oracle VM Virtual Box.

After the name, ISO image and where it is going to be stored, it is necessary to select the VM type and version. Them being Linux and Ubuntu 22.04 LTS (Jammy Jellyfish) (64 bit) respectively. After the previous fields were correctly filled, the menu should look like the Figure 21.

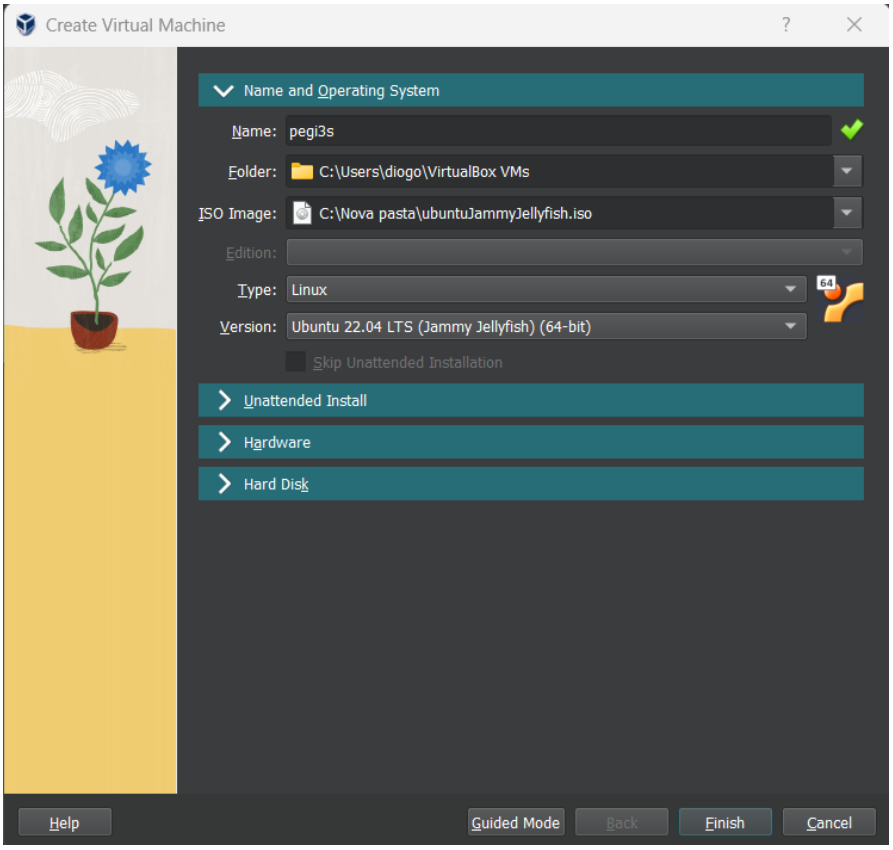


Figure 21 - Name and operating system filled up on the new VM.

The hardware section should take into account the better performance of the VM but also whether the computer running the VM can handle its capacities. It should neither be so slow that running a single Docker image takes an unnecessary amount of time, neither should it be

so heavy where the host computer is unable to run the VM itself. Therefore, the top performance recommended levels presented by the Oracle VM VirtualBox where respected. They were 5000 MB of Ram and 6 CPUs as in Figure 22.

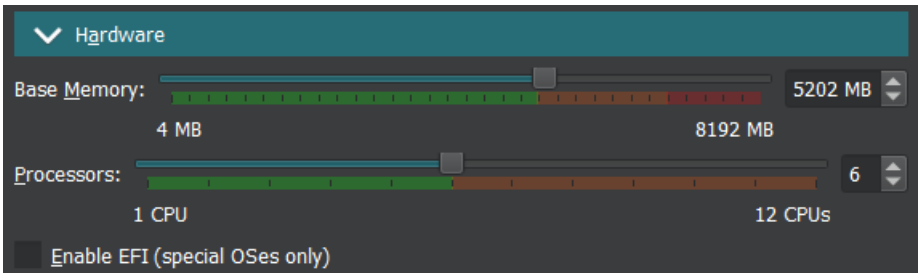


Figure 22 - Hardware choices for the new VM.

Finally, a hard disk had to either be either created or the user had to utilize and already existing hard disk. Here, it was chosen to start from an already existing one with the downside of the size of the hard disk being fixed. In order to do that, a list of already existing hard disks was obtained from OSBoxes (Figure 23) (OSBoxes, 2024). A website that offers ready-to-use Unix/Linux guest operating systems for both architectures 32bit and 64bit for free (OSBoxes, 2024).

Figure 23 - Ubuntu 22.04 obtained from OSBoxes (OSBoxes, 2024).

The choice of starting from an already existing rather than a new one came from the fact that every time the latter was tried, it would pre-allocate the disk size as 100 GB, not allowing the user to switch this setting. The space was far too large for the necessities of the VM; therefore, an already existing hard disk was preferred. This option meant that the disk was always going to be pre-allocated to 50 GB, which was preferred to the 100 GB option (Figure 24).

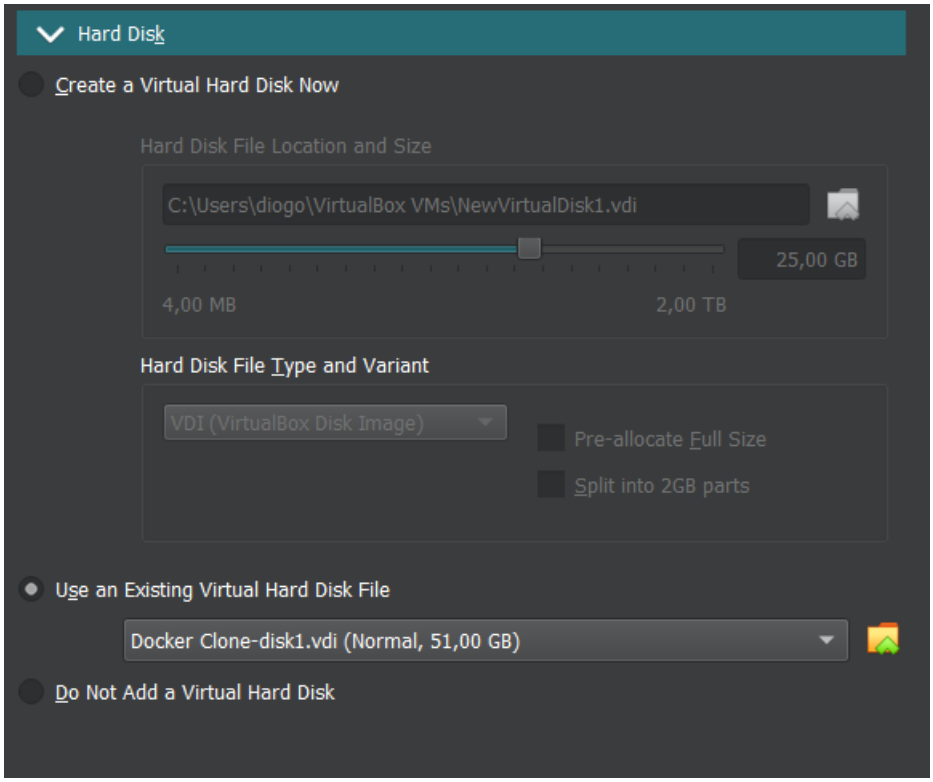


Figure 24 - Hard disk section filled.

The Virtual Hard Disk (vdi) that is placed on the figure above is a copy of an already created vdi from the OSBoxes' one. This was particularly helpful when a user wanted to create a clone of an already existing VM, in order to test on it without changing the original VM's hard disk.

After creating the VM, it should act as though it is a computer booting up for the first time, so it was necessary to set it up accordingly. The virtual box had to fulfill the following requirements:

- The username and password had to be both pegi3s or something relating to it;
- When booting up the browser, it should open the Bioinformatics Docker Images Project's website;
- Docker needs to be installed and correctly set up;
- Portainer needs to be installed and correctly set up.

The username and password changes can be performed in the Settings app under the Users tab. Here, a Linux user can set up their login information. After correctly performing these steps, the login menu should look like the one in Figure 25.

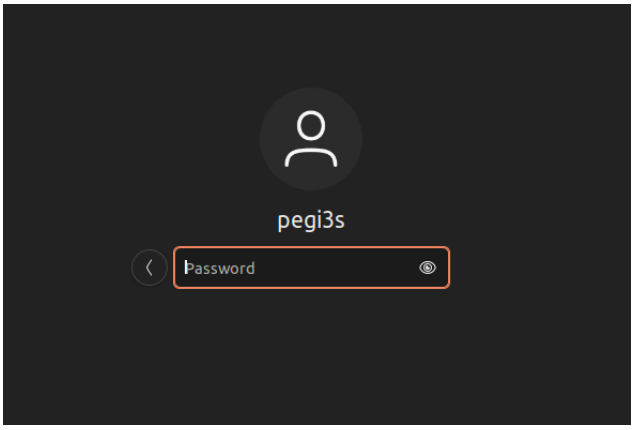


Figure 25 - Log in for the VM.

Afterwards, it was necessary to correctly set up the browser settings. For this, it was necessary to go to the browser's settings and on the home tab, switch the default homepage to the bioinformatics pegi3s link. The results of the correct configuration can be seen in Figure 26.

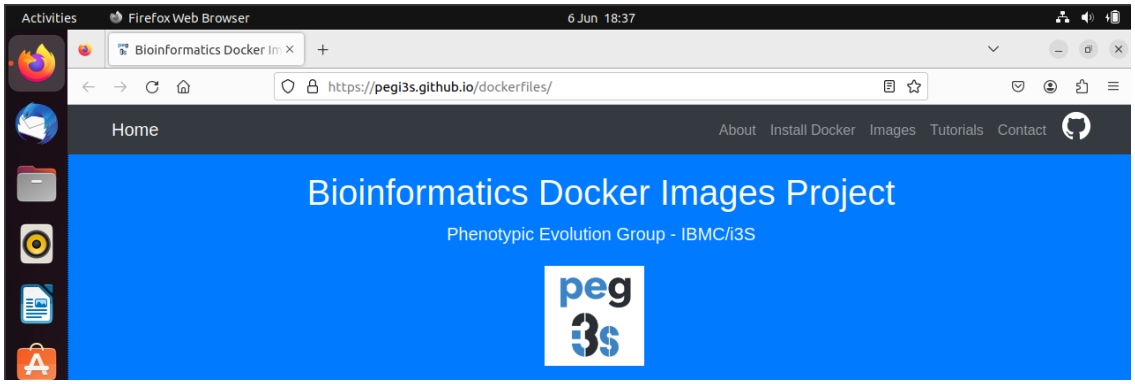


Figure 26 - Default browser of the VM.

The most important step would be to correctly set up Docker on the VM. To do this, the following shell commands had to be run in the Linux terminal (Code 1):

```
    # Add Docker's official GPG key:
    sudo apt-get update
    sudo apt-get install ca-certificates curl
    sudo install -m 0755 -d /etc/apt/keyrings
    sudo   curl   -fsSL   https://download.docker.com/linux/ubuntu/gpg   -o
/etc/apt/keyrings/docker.asc
    sudo chmod a+r /etc/apt/keyrings/docker.asc

    # Add the repository to Apt sources:
    echo \
      "deb        [arch=$(dpkg        --print-architecture)        signed-
by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \
      $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
      sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
sudo apt-get update
```

Code 1 - Code to set up Docker's apt repository (Docker5, 2024).

The code above firstly ensures the safe communication between the Docker repositories and the computer running Docker. The GPG key ensures that packages obtained from the repositories haven't been tempered with or compromised during the download process. The GPG is an implementation of a standard know as PGP (Pretty Good Privacy). It uses a system of "public" and "private" keys for the encryption and signing of messages and data (Privex, 2024).

The second part of the code adds the repository to the Advanced Package Tools (APT) sources for the VM to use a specific source for downloading and installing Docker packages.

After this is done the shell command in Code 2 is run to install the latest version of the Docker packages.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Code 2 - Install Docker packages (Docker5, 2024).

After this is done, Docker should be correctly set up on a computer. To check if this is true, the command in Code 3 is used.

```
sudo docker run hello-world
```

Code 3 - Hello-world command from Docker.

The command would run successfully every time. However, when trying to run the command without the sudo part (which calls for super user permissions every time from the user which is not very user friendly, plus it can get annoying overtime), the terminal would present us with the error message shown in Figure 27.

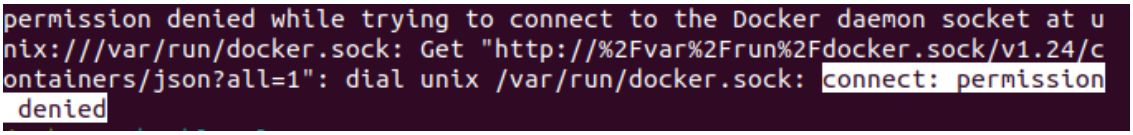


Figure 27 - Docker installation error message.

The error from the figure above happened every time Docker was set up on a new VM. This happened because the Docker daemon binds to a Unix socket rather than a TCP port. By default, the Unix socket is owned by the root user, so other users can only access it using sudo. The Docker daemon itself always runs with root privileges. So, in order for it to always run with root privileges, and to avoid the user adding sudo before every command, a Unix socket needs to be created. It being accessible by members of a Docker group (Docker6, 2024). This can be achieved by creating a group and adding Docker to it with the commands in Code 4.

```
Sudo usermod -aG docker ${USER}
Su - ${USER}
```

Code 4 - Commands to create and add a Docker group to a Unix socket.

The first command adds the current user to the Docker group, granting them permission to execute Docker without using "sudo". The second command switches to that user's account to apply the changes to the user's session, ensuring that the new group membership is recognized without needing to log out and back in again.

After Docker was correctly set up on the VM, it was necessary to add a GUI so the user could interact with the Docker images that were present on the computer, for this, it was necessary to set up Portainer on the VM. This choice came from the fact that it was the program being used on the outdated VM.

However, when trying to install and set up Portainer, it was noted that the desired version would have to be paid., in opposition to what happened when developing the previous VM. This wasn't recommendable for the project since it should avoid adding any extra costs to the department. So, it became necessary to create a GUI that allowed the users to view what Docker images they had on their machines and delete them. This came with the advantage of making a version that would better fit the projects necessities.

5.1.1 Docker Manager

The Docker Manager was an application developed during the internship in order to fulfill the needs of having a GUI that allows users to interact with the Docker images present on the machine they are using. As part of the development process, a table for both Functional Requirements and Non-functional requirements alongside a use-case diagram was created.

The functional requirements table lists specify the functionalities that the software must meet, for example image listing or deletion. These specifications are listed in Table 5 and outline what the system must do to satisfy user demands and accomplish its goals.

Table 5 - Functional requirements table for Docker Manager.

ID	Description
FR01	The system should be capable of displaying a comprehensive list of all images stored on the user's machine.
FR02	The system should provide a GUI for interacting with Docker images.
FR03	The system should allow the user to view the information of a Docker image like: Name, Image ID, Date and Status.
FR04	The GUI should be responsive and adaptable to sensible sizes.
FR05	The system should be able to select and uninstall a Docker image from the list.
FR06	The system should be able to delete unused Docker images.
FR07	The list view of the Docker images should come with a scroll wheel.

FR08	The system should allow the user to refresh the list view of Docker images.

The non-functional requirements table highlights quality attributes and constraints that the software must adhere to, such as performance, security, usability and scalability. These specifications outline how the system must function and behave outside of its main features in order to provide a positive user experience and optimal system performance (Table 6).

Table 6 - Non-functional requirements table.

ID	Description
NFR01	The system should be scalable
NFR02	The system should be able to handle a big volume of data
NFR03	The system should always be available and functioning correctly
NFR04	The system should run regardless of the operating system and machine
NFR05	The system should be of low complexity and intuitive
NFR06	The system should be available as a Docker image and also a standalone software

A use case diagram was created to visually represent the interactions between users and the system, providing a clear understanding of the functionalities offered by the Docker Manager. The use case diagram acted as the application's blueprint, detailing the different features and functionalities that users can access through Docker Manager. The diagram facilitates the identification and documentation of the application's needs by providing a visual representation of the interactions between users and the system. The use case diagram for Docker Manager can be seen in Figure 28.

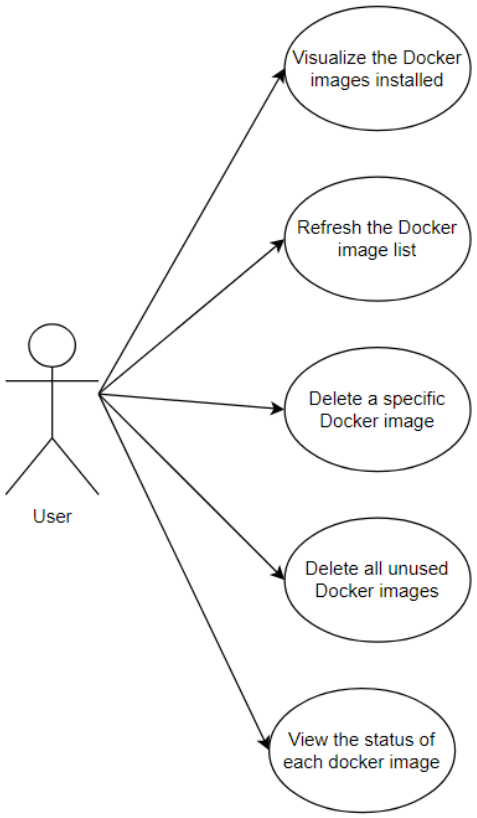


Figure 28 - Use Case Diagram for Docker Manager.

As we can see from the tables and diagram above, the application should fulfill all the previously mentioned requirements. For this a mockup was designed using Docker desktop as a reference with the help of Canvas (Figure 29).

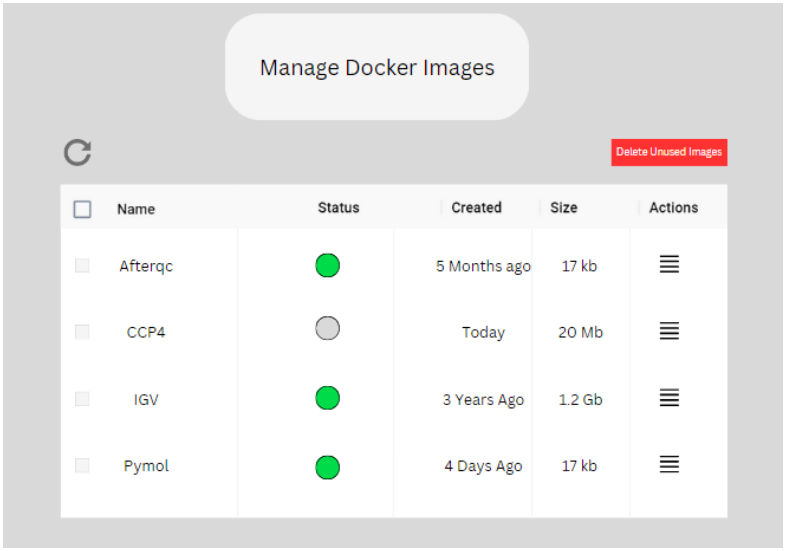


Figure 29 - Docker Manager mockup.

As we can see on the image above, the GUI would only be a single window with the application name above (the name has been since changed to the current version). Above the list view, there are 2 buttons, one that allows the user to refresh the list and on the right one that deletes all unused Docker images. Finally, the image list should show the images' name, status, when they were created and their size. The action tab was latter removed, since the only action was to delete the image.

The GUI was developed in Python with the library Tkinter. Creating a window in Tkinter is easy, since it's as simple as using Code 5.

```
# Create the main window
window = tk.Tk()
window.title("Manage Docker Images")
window.geometry("1000x500")
```

Code 5 - Tkinter window creation

It's important to highlight the latter lines of code since they are the foundation of all GUI developed during the course of the internship. The geometry can be adjusted. This was chosen as it was the one that better fitted the proposed layout.

The code has been split between two files: main.py and manageDocker.py. The main.py file handles all the basic functions for setting up the layout and interaction with the Docker images. The manageDocker.py file manages the communication between the application and the terminal. The latter serves as the backbone of the Docker Manager, as it populates the Docker list and allows the user to delete Docker images, which is one of the main purposes of creating this GUI. To communicate with the terminal, the subprocess library was implemented. In Code 6, we can see an implementation of the subprocess library in the function to delete Docker images.

```
def delete_docker_image(image_id):
        try:
            subprocess.run(f"docker rmi -f {image_id}", shell=True,
check=True)
            return True
        except subprocess.CalledProcessError as e:
            print("Error:", e)
            return False
```

Code 6 - Function to delete Docker images in Docker Manager.

The function above receives image_id as an argument. It then constructs a command string to be executed in the terminal using the subprocess library. If the command runs successfully, the function returns True, indicating that the Docker image was deleted. Otherwise, it prints the error to the Python console and returns False, indicating that something went wrong during execution. This approach ensures that if an error occurs, the GUI will continue to function properly without crashing.

The first version of the Docker Manager has been made available to the public, and any user can obtain it by pulling it from the Docker pegi3s repository. The GUI that is available with an image selected is shown in Figure 30.

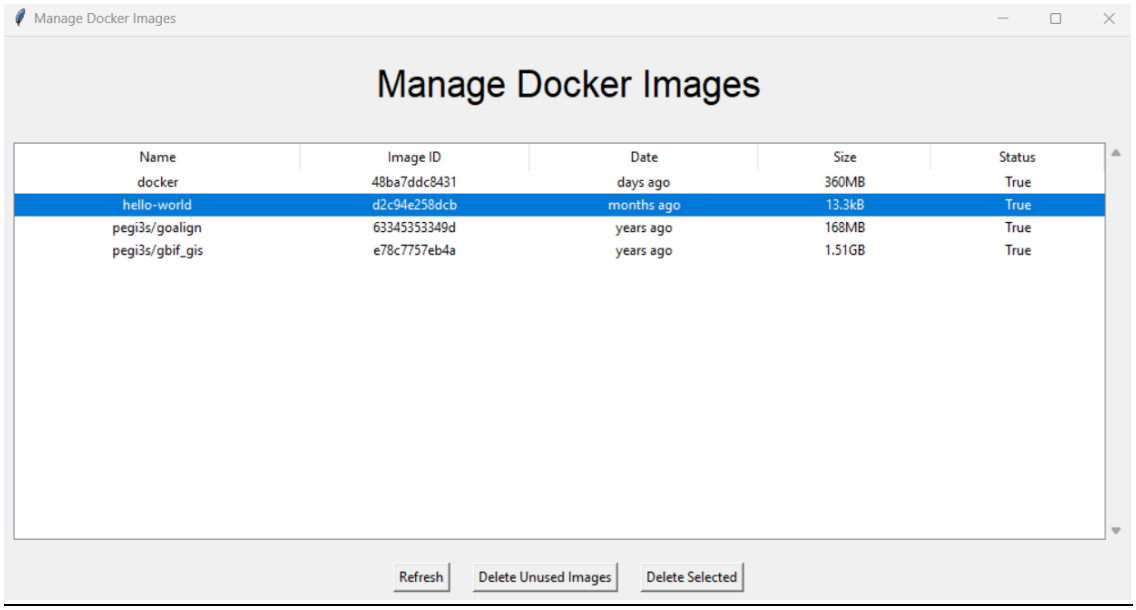


Figure 30 - Manage Docker Images Menu with an image selected.

As we can see in the image above, a user can obtain a list of all the Docker images present on their machine. The images can be identified by their name and ID. It is also shown how long the images were created, their size and their status. To delete an image, a user simply needs to select it as shown in the picture and press the “Delete Selected” button. This action updates the table automatically.

The feedback has been positive; however, a small bug was found (If an image is created within the minute the Docker Manager is opened, the creation date it displayed as “less than a minute ago”, which is too big for the tree view. Consequently, the sentence appears cut off), and a few suggestions have been made to add to the functional requirements.

The suggestions that were presented can be consulted in Table 7.

Table 7 - User recommendations for Docker Manager.

ID	Description
FRN01	Add tag to the Docker list info
FRN02	Add a search option to the Docker list based on Name
FRN03	Add a sorting option based on size

In Docker, a tag is a label that is applied to a Docker image that helps in identifying and managing different versions of the same Docker image. A tag is important in controlling what version is being used, the stability and flexibility of the image. With a tag, a user can ensure that deployments use a known version of the image, and it can also make it easier to switch between images.

Regarding the remaining recommendations, both would be quality of life improvements to the application. As of now, there have been no updates to the Docker Manager since all these changes are low priority and the software is functioning as expected meaning there is no urgency in updating it.

After the application was fully developed, a Docker image was created using the software. To do that, firstly the code had to be run inside a Linux environment. For this it was used the computers of the laboratory since they would be faster than the VM created. Then, a file called "Dockerfile" was created. This file is necessary to create a Docker image, as it contains the instructions for building the image (Code 7).

```
FROM pegi3s/docker

RUN apt-get -y update
RUN apt-get install -y python3 zip

COPY my_code.zip /opt
WORKDIR /opt
RUN unzip my_code.zip && rm my_code.zip
ENTRYPOINT ["./my_code"]
```

Code 7 - Instructions to build the Docker Manager in Docker.

The "FROM pegi3s/docker" specifies the base image for the Docker image being built. The "pegi3s/docker" is an existing image that provides an Ubuntu 22.04 operating server with Docker pre-installed. This setup allows for the usage of the Docker commands within the code that need to be executed in the terminal.

The second line is meant to update the package list for the "apt-get" package manager, ensuring that it has the latest available packages. The "-y" flag will automatically answer "yes" to any prompts, allowing the update to run without user interaction. The following line installs both python3 and zip. These are needed to run Python scripts and handle zip files, respectively. Afterwards, it is going to copy the my_code.zip (this needs to be in the same directory as the Dockerfile) into a directory that is going to create called "/opt". Following this, the working directory is selected as "/opt". All subsequent commands are going to be run from this directory.

The second to last line unzips the files (which contain both the "main.py" and "manageDocker.py" scripts). After these files are successfully unzipped, the "my_code.zip" is deleted.

The final line sets the entry point for the Docker container. When the container starts, it will execute the "my_code" script that was previously unzipped and is located in the "/opt" directory. After creating the Docker file, the last step to creating a Docker image is building it (Code 8).

```
sudo docker build ./ -t pegi3s/docker_manager
```

Code 8 - Build Docker image command.

This command runs with superuser privileges to ensure Docker has the necessary permissions to perform the build operation. The “-t” flag tags the image with a name, which is specified immediately after the flag. This command needs to be run on the directory that has the Dockerfile. In Figure 31, the Docker image was successfully built with all dependencies and necessary files correctly loaded onto it.

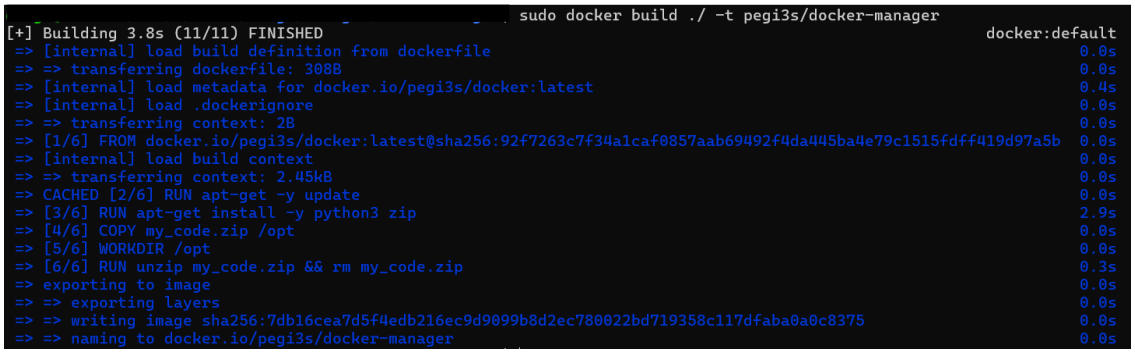


Figure 31 - Successfully built docker-manager.

To run the Docker image that was just built, the following command shown in the Code 9 needs to be run on the terminal:

```
    docker run --rm -ti -e USERID=$UID -e USER=$USER -e DISPLAY=$DISPLAY -v
/var/db:/var/db:Z        -v        /tmp/.X11-unix:/tmp/.X11-unix        -v
$HOME/.Xauthority:/home/developer/.Xauthority                          -v
/var/run/docker.sock:/var/run/docker.sock -v /tmp:/tmp pegi3s/docker-manager
```

Code 9 – Command to run docker-manager image.

The “—rm” tag means that the container is automatically removed after it exits. The “-ti” are two separate options combined together, and they are meant to ensure that the Docker image is interactive. The following section is used on every image that is a GUI. Finally, “pegi3s/docker-manager” is the name of the Docker image to run the container from. If this image does not exist in the user computer, Docker will pull it from the Docker repositories.

Since this is a GUI, before running the code mentioned above, in order to disable access control, allowing any host to connect to the X server allowing the usage of GUI, a command of “xhost +” must be sent first (IBM, 2024).

As of now, any user can pull this image from the pegi3s Docker Project website (Figure 32).

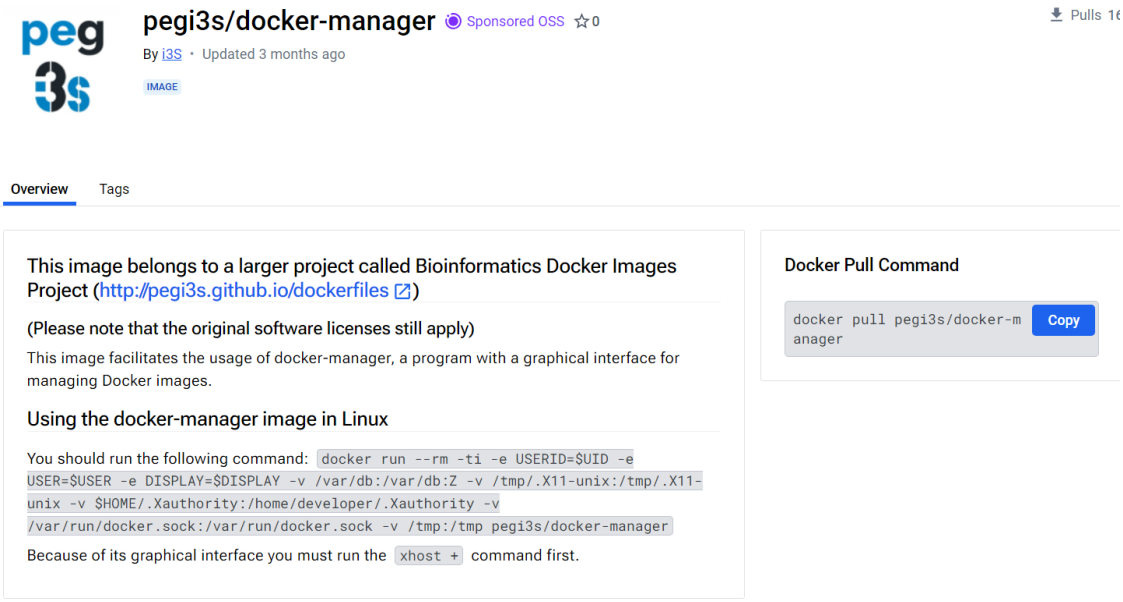


Figure 32 - Docker Manager page from pegi3s (pegi3s1, 2024).

After creating this Docker image, Portainer was no longer necessary to add to the new VM. This is because to manage the Docker files in a user's machine, all they needed to do was run the aforementioned Docker image. With this, the VM was successfully updated.

Currently, this VM is only available on GitHub since better software options like Docker Desktop or the previously mentioned Ubuntu app for Windows have been found. These alternatives achieve the same results as the VM while allocating more resources to running Docker images since they run directly on the user's computer rather than in a VM. This results in faster running times. The VM is now briefly mentioned for people that have MacOS since there is no way to run a Linux environment in that os and the Docker Desktop does not allow users to run GUI.

5.2 Revision of all Docker images

After updating the VM, it was necessary to revise all the Docker images checking for the following:

- whether they all follow the same structure on the website;
- the required type of test data for each Docker image;
- which ones have a GUI;
- if a Docker image has multiple options for creating the run command;
- other observations.

This step on the internship was crucial on better understanding how the Docker images are structured and helping on the following tasks especially on the creation of the JSON file that was crucial on the development of the GUI. The acquired data was organized in Table 8.

Table 8 - Docker images revised data.

Nome	Test Data	Various Options	GUI	Website Standard
abyss	<10MB	N	N	N
adops	N	N	Y	N
aegean	GFF3	Y	N	N
afterqc	>1GB	N	N	Y
alphafold_db	Uniprot ID	N	N	Y
alter	txt	N	Y	Y
autoaugustus	N	Y	N	N
bdbm	N	N	Y	N
bedtools	N	Y	N	Y
bioconvert	N	Y	N	Y
blast	N	N	N	N
bowtie1	Genome+ fastq	N	N	Y

The table above was abbreviated to showcase the most relevant information on how the Docker images were revised.

The “Test Data” column indicates which images have available test data being provided to the users and specifies the type of test data. For example, the abyss image provides a FASTA file smaller than 10 MB, whereas the afterqc image provides a FASTAQ file larger than 1GB. The “txt” tag indicates that it’s being provided a text file. These files are often small FASTA files that are stored as “.txt” files. If no test data is provided, the images are marked as “N”. Any other type of file provided is tagged with the file’s type name. It should be noted that providing the missing data was not a goal of the work but rather to understand the level of files that the application to be developed was going to deal with.

The various option column is meant to distinguish the images that have multiple functionalities (for example, “bedtools” is a list of tools with varying functions). If an image has various options it’s marked with “Y” for yes, and if it doesn’t it’s marked with “N” for no. Same logic is applied on whether the image is a GUI or not.

The “Website Standard” column is meant to check whether images follow a standardized structure on the website or not. When opening any Docker image on the pegi3s website, it should display the title at the top. The overview tab should start with a description of the Docker image, including the link to its manual. Below it, it should provide information on running the image in Linux, including the necessary command to run the image and details on what users should modify to create their personal command. Additionally, it should include any other relevant commands or explanations of specific options in the command. Underneath all of this it should provide the test data and, finally, instructions on how to run it on Windows (now redundant due to new findings over the course of this project). If the website follows this order it’s marked with “Y”, and if it doesn’t it’s marked with “N”. Figure 33 showcases a standard pegi3s website.

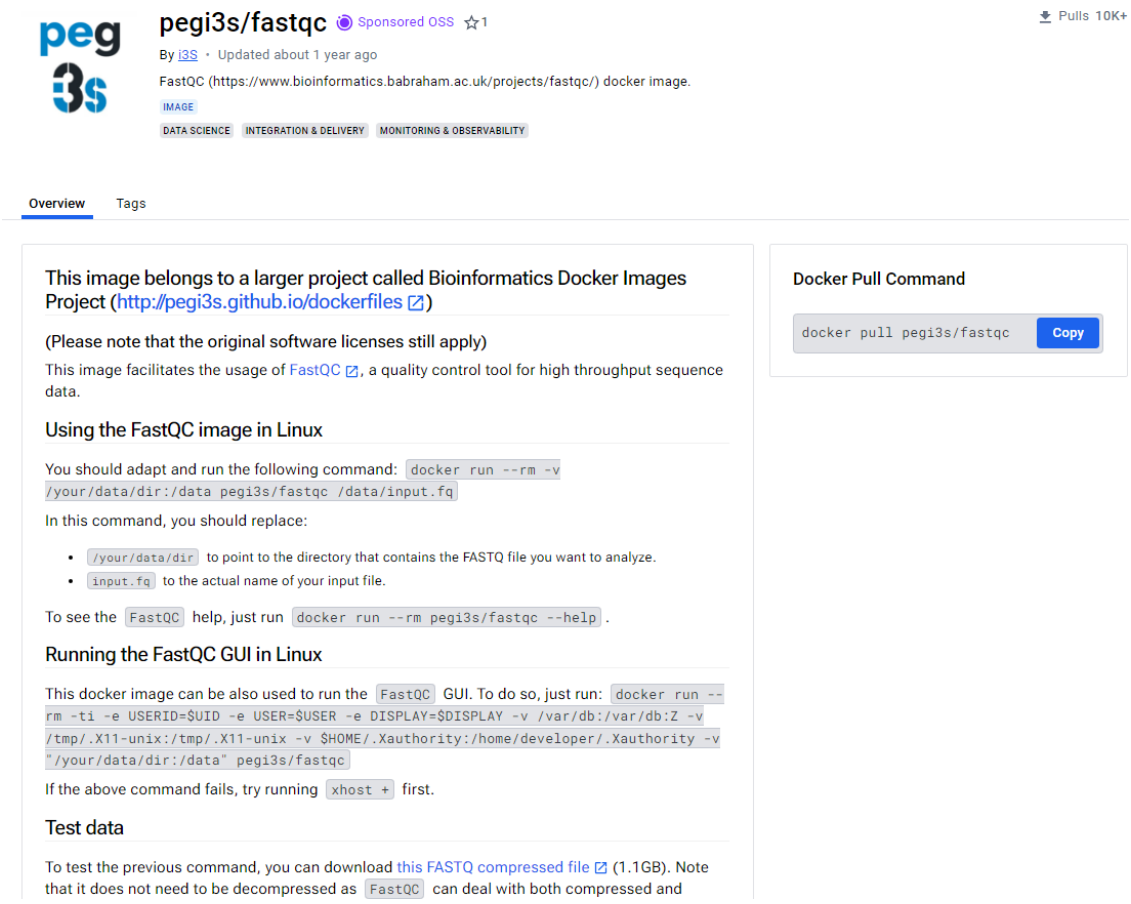


Figure 33 - fastqc Pegi3s' page (pegi3s2, 2024).

All images' websites should follow the structure presented above.

Finally, the column "Observations" was omitted due to its very comprehensive nature. In this table there are remarks on non-functioning links (some images' manual links don't work) and the possible data type for the images. Another important remark is that "xhost +" is necessary to run any GUI, as they only run when this command is invoked. These were the most common types of observations.

Upon the completion of this task, a better understanding of the project and the images themselves was achieved which would prove valuable on the development of the GUI to run Docker images.

5.3 Creation of a standardized set of test data and compilation of the generated data

The creation of a standardized set of test data and compilation of a collection of the generated data was also pursued as this facilitates the test of the Docker images. It should be however noted that creating test data for all images was not a requisite. After gathering what

type of data each image required, it was possible to select a set of standardized files that covers most molds. The origin of each data file can be checked in Table 9.

Table 9 - Data types and their origins.

Data Type	Origin
Small FASTA	Created from desktop
Big FASTA	Created from the FASTQ file
Not aligned FASTA	Previous test data from goalign
FASTQ	Previous test data from fastqc
GFF3	Previous test data from aegean
Newick	Output from fastree image
Pdb File	Previous test data from whiscy
Mega	Conversion of small FASTA with alter image
Binary	Previous test data from raxml
Genome	Previous test data from bowtie1
NEXUS	Conversion of small FASTA with alter image
Sam/Bam File	Previous test data from qualimap
Data test for assemblers	Previous test data from abyss

A FASTA file (Figure 34) is a text-based format for representing either nucleotide sequences or peptide sequences, in which base pairs or amino acids are represented using single letter-codes. In FASTA format, a sequence consists of lines of sequence data after a single line description. The description line is distinguished by the sequence data using the “>” symbol in the first column. Sequences are usually represented in the standard IUB/IUPAC amino acid and nucleic acid codes, with the following exceptions (Zhang Group, 2024):

- Lower-case are accepted and are mapped to upper-case;
- A dash or single hyphen is used to represent a gap of indeterminate length;
- U and * are acceptable letters in amino acid sequences;
- Numerical digits in the query sequence should be removed or replaced by appropriate letter codes.

```
>D_melanogaster_128up-PA
ATGAGCACAATATTGGAGAAAATCTCGGCCATCGAGTCGGAGATGGCCCG
AACCCAAAAGAACAAGGCCACCTCGGCCCATTTGGGTCTACTGAAGGCGA
AGCTGGCTAAGCTGCGACGCGAACTGATTTCCCCCAAAGGAGGCGGCGGC
GGAACCGGCGAAGCTGGCTTCGAGGTGGCCAAGACTGGAGATGCCCGGGT
GGGATTCGTAGGGTTTCCTTCTGTGGGTAAATCCACACTGCTCTCCAACT
TGGCTGGCGTTTACTCCGAGGTGGCGGCATACGAATTCACAACGTTGACC
```

Figure 34 - Provided FASTA file format.

Three types of FASTA files are used. The Small FASTA file occupies 9 KB and contains sequences from the same gene in different species, while the Big FASTA has 2.87 GB. These varying sizes exist because some programs are designed to handle high amounts of data, requiring a bigger FASTA file to utilize their capacities effectively. The unaligned FASTA file contains sequences that are not arranged in a multiple sequence alignment.

A FASTQ file is similar to FASTA though there are differences in syntax as well as integration of quality scores. Each sequence requires at least 4 lines (NGS Analysis1, 2024):

- The first line is the sequence header and starts with "@";
- The second line contains the sequence;
- The third line starts with "+" and can have the same sequence identifier appended;
- The fourth line contains the quality scores.

A GFF3 (General Feature Format) file is a tab-delimited text file that accommodates a wide range of data, includingCDS, microRNAs, binding domains, ORFs and more (Figure 35). Holding information on any and every feature that can be applied to a nucleic acid or protein structure. The original GFF format suffered a lot of variations, making many versions of this file incompatible with each other. GFF3 is the last accepted format that attempts to address the many issues that were missing from previous versions. GFF3 has nine required fields. Not all of them need to be utilized, meaning they can either be left blank or with a default value of "." (NGS Analysis2, 2024) :

- Sequence ID;
- Source, that describes the algorithm that generated this feature;
- Feature type, which describes what the feature is (mRNA, domain, etc.) and its constrains;
- Feature Start;
- Feature End;
- Score, typically E-values for sequence similarity and P-values for predictions;
- Strand;
- Phase, that indicates where the feature begins with reference to the reading frame. The phase is one of the integers 0, 1, or 2, indicating the number of bases that should be removed from the beginning of this feature to reach the first base of the next codon;
- Attributes, a semicolon-separated list of tag-value pairs, providing additional information about each feature.

```
##gff-version 3
##sequence-region   chr1 1 10000
chr1    .   gene            1000    5000    .   +   .   ID=gene00001;Name=GeneA
chr1    .   mRNA            1000    5000    .   +   .
ID=mrna00001;Parent=gene00001;Name=TranscriptA
chr1    .   exon            1000    1200    .   +   .   ID=exon00001;Parent=mrna00001
chr1    .   exon            1500    1700    .   +   .   ID=exon00002;Parent=mrna00001
chr1    .   exon            3000    3200    .   +   .   ID=exon00003;Parent=mrna00001
chr1    .   CDS             1100    1150    .   +   0   ID=cds00001;Parent=mrna00001
chr1    .   CDS             1600    1650    .   +   2   ID=cds00002;Parent=mrna00001
chr1    .   CDS             3100    3150    .   +   1   ID=cds00003;Parent=mrna00001
chr1    .   gene            7000    9000    .   -   .   ID=gene00002;Name=GeneB
chr1    .   mRNA            7000    9000    .   -   .
```

Figure 35 – Provided GFF3 file format.

A Newick file, Figure 36, is a simple format that is used to write out trees in a text file. Despite it being a hard-to-read format for humans, it is very useful for exchanging trees between different types of software (Mega, 2024).

```
(D_ficusphila_128up-PA:0.09291,(D_yakuba_128up-PA:0.01851,D_erecta_128up-
PA:0.02134)0.903:0.00692,(D_melanogaster_128up-PA:0.01935,(D_sechellia_128up-
PA:0.01896,D_simulans_128up-PA:0.00422)0.932:0.00803)0.894:0.00624);
```

Figure 36 - Provided Newick file format.

A PDB (protein data bank) file is a standard for files containing atomic coordinates (Figure 37). The complete PDB file specification provides for a wealth of information, including authors, literature references, and the identification of substructures such as disulfide bonds, helices, sheets, and active sites (The Regents of the University of California, 2002).

```
HEADER    CELL CYCLE, SIGNALING PROTEIN           24-OCT-00   1G3N
TITLE     STRUCTURE OF A P18(INK4C)-CDK6-K-CYCLIN TERNARY COMPLEX
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: CYCLIN-DEPENDENT KINASE 6;
COMPND   3 CHAIN: A, E;
COMPND   4 SYNONYM: CELL DIVISION PROTEIN KINASE 6, PROTEIN KINASE CDK6;
COMPND   5 EC: 2.7.1.37;
COMPND   6 ENGINEERED: YES;
COMPND   7 MOL_ID: 2;
COMPND   8 MOLECULE: CYCLIN-DEPENDENT KINASE 6 INHIBITOR;
COMPND   9 CHAIN: B, F;
COMPND  10 SYNONYM: P18(INK4C), CYCLIN-DEPENDENT KINASE INHIBITOR 2C, P18,
COMPND  11 INHIBITS CDK4, CYCLIN-DEPENDENT KINASE 4 INHIBITOR C, P18-INK4C;
COMPND  12 ENGINEERED: YES;
COMPND  13 MOL_ID: 3;
COMPND  14 MOLECULE: V-CYCLIN;
COMPND  15 CHAIN: C, G;
COMPND  16 SYNONYM: K-CYCLIN, FUNCTIONAL CYCLIN D HOMOLOG;
COMPND  17 ENGINEERED: YES
```

Figure 37 - Provided pdb file format.

A Mega file, Figure 38, is a basic text file containing DNA sequence, protein sequence, evolutionary distance, or phylogenetic tree data.

```
#mega
TITLE: MSA converted with ALTER 1.3.3

#D_melanogaster_128up-PA  ATGAGCACAA TATTGGAGAA AATCTCGGCC ATCGAGTCGG AGATGGCCCG
#D_sechellia_128up-PA     ATGAGCACAA TATTGGAGAA AATCTCGGCC ATCGAGTCGG AGATGGCCCG
#D_simulans_128up-PA      ATGAGCACAA TATTGGAGAA AATCTCGGCC ATCGAGTCGG AGATGGCCCG
#D_yakuba_128up-PA        ATGAGCACAA TTTTGGAGAA AATCTCGGCC ATCGAGTCGG AGATGGCCCG
#D_erecta_128up-PA        ATGAGCACAA TTTTGGAGAA AATCTCGGCC ATCGAGTCGG AGATGGCCCG
#D_ficusphila_128up-PA    ATGAGCACAA TTTTGGAGAA AATCGCGGCC ATCGAGTCGG AGATGGCCCG

#D_melanogaster_128up-PA  AACCCAAAAG AACAAGGCCA CCTCGGCCCA TTTGGGTCTA CTGAAGGCGA
#D_sechellia_128up-PA     AACCCAGAAG AACAAGGCCA CCTCGGCCCA TTTGGGTCTG CTGAAGGCGA
#D_simulans_128up-PA      AACCCAGAAG AACAAGGCCA CCTCGGCCCA TTTGGGTCTG CTGAAGGCGA
#D_yakuba_128up-PA        AACCCAGAAG AACAAGGCCA CCTCGGCGCA TTTGGGTCTG CTGAAGGCGA
#D_erecta_128up-PA        AACCCAGAAG AACAAGGCCA CCTCGGCGCA TTTGGGTCTG CTGAAGGCAA
#D_ficusphila_128up-PA    CACCCAGAAG AATAAGGCCA CCTCGGCGCA TTTGGGTCTC CTGAAGGCTA
```

Figure 38 - Provided Mega file format.

A binary file is a .txt file that contains ones and zeroes.

A genome file is a FASTA file that contains the complete DNA sequence of a specific organism (for this data set it's for Saccharomyces cerevisiae).

A NEXUS file, Figure 39, stores information about taxa, morphological and molecular characters, distances, genetic codes, assumptions, sets, trees, etc. This file format begins with the character "#NEXUS" but is otherwise organized into major units known as *'blocks'*. Some blocks are recognized by most of the programs using the NEXUS file format, whereas other blocks are private blocks (recognized by only one program) (Lewis, 2024).

```
#NEXUS
begin data;
dimensions ntax=6 nchar=1104;
format datatype=dna;
matrix
D_melanogaster_128up-PA
ATGAGCACAATATTGGAGAAAATCTCGGCCATCGAGTCGGAGATGGCCCGAACCCAAAAGAACAAGGCCACCTCGGCCCATTTGGGTCTACT
GAAGGCGAAGCTGGCTAAGCTGCGACGCGAACTGATTTCCCCCAAAGGAGGCGGCGGCGGAACCGGCGAAGCTGGCTTCGAGGTGGCCAAGA
CTGGAGATGCCCGGGTGGGATTCGTAGGGTTTCCTTCTGTGGGTAAATCCACACTGCTCTCCAACTTGGCTGGCGTTTACTCCGAGGTGGCG
GCATACGAATTCACAACGTTGACCACTGTGCCGGGATGCATTAAGTACAAGGGCGCTAAGATCCAGCTGCTGGACTTGCCCGGTATCATTGA
GGGCGCTAAGGATGGCAAGGGTCGAGGTCGTCAGGTGATTGCTGTCGCTCGCACCTGTAACCTCATTTTCATGGTGCTGGATTGCCTGAAAC
CGCTTGGCCACAAGAAACTCCTCGAGCATGAATTGGAGGGCTTCGGCATCCGGCTTAACAAGAAACCACCAAATATCTACTACAAGCGGAAG
GACAAGGGTGGCATCAATCTGAACAGCATGGTTCCGCAGTCCGAGTTGGACACGGATCTGGTGAAGACCATTCTATCCGAGTACAAGATCCA
CAATGCGGACATCACCCTGAGATACGACGCCACTAGTGACGACCTCATTGACGTTATCGAGGGCAACCGCATCTACATACCCTGCATATATC
TGCTGAACAAGATCGATCAGATCTCCATCGAGGAGCTGGACGTCATCTACAAGATCCCGCATTGCGTGCCCATCTCGGCCCATCACCACTGG
AACTTTGACGATCTGCTGGAGCTGATGTGGGAATACCTGCGACTGCAGCGCATCTACACCAAGCCCAAGGGCCAGCTGCCCGATTACAACTC
GCCCGTGGTACTCCACAACGAGCGCACCAGCATTGAGGATTTCTGCAACAAGCTGCATCGCTCCATTGCCAAGGAATTTAAATATGCGCTGG
TTTGGGGCTCATCTGTGAAGCATCAGCCACAGAAGGTGGGCATCGAACACGTTCTCAACGACGAGGATGTGGTCCAGATTGTGAAGAAGGTT
```

Figure 39 - Provided NEXUS file format.

Sequence alignment and map (SAM), Figure 40, or binary alignment and map (BAM) files are frequently used in bioinformatics to export alignment data for huge numbers of aligned reads. From these files, it is possible to compare gene expression, survey biodiversity, analyze

DNA methylation, or investigate DNA-protein interaction, among many other NGS applications (Zymo Research, 2021).

```
@SQ     SN:chr19_KI270866v1_alt LN:43156
@PG     ID:bwa  PN:bwa  VN:0.7.17-r1188 CL:bwa mem /data/chr19_KI270866v1_alt.fasta
/data/7859_GPI.read1.fq /data/7859_GPI.read2.fq
chr19_KI270866v1_alt_30925_31460_0:3:0_0:4:0_0  99      chr19_KI270866v1_alt    30925   60
        151M    =       31310   536
        CCTGAGGGCGGGCTGAGCTTGCAGGGCCACATGACACTCCCTTGGGTGCCTGCCTGCAACACGCACTGTGGGGTAACCCGAGTC
CCCCTTGCCACGCTCACATAGTTGAGGTTGTGAGTTATTCTCAGTGATGACCTTGCTCTGAACTGCA
        IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII     NM:i:3
        MD:Z:62C39G35T12        MC:Z:151M       AS:i:136        XS:i:0
chr19_KI270866v1_alt_30925_31460_0:3:0_0:4:0_0  147     chr19_KI270866v1_alt    31310   60
        151M    =       30925   -536
        CCTTGTGCACTTGCCTGAGCTCAGCAGTGGTGTGAGGAGGGCTCTGACTGTGAAGCCCAGGGTGCAGGGGTGCAGTGCAGGGTC
TGATGCAGGCACTGTCAATGCTGTGGGCATGGCCGGGCTTACGTGCAGGGCTATCACGCTGTGGCAG
        IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII     NM:i:4
        MD:Z:83G15G36C1A12      MC:Z:151M       AS:i:131        XS:i:0
chr19_KI270866v1_alt_5965_6405_0:5:0_0:7:0_1    83      chr19_KI270866v1_alt    6255    60
        151M    =       5965    -441
        GGATTGATTGAGGCCAGGAGTTCAATACCAGCCTGGGCAACCTAGTGAGACCCCATCTCTTTAAAAAAGCCTATATTTTATTCC
AGAGTGTTCCGTTTTAACGCAAGCACCCAGTTCTCTTGACTAGTTCTGATTCACAGCAGTGCTGGAG
        IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII     NM:i:7
```

Figure 40 - Provided SAM file format.

The assembler test data contains 2 FASTQ files, and these are meant to be used with de novo sequence assemblers that are a type of program that assembles short nucleotide sequences into longer ones without the use of a reference genome (Khan & Pervez, 2018).

Some images required specific data for them to run that did not fit in any of the file types presented above. For these images, the test data was either retained or created if it was lacking.

After gathering a complete set of test data that fulfilled most of the image requirements, it was necessary to run all images so as to obtain a complete compilation of the generated data. For simple Docker images whose commands do not require the alteration of parameters other than the input files or output name, the images where ran normally. For Docker images with a GUI, the images were assessed to ensure they were still functioning. For images with multiple options, the example that was provided on the website was used.

In order to run any Docker image, the following steps were taken:

1. Open the pegi3s website and locate the image intended to be run and go to the specific page (Figure 41).Figure 41

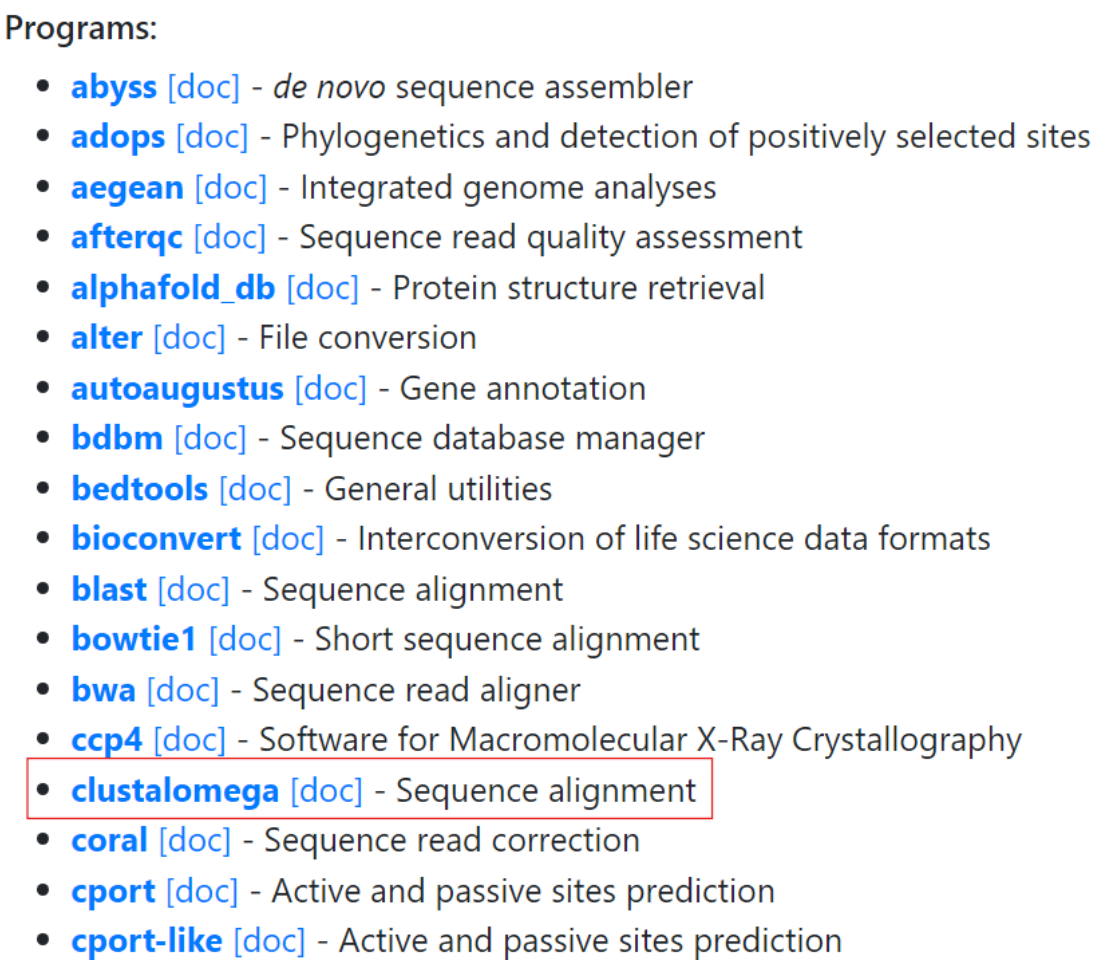


Figure 41 - Locating the image that is intended to be run (pegi3s3, 2024).

2. On the Docker image page, locate the run command and its constraints.

As shown in Figure 42, the clustalomega run command needs to be adapted first. The working directory is the directory on which the Docker image will be running. This directory needs to contain the FASTA file to be analyzed. The "-i" tag specifies the location of the input FASTA file, so the user needs to switch the following text with the actual FASTA file name. The "-o" tag specifies the location and name of the generated output file.

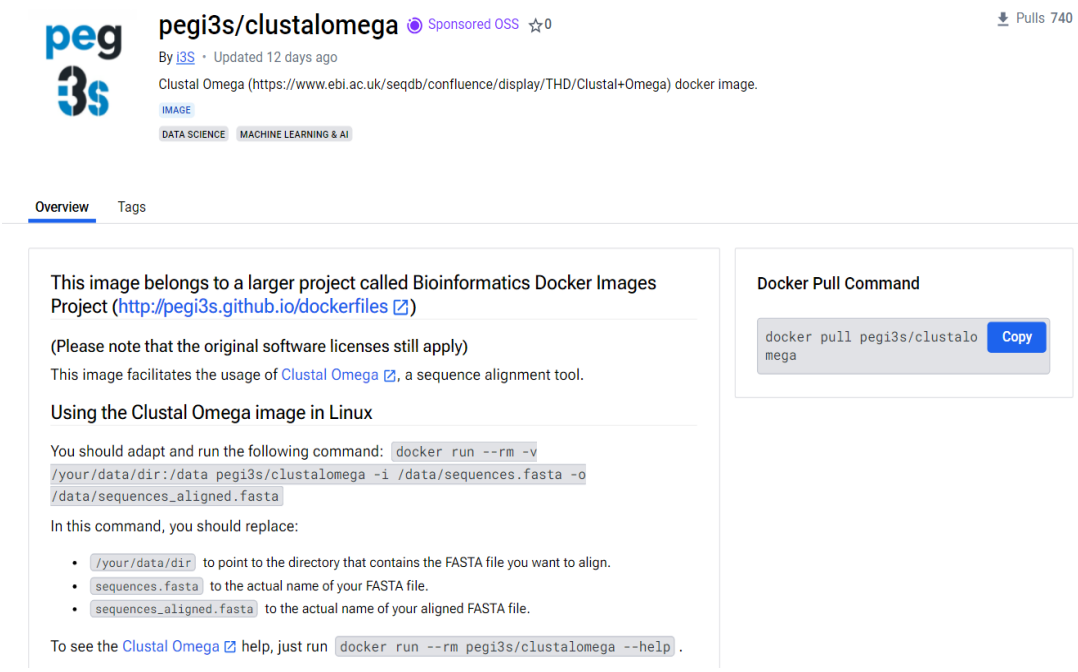


Figure 42 - Pegi3s/clustalomega webpage (pegi3s4, 2024).

3. On the computer select the desired working directory and insert the input file here. The working directory should contain the input file as showed in Figure 43.

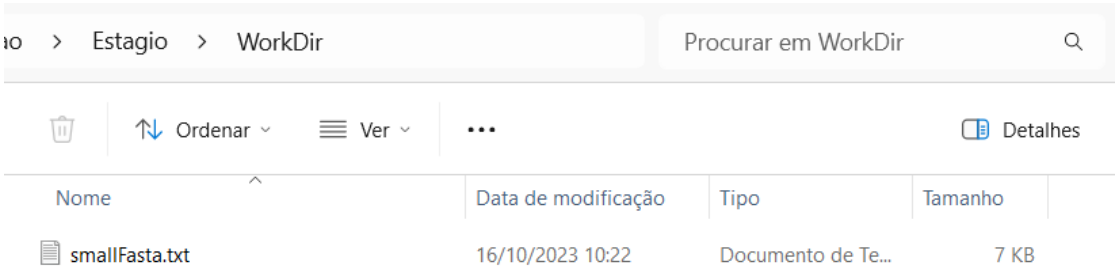


Figure 43 - Working directory containing the required FASTA file.

4. With the Docker Desktop running in the background, open the Windows terminal. When Docker Desktop is running in the background, a user is able to run Docker commands with the Docker daemon (Figure 44).

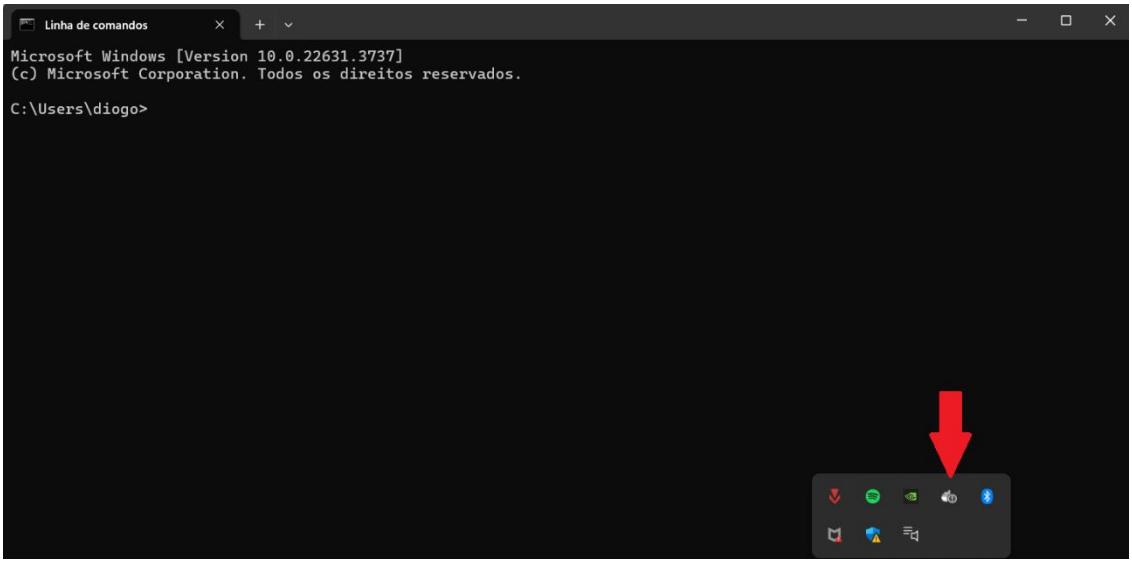


Figure 44 - Windows command line with Docker Desktop running in the background (red arrow).

5. Adapt the command line and run the image in the command line.

In order to adapt the command line, it was necessary to copy the command line shown in Figure 42 and replacing the specified parts with the working directory and input folder depicted in Figure 43. Finally, the adapted command line was run in the terminal (Code 10).

```
docker run --rm -v c/Users/diogo/Estagio/WorkDir:/data
pegi3s/clustalomega -i /data/smallFasta.txt -o
/data/clustalomegaOutput.fasta
```

Code 10 - Clustalomega command line.

The command line above generates an output file called “clustalomegaOutput.fasta”.

6. Check the results.

If the image runs successfully, the output is then stored in a folder called “name_of_image Output” (for clustalomega it would be “clustalomega Output”) alongside the other image outputs. After the output was obtained, the Docker image would be deleted from the computer to avoid cluttering of the computer by using Docker Desktop or Docker Manager.

With the objective of tracking the progress, Table 10 was created.

Table 10 – Abbreviated table to track the creation of a compilation of generated data.

NOME	Test Data Type	Result
abyss	2 fastq	Run
adops	fasta	GUI
afterqc	FASTA	Run
alphafold_db	-	Run
alter	fasta	Run
bdbm	fasta	Run

bioconvert	fasta	Run
bowtie1	genome especifico + fastq	Run
bwa	website	Run
ccp4	fasta	Run
clustalomega	notalligned Fasta	Run
coral	fasta	Run
cutadapt	FASTA	Run
dnasp-v6	-	GUI

GUI images were typically not run since the output depends on the user desire for the most part, so providing results for these groups of Docker images would be unnecessary.

For Docker images that have longer run times, the computers from the lab were used and the output was stored in those. These programs were for example de-novo assemblers and auto-phylo.

By the end of this task, the Ubuntu app for Windows was discovered and it worked as a Linux terminal in Windows that ran in a Windows subsystem for Linux (WSL). Only difference from running these commands was that it was necessary to add “/cmt” before the directory path.

5.4 Development of a graphical interface

The development of the graphical interface was the main focus of this project. The work that was carried out before this task had the objective to further deepen the knowledge on Docker images and to better prepare for the creation of the assistant material like the JSON, the set of test data provided and the results for each Docker images. During the duration of this project, this GUI was called ”Docker App”.

The objective of this GUI is to enable users with no prior knowledge of bioinformatics to easily and intuitively run Docker images to meet their needs. The software can generate the command line with minimal direct interaction from the user. This is made possible by a user-friendly and intuitive interface.

Similar to what was done with the Docker Manager GUI, it first was delineated the functional requirements and non-functional requirements. The functional requirements that the software must meet are shown in Table 11.

Table 11 - Functional requirements table for the Docker App.

ID	Description
FR01	The system should provide the user with an interactable interface.
FR02	The system should provide the user with a link to the pegi3s website.
FR03	The system should present the user with a clickable tutorial video that takes the user to the pegi3s YouTube.

FR04	The system should include a button that allows the user to easily copy the email address for contacting the pegi3s group.
FR05	The system should provide a button that allows the user to open the Docker Manager software, granting access to its full functionalities.
FR06	The system should generate a nested menu according to the ontology from a .dio and .diaf file that needs to fetch from outside the Docker image in the pegi3s GitHub repository.
FR07	The system should provide the user with a button that opens a menu for selecting a Docker image based on an ontology.
FR08	The system should be capable of retrieving data from a JSON file that contains information about all Docker images.
FR09	After selecting an image, the system should update the GUI with the title and a brief description of the selected image. Additionally, the system should display buttons for: • Opening the documentation website; • Visiting the pegi3s website; • Accessing the GitHub repository; • Downloading the test data; • Downloading the generated results; • Opening a secondary window for running the Docker image.
FR10	The system should allow the user to exchange the selected Docker image and update the GUI accordingly.
FR11	The software should obtain data from a config file when opening the secondary window.
FR12	The secondary window should update the title according to the selected image.
FR13	When opening the secondary windows, it should verify whether the directories for user notes, the latest invocations, and executable files, as specified in the config file, already exist.
FR14	If the files previously mentioned do not exist, the system should be able to create them following the instructions in the config file.
FR15	The secondary window should populate the text boxes for the developer notes and run command box with information from the JSON file and user notes from the User Notes folder.
FR16	The software should be able to obtain the list of all Docker image versions and filtrate it according to the selected image.
FR17	The user should be able to select which version of the image to work with in the secondary window.
FR18	The secondary window should have buttons for opening the documentation and pegi3s page of the selected image.
FR19	The text for the button of the input file and number of buttons should update according to the necessities of the selected image.
FR20	The secondary window should have an input field that can be filled by pressing the select file button.
FR21	The software should update the run command and consequently the run command box with the desired input.
FR22	The secondary window should have a field to write the output name of the generated file (if the required image has one).

FR23	Pressing the push button in the secondary window should update the run command and consequently the run command box with the desired output name
FR24	The secondary window should have a section for the developer notes that updates with information from the JSON file about: • Which versions are useful • Versions with bugs • Versions that no longer work • Versions that are no longer tested • When the recommended version was last tested on • Whether the image works with Podman or Singularity
FR25	The secondary window should have a field for users to write their own personal notes
FR26	Personal notes should be stored in users' computers, so they are able to be obtained even after rerunning the Docker image.
FR27	Above the run command box there should be a tooltip that contains info about the parameters in the command line.
FR28	The secondary window should contain a run command box that displays only the parameters of the command line for running a Docker image.
FR29	If the image does not have parameters, the run command box should inform the user and be non-interactable.
FR30	The image should have a button to call for the test data invocation that updates the run box with the test data specific parameters.
FR31	If the image does not require test data, it should warn the user.
FR32	It should have a button to view the latest invocations of the selected image and update the run command box with them.
FR33	The user should be able to interact with the run Command box.
FR34	There should be a run button that when pressed creates and runs the command line by joining the command line basis and the parameters displayed in the run box.
FR35	When running a Docker image, the secondary window layout should fully change and provide the user with non-interactable text box that shows the command line of the running process.
FR36	After the image fully runs, it should display the run time and a button to go back to the previous layout.
FR37	When running a Docker image, the invocation should be stored in a folder called "LastestInvocations" with the time of invocation in the title of the file outside the container.
FR38	The user should be able to create an executable file which is stored in the executable files folder outside the container.
FR39	The executable files created should be single clickable.
FR40	The system should support multiple secondary windows running at the same time.

Table 12 presents the non-functional requirements of the developed GUI, detailing specifications for how the system should operate and behave beyond its core features to ensure a positive user experience and optimal system performance.

Table 12 - Non-functional requirements table for the Docker App.

ID	Description
NFR01	The system should be scalable
NFR02	The system should be able to handle a big volume of data
NFR03	The system should always be available and functioning correctly
NFR04	The system should run regardless of the operating system and machine
NFR05	The system should be of low complexity and intuitive
NFR06	The system should be available as a Docker image
NFR07	Easily show the user the different available options to achieve a given goal by classifying the different Docker images
NFR08	Easily show the user the kind of input files that are needed to run the Docker image
NFR09	Easily show the user what kind of output is obtained when using a given Docker Image
NFR10	Easily show the user how to run the Docker Image using the provided test data
NFR11	The system should promote good research practices by: 1) Providing easy access to documentation on the Docker image and the program implemented in it (information on DockerHub, links to the manual software; user notes; past invocation (likely preferred) commands); 2) Saving the run file so that it can be checked later on; 3) Not letting the user choose the "latest" version but rather the corresponding tagged version as well as all other tagged versions; 4) By warning the user of bugs, and no longer tested (unrecommended) images
NFR12	Making the whole experience as painless as possible in order to promote the use of different approaches to address the same problem. Indeed, if it is easy, why not run different software applications to guarantee robustness?

Before starting to work on creating the GUI, a mockup of the secondary window was designed. This representation can be viewed in Figure 45.

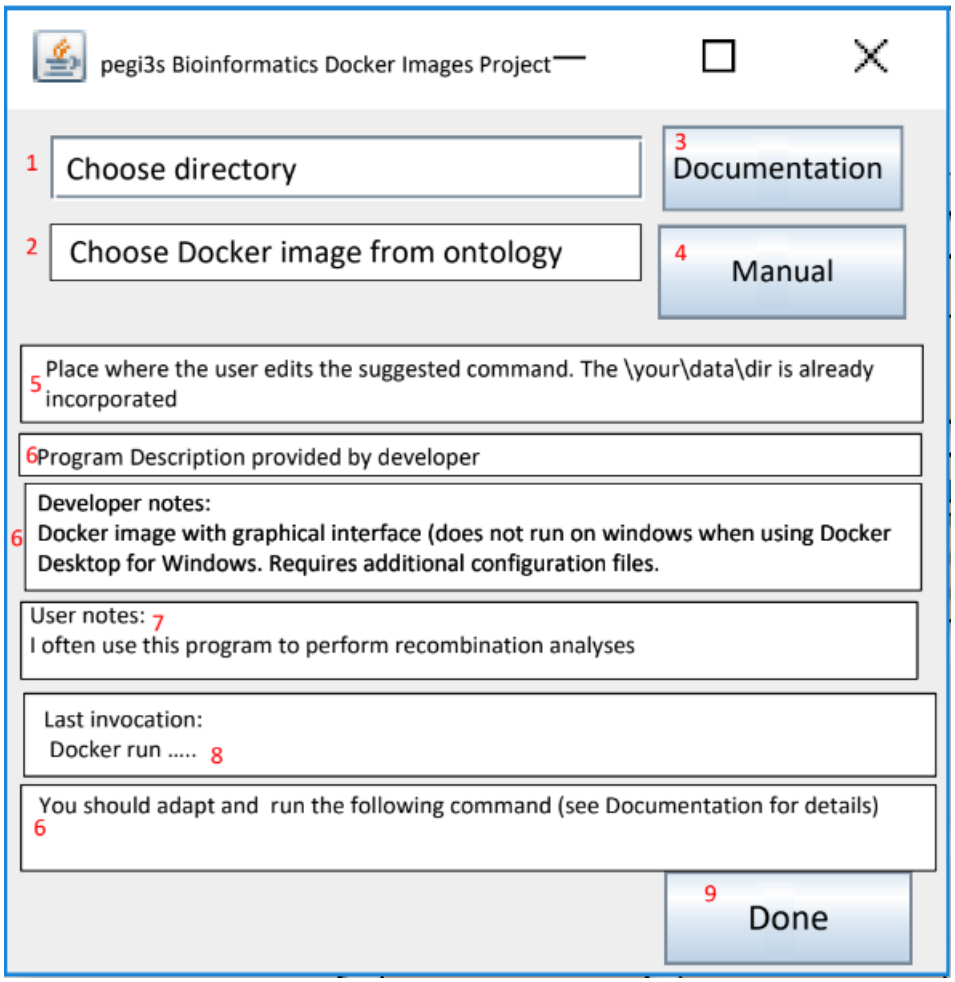


Figure 45 – Docker App secondary window mockup.

Since then, the secondary window has undergone a lot of changes based on improvements suggested by users, and the current layout is completely different from the first planned layout. The current secondary window presents a lot more options for the user and some text boxes were later joined to avoid redundancy. These changes will be later explained when analyzing the current GUI.

The Docker app is distributed as a Docker image. The Dockerfile creates a Docker image based on pegi3s/docker, installs necessary packages and dependencies for Python development and graphical applications, sets up environment variables, copies required files into the container, sets the working directory, and specifies the main entry point for the containerized application. Similar to what happened with the Dockerfile for the Docker App. Only difference being that in order for the hyperlinks to work, it is necessary to install Firefox in the Docker image, so the interactive prompts had to be disabled with the assistance of "ENV DEBIAN_FRONTEND=noninteractive".

After creating the Docker file, the Ubuntu app for windows was used to build the Docker image. Th Docker build command was used (Code 11) inside the directory containing the Dockerfile.

```
sudo docker build ./ -t pegi3s/docker-app
```

Code 11 - Command to build the Docker app.

Finally, to run the Docker app, the Code 12 is run inside the Ubuntu app.

```
docker run --rm -ti -e USERID=$UID -e USER=$USER -e DISPLAY=$DISPLAY -v
/var/db:/var/db:Z -v /tmp/.X11-unix:/tmp/.X11-unix -v
$HOME/.Xauthority:/home/developer/.Xauthority -v
/var/run/docker.sock:/var/run/docker.sock -v /tmp:/tmp -v
/mnt/c/Users/diogo/Estagio/DockerApp:/data pegi3s/docker-app
```

Code 12 - Command to run the Docker App with a specified working directory.

The Docker app runs as a Docker image and when opening the GUI, the user is presented with the menu in Figure 46.

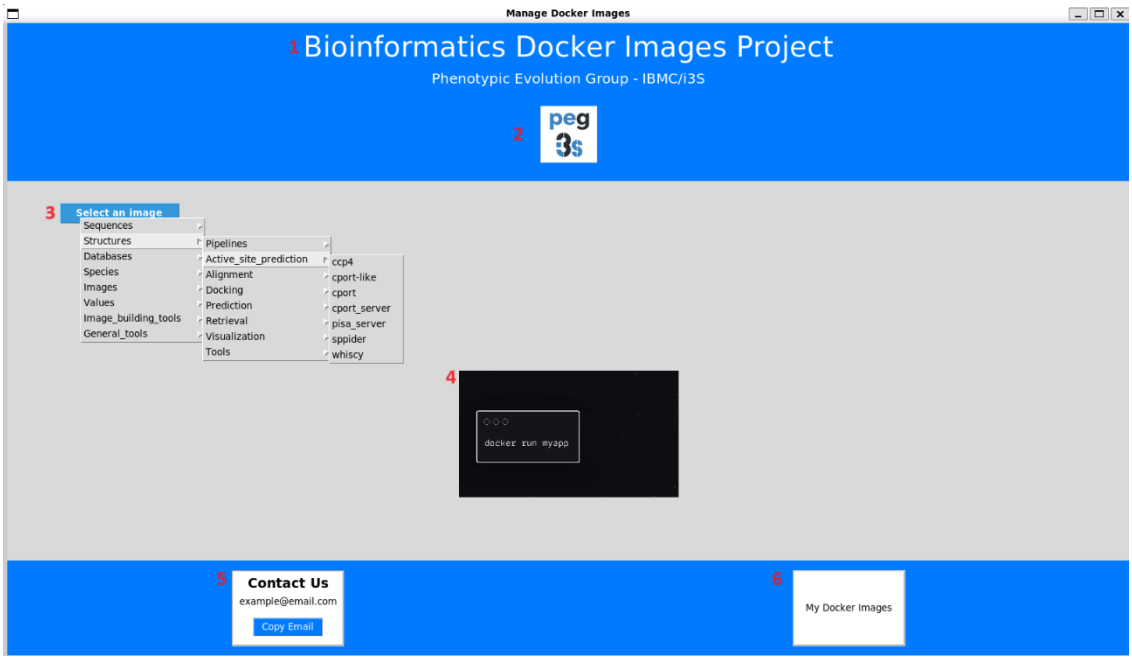


Figure 46 - Opening menu of the developed GUI with numbered features.

The different features are numbered to help explain the capabilities of the GUI. When opening the GUI, the user is presented with the title of the Docker project and the name of the group below it in a subtitle (1). Below them, there is the pegi3s logo (2), which also functions as a hyperlink to the pegi3s website.

The GUI has a button that opens a context that allows the user to select an image according to the ontology of the Docker images (3). This ontology is obtained from the pegi3s and is split up in 2 files, a .obo file (Figure 47) and a .diaf file (Figure 49).

```
[Term]
id: DIO:0000001
name: Sequences
def: "Any program that accepts as input nucleotide, protein or short read sequences"

[Term]
id: DIO:0000002
name: DNA
def: "Any program that accepts DNA sequences as input"
is_a: DIO:0000001

[Term]
id: DIO:0000003
name: Pipelines
def: "Pipelines that use nucleotide sequence files as input"
is_a: DIO:0000002

[Term]
id: DIO:0000004
name: Alignment
def: "Any program that allows the alignment of DNA sequences"
is_a: DIO:0000002
```

Figure 47 - Pegi3s .obo file structure.

The .obo file gives the ontology structure, and it follow an Resource Description Framework (RDF) standard that is human readable ((OBO technical WG, 2024)). It is composed by an id, a name and a definition for the top level. All other categories have a “is_a” tag. This tag represents in what broader category the item belongs. For example, in the image above, DNA is directly under Sequences and Pipelines and Alignment are both categories belonging to DNA (Figure 48).

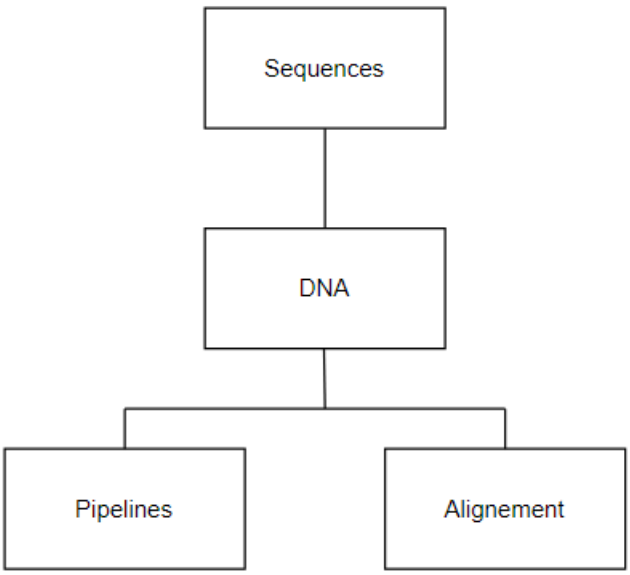


Figure 48 - Example of the structured abbreviated obo file.

It should be noted that these are only the categories where the images can be placed. In order to populate the different categories, the .diaf file (Figure 49) contains all the images and the category they belong id they belong to.

```
DIO:0000004     tcoffee
DIO:0000004     translatorx
DIO:0000005     hyphy
DIO:0000005     omegamap
DIO:0000005     paml
DIO:0000005     pss-3d
DIO:0000006     dnasp-v6
DIO:0000006     kakscalculator
DIO:0000007     aegean
DIO:0000007     cga
DIO:0000007     metaeuk
DIO:0000007     prosplign-procompart
```

Figure 49 - Contents of the .diaf file.

The info from the .diaf would allow to populate the example from Figure 48 like so (Figure 50).

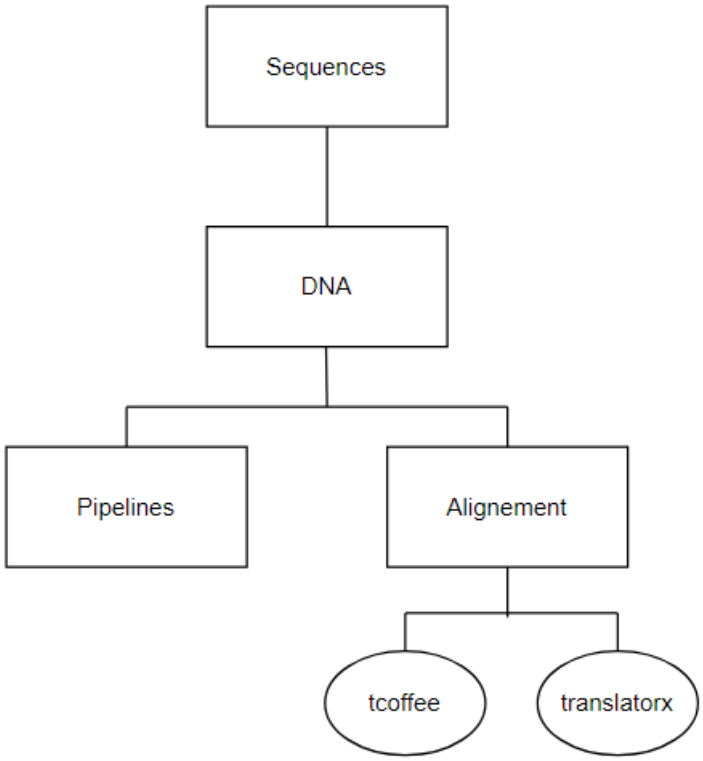


Figure 50 – Example of the structured abbreviated obo file with images from the diaf file.

The example shown above is only an abbreviation of the actual ontology, since the actual ontology contains over 100 images and a great deal of categories with up to 4 degrees of

hierarchy. An image can also belong to multiple categories at the same time. For example, diamond belongs to both the “Sequence_homology_search” category inside DNA and also the “Sequence_homology_search” belonging to Protein.

In order to create the context menu (3) of the Figure 46, the categories in the. obo are first correctly sorted and afterwards the images from the .diaf file are placed on their corresponding categories.

Following the layout in Figure 46, there is a video (4) playing that shows a tutorial on how to use Docker. Clicking on this video takes the user to the pegi3s YouTube page. On the bottom there are 2 buttons, one that highlights and allows the user to copy the email to contact pegi3s. The other button opens Docker Manager, the discussed software in chapter 5.1.1.

After the user selects an image, the GUI interface updates accordingly (Figure 51).

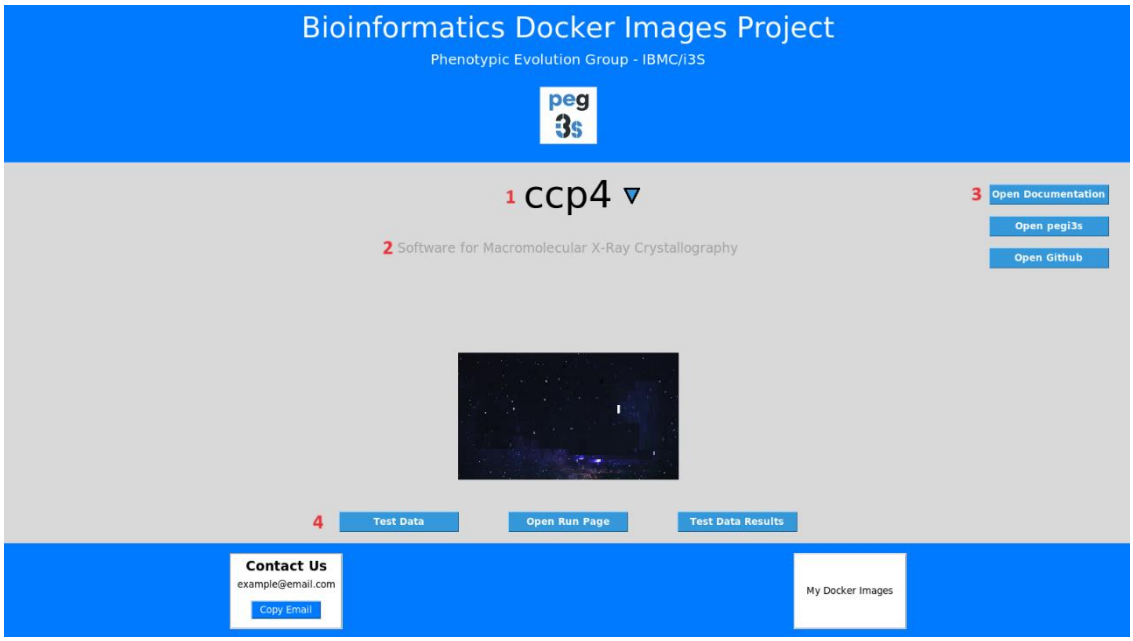


Figure 51 - Docker App interface after selecting an image.

After selecting an image, the GUI displays the image title with a downward triangle beside it (1). This image title and triangle are interactable and allow the user to select another image from the context menu previously displayed.

Below the title there is a description of the Docker image (2). The description is obtained from the JSON file containing all sorts of information on the Docker image. The different categories in the JSON file came from analyzing the Docker images and gathering what would be the most useful information to incorporate in the GUI.

The JSON file is accessed with the help of the “json” library in Python. After obtaining the manifest of the JSON file from the pegi3s GitHub, the information is decoded and loaded to a

variable in the code. The information regarding a specific image is loaded with the help of Code 13.

```
    image_names = [image["name"] for image in imagens_docker]  # Lista de
todos os nomes de imagens no JSON
        if image_name in image_names:
            image_data = next(image for image in imagens_docker if
image["name"] == image_name)
        else:
            messagebox.showwarning("Warning", "Selected image not found in
JSON.")
```

Code 13 - Code to load the data on a specific image.

After fetching the data on a specific data, the information can be obtained with Code 14.

```
    text = image_data["description"]
```

Code 14 - Code to obtain the information about the description.

This method is used whenever there is a need to obtain information stored in the JSON file. For the image displayed in Figure 51, ccp4, the excerpt of the JSON file is shown in Figure 52.


```
{
    "name": "ccp4",
    "description": "Software for Macromolecular X-Ray Crystallography",
    "status": "Usable",
    "recommended": "7.0.1",
    "latest": "7.0.1",
    "useful": [""],
    "bug_found": [""],
    "not_working": [""],
    "recommended_last_tested": "",
    "no_longer_tested": ["7.0"],
    "pegi3s_url": "https://hub.docker.com/r/pegi3s/ccp4/",
    "manual_url": "https://www.ccp4.ac.uk/documentation-2/",
    "github_url": "https://github.com/pegi3s/dockerfiles/tree/master/ccp4",
    "comments": [""],
    "gui": false,
    "podman": "untested",
    "singularity": "untested",
    "input_data_type": ["fasta"],
    "invocation_general": "docker run --rm -v /your/data/dir:/data pegi3s/ccp4 bash -c ",
    "invocation_general_comments": ["'/your/data/dir' to point to the directory that contains the input folder with all protein structures you want to analyze.","Additionally, this will be
the same directory where a folder with the results will be created."],
    "usual_invocation_specific": "\"/ccp4/bin/run data/fastaFolder data/outputFolder\"",
    "usual_invocation_specific_comments": ["'inputFolder' to point to the folder that contains the protein structures to analyze.","'outputFolder' to point to the folder where sub-folders
with the results for each submitted structure will be saved, as well as the configuration file for PISA."],
    "test_invocation_specific": "",
    "test_data_url": "",
    "test_results_url": "",
    "icon":""
  },
```


Figure 52 - ccp4 section of the JSON file.

The JSON variables are explained in Table 13. It should be noted that during the development of this project, a new pegi3s website was being developed that shared the same JSON file, so the “icon” information is irrelevant to this GUI.

Table 13 - Contents of the JSON file.

Variable	Description
Name	Image name
Description	Image Description
Status	Whether the image runs or not
Recommended	The recommended versions
Latest	The latest version
Useful	The useful versions
Bug_found	Versions with bugs

Not_working	Versions that no longer work
Recommended_last_tested	The last time the recommended version was tested
No_longer_tested	Versions that are no longer tested
pegi3s_url	The URL for the pegi3s website
manual_url	The URL for the manual website
github_url	The URL for the GitHub repository
comments	Developer comments
gui	Whether it is a GUI or not
podman	If it runs in Podman
singularity	If it runs in Singularity
input_data_type	Type of input data
invocation_general	Immutable part of the Docker run command
invocation_general_comments	Comments on the invocation general command
usual_invocation_specific	Interchangeable part of the Docker run command
usual_invocation_specific_comments	Comments on the interchangeable part of the Docker run command
test_invocation_specific	Interchangeable part of the Docker run command used to run test data
test_data_url	The URL for the test data file
test_results_url	The URL for the results of the run with the test data
icon	Icon of the Docker image (for the new pegi3s website)

Following the layout in Figure 1, the GUI has three buttons on the right (3). Clicking these buttons will take the user to the documentation website, the pegi3s website, and the GitHub repository. The hyperlinks for these websites are obtained from the JSON file.

Finally, three more buttons are placed below (4). They are arranged this way based on the principle that the user must first obtain the test data to run it. After running the test data, the user can then compare the results with those from a previous trial of the same test to check if it assures the replicability. The data for the test data and the results were gathered during the task described in chapter 5.3 (“Creation of a standardized set of test data and compilation of the generated data”). These files are stored in a local server (the files are too big to be stored in GitHub), and the link to obtain the data is present in the JSON file.

After selecting the Docker image, the user can then open the window to run the Docker image (Figure 53).

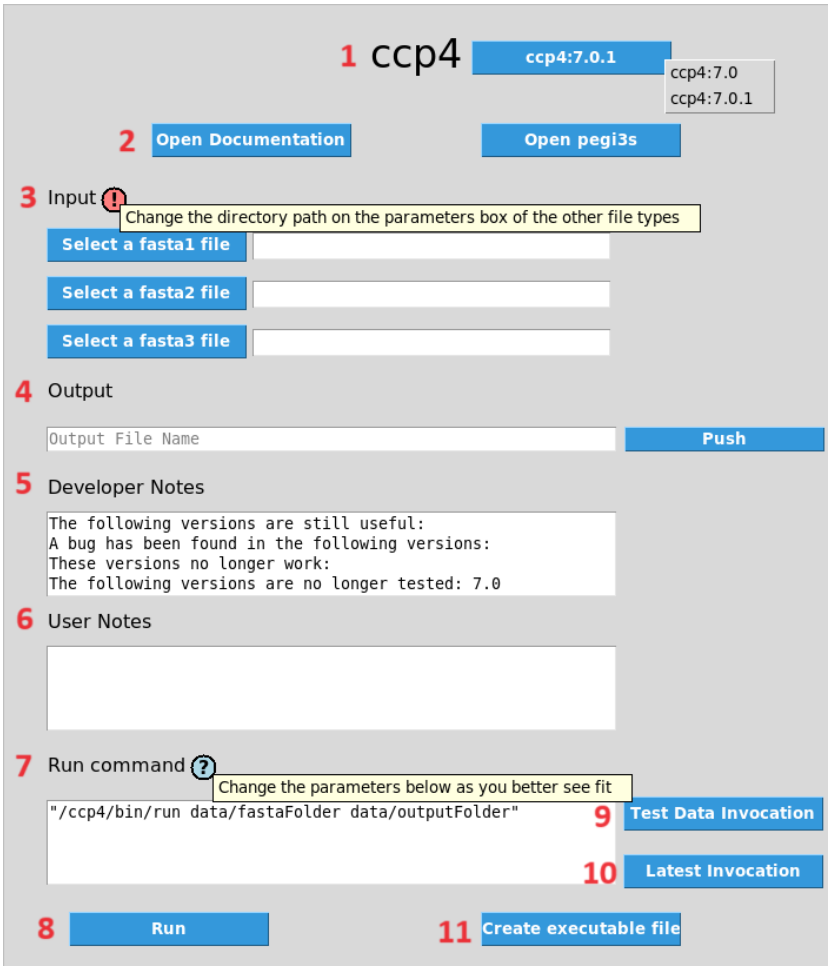


Figure 53 - Window to run the Docker image.

The user is presented with the selected image in the title and beside it a button that opens a context menu that contains all versions of the Docker image (1). These versions are obtained by pulling a text file from the pegi3s GitHub repository, which contains all versions of all Docker images. This file is updated nightly. Although the same result could be achieved by running a specific command that lists this information, it was not desirable because the method took too long.

To assure the reproducibility in the research, the latest option is suppressed in the context menu. This is because when selecting the latest option, the user does not know what version its calling. And the latest version always matches another version as it is shown in Figure 54.

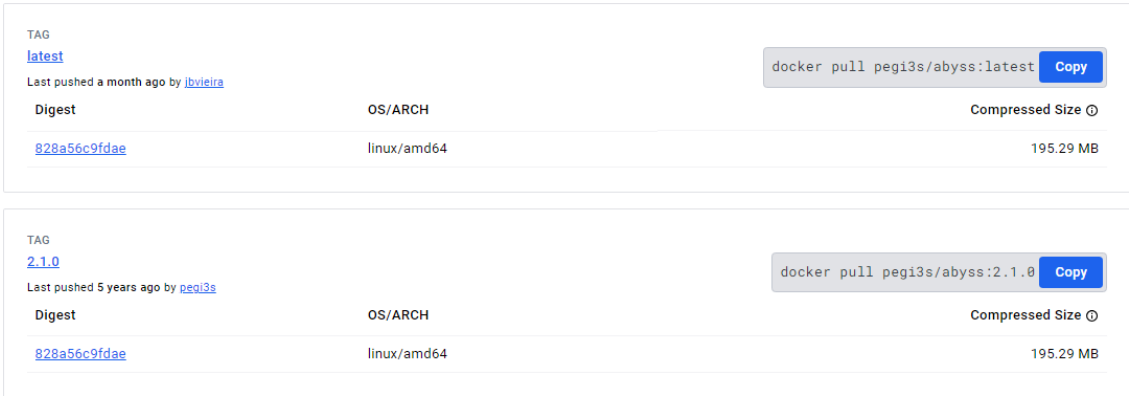


Figure 54 - Abyss latest version and version 2.1.0.

The digest of a Docker image is a unique immutable identifier for a container image to deploy. And on the image above both digests are the same meaning they are the same Docker image, only difference being the name.

Afterward, the buttons for the pegi3s website and the documentation from the main window are reused. This is because multiple windows of different Docker images can be run at the same time, and this ensures the user always has the relevant information at hand.

In the input section (3), the window is populated based on the number of input files specified in the JSON file. Figure 53 is adapted for scenarios with more than four input types. Although this situation does not occur with any current image in the pegi3s repository, the software is prepared for such a case. If this happens, a red exclamation mark will appear warning the user that input files beyond the three shown need to be manually replaced (currently, no image requires 4 or more input files). The user can select a file by clicking on "Select a xxxxx file". This action opens a window where the user can choose the file they want to use with the Docker image. Upon selecting the file, the software automatically updates the run command box.

In the output (4) section the user can type the output name; by pressing push, it will automatically update the run command box.

The developer notes box (5) is non-interactive and compiles various typers of information about the Docker image from the JSON file. It presents information on what versions are currently operable, which versions have bugs, the versions that no longer work, the versions that are no longer tested, when the recommended version was last tested on, whether the image runs or not in Podman or Singularity and also comments from the developer (if they exist).

The user notes text box (6) is intended for users to provide comments they find relevant about the Docker image. Since Docker apps run as containers, storing user notes for the Docker images requires this information to be saved outside the container, specifically in the working directory specified in the Docker command. Additionally, within the working directory, there must be a config file (Figure 55).

```
dir=/mnt/c/Users/diogo/Estagio/DockerApp
past_invocations=/Manage_docs
user_notes=/Manage_docs
executable_files=/Manage_docs
```

Figure 55 - Config file contents.

The "dir" specifies the directory path from which the Docker images will be executed. The software checks for the existence of three folders — "past_invocations", "user_notes", and "executable_files" — within the specified directory in the config file. If these folders exist there, it stores the corresponding information inside them. If not, it creates these folders inside the specified directory (from the example above, it would be "workdir/Manage_docs").

When storing user notes, the software creates a .txt file named after the Docker image. For instance, user notes for "ccp4" would be stored as "ccp4.txt".

Above the run command box (7), the user can access a tooltip specific to the image. This information is stored in the JSON file. Additionally, the run command box is populated with "usual_invocation_specific" from the JSON file, which represents the part of the Docker image that the user can modify to tailor the run command according to their requirements. This section usually contains the input files names or output name of the results. It can also include specific parameters that are unique for the Docker image.

When the user runs a Docker image (8), the system uses the basic run command provided in the JSON file and updates its working directory to match the one specified in the config file. It also fetches the specified version of the Docker image in the context menu next to the title. Next, it combines this updated run command basis with the content in the Run command box to form a complete Docker run command (represented in Figure 56).

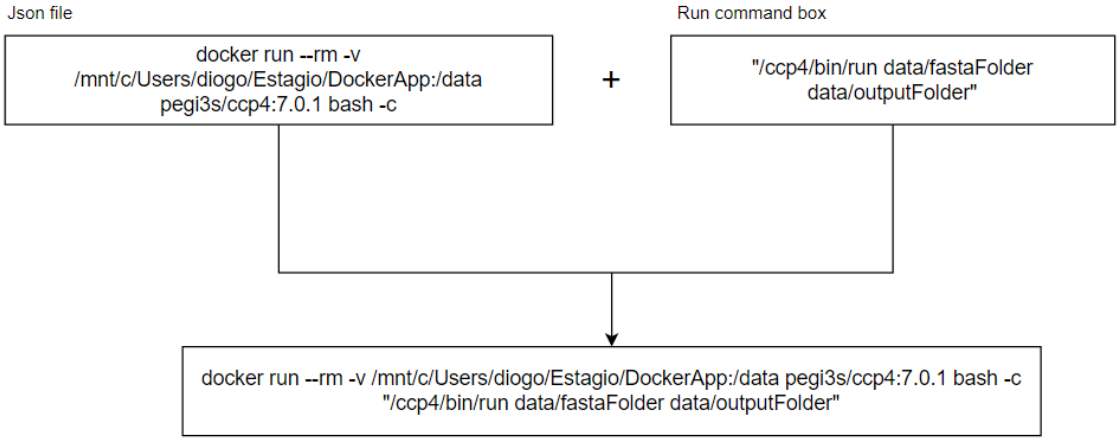


Figure 56 - Diagram showcasing the creation of the full run command.

When running the image, the secondary window interface is updated (Figure 57).

Figure 57 - Layout of the secondary window when running the Seda Docker image.

On the figure above, the output of the console is displayed in a text box (8.1). After the image runs, its runtime is displayed (8.2). By pressing the okay button, the window updates to the menu in Figure 53.

After running a Docker image with the Docker app, the run command is stored in a .txt file inside a folder with the image name inside the Past invocation folder (Figure 58).

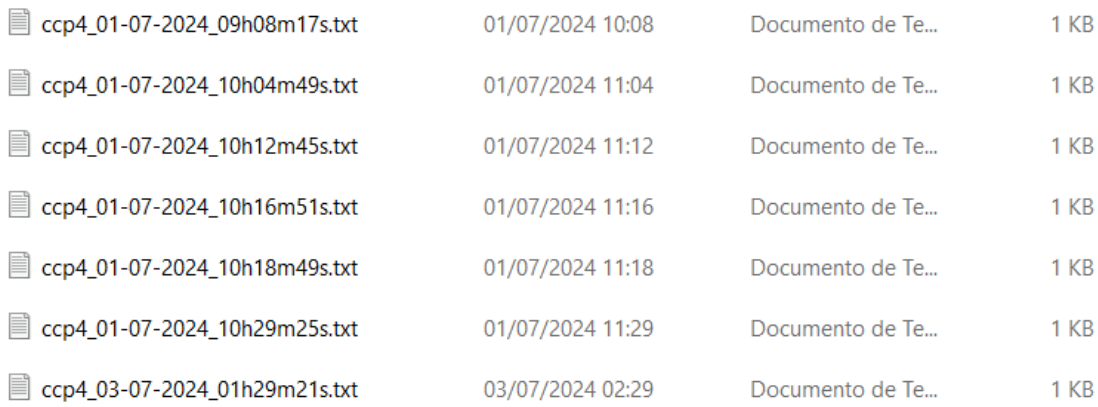


Figure 58 - Latest Invocations for the ccp4 Docker image.

The name of the files has the image name, the date and the time that the Docker run command was invoked as show in the image above.

The “Test Data Invocation” button (9) updates the run command box with the command interchangeable part of the Docker run command used when testing the image.

The "Latest Invocations" button (10) opens a new window displaying all previous invocations of the selected Docker image. Users can select any of the existing stored invocations, and the run command box will update with the parameters used in that specific invocation of the image. This is useful in order to ensure the repeatability of the obtained results and since these files are stored in a file outside of Docker, a researcher can share the commands used and other researchers can replicate the same results.

Finally, a user can click the “Create an Executable File button (11). By doing so, this will create a .sh file in the “Executable Files” folder containing the Docker run command. A .sh file is an executable file for Linux. The user can then run this .sh file outside of the container. This is especially useful for the researcher to set up the experiment in advance, saving time since he does not have to wait to obtain the results before creating the Docker run command.

Finally, after completing the GUI, it was selected a name for the distribution. The name selected for the distribution was “DocNRun-pegi3s”.

6. Conclusion

This chapter presents a general summary of all the work carried out, highlights the main results obtained, and states whether the objectives were achieved or not. Furthermore, possible future works are mentioned aiming to explore new avenues that can build upon the current research, thereby contributing to the ongoing development and refinement of the field.

The old VM has been discontinued and replaced by the newly developed one. Although better solutions (Docker Desktop and Ubuntu app) have been found for running Docker images on a user's computer, which allocate resources more efficiently, the developed image remains useful for people with MacOS computers, as the new solutions only works for computers with Windows.

Due to the new changes in the method of running the Docker images, a new application for managing them was developed, the Docker Manager. This software is currently being distributed in the pegi3s website as a Docker image (https://hub.docker.com/r/pegi3s/docker-manager/).

Reviewing all Docker images provided a better understanding of them and significantly helped determine which features would be most relevant to add to the developed GUI. This review also guided the creation of the JSON file by identifying the most relevant information to include.

Creating a standardized set of test data reduced the variety of test data provided, making it easier to offer users specific test data for each image. This standardization also simplified the process by allowing the use of existing test data rather than obtaining new test data each time an image is added to the pegi3s repositories. However, a few images in the repository still do not have assigned test data because they were either added after this task was completed or the attributed data set was not the most suitable. With the assigned test data, results were collected for most of the Docker images. These results currently sit in a local server.

Finally, the GUI was developed. The software met all of the predefined Functional and non-functional requirements. Probably the most important feature of the created application is that it is prepared to handle new images being added to the pegi3s repositories without the necessity of changing the Docker image itself since it gathers all its data from outside the Docker image. This feature means that this image is able to remain the same regardless of the amount of Docker images in the pegi3s repository. Updating the JSON file and .dio and .diaf files is enough to keep the software up to date. This software will also be distributed in the pegi3s website as a Docker image. The comparison between DocNRun with similar related works, can be viewed in Table 14.

Table 14 - Comparison between DocNRun and related works.

SIM	DockStation	Docker Desktop	Portainer.io	GUI Developed
Independence	Yes	Yes	No	No
Runs Images	Yes	Yes	Yes	Yes
Image management	Yes	Yes	Yes	Yes

Customization of the GUI	Yes	Yes	Little	No
Runs in multiple os	Yes	Yes	Yes	Yes
Added security	Yes	Yes	Yes	No
Made for non-bioinformatics	No	No	No	Yes

The developed GUI lacks in some features that other software of similar nature presents the user. The main advantage of the developed GUI is that is strictly developed to assist people with no background in Bioinformatics with using Docker, making it unique in a sense that it's primarily adapted to this very directive, distancing itself from the other software that exist since they are created for being used by developers or people with knowledge in the area. Currently, there are no applications developed specifically to facilitate Docker utilization by individuals outside this field who may need to work with Docker images.

The main difficulties where found in creating the ontology, especially finding the best way to code in structure of the menu with de .dio and .diaf files.

Following software development, there are critical stages such as testing, operations, and continuous improvement. These phases are vital components of the software lifecycle. While it is not feasible to address them fully within the brief timeframe of the current project, their significance is recognized by the author.

For continuous improvement, the DocNRun app could be expanded by incorporating tutorial inside the main menu of the app. These videos would be obtained from the pegi3s YouTube and organized in the section where the current video sits. The user would be able to cycle through the videos as he wished. Currently the video is incorporated within the Docker image, which means that to change the video, the Docker image would have to change, which is not desirable.

Finaly, the DocNRun starting menu was made in the image of the old pegi3s website. However, the website will be updated to have a more modern look, so another future work would be to update the GUI to look more like the new website.

7. References

Amazon. (2024, 05 16). *Amazon*. Retrieved from What is Docker?: https://aws.amazon.com/docker/

Apostal, S. F., Apostal, D., & Marsh, R. (2018). Containers and Reproducibility in Scientific Research .

Bryan, B. (2016, 8 14). *B3N*. Retrieved from RHEL/CentOS, Debian, Fedora, Ubuntu & FreeBSD Comparison: https://b3n.org/centos-debian-fedora-ubuntu-freebsd/

Chamber, K., Collings, A., & Graf, C. (2019). Towards minimum reporting standards for life scientists.

Cutting, V., & Stephen, N. (2021, 8). A Review on using Python as a Preferred Programming Language for Beginners. p. 1.

Daniel Nüst, V. S. (2020, 11 10). Ten simple rules for writing Dockerfiles for reproducible data science.

Docker1. (2024, 6 22). *Use containers to Build, Share and Run your applications*. Retrieved from docker: https://www.docker.com/resources/what-container/

Docker2. (2024). *docker.docs*. Retrieved from Overview of Docker Desktop: https://docs.docker.com/desktop/#:~:text=It%20provides%20a%20straightforward%20GUI,can%20focus%20on%20writing%20code.

Docker3. (2024). *ubuntu*. Retrieved from Docker Hub: https://hub.docker.com/_/ubuntu

Docker4. (2024, 05 16). *Docker*. Retrieved from Use containers to Build, Share and Run your applications: https://www.docker.com/resources/what-container/

Docker5. (2024, 6 6). *Install Docker Engine on Ubuntu*. Retrieved from Docker: https://docs.docker.com/engine/install/ubuntu/#prerequisites

Docker6. (2024, 6 6). *Linux post-installation steps for Docker Engine*. Retrieved from docker.docs: https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user

Docksal. (2024, 5 27). *Docker Web UI: Portainer.* Retrieved from Docksal: https://docs.docksal.io/use-cases/portainer/#:~:text=Portainer%20is%20a%20lightweight%20management,as%20it%20is%20to%20use.

DockStation. (2017). DockStation and why we created it.

Elixir. (2024, 5 21). *Elixir*. Retrieved from https://elixir-europe.org/services

GeeksForGeeks. (2023, 4 21). *Geeks For Geeks*. Retrieved from Kernel in Operating System: https://www.geeksforgeeks.org/kernel-in-operating-system/

i3s. (2024, 05 21). *i3s*. Retrieved from Sobre o i3s: https://odisseiagenetica.i3s.up.pt/sobrei3s/

IBM. (2024, 06 08). *xhost command*. Retrieved from IBM: https://www.ibm.com/docs/en/aix/7.2?topic=x-xhost-command

Ioannidis, J. P., Allison, D., Ball, C., & Coulibaly, I. (2009). Repeatability of published microarray gene expression analyses. *Nature Genetics*.

JFrog. (2022, 6 16). *Why Use Ubuntu as a Docker Base Image When Alpine Exists?* Retrieved from JFrog: https://jfrog.com/devops-tools/article/why-use-ubuntu-as-a-docker-base-image-when-alpine-exists/

Kazlouski, I., & Smith, S. (2024, 6 30). *Dockstation*. Retrieved from GitHub: https://github.com/DockStation/dockstation

Khan, A., & Pervez, M. (2018). A Comprehensive Study of De Novo Genome Assemblers: Current Challenges and Future Prospective.

Knuth, D. E. (1992). Literate Programming.

Kuhlman, D. (2012). Part 1 - Beginning Python. In D. Kuhlman, *A Python Book: Beginning Python, Advanced Python, and Python Exercises* (p. 1).

Lewis, P. O. (2024, 6 19). *The NEXUS file format*. Retrieved from https://plewis.github.io/nexus/

López-Fernández, H., Ferreira, P., Reboiro-Jato, M., Vieira, C., & Vieira, J. (2022). The pegi3s Bioinformatics Docker Images Project. *Prartical Applications of Computational Biology & Bioinformatics, 15th International Conference (PACBB 2021)*.

Maryam Zaringhalam, L. F. (2020). Data and Code for Reproducible Research: Lessons Learned from the NLM Reproducibility Workshop. *Office of Strategic Initiatives*.

Mega. (2024, 6 19). *Newick Format*. Retrieved from https://www.megasoftware.net/mega4/WebHelp/glossary/rh_newick_format.htm#:~:text=NEWICK%20is%20a%20simple%20format,needed%20to%20end%20the%20tree).

Microsoft. (2024, 5 21). *Azure*. Retrieved from What is a virtual machine (VM)?: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-virtual-machine#:~:text=A%20virtual%20machine%20is%20a,on%20many%20people's%20work%20computers.

Naor, A. (2021, 2 19). *freeCodeCamp.* Retrieved from What Is an ISO File?: https://www.freecodecamp.org/news/what-is-an-iso-file-explained-in-plain-english/

NGS Analysis1. (2024, 06 19). *FastQ Format*. Retrieved from NGS Analysis: https://learn.gencore.bio.nyu.edu/ngs-file-formats/fastq-format/

NGS Analysis2. (2024, 6 19). *GFF3 Format*. Retrieved from NGS Analysis: https://learn.gencore.bio.nyu.edu/ngs-file-formats/gff3-format/#:~:text=General%20Feature%20Format%20(GFF)%20is,be%20handled%20by%20this%20format.

Noviantika. (2024, 2 22). *What Is Ubuntu? A Quick Beginner's Guide*. Retrieved from Hostinger Tutorials: https://www.hostinger.in/tutorials/what-is-ubuntu#:~:text=Ubuntu%20is%20a%20popular%20free,which%20was%20difficult%20to%20install.

Open Shift. (2024, 5 21). *About EFI mode for virtual machines.* Retrieved from RedHat Openshift: https://docs.openshift.com/container-platform/4.9/virt/virtual_machines/advanced_vm_management/virt-efi-mode-for-vms.html#:~:text=(EFI)%20mode.-,About%20EFI%20mode%20for%20virtual%20machines,BIOS%2C%20enabling%20faster%20boot%20times.

Oracle. (2023). *VirtualBox*. Retrieved from About VirtualBox: https://www.virtualbox.org/wiki/VirtualBox

OSBoxes. (2024, 06 05). *OSBoxes*. Retrieved from Ubuntu: https://www.osboxes.org/ubuntu/

Panko, H. (1996). Spreadsheets on trial: a survey of research on spreadsheet risks.

pegi3s1. (2024, 6 8). *pegi3s/docker-manager*. Retrieved from pegi3s: https://hub.docker.com/r/pegi3s/docker-manager/

pegi3s2. (2024, 7 10). *pegi3s/fastqc*. Retrieved from pegi3s: https://hub.docker.com/r/pegi3s/fastqc/

pegi3s3. (2024, 7 10). *Bioinformatics Docker Images Project*. Retrieved from pegi3s: https://pegi3s.github.io/dockerfiles/

pegi3s4. (2024, 7 10). *pegi3s/clustalomega*. Retrieved from pegi3s: https://hub.docker.com/r/pegi3s/clustalomega/

Petar Brlek, L. B. (2024, 3 13). Implementing Whole Genome Sequencing (WGS) in Clinical Practice: Advantages, Challenges, and Future Perspectives.

Portainer1. (2024, 5 27). *Portainer.* Retrieved from https://www.portainer.io/

Portainer2. (2024, 7 1). *Settings*. Retrieved from Portainer Documentation: https://docs.portainer.io/admin/settings

Portainer3. (2024, 7 2). *Add a new container*. Retrieved from Portainer Documentation: https://docs.portainer.io/user/docker/containers/add

Portainer4. (2024, 07 2). *Advanced container settings*. Retrieved from Portainer documentation: https://docs.portainer.io/user/docker/containers/advanced

Privex. (2024, 6 6). *What is GPG/PGP and how do I use it?* Retrieved from Privex: https://www.privex.io/articles/what-is-gpg#what-is-gpg-pgp

Python1. (2024, 5 27). *Python*. Retrieved from What is Python? Executive Summary: https://www.python.org/doc/essays/blurb/

Python2. (2024, 5 28). *python*. Retrieved from tkinter - Python interface to Tcl/Tk: https://docs.python.org/3/library/tkinter.html

Samuel S, M. (2022). Computational reproducibility of Jupyter notebooks from biomedical publications.

The Regents of the University of California. (2002). pdbFormat. p. 1. Retrieved from https://www.biostat.jhsph.edu/~iruczins/teaching/260.655/links/pdbformat.pdf

Ubuntu. (2024, 06 05). *ubuntu*. Retrieved from ubuntu 22.04: https://releases.ubuntu.com/jammy/

Vieira, J., López-Fernández, H., Jato, M., Vieira, C., Ferreira, P., & Cavadas, B. (2024, 05 21). *i3s*. Retrieved from Bioinformatics Docker Images Project: https://pegi3s.github.io/dockerfiles/

Virtual Box. (2024, 5 21). Retrieved from https://www.virtualbox.org/

Wang, Z., & Ma'ayan, A. (2016). An open RNA-Seq data analysis pipeline tutorial with an example of reprocessing data from a recent Zika virus study.

Zhang Group. (2024, 06 19). *What is FASTA format?* Retrieved from Zhang Group: https://zhanggroup.org/FASTA/

Ziemann, M., Poulain, P., & Bora, A. (2023). The five pillars of computational reproducibility: bioinformatics and beyond. *Briefings in Bioinformatics*.

Zymo Research. (2021, 11 23). *What are Sam and Bam files*. Retrieved from Zymo Research: https://zymoresearch.eu/blogs/blog/what-are-sam-and-bam-files