Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Accelerating reproducible research in computational sciences

Author: Awday Korde (11435275)

1st supervisor: Dr. Adam S.Z. Belloum
2nd reader: Dr. Xiaofeng Liao

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 23, 2018

Accelerating reproducible research in computational sciences

ABSTRACT

Initially introduced as the act of merely reproducing someone's laboratory experiments, reproducible research has gradually mutated into more than just the act of providing a detailed description of the experimental setup. The marriage between theoretical and experimental sciences that drives the field of computational sciences has introduced a series of concerns when matters of reproducibility are instigated. These concerns are caused by a continuous increase in complexity of systems that choose to employ a large number of functionalities and methods in order to create novel solutions that aim at solving real-world problems. One of the most significant drawbacks that are encountered very often nowadays with such complex systems is the inability to reproduce the full software environment that generated these state-of-the-art results. In the discipline of machine learning, despite numerous attempts to avoid it, these issues have prevailed throughout time and have caused several publications that claim novel solutions to be rendered unreproducible. In this thesis, we initially investigate the current efforts taken towards achieving fully reproducible systems and identify the current limitations these efforts encounter. Furthermore, we present our prototype architecture and implementation that addresses the key drawbacks necessary towards attaining reproducible workflows. This is done by challenging the design of traditional workflow systems and envisioning the workflow as a conglomerate of isolated logical environments that facilitates reproducible workflows across environments effortlessly.

KEYWORDS

Reproducible research, empirical research, scientific workflows, machine learning, virtual environments, container-based virtualization

Contents

List of Figures

List of Tables

Listings

1 Background and rationale

The concept of reproducibility in the field of machine learning and statistical analysis has become a growing concern, especially when handling complex scientific workflows. Ranging from smaller studies to more extensive studies that tend to cover multiple information-theoretical properties, parameters, transformations, and algorithms; the vast field of machine learning has been continuously growing in complexity until the point where re-implementing algorithms can even take months [1]. Unfortunately, due to its increasing velocity, it is growing at a pace where vital information about each novel or state-of-the-art algorithm is either lost, misinterpreted or unreproducible. This matter has slowed down the advancements of research and has rendered a vast majority of empirical research the impossibility of being re-implemented or re-used. This phenomenon has led to a debilitating paradox [2, 3] where publications that focus only on summarizing their empirical results are accepted by the journal and conference reviewers solely based on faith. This faith-based empiricism factor has driven scientists and researchers that activate in the field of empirical research into a series of distrust in scientific communities and has challenged the credibility factor of numerous publications. An article published previously shows us that out of 259 bio-medical articles (51.7% of which presented novel findings) that contained empirical data, only four (1.5%) articles were identified as efforts to replicate previous knowledge [4]. The lack of publications that openly publish their source code, data and experiment details has prefaced, especially due to its highly empirical nature of work, a series of poorly documented work in many cases has led to research discontinuation.

Multiple initiatives that attempt to mitigate the risk of achieving highly unreproducible work have stemmed with the aim of conserving numerous aspects of the experiment themselves. Some efforts have focused on creating an open ontological standard with the purpose of describing various stages of the workflow to avoid misinterpretation (Predictive Model Markup Language [5]). Other initiatives focus on ensuring the experiments survivability by facilitating the release of software developed by researchers (software repository such as Github or Bitbucket). While many of these initiatives [5–8] are iteratively investigating techniques that conserve parts of entire workflows, little to none of them are combining all the aspects into one solution that would ease the steps of reproducing smaller to more massive scale experiments. In an effort to create a more modular and portable workflow, tools for composing workflows have emerged which allow users to interactively set-up their end-to-end experiments by providing a set of features, widgets, and plugins that streamline the process of conserving the end-state of complex workflows. These initiatives are called scientific workflow platforms; they combine various interactive functionalities, state-of-the-art algorithms and collaborative tools into a multi-functional platform that provides effortless methods of conceiving building blocks for end-to-end workflows.

With the features mentioned above in mind, researchers need not pay attention to various logging's or meta-data required to ensure that every step of the workflow is documented, stored and version managed such that once an experiment is complete, it can be entirely exported along with its necessary data sources and code-base. In an ideal world, the usage of such platforms would provide users with sufficient freedom to define their custom software but due to current limitations, these platforms have constantly relied on wrappers and connectors to ensure that users can take full advantage of user-define libraries when using these platforms. In an attempt to improve the adaptability of these workflows, data pipelines are progressively being utilized to process a significant amount of data through numerous sequential components which each take in a set of input data and produce the desired output. The length or complexity of such pipelines can vary based on the amount of workload required, one pipeline can be comprised of numerous third-party tools and can vary from a simple function to fully-fledged workbenches that require the usage of job schedulers, resource negotiators and many other tools that manage the execution of each component.

The main issue identified by system administrators with traditional pipeline setups is the constant integration with third-party or user-defined tools in the pipeline. Software packages, libraries and dependencies emerge on a daily basis, with the ever-increasing requirements of such tools in HPC (High Performance Computing) environments, a set of conflicts emerges when integrating software dependencies in multi-tenant shared environments. Software environment requirements demands have increased drastically, numerous tools such as Virtual environments [9] and software modules [10] attempts to address the issues of environment incompatibility or version mismatch, but due to underlying software dependencies and configurations, have failed to address the topic of reproducibility across computing environments. In Figure 1 we depict the specific fields and currently available solutions that concern our proposed topic. In this Thesis, We are going to address three types of systems that are currently utilized in empirical scientific research, specifically the application of machine learning workflows, this is done such that we can cover sufficient material that will help us build up the theoretical framework and motivation required to efficiently formulate and reason our proposed solution.

Machine learning toolkit in the figure below refers to the vast software packages and ecosystem that is currently utilized in scientific computing, the focus of this report shall be strictly on the

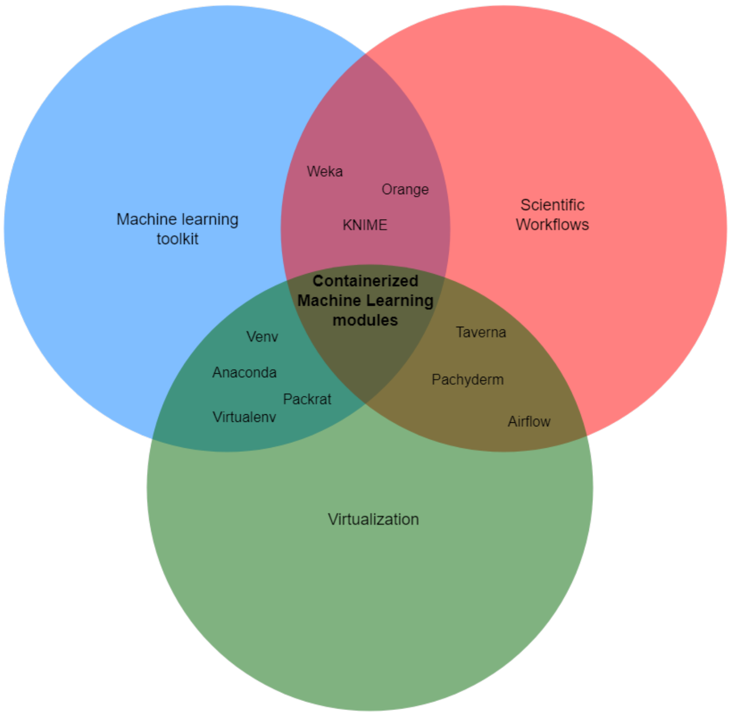


Figure 1: Venn Diagram

usability of such software packages as module components throughout a scientific workflow. This aspect is defined as a module due to the frequent use of such packages across numerous projects; users take advantage of the packages mentioned above to adequately formulate the specific steps of their workflows. A module can be comprised of pre-processing, algorithms or evaluation packages offered by either open source communities or software companies. We briefly mentioned scientific workflow above; this is due to the fact that scientific workflows are end-to-end workbenches which are comprised of multiple complicated execution steps. Each step consists of a series of one or more computational procedures. Workflows are designed in a manner that users can compose, execute and manage each step in an interactive and real-time environment. Over and above that is virtualization, which typically is applied in use cases where the isolation of resources, networks, and operating systems is required. This isolation is applied so that multiple applications can effectively run on the same machine, but due to the additional layers of abstraction added by virtualization techniques, processes do not interfere with each other and are executed independently from each other. Virtualization mechanisms in our experiments are going to be utilized to segregate each step (module) in scientific workflows and will provide users with the ability to reproduce the same experiment across multiple environments. Virtualization is key to our solution due to its ability to isolate hardware, network and process usage from each other, this allows the users to distribute and share their workflows with their peers or export their work on any desired environment. Last but not least lies the relationship between all three technologies that from our point of view, is a missing component that would further encourage the reproducibility aspect of scientific experiments, that is Containerized Machine Learning Modules (CMLM). CMLM serves as a gateway towards digitally preserving scientific investigations that would encapsulate essential open-sources tools to capture essential technical run-time characteristics. In this thesis, we will present the conceptual groundwork and performance metrics required for achieving and evaluating reproducible scientific experiments. This goal will be met by initially introducing the usage of scientific workflows nowadays, followed by two popular platforms namely, Weka and KNIME. Furthermore, we will introduce the concept of virtualization and its usability nowadays, along with the comparison of two popular container-based virtualization mechanisms. Moreover, we will define the methodology that depicts the setup and execution of our experiments. Finally, in the results chapter we will plot and discuss the findings resulted from our investigations, this is followed by a future work chapter that is going to encapsulate the necessary work required to extend our implementation with other HPC tools. In the following chapters, we will present the research questions that will drive the entire study, with a keen focus on (i) reproducibility aspect of empirical research, (ii) portability of solutions across environments, and (iii) applicability within the field of machine learning.

2 Research questions

In this thesis, we propose to extend the functionalities of platforms that try to achieve fully reproducible workflows. This aspect is achieved by making use of containers, or more specifically a container-based virtualization mechanism. This mechanism is known for its ability to encapsulate active applications away from the host system processes. By doing so we introduce an abstraction layer between the host machine and the running processes, the purpose is that of facilitating the capability of achieving fully portable and modular workflow components. A component here is defined as a function that helps researchers shape up the workflow according to their specification; They can choose from data sources, data transformations and algorithms to achieve so. The workflow itself is composed of components that activate as connected edges which execute a series of functions in a Directed Acyclic Graph (DAG) based formulation, allowing users to design reproducible workflows. The contributions of this thesis are twofold; First, we present a thorough literature study of two widely used workflow platforms and two container-based technologies with the aim of unveiling the benefits and limitations of such systems and mechanisms in reproducible sciences. Secondly, we plan on using the results of the prior section to perform two types of validations; (i) By initially quantifying the degree of reproducibility in our proposed solution (ii) Carrying out a performance comparison between the two proposed container-based workflows when faced with two machine learning use-cases.

The purpose of this study is that of identifying and validating the matter of reproducibility and re-usability in scientific workflows by using widely-used container-based virtualization mechanism as well as the performance of the two container mechanisms once faced with two machine learning use cases. The three main target groups the result of this study is going to focus on are as follows: (i) Researchers that are about to or are currently implementing complex components in workflows and want to achieve a higher level of reproducibility to avoid inaccurate or inconsistent solutions. (ii) A method that complements the requirements of conference review experts that require a higher degree of reproducibility when a paper submission is reviewed. (iii) Research that aims to build upon the results of other state-of-the-art machine learning systems, but fail to do so due to inconsistent/incomplete solutions. In the following section, we are going to elaborate on the specific goal by using a Goal-Question-Metric perspective (i.e., purpose, issue, object, viewpoint). The general aim of this study is going to be phrased into research questions along with it's relevance to this study as follows:

RQ1: *What are the* ***limitations*** *imposed by widely-used scientific workflows platforms when con-*

cerned with highly empirical work that is difficult to reproduce?

Rationale: This research question aims at identifying the faults or drawbacks encountered by widely-used collaborative scientific workflow systems. The demand for platforms that handle a multitude of computing tasks has seen an increase [1] in recent years. This is due to the increasingly empirical nature of numerous fields (examples of such can be bioinformatics, economics, artificial intelligence) rather than that of a philosophical approach. Over the years the increasing amount of empirical work has as increased in complexity when attempting to develop and create systems that are considered state-of-the-art or novel. One aspect that has been repetitively omitted when inventing these complex systems is the aspect of reproducibility and re-usability. Thus, it has brought us to a time where newly created state-of-the-art algorithms or methods cannot be adequately reproduced or re-applied in different scenarios. Here we plan on identifying the current drawbacks of scientific workflow platforms that aim at easing the process of creating reproducible and re-usable end-to-end workflows such that we can pinpoint the specific aspects that can be improved to achieve fully-reproducible systems.

RQ2: *What are the **trade-offs** when introducing a thin layer of virtualization to ensure the integrity of entire scientific workflows?*

Rationale: Virtualization technologies have allowed us to envision entire system as independent blocks that activate as guest operating systems on top of the host system itself. light-weight virtualization has pushed the limits of distributed applications and shifted the abstraction level when purely envisioning a system as just an isolated block into one where it is broken down into smaller functional chunks that once glued together make up an entire system. The primary application of such light-weight virtualization technique in scientific research has been that of entirely fitting systems into smaller blocks that are initially created for serving one specific function rather than that of an entire system. Here we leverage light-weight virtualization mechanisms to break down complete workflow systems into smaller logical blocks and use that to investigate the performance and advantages of such methods in practice.

The first research question will be addressed by means of a thorough assessment of two well-known scientific workflow platforms, namely Weka and KNIME. Each platform will be analyzed from two different perspectives, firstly the techniques implemented that allow the system to export workflows in a reproducible manner across environments and secondly, the pitfalls encountered once dealing with incompatible custom-defined software.

The second research question will be addressed by initially introducing the concepts and applications of virtualization techniques in HPC, followed by the comparison of two well-known container-based virtualization technologies. Afterwards, we will introduce two own implementations of container-based workflows that will be tested and compared with a non-virtualized system in order to determine the trade-off when utilizing the workflow in practice as well as the the level of encapsulation.

In the next Chapter we will start addressing the first research question by introducing the concept of scientific machine learning workflows. Furthermore we will describe and discuss the differences between two widely-used platforms in order to determine the drawbacks and similarities of both platforms. Moreover, after we identify the main characteristics and set-backs of such work-flows, we will dive into how our solution came to be, and how, by means of a result-centric manner, our solution relates to the topic at hand. We will further elaborate on the main state of the art solutions, this is done such that we can sufficiently argue their results along with their set-backs.

3 Literature Study

With this literature study, we plan on systematically identifying and unfolding two widely-used workflows that are currently put in practice, from an academic standpoint. In the following subsections, it is fundamental to achieve a good trade-off among the coverage of the existing research on the topic considered. Thus, our search strategy consisted of an automatic search, where by means of a filtering and querying process, it enabled us to have more control over the number of publications investigated. This section is entirely driven by our research questions, having a keen focus on;

1. Solution-oriented research papers, where the type of research that is being carried out aims at solving an existing problem by proposing a novel/illustration/improved version of the existing solution.

2. Publications that address scientific work-flows that are specifically designed for handling large-scale applications or tasks.

3. The level of interpret-ability and reproducibility of the proposed solution itself.

That being said, this literature study will provide a solid theoretical framework on which our proposed solution is based upon.

3.1 Scientific machine learning workflows

The amount of research on proposed scientific workflows has been steadily increasing in recent years; this is due to the necessity of achieving a structured and unified method of approaching experiment designs. This necessity was instigated by numerous publications that failed to provide sufficient observations and methods which would enable other scientists to reproduce their results. Reproducibility in this context ranges from low-level atomic activities of developer actions to outcomes of entire projects [11]. The significant amount of recent breakthroughs in many sciences has shown that the complexity and comprehensiveness of publications that propose novel or state of the art solutions has vastly increased. Measures have been taken by major conferences to ensure that experiments undertaken along with the submitted results of the publications submitted are fully reproducible [12]. The Sad Tale of the Zigglebottom Tagger [2] outlines the consequences that are typically encountered nowadays. The paper tells the story of a researcher that, even after multiple unsuccessful attempts, cannot seem to reproduce the same results as described in the targeted publication. Whereby means of a faith-based empiricism, the researcher believes that their own reproduced solution is inconsistent or inaccurate and that there is some component or parameter that they are overlooking. Numerous key details and observations are omitted from scientific publications, and as a consequence, this renders the results unusable which ultimately tends to lead to slower progress in its scientific field.

Many efforts have been taken towards facilitating the reproducibility aspect of experiments that are designed by authors of such state-of-the-art solutions. Some efforts [13–15] focused on leveraging scientific workflows to track and store immutable provenance information with the aim of preserving crucial meta-data about the history of the workflow and the specific outcomes. Where, by utilizing a series of one or more history-tracing extractors, users and scientists alike could model the specifications of multiple dependent tasks in a workflow into a thoroughly documented workflow template. This template, when shared with other peers or published openly, would be utilized in combination with a multitude of tools in order to reconstruct the control-flow patterns of each of the tasks defined. Each task mentioned before is attested and verified by the author with the use of legally valid digital signatures which would further enforce the integrity of the workflow and preserve data and process related provenance. In this Thesis report we will omit the aspect of provenance in scientific workflows throughout our literature study and we will focus our efforts in establishing a solid groundwork towards identifying the software limitations when designing such workflows.

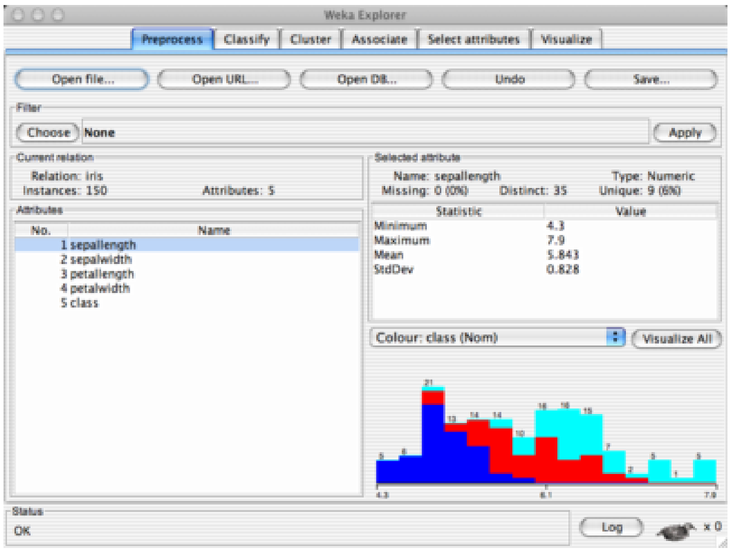
Many scientific workflows have thrived towards seamlessly streamlining the process of designing fully-functional computational pipelines. To name a few, WEKA [16], KNIME [17], WINGS [18] and OpenML [7], the purpose of each of these platforms is that of creating a standardized workflow for statistical analysis that would allow scientists and researchers to work more efficiently by collaborating throughout their undertaken experiments. With the release of the mentioned platforms stemmed multiple benefits, while some provided a user interface that would a users visualize and represent the models performance over a datasets, other platforms would focus on modularizing the scientific knowledge (preserve the state of the flow that was used to achieve a particular outcome) as much as possible. The integration of graphical interfaces and modular libraries in these platforms provides scientists the ability to investigate and compare their datasets or results by interactively plotting their analysis in real-time. These platforms provide users with the ability to set up a specific workflow structure for their current project; these workflows are meant to keep track of the modifications and changes iteratively. Furthermore, some platforms have achieved widespread acceptance in academia [19]; this aspect additionally stresses the need and awareness of such tools and frameworks in scientific research. Each of them provides a specific set of tools and algorithms that are developed and maintained by scientists or developers over-time, examples of such would be regression, classification and clustering techniques. The concept of modularized components in workflows is increasing, this is due to the numerous steps (pre-processing, training and evaluating) that need to be added for an end-to-end machine learning workflow to be complete. One standard feature that can be seen across each platform is that of having a visual programming paradigm concept where by employing a drag-and-drop approach it enables users to plug and execute various components of the workflow interactively. This functionality can be reached from the dashboard of each initiated project or workbench. We have introduced a minimal representation of what scientific machine learning workflows currently exist, in the following sub-sections we will dive into the technical implementation of such and further investigate the workflows reproducibility capability and modular traits.

3.1.1 WEKA - The Waikato Environment for Knowledge Analysis

Designed with the outlook of accomplishing a modular object-oriented type architecture, Weka has been re-built from the ground-up in the Java [20] Language, initially written in C [21] and Prolog [22], Weka is currently distributed under the GNU(General Public) License. Due to its gain in popularity, the Weka project has continuously improved since its release in 1994 [16]. The main reason behind the decision of re-writing the source code of the Weka platform was due to the increasing difficulty users would face when dealing with the management of dependencies, libraries, and configurations of the system. With the upbringing of the newly re-built system, Weka focused its attention on the ability to "Write Once, Run Anywhere" [19], this allowed the platform to incorporate a more modular approach of packaging their distributions. While this change allowed the system to have more platform agnostic capability, Java was still in its early stages at that time, (less than two years old at that time) consequently making it an uncertain decision whether it would handle the computational workload necessary when dealing with machine learning algorithms. The newly re-written system highly facilitated the acceptance of the software in the community, resulting in the development team receiving a SIGKDD Data Mining and Discovery Service Award [1] in 2005.

With newer versions, the Weka project extended towards adding more pre-processing features, graphical improvements and embracing the support for standards in the academic community. In addition to its primary features, at the core Weka workbench lies a variety of state-of-the-art machine learning algorithms. Users could take advantage of the available algorithms, pre-processing

[1] http://www.kdd.org/awards/sigkdd-service-award



Filter: None

Current relation
Relation: iris
Instances: 150 Attributes: 5

Attributes

No.	Name
1	sepallength
2	sepalwidth
3	petallength
4	petalwidth
5	class

Selected attribute
Name: sepallength Type: Numeric
Missing: 0 (0%) Distinct: 35 Unique: 9 (6%)

Statistic	Value
Minimum	4.3
Maximum	7.9
Mean	5.843
StdDev	0.828

Colour: class (Nom)

Status
OK

Figure 2: Preprocess Panel of Weka

tools, and drag-and-drop functionality to experiment and analyze the results of their trained models on new datasets in an effective way. The graphical user interface named the "Explorer" is used to reach any of the functionalities provided by the system and can be seen in Figure 2. Each functionality would be located on the panel part of the interface itself, where all the plug-ins are ordered by their intended use in the workflow. The initial stage of the workflow would typically start off with the data source that is intended to be utilized in future stages of the experiments. Datasets could either be loaded into the main memory by importing it from a CSV file or extracted from a database. The ability to extract information was facilitated by the Java Database Connectivity [23], whereby making use of SQL queries, users could retrieve their relevant information from any database that would have the compatible supported drivers. Weka's Explorer is only designed to support batch-type data processing, due to this aspect, the system itself encounters multiple performance issues when handling large data sets. This typically causes the system to load the entire corpus into the main memory leading to a slower overall performance; the user can overcome this issue by using a set of streaming algorithms that would split the data-set into mini-batches.

One of the dependencies of using any of Weka's interfaces is that of being required to provide the system with a certain amount of heap space. This is considered one of Weka biggest stumbling block [19]; the user needs to specify the amount of memory (lower than the host's memory) allocated to the Java Virtual Machine (JVM) such that swapping can be avoided. This is due to the just-in-time (JIT) [24] compiling trait of JVM. Numerous attempts have been made towards resolving this impediment by automating the just-in-time compiler of JVM [25] so users would not specify the amount of heap space.

One of the critical features of Weka is that of not only having a rich collection of state-of-the-art algorithms but more so having a modular implementation of the algorithms. By employing a modular implementation, Weka enables users to combine various algorithms (bagging, boosting and so forth) without even specifying a single line of code (Interactive drag-and-drop functionality). The Weka Architecture relies on partitioning significant components into Java classes such that new functionalities (filters, algorithms and so forth) can be added much quicker. Weka is a promising and elaborate tool that provides users with a graphical interface from which they can take advantage of the various functionality offered by the system, in the following section we will investigate a similar tool named KNIME. This is done so that we can distinguish the functionality of both systems as well as their drawbacks within scientific research.

3.1.2 KNIME - The Konstanz Information Miner

KNIME [17], initially intended as a data pipeline for data mining challenges within scientific research communities, it is used to empower researchers with the ability to streamline the process of implementing end-to-end data mining workflows. The combination of interactive interface, visual intuitiveness, and collaboration tools allowed KNIME to become a favorite data mining medium when developing machine learning models. Multiple efforts were taken by other well-established competing data pipelining tools that were aiming at providing similar data-mining functionalities, but one of the key developments of the project itself was thanks to the community contributions and open source model. Many of the interactive graphical functionalities of Weka can also be seen in KNIME; users can interactively connect building blocks, named Nodes, in a Directed Acyclic Graph (DAG) based workflow formulation. Nodes represent a collection of data preprocessing tools, algorithms, and visualizations in the workflow. In Figure 3 we observe connecting nodes that can be comprised of transformations, algorithms, and visualizations that help shape up the workflow according to the users' hypothesis or needs. Much like Weka, a vital aspect of the design is that of being modular. The decision of having a modular design was taken to encourage the development of independent algorithms; this would allow the users to customize and adapt their workflow based on their specific needs. While doing so, the development team of KNIME focused on implementing open transformations and preprocessing nodes that would not rely on specific data types and provide a more generic functionality. The system leveraged a series of connectors and extensions that take advantage of the currently available big data tools and frameworks[2]. The project supports popular big data frameworks such as Spark [26] and Hive [27] which were integrated to levarage

[2]https://www.knime.com/software

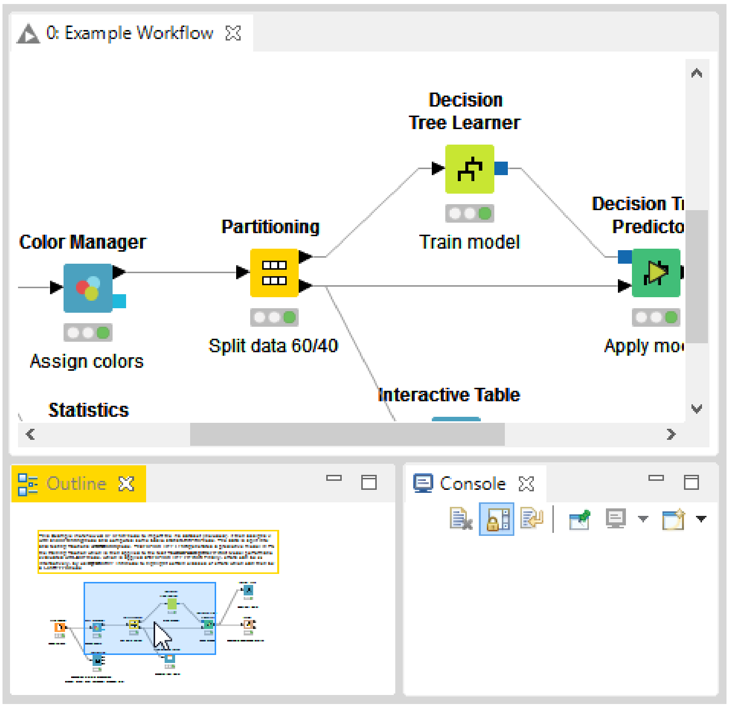


Figure 3: Workflow view

their ability to parallelize and distribute the workload of KNIME nodes respectively and store large amounts of information across a cluster of servers. The KNIME system also allows users to further fine-grain each node into nested-nodes called Meta nodes [3], this allows for the creation of complex workflows which encapsulate a series of actions or processes into a single node. Thus treating the meta-node as a form of a sub-workflow, each encapsulated sub-workflow may be recursively executed, meta-nodes can be utilized when more complex scripts or transformations are required (bagging, cross-validation and so forth).

The architecture is divided into two main components, the KNIME server, which is dedicated to collaboration and deployment of models in organizations, and the KNIME analytics platform, where researchers and users can interactively set-up their project and develop their workflows by taking advantage of more than 1500 modules available. The server provides a secure layer of access management on each level (Node, workflow, and application) such that users are allowed only to interact or edit resources for which they have permission to do so. Both engines are built in Java[4] and suffer from the same limitation that Weka does as well as the manual JVM heap space allocation. New components are incrementally added to the platform that attempt to extend its support towards other programming languages and frameworks.

[3] https://www.knime.com/metanodes
[4] https://github.com/knime

3.1.3 Discussion

As seen in the above work-flows, they both offer a variety of functionalities, support for popular frameworks and a rich collection of state-of-the-art algorithms. While both are open-source systems and are widely used in academic circles, the aim of the platforms in research is that of offering a medium for researchers and scientists alike where they can share their experiments such that every published result can be reproduced starting from the data source until the evaluation phase. The design of both systems allows users to add their customized functions,transformations, and algorithms by using the currently supported programming languages and wrappers(Java, Python,R, and Weka). The matter of reproducibility has been addressed to a certain extent, due to the monolithic design (hypervisor-based virtualization system), both systems suffer from the same consequences that have not been adequately addressed. Which is represented by the limited amount of frameworks supported, version dependencies, portability of workflows and the inability to isolate different stages of the workflow (ingestion, preprocessing, training and evaluation). The isolation of each stage of the workflow allows scientists to preserve the state of each execution undertaken, underlying dependencies, hardware independence, availability accompanied with the appropriate ability to handle heavy workloads in an HPC (High-Performance Computing) environment. Hypervisor-based virtualization has caused a significant amount of doubt in HPC environments [28–30], thus being avoided for as much as possible. Its promising abilities to isolate processes and applications align with the reproducibility aspect of our subject and VMs have been investigated previously [31] for their applicability in HPC environments. In the following section, we will investigate the usage of various container-based virtualization systems and the benefits they might bring.

3.2 Container-based virtualization

Virtualization technologies have arrived with their own set of essential functionalities, and they have gained quite a momentum due to their ability to encapsulate independent applications or processes. One benefit that comes along with virtualization is that of granting the encapsulated application the ability to achieve resource and process independence on the same host OS (Operating system) which is necessary when trying to avoid any interference between other running processes. Multiple Guest OS can run on a single machine due to the systems hypervisor on which it enables the usage of virtualized instances, or more specifically VMs(Virtual machines) or Containers, on the same host machine. Each VM that is provisioned on the same machine function as independent blocks, they abstract essential components such as storage and network to restrict the amount of resources the VM is allowed to tap into (specified by the user).

Due to their ability to provide such an abstraction layer, commercial virtualization technologies, such as (Xen and VMware), are considered one of the foundation technologies of cloud computing [31]. In recent studies [32, 33], it is shown that the use of virtual machines, or more specifically hypervisor-based virtualization technologies, achieve a high overhead in the execution of CPU-intensive applications and incur a performance degradation in network latency. Whereby abstracting an entire operating system that executes jobs and processes in complete isolation results in issues such as double-cache, network-driver abstraction and an overall performance degradation in HPC environments [33, 34].

Container-based virtualization, a "lightweight" version of the hypervisor-based virtualization, aims at mitigating the performance overhead and introduces a new set of features that prevail those of hypervisor-based virtualization technologies. Due to their ability to share resources with the host machine, containers are able to avoid some of the penalties incurred on the hardware level isolation and reach near-native performance when tested against CPU-intensive applications [31, 34]. Containers come with the advantage of achieving lower start-up times [35] than that of a traditional hypervisor based virtualization, this is due to the degrees of isolation each of these virtualization technology employ. Each of the aforementioned technologies handles processes, filesystems, namespace and spatial isolation differently such that resources and operating systems can be provisioned independently for each running instance. Both virtualization techniques allow the sharing of resources and aspire to optimize the cost of provisioning additional isolated resources on top of host machines. Where each provisined instances would propagate a series of hardware abstractions for each virtual machine or container in order to segregate different components of the application.

In recent years, the focus has shifted towards the usage of container-based virtualization in HPC applications. These applications have a large amount of processor-intensive tasks, thus requiring the usage of parallel infrastructures so that each job can be divided in multiple mini-batches and distributed across a network of connected machines. Once this workload would be encountered by a hypervisor-based virtualization technology, multiple performance penalties would emerge once executed on top of the hypervisor context needed for Hypervisor-based virtualization [36]. Containers have gained increasing attention in HPC applications due to the benefit of removing thethe hypervisor dependency, just-in-time compilation, the performance degradation and the slow booting times of VMs.

The additional benefits that containers bring to the table made it a viable choice for general application isolation, infrastructure deployment and packaging of software services compared to the traditional hypervisor-based virtualization approaches, especially in I/O intensive applications. Additionally, the process, filesystem, and resource isolation layer bring the advantage of sealing shut an application along with all the global resources and environment dependencies. This prevents users from running into incompatibility issues that would otherwise render the entire application/job unusable until the required internal dependencies are addressed. Container technologies such as Singularity, Docker, OpenVZ and Linux containers (LXC) have rapidly contributed to the development and wide-spread of container-based virtualization mechanisms. Each of the technologies mentioned above implements their method of achieving process hardware and network isolation, while some focus on specific applicability in the industry, such as Docker [37], others focus on the portability containers across HPC environments, such as Singularity. Container-based virtualization has gained an increasing popularity in the scientific field as well, one of the most desired applicability of contains would be the reproducibility and portability aspect.

In the previous section, we have provided a high-level overview of which virtualization techniques and why they are widely-used nowadays as well as the applicability of each in scientific research. In the following Sub-section, we will dive into how two well-known container technologies (*Docker* and *Singularity*) achieve isolation and effectively overcome the limitations imposed by traditional hypervisor-based virtualization methods. With this piece of information, we will investigate the extensive experiments done throughout the time that concern container-based virtualization and pinpoint the advantages that these container mechanisms can introduce in scientific workflows. The goal of this subsections is not of promoting any particular technology but that of investigating the technical solutions that address the challenge of reproducibility across numerous domains. The following sub-sections will provide us the necessary literature such that a solution can be derived based on the benefits that container technologies might bring into scientific workflows.

3.2.1 Docker Containers

Released in 2013, Docker [37] has rapidly become one of the most important leaders and contributors in the open source community due to their commitment and wide-spread of their container-based virtualization technology. Docker provides users and industry-wide experts alike the ability to wrap software applications or services into a reproducible, re-usable and rapidly-deployed self-contained unit, allowing them to migrate their application to any platform or operating system efficiently. The industries "Dev-Ops" approach has vigorously enforced the usage of Docker containers. The approach is that of documenting each stage of the deployment such that each documented stage contains its own set of dependency and requirements that are necessary for its execution. Similar to the DevOps approach, Docker containers are built from the OS up, each container build requires a list of instruction that is executed at runtime, named a Dockerfile.

A Dockerfile, as seen in Listing 1, is a clear record that contains specific requirements listed by the user, each requirement is seen by the Docker *daemon* as an independent readable/writable layer. A list of instruction can contain commands such as FROM, which initializes a new build phase and selects a base image for the container, or MOUNT, that allows containers to mount volumes located on the host machine into the container at runtime.

Listing 1: Dockerfile example

```
FROM ubuntu:14.04
COPY . /app
WORKDIR /app
RUN pip install numpy
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Each layer is comprised of a list of generated instructions and previously (the default is now OverlayFS) was stored in the AUFS (Advanced multi-layered Unification Filesystem) storage driver that merged multiple image layers into a single representation [37] in the union file system. This allows for a faster start-up time by preserving and compressing the disk space of each layer such that the integrity of the container is kept.

Each of the executed layers is represented by a unique ID, once an exact similar layer is being executed in another Dockerfile, the same layer can be reused across an unlimited number of Dockerfiles. An uninstantiated container, also named as an image, is comprised up of multiple layers, once a Dockerfile is built and the list of instructions are fully executed, an image is created containing the specified requirements, meta-data containing the history of layers, and a randomly generated UUID (refer to Listing 2).

A Docker image is made up of the layers specified in the Dockerfile, once an image is instantiated it first pulls all layers located remotely, stores them in the AUFS, and then executes all the remaining layers that are can be executed locally. Both the generated image and Dockerfile have important functionalists because they provide users and existing CI (Continuous integrations) tools, such as Travis CI[5] and Jenkins[6], the usage of simple scripts that defines the exact layers required and instructions scripted by the users. Both the Dockerfile and the image object generated can be stored and version managed in external/internal repositories, examples can be code repositories(Git, Bitbucket) and image repositories (Docker Hub, private image repository) respectively.

Listing 2: Instantiating a Docker container

```
awdayk@YOGA:~/hello-world\$ docker run -it ubuntu:14.04 /bin/bash
Unable to find image 'ubuntu:14.04' locally
14.04: Pulling from library/ubuntu
99ad4e3ced4d: Pull complete
ec5a723f4e2a: Pull complete
2a175e11567c: Pull complete
8d26426e95e0: Pull complete
46e451596b7c: Pull complete
Digest: sha256:ed49036f634....
Status: Downloaded newer image for ubuntu:14.04
root@fcc8e83cf641:/#
```

In Listing 2, an example command is shown of a docker command that is utilized to instantiate a container. Once this command is executed, the docker-client first initializes the process by sending the necessary meta-data (image specified, version, image repository and so forth) to the Docker *daemon*. Then the *daemon* itself will attempt to pull the *ubuntu version 14.04* image from an image repository that is, if the image cannot be found locally, by downloading each layer specified in the *ubuntu* image sequentially until fully loaded. Furthermore, once the Docker daemon fires up the container instance, it allocates a file system to the container, allowing it to have its directory structure and files, creates a network interface that grants the container access to connect to the default network (*designates an IP address for the container*) and initializes the container by executing the */bin/bash* command in an interactive terminal [31]. The command mentioned above spins up a container that inherits the Ubuntu distribution as a base image; traditionally users would execute the *docker run* command pointing at the dockerfile (as seen in Listing 1) such that multiple instructions can be carried out during runtime. Docker images allow developers and users alike to escape the "Dependency hell" often encountered nowadays [37].

[5]https://travis-ci.org/
[6]https://jenkins.io/

The images enable users to easily export a binary object containing all the software dependencies, files used and underlying software specification of their experiments and applications. This would typically be a difficult task for researchers that want to understand, evaluate or alter those dependencies when using traditional hypervisor-based virtualization technologies [38]. Due to the ever-changing and actively developed tools, system administrators have to deal with issues such as library versions, dependencies, and software compatibility [31]. Usually, the code and data published alongside research papers would be accompanied with various undocumented assumptions and configurations [39]. Dockerfiles and docker images encourage users to create their isolated environment based on the code and tools they require such that platform portability can be achieved regardless of the tools, configurations, and dependencies needed. Another vital trait that Docker achieves is the reproducibility aspect of experiments where researchers can immediately reconstruct the computational environment such that the source code can be ported and executed across numerous platforms.

The process of materializing a container from an image involves the creation of multiple levels of abstraction (hardware, process, and network) that allow the creation of a loosely coupled container instance on top of the host machine. In order to achieve that, Docker implements a client-server type architecture, namely the docker *client* and docker *daemon*. The docker client can be controlled by using the Docker CLI or API; users can perform commands such as docker build, docker run and docker pull to interact with a single or multiple docker daemon(s) (*dockerd*). The docker daemon resides on top of the host-machine and listens to incoming Docker API requests, when once a request is received, it initializes the creation of Docker object (images,containers, and networks) or executes a specific action on the daemon level. The Docker daemon is also capable of communicating with multiple docker daemons and create a distributed network that allows containers to be distributed across multiple Docker daemons.

To do so, Docker uses a built-in feature named Swarm [40]. Swarm is a container-network solution that provides standardized interfaces between Docker daemons and their network drivers [41]. This functionality allows containers living on different docker daemons to communicate and interact with one another through an overlay network, this simplifies the network configuration and ensures that each service (a group of containers that have the same functionality but are replicated across different hosts) is under the same network subnet. Among other features, Swarm works as a cluster manager and load balancer for managing container; it makes sure that each exposed service is evenly distributed across multiple nodes (a node is defined as an instance of the Docker engine participating in a swarm[7]. Docker Swarm is also in charge of maintaining the replicas resilient (auto-healing is triggered whenever one replica crashes, docker swarm replaces the crashed replica with a new replica).

Docker containers provide a promising range of functionalities that maintain the integrity and structure of executed experiments across various environments as well as it increases the awareness of reproducibility among researchers due to its high adoption in the software community with its rich ecosystem. Nonetheless, despite its advances, Docker containers are not perfect, and they face a series of limitations and potential shortcomings. [38] Pinpoints the issues that are currently faced when reproducing research with docker, namely computer security issues that have not been thoroughly evaluated, possible limitations in reproducibility introduced by the OS-level virtualization and the lack of significant adoption of Docker in scientific circles. These limitations were identified by reviewing current approaches in hypervisor-based virtualization and workflow systems and successfully deriving four significant challenges. Namely, Dependency Hell, Imprecise documentation, Code rot and Barriers to adoption. [42] proposes a new approach for modeling containers in a multi-node setup that addresses the lack of management and runtime system evolution in Docker containers. This was achieved by introducing an event processing application that is used to verify, reason, deploy and manage the life-cycle of Docker containers.

Other publications such as [41, 43–45] compare the performance of Docker containers in I/O, CPU, GPU, TCP Throughput, energy consumption against bare metal systems, hypervisor-based and other container-based technologies. Docker containers manage to reach near-native performance in nearly all the tests faced while not incurring any notable drawbacks, in most cases [8] performing significantly better than its hypervisor-based virtualization counterpart. In the following sub-section, we will continue to investigate another container-based virtualization technology, namely Singularity, to accurately distinguish and differentiate the benefits and disadvantages of both container-based solutions.

3.2.2 Singularity Containers

[46] proposes a new container-based virtualization technique that aims at solving various difficulties encountered by Docker containers named Singularity. With the promise of offering a more secure, scalable, compatible and reproducible solution, Singularity containers have opened a new door towards the usage of containers in supercomputer and HPC environments. Where concerns

[7]https://docs.docker.com/engine/swarm/key-concepts

[8]Higher energy consumption when executing computationally-intensive tasks were encountered in some cases.

such as the lack of performance, security, and usability are partially or fully mitigated by removing the necessity of granting users root-level access in multi-tenant shared environments. Such abstraction layer that is created by Singularity, as touched upon in previous sections, bring various advantages that enable the usage of customized environments that results in reducing the effort scientists take when building custom software.

As many of the container-based technologies found nowadays, Singularity containers take advantage of the *chroot* and *bind mount* Linux kernel features and MPI (Message Passing Interface) implementations to introduce the usage of containers across a wide range of HPC architectures. Singularity also introduces the capability of leveraging existing docker images (described in Section 3.2.1) to generate singularity container definitions. This allows users to convert their existing Docker image manuscript into a re-usable single-file image that can be ported in a wide-range of architectures. The usage of Docker containers in scientific computing has proven to bring forth a myriad of security and usability challenges [47]; this exposes multiple failures points when utilizing Docker in multi-tenant shared environments that cannot be immediately addressed. Singularity containers emerged due to the necessity of achieving entirely isolated portable environments in scientific computing that provides users and researchers alike the ability to reproduce end-to-end workflows seamlessly secure.

Listing 3: Singularity container instantiation

```
singularity exec ubuntu.simg cat /etc/os-release
NAME="Ubuntu"
VERSION="14.04, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
```

Users have the freedom to share, collaborate or publish their experiments and source code by making use of the Singularity hub [48]. A singularity container can be launched, by using the command shown in Listing 3, and interacted with in a shell terminal where users can execute their required tasks.Once finished, users can commit their changes and push their newly created image to the Singularity hub repository. The Singularity hub adds an additional layer of reproducibility and automation with the purpose of capturing, storing and analyzing the meta-data of each created container. Considering the urgency of reproducible software and workflows, Singularity hub complements the needs of HPC communities by enabling the use of web-hooks and version control management. Users benefit from such features by serving the necessary scripts via a RESTful interface that allows the automation of builds and version control.

Singularity containers are represented by a single-file image that comes packaged with the necessary Linux distribution, files, libraries, dependencies and environment requirements. They also come with the promise of addressing most of the security concerns found in most container technologies utilized in HPC environments [49]. The reason why Singularity touches upon such concerns is that the use of other[9] container technologies enables users to execute arbitrary code with escalated privileges (root) on the host machine. This has rendered multiple container solutions unusable in HPC environments due to the high level of control users would inherit when instantiating a container, allowing the usage of root-level commands would introduce security risks when handling environments that are shared across a vast amount users. For this reason, Singularity containers have become an appealing choice in multiple supercomputing or cluster-computing environments [47]. This is because of Singularity integrating essential features from other container technologies while also preserving the integrity and security of containers in a balanced approach. Another functionality that Singularity containers have adopted is being able to bootstrap a singularity container by using a Docker image.

Listing 4: Creating a blank image

```
awdayk@YOGA:~/hello-world\$ singularity image.create ubuntu14.04
Creating empty 768MiB image file: ubuntu14.04
Formatting image with ext3 file system
Image is done: ubuntu14.04
```

Singularity containers leverage the usage of the docker image manifest by tapping into the Docker registry, pulling the desired image, extracting each layer of the image and fully converting it into a single Singularity image. The connection between Singularity and the Docker registry is made through the Docker registry API that uses a RESTful interface to extend access to all image manifests of the respective repository that provides the capability of retrieving the necessary layers from the requested image.

[9]Namely Docker among others.

A Singularity container can be described as a merely a file that is created by specifying the compatible Linux distribution (distribution-specific tools and kernel-dependent features [10]) and the size of the image; an example can be observed in Listing 4. Once come into shape, the file created is an empty file system containing only the folder tree structure of Linux. To populate the file system with user-defined packages and dependencies a list of instruction named a bootstrap definition recipe is required such that each specification can be acquired and placed accordingly in the created image file. This command can be reached by simply specifying the image file and manifest intended to be used in the bootstrapping procedure. Similar to the COPY or MOUNT feature of Dockerfile, users can automatically access relevant file system located on the host machine when executing a container.

Listing 5: Singularity image manifest

```
Bootstrap: docker
From: ubuntu

%help
Help section

%setup
    touch Hello-worlds.txt
%files

   /home/awdayk/hello-world/hello-world.txt
   /hello-world.txt
%labels
    Maintainer Awday Korde
```

This functionality allows users to share files across the contexts of the containers that carry the same access rights as the user executing the command; an example can be seen in Listing 5. The manifest brings forth numerous sets of parameters users are allowed to specify in order to create an ideal execution environment. Two main sections divide the manifest mentioned above, the Header, and Section parts. The Header of the manifest, users specify the Linux distribution by defining the core operating system and kernel version which the container is going to inherit at build time.

[10]http://singularity.lbl.gov/faq

Each build base (references to layers the users want to make use of) brings forth a set of particular details the users needs to address to retrieve the desired image, such as using "shub" or "docker" to retrieve images hosted on Singularity hub or Docker hub in the order described. In the Section part of the manifest, among other user-defined parameters, users can specify in the "%post" Section of the manifest a series of scripts that are to be executed during runtime of the bootstrapping procedure. Configuration, installation and download commands are specified in the "%post" section and executed so users can install and pre-populate the container environment with the desired software packages. The copying of host files into the container file system cannot be executed in this section of the manifest, this section only allows users to specify download commands such as *wget* and *curl* to retrieve packages, data, and files. The section that is used to copy files from the host machine into the container at build time is specified as "%files", where it strongly reassembles and shares the functionality as that of the *cp* command in Linux. The execution of the "%files" section of the singularity manifest always occurs before that of the aforementioned "%post" command. The singularity manifest also provides users with the ability to inject tests, checks and environment variables into containers during the build time execution.

Once the container is bootstrapped and instantiated, the materialized container acts as an encapsulated read-only (defaults to a compressed immutable format) environment, where each function and script invoked within the container will inherit the underlying distribution and dependencies specified in the bootstrap recipe. As soon as an execution command is sent to the container, the script executed will not face any imposed limitation concerning processes or threads. Another aspect of Singularity that adds up to features that allow for the execution of programs is the ability to redirect IO, pipes, arguments, files, shell redirects located on the host machine directly from inside the container [46].

This shows that Singularity containers do not entirely achieve complete isolation from the host machine but primarily focuses on achieving collaboration between the host machine and container processes. Singularity containers also provide the users with the functionality of inspecting the content of a container by using the command shown in Listing 6. Checks allow system administrators to perform all the necessary tests required when initializing a container instance, this functionality when coupled with kernel interruption on failed tests or non-success stages brings an additional layer of verification which would indicate the existence of unexpected execution behavior on runtime.

Listing 6: Singularity check functionality

```
awdayk@YOGA:~/hello-world\$ singularity check /tmp/Centos-7.img
START 1-cache-content.py tags[default clean bootstrap] level[LOW]
PASS: (retval=0) python /usr/../singularity/../1-cache-content.py
```

Singularity containers emerged as yet another method of providing an efficient image-based technique that is tailored for the existing HPC ecosystem. The ability to convert Docker images into Singularity images adds an additional layer of re-usability with the existing large amount of Docker images. This allows users to wrap and port their current docker image manifest into an HPC environment which mitigates security and usability concerns in HPC systems. While sharing some of its features with its other competitor container mechanisms, Singularity brings forth a number of advantages when operating in HPC systems, such as the omission of cgroups, root level operations and the ability to bind, mount, or overlay system libraries or distributed file systems [47]. Due to their high demand in large-scale supercomputing environments, users can create, modify and update their solutions on their local machine and deploy the containerized solution to their desired HPC cluster or cloud provider without running into compatibility constraints. Many clusters, supercomputers, and commodity systems have included Singularity containers in their stack, examples of such clusters are the Cray XC supercomputer [47] and the San Diego Supercomputer Center [11].

Cloud computing research has also adopted the usage of container-based virtualization; this enabled the usage of singularity containers in task-based parallel applications [50]. [50] proposes the usage of container-based parallel execution of code that leads to a more efficient use of resource when compared to traditional KVM solutions. As part of the Open Container Initiative [12], [51] suggests an automatic generation of standard runtime specifications for Docker and Singularity containers as well as an object ingestion process that preserves the specification of a container at execution time, archiving and extraction tool of the intended specification respectively. The Boutique[13] Framework has tightly integrated the usage of Singularity containers, the framework automatically publishes, executes and deploys applications across numerous environments. Users can specify their desired execution flow by leveraging a JSON structure that specifies the command-line template, input, and outputs [52].

Singularity containers are implemented as a means to complement and encourage the aspect of reproducibility that allows the deployment automation of application sets and workflows

[11]http://www.sdsc.edu/support/user_guides/comet.html
[12]https://www.opencontainers.org/
[13]http://boutiques.github.io/

across environments. Singularity has proven on-par with native-executions when computationally intensive-applications were concerned; multiple studies address the performance efficiency of Singularity containers against other competing virtualization mechanisms [47, 50, 53–56]. Singularity containers currently appear to be one of the most secure, robust and attractive solutions in HPC environment, this due to its ability to natively support technologies such as the Luster file system [57], the InfiniBand architecture [58] and several other resource managers. Finally, Singularity containers bring their own set of flaws as well, the primary aim of singularity was that of being able to run container-based applications in any HPC environments, ironically the challenge faced nowadays is that of Singularity not being supported on all supercomputing platforms [47]. Even though Singularity comes with the promise of achieving a higher level of security than Docker, its main container connector occasionally escalates to root privileges when executing *for loops* in the process of mounting images [54]. Other flaws that can be considered would be that of not possessing native support for I/O or networking virtualization; this typically means that with Singularity users cannot benefit from shared virtual networking or multi-container orchestration [50]. In the next section, we will discuss the differences between both Docker and Singularity, and observe the similarity between the features embraced by both container mechanisms.

3.2.3 Discussion

In the previous section, we have laid out an in-depth depiction of two container-based virtualization mechanisms that are currently adopted in industry-wide and scientific computing communities. We have found that both technologies share some features with each other but do differ when investigating their underlying design and in some characteristics, the mechanisms differ in the interaction and dependency between the container engine and host machine. The first identified characteristic, among others, would be that of both container mechanisms requiring a manuscript, or simply an instruction list when bootstrapping a container instance.

Both technologies tackle the degree of reproducibility from different perspectives; Docker focuses more on adopting a Development and Systems Operation (DevOps) philosophy where communities have embraced it as a de-facto standard format for micro-service type deployments (One service per container). Whereby Singularity harnesses the portability of containers to bring the mobility of computing in the HPC community to satisfy the requirements of achieving scientific computational usage in cluster environments and computational centers. The different angles narrow down to the requirements of both technologies in different communities, that is for industry usage and scientific world respectively [46].

We have discovered multiple aspects of both container systems that bring along their own set of limitations or design flaws, with this in mind; containers are pushing the frontier of virtualization techniques where resource, filesystem, process and network isolation is of high concern. Both container mechanisms have proven to be potential solutions when reproducibility and portability of workflows and experiments are required, they enable the use of applications throughout numerous platforms and provide through different methods process and application isolation.

Table 1: Comparison table between Docker and Singularity

	Docker	Singularity
Namespace isolation	Yes	Yes
Resource isolation	Yes	Not supported
Network Isolation	Yes	Host network
Image portability	Multi-layered image	Single-file image
Storage drivers	OverlayFS	ext3
Host FS access	Mount/Volume	Mount
Image repository	Docker hub	Singularity Hub
Security	Requires escalated privileges	Doesn't require escalated privileges [a] [a]Users require root access to preform build (create), bootstrap or modify container
OS requirements	Ubuntu CentOS Fedora and Debian	Only compatible Linux Distribution
Community adoption	Adopted in enterprise and open-source applications	Slowly getting adopted in HPC communities

In table 1 we depict the main differences between Singularity and Docker, this is done such that we can get a broader overview of what these mechanisms ultimately entail. Namespace isolation plays an important role when users are concerned with the ability of spinning up a significant amount of containers on a single host machine or cluster of machines. This is because namespace isolation enables containers to activate in an enclosed userspace environment such that the container itself has its own set of processes and network drivers. The reason behind this partition is that it allows the processes and functions that are executed within the container not to interfere with other running processes on the host machine or other running containers on the same machine. An example would be that of Container A mounting a directory D and changing the directory name that is located on the host machine and Container B accessing the copied directory D without being aware that a specific action has been executed by Container A. This limits the visibility of other containers/processes that are executed on the same machine, thus creating the illusion of an independent execution across containers instances.

Docker brings its own implementation of namespaces, named *libcontainer*, which functions as a cross-system abstraction layer which enables the container interaction with Linux native namespaces. By *default* (Namespace isolation can be activated), Singularity offers no namespaces isolation due to the nature the containers were designed for, that is for HPC workflows and environments. Singularity relies on resource managers and queuing systems, such as SLURM [59], HTCondor [60] and Torque [61], to achieve process isolation; hence Singularity containers share almost everything with the host machine as would a normal script running on the host machine would.

Users can view, kill and modify host or other running container processes (Assuming the user has the privileges to do so) from inside the singularity container. Resource isolation or restriction refers to the hosts' ability to limit the upper bounds of resources (CPU, memory, I/O block and so forth) that is allowed to be consumed by a container. Docker, through its *libcontainer* implementation, leverages the Linux kernel *cgroup* feature to restrict the number of resources a running container can consume. However, Singularity does not support any kind of resource isolation due to it relying on resource schedulers to control the upper bound resource limit of their containers. Network isolation shares many specifications to that of the resource isolation; the Docker implementation allows containers to be spawned in a privately encapsulated network space on the host machine allowing for an added network isolation layer that utilizes the network kernel *namespace* feature through *libcontainer*. Singularity offers no network container isolation thus inheriting the host's network interface and offers network transparency across all running containers which eliminates the network virtualization overhead and introduces the need for an orchestrator or job-scheduler when multi-container deployment is required.

The main differences lie in the design of both system, Docker being designed for enterprise-grade application deployment that is made up of a conglomerate of light-weight services, whereas Singularity is designed for general scientific use cases where the focus is that of executing a single heavy-weight job targeting only one specific application. Hence, Singularity's focus is on single-application execution rather than multi-application execution. Both mechanisms employ different storage drivers, Docker by default implements an Overlay File System, also named OverlayFS, whereas Singularity implements an extended file system, named ext3, typically utilized by Linux kernels. Both implementations bring forth a set of trade-offs and influence the containers density and other performance aspects when dealing with I/O related operations.

The OverlayFS of Docker, faster than its previously implemented predecessor (AUFS storage driver), unifies a pair of two directories into a single directory on the Linux host. Usually referred to as layers, these directories enables a hard-linkage between layers in a multi-layer docker image setup. For example, a six-layered image retrieved by using the *docker pull* command creates seven directories where the first six directories correspond to its respective layer and content, while the last folder depicts the symbiotic link between each layer by referencing the layer identifier from its lowest directory (child-layer) til its upper directory (parent-layer). Singularity ext3 driver storage simplifies the process by creating a single-file image, as soon as a *singularity create* command is executed, which contains a traditional Linux filesystem organized by root directories followed by a series of sub-directories. Docker OverlayFS caches previously downloaded image layers, enabling for a faster build time, where Singularity does not support image caches requiring the image to be fully downloaded every time a *Singularity create/build* is executed. Both technologies bring forth their own image repositories, Docker hub, and Singularity hub, where users can choose to publish, update or retrieve an image from either a public or private registered image repository. The security measures taken by both Docker and Singularity containers vary and allow users to execute commands based on the degree of privileges they have. Singularity employs more robust secures measures than Docker does when the creation or execution of images is concerned.

Singularity comes with the promise of retaliating the security flaws identified in Docker (as previously addressed in Sub-section 3.2.2) by forcing the container to inherit the users' privileges from the host machine during runtime which results in limiting the use of root privileges when executing commands within containers. Both mechanisms are compatible with numerous operating systems such as Windows, MacOS (by leveraging hypervisor-based virtualization) and Linux distributions. While Docker natively supports Linux distributions such as CentOS, Debian, Fedora, and Ubuntu, Singularity supports only specific Ubuntu and Debian distributions due to the restrictions of some kernel-dependent features. Numerous cloud-providers have widely adopted docker containers and have integrated them into multiple third-party applications such that it has become the de-facto standard format for micro-service type deployments whereas Singularity containers gained some traction in HPC environments where the usage of a secure container is required when activating containers in a multi-tenant shared scientific computing environment. In the following chapter, we will start describing the methodology design that is going to leverage the technologies described above to achieve reproducible analytical workflows.

4 Methodology definition

In this Chapter, we will elaborate on our experiment processes and use case selection. The main idea behind these experiments is of validating the reproducibility aspect of the workflow and that of measuring the performance of container-based virtualization mechanism in combination with machine learning techniques.

More specifically, it is fundamental to achieve a good trade-off between the performance of such technologies and the re-usability aspect of such modules in empirical scientific research. To achieve this trade-off we will run two rounds of experiments, which are going to be defined below, they will provide us with sufficient information to evaluate the performance impact of using container mechanisms, that wrap user-specific function when compared to the performance of a bare-metal approach. The main takeaway point we expect to retrieve from these experiments would be whether the container mechanisms mentioned above are sufficiently mature enough to handle the required workload when dealing with machine learning use cases. Hence, our experiments will test the performance of two well-established machine learning applications, precisely that of computational linguistics and machine-learned ranking. In the following sections we will start depicting the hardware and software specification, methods utilized, and assumptions taken for each experiment as well as the specific parameters used for each algorithm.

Hardware specification. The aim of this section is that of providing information regarding the hardware configuration in which our experiments have been executed. In Table 2 we show the hardware specification that was utilized as well as the respective Docker and Singularity versions used. In order to make sure that we made a fair comparison in our experiments, we used the same compiler, software libraries and Linux distribution across all environments (Docker, Singularity and Host machine).

CPU type	Intel Core i5-7200U
CPU speed	2.50GHz
CPU # cores	4
Memory(RAM)	8 GB
Storage size	512 GB
Storage type	SSD
OS Type	Linux
OS Distribution	Ubuntu 16.04 LTS
Docker version	18.03.0-ce
Singularity version	2.4.5-dist

Table 2: Hardware configuration

In the following section, we will describe the setup that is going to be followed by both of our experiments. At first, we will describe the experiment setup on a more generic level explaining the various features and functions that are going to be utilized. This is done through the use of illustrations that will be portraying the system architecture. This will provide us a wider-overview on how our experiments can be applied to various use cases, *the purpose of these experiments is not that of promoting a specific technology or algorithm but that of validating the purpose of our study*. This section will be followed by the description of each of our use cases, that will undergo a series of experiments in order to ensure the integrity of our analysis and results.

4.1 Experiment setup

Initially, the first part of the stages that are defined below is made up of both Singularity and Docker's workflow. Our focus is that of introducing the aspect of re-usability and reproducibility while also keeping in mind the overhead this may entail. Thus, in order to ensure that our approach might be even considered a viable solution at a low-level of complexity, we decided to keep both container-technologies at the default configurations (some minor changes were undertaken to maintain similar configurations). The purpose of this decision is that of emphasizing on the usability of container mechanisms, introducing multiple layers of complexity at this point would entail lower odds of adoption among users and researchers alike. We have divided the flow of both systems into two different illustrations; this is due to the fact that the underlying design of both container mechanism differ from one another. As mentioned in Sub-section 3.2.3, both mechanisms have some similar features but are optimized for specific use cases.

4.2 Docker workflow

In Figure 4 we depict the specific steps executed when building the initial and custom images for our modules in Docker containers. As a first step we created a dockerfile (see Sub-section 3.2.1) that contains a set of software dependencies the component itself would require and a Linux distribution similar to the one the host machine utilizes (Ubuntu 16.04). With this in mind, after executing the build process in docker, the Docker client sends a build request to the Docker daemon. The Docker client in our case has explicit access (assigned to Docker access group on the host machine) to the Docker Daemon, inherently enabling escalated privileged calls to the Docker daemon power and ultimately allowing the users that are specified in the Docker group to inherit root-equivalent privileges. After the Docker daemon successfully receives the build request, it will initiate the build procedure. Which in combination with the instruction specified in the dockerfile manifest will propagate a series of layer retrievals that will materialize the base image and stored the layers in the OverlayFS (see Sub-section 3.2.3). After the base image has been retrieved and stored, the Docker daemon will execute the remaining series of instructions in the dockerfile. Each instruction is a series of one or more commands that constitute a set of differences from the

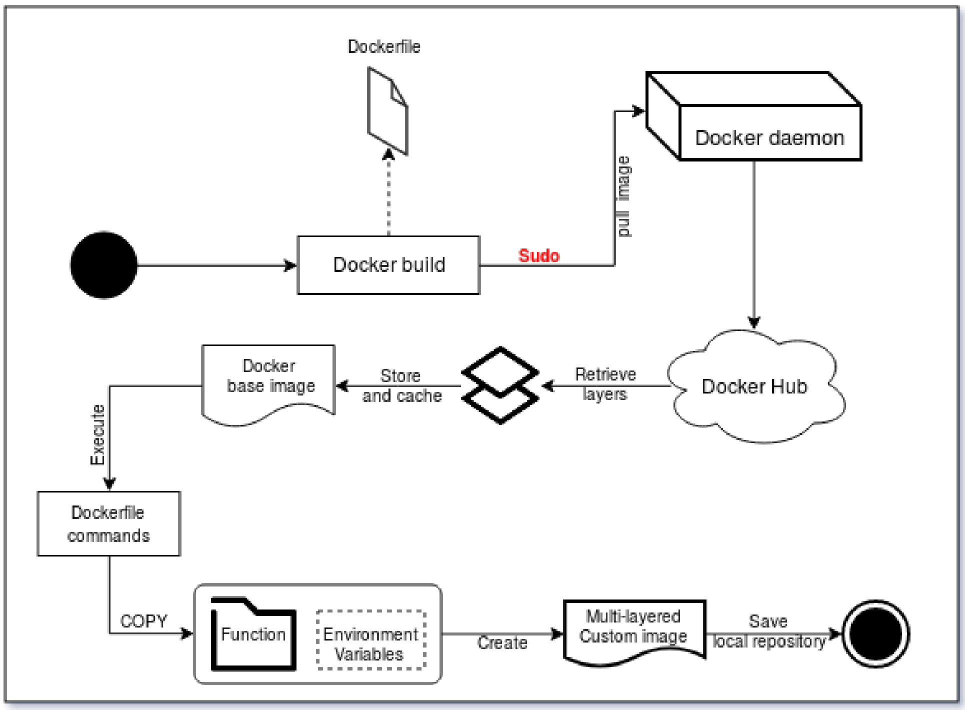


Figure 4: Build flow Docker

previously executed layer. Each executed command is contained in one thin read/write layer and executed in a sequential manner such that it is added on top of the previous layer. In our experiments, each layer constituted of a series of instruction that would import (by using the COPY instruction) the required files into one R/W container layer and another layer for the required packages and dependencies. Furthermore, the process of building our specified containers ends as soon as each instruction has been successfully executed and similar to the base image, our custom image is stored and cached into the overlay file system such that it can be started up at any point in time. Each module in our Docker experiments is materialized into a container and follows the aforementioned build flow. This is done such that there is a consistent build and run process across modules. In Figure 5 we depict an overview of the execution flow that will be employed by each stage in our experiments. The execution of the models will be commenced in a sequential manner such that each module will follow a process similar to that of an ETL (Extract-Transform-Load) process. The ETL process follows a specific pull, process, and dump pattern, in our approach we employ a similar process where each module created will ingest, process and dump the data into a

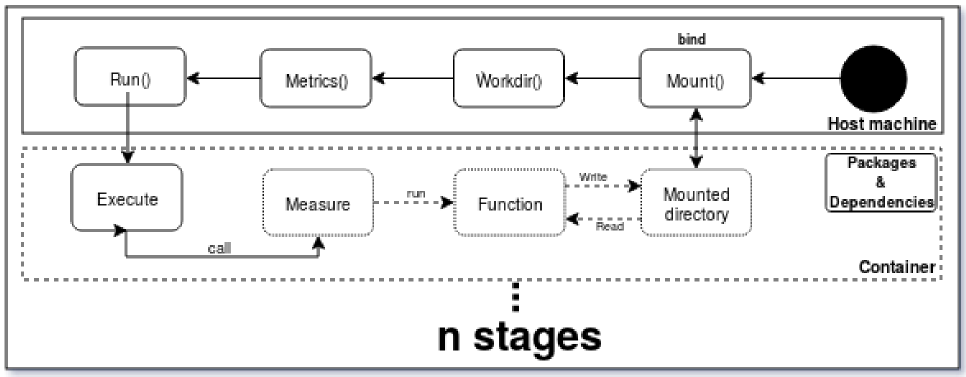


Figure 5: Execution flow Docker

shared storage driver.

This enables us to pass the processed documents from a container to another in a consistent manner and enables us to define inter-dependencies between containers, such as module B can only be executed if and only if module A precedes. In the aforementioned we bring forth four main steps, that is mount, workdir, metrics and run. Each representing instructions specifying which data repository to mount, designation of the working directory (specified as the absolute path of the function), metric function (extracting the required metrics for analysis purposes) and the execution command (containing the required arguments for each module) of each respective container. As soon as the execution flow is initialized, each step will propagate a series of actions that will shape up the intended workflow and will initiate the execution of the specified modules in a containerized environment as foreground processes. After each execution, the container cache will be cleaned up along with the removal of its file system, expect the removal of inherited mounted volumes such as the specified data repository located on the host machine. With the clean-up and removal of the unwanted file system, the life-cycle of a Docker module ends such that the following inter-dependent module can be initiated until the workflow is fully executed. We will continue by depicting the design of the Singularity container build and execution process, while similar to that of the docker workflow, it contains multiple design differences that will impact the method in which the container materialize.

4.3 Singularity workflow

In Figure 6 we illustrate the build process of the singularity flow, while at hindsight has a similar layout with that of the aforementioned docker build flow (see Figure 4), it follows a different build pattern that will be described below. As soon as the build process is initiated by using the *singularity build* command, we specify each image as an argument before the execution of the writable

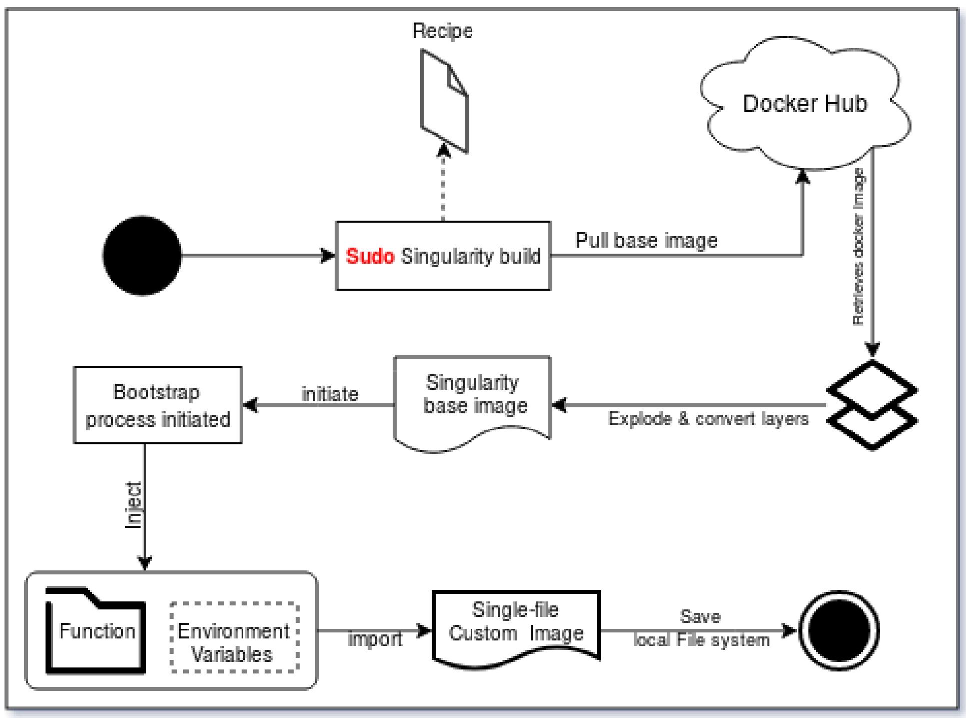


Figure 6: Build flow Singularity

images. The writable aspect of the images will enable us to perform read/write operations and conduct changes within the container without having the restriction of a read-only image. In order to execute a singularity build command as a writable ext3 single image that requires a Singularity recipe file (See section 3.2.2), the command needs to be carried out with escalated privileges (root) for the installation of software packages and dependencies. As soon as the command is initiated, Singularity invokes the get image function by initially specifying the required URI (In our case we pull an image from a public Docker image registry with the *docker://* prefix) and the image name along with the version number which is parsed through by making use of regular expression look-ups. Once this command is executed, the underlying Singularity application will initially retrieve all the layers specified in the base image and will gradually start assembling all layers into one single ext3 singularity image (See section 3.2.2).

After the process of retrieval and conversion has been successfully realized, the bootstrapping procedure of the container will be initialized. This is done, similar to that of the dockerfile approach, by populating the image through the instruction provided by the singularity recipe file. The execution of the instructions specified in the *%files* and *%post* section of the recipe will be carried out from inside the containers such that each module will be contained with its specific packages, files and dependencies. As soon as each instruction is carried out successfully, the state of the container will be stored as a custom image file containing the specified Linux distribution, files, meta-data and variables required for each respective module. The build flow of Singularity containers ends by storing the image into the local file system as a single file image that contains a virtual file system tailored for each module. In Figure 7, the execution flow of a Singularity strongly resembles that of the Docker execution flow specified above. The differences here are the different methods in which both mechanism mount the storage drivers, handle the execution of the functions or the techniques in which each mechanism allows us to spawn commands within the container instance (See section 3.2.2). The execution of both build and run flows will be carried out in a consistent

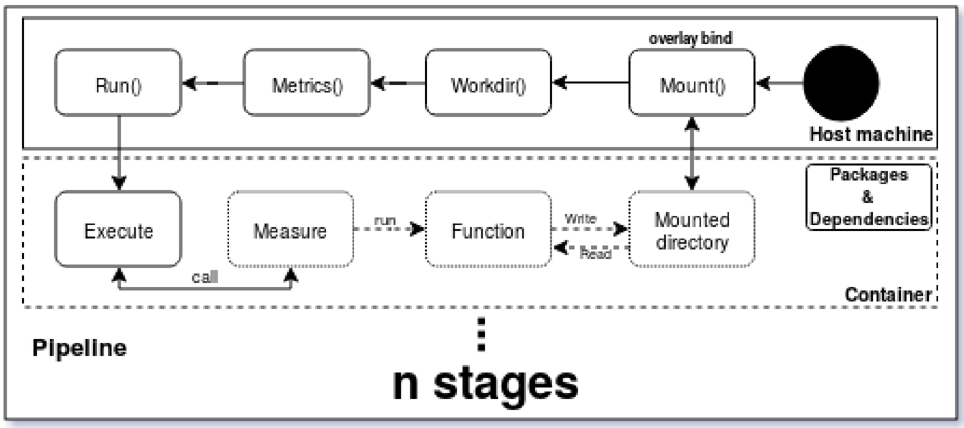


Figure 7: Execution flow Singularity

fashion such that each flow will be using default configuration values and the same configurations for the modules and Linux distributions. We will not take into account the time elapsed, resource usage or I/O operations undertaken when carrying out the build flow of both mechanism due to the fact that these figures do not represent the overall performance or applicability of such mechanism in scientific research. Hence, our focus is that of investigating the runtime executions of such mechanism, information such as I/O performance, CPU usage, memory usage and many other metrics will be extracted for the purpose of comparing their performance to that of the native one (bare-metal). Each experiment will be performed 50 times (Number of executions performed for each module), this is done in order to ensure the reliability and validity of the metrics extracted from all the workflows by aggregating all the results of the experiment executions. We avoided introducing any variations that would slow down or damage the integrity of our experiments, that would consequently damage the quality of our data and the contribution of our study.

In the following two sections we will expand on the specific use cases that were selected, along with the execution architecture of each stage. This section will provide the functional layout for our main contribution, here we will start expanding on specific widely-used application of machine learning in scientific research. This is achieved with the purpose of mimicking the Directed Acyclic

Graph (DAG) formulation of KNIME and WEKA (see Sub-sections 3.1.2 and 3.1.1) and to build upon on main objective of achieving reproducible and re-usable scientific workflows.

4.4 Experiment one: Machine-learned ranking using LambdaMART on Expedia Hotel Search data

The first use case has been retrieved from a Kaggle [14] challenge, namely the 2013 Expedia personalized hotel searches challenge [15]. The purpose of the challenge was that of optimizing the process of matching users to hotels, this was done by implementing a recommendation engine that would rank, for each user independently, the most likely hotel to be booked based on information such as hotel characteristics, location attractiveness, user's purchase history and click data. Hotel here refers to numerous establishments, examples of such can be hotels, apartments or B&Bs. The objective of the participants was that of ranking the expected user response that was represented by a click on a hotel or purchase of a room. A train and test are provided that have the size of 2.4 GB and 1.5 GB, respectively, but due to the nature of our experiments we have downsampled both datasets to 100,000 rows each. Participants are expected to submit a CSV (Comma Separated Value) file containing the searchId and propertyid sorted corresponding to the ranking of the entry. We choose this use case due to the popularity of recommendation engines among numerous industries and the interest in such systems in scientific research. The algorithm chosen by the winners of this challenge was either an ensemble of gradient boosting machines [62] or the LambdaMART [63] ranking algorithm. In our use case we decided to use the LambdaMART algorithm along with numerous pre-processing steps (Which will be described below) in order to make sure we mimic the exact steps of a machine learning workflow (training and prediction-wise) that would entail data ingestion, pre-processing of samples and training of the model on the preprocessed samples. Furthermore each of the processing stages will be segregated into separate containers to make sure that the integrity and preservation of the pipeline is kept that would enable the immediate re-usability of such modules. In Figure 8 we depict the main functionalities of our workflow that consist of three pre-processing steps and one training step.

The purpose and functionality of each step will be described in Table 3, for now the focus will be the interaction and requirements of each of the established steps. All steps make use of the Python version 3.5.2 programming language, while some of the steps share some specific package dependencies such as numpy or pandas, which are well-known scientific computing software packages,

[14] https://www.kaggle.com/

[15] https://www.kaggle.com/c/expedia-personalized-sort

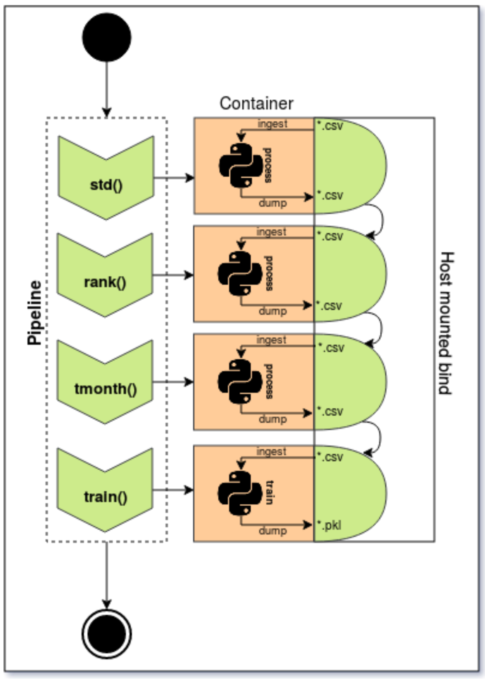


Figure 8: Experiment one execution flow

other steps come wrapped with software packages that are only required for that specific task, such as pyltr, which is a Python learn to rank toolkit[16] that contains ranking models, evaluation metrics and carries numerous other features. Each step follows a inter-dependent design pattern, where in order to achieve the desired outcome, the user will have to run the full pipeline as specified by the authors dependencies between modules or stages. The author in this situation refers to the publisher of the pipeline or algorithm, this is done with the aim of achieving exactly the result as it would be specified in the published paper. While other users or researchers can take advantage of such design in order to either build upon the work of their peers or retrieve the required modules defined in the pipeline and make use of it in different experiments or use cases.

Each component of the pipeline will be initialized with a shared storage driver, as seen in Table 3 after each component is executed it will dump the resulting object into the shared storage driver. This enables the users to maintain a consistent data flow between components as well as facilitating the ability to investigate the outcome of each module independently, similar to that of KNIME(refer to section 3.1.2).

[16]https://github.com/jma127/pyltr

Stage	**Description**
std()	Function that aggregates the values of multiple columns and maps the median,standard deviation and mean value of each aggregate group in three new columns
rank()	Function that ranks the relationship between sets of elements and controls how ranks are assigned to equal values
tmonth()	Function that converts the visit month of users from a categorical one to a numerical value
train()	Function which splits the data set into training and testing samples on equally distributed search ids, generates the target label and fit the LambdaMART model(*Parameters: number of boosting stages(estimators) = 150, fraction of queries (queries_subsample) = 0.5, max tree leaf nodes (max_leaf_nodes) = 15, min number of values required to be a leaf node (min_samples_leaf) = 64, Maximum depth of regression estimators (depthmax_depth) = 6*)

Table 3: Stage definition - Expedia Kaggle challenge

4.5 Experiment two: A Simpler and more generalizable story detector using verb and character features

The second and last selected use case activates in a different sub-field of machine learning than the previously specified use case, namely, computational linguistics or more specifically, Natural Language Processing (NLP). Natural language processing is a sub-field of artificial intelligence that is concerned with the interaction or reasoning between the computer and natural languages, such as the English language. Here the focus would be that of applying NLP-techniques for the detection of stories in a paragraph. A story here is defined as a set of one or more actors that have taken an action that would result into an outcome. Automatic story detection overcomes the human-driven methods by automating the process in which stories or events can be discovered in news articles or social media posts. The usage of story detector systems is required when users or journalists are concerned with the detection or parsing of relevant parts of the text in large corpus. While otherwise a time-consuming task, the aim of these systems is that of accurately identifying these story blocks within large corpus or paragraphs and mitigate the risk of missing vital information about the subject at hand.

While there have been many efforts [64, 65] in conceiving an automatic story detector that would accurately identify a story, the detectors developed would not capture the essence of the stories. More specifically, the lack of a more generic approach led to a situation where once a story detector would be used on a different corpus other than the one that it has been trained, users would notice a high performance degradation and a lower model accuracy when identifying sto-

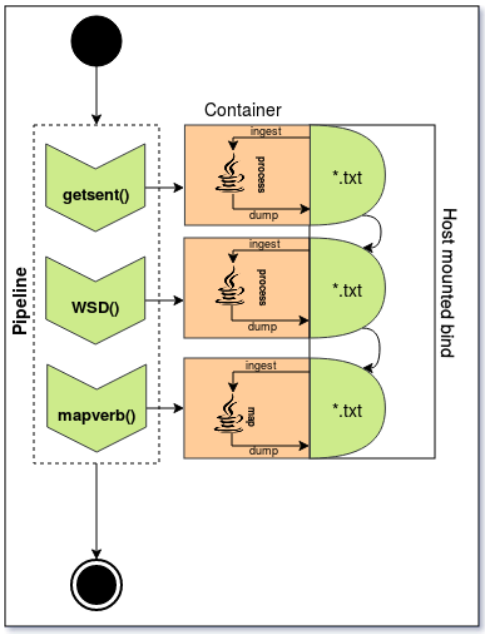


Figure 9: Experiment two execution flow

ries [66]. [66] proposes a state-of-the-art story detector by leveraging features that focus on events involving characters and actual characters themselves, namely verb and character features. In our experiments, we employ [66] story detector as one of our use cases, here we plan on segregating each of the main components of the developed story detector into independent working blocks. In order to do so, we initially look up the components that can be re-used in different contexts, this helps us determine which component will benefit more if isolated from the others. The main idea behind our approach is that of allowing each component to activate independently from each other, such that (as described at the beginning of section 4) each of the aforementioned components can be defined as module in our experimentation phase. As observed in Figure 9 we divided the system into three main components that are going to be utilized in our experiments. As mentioned previously, the setup of both use cases will be somewhat similar, the only main factors differentiating them from each other is the content of each container, dependency-wise and functionality-wise. The second use case follows a Java implementation, thus each component that is described below will be compiled during the execution of the program. In order to successfully embed each component with its own module, multiple separation of the system were required. This is due to the fact that each major component made use of a set of software libraries that were required once the files would be compiled and executed. Hence, we have identified 3 major components in the story-detector system, the first component is in charge of sentence splitting, annotating and tokenizing the raw text that returns an array list of split and annotated sentences. The second component will use the first components output sentence to pass it through the *it makes sense* Word Sense Disambiguation(WSD) system [67] such that an array of wordnet sense keys is returned for each predicate. Here in order to use the WSD system to be used in combination with WordNet key mapping, the Java WordNet Interface (JWI) [68] and Java VerbNet Interface [17] were used to facilitate this interaction. Furthermore, The resulting key mapping will be piped into the second component that given a wordnet synset key, returns an array list of the corresponding verbnet classes. As illustrated in table 4,

Stage	**Description**
getsent()	Given a text in String format, it returns an array list of sentences that are separated and cleansed.
WSD()	Given a split sentence it returns the wordnet sense keys for each token
mapverb()	Given a wordnet synset key, return an array of the corresponding verbnet classes

Table 4: Stage definition - Story detector

the three major components depicted for the story detector can be seamlessly integrated with each other in order to achieve the desired story detector system. This enables researchers and developers to tap into the ability of container mechanisms depicted previously, hence, the components can function on an independent level as well as a group level by allowing the usage of a shared storage driver. The shared storage driver here, that is shared across components, activates as a medium between each component such that each containers can make use of a persistent file system directory or storage. Thus, creating a pipeline that can makes use of one or more sources of data (Distributed file system, file system or external storage's) in such a way that it will not affect the integrity of the component itself, or more specifically, the integrity of the containers content and functionality. Furthermore, each component is expected to perform a specific action, which in our experiments would be that of satisfying the objective of each component. With this aspect in mind, which is briefly depicted in Table 4, the order and output of each component is utilized in a sequential formulation by adjusting the order of the components in the layout described by the author of the pipeline itself. This allows the users to experiment with a multitude of component combinations in their own custom order, but would only achieve fully reproducible results if the user does not introduce any changes in the order of the initial pipeline.

With the aforementioned specifications in mind, the layout of the experiments themselves represent the extent to which we validate the objective of our study, which is providing the ability to reproduce and re-use an end-to-end machine learning workflow. In order for this to be even considered as a valid solution for such systems or workflows, one criteria defined when measuring the success factor of our approach was the degree of performance overhead incurred by leveraging

[17] http://projects.csail.mit.edu/jverbnet/

such container mechanisms. The re-producability and re-usability aspect of our approach is that of attaining a platform-agnostic feature and low-level detail reproducible aspect of the system so that the entirety of the developed pipeline can be re-used either as a whole (which is in the exact same way the author of the pipeline intended the system to be used) or in a component-based formulation by users that are seeking only to reproduce *some* aspects of the system (that is, only the component that are necessary for the user). In this way, the advancement of research can be achieved in multiple ways, first being that of building on top of the groundwork that is laid-out by researchers that publish their end-to-end pipelines, and secondly, re-using only certain components of the system in order to derive a solution based on a combination of components extracted from other published pipelines and user-defined components. In the following Chapter we will start presenting the results of our aforementioned specified experiments that are going to be accompanied with the findings and setbacks encountered once implementing such pipeline.

5 Results

The initial assumptions behind our experiments were that of investigating whether container mechanisms are able to reshape the ways in which current machine learning workflows are designed. Here, we challenge the design of traditional workflow by implementing a container-based workflow that, in the context of our thesis, reduces the pitfalls encountered in empirical scientific research. One of the main pitfalls described previously in our study (check Chapter 1), was the vast amount of highly empirical studies that could not be reproduced due to the lack of or the inconsistent description of system specifications that has slowed down the advancements of research. Due to this aspect, our results, that are going to be depicted and discussed below, improve the overall portability and re-usability of machine learning workflows that require complex dependencies between various components of the system while also preserving the integrity of each component. Our thesis focuses on scientific publications that aim to publish their code and data in a private or public community and aim at facilitating the ability to reproduce their own results or experiments. That being mentioned, our study will focus on two aspects required for validating the reproducibility and applicability of our solution. To asses the reproducibility degree of our solution we decided to employ the framework and guidelines defined in [69], where the authors of the paper quantify the degree of reproducibility in computational biology publications. We will examine and determine the number of drawbacks mitigated by employing our solution, and we will depict the benefits of such in practice. Furthermore, in order to determine the applicability of our solution, we have laid out a number of experiments that allow us to measure the performance of such a workflow. By doing so, we take into account factors such as CPU usage, resident set size during its lifetime, context-switches and other resource indicators for each component in the workflow and compare these values with a system that is not virtualized (bare metal). The two main points behind our experiments are that of investigating:

- The extent to which our solution can solve the matter of unreproducible empirical work in scientific research
- The performance overhead of a container-based multi-component machine learning workflow

The granularity level chosen for our workflow enables the usage of a combination of modules that encapsulate the entirety of a components underlying dependency. Each component specified requires two or more arguments depending on the type of component, this varies based on the functionality of each specific component. However, while some components can be combined together to form an entire workflow, in some cases (see Sub-section 3) a component requires a specific data type as input such that the desired output can be achieved. This allowed us to segregate the functionality of one component in such a way that it can be used in combination with data sets or data sources that share the same specification traits.

Moreover, the usage of the Docker and Singularity manifest files (see Sub-section 3.2.1 and 3.2.2) allowed us to bootstrap the content of the containers by using a series of instruction that would populate the container during execution. This, in combination with the shared storage connected with the pipeline at runtime, allowed us to specify the path where each I/O operation would be initiated. Furthermore, in order to extract the required performance metrics for each component, such that it can be further analyzed, we make use of system calls (*time* and *getrusage*) to extract various time and resource-specific statistics about the life-cycle of the component.

Later on, in this chapter, we will describe the results of our experiments by making use of tables and figures that will contain the averaged values of each extracted statistic. We repeated the experiments 50 times such that we would increase the confidence and decrease the uncertainty of our estimations while also ensuring the precision of our extracted statistics; this is done by measuring the performance of multiple executions. Furthermore, based on all the extracted statistic we will reason the results in accordance with our initial assumptions and evaluate whether or not our proposed solution confers the right approach to the issues identified in Chapter 1 and Section 3.1.

5.1 Degree of reproducibility

To assess whether our solution is able to encapsulate numerous aspects of the system, we employ [69] reproducibility guidelines so we can determine how our proposed workflow is able to surpass and cover them. One of the major drawbacks in reproducible empirical research is the ability to reproduce an algorithm or system on a different environment as they are intimately tied to specific features of the operating environment [70]. This tends to lead to a series of internal compiler errors or runtime error caused by the unavailability of the environment, missing third-party packages or other conflicting underlying dependencies [71]. In Table 5 we depict the reproducibility guidelines extracted from a publication [69] that quantified various aspects of empirical research in computational biology to determine the most vital aspects when reproducing empirical research. Both of our workflows are based on container technologies, specifically Docker and Singularity containers. This design decision has enabled us to go beyond the ability of hypervisor-based virtualization or virtual environments and facilitate the ability to isolate processes and software dependencies in a thin virtualized without introducing the additional overhead entailed. However, the ability to

Table 5: Reproducibility Guidelines

Guideline	Description
Input data	The source of the original dataset utilized
Dataflow Diagram	Diagram that represents the steps in which the computations are performed.(This involves software tools, scripts and custom defined software that requires an input data source)
Software	Specification of the software tools utilized along with the specific version and source for all of the scripts or components in the workflow.
Configurations	Values and parameters utilized to execute the workflow
Intermediate data	Key intermediate data that resulted from important steps that would aid users determine whether they reproduced the method according to the author specification

reproduce algorithms or entire workflows does not entirely rely on the software packages themselves, but also on the underlying configurations and operating system compatibility. In Chapter 4, we have depicted the workflow proposed in a way that each component created is encapsulated in a virtual space. This feature allows users to contain the underlying software dependency and OS distribution of each component in a single file format that grants users the ability to export their workflows and capture the configuration parameters of each component. Not only does this maintain the integrity of the container itself, but it also accounts for the immutability of the objects inside the image itself once exported as a read-only image. Thus, a container image relies on the host OS kernel for implementing the container execution environment, isolation and resource usage based on the requirements of the image or manifest file. This allows users to achieve an easier and more descriptive method of defining and attaining the software and configuration guideline as specified in the table above.

As for the input and intermediate data guideline, our architecture design allows users to read and write from a wide range of data sources, file systems (LFS, HDFS, S3, EXT3, EXT4 and so forth) and web services. We decoupled the storage space from the component execution environment such that each component can ingest data and output artifacts into a shared mounted storage which is mounted in the container at runtime execution. The data flow diagram can be easily depicted based on the workflow itself, that is defined in *makefile* scripts that draws the relation and interdependency between each component in a DAG formulation. Hence, the design of our workflow provides users the ability to create reproducible workflows that captures the bare requirements for a component to function in a different computational environment. The modular aspect encourages users to envision their computational workflows in a more modular and decoupled formulation that is easier to reproduce and share among private or public communities.

Based on all the observation depicted above, we attain a high level of reproducability due to the possibility of easily exporting, version managing and securing the integrity of each component independently. Once an entire workflow is exported as multiple image-ready components, users have the ability to share, edit, or investigate the functionality of each container. Thus avoiding the VM approach of acting as a "black box" which does not allow users to perform any changes in the execution pipeline. By breaking down the pipeline into simpler, more de-coupled components we do not only capture the complete executable environment but also secure that the same software package version will be utilized across computational environments, thus avoiding running into missing third party packages issues. In the next section we will investigate the applicability of our workflow when faced with two machine learning use cases (Described in Chapter 4), this is done in order to determine the performance overhead implied when utilizing container-based mechanism as component in a workflow.

5.2 Performance indicators

In the first use case we made use of four main components that make up the entire work-flow, as specified in Chapter 4, the first three components serve as pre-processing modules, and the last component is where the model is trained and saved into the shared mounted storage. Each component makes use of multiple I/O bound operations; thus each component entails a degree of computation necessary for achieving the desired output. In the second use case; three main components were identified that make up the story detector pipeline. While both systems have similar layouts, the main factor that is differentiating between them is the underlying software functions and package dependencies.

5.2.1 CPU time

Below we have plotted the average CPU time for each component specified in the first use cases. In hindsight, we observe the results across the experiments when using the Docker-based or Singularity-based workflow show near or even better than native timings. Here we measure the User CPU-time for our running components, where for each component initiated we measure the duration the CPU was utilized for performing the computations and processing required. The CPU-time allows us to quantify the amount of time the system was actively working with the input as well as the amount of processing power delay acquired when achieving the desired outcome of a process which in our case is specified per component analysis.

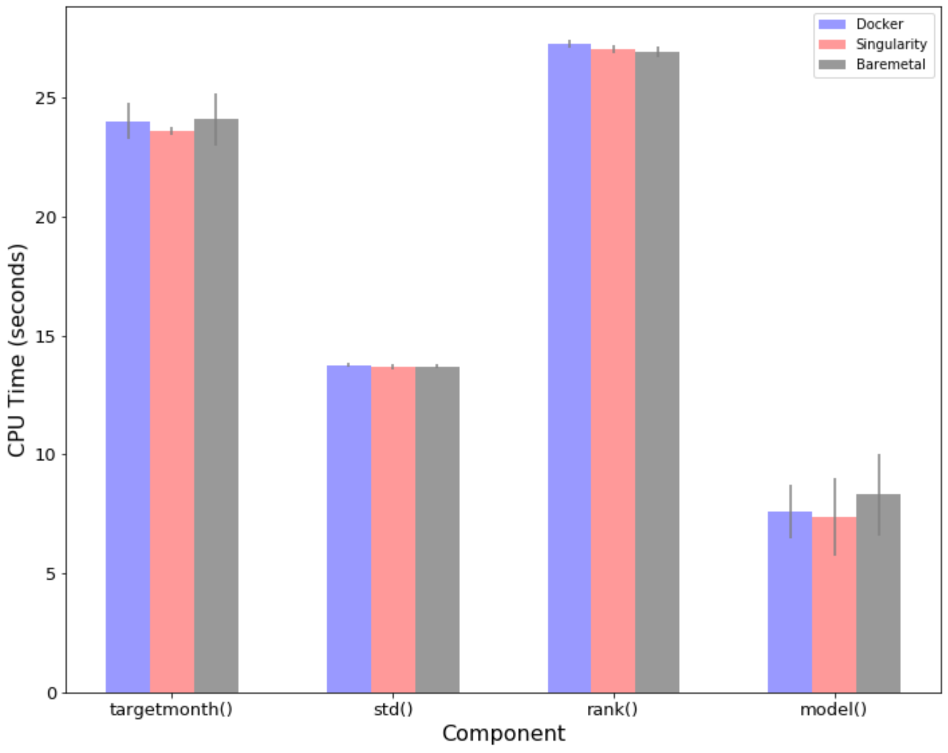


Figure 10: CPU time in the First use case

Each component was executed independently such that other components would not interfere with each other while performing the computations. In Figure 10, we observe the execution of the first pipeline execution where the main programming language that was used here was Python in combination with other third-party dependencies(*scipy*, *numpy*, *pandas* and so forth). We discover that the results extracted from the Singularity pipeline were able to achieve a slightly better overall performance than that of the Docker pipeline and the native execution. On a more independent performance of each component, we notice that singularity spends less time processing the data on tasks that require light-weight transformations, while at tasks that require heavy-weight transformations it performs either poorer or on-par with native performance. While the Docker pipeline in this experiment overall performance is the poorest among all the other methods. However, the Docker-based pipeline still exhibited near-native performance at most of the components specified in its work-flow.

We observe a higher variance in the standard deviation across components such as targetmonth and model of the pipeline, this may indicate the performance variations brought forth by map/apply functions and the early stop feature of the LambdaMART algorithm. Whilst the other two components indicate a steadier CPU time due to the functions we utilized to manipulate data structures in pandas.

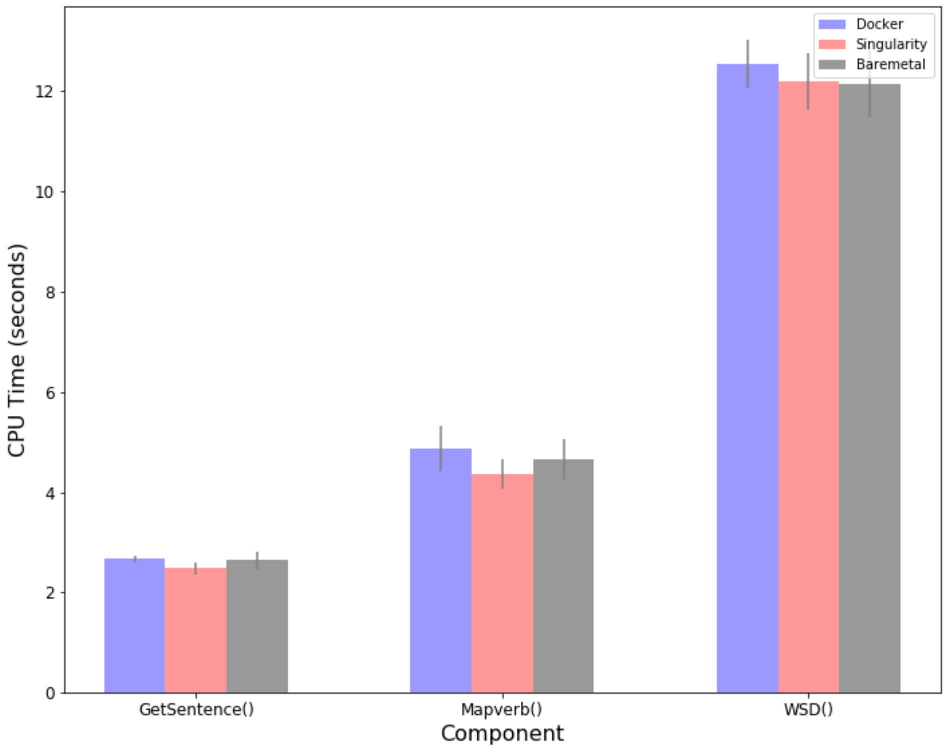


Figure 11: CPU time in the Second use case

Figure 11 shows the CPU-time average of the components in the second use case that is fully written in Java, unlike the previous Python written pipeline. Here, we notice similar performance indicators as in the previous use case, where the Singularity-based pipeline outperforms native, and the Docker-based pipeline in two out of three component and manages to reach near-native performance in the third and last component. Similarly, we notice that components that are computationally expensive (WSD component) tend to perform slightly poorer in container-based environments. The two aspects identified here are (i) Singularity-based solution executes the components faster than the Docker-based and native workflows. (ii) Singularity-based and Docker-based workflows encountered a slight overhead when dealing with computationally-heavy components.

Both of these aspects are argued as follows: (i) In both figures we observe that the Singularity-based solution managed to obtain a better overall performance than native or docker, this is caused by the hardware virtualization. Due to the design of the container mechanism, singularity containers does not fully emulate the hardware virtualization paradigm, with the exception of kernel namespaces environment. Hence, the creation of a singularity image yields high CPU performance due to the created encapsulated space that once a command is executed inside this space, it is essentially a single meta-data lookup. The single file images start-up, unlike docker, makes little to no use of cached data and the lack of a heavy implementation, performance costs of emulation and redundancy allows it to perform better than Docker. Furthermore, docker default CPU settings restrict the docker daemon from using the CPU for larger amounts of time, thus resulting into the additional overhead. (ii) The marginal overhead encountered in both docker and singularity when dealing with a heavy workload is caused by the implementation of kernel namespaces that add a slight performance penalty as the compiler requires to search for additional items. The WSD component makes use of mapping of each word to verbs that iterates over a massive amount of verb stored in dictionaries. This requires the system to parse through each file independently thus incurring a slight overhead when accessing each document in the file system.

5.2.2 CPU Utilization

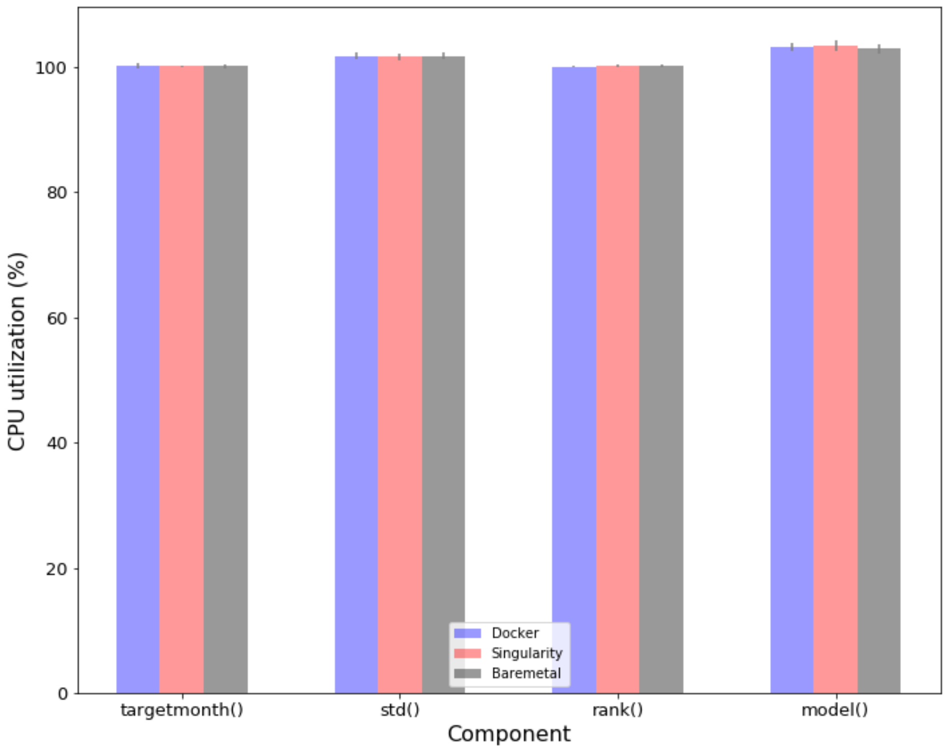


Figure 12: CPU utilization in the first use case

We plot the percentage of CPU that has been used by each component within both use cases. This estimation allows us to measure the system performance once executing a specific process which in our case is that of executing the functions (see Chapter 4) that are encapsulated in the respective container mechanism. The CPU percentage is calculated as follows:

$$\frac{U+S}{E} \tag{1}$$

Here S stand for the total number of seconds the component spent in kernel mode, U represents the total number of seconds the component sent in kernel mode, and finally, the E is the total real time duration spent executing the component. High CPU usage may indicate that the function itself (in our case component) is highly demanding CPU processing power in order to gracefully execute the function, while a lower CPU usage typically shows that the function itself wasn't as power demanding. This indicator helps us determine whether a running component is going to demand more CPU power than it would do when executed on the bare-metal host. As observed in Figure 12, we do not observe any unusual spike in the performance extracted from the first use case. One aspect that we have remarked in this is the brief increase in the first component; this can be caused by the hardware emulation and the fact that the component is executed through a single look-up file while in kernel mode, this allowed the component to access all memory locations and system resources.

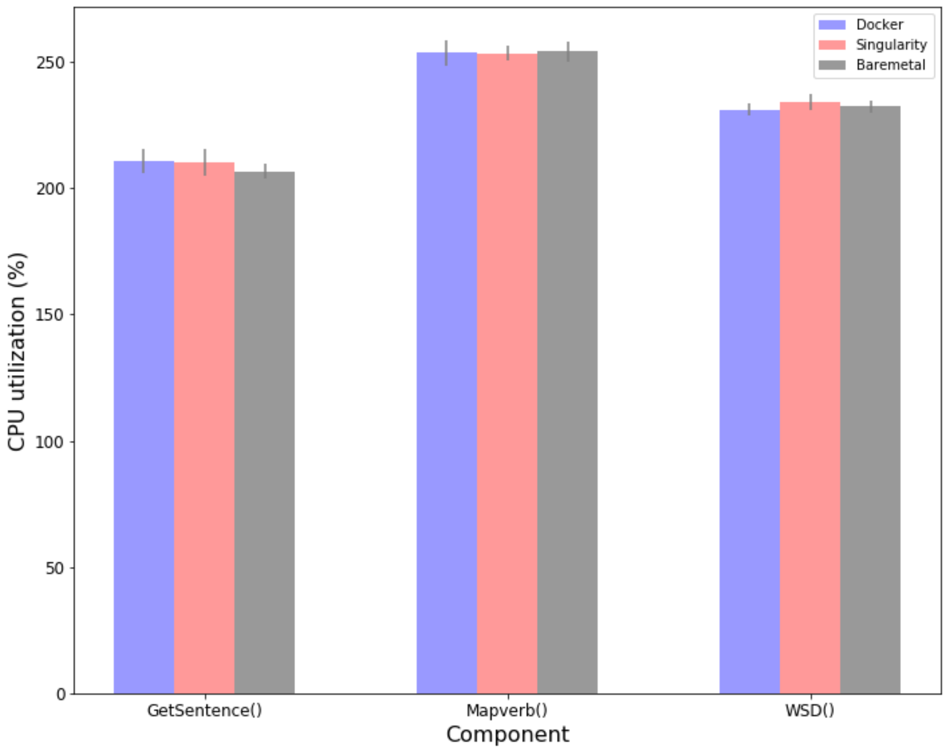


Figure 13: CPU utilization in the Second use case

As seen in Figure 13, the second use case shows strong multi-core utilization (multiple core system user during the experiments, see Chapter 4), while the previous figure only used up only half as much as this use case. The overall CPU utilization values do not vary highly when compared to the bare-metal environment with the container-based mechanisms. In this part of the metrics, we observe that the Docker and Singularity based workflow perform somewhat similar, having only encountered a slight overhead on the processing power itself when compared to the bare-metal environment. Hence, both implementations manage to achieve near-native performance as well as having the ability to deal with the same amount of multi-threaded workload as the bare-metal machine would.

5.2.3 Memory intake

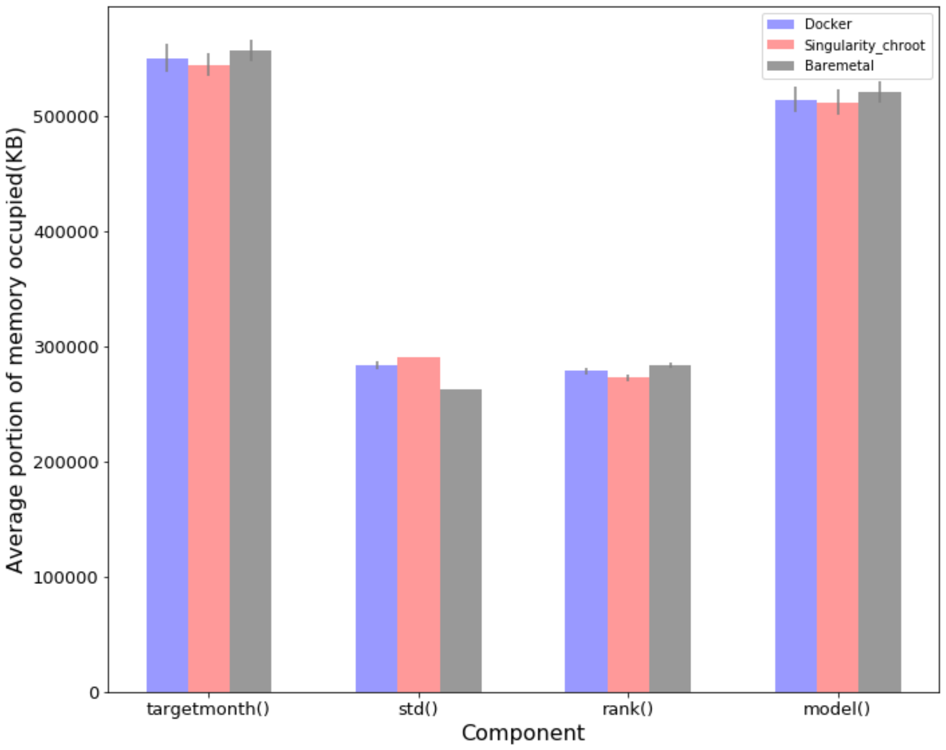


Figure 14: Memory intake in the First use case

The following figures show us the resident set size expressed in KB for each component in both use cases. The Resident Set Size(RSS) represents the maximum amount of main memory a process, or in our case component, has occupied in real RAM (not swapped) over its lifetime (where less is better). This accounts for the code, shared libraries, data and other memory types that are involved when executing a specific component. This metric will allows us to determine the amount of memory intake each component has consumed considering the similar layout across environments. Furthermore, the memory usage metric collected serve as an indicator on the memory deviation when compared to the bare-metal environment, here we investigate whether components occupies larger parts of the RAM when utilizing container-based mechanism. This metric is useful especially in machine learning; this is due to the reasonable amount of memory a machine requires to be able to handle large amounts of information. In the first use case, plotted in Figure 14, we deduce that the overall memory intake of all the components across environment are somewhat similar, no abnormal or memory spikes were registered when conducting our experiments. As in the previous experiments, the singularity workflow comes in first as the environment that utilized the least amount of memory, followed by the bare-metal environment and the docker environment coming in last. Another factor observed while conducting the experiment on this use case is the std component (That aggregates the values of multiple columns and maps the median,standard deviation and mean of each group, refer to Chapter 4), where the bare-metal environment manages to slightly overcome the Singularity and Docker work-flow in the matters of memory consumption.

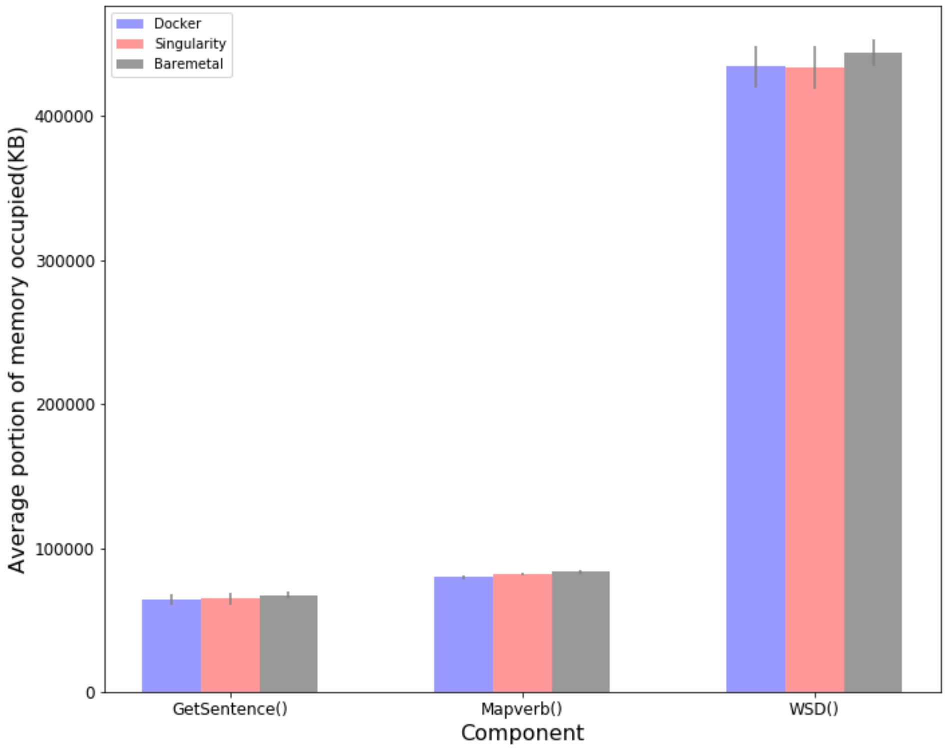


Figure 15: Memory intake in the Second use case

In Figure 15 we plot the maximum RSS values for the second use case, whereas observed previously the container-based workflows shows lower or similar values to that of a bare-metal approach, this shows us that container-based system does not add additional overhead to the system itself. The reason behind the container-based workflow consuming less memory in some task is due to the amount of code, shared libraries and data types located on the containers themselves. Where in comparison to that of bare-metal, it significantly reduces that amount of libraries needed to look-up when invoking a library. Another aspect observed here is that the Docker-workflow itself sometimes performs poorer than the Singularity-based one, this is caused by the go-routines that Docker itself spins up. The go-routines are spawned by *Golang* (the language Docker is implemented in) and its virtual memory usage tends to show high RAM allocation due to its pre-allocated stack space. However, accessing smaller files tends to yield better performance in Docker while larger files on the host system are inclined to perform better. Here singularity performs the best due to the escalated runtime required to build the sandboxed image environment of the function which was executed in kernel mode allowing access to all memory locations.

5.2.4 Context-switches

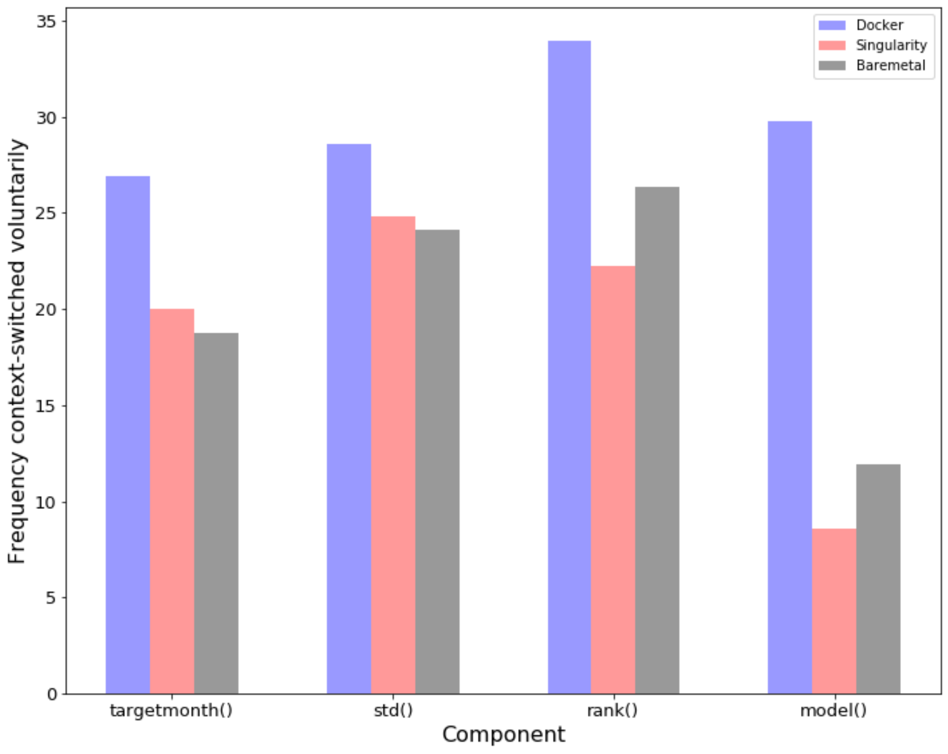


Figure 16: Context-switches in the First use case

Furthermore, we depict the frequency of context switches per component in both use cases. When managing multiple processes, an operating system switches context from one process to another, where a process here is defined as an executing instance of a program. The context here represents the content a CPU would register that provides fast executing memory within the CPU, that typically is used to quicken the execution of a certain process which in our case a component is made up from multiple processes. These context switch costs vary from process to process partly because the cost of executing the kernel code to do context switches is affected by cache performance [72]. Here we measure the context switches in the circumstances of our setup in order to ensure that there are not many context-switch when executing a process within a container that would entail additional CPU costs. Context switches can only occur in kernel mode via system calls; this will help us identify which container-technology makes use more of system calls when running specific tasks in a container-based setup. In Figure 16 we plot the voluntary (explicitly changing process thread) context-switches from the first use case, as seen the values extracted here highly vary from each other. Here the higher the frequency of context switches we observe the more computationally intensive that specific operation will be. In hindsight, we remark the highest frequency of context changes in the Docker-based workflow than the other two environments. Operations executed in the singularity-based environment made overall less frequent usage of context changes for the first two components while for the last two, the bare-metal environment achieved the least frequent context changes. Considering the CPU usage estimation from Figure 12, the first round of experiment (first use case) was mostly executed on a single thread in all three environments thus widely differing from the one from the seconds use, where we see (Figure 17) more frequent context changes.

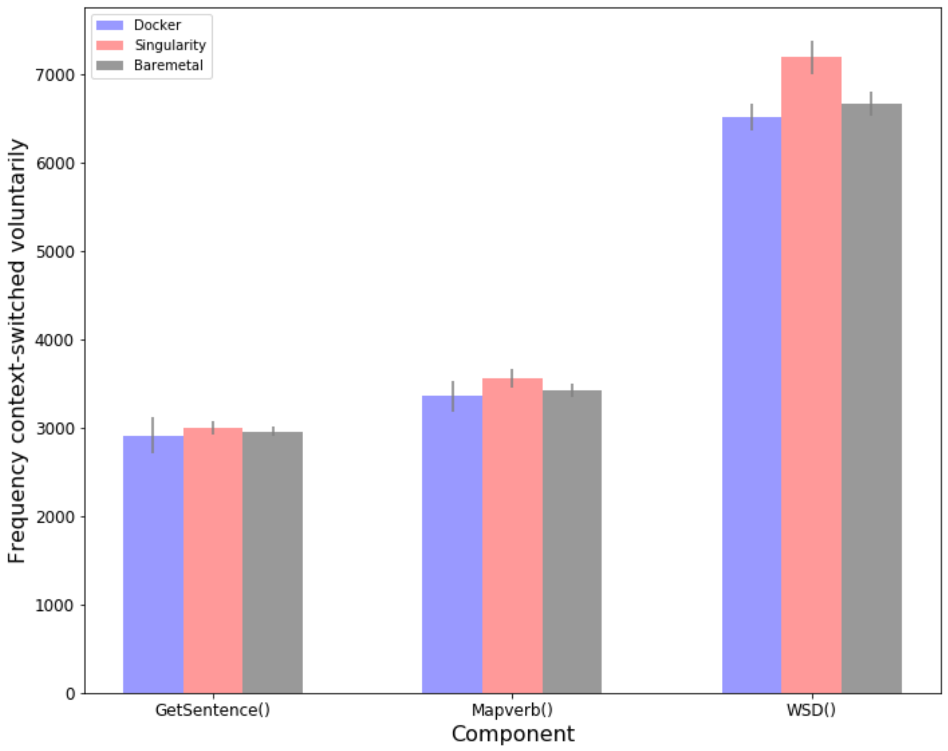


Figure 17: Context-switches in the Second use case

The reason behind the Singularity-workflow behaving as such, in the model and rank components, is that singularity does not support context changes; thus I/O operations flow directly between environments reducing the operational overhead and execution times(as seen in Figures 10 and 11). This also means that each operation executed inside the container continuously redirects all IO in and out of the container directly between the environments. Typically, the Docker-based workflow would yield the most frequent context changes due to the existence of the docker daemon itself, where for each container executed, the container itself is launched as a child root owned by the daemon, thus granting it access to kernel mode [46] (where all the context switches would occur). As briefly mentioned earlier in this paragraph, in Figure 17 we notice more frequent context changes, this is due to the fact that once a process requires the use of multiple processors it will make use of numerous threads, thus the context changes occur more often as a result of the process voluntarily relinquishing their time in the CPU [18].

5.2.5 Output operations

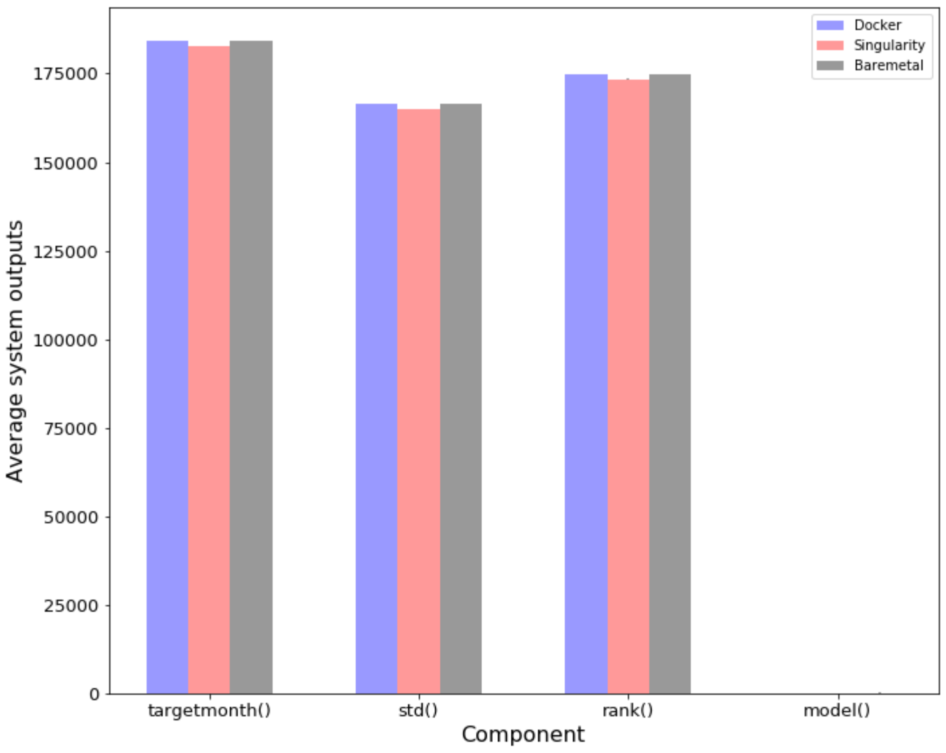


Figure 18: Output operations in the First use case

In the last round of extracted metrics, we plot the number of output operations written on the binded mounted storage across the container-based environments as well as the number of outputs in the bare-metal environment. The output transfer operations is represented by $\frac{B}{512}$ [19], where B stands for the number of bytes written to disk. Here each write performed on the mounted storage is continuously propagated to the host file system, this allows files that are written to the shared directory maintain consistency across environments. The initial write propagating from the initial

[18] http://www.linfo.org/context_switch.html

[19] https://github.com/torvalds/linux/blob/5924bbecd0267d87c24110cbe2041b5075173a25/include/linux/task_io_accounting_ops.h#L30

container file system, where Docker leverages the usage of OverlayFS and Singularity uses the extended file system (ext3) where each of these file systems brings forth a set of features that enables the usage of bind propagation towards the hosts' file system. The main point takeaway point we expect to get from this extracted metric values is whether the container-mechanisms themselves have an impact on the writing performance of a file (which can either be a model object, data set and so forth). This aspect is essential when dealing with a large amount of information being processed and written into the storage driver; this performance indicator will allow us to determine whether both container-mechanisms are able to manipulate the data itself quickly and efficiently. As observed in Figure 18, the system output between the various workflow environments follows somewhat similar values, where here the environment that achieved the lowest system outputs is the singularity-based workflow. However, the Docker-based workflow achieved on-par write per-

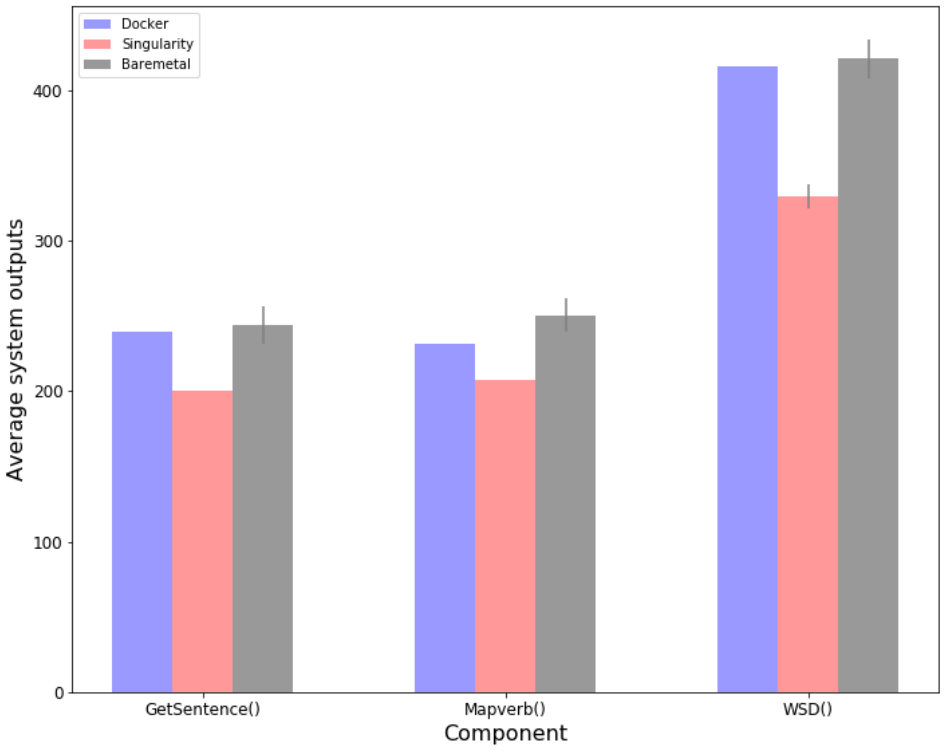


Figure 19: Output operations in the Second use case

formance with the bare-metal environment which can be due to the underlying mechanism of the OverlayFS where near-native write performance is achieved because the mount itself bypasses the storage driver and does not incur any potential overhead that would typically be encountered when dealing with thin provisioning and copy-on-write operations. In Figure 19, we observe a different behaviour from that of the first use case, where the Docker-based workflow manages to achieve better write performance than that of the bare metal run across all component. Moreover, We remark that the singularity-based workflow remains the environment that achieves the lowest output operations among the three environments. The speed that the singularity-based workflow exhibits here are due to its file system implementation of ext3 that allows the component to achieve an overall better I/O performance than other file system types. However, the Docker-based workflow own Overlay file system while achieving a high overall performance brings its own set of impediments when handling I/O operations in large files. The downside when an application or a process attempts to write a new value to a file on the file system is that every value written must copy on write up the file from the underlying image. When this occurs, the overlayFS storage driver will start searching for each image layer for the file, where the look-up order here is from the top layers to bottom layers. As soon as the file is located, it is entirely copied onto the top writable layer of the container itself. The existence of the copy-up and write-up operations leads to a higher latency because the overlayFS would perform these operations on the *entire* file itself, regardless of the fact that only *part* of the file is modified or its corresponding file size. Due to this aspect, the Docker-based workflow performs poorer than that of the singularity-based one. Singularity outperforms all other environments due to its native ability to redirect all IO in and out of the container directly between the environments reducing the operational overhead significantly.

5.2.6 Singularity container image format

Another aspect explored throughout our experiments is the various types of singularity images that can be produced by Singularity. The three types defined[20] are:

- **ext3**: Format that allows the users to interact openly with the container environment allowing changes to be made inside the environment, presents itself as a single image file.
- **squashfs**: Production-ready image, presents itself as a single compressed immutable read-only image file.
- **chroot directory**: Sandbox environment that allows the creation of a container as a writable directory for development purposes.

As seen from above each of the aforementioned types are specifically tailored for the environment desired the be created by the user. Users can convert the images from one format to the another freely so once a user has fully developed the component itself they can convert the image into an immutable production ready image. One aspect that we investigated with the various types of images was their performance when faced with our first use case (see Table 3 for specific details

[20]`https://singularity.lbl.gov/docs-build-container`

on the functionality of each component), this was done in order to determine whether there are significant performance differences between the various image formats. After executing 50 rounds of experiments with our first use case, we noticed that in most of the metrics extracted from them there was no statistical significance between the different types.

Table 6: Singularity-based workflow performance for each image type

Component	Type	Elapsed seconds	Context switch
targetmonth	ext3	24.25	873.51
	squashfs	24.53	5835.38
	chroot	24.04	19.98
std	ext3	**13.85**	836.88
	squashfs	14.53	6161.18
	chroot	14.09	24.84
rank	ext3	27.40	1343.51
	squashfs	27.92	9680.30
	chroot	27.25	22.26
model	ext3	8.01	1548.55
	squashfs	8.80	11117.62
	chroot	7.65	8.56

However, we have noticed two metrics that were affected by the change in type, namely the elapsed time and frequency of context-switches. As observed in Table 6, the context-changes vary highly from one type to another, this is due to the number of times the component itself has to switch from operating in kernel-model to user-mode that is caused by the immutable degree of the materialized image. Thus, this requires the component to execute many more system calls in order to achieve the desired computations. Each of the image types specified brings forth a specific degree of immutability that requires the executed instance of that image to perform more system calls the higher the degree of immutability. However, as specified previously in this section, as soon as a component increases the frequency of context-switches that is required the higher the computational cost to perform that task. This can be observed in the table above, where once a component is executed across the three different types, it tends to yield lower execution times once a component requires less frequent context-changes. Another remark that we identified in our experiments is a faster execution time once the ext3 image type is combined with the *std* component execution, this is due to the fact that the *std* component outputs an entire dataset that will be used by the remaining components. Here the execution time is due to the I/O performance of the ext3 filesystem implementation of Singularity, that allows for an improved write operation when heavy-write operations are required.

6 Future Work

We plan on extending the currently developed workflow to different use cases as well as extending its current functionalities and features. First, in order for our workflow to take full advantage of the HPC ecosystem, we plan on integrating our workflow implementation with currently available third-party software as well as integrate data provenance for each of the component designed. Third-party software that can be comprised of;

- Resource negotiators that would take care of the resource allocation of each component on a larger scale.
- Job schedulers that would ensure a coherent execution order and would reduce the interference among all the running jobs in the computing environment.
- Container orchestrator that would take care of the scaling up/down and life-cycle of containers in multi-container applications.
- Monitoring system that would extract the metrics/outputs of each executed component which would be further stored and visualized in a graphical user interface. As well as the ability to take actions or alert sysadmins of certain metrics based on the degree of resources allocated of each job.
- Private image and package repository that would keep track of the lineage of the utilized packages and images in each experiment.
- and other relevant third-party HPC tools based on the environment.

Each of the mentioned items allows the proposed workflow to utilize the most popular frameworks and tools within the HPC community. This brings system administrators and users the benefit of overseeing the resources and jobs that are currently executed in containers effortlessly.

Furthermore, throughout our study we have intentionally omitted the aspect of provenance within our designed workflows. We plan on embedding a data provenance image layer for each component type, this allows our implementation to extract user-specific meta-data as well as maintain a digitally signed record of every entity, activity and person involved in the creation of each workflow. By implementing such feature, users can attest the validity of each component such that there will exist a clear distinction between each contribution. This aspect will not only facilitate users the ability of modelling various workflow specifications but also enable the usage of legally binding signatures [14] that would further certify the validity of each created component. The data provenance layer would allow users to determine the inter-dependent relation [13] (BranchA, TaskA1, TaskA2, ParamaterA1, PredecessorA1, EventsA1 and so forth) between each action and entity in the order originally depicted by the author itself. A workflow management system would allows us to determine the data provenance and information flow of each action created within the workflow as well as automatically generate workflow templates [15] which would encapsulate execution-independent specifications of each task.

Secondly, we plan on employing our workflow in a medical use case that leverages exascale learning in medical image data. The use case aims at improving the currently utilized state-of-the-art algorithms in cancer diagnostics and medical treatment planning by using a multitude of data sources and modeling techniques throughout numerous computational centers. On a high-level overview the workflow is separated in multiple stages; initially high-resolution patches are extracted from raw Whole Slide Images(WSI), which are surgical tissues or biopsy images retrieved by means of a high-resolution scanner. The patches are extracted from normal tissue or tumor Region Of Interest (ROI) that were previously annotated by field experts. These patches will serve as the target variable during the statistical modeling phase. Once each image has been processed, the next step specified is that of using a multitude of unsupervised an supervised algorithms that perform a series of model trainings. These models would be re-iterated on the next step by of the workflow by executing multiple training runs in order to address the model robustness and generalization.

Each of the above-mentioned steps is comprised of a multitude of pre-processing steps that are monitored and stored in a distributed fashion across nodes. Each depicted step will make use of numerous third-party frameworks or libraries required by the machine learning and deep learning algorithms utilized in this workflow. As an initial step towards identifying the number of components this use case will utilize we need to segregate each main component from each other such that we can encapsulate and distribute the image across the targeted environments without running into incompatibility or dependency conflicts. One of the primary requirements of this workflow is that of having to train the models in across different data centers and locations (private data not allowed to leave hospital data center) by using the same workflows across environments. This use case would provide us the opportunity of reproducing similar setups across different HPC environments and thus further validate the purpose of our study.

7 Conclusion

With this thesis, we initially unveiled the drawbacks encountered in reproducible empirical research that impacts the velocity at which research is advancing nowadays. This phenomenon has slowed down the advancements in empirical research and has created a series of distrust in publications that propose state-of-the-art or novel solutions. After establishing the main problem statement, we went on and identified the current limitations that scientific workflow platforms are confronting with reproducible sciences that are concerned with highly empirical work. This has allowed us to determine the current limitations encountered with the currently available scientific workflow platforms. Even though these platforms are designed with the concept of "Write once, Run anywhere", they still encounter a significant amount of difficulties(see Chapter 3) when attempting to reproduce a workflow across different environments. Traditionally, in order to reproduce a similar execution environment, system administrators have to constantly integrate an ever-changing spectrum of tools and dependencies that raise a series of software conflicts in shared multi-tenant computing environments.

With the interest of overcoming this drawback, we proposed a simplistic yet generalizable approach to reproduce machine learning workflows in a more effective fashion in which we capture and encapsulate low-level atomic changes and configurations. In order to achieve this goal we initially made us of two container-based virtualization mechanisms that allowed us to segregate and wrap each component within the workflow into a thin virtualized layer. This layer has not only allowed us to break down the workflow into multiple blocks, but it has also accelerated the process of spinning up complex environments without interfering with the dependencies of the host system. One of the main concerns that we aimed at solving with our solution was that of overcoming the limitations identified in section 3.1 with traditional workflow platforms. Our proposed container-based workflow successfully eliminates the need for connectors that bridge the gap between programming languages by abstracting away each component in an enclosed virtual space. The usage of containers has also eliminated other drawbacks such as the need for instruction level emulation or JIT compilation, "dependency hell", and the portability of these workflows across environments.

In our experiments, we made use of two popular container-based mechanisms when implementing the workflows namely, Docker and Singularity. Both of the mentioned container mechanisms are tailored for either industry usage or scientific applications respectively. With this aspect in mind, we have implemented two separate workflows where each makes use of one of the previously mentioned container-based mechanisms. This was done to determine whether our proposed solution would satisfy the applicability requirement in practice. This requirement was achieved by means of performance testing our workflows with two machine learning use cases. The two use cases comprised of one use cased that was utilized as a story detector in large corpora while the second use case was a recommendation engine utilized to recommend the most likely hotel to be booked. The results of our experiments show that both of our container-based workflow implementations perform on a near-native level, while the singularity-based workflow overcomes both native and the Docker-based workflow in most cases. This shows us that our proposed singularity-based workflow is able to overcome most of the limitations encountered in reproducible sciences while also achieving high performance when tested against two highly-empirical use cases.

To conclude, our solution overcomes the current limitations and drawbacks encountered in reproducible sciences as well as the shortcomings of widely-used scientific workflows by providing users the ability to break-down workflow components into reproducible blocks. This approach has been proven to be promising based on our experiments and has shown that traditional workflows are able to evolve into a more modular-based concept in order to achieve a higher degree of reproducibility with less effort. The ability to share and port workflows across environments allows scientists and researchers to build upon the previously established work without requiring them to stumble over incompatibility or dependency conflicts which would traditionally slow down the advancements in research.

References

[1] Sören Sonnenburg, Mikio L Braun, Cheng Soon Ong, Samy Bengio, Leon Bottou, Geoffrey Holmes, Yann LeCun, Klaus-Robert Müller, Fernando Pereira, Carl Edward Rasmussen, Gunnar Rätsch, Bernhard Schölkopf, Alexander Smola, Pascal Vincent, Jason Weston, and Robert Williamson. The Need for Open Source Software in Machine Learning. *J. Mach. Learn. Res.*, 2007. ISSN 1532-4435. doi: citeulike-article-id:11849756.

[2] Ted Pedersen. Empiricism Is Not a Matter of Faith. *Computational Linguistics*, (3):465–470. doi: 10.1162/coli.2008.34.3.465.

[3] Dr. Chris Drummond. Replicability is not reproducibility: Nor is it good science. *Proceedings of the Evaluation Methods for Machine Learning Workshop 26th International Conference for Machine Learning*, 2009.

[4] John P A Ioannidis. Why most published research findings are false, 2005. ISSN 15491277.

[5] Alex Guazzelli, Michael Zeller, Wc Lin, and G Williams. PMML: An open standard for sharing models. *The R Journal*, 1(1):60–65, 2009. ISSN 2073-4859. doi: doi=10.1.1.538.9778.

[6] Joaquin Vanschoren and Hendrik Blockeel. Experiment databases. In *Inductive Databases and Constraint-Based Data Mining*. 2010. ISBN 9781441977373. doi: 10.1007/978-1-4419-7738-0_14.

[7] Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl, and Luis Torgo. OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60, 2014.

[8] Patrick Vandewalle, Jelena Kovacevic, and Martin Vetterli. Reproducible Research in Signal Processing - What, why, and how. *IEEE Signal Processing Magazine*, 2009. ISSN 10535888. doi: 10.1109/MSP.2009.932122.

[9] Yann Renard, Fabien Lotte, Guillaume Gibert, Marco Congedo, Emmanuel Maby, Vincent Delannoy, Olivier Bertrand, and Anatole Lécuyer. Openvibe: An open-source software platform to design, test, and use brain–computer interfaces in real and virtual environments. *Presence: teleoperators and virtual environments*, 19(1):35–53, 2010.

[10] John L Furlani, Sun Microsystems, Peter W Osel, and Siemens Components. Abstract Yourself With Modules. In *Proceedings of the 10th Conference on Systems Administration {(LISA} 1996), Chicago, IL, USA, September 29 - October 4, 1996*, number Lisa X, pages 193–204. ISBN 1-880446-81-2.

[11] Audris Mockus, Bente Anda, and Dag I.K. Sjøberg. Experiences from replicating a case study to investigate reproducibility of software development. In *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research (RESER 2010)*, page 5, 2010.

[12] I. Manolescu, L. Afanasiev, A. Arion, J. Dittrich, S. Manegold, N. Polyzotis, K. Schnaitter, P. Senellart, S. Zoupanos, and D. Shasha. The repeatability experiment of SIGMOD 2008. Technical Report 1.

[13] Ilkay Altintas, Manish Kumar Anand, Daniel Crawl, Shawn Bowers, Adam Belloum, Paolo Missier, Bertram Ludäscher, Carole A. Goble, and Peter M. A. Sloot. Understanding collaborative studies through interoperable workflow provenance. In *IPAW*, 2010.

[14] Michael Gerhards, Sascha Skorupa, Volker Sander, Adam Belloum, Dmitry Vasunin, and Ammar Benabdelkader. Hist/plier: A two-fold provenance approach for grid-enabled scientific workflows using ws-vlam. *2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 224–225, 2011.

[15] Michael Gerhards, A. Belloum, F. Berretz, V. Sander, and S. Skorupa. A history-tracing xml-based provenance framework for workflows. *The 5th Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, 2010.

[16] Eibe Frank, Mark Hall, Geoffrey Holmes, Richard Kirkby, Bernhard Pfahringer, Ian H Witten, and Len Trigg. WEKA: A Machine Learning Workbench for Data Mining. *Springer US. In: Data Mining and Knowledge Discovery Handbook*, pages 1305–14, 2005. ISSN 14337851. doi: 10.1007/0-387-25465-X_62.

[17] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. KNIME - the Konstanz information miner. *ACM SIGKDD Explorations Newsletter*, (1):26. ISSN 19310145. doi: 10.1145/1656274.1656280.

[18] Yolanda Gil, Varun Ratnakar, Jihie Kim, Pedro Gonzalez-Calero, Paul Groth, Joshua Moody, and Ewa Deelman. Wings: Intelligent workflow-based design of computational experiments. *IEEE Intelligent Systems*, 26(1):62–72, 2011.

[19] Mark A Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, (1): 10–18. ISSN 19310145. doi: 10.1145/1656274.1656278.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. ISBN 0-321-24678-0.

[21] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. ISBN 0201889544. doi: 10.1002/spe.4380180707.

[22] Mike Spivey. *The craft of prolog*, volume 17. 1991. ISBN 0-262-15039-5. doi: 10.1016/0167-6423(91)90046-Z.

[23] Barry Burd. Using Java Database Connectivity. *Java® For Dummies®*, (816):363–372. doi: 10.1002/9781118257517.ch16.

[24] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. Automated just-in-time compiler tuning. *Proceedings of the 8th annual IEEE / ACM international symposium on Code generation and optimization - CGO '10*, page 62. doi: 10.1145/1772954.1772965.

[25] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java® Virtual Machine Specification. *Managing*, pages 1–626.

[26] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster Computing with Working Sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 10, 2010. ISSN 03642348. doi: 10.1007/s00256-009-0861-0.

[27] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Sort*, pages 1626–1629. ISSN 2150-8097. doi: 10.1109/ICDE.2010.5447738.

[28] J. P. Walters, Vipin Chaudhary, Cha Minsuk, Salvátore Guercio, and Steve Gallo. A comparison of virtualization technologies for HPC. In *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, 2008. ISBN 0769530958. doi: 10.1109/AINA.2008.45.

[29] a Gavrilovska, S Kumar, and H Raj. High-performance hypervisor architectures: Virtualization in hpc systems. In *HPCVirt 2007*, 2007. ISBN 1595930906. doi: 10.1.1.108.4135.

[30] Andrew J. Younge, Robert Henschel, James T. Brown, Gregor von Laszewski, Judy Qiu, and Geoffrey C. Fox. Analysis of Virtualization Technologies for High Performance Computing

Environments. In *2011 IEEE 4th International Conference on Cloud Computing*, 2011. ISBN 978-1-4577-0836-7. doi: 10.1109/CLOUD.2011.29.

[31] M G Xavier, M V Neves, F D Rossi, T C Ferreto, T Lange, and C a F De Rose. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, (LXC):233–240, 2013. ISSN 1066-6192. doi: Doi10.1109/Pdp.2013.41.

[32] M. Raho, A. Spyridakis, M. Paolino, and D. Raho. Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing. In *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pages 1–8, Nov 2015. doi: 10.1109/AIEEE.2015.7367280.

[33] Jack Li, Qingyang Wang, Deepal Jayasinghe, Junhee Park, Tao Zhu, and Calton Pu. Performance overhead among three hypervisors: An experimental study using hadoop benchmarks. In *Proceedings - 2013 IEEE International Congress on Big Data, BigData 2013*, pages 9–16, 2013. ISBN 9780768550060. doi: 10.1109/BigData.Congress.2013.11.

[34] Stephen Soltesz, Stephen Soltesz, Herbert Pötzl, Herbert Pötzl, Marc E Fiuczynski, Marc E Fiuczynski, Andy Bavier, Andy Bavier, Larry Peterson, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, pages 275–287. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1272998.1273025.

[35] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

[36] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, Feb 2013. doi: 10.1109/PDP.2013.41.

[37] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. ISSN 1075-3583.

[38] Carl Boettiger. An introduction to Docker for reproducible research, with examples from the {R} environment. *CoRR*.

[39] Jürgen Cito, Vincenzo Ferme, and Harald C. Gall. Using docker containers to improve reproducibility in software and web engineering research. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9671, pages 609–612, 2016. ISBN 9783319387901. doi: 10.1007/978-3-319-38791-8_58.

[40] Nikyle Nguyen and Doina Bein. Distributed MPI cluster with Docker Swarm mode. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference, CCWC 2017*, 2017. ISBN 9781509042289. doi: 10.1109/CCWC.2017.7868429.

[41] H Zeng, B Wang, W Deng, and W Zhang. Measurement and Evaluation for Docker Container Networking. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 105–108, oct 2017. doi: 10.1109/CyberC.2017.78.

[42] Fawaz Paraiso, Stéphanie Challita, Yahya Al-Dhuraibi, and Philippe Merle. Model-driven management of docker containers. In *IEEE International Conference on Cloud Computing, CLOUD*, pages 718–725, 2017. ISBN 9781509026197. doi: 10.1109/CLOUD.2016.98.

[43] Congfeng Jiang, Dongyang Ou, Yumei Wang, Xindong You, Jilin Zhang, Jian Wan, Bing Luo, Weisong Shi, Congfeng Jiang, Dongyang Ou, and Yumei Wang. Energy efficiency comparison of hypervisors. *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, pages 1–8. ISSN 22105379. doi: 10.1109/IGCC.2016.7892607.

[44] Pengfei Xu, Shaohuai Shi, and Xiaowen Chu. Performance Evaluation of Deep Learning Tools in Docker Containers. *CoRR*.

[45] J Bhimani, Z Yang, N Mi, J Yang, Q Xu, M Awasthi, R Pandurangan, and V Balakrishnan. Docker Container Scheduler for I/O Intensive Applications running on NVMe SSDs. *IEEE Transactions on Multi-Scale Computing Systems*, PP(99):1, 2018. doi: 10.1109/TMSCS.2018.2801281.

[46] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLoS ONE*, 12(5), 2017. ISSN 19326203. doi: 10.1371/journal.pone.0177459.

[47] Andrew J Younge, Kevin Pedretti, Ryan E Grant, and Ron Brightwell. A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 74–81. IEEE, 2017.

[48] Vanessa V. Sochat, Cameron J. Prybol, and Gregory M. Kurtzer. Enhancing reproducibility in scientific computing: Metrics and registry for Singularity containers. *PLoS ONE*, 12(11), 2017. ISSN 19326203. doi: 10.1371/journal.pone.0188511.

[49] Vanessa V Sochat, Cameron J Prybol, and Gregory M Kurtzer. Enhancing reproducibility in scientific computing: Metrics and registry for Singularity containers. *PloS one*, 12(11): e0188511, 2017.

[50] Cristian Ramon-Cortes, Albert Serven, Jorge Ejarque, Daniele Lezzi, and Rosa M Badia. Transparent Orchestration of Task-based Parallel Applications in Containers Platforms. *Journal of Grid Computing*, 16(1):137–160, 2018.

[51] Klaus Rechert, Thomas Liebetraut, Stefan Kombrink, Dennis Wehrle, Susanne Mocken, and Maximilian Rohland. Preserving Containers. *Tage 2017*, page 143.

[52] Tristan Glatard, Gregory Kiar, Tristan Aumentado-Armstrong, Natacha Beck, Pierre Bellec, Rémi Bernard, Axel Bonnet, Sorina Camarasu-Pop, Frédéric Cervenansky, Samir Das, and Others. Boutiques: a flexible framework for automated application integration in computing platforms. *arXiv preprint arXiv:1711.09713*, 2017.

[53] Ákos Kovács. Comparison of different Linux containers. In *Telecommunications and Signal Processing (TSP), 2017 40th International Conference on*, pages 47–51. IEEE, 2017.

[54] Balazs Gerofi, Rolf Riesen, Robert W Wisniewski, and Yutaka Ishikawa. Toward Full Specialization of the HPC Software Stack. *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017 - ROSS '17*, pages 1–8. doi: 10.1145/3095770.3095777.

[55] Carlos Arango, Rémy Dernat, and John Sanabria. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. *arXiv preprint arXiv:1709.10140*, 2017.

[56] Jonathan Sparks. HPC Containers in Use.

[57] N Halbwachs, P Caspi, P Raymond, and D Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, (9):1305–1320.

[58] Gregory F. Pfister. An introduction to the InfiniBandâĎć architecture. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 617–632. 2001. ISBN 9780470544839. doi: 10.1109/9780470544839.ch42.

[59] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[60] EM Fajardo, JM Dost, B Holzman, T Tannenbaum, J Letts, A Tiradani, B Bockelman, J Frey, and D Mason. How much higher can htcondor fly? In *J. Phys. Conf. Ser.*, volume 664. Fermi National Accelerator Laboratory (FNAL), Batavia, IL (United States), 2015.

[61] Hidemoto Nakada, Atsuko Takefusa, Katsuhiko Ookubo, Makoto Kishimoto, Tomohiro Kudoh, Yoshio Tanaka, and Satoshi Sekiguchi. Design and implementation of a local scheduling system with advance reservation for co-allocation on the grid. In *Computer and Information Technology, 2006. CIT'06. The Sixth IEEE International Conference on*, pages 65–65. IEEE, 2006.

[62] Ping Li, Qiang Wu, and Christopher J Burges. Mcrank: Learning to rank using multiple classification and gradient boosting. In *Advances in neural information processing systems*, pages 897–904, 2008.

[63] Christopher JC Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

[64] Betul Ceran, Ravi Karad, Steven Corman, and Hasan Davulcu. A Hybrid Model and Memory Based Story Classifier. *Third Workshop on Computational Models of Narrative (CMN)*, 824: 60–64, 2012.

[65] Andrew S Gordon and Reid Swanson. Identifying Personal Stories in Millions of Weblog Entries. *Papers from the 2009 ICWSM Workshop, Data Challenge Workshop*, pages 16–23, 2009.

[66] Joshua Eisenberg and Mark Finlayson. A simpler and more generalizable story detector using verb and character features. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2708–2715, 2017.

[67] Zhi Zhong and Hwee Tou Ng. It Makes Sense: A wide-coverage word sense disambiguation system for free text. *Proceedings of ACL 2010 (System Demonstrations)*, pages 78–83, 2010.

[68] Mark Alan Finlayson. Java Libraries for Accessing the Princeton Wordnet: Comparison and Evaluation. In *Proceedings of the 7th International Global WordNet Conference (GWC 2014)*, pages 78–85. ISBN 978âĂŞ9949âĂŞ32âĂŞ492âĂŞ7.

[69] Daniel Garijo, Sarah Kinnings, Li Xie, Lei Xie, Yinliang Zhang, Philip E. Bourne, and Yolanda Gil. Quantifying reproducibility in computational biology: The case of the tuberculosis drugome. *PLoS ONE*, 2013. ISSN 19326203. doi: 10.1371/journal.pone.0080278.

[70] Jan Vitek and Tomas Kalibera. Repeatability, reproducibility, and rigor in systems research. In *Proceedings of the ninth ACM international conference on Embedded software - EMSOFT '11*, 2011. ISBN 9781450307147. doi: 10.1145/2038642.2038650.

[71] Christian Collberg, Todd Proebsting, and Alex M Warren. Repeatability and Benefaction in Computer Systems Research. *Communications of the ACM*, 2016. doi: 10.1145/2812803.

[72] Jeffrey C. Mogul and Anita Borg. The Effect of Context Switches on Cache Performance. In *ASPLOS'91*, 1991. ISBN 0897913809. doi: 10.1145/106975.106982.