UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1846

Enabling Scalable Data Analysis on Cloud Resources with Applications in Life Science

MARCO CAPUCCINI

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2019

ISSN 1651-6214
ISBN 978-91-513-0730-5
urn:nbn:se:uu:diva-390666

Dissertation presented at Uppsala University to be publicly examined in B42, Uppsala Biomedicinska Centrum, Husargatan 3, Uppsala, Thursday, 10 October 2019 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Docent Johan Tordsson (Umeå University, Department of Computing Science).

Abstract
Capuccini, M. 2019. Enabling Scalable Data Analysis on Cloud Resources with Applications in Life Science. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1846. 71 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0730-5.

Over the past 20 years, the rise of high-throughput methods in life science has enabled research laboratories to produce massive datasets of biological interest. When dealing with this "data deluge" of modern biology researchers encounter two major challenges: first, there is a need for substantial technical skills for dealing with Big Data and; second, infrastructure procurement becomes difficult. In connection to this second challenge, the computing model and business trend that was originally popularized by Amazon under the name of cloud computing represents an interesting opportunity. Instead of buying computing infrastructure upfront, cloud providers enable the allocation and release of virtual resources on-demand. These resources are then billed with a pay-per-use pricing model and physical infrastructure management is delegated to the provider. In this thesis, we introduce a number of methods for running Big Data analyses of biological interest using cloud computing. Considerable efforts were made in enabling the application of trusted, bioinformatics software to Big Data scenarios as opposed to reimplementing the existing codebase. Further, we improve the accessibility of the technology with the aim of reducing the entry barrier for biologists. The thesis includes 5 papers. In Papers I and II, we explore the applicability of Apache Spark, one of the leading Big Data analytics platforms in cloud environments, to two drug-discovery use cases. In Paper III, we present a general method for running bioinformatics analyses on the cloud using the microservices-oriented architecture. In Paper IV, we introduce a method that combines microservices and Apache Spark with the aim of providing the best of both technologies. In Paper V, we discuss how to reduce the entry barrier for the allocation of cloud research environments. We show that all of the developed methods scale well and we provide high-level programming interfaces for improving accessibility. We have also made the developed software publicly available.

Keywords: cloud computing, bioinformatics, Big Data, microservices, containers, MapReduce

Marco Capuccini, Department of Information Technology, Division of Scientific Computing, Box 337, Uppsala University, SE-751 05 Uppsala, Sweden.

© Marco Capuccini 2019

ISSN 1651-6214
ISBN 978-91-513-0730-5
urn:nbn:se:uu:diva-390666 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-390666)

To everyone who supported me throughout my studies

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **M. Capuccini**, L. Carlsson, U. Norinder, and O. Spjuth, "Conformal prediction in Spark: large-scale machine learning with confidence," in 2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC), pp. 61–67, IEEE, 2015.

II **M. Capuccini**, L. Ahmed, W. Schaal, E. Laure, and O. Spjuth, "Large-scale virtual screening on public cloud resources with Apache Spark," Journal of cheminformatics, vol. 9, no. 1, p. 15, 2017.

III P. Emami Khoonsari, P. Moreno, S. Bergmann, J. Burman, **M. Capuccini**, M. Carone, M. Cascante, P. De Atauri, C. Foguet, A. Gonzalez-Beltran, et al., "Interoperable and scalable data analysis with microservices: Applications in Metabolomics," Bioinformatics, btz160, 2019.

IV **M. Capuccini**, M. Dahlö, S. Toor, and O. Spjuth, "MaRe: a MapReduce-Oriented Framework for Processing Big Data with Application Containers," arXiv preprint arXiv:1808.02318v2, 2019.

V **M. Capuccini**, A. Larsson, M. Carone, J. A. Novella, N. Sadawi, J. Gao, S. Toor, and O. Spjuth, "On-demand virtual research environments using microservices," arXiv preprint arXiv:1805.06180v4, 2019.

Reprints were made with permission from the publishers.

List of additional papers

- J. Gao, N. Sadawi, I. Karaman, J. Pearce, P. Mereno, A. Larsson, **M. Capuccini**, P. Elliott, J. K. Nicholson, T. Ebbels, et al., "Metabolomics in the cloud: Scaling computational tools to big data," arXiv preprint arXiv:1904.02288, 2019.
- K. Peters, J. Bradbury, S. Bergmann, **M. Capuccini**, M. Cascante, P. de Atauri, T. M. Ebbels, C. Foguet, R. Glen, A. Gonzalez-Beltran, et al., "Phenomenal: processing and analysis of metabolomics data in the cloud," GigaScience, vol. 8, no. 2, p. giy149, 2018.
- L. Ahmed, V. Georgiev, **M. Capuccini**, S. Toor, W. Schaal, E. Laure, and O. Spjuth, "Efficient iterative virtual screening with Apache Spark and conformal prediction," Journal of cheminformatics, vol. 10, no. 1, p. 8, 2018.
- O. Spjuth, **M. Capuccini**, M. Carone, A. Larsson, W. Schaal, J. Novella, P. Di Tommaso, C. Notredame, P. Moreno, P. E. Khoonsari, et al., "Approaches for containerized scientific workflows in cloud environments with applications in life science," tech. rep., PeerJ Preprints, 2018.
- J. A. Novella, P. Emami Khoonsari, S. Herman, D. Whitenack, **M. Capuccini**, J. Burman, K. Kultima, and O. Spjuth, "Container-based bioinformatics with Pachyderm," Bioinformatics, vol. 35, no. 5, pp. 839–846, 2018.
- **M. Capuccini**, A. Larsson, S. Toor, and O. Spjuth, "KubeNow: a cloud agnostic platform for microservice-oriented applications," in 2017 Imperial College Computing Student Workshop (ICCSW 2017), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- S. Toor, M. Lindberg, I. Falman, A. Vallin, O. Mohill, P. Freyhult, L. Nilsson, M. Agback, L. Viklund, H. Zazzik, O. Spjuth, **M. Capuccini**, et al., "SNIC Science Cloud (SSC): a national-scale cloud infrastructure for Swedish academia," in 2017 IEEE 13th International Conference on e-Science (e-Science), pp. 219–227, IEEE, 2017.

Sammanfattning på Svenska

Inledning

Big Data är på mångas läppar nuförtiden. Det har utan tvekan att göra med den imponerande mängd data det moderna samhället är i stånd att producera. Biovetenskap har trätt in i Big Data-eran i samband med det stora prisfall på DNA-sekvensering som inträffade i slutet av 2000-talet. Om det på 90-talet kostade cirka tre miljoner amerikanska dollar att göra en DNA-sekvensering av en människas arvsmassa, kostar det nu inte mer än ca tusen dollar. Allteftersom kostnaden för DNA-sekvensering sjunker, har fler och fler forskningslaboratorier råd att använda teknologin. Eftersom sekvensering av DNA ger insikt inom många kunskapsdomäner–medicin, farmakologi, evolutionsbiologi och antropologi för att nämna några–har ett stort antal forskningsorganisationer på kort tid blivit tvungna att hantera och analysera stora mängder data. För att ge en bild av mängden data från DNA-sekvensering, tänk på att en människas arvsmassa kan representeras i omkring 750 megabyte, och att det krävs 30 gånger så mycket data för att få en tillräckligt pålitlig bild. Eftersom ett flertal forskningsprojekt parallellt kan sekvensera några tusen genom, är det enkelt att föreställa sig hur detta leder till en dataexplosion inom organisationer. Mycket riktigt så lagrar the European Bioinformatics Institute–en av de ledande organisationerna inom området–över 100 petabyte genomikinformation år 2018. Big Data inom biovetenskap är dock inte enbart begränsat till genetik. Biologisk information är påfallande heterogen och sträcker sig till andra domäner inom biovetenskap som närmar sig liknande Big Data-trender.

Att modern biovetenskap närmast svämmar över av data har minst två stora implikationer. För det första måste forskare inom biovetenskap idag kunna hantera enorma mängder data, och då biologi inte är ett IT-relaterat fält, upplever många svårigheter i att göra det. För det andra krävs fullgod IT-infrastruktur–data från de modernaste studierna kan helt enkelt inte hanteras på en forskares laptop–och framskaffande av detta blir en utmaning. I samband med denna andra implikation uppenbarar sig en intressant möjlighet– affärstrenden som från början populariserades av Amazon under namnet cloud computing. Istället för att köpa datorer och annan e-infrastruktur i förhand, låter molntjänstleverantörer dig initiera och frigöra virtuella resurser, t ex datorer, lagring, nätverk, på begäran. Dessa resurser betalas sedan utifrån användning och allt strul med att upprätta och underhålla den fysiska infrastrukturen överlåts åt leverantören. Då datorkrav varierar mycket under ett forskningsprojekt och då biologer inte ska behöva ansvara för att upprätta e-infrastruktur, är det en tilltalande modell för biovetenskap.

Mål

Det övergripande målet för arbetet som presenteras i den här avhandlingen är att möjliggöra Big Data-analyser av intresse för biologin genom användningen av molntjänster. Under arbetet angrep vi två huvudsakliga utmaningar. För det första, då biologiska data är mycket heterogena existerar ett vidsträckt landskap av programvaruverktyg som behöver anpassas både till cloud och Big Data. Programvaran som utvecklats med endast småskalig information i åtanke går inte att tillämpa på Big Data utan vidare. Dessutom, då biovetenskapliga analyser traditionellt utförs på högpresterande datorsystem, är programvara för bioinformatik vanligtvis utvecklad med denna modell i åtanke; att utföra biovetenskapliga analyser i molnet är ofta inte helt enkelt. För det andra krävs betydande IT-expertis för att behandla Big Data, och även om molnet överlåter hanteringen av den fysiska infrastrukturen, innebär modellen där virtuella resurser tilldelas på begäran ett stort inträdeshinder för biologer.

Resultat

I den presenterade forskningen har vi utvecklat och tillämpat metoder för att utföra analyser av biologiska storskaliga data med hjälp av molnteknologi. Stor energi lades på att göra det möjligt att tillämpa den heterogena, betrodda programvara som karaktäriserar biovetenskap, till Big Data-scenarier där molnteknologier används. Dessutom bidrog vi till att göra teknologin mer tillgänglig, så att inträdessteget för biologer ska bli lägre.

I artikel I och II utforskade vi tillämpningen av ett ramverk för Big Data Analytics som vanligtvis används i molnmiljöer, för två typfall inom läkemedelsutveckling. I båda fallen visar vi goda resultat vad gäller tid för genomförande. Resultaten i artikel I och II gav en god grund för forskningen i de andra artiklarna, men metoderna som utvecklades saknade den allmängiltighet vi strävar efter. I artikel III presenterar vi en generell metod för att utföra bioinformatiska analyser i molnet. Detta resultat möjliggjordes av tillämpningen av en mikrotjänst-arkitektur som gör det möjligt att kapsla in och använda vilken programvara som helst. I artikel IV introducerar vi en metod där mikrotjänster och ramverk för Big Data Analytics kombineras med målet att tillhandahålla det bästa av de båda teknologierna. Slutligen, i artikel V diskuterar vi också hur man kan sänka inträdessteget för att initiera molninfrastruktur på begäran som metoden i artikel III kräver.

Abbreviations

CI/CD	Continous Integration/Continous Delivery
CLI	Command-Line Interface
CNI	Container Network Interface
CP	Conformal Predictor
DAG	Directed Acyclic Graph
DNS	Domain Name Server
EBI	European Bioinformatics Institute
FaaS	Function as a Service
HCL	Hashicorp Configuration Language
HDFS	Hadoop Distributed File System
HGP	Human Genome Project
HPC	High-Performance Computing
ICP	Inductive Conformal Predictor
IP	Internet Protocol
IT	Information Technology
IaC	Infrastructure as code
IaaS	Infrastructure as a Service
LXC	LinuX Containers
ML	Machine Learning
MPI	Message Passing Interface
MS	Mass Spectrometry
NCM	Non-Conformity Measure
OS	Operating System
PaaS	Platform as a Service
RBAC	Role-Based Access Control
RDD	Resilient Distributed Dataset
REST	REpresentional State Transfer
SBVS	Structure-Based Virtual Screening
SKA	Square Kilometer Array
SNP	Single Nucleotide Polymorphism
SOA	Service-Oriented Architecture
SaaS	Software as a Service
VM	Virtual Machines
VRE	Virtual Research Environment
WS	Workflow System
WSE	Weak Scaling Efficiency
vCPU	virtual Central Processing Unit

Contents

List of additional papers v

Sammanfattning på Svenska vi

Abbreviations viii

1 Introduction 10
- 1.1 Big Data in life science 10
- 1.2 Data-processing models 12
- 1.3 Architectural models for e-infrastructure 15
- 1.4 The case for cloud computing in life science 17

2 Aims 23

3 Methods 24
- 3.1 Microservices-oriented architecture 24
- 3.2 Application containers 25
- 3.3 Container orchestration 27
- 3.4 Infrastructure as Code 31
- 3.5 MapReduce 32
- 3.6 Apache Spark 33
- 3.7 Conformal Prediction 36

4 Results and discussion 39
- 4.1 Scalable cheminformatics analyses on cloud resources with Apache Spark (Papers I and II) 39
- 4.2 Scalable pipelines on cloud resources with microservices (Paper III) 43
- 4.3 MapReduce-oriented processing with application containers (Paper IV) 46
- 4.4 A blueprint for on-demand virtual research environments (Paper V) 49
- 4.5 Lessons learned 52
- 4.6 Future outlook 55

5 Conclusions 57

6 Acknowledgements 58

References 61

1. Introduction

1.1 Big Data in life science

Scientific and industrial innovation is increasingly driven by data-intensive applications [1]. From climate science and theoretical physics to stock trading and self-driving cars, Big Data has become fundamental in many aspects of today's society [2, 3, 4, 5, 6].

Life science has entered the Big Data era with the advent of high-throughput genomics. In the early 90s, the Human Genome Project (HGP) set to determine the sequence of nucleotides that constitute a human DNA [7]. This would later have important implications in many knowledge domains, including medical science, pharmacology, evolutionary biology and anthropology – not to mention the development of a multi-billion dollar industry [8, 9, 10, 11, 12]. The HGP achieved its goal in 2003, with an estimated total cost of $2.7 billion. In 2017, the cost of sequencing a human genome was about $1K [13]. This means that accessing a good number of DNA sequences hase become relatively inexpensive, opening up incredible opportunities for life and health science – including understanding genotype-phenotype relationships and enabling personalized medicine [14]. Furthermore, as high-throughput sequencing instruments dropped in price, more and more organizations became able to afford the technology. To get an idea of the amount of data that such instrumentation is capable of producing, let us first consider the size of an individual's genome. A single human DNA is composed by approximately 3B nucleotides, each of which can only be either adenine, guanine, cytosine or thymine. By coding nucleotides in a two-bit alphabet, a human-sized genome can therefore be represented in roughly 750 megabytes. In addition, 30-fold more data needs to be collected to obtain adequate accuracy [15]. For reproducibility purposes this data is usually stored and made available in public repositories – potentially making organizations that do not own high-throughput instruments Big Data consumers. Since multiple research projects may sequence a few thousand human-sized genomes over a relatively short time span [16], it is easy to envision how this field of study leads to a data explosion at the organization level.

Let us take as an example one of the world-leading organizations in life science. The European Bioinformatics Institute (EBI), part of the Eurpean Molecular Biology Laboratory, in Hinxton, United Kingdom, manages one of the globally largest biological data repositories [17]. In the early 2000s, the EBI repository contained only a few terabytes of data. The release of

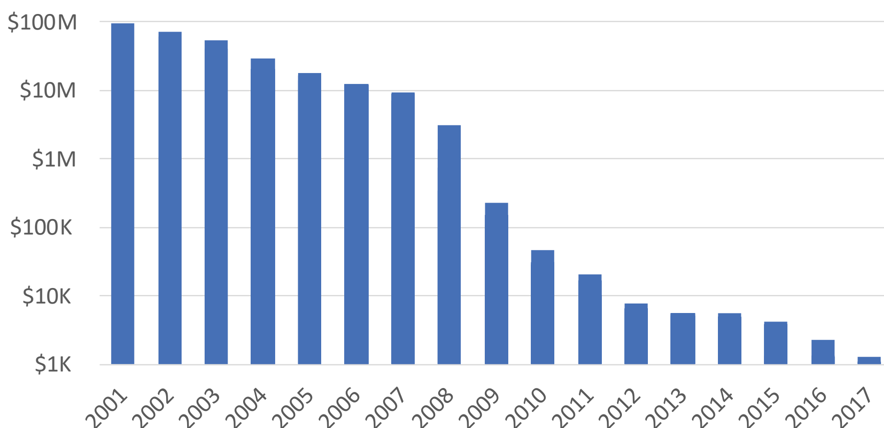


Figure 1.1. **Production-oriented costs for human-sized genome sequencing by year.** Each bar quantifies the production-oriented cost for sequencing a human-sized genome in the relative year, as reported by the National Human Genome Research Insitute, in Maryland, USA [13]. The Y axis is on logarithmic scale, and the currency is in US dollar. Starting from 2008, the drop in sequencing costs proceeded at a considerably higher rate. The transition to high-throughput instruments accounts for this remarkable result.

the first high-throughput sequencing platform, in the mid 2000s, allowed for a 50K-fold drop in DNA determination costs [18]. After the first period of skepticism [14], the adoption of this new technology led to a considerable reduction in sequencing expenses in the following years (see Figure 1.1). At the same time, the EBI's data repository started to grow dramatically in size, more than doubling every year until 2012 and exceeding 100 petabytes in 2018 [17].

Genomics data is not the only kind of information that life scientists have to do contend with. In fact, biological data is remarkably heterogeneous and encompasses information from a wide range of experiments. While research projects usually focus on distinct domains, organizations need to provide scientists with a platform to handle a wide variety of data types. To mention just a few, these may include: protein representations, interactions and identifiers, molecular representations and properties, metabolite structures and spectra, genome-to-phenome relationships and microarray data for gene expression [19, 20, 21, 22, 23, 24]. Similarly to the case mentioned above for genomics, the recent advancements in high-throughput methods have accounted for the production of massive datasets in each of these information domains [17]. To get an impression of the impact that these new techniques have had at the organization level, let us again take EBI as an example. Mass Spectrometry (MS) and microarray technologies are two broadly-adopted high-throughput methodologies, across different life science domains [25, 26]. As of 2018,

the MS and microarray data volumes stored by EBI are rapidly approaching 10 petabytes and 1 petabyte respectively [17]. Between 2007 and 2018 both these volumes grew at increasing rate together with the EBI's storage capacity, which reached an astounding 160 petabytes in 2018.

By now the reader is hopefully convinced that life science is rapidly turning into a data-intensive field. Indeed, while in the mid 1960s biological datasets could be published in print [27], the amount of data that the modern life scientist has to deal with is often out of the storage capabilities of a single workstation – not to mention the computing power that is required to analyze such information. In fact, biologists do not collect data to merely store it in large-scale data centers, but rather they process it to derive knowledge and novel results. The next section introduces this topic.

1.2 Data-processing models

I will first introduce the topic of data processing in the life sciences by showing a real-world analysis in the field of metabolomics (see Figure 1.2). This branch of life science aims at studying biological interactions of metabolites: the small molecules that are produced by chemical processes inside a living organism. Similarly to many other cutting-edge studies in the field, the presented analysis leverages high-throughput MS data to identify metabolites in biological samples. The original study [28], which we reproduced in Paper III, makes use of several experiments that were performed over a relatively large amount of samples, producing data volumes comparable to those generated in genomics studies. However, Big Data constitutes only one side of the challenge here. The other side is the remarkably heterogeneous landscape of MS data-processing software [29]. Indeed, MS analytics platforms, such as OpenMS [30], encompass hundreds of tools which, according to the use case at hand, need to be carefully selected and employed in a data pipeline. The reason for the introduction of such complexity resides in the notably fast moving pace of this field – not to mention the ubiquity of MS across life science disciplines [25]. Consequently, MS analytics have traditionally been tailored to specific sampling strategies and experimental environments, requiring substantially different data-processing techniques – to say nothing of the fact that software quality can vary considerably across different projects. For this reason it is important to construct MS analyses composed of minimal, complete and interoperable processing tools. In this way, provided that good data standards are adopted, such processing elements can be selected and reused in several pipelines, according to sampling techniques, experimental settings and goals of the current study.

In figure 1.2 a Directed Acyclic Graph (DAG) is used to give a high-level definition of the data pipeline. Formally, this data structure consists of a finite set of vertices and ordered vertices pairs, representing connections (or edges),

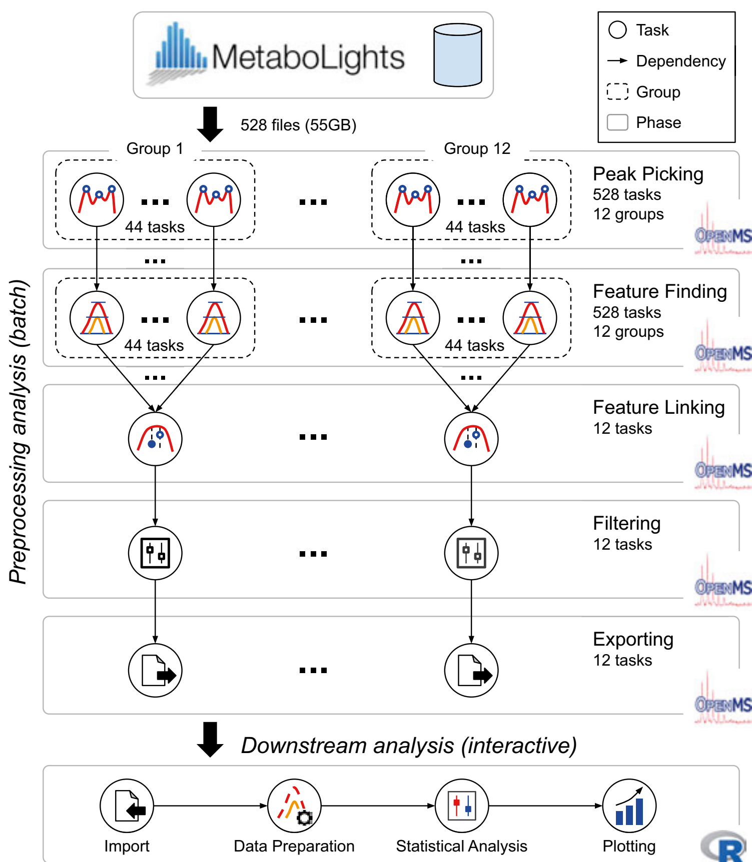


Figure 1.2. **Real-world large-scale metabolomics pipeline.** The diagram gives a high-level overview of a metabolomics data pipeline that we ran in Paper III. The pipeline is represented by a Directed Acyclic Graph (DAG), with vertices and edges constituting tasks and dependencies respectively. The first task consists of ingesting the raw MS data from the MetaboLights repository [22]. This dataset is constituted by 528 machine-readable files, organized into 12 groups of 44 files, and is 55GB in size. The rest of the pipeline is divided in preprocessing and downstream analysis, where the former is run in batch mode and the latter is run interactively. The prepossessing analysis, which is implemented using command-line tools from the OpenMS platform [30], can be divided into five phases: (1) peak picking, (2) feature finding, (3) feature linking, (4) filtering and (5) exporting. Phase (1) is composed by 528 tasks (one for each input file) and reduces the data size by filtering out peak intensities in the input spectrum. Phase (2) filters mass traces that are reconcilable with metabolites features. This phase also includes 528 tasks. Phase (3) links the selected features across different files in the same group, producing a consensus map. Since the input files are organized in 12 groups, phase (3) is comprised of 12 tasks. Phase (4) and (5) filter and export interesting data for the downstream analysis, and also include 12 tasks. Finally, the downstream analysis, which is implemented in R [31], imports the data from the last prepossessing phase, prepares the data appropriately, performs a statistical analysis and plots the results.

where it is impossible to start from a certain vertex and eventually loop back to the starting point, by following the connections. DAGs have at least two interesting properties for our use case. First, by representing data-processing software instances, or tasks, as edges and task dependencies as vertices, one can effectively use DAGs to define sophisticated data pipelines. The resulting programming paradigm is remarkably easy to understand and work with – one can simply think about data flowing through the processing elements following the graph edges. In addition, as vertices are in principle isolated and independent, it is possible to reuse any number of processing elements in multiple pipelines with various dependency relationships. Second, DAG-represented pipelines intrinsically define scheduling and parallelism of an analysis. In fact, those tasks that are represented by independent edges can be executed in parallel, and the completion of an upstream task implicitly triggers the scheduling of the next one. This is of great importance, as the datasets that we are considering here are impractical to handle on a single workstation, hence making scheduling and parallelization over distributed systems a necessity.

We have already highlighted the increasing size of datasets across life science disciplines, and we can observe a similar trend when it comes to the proliferation of bioinformatics data-processing tools [32, 33]. Indeed, the remarks that we have made, regarding fast moving high-throughput technology and experimental settings tailoring of MS analyses, can be extended to bioinformatics pipelines in general. To this extent, in order to get an idea of the heterogeneity in bionformatics software, you may consider that a single genomics software package such as GATK comprises more than 300 processing tools [34]. Therefore, easy pipeline composability and intrinsic parallelism may be the reason why bioinformaticians have traditionally found it practical to define data pipelines as DAGs [35]. Furthermore, it is common in the field to aid the definition of DAGs by using Workflow Systems (WSs). These software platforms allow the user to define data pipelines through programming languages, domain-specific languages or even graphical user interfaces [35, 36, 37, 38]. The user-defined DAGs are then seamlessly translated to a concurrent scheduling plan, which is automatically materialized over a distributed platform. This way of working has become so popular that there now exist communities of practice that actively collaborate in developing and maintaining reusable bioinformatics workflows [39, 40, 41, 42].

One drawback of using WSs is that, while scheduling and parallelization are intrinsically defined by the user, there is no implicit data-locality information in DAG-defined pipelines. As a result, WSs usually end up scheduling a large number of tasks that access data remotely. While some argue that this is not a problem from a performance perspective [43], careless transfer of massive amounts of information inevitably results in poor power efficiency [44]. Bioinformaticians therefore often adopt Google's MapReduce [45] to solve this issue [46]. This programming model and parallel implementation (described in Section 3.5) was originally developed to enable the efficient processing of Big

Data with commodity hardware. To this extent, one of the enabling characteristics is the ability of scheduling tasks close to the data, thus allowing near-zero remote access. Nevertheless, MapReduce and its modern open-source implementations [47, 48] only offer limited support for reusing existing software tools. For this reason, some large research initiatives, such as the community behind the ADAM platform [49], have completely reprogrammed their pipelines around this model. However, as the reader may be aware, communities of practice can seldom sustain this effort, and this is most likely the reason why bioinformaticians commonly run workflow-oriented pipelines instead.

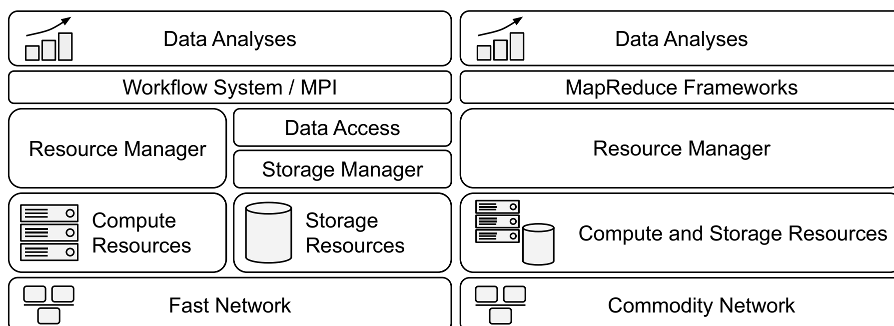
Both WSs and MapReduce frameworks typically run analyses in batch mode. This means that pipelines are defined at once and then submitted to a distributed platform, which comes back with the results asynchronously. While this is convenient for well-defined, production-oriented workflows, life science has an exploratory nature and therefore alternative models are emerging. In particular, there is growing interest in notebook environments, such as Jupyter [50] and Apache Zeppelin [51]. These platforms come as web applications, where the user can write code in a notebook-like interface for the purposes of quickly and interactively exploring the data. Such interactive notebooks can then be shared and collaboratively edited with other researchers. Returning to Figure 1.2, the downstream analysis is composed of a few consecutive tasks, which utilize relatively small datasets output from the preprocessing phase. In this final stage of the workflow some statistical analysis and plotting is performed. Coding a statistical analysis for a reduced dataset is relatively straightforward when using the R programming language [31], and plotting often only requires trying a few different visualizaion techniques to find the most suitable one. Hence this part of the analysis was performed interactively using Jupyter.

The choice between WSs, MapReduce or notebook environments largely depends on the problem at hand, and often also on personal preference. However, no matter which processing model is selected, the access to adequate e-infrastructure is a necessity when dealing with large datasets. This topic is discussed in the next section.

1.3 Architectural models for e-infrastructure

Life science is quickly becoming a data-intensive discipline, requiring organizations to provide adequate e-infrastructure for information processing and storage. High-Performance Computing (HPC) systems have traditionally provided the means to achieve such a goal [52].

Figure 1.3a shows a typical layered architecture for a HPC system. This architectural model decouples compute and storage, which has two major implications. First, there is a need for an underlying network layer that provides low latency between compute and storage resources, and second developers



(a) HPC. *(b)* Big Data.

Figure 1.3. **HPC and Big Data architectures.** The figures summarize the two models in a layered architecture. HPC systems are characterized by fast interconnection, which allows for decoupling compute and storage resources. Consequently, compute and storage management are also decoupled, with data access provided as a layer on top of the storage manager. In this model, data analyses can be implemented using WSs and MPI libraries which operate horizontally over the decoupled resources. In contrast, Big Data infrastructure builds on top of the commodity network. Thus, to avoid communication bottlenecks, compute and storage are coupled on the same hardware infrastructure. Finally, a resource manager operates on top of the coupled infrastructure and data analyses are implemented using MapReduce frameworks, which minimize network access.

typically need to manage data shuffles when implementing their analyses. To this extent, WSs are an increasingly popular choice in life science because of their convenient management of existing processing tools. When using these frameworks, pipeline developers simply include network shuffles between storage and compute resources as part of their DAGs. To complement this model, Message Passing Interface (MPI) [53] libraries are also often used on HPC systems, as they provide developers with a higher degree of freedom when implementing analyses. In fact, as the name suggests, MPI primarily aims at facilitating communication between disperse processes in a computing cluster, allowing the tuning of low-level implementation details with the aim of achieving better performance. Nevertheless, it is important to point out that bioinformatics analyses usually comprise a multitude of loosely-coupled processing units, thus normally not needing sophisticated synchronization mechanisms over a large number of machines. For this reason using WSs or even simple scripts, rather then MPI, are what most bioinformaticians strive for.

HPC architectures are often associated with cutting-edge, and expensive, hardware resources which ensure fast data access along with a low failure rate. In the late 2000s, together with the introduction of MapReduce, Google laid the foundations for an architectural design that is instead tailored to commodity hardware infrastructures [45]. This alternative cluster-computing model, often referred to as Big Data architecture, co-locates compute and storage

resources, with the goal of minimizing network access – thus enabling the handling of massive datasets on commodity networks (see Figure 1.3b). Furthermore, in Big Data architectures locality-aware scheduling is seamlessly handled by the MapReduce framework of choice, which can enforce job-scheduling constraints to the underlying resource manager (also taking care of possible hardware failures).

The Big Data infrastructure model, along with MapReduce and its open-source counterparts, has been remarkably successful in enabling large-scale analytics in tech companies such as Google, Facebook and Spotify [45, 54, 55]. One could therefore question why Big Data infrastructures are not so common within scientific organizations. First, and perhaps most importantly, scientific applications have a long tradition in adopting HPC for pushing their boundaries [56]. This implies that important investments were made in the development of such technology and strong HPC expertise was built within scientific organizations. In addition, modern MapReduce-oriented frameworks can run on HPC resources with relatively little effort [57], eliminating the need for investing in new e-infrastructure. Furthermore, regardless of the choice between HPC and Big Data, it is important to point out that research organizations traditionally make consistent upfront investments in purchasing hardware [56] – not to mention the continuous costs to support and maintain such infrastructure. To this extent, cloud computing has the potential to disrupt this investment model, enabling the expansion of the computing and storage capabilities of an organization on demand with a pay-per-use pricing model.

1.4 The case for cloud computing in life science

Figure 1.4 shows the traditional life science data ecosystem. This conceptualization was first introduced by Lincoln D Stein [58], and it is still in part adopted by large research institutions such as EBI (United Kingdom) and the Science for Life Laboratory, in Sweden [17, 52]. In the figure, research organizations, laboratories and public data curators are represented as independent entities, but in reality there can often be intersections between them. High-throughput instruments, operated in research laboratories, generate data which is submitted to public database curators which make them available in data archives. Some example of these are MetaboLights [22] and the European Nucleotide Archive (part of EBI) [59]. Power users can therefore copy and process interesting data in the HPC resources that are provided, on premises, by the belonging research organization. In addition, they may also make enriched data (produced as part of their research) available in value-added archives. Finally, casual users access small portions of the data from the archive of choice, to perform small scale analyses on their workstations.

The introduced traditional approach has two major limitations. Firstly, when entering the Big Data era copying information locally becomes more

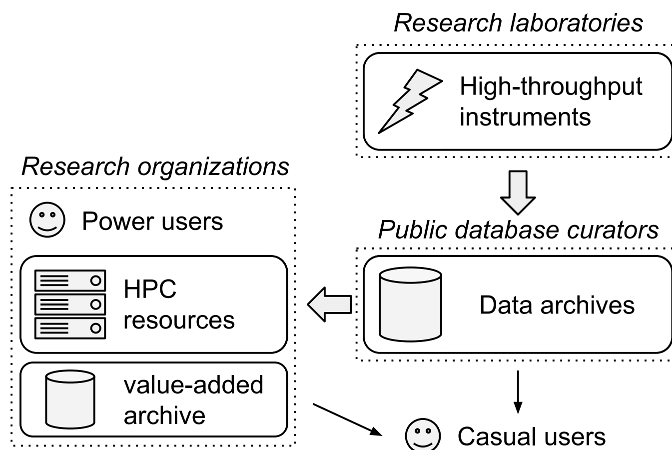


Figure 1.4. **Life science data ecosystem: the traditional approach.** Research laboratories, public database curators and research organizations may act as independent entities, leveraging on physical infrastructure – often geographically dispersed. Research laboratories, owning high-throughput instruments, ideally submit datasets to public database curators which make them publicly available, in their data archives. Power users download copies of interesting data in the HPC resources provided by their research organization, run some analysis and may publish results in value-added archives. Finally, casual users download small portions of the datasets for small-scale analyses on their workstations.

and more inefficient, in term of storage usage, as datasets grow in size. In addition, as data is commonly shuffled over the Internet, this can be a slow process, potentially causing network contention in the institutions. Secondly, research organizations and public database curators need to setup and maintain in-house e-infrastructures for storing and processing the datasets. Also in this case entities may act independently, which may be not only immediately expensive when purchasing the hardware, due to economies of scale, but also expensive in the long term due to infrastructure maintenance costs. In addition, in this investment model, there is a considerable upfront commitment when buying and setting up HPC and storage systems, which may remain unused for long periods of time whilst still requiring maintenance and electricity. After all, power users need considerable HPC resources only when a new important dataset becomes available, or when a scientific meeting of interest is approaching, causing the loads to fluctuate considerably [58]. Research organizations therefore have the dilemma of providing enough compute capacity for peak periods of usage or purchasing smaller systems that will be more likely to be continuously at full capacity. The first option has the disadvantage of wasting resources, while the second option causes major frustration in power users, as they may not be granted enough compute hours in periods of peak usage.

The technology, and business trend, developed under the umbrella of cloud computing [60] opens up interesting alternatives to the traditional life science data ecosystem. This includes a large ecosystem of software products that, instead of being run on premises, are hosted outside the boundaries of the user's institutions – hence offered "as a service". With this new software-providing model comes also an alternative pricing option. Instead of purchasing the product with an upfront payment, the user pays as the business goes, with a pay-per-use billing system.

In the cloud community, it is common to categorize cloud services in three layers. The top layer, called Software as a Service (SaaS), contains the services that end users consume. Making a parallel with Figure 1.4, one can include in this layer a web portal that casual users consume, in order to visualize and downloads small-scale data. The middle layer is called Platform as a Service (PaaS). As the name suggests, it contains the software platforms that supports the applications in the SaaS layer. Also with reference to Figure 1.4 one could include web frameworks such as Django [61] in this layer. From the perspective of a web developer, PaaS becomes a game changer. In fact, "as a service" here means that he or she is no longer required to setup Django, but can instead conveniently consume it as a managed service. This is even more interesting when it comes to our use cases in high-throughput life science. Indeed, we can now consume on-demand HPC or Big Data platforms with no need for continuous in-house management. Finally, on the bottom comes the Infrastructure as a Service (IaaS) layer. The idea behind the services in this layer was originally introduced by Amazon and it disrupted the way many organizations procure their e-infrastructure [60]. Users can simply allocate and release virtualized resources on demand, and be billed only for the allocation period. Furthermore, similarly to the upper layers, the physical resources are located outside the hosting institutions and fully managed by the IaaS providers.

By leveraging cloud computing, it is possible to conveniently reorganize the life science data ecosystem as shown in Figure 1.5. Research institutions, such as EBI, are currently moving towards this model to better cope with the increasing amount of information in the field [62]. In this new paradigm, a cloud provider purchases a large server farm, receiving a reduced price due to economies of scale. The cloud provider offers resources to public database curators and research organizations on demand, and bills them according to usage periods. Under these settings, the data that is sent from research laboratories to database curators is stored on PaaS archives by the cloud provider. Power users that need to analyze such data can now flexibly allocate PaaS HPC or Big Data resources on the same cloud, and may store results in value-added archives. Finally, from the perspective of the casual users not much changes, they can still access small amount of data from web interfaces.

The cloud-oriented way of organizing data in life science has some interesting implications. Firstly, data can be conveniently located in a geographically-

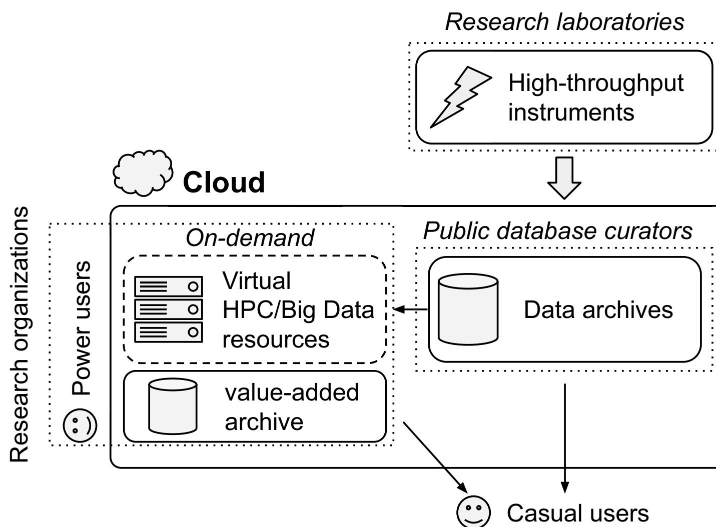


Figure 1.5. **Life science data ecosystem: the cloud-based approach.** Research laboratories, public database curators and research organizations may still act as independent entities, but now leverage cloud-hosted virtual infrastructure – possibly located in the same data center. Research laboratories ideally submit datasets to public database curators, which persist and make them available on cloud storage, as data archives. Power users independently allocate HPC or Big Data resources on demand, close to the data that needs to be processed, and may keep results persistent on cloud storage in the form of public value-added archives. Casual users still download small portions of the datasets for small-scale analyses on their workstations.

dispersed manner, inside cloud regions that are close to research laboratories – as opposite to many research labs that submit to a database curator which relies on a centralized infrastructure. While data transfers from labs to cloud providers still represent a bottleneck here, it is at least mitigated by the proximity to the server farm, and power users can cleverly avoid further data shuffles. In fact, in this cloud-oriented paradigm, HPC and Big Data resources can be now allocated straight from where the data is located, along with the storage needed for value-added archives.

The second implication is a shift in the investment strategy. Again, in this model research organizations and public database curators no longer need to pay upfront for acquiring hardware. This implies that data archives can now conveniently scale up as more data becomes available, while research organizations are no longer subject to the dilemma of providing enough capacity for peak usage periods, or maximizing their usage – power users can simply allocate and release infrastructure as needed. To this extent, it would nevertheless be naive to jump to the conclusion that acquiring computing power this way is less expensive. In fact, cloud infrastructure is generally more expensive than

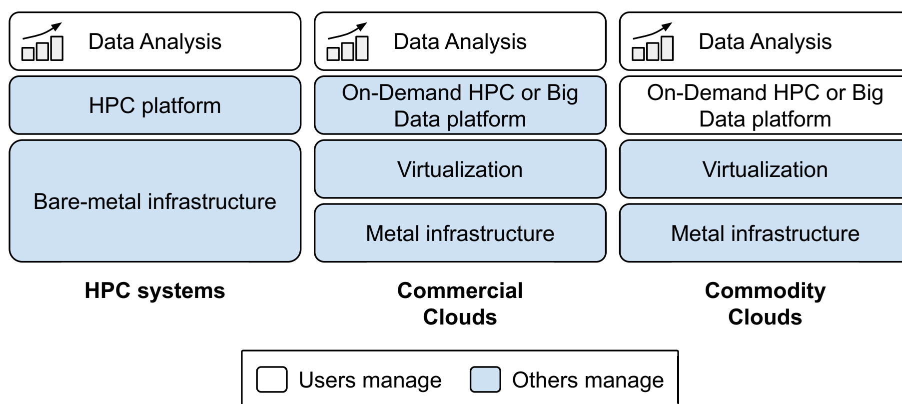


Figure 1.6. **User management in HPC systems, commercial clouds and commodity clouds.** Traditional HPC systems are build on bare-metal infrastructure that is maintained in house. In this model, system administrators set up and maintain the bare-metal infrastructure as well as the HPC software platform, so that users can focus on their analyses. Commercial cloud providers manage the bare-metal infrastruture, as well as the virtualization layer that enables on-demand allocation and resource sharing. It is also common in this model to provide managed HPC and Big Data software platforms on demand, so that users can still focus on their analyses. Finally, in commodity clouds, the bare-metal and virtualization layers are managed and provided as a service, while users need to manage both the on-demand HPC, Big Data platforms and their analyses.

its bare-metal counterpart, and the economy starts to look good only when users diligently release unused resources [60].

In my opinion the shift in investment strategy, as well as instructing users to carefully release unused infrastructure, do not represent major issues when moving life science to the clouds. However, I can identify at least two relatively important obstacles. First, life scientists often need to deal with sensitive human data. While commercial cloud providers guarantee rigorous security standards, there are ethical concerns, as well as law restrictions, that simply makes it unacceptable for such data to leave the boundaries of an institution [63]. The cloud community has coped with this and other similar limitations, by introducing the hybrid cloud. In this paradigm, institution can run private clouds on premises and run analyses in house when needed. As the in-house system is based on the same technology as the outsourced cloud, switching between the two should be seamless – at least in theory. In fact, while large industries can afford Information Technology (IT) consultants, and pricey middleware software, to smoothly move between private and public infrastructure, research organizations seldom purchase such services – causing major frustrations for their affiliates.

The second obstacle in moving life science to cloud resides in the way scientists are currently familiar with managing e-infrastructures (see Figure 1.6).

As I have already mentioned in section 1.3, HPC systems are the most popular computing resources in research organizations. Scientists are used to consuming HPC platforms as a service, by simply submitting jobs to resource managers (see Figure 1.3a). When moving to the public cloud, the on-demand computing platform is still managed, but researchers now need to explicitly create the on-demand cluster before submitting their jobs. Even if commercial cloud providers have convenient APIs to make this seamless, a certain level of IT competence is still needed in terms of configuration and, since we want researchers to focus on science, this becomes impractical. This problem is amplified when researches need to leverage commodity clouds, which seldom provide HPC and Big Data PaaS, requiring scientists to deal with the IaaS layer before a job can be submitted. In addition, the landscape of public clouds is remarkably heterogeneous, with many companies pushing their own substantially different PaaS. Hence, by tailoring their analyses to a specific provider, there is a serious risk for researchers to incur vendor lock-in.

By now, the reader can hopefully foresee remarkable opportunities and interesting challenges in shifting life science to the clouds – which brings us to the aims of this thesis.

2. Aims

The overarching aim of the methods introduced in this thesis is to enable scalable life science analyses on cloud infrastructures. With this goal in mind, the presented research was built around the following specific aims:

1. Improving the accessibility of cloud-agnostic e-infrastructure for life scientists (Paper II and V).
2. Enabling scalable data pipelines of existing software tools (Papers I, II, III, IV and V).
3. Improving the accessibility of scalable data pipelines for life scientists (Papers I, II, III, IV and V).

Even though life science applications are used as major demonstrators, considerable efforts were put into making the methods generally applicable across multiple scientific domains. We therefore envision future applications of the developed methods in other fields.

3. Methods

3.1 Microservices-oriented architecture

The microservice-oriented architecture is a software engineering design pattern where complex service-oriented applications are composed of a set of interchangeable, minimal and complete services – referred to as microservices [64].

Microservices can be deployed independently and are compatible with one another through language-agnostic APIs. For this reason, in communities of practice an analogy between microservices and building blocks has often circulated (see Figure 3.1). Following this analogy, it is easy to see how this pattern promotes scalability, nicely coupling with the elasticity of cloud resources. Indeed, microservices are today consider the gold standard of cloud-native applications. However, the marriage between microservices and cloud is not merely a matter of improved scalability. As previously discussed, the "as a service" concept is central in cloud systems. This means that in the layered SaaS, PaaS, IaaS architecture, cloud services can be also consumed programmatically via language-agnostic APIs – such as those enabled by REpresentational State Transfer (REST) [65]. Therefore, writing software to automate the operations of infrastructure and services becomes an appealing possibility. To

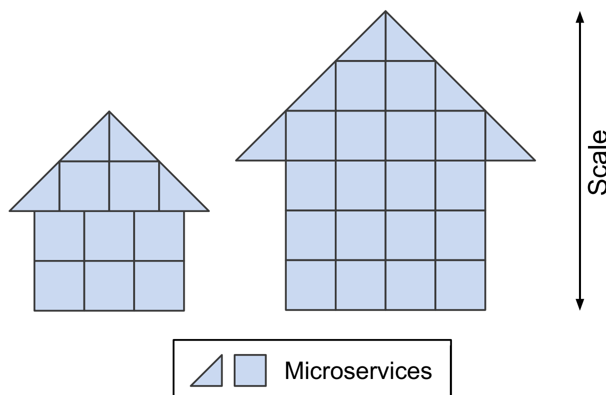


Figure 3.1. **The analogy between microservices and building blocks.** In this figure we make an analogy between microservices and building blocks. Like building blocks, microservices are compatible one another and interchangeable. Hence, by varying their replication factor one can build models at any scale.

this extent, software engineers adopt a set of practices, often referred to as DevOps [64], with the aim of minimizing the time of delivering system changes to production environments while maintaining good quality standards. Further, such practices encompass continuous monitoring, performance feedback and anomaly detection, possibly enabling autonomous changes to a system with the goal of maximizing its resilience.

Microservices dramatically improve the agility of DevOps practices. Indeed, following the monolithic approach even the smallest change in the codebase often necessitates the redeployment of the whole system. Thus, even if cloud services enable automated operations, applying small and continuous changes in production system would be unreasonable. In contrast, following the microservices design pattern, applications are composed of light-weight and independent components. This means that small changes in the codebase now only require redeployment of the relative microservices, considerably improving the agility of the system. In addition, as microservices are exchangeable with one another, when one of the building blocks breaks down it can be automatically replaced by a new one.

In this section, I discussed microservices and their convenience in cloud-oriented environments. For simplicity, I leave out possible implementations for these interchangeable, minimal and complete components. While one can think of many possible solutions to achieve this goal, in the past few years application containers have become the de-facto standard enabling technology.

3.2 Application containers

Application containers, also referred to as software containers or simply containers, represent the cutting-edge enabling technology for microservices. Formally, the Open Container Initiative (OCI) defines a standard container as a unit that encapsulates a software component with all its dependencies in a self-describing and portable format, so that any compliant system can run it with no need for additional dependencies, regardless of the container content [66]. Standard containers have two main advantages when used to implement microservices. Firstly, an improved agile process in which developers have the freedom of choosing their preferred programming environment becomes possible – assuming that microservices can communicate via language-agnostic APIs. Secondly, container-based microservices are by definition portable across multiple vendors, provided that the enabling runtime can operate on a wide variety of systems.

While Virtual Machines (VMs) images could in principle be used to implement standard containers, the microservices community usually adopts a runtime implementation that differs substantially from hypervisors (see Figure 3.2). In fact, when running a VM, hypervisors hold both a copy of a guest Operating System (OS) and a copy of the required virtualized hardware [67].

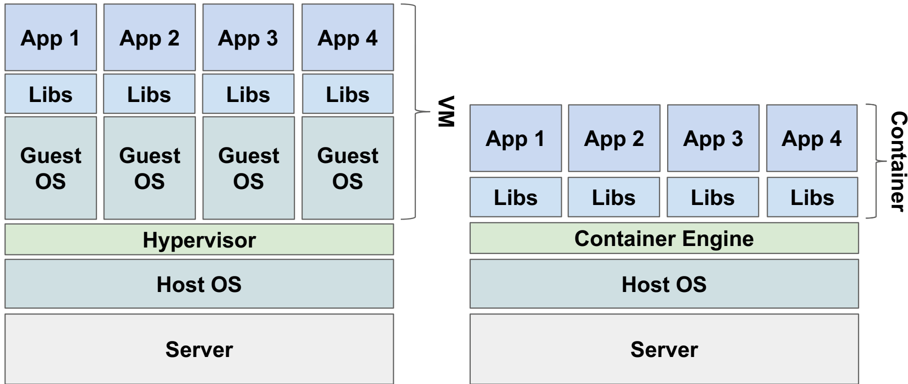


Figure 3.2. **Virtual machines and containers.** Virtual Machines (VMs) could in principle be used at runtime to implement an OCI standard container, thus encapsulating an application along with its libraries. However, the microservices community usually adopts an implementation that enables a lighter encapsulation mechanism. In contrast to hypervisors, container engines do not hold a full copy of a guest Operating System (OS), and required virtual hardware, but instead leverage on host OS kernel namespaces to provide isolation.

Likewise monolithic Service-Oriented Architecture (SOA) represents an obstacle to DevOps agility, if container were implemented directly on top of traditional hypervisors performance issues would hinder effective microservice operations. Indeed, VMs usually require minutes to initialize the host OS, while in DevOps practices we would ideally like to instantiate a microservice in a fraction of a second.

Container engines represent the cutting-edge enabling technology for microservices. Their main advantage is that they do not rely on hardware virtualization, but instead dock the containters straight to the host OS. Consequently, isolation is provided by leveraging on top of kernel namespaces. This implementation makes containers considerably lighter and faster to instantiate than VMs, but also more tightly coupled to the underlying OS. To this extent, if a container gets compromised an attacker could attain complete access to the host system. For this reason, a combination of both VMs and containers is what most organizations require. Simply, a VM-based infrastructure can be allocated at first and then containers can be flexibly scheduled on top. Further, to provide good agility, new VMs can be proactively instantiated so that when a service needs to be scaled up a new container can be quickly spawned.

Starting from the mid 2010s, containerization considerably increased in popularity and consequently many organizations are currently shifting their infrastructure in this direction. The enabling technology was however available at least two decades earlier. The *jail* command, which builds on top of UNIX's *chroot*, was first introduced by Bill Cheswick in the 1990s [68]. Like modern container engines, *jail* enables the creation of process sandboxes

```
1  FROM python
2  COPY ./my_script.py /my_script.py
3  CMD ["python", "/my_script.py"]
```

Figure 3.3. **Dockerfile example.** This Dockerfile builds a container image that packages a Python [71] script. On line 1, the `FROM` directive instructs the Docker builder to start from a previously-packaged image, which contains all of the necessary Python dependencies. On line 2, the `COPY` directive copies the the Python script from the developer's workstation to the container image. Finally, on line 3, the `CMD` directive specifies the command to run when starting the container on the user's machine.

isolating entities such as filesystems, users and networks. Later, in the mid 2000s, Google announced *cgroups* [69]: a kernel feature that introduced the possibility to isolate and limit the resource usage of a process. This feature was merged into Linux and eventually lead to the development of the LinuX Containers (LXC) project [70]. LXC, first released in 2008, complies with the OCI standard container definition. One can then arguably ask why containers were not popularized for almost another decade. The main reason is that at the time LXC, and other similar projects, did not include any feature to easily package and deliver containers – thus making it hard to adopt DevOps practices.

Application containers really started to take off in 2013 with the first open-source release of Docker [72]. The project introduced a software ecosystem to ease the packaging and distribution of containers – and ultimately of self-contained software stacks. These include a convenient machine-readable language for the definition of container images, public and private image registries, a RESTful API and a Command-Line Interface (CLI). Leveraging on this software ecosystem developers can conveniently define container images in a Dockerfile (see Figure 3.3) and make their builds available in public or private registries, such that communities of practice can seamlessly run them on any Docker-compliant infrastructure.

To conclude this section, I would like to point out that Docker, Inc. formerly known as dotCloud, Inc., started as a small startup only a few years ago and it is today capable of raising hundreds of millions of dollars in ventures [73]. This perhaps highlights the importance of democratizing software.

3.3 Container orchestration

Application containers provide the means for implementing isolated, portable and self-contained microservices. Hundreds of these encapsulated software components are often employed when composing a microservices-oriented application. Such an ensemble of containerized services needs to be scal-

able, fault tolerant, highly available and often geographically dispersed for the application to succeed. This is where container orchestration platforms are important, as they provide the means to achieve such a goals.

There are many orchestration platforms available in the open-source landscape. These include:

- Kubernetes, introduced by Google [74].
- Mesosphere, implemented on top of Apache Mesos [75].
- Docker Swarm, introduced by Docker, Inc [76].
- Nomad, introduced by Hashicorp [77].

Among these, Kubernetes is the platform that has collected the largest open-source community, becoming the industry standard for cloud-native applications [78]. The project has been so successful that even competitors such as Mesosphere, Swarm and Nomad nowadays include Kubernetes compatibility to ensure interoperability within the microservices software landscape.

The key features of a container orchestration platform, as described by Asif Khan (chief engineer at Amazon Web Services) [79], can be grouped in five main categories: (1) cluster state management, (2) Container Network Interface (CNI) and service discovery, (3) security assurance, (4) Continuous Integration and Continuous Delivery (CI/CD) and (5) monitoring. The rest of this section describes these key feature groups.

Cluster state management

In the previous section, I have mentioned that for security reasons application containers should be instantiated on top of a cluster of VMs. Managing the state of this cluster in a reliable and fault tolerant manner is one of the main features that an orchestration platform should provide. This comprises a series of mechanisms to allocate resources and to schedule containers across the available VMs, as well as reactive policies to adapt container placements according to certain events (such as VMs becoming unavailable or cluster scaling). Furthermore, cluster state information should be preserved in a reliable, highly-available storage system and single points of failure should be eliminated. To this extent, the orchestration platform should exhibit a graceful degradation of services when a failure occurs, and system administrators should be promptly notified.

CNI and service discovery

When running top of a host machine, container engines such as Docker create a default network for the running containers. In this way, containers can communicate within the host via Internet Protocol (IP). However, in our scenario, containers are dispersed across a cluster of VMs. This is why orchestration platforms should normally provide a CNI service that makes containers reach-

able across multiple hosts. There are multiple CNI implementations available in the open-source landscape, which follow two main approaches. Layer 2 (L2) CNIs [80] implement a cluster-wide logical network, which is encapsulated over the existing network layer. In contrast, Layer 3 (L3) CNIs operate straight on the existing network layer using IP forwarding mechanisms to translate between host and container addresses, as well as a central storage to exchange routing information across the cluster. While L2 CNIs provide better isolation, L3 CNIs are more scalable and easier to debug.

In both CNI implementations, each container (or group of containers) is addressable over IP. However, as events such as cluster scaling or failures could influence container placements over the cluster, container addresses are dynamic by definition. This is why orchestration platforms should also provide a service discovery mechanism that allows applications to discover the address of a certain service. Service discovery mechanisms can also be categorized into two main classes: client-based and server-based. When adopting client-based service discovery, the application is responsible for querying a key-value storage that resolves a service name to an IP address. Intuitively, this mechanism can be implemented using a traditional Domain Name Server (DNS). In contrast, by adopting server-based service discovery, clients route their requests to a load balancer, which then directs them to the desired container (according to the routes defined in a central key-value storage). This mechanism is also relatively easy to implement with a reverse proxy. In conclusion, service discovery can also be used to balance the load between container replicates. Simply, multiple container addresses can be mapped to the same service name and requests can be evenly distributed.

Security assurance

Similarly to other software solutions, microserivices-oriented application are potentially prone to malicious attacks. It is therefore important for orchestration platforms to provide a set of functionalities that allows the enforcement of security standards. While giving a comprehensive description of security assurance in container-based environment is out of the scope of this thesis, it is useful to discuss a few important features that orchestration platforms should provide in this context.

Container images represent the source from which containers are instatiated. It is therefore important to trust their hosting registry. To this extent, orchestration platforms should normally enforce running containers to have originated from images that are signed by a trusted registry. In addition, there should be a community effort in setting up a periodic validation and fixing process, to ensure that images in trusted registries are not prone to code vulnerabilities.

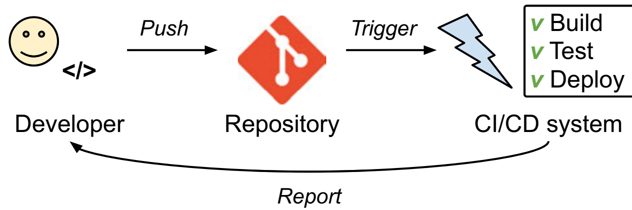


Figure 3.4. **CI/CD cycle.** The diagram shows a typical CI/CD cycle. The developer pushes a commit to a repository, which triggers the CI/CD systems. The relative service is then built, tested and deployed automatically. Finally, a results from the CI/CD process are reported to the developer.

Role-Based Access Control (RBAC) [81] should also normally be enforced by orchestration platforms. RBAC can restrict users or containers from consuming certain services or accessing sensitive information. To this extent, secret management should also represent a core security-related feature in orchestration platforms.

Finally, there should be a mechanism for monitoring runtime anomalies in the underlying VMs. In fact, the shared kernel model that application containers introduce make the system more prone to the manipulation of the hosting OS. To this extent, the orchestration platform should also enable smooth and periodic OS updates in the underlying VMs, where containers can be detached and reattached as needed.

Continuous Integration and Continuous Delivery

The introduction of Docker considerably improved the agility in delivering microservices (and self-contained software stacks in general). CI/CD is a DevOps practice that employs an automated pipeline, allowing for continuous integration and delivery (or deployment) of microservices (see Figure 3.4). In more detail, CI consists of continuously integrating software changes in the codebase, by running automated unit and integration tests with the aim of detecting issues as fast as possible. If the tests are passed, a subsequent CD phase is triggered and the newly-built microservice is released and deployed to the orchestration platform. The advantage of CI/CD, in the context of container-based microservices, is that allows developers to become central players through the whole software life-cycle. Indeed, by using containerization and language-agnostic APIs they now have great freedom in choosing their preferred software stacks, while ensuring interoperability with the rest of the application ecosystem. This characteristic of container-based software couples nicely with CI/CD, which further empowers developers with the ability to easily managing testing and operations of their software releases.

CI/CD can be outsourced from managed services such as GitLab [82], hosted in-house as a decoupled service via open-source platforms such as

Jenkins [83] or be an integral part of the orchestration platform – as is the case for RedHat OpenShift [84] (a fork of Kubernetes).

Monitoring

Common monitoring techniques like tracerouting, network usage and logging should normally be provided by orchestration platforms [79]. These should be applied both at the VM infratructure level and at the container level. At the VM infrastructure level it is important to monitor resource usage to guarantee adequate quality of service, as well as security-related metrics. Logging is important at the container level, and a mechanism for easily retrieving logs from dispersed microservices should be provided. Also, at the container level, a mechanism for monitoring how much of the host resources a container is consuming should be enabled. Finally, linking back to security assurance, a mechanism for scanning container images with the aim of detecting possible vulnerabilities should ideally be an integral part of the container monitoring.

3.4 Infrastructure as Code

The "as a service" nature of the cloud makes its operations easily manageable via language-agnostic APIs. Indeed, cloud APIs allow to programmatically script operations with the goal of improving automation in infrastructure management and provisioning. This automated way of operating resources led to the development of a DevOps practice known as Infrastructure as Code (IaC).

The idea behind IaC is to define the infrastructure, as well as its provisioning, through machine-readable documents. The immediate advantage of adopting this practice is that one can employ software tools to interpret these documents, with the goal of setting up infrastructures automatically. This encourages IT administrators to define every aspect of the system in great detail, which leads to less manual configuration. As a result, the infrastructure becomes highly documented and repeatable. This is of great importance in the context of life science, as repeatability improves reproducibility in computational experiments. In addition, IaC also opens up the possibility of adopting CI/CD for infrastructure management. In this way, as it commonly happens for software, infrastructure can be rigorously updated, tested and deployed, improving quality assurance in its operations.

Numerous IaC languages and automation tools are currently available. IaC languages such as the Hashicorp Configuration Language (HCL) [85], use a declarative approach which enables the description of the desired state of infrastructure entities. HCL, and other declarative IaC languages, can then be parsed by the companion automation tool (Terraform in the case of HCL) creating or applying changes to the existing infrastructure, with the goal of aligning its state to the one defined in the IaC document. In contrast, imperative

IaC languages allow one to script the control flow of the operations, giving a higher degree of flexibility. A prominent example is Apache Libcloud [86] which comes as a Python library, hence allowing code operations using common programming language constructs. While imperative IaC languages give more flexibility in defining the operations' control flow, declarative languages make incremental updates of the infrastructure simpler. For this reason a mix of both approaches is normally what infrastructure developers should strive for. For this reason, IaC tools such as Ansible [87] provide a definition language that enables the mixing of both approaches. Finally, another important characteristic to consider when choosing an IaC language resides in its ability to support multiple cloud providers. Indeed, to avoid vendor lock-in, IaC tools should normally be cloud agnostic.

3.5 MapReduce

I briefly mentioned MapReduce in Section 1.3. This programming model, and associated implementation, was originally developed by Google in the mid 2000s [45], introducing two interesting ideas. Firstly, by bringing processing code to the data, rather then moving the data where the program is stored, one can achieve considerably lower network usage in Big Data use cases. This, coupled with a mechanisms that can tolerate hardware failures, enables the running of data-intensive applications on commodity computer clusters. Secondly, by introducing a clever programming model, inspired by functional programming, an underlying implementation is capable of hiding complex parallel computing tasks such as data distribution, locality-aware scheduling and fault recovery.

In the original definition of MapReduce [45], each jobs takes a set of key/-value pairs as input and outputs a set of processed key/value pairs. The programming model allows the user to define the data analysis by coding two functions: `map` and `reduce`. The `map` function takes the input key/value pairs and produces a set of intermediate key/value pairs – using the user-specified code. The underlying MapReduce implementation groups the intermediate results by key, and passes them to the `reduce` fuction. Finally, the reduce function combines the intermediate results in each group to typically one or zero key/value pairs – according to the user's code.

It is useful to show a simple yet interesting use case in genomics, to make this programming model clearer for the reader. DNA sequences are commonly represented in text files, using a language of 4 characters: `A`, `T`, `G`, `C`. Given a large DNA sequence, it may be useful to count `G` and `C` occurrences; for example there is evidence that GC-rich genes are more efficiently expressed than GC-poor genes [88]. Figure 3.5 shows the pseudocode for counting GC occurrences using the MapReduce programming model. It is common for MapReduce frameworks to split the text data line-wise, producing input pairs where

```
function map(key, value)
    gc_count = count_gc(value)
    return (0, gc_count)

function reduce(key, iterator)
    result = sum_elements(iterator)
    return (key, result)
```

Figure 3.5. **GC count in MapReduce.** The figure shows a MapReduce implementation for counting GC occurrences in a DNA string. The `map` function counts GC occurrences in the input key/value pair and returns 0 as key and the GC count as value. These partial sums are aggregated by the underlying MapReduce implementation in a single group, as we always return 0 as key in the `map` function. The `reduce` function takes an iterator that points to this group, and returns its cumulative sum as value (and again 0 as key).

key and value represent respectively: a line number and the string content at that line. Accordingly, the pseudocode in Figure 3.5 assumes the MapReduce implementation to work as follows: each input key/value pair is processed one by one by the `map` function, which counts GC occurrences in each input line returning 0 as key and the count as value. Please notice that the `map` function returns a fixed value as key, as we eventually want all of the partial counts to be summed together. Indeed, when always returning 0 as key, the underlying MapReduce framework groups the intermediate results as a single group. Hence, the `reduce` function takes an iterator pointing to this group and sums all of its elements together, returning the final result as value and 0 as key.

Google's MapReduce implementation is nowadays legacy. However, it laid the foundations for a vast landscape of open-source frameworks that are build around the same concepts. In such landscape, Apache Hadoop [89] is undoubtedly the most widely adopted MapReduce implementation. In addition, many extending tools and frameworks that overcome some of the limitations of the MapReduce model were implemented as part of the Hadoop ecosystem. Among these Apache Spark is the one that has collected the largest open-source community [90].

3.6 Apache Spark

Apache Spark is a cluster-computing framework for Big Data analytics [48]. The project originated with the aim to overcome the lack of dataset caching in MapReduce implementations, such as Hadoop. The lack of this feature penalizes iterative and interactive appications, where a dataset needs to be frequently accessed. For instance, older versions of Hadoop were posing scalabil-

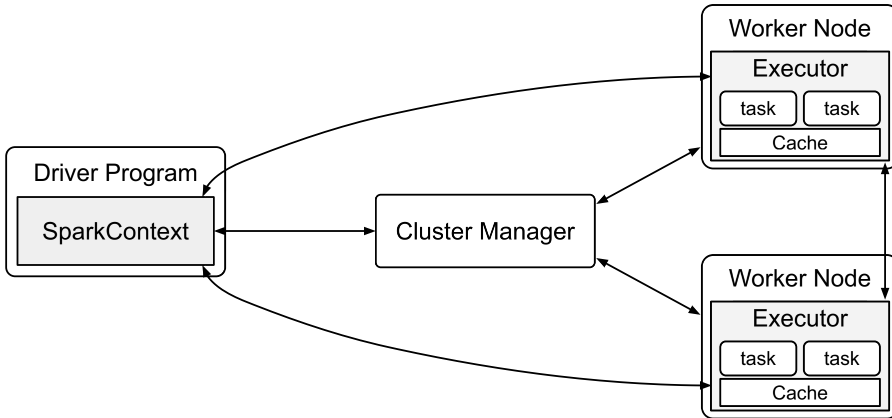


Figure 3.6. **Apache Spark clustering model.** The Apache Spark clustering model consists of a driver program, a cluster manager and one or more worker nodes. The driver program, written by the user, defines the control flow of the analyses using the *SparkContext* object. The *SparkContext* communicates with the cluster manager, which grants physical resources on the worker nodes. Thus, each application runs on a set of isolated executor processes on the granted resources. Finally, the executors, coordinated by the *SparkContext*, execute processing tasks and may keep data persistent on their local cache as needed.

ity issues when implementing Machine Learning (ML) algorithms and interactive dataset querying services [48]. By introducing in-memory processing, as well as a more flexible API, when compared to MapReduce, Apache Spark overcame these issues and immediately attracted a large number of users and contributors. Indeed, as of 2019 Spark is the most active open-source project in Big Data analytics, with more than one thousand contributors and more than one thousand adopting organizations – not to mention its success stories across several domains [90]. This resulted in the development of a number of high-level APIs, which are actively maintained by the Spark community. These include APIs for ML, Structure Query Language (SQL), graph and stream processing.

Figure 3.6 show the Apache Spark clustering model. Spark offers an API that is accessible in Scala [91], Python [71], Java [92] and R [31]. Using one of these APIs, the user writes a driver program that defines the control flow of the data analysis. The *SparkContext* object is central in the control flow definition, as it abstracts the coordination of executor processes across the cluster. When it is instantiated, the *SparkContext* negotiates physical resources with a cluster manager. This can be the stand-alone Spark cluster manager, as well as Hadoop YARN [93] (the Hadoop resource manager), Apache Mesos [94] or Kubernetes. Once physical resources are granted, the executor processes are started across the worker nodes. Each executor process is isolated and multi-threaded, and executes concurrent tasks under the directions of the *SparkCon-*

```scala
1  val gcCount = sparkContext.textFile("hdfs://dna.txt")
2      .map(line => countGC(line))
3      .reduce((count1, count2) => count1 + count2)
```

Figure 3.7. **GC count in Apache Spark.** The figure shows an Apache Spark driver program, written in Scala, that counts GC occurrences in a DNA string. On line 1, using an instance of the *SparkContext*, the program creates an RDD from a text-represented DNA file, stored in a Hadoop File System (HDFS) [95] storage. Then, on line 2, the *map* transformation counts GC occurrences on each line, producing a new RDD. Finally, on line 3, the reduce action sums together all the partial sum from the intermediate RDD, returning a value to a driver program – which is represented by `gcCount` on line 1.

text. Finally, each executor has an allocated memory space for caching data that needs to be accessed iteratively.

When writing the driver program, developers typically use the *SparkContext* to create one of the available distributed dataset abstractions. Here, it is interesting to discuss Resilient Distributed Datasets (RDDs), as they are central in the Spark programming model – and because they are used in Papers I, II and IV. Formally, an RDD is a distributed, fault-tolerant collection of objects that can be manipulated in parallel [96]. RDDs are read-only, and they can be either created by loading data from a storage system or by transforming an existing dataset. Similarly to MapReduce, RDDs are exposed through a functional programming API that enables users to code transformations via local functions that are seamlessly distributed across the cluster. To make this more clear, in Figure 3.7, I show again an example of how to count GC occurrences in a DNA string, but this time using an Apache Spark driver program. The first line creates an RDD that points to a DNA string in a distributed storage. The second line applies a `map` transformation which counts GC occurrences on each input line, returning an intermediate RDD. This new RDD contains a collection of partial sums which are distributed across the cluster. At this point, we want to sum all of these partial results and return the value to the driver program. This is done on line 3 by calling `reduce`. This operation is not a transformation as it does not create a new RDD. In the Spark terminology an operation that returns a value to the driver program is called an action. The difference between transformations and actions is important for understanding the control flow of a driver program. Indeed, transfromations are not blocking and they create RDDs that are computed only when an action is called. This lazy computational model allows Spark to optimize its internal execution plan. For example, if there are two or more `map` transformation in a row, Spark can fuse them in a single stage and compute the result in one pass.

Apart from `map` and `reduce`, RDDs offer many other transformations and actions to perform more complex manipulations of the underlying datasets,

and a comprehensive list can be found in the RDD API documentation [97]. Here, it is interesting to also briefly discuss the `cache()` method from the RDD API, as in-memory processing is one of the main characteristics of Apache Spark. The `cache()` method marks an RDD, so that the Spark engine will keep it persistent in memory after it is computed. You can imagine an interactive setting for the example in figure 3.7. After counting the GC occurrences, the user may want to count the AT occurrences as well. Thus, instead of loading the data again from the distributed storage, one may mark the RDD that is generated on line 1 using the `cache()` method, so that when transforming it again it will not cause further disk access.

To conclude, it is also interesting to discuss the Apache Spark fault tolerance model. Instead of replicating RDD data or using checkpointing strategies, Spark uses an approach known as "lineage" [96]. Simply, each RDD has enough information to recompute lost partitions. In data-intensive processing, this way of dealing with failures is more efficient than performing replication, as it saves both time and storage during processing, while still retaining sufficient performance in case of failure. Indeed, if multiple partitions are lost from a node failure, they can be quickly recomputed in parallel over the healthy workers.

3.7 Conformal Prediction

Machine Learning (ML) is a family of methods to derive predictive models or knowledge from data. In supervised classification settings, data consist of a sequence of training examples $(x_1, y_1), \ldots, (x_l, y_l)$, where the so-called object x_i is an attribute vector and y_i a known label representing the class for the i-th example ($i = 1, \ldots, l$). Given such input, the goal of a ML algorithm is to build a model to predict classes $y_{l+1}, y_{l+2}, \ldots, y_m$ for a set of unseen objects $x_{l+1}, x_{l+2}, \ldots, x_m$ ($m > l$).

Conformal prediction is a method proposed by Vovk et al. [98] that enables the assignment of object-specific confidence values to ML-based predictions. The method applies to any existing ML algorithm and can be used both in supervised classification and regression settings. In classification, instead of predicting a single value y, for an unseen object x, a Conformal Predictor (CP) produces a prediction set $\{y_1, ..., y_k\}$ which is a subset of all of the possible known a priori labels. The size of such prediction set depends on a Non-Conformity Measure (NCM), which is defined before training, and on a user-specified significance level ε. The appealing property of CPs is that Vovk et al. provide proof that any unknown label, which we aim to predict, belongs to the produced prediction set with a probability that is greater or equal to $1 - \varepsilon$ [98] – assuming that examples are exchangeable. In this sense, any CP is valid by construction and evaluations are therefore conducted in terms of

Algorithm	NCM example
Support Vector Machines [99]	Signed distance for the measured example from the separating hyperplane
Random forests [100]	Number of trees that predict the wrong label for the measured example
Neural networks [101]	One minus the output neuron corresponding to the label of the measured example

Table 3.1. *Non-conformity measure examples for three commonly used conformal ML algorithms.*

efficiency. Intuitively, an efficient CP should produce small prediction sets for small user-specified significance levels.

The chosen NCM strongly influences the efficiency of a CP. Formally, a NCM is a function that assigns a measure of strangeness to examples, with respect to the training data. This function is commonly defined by leveraging a predictive model that was previously built from the training data (see Table 3.1 for a few examples) and is fundamental when building a CP.

In the original definition by Vovk et al. CPs were built solely by using transductive learning [98]. This approach has the disadvantage of being computationally expensive, as the ML model needs to be retrained for each new prediction. Luckly, an alternative kind of CP, known as an Inductive Conformal Predictor (ICP), which does not require the retraining of the model for each prediction, was later introduced, overcoming the issue [102].

In classification settings, given a sequence of training examples, an ML algorithm and a NCM, an ICP can be constructed as follows: first, the training data is split into a calibration set and a proper training set. The proper training set, which normally contains approximately 80% of the training data, is used to train the ML model with the given algorithm. The model is then used, together with the NCM, to compute the non-conformity scores $\alpha_1, \ldots, \alpha_q$ for each calibration example (where q is the size of the calibration set).

At this point the ICP, which consists of the model-based NCM and the scores $\alpha_1, \ldots, \alpha_q$, is trained and ready to use for computing prediction sets. For an unseen object x, we consider each of the possible classes $l_1, \ldots, l_k$. For each l_i ($i = 1, \ldots, k$), we first compute the non-conformity score of x with respect to class l_i, which we call $\alpha_x^{l_i}$. Then, the we compute the p-value of x for class l as the following fraction.

$$\frac{|\alpha_x^{l_i} \leq \alpha_i : i = 1, \ldots, q| + 1}{q + 1}$$

The class l_i is then added to the prediction set if the corresponding p-value is greater than or equal to the user-provided significance level ε. Following this procedure the true label of x will be in the prediction set with probability at least $1 - \varepsilon$, assuming that the examples are exchangeable [98].

To conclude this section, it is interesting to discuss how CPs can be applied in practice. To this extent, two major approaches are reported in the literature [102]. First, we can let the user define a significance level, and reject predictions for any CP output that is not a singleton. This approach may be useful in a pipeline environment where examples are continuously evaluated, and we want human intervention for those that cannot be predicted with sufficient confidence. Another approach, which better conforms with the goal of assigning object-specific confidence to predictions, utilizes the p-values that CPs produce for each of the possible classes. Indeed, it is natural to predict the class with the highest corresponding p-value, referred to as credibility, and output one minus the second highest p-value as confidence.

4. Results and discussion

4.1 Scalable cheminformatics analyses on cloud resources with Apache Spark (Papers I and II)

In Papers I and II, we aimed at exploring how the Big Data architecture (see Figure 1.3b) can be taken advantage of to run scalable cheminformatics analyses on cloud resources. Cheminformatics is a field that encompasses numerous computational methods for deriving predictive models and knowledge from chemical data [103]. Chemical information usually consists of molecular structures or other physicochemical properties. This data is represented using standard, machine-readable formats and it is usually stored in text files. MapReduce-oriented frameworks excel at handling such datasets, as text files are easy to split across a computer cluster and because cheminformatics analyses often allow processing of information elements independently. Hence, MapReduce frameworks, and in particular Apache Spark, represent a good opportunity to explore the applicability of Big Data analytics frameworks to cheminformatics analyses – which are otherwise traditionally performed on HPC resources. For this reason, we use Apache Spark in both papers as the underlying cluster-computing platform.

Linking back to the aims of this thesis, in both Paper I and II we improve the accessibility of the distributed analyses by providing the user with high-level APIs to enable the customization of data-processing pipelines while hiding the complexity of the underlying infrastructure. Further, in both papers we built software methods that rely on existing tools to perform data processing. Although in these two papers the methods to access the external tools are specifically designed for the relative use cases, they nevertheless laid the foundations for the generic methods that we show in Papers III and IV. Finally, in Paper II we put strong emphasis on enabling the running of the methods on inexpensive public cloud resources. This is in part enabled by Apache Spark, with which analyses can be performed on commodity hardware whilst retaining good performance.

In the rest of this section I briefly present the results and findings of Papers I and II.

Paper 1 – Conformal prediction in Spark: large-scale machine learning with confidence

In the field of cheminformatics, ML has been frequently employed to build predictive models. These models may be used to predict various compound

```scala
1  // Instantiate an underlying algorithm
2  val lg = new LogisticRegression(properTrainingRDD)
3
4  // Train an ICP
5  val icp = ICP.trainClassifier(
6      lg, numClasses=2, calibrationRDD)
7
8  // Compute predictions over a validation set
9  val predictionSetsRDD = validationRDD.map {
10     example => icp.predict(example.features)
11 }
```

Figure 4.1. **CP in Spark API example.** This example shows how to train an ICP using our RDD-based implementation of the conformal prediction framework. Proper training, calibration and prediction sets are distributed across a computer cluster and abstracted using RDDs. On line 2, the `LogisticRegression` underlying algorithm is instantiated using the proper training set as input. This class is an extension of `UnderlyingAlgorithm` which enables the user to define custom training procedures and NCMs. On line 5, a binary classification ICP is trained using the logistic regression underlying algorithm `lg` and a clalibration set. Finally, on lines 9 to 11, the trained ICP is used to compute prediction sets over the validation set.

properties such as molecular toxicity and binding affinity to drug targets. Such prediction can then contribute to critical decision making in research organizations. For instance, during the development process of new drugs, certain candidate compounds may be discarded if they are predicted to cause dangerous interactions in the human body. Hence, assigning appropriate confidence values to predictions is often crucially important for the successful application of ML methods in this field.

Numerous success stories on the application of CPs in cheminformatics were reported in the literature [104, 105, 106, 107, 108]. Nevertheless, to the best of our knowledge, these were limited to applications on relatively small training sets. In Paper I, we introduce a distributed implementation of the conformal prediction framework which enables both training CPs and computing prediction sets over large-scale datasets. The implementation comes as a thin layer on top of the Apache Spark RDD-based ML library, along with a high-level Scala API (see Figure 4.1 for an example). Although we originally aimed at using the implementation in cheminformatics, our software is generally applicable. We therefore benchmarked the software against the two largest classification datasets that were available at the time on the UC Irvine ML repository [109]. Both datasets were originally introduced by Baldi et al. [110] and they contain simulated data for the signal-versus-background classification problem in high-energy physics. The benchmarks show good

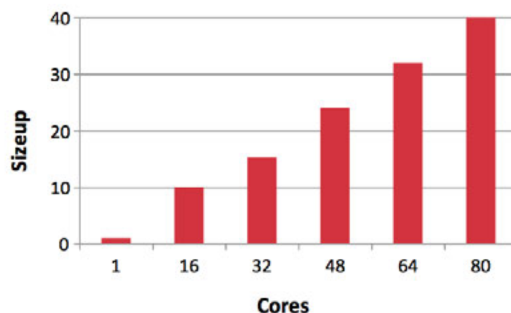


Figure 4.2. **Sizeups for the Spark-based conformal prediction implementation**. The sizeup is a performance metric that indicates how much larger a dataset we can process in the same amount of time by adding compute resources; cores in this case. This plot shows the sizeups for training a CP using our Spark-based implementation. The HIGGS dataset from the UC Irvine ML repository [109] was used as training set. More information about the experimental settings can be found in Paper I.

scalability for our implementation (see Figure 4.2), as well as confirming the validity of the produced prediction sets.

To conclude this section, I would like to mention that after publishing Paper I, we generalized our implementation for use with any existing ML library. This was a relatively simple task, as the conformal prediction framework can be applied to any ML algorithm. Hence, our software can be used with any of the newly-introduced ML libraries in Apache Spark as well as with increasingly popular libraries such as Tensorflow [111]. The implementation is publicly available on GitHub [112].

Paper 2 – Large-scale virtual screening on public cloud resources with Apache Spark

Cheminformatics methods play an important role in the modern drug development life cycle. In particular, in the early development stages of a new drug, Structure-Based Virtual Screening (SBVS) is often employed to identify putative drug leads in a library of molecular representations [113]. A typical SBVS workflow consists of a preliminary preprocessing phase, which adapts the molecular library of choice to the application scope, and a massively parallel molecular docking phase, where a target receptor binding affinity is evaluated against each chemical in the library. This second phase is normally implemented by relying on an external molecular docking software. This external program takes a batch of chemical conformations as input and returns a pose, representing the orientation of a chemical in the target receptor, and a binding affinity score for each of the input chemicals. Thus, after the parallel

```scala
1  val top10 = new SBVSPipeline(sparkContext)
2      .readConformerFile("hdfs://path/to/library.sdf")
3      .dock("/path/to/receptor.oeb",
4          OEDockMethod.Chemgauss4,
5          OESearchResolution.Standard)
6      .getTopPoses(10)
```

Figure 4.3. **SBVS in Spark API example.** The example shows how to setup a simple SBVS pipeline using our Spark-based implementation. On line 1, the `SBVSPipeline` object is initialized passing an instance of the Spark context. On line 2, a molecular library in structure-data file format [123] is loaded from HDFS. On lines 3 to 4, the parallel docking phase is configured by passing a path to a target receptor in the OpenEye binary format, an OpenEye-supported docking method and a search resolution. Finally, on line 6, `getTopPoses` returns the ten highest scoring poses.

docking phase, a number of top-scoring poses can be considered as drug leads for the target receptor of choice.

SBVS is cheaper and faster than high-throughput screening [113], its automated in vitro counterpart, and a few HPC-based implementations, were reported in the literature [114, 115, 116]. In Paper II, we introduce a method that allows to run scalable SBVS pipelines on commodity, public cloud resources using Apache Spark. In the provided open-source implementation [117], we make use of the OpenEye Docking (OEDocking) software [118] to run the parallel molecular docking against batches of chemical representations. The OEDocking software comes as an executable that our implementation calls as an external process using the RDD `pipe` method. Hence, batches of molecular representations are simply passed to the external program and loaded back into Apache Spark using standard input and output respectively.

Similarly to Paper 1, we provide a high level library that reduces the entry barrier to access our method (see Figure 4.3) and we show benchmarks against a real-world use case. In particular, we tested the implementation against SureChEMBL [119], one of the largest libraries of patented chemicals, containing $\sim$2.2M compound (as retrieved in ready-to-dock format from the ZINC database [120]). Further, the benchmark was executed on public cloud resources that were kindly provided by CityCloud [121]. The performance metrics show good scalability (see Figure 4.4) and we provide HCL-based IaC documents to replicate our experimental settings on CityCloud, as well as on any other OpenStack-based cloud platform [122].

To conclude this section, it is interesting to mention that Paper 1 and 2 laid the foundations for a follow up study in which Ahmed et al. introduce an improved iterative method for large-scale SBVS [124].

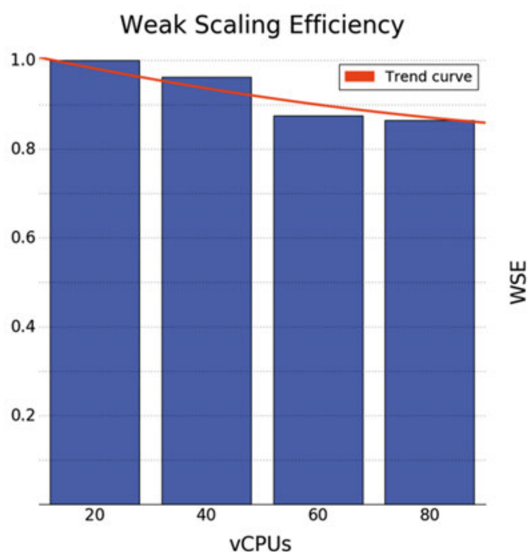


Figure 4.4. **Weak scaling efficiency for the Spark-based virtual screening implementation**. The Weak Scaling Efficiency (WSE) is a performance metric that shows how well a system scales when increasing both the problem size and the processing units – virtual Central Processing Units (vCPUs) in this case – such that the amount of work per processing unit stays constant. This plot shows the WSEs for a virtual screening analysis over the SureChEMBL dataset [119] retrieved in ready-to-dock format from the ZINC database [120]. More information about the experimental settings can be found in Paper II.

4.2 Scalable pipelines on cloud resources with microservices (Paper III)

During my PhD studies I worked extensively as part of the PhenoMeNal consortium [42]. The main contribution that we made in PhenoMeNal was to provide an open-source platform for the analysis of large-scale metabolomics. From the very beginning we made the decision to build PhenoMeNal on top of cloud infrastructure, in line with current trends in the life science data ecosystem. In addition, we set to design the system so that the entry barrier to access the analyses would be as low as possible for bioinformaticians. With this in mind, apart from empowering users in allocating hybrid cloud resources on demand, the PhenoMeNal platform enables users to effectively run data pipelines, that are normally executed on HPC, in cloud-native settings.

In the bioinformatics HPC ecosystems, a WS is usually employed to execute a user-defined DAG by submitting jobs to a resource manager. When implementing the same bioinformatics pipelines in cloud-native settings, we aimed at making the transition between HPC and cloud as smooth as possible. In fact, one of our goals was to enable the users to switch between the two systems as needed. In order to achieve this goal, emulating an HPC environ-

ment on cloud resources is a possibility, and a few software project attempted to provide such a solution [125, 126, 127]. However, the community around high-performance cloud computing is still relatively small – not to mention that in-house clouds are often built on top of commodity hardware. Instead, the majority of the cloud-native community embraces microservices along with their companion architecture and enabling technology. Luckily, resource management in HPC systems is not substantially different from job scheduling in contrainer orchestrators, thus our choice was to define PhenoMeNal as a microservices-oriented application and to implement container orchestration support in commonly used WSs. Since WSs usually hide the complexity of job scheduling, this aspect is often decoupled from the DAG definition, allowing users to run PhenoMeNal pipelines on HPC with minor adaptions.

The benefits of using microservices for bioinformatics pipelines are however not merely limited to the transition of bioinformatics WSs to the cloud-native environment. Indeed, in PhenoMeNal we identified three major additional benefits that application containers bring to bioinformatics analyses – and to scientific workflows in general. Firstly, the improved software delivery mechanism provided by Docker (and similar engines) gives bioinformaticians a higher degree of autonomy in bringing the required software to the computing platform – as opposed to traditional HPC infrastructures where software needs to be approved and installed by system administrators. Furthermore, the improved isolation mechanism provided by application containers allows the heterogeneous scientific software stack ecosystem to share the same computing platform, with no need of managing complex, possibly conflicting dependencies. Secondly, by building container images, bioinformaticians intrinsically document the software "mashup" that is needed for an analysis – which is not seldom lost after the publication of a study [128]. This, together with the adoption of a rigorous CI practice, considerably improves the reproducibility of bioinformatics pipelines. In this sense, CI is also beneficial as it makes sure that the containers comply with the latest software ecosystem. Finally, we found microservices to play an important role in improving the interoperability of scientific software. This is not only because of improved isolation in software components, which allows employing a remarkably heterogeneous ecosystem in the same analysis, but also because microservices-oriented software engineering advocates language-agnostic communication between containers. In PhenoMeNal this contextualized in the adoption of a set of standard format for metabolomics data.

In Paper III, we show four metabolomics analyses that were implemented by the PhenoMeNal consortium with the aim of demonstrating the effectiveness of microservices-oriented pipelines. The four demonstrators were implemented using the Galaxy [36] or the Luigi (by Spotify) [129] WSs. While Galaxy provides a web interface to define DAGs, Luigi provides an API that we serve to the users via Jupyter notebooks. Both WSs were previously employed in HPC environments to run bioinformatics analyses and we imple-

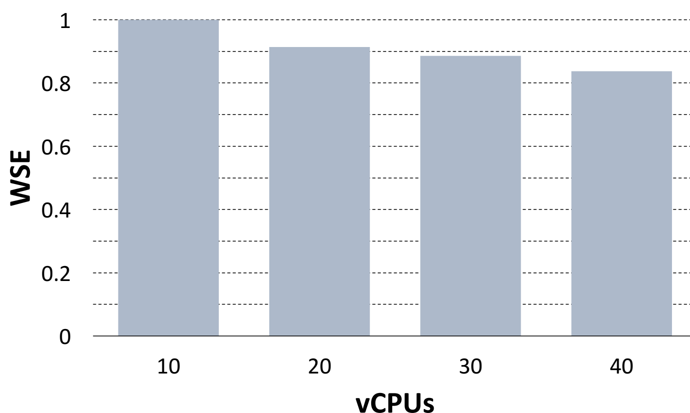


Figure 4.5. **Weak scaling efficiency for Demonstrator 1**. This plot shows the weak scaling efficiency of Demonstrator 1 for an increasing number of vCPUs. The input data comes from one of the largest MS datasets in the Metabolights repository [28]. More information about the experimental settings can be found in Paper III.

mented the support for Kubernetes scheduling in both systems to enable our use cases. To this extent, it is interesting to mention that the Kubernetes support that we developed was merged in both Galaxy and Luigi and is currently maintained by the open-source community.

Demonstrator 1, of which I led the development, is implemented in Luigi and shows good scalability of the proposed system (see Figure 4.5). The pipeline, which I have already introduced in Section 1.2 (see Figure 1.2), is based on one of the largest MS datasets in the Metabolights repository [22] making it well suited for parallelization. Demonstrator 2 is implemented in Galaxy, and is also based on MS data. In this demonstrator the aim is to showcase interoperability in a remarkably heterogeneous set of software tools that may be employed in clinical settings. Lastly, Demonstrators 3 and 4, implemented in Galaxy, aim at showing pipelines that are based on nuclear magnetic resonance and fluxomics technologies. Both technologies employ considerably different data processing techniques, if compared with the more mainstream MS analyses, thus the aim with these demonstrators is to show the generality of the proposed methodology.

4.3 MapReduce-oriented processing with application containers (Paper IV)

The solution that we proposed in the previous section has a few limitations. Firstly, WSs can normally only ingest data from POSIX storage systems. While a few commercial cloud providers offer solutions such as Amazon Elastic File System [130], object storage [131] is usually what most public and in-house clouds provide instead. In addition, although object storage is nowadays usually provided even in commodity clouds, the API landscape for accessing these kind of systems is remarkably heterogeneous. Indeed, each cloud provider usually implements its own object storage API – causing vendor lock-in. Consequently, to enable the demonstrators in Paper III, we set up a cloud-agnostic distributed POSIX storage on top of the IaaS layer – which ended up being impractical as the data needed to be ingested in the providers each time that the resources were allocated on demand. Secondly, WSs usually need a shared storage space for saving intermediate data. As such storage space is often decoupled from the compute resources this leads to poor data locality. Lastly, the batch-oriented nature of WSs makes it hard to define and run interactive, container-based analyses using notebooks systems such as Jupyter and Apache Zeppelin [50, 51].

Ingestion from heterogeneous cloud storage, locality-aware scheduling and interactive processing are commonly provided by Big Data analytics frameworks such as Apache Spark. Hence, the solution that we propose in Paper IV is to implement the support for processing data with application containers on top of the Big Data analytics stack. In more detail, in Paper IV we introduce MaRe, a thin layer on top of Apache Spark, which provides a high-level API for running MapReduce-oriented analyses with Docker containers (see Figure 4.6). Similarly to Papers I and II, the API is based on Scala and is implemented by leveraging the RDD API. This ensures full interoperability with the Apache Spark ecosystem, allowing one to mix pure Spark code and MaRe analyses.

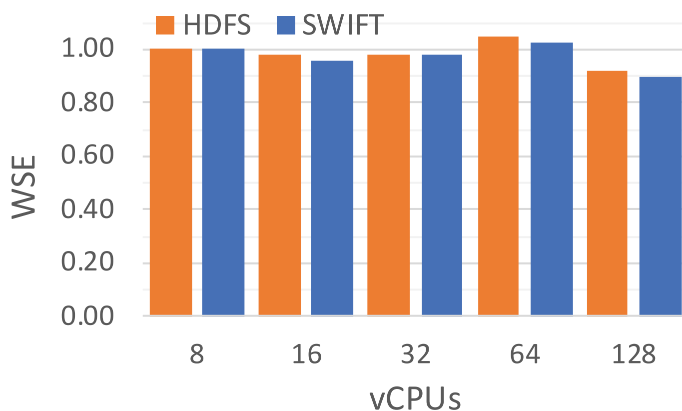
We benchmark MaRe on two real-world use cases. First, similarly to our procedure in Paper II, we show a MaRe implementation for a SBVS pipeline. Again, the molecular docking software that we used was OEDocking, this time provided as a Docker image, and we screened against the SureChEMBL library. Second, we showed how MaRe can be used to implement a genomics analysis for a human dataset retrieved from the 1000 genome project [16], comprising assembly and Single Nucleotide Polymorphism (SNP) calling.

Both analyses can be implemented in MaRe with less than 50 lines of code, as we show in Paper IV. Further, the performance tests, that we ran on the cPouta cloud provider [132], show optimal scaling for the SBVS use case and good scaling for the genomics use case (see Figure 4.7). As we discussed in the paper, the difference in performance between the two demonstators is due to the fact that SNP calling needs large batches of records to be grouped and processed at once, requiring more memory on single nodes and more net-

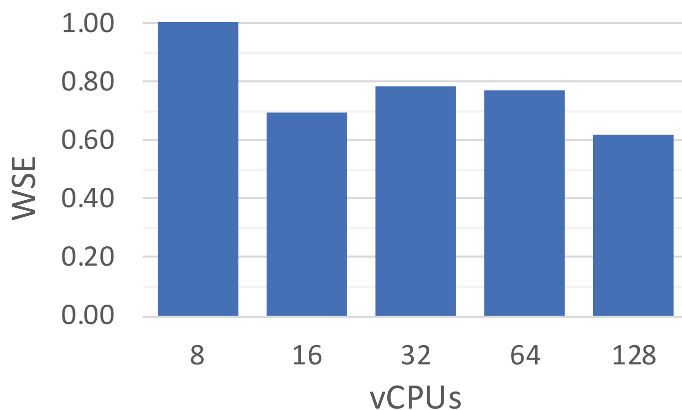
work access. Finally, the scalability metrics are shown against three different cloud storage systems and the analyses can be run interactively using Apache Zeppelin.

```
 1  val gcCount = new MaRe(genomeRDD).map(
 2      inputMountPoint = TextFile("/dna"),
 3      outputMountPoint = TextFile("/count"),
 4      imageName = "ubuntu",
 5      command = """
 6          grep -o '[GC]' /dna | wc -l > /count
 7      """
 8  ).reduce(
 9      inputMountPoint = TextFile("/counts"),
10      outputMountPoint = TextFile("/sum"),
11      imageName = "ubuntu",
12      command = """
13          awk '{s+=$1} END {print s}' /counts > /sum
14      """
15  )
```

Figure 4.6. **GC count using MaRe.** The figure shows how to count GC occurrences in a text-represented genome using MaRe together with an Ubuntu Docker container [133]. Instead of coding data transformations, MaRe enables defining them using commands from one or more application container. On line 1, the `MaRe` object is instantiated passing an input RDD containing the genome in text format. On lines 1 to 8, the *Map* phase of the analysis is defined using the `map` method. The `inputMountPoint` and `outputMountPoint` parameters specify where the RDD splits will be mounted in the container and where the processing command is expected to write the intermediate results. Further, the Ubuntu image is specified on line 4 and a command combining `grep` and `wc`, to count GC occurrences, is provided on line 6. On lines 8 to 10, the *Reduce* phase of the analysis is defined using the `reduce` method, where the first three parameters are similar to the previous `map` method, and where the `awk` command is used to sum together the partial results from the *Map* phase. For further details regarding the MaRe API please refer to Paper IV.



(a) **Weak scaling efficiency for the MaRe-based virtual screening implementation.** The analysis was run over the SureChEMBL dataset [119] retrieved in ready-to-dock format from the ZINC database [120]. The performance metric is shown for two storage backends: HDFS [95] and SWIFT [134]



(b) **Weak scaling efficiency for the MaRe-based single nucleotide polymorphism calling implementation.** The analysis was run over a full human dataset retrieved from the 1000 genome project [16].

Figure 4.7. **Weak scaling efficiency for two real-world, bioinformatics use cases implemented in MaRe.** More information about the experimental settings can be found in Paper IV.

4.4 A blueprint for on-demand virtual research environments (Paper V)

A public instance of PhenoMeNal, which includes a few resources to try out the platform, runs continuously on the EMBL-EBI cloud provider [135]. Users can access the web interface, register an account, and run some simple metabolomics pipelines that we made available, or build their own. This kind of web-based system, which serves the needs of a community of practice, is commonly called a Virtual Research Environment (VRE) [136]. The public PhenoMeNal VRE sits on top of Kubernetes and is therefore designed as a microservices-oriented application. A few other research initiatives, such as the EXTraS project (in astrophysics) [137] and the Square Kilometer Array (SKA) project (in radio astronomy) [138], built similar microservices-based VRE platforms where the resource allocation is either static or operated by IT administrators. PhenoMeNal stands out from the crowd as, in addition to this way of managing resources, we set to empower the users to allocate the required infrastructure on demand, leveraging on cloud systems. The landscape of cloud resources that are available to the PhenoMeNal community are remarkably heterogeneous. On the one side, we have the infrastructure which is usually available in academic settings (which can come as commodity cloud installations, often based on OpenStack), but also as independent workstations or rack servers that are owned by research laboratories. On the other side, we have commercial cloud providers which represent an appealing possibility when data regulations allow for running the analyses on public infrastructure.

PhenoMeNal achieved the goal of enabling the allocation of VRE resources on demand, in both public and private settings. The users can easily instantiate the system either using a public web interface or by using a CLI. Further, the on-demand installer supports the mainstream commercial cloud providers (Amazon [139], Google [140] and Microsoft [141]) as well as OpenStack, the leading on-premise provider, and even local server installations. Basing on our experience in enabling on-demand VREs in PhenoMeNal, and by keeping in mind requirements from similar systems such as EXTraS and SKA, in Paper V we present a general methodology for building on-demand, microservice-oriented VREs.

The layered architecture in figure 4.8 summarizes our methodology on a high level. Similarly to other microservice-oriented applications, on-demand VREs leverage cloud providers to allocate infrastructure on demand and uses a middleware to orchestrate microservices in the uppermost layer. Such microservices are implemented using application containers and they are either long-running or short-lived. Long-running microservices include WSs, monitoring and user interfaces, while short-lived microservices consist of containerized tools, submitted by the WSs. Further, to enable our use case on commodity cloud providers, VREs should give the option to run cloud storage and load balancing (for Internet traffic) as containerized microservices. In-

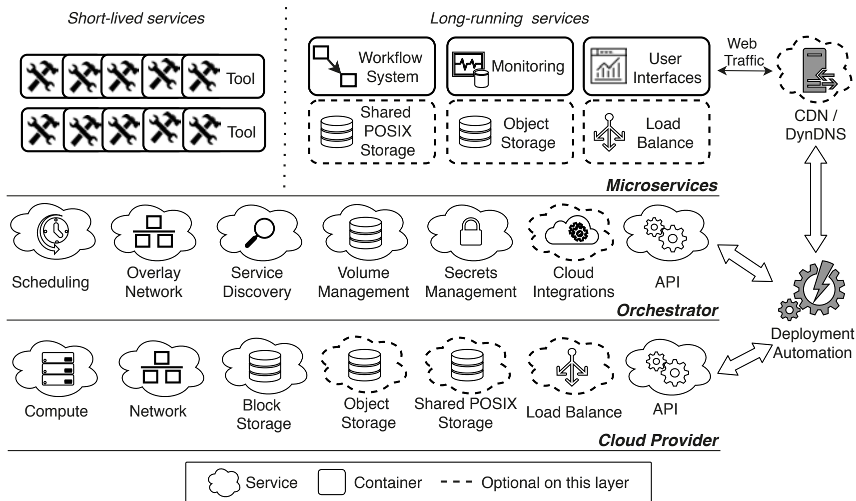


Figure 4.8. **The architecture of on-demand VREs using microservies.** The architecture follows common design practices of microservice-oriented applications and is organized in three layers. The container-based microservices sit on top of an orchestrator that is installed on demand using a deployment automation. On top of the orchestrator, WSs, monitoring and user interfaces run as long-running, container-based microservices, while short-lived, containerized processing tools are instantiated as needed by the WSs. An important aspect of this design is that, to enable VREs on commodity resources, cloud storage and load balancing (which is run in conjunction with a Content Delivery Network and a Dynamic Domain Name System) can be optionally implemented on the microservices layer. More details on the architecture are discussed in Paper V

deed, in our experience, OpenStack-based installation provided by academic institutions seldom offer such services. Finally, deployment automation is central in this design, as it provides the means to abstract resource allocation for the scientists. To this extent such component should operate on multiple levels in the layered architecture and optionally set up domain name system records and a content delivery network [142], to make VRE instantiation as seamless as possible.

With this design in mind, in the early stages of the PhenoMeNal project we used MANTL [143] and Kubespray [144] to automate the VRE deployments. We were active in both open-source communities providing feedback and pushing code to the respective repositories. Nevertheless, both systems had the problem of taking a considerable amount of time in setting up the orchestration middleware – not to mention that MANTL, originally supported by CiscoCloud, was discontinued. When allocating PhenoMeNal on demand this represented a severe issue, as often the VRE allocation took considerably longer than running the analyses. Hence, in Paper V, along with the methodology presented in Figure 4.8, we introduce KubeNow: a comprehen-

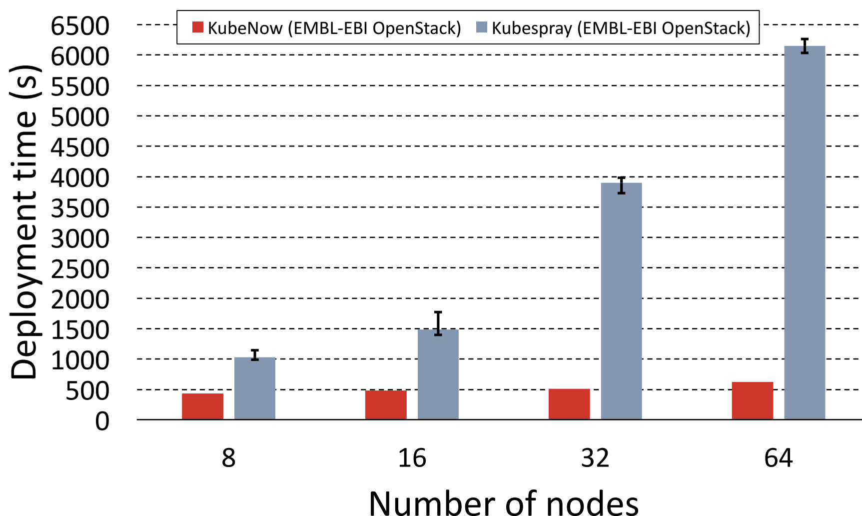


Figure 4.9. **KubeNow and Kubespray deployment time comparison.** The plot shows the deployment time for different cluster sizes (number of nodes) when using KubeNow and when using Kubespray. While the deployment time increases considerably with the number of nodes for Kubespray, for KubeNow it increases by a considerably smaller factor, thus providing better deployment speed scalability. The experiments were performed on the EMBL-EBI OpenStack [135]. More information about the experimental settings can be found in Paper V.

sive platform for quickly instantiating on-demand VREs. KubeNow, of which I have lead the development, considerably speeds up the allocation of VREs by the introduction of two concepts. First, the configuration of cloud instances is fully decentralized and injected via *cloud-init* [145], leading to a better deployment time scalabililty when more and more VMs are allocated. Second, instead of starting from vanilla Linux distributions, KubeNow boots VMs from its own preprovisioned immutable image. The experimental results show how adopting KubeNow for instantiating VREs leads do a dramatic improvement in deployment time scalability, if compared to Kubespray (see Figure 4.9).

The contribution that KubeNow offers to our use case is however not only in terms of deployment speed. The systems comes with an easy-to-use CLI that allow users to configure and instantiate the orchestration middleware, as well as a shared storage system, to enable containerized tools synchronization, and load balancing (if not provided by the underlying cloud service). The resulting virtual infrastructure is contextualized from IaC documents which enabled a robust CI practice for PhenoMeNal VREs. Indeed, KubeNow has been adopted by PhenoMeNal to enable on-demand VREs and it is currently maintained by its community. Further, the project is publicly available on GitHub

[146] and has received good feedback by many open-source users – with over 250 stars and code contributions from a few external developers.

4.5 Lessons learned

Programmatic vs container-based: which wins the battle?

I was lucky enough to start my PhD studies while application containers were just starting to gain popularity in the IT industry. The technology is now quickly becoming ubiquitous in SOA, and I believe that with PhenoMeNal we were among the first to employ it in data processing. The methods that we, and other people working in our area developed, advocate the usage of application containers to encapsulate data-processing software, which is then orchestrated by a WS to run data pipelines. On the other hand, MapReduce-oriented frameworks, which require to code data transformation programmatically, maintain great popularity in the Big Data community. Indeed, Apache Spark is still the largest open-source project in the field. One important point to raise is that defining data transformation programmatically does not necessarily affect portability and reusability. Indeed, code can be packaged and reused in multiple analyses using tools such as Maven [147] and programming languages that support many underlying systems have been available for over a decade [92, 71, 31]. Nevertheless, programming libraries generally do not provide any isolation mechanism and when using many of them at the same time, one will almost inevitably have to deal with conflicting dependencies. Also, software libraries are seldom self-contained, as they may require the installation of external dependencies, and they are normally confined to a single programming language. Application containers nicely solve these problems as they isolate data transformations along with their dependencies, also providing a convenient delivery mechanism. In addition, writing a Dockerfile to define the content of a container image is considerably easier than configuring dependencies with tools like Maven, and it is generally easier to persuade researchers to adopt it as a convenient way to install software on their workstations. To this extent, there is no immediate benefit in coding research software in a way that it will be maintainable, portable and free from conflicting dependencies. Doing this require considerable knowledge of software engineering and researchers seldom get credit for writing good code – academic credit is normally given for new publishable scientific insights that their code is capable of producing. As a result, programmatic definition of data analyses may lead to poor reproducibility. Application containers pose a good remedy to this issue as, in my experience, researchers are more keen to adopt them as a packaging mechanism, if compared to robust software development practices, as they see the immediate benefits of containerized environments. These can simply be in terms of simplicity when installing a bioinformatics software stack, but also a convenient way to keep a workstation free from conflicting

software. Hence, by keenly adopting this relatively easy way of packaging software, researchers intrinsically document the required software "mashup" for their analyses, considerably improving reporducibility.

In terms of performance, executing data transformations with containers poses some issues. Starting an application container as opposed to executing code within the same process certainly causes overheads. Also managing the data flow between containers is harder than the single process scenario where the programming libraries can access the same memory space. In addition, orchestrating container based processing across node introduces a layer of complexity that is hard to cope with. Orchestration platforms such as Kubernetes do a great job at making this management seamless, but they are generally built for SOA – thus mostly for providing highly-available services rather than for Big Data analytics. As a result data processing on orchestrators did not yet reach the maturity that cluster-computing platforms such as Apache Spark have achieved.

As the reader may have already imagined, there is no winner in the battle between programmatic and container-based data processing: it largely depends on the use case at hand. In my opinion, the programmatic environment that Apache Spark, and similar, provide is superior in terms of performance, maturity and interoperability with the cloud-native ecosystem. Thus, when it is possible to implement a use case using a few trusted programming libraries, that do not require complex dependencies, this is the way to go. Nevertheless, this is seldom the case in life science. Bioinformatics tools often come as command-line software, with little support for programmatic APIs, and require being employed with many other tools in heterogeneous pipelines – not to mention the landscape of dependencies that needs to be configured. Even if there are a few community efforts in reimplementing the whole genomics software ecosystem using Apache Spark [49, 148], this is not generally what most research institution would strive for. Indeed, in my experiece, most researchers would rather adopt the container-based approach in order to reuse the trusted toolchain, and pay the price in terms of performance, rather than reimplement the whole ecosystem from scratch.

Lastly, a hybrid programmatic and conatiner-based approach is certainly possible. One can easily build a pipeline that runs MapReduce jobs as well as Kubernetes (or similar) jobs. In addition, my recommendation is to orchestrate consolidated data pipelines with WSs even if the programmatic approach is chosen. Indeed, in my experience submitting analyses manually or executing them interactively is beneficial only in the early stages of a project, when the pipelines are not yet consolidated. Once the analyses are ready for production, WSs provide the means to achieve better automation, interoperability, separation of concern, monitoring and data provenance information.

The advantages of repeatable infrastructure

In KubeNow, and PhenoMeNal, we have adopted IaC to define the required supporting infrastructure for the bioinformatics pipelines. This is somewhat innovative if compared to the traditional model where research organization buy hardware which is then manually set up by IT administrators. Our main motivation for using IaC is to enable automation in conjuction with IaaS and tools such as Ansible and Terraform, therefore making the on-demand allocation of VREs seamless for the researchers. IaC benefits are however not merely in terms of automation. Perhaps the most important advantage of adoption IaC is that the infrastructure becomes repeatable. This means that given the contextualization parameters for the IaC documents, anyone can reproduce the same infrastructure that we used to enable our demonstrators. This is of course beneficial in terms of reproducibility. In addition, as in KubeNow, we put considerable effort into making the IaC documents cloud-agnostic; the immediate advantage that the PhenoMeNal community benefited from is that researchers could simultaneously leverage any of the available cloud providers, without noticing any major difference in the underlying platform. Indeed, by using KubeNow, we were able to simultaneously take advantage of nation-scale academic cloud resource as well as research credits that where kindly provided by commercial clouds – including Amazon Web Services [139], Google Cloud Platform [140], Microsoft Azure [141], City Cloud [121], Helix Nebula Science Cloud [149], CSC c-Pouta [132], EMBL-EBI Embassy [135], SNIC Science Cloud [150] and de.NBI Cloud [151].

Lastly, IaC documents provide a thorough documentation of the computing infrastructure, which may not be the case in the traditional approach where bare-metal systems are manually configured. This bring us to the next section.

The importance of documenting research software

As obvious as it may sound, properly documenting research software plays a major role in making it accessible to other scientists. Yet, in my experience, researchers are too often reluctant to do so. This is of course a direct cause of the fact that it is not good and well documented code that primarily gives academic credit, but rather the scientific insights that it can produce. As a result, a substantial part of the research software is developed in conjunction with a study and readily abandoned as soon as results are published. In bioinformatics this poses issues in terms of reproducibility, as software is central in producing the data that is used for making conclusion.

During my PhD studies I have spent a considerable amount of time in documenting my software, and I believe that apart for improving the reproducibility of my research, that payed back in at least two other areas. First, writing a detailed user documentation for KubeNow is one of the main reasons why the PhenoMeNal community was persuaded in adopting it. Second, machine-

readable documentation, which includes IaC documents, unit and integration tests, improved the sustainability of my software by many folds. Indeed, this enabled a fully-automated testing process which has not only made development faster on my side, but it also considerably sped up the review process of code contributions made by other collaborators.

4.6 Future outlook

Big Data and cloud computing surely are fast-moving fields. In this section I would like to discuss a few emerging technologies that could be beneficial in life science analyses.

First, "serverless" computing is becoming quite a buzzword when talking about cloud-native applications. The term may be misleading, as the technology ultimately requires servers to run code. However, the appealing characteristic of this new cloud-native paradigm is that infrastructure operations are fully managed by the cloud provider. It is still unclear if serverless computing is a subset of PaaS or SaaS and a few methods to hide IaaS operations were often discussed in the literature [152]. However, the currently leading technology uses programming functions as deployment units, and it is referred to as Function as a Service (FaaS). In this service-oriented software development paradigm, users define a set of stateless functions that are automatically invoked in the cloud infrastructure when a user-defined event occurs. Underlying server resources are scaled up and down automatically according to the event arrival rate, and functions are billed for execution time. Please notice that this contrasts with the IaaS pricing model where virtual infrastructure is also billed according to resource flavor – and even when the allocated resources are idle. Linking back to the aims of this thesis, FaaS represent a big step towards the democratizing IaaS operation. In fact, while KubeNow and similarly PaaS still require users to deal with resource parameters tuning such as the number of workers nodes and CPUs per instance, the FaaS model completely delegates this task to the cloud provider; the researchers can truly focus on writing their analyses as programming functions. On the other hand, this model puts strong focus on programmatic definition of the workloads and bringing the vast landscape of required scientific software dependencies in FaaS platforms may represent an issue. Luckily, commercial cloud providers and open-source FaaS, such as Apache OpenWhisk, are starting to provide support for running Docker-based functions, which may solve this issue [152, 153]. Nevertheless, it is still unclear if the FaaS model, which is essentially event-driven, will easily apply to bioinformatics applications – and this is certainly an interesting area of research. Further, if on one hand the FaaS time-oriented pricing model is convenient for CPU-intensive functions, on the other hand it may prove to be ineffective for IO-bound tasks, which characterize Big Data applications. Also, when adopting the technol-

ogy as offered by a certain cloud provider one would almost inevitably incur vendor lock-in. Indeed, each cloud provider adopts its own function definition API and tooling ecosystem. The reader may object that one could deploy an open-source FaaS platform across clouds to avoid this issue, but in my opinion this would make little sense as the operation of such middleware would be again on the user side, while the whole point of using FaaS is to benefit from a fully-managed solution.

Second, the rise of lightweight virtualization technologies, which promises hardware-level isolation while providing the performance of application containers, may reshape cloud-native applications in the future. Indeed, the typical cloud-native stack employes the orchestration layer to cope with the fact that containers ultimately need to be scheduled on VMs, for security concerns. This is in principle an unnecessary complication, thus it would be beneficial if microservices could run straight on top of the IaaS layer. Unikernel technology, originally introduced by the University of Cambridge with MirageOS [154], provides the means to achieve this goal. The developers are provided with a toolchain that enables them to compile minimal and complete application-specific kernels, that can therefore be booted by hypervisors with overheads that are comparable to those of containers. While the technology is interesting, and several unikernel toolchains are currently being developed [154, 155, 156, 157], I think it would be hard to employ it in life science, in the current status. Indeed, unikernel toolchains can be seen as light-weight bare metal libraries and they often only support a single programming language. Thus, the current state of the art would require us to completely reimplement the bioinformatics codebase to enable a shift towards this technology – which is obviously unfeasible. Another interesting, somewhat similar, project is Kata containers [158]. Rather than focusing on performance, the Kata containers project made a remarkable effort in integrating Docker and OCI runtimes with hypervisors. In this way, one can trade some performance for certain microservices that require better security standards, and deploy them as VMs alongside containers on the orchestration platform of choice.

To conclude, I would like to point out that I see great potential in the convergence of cloud and HPC. Even though the two communities of practice focus on different goals, a few efforts in bringing HPC solutions to cloud and vice versa were made. As I have already mentioned in Section 4.2 a few attempts to emulate HPC on cloud were reported and Singularity [159], a container engine for HPC, is becoming more and more popular. Even though the community around these solutions is small, if compared to the community for cloud-native applications, these examples indicate that research at the intersection of the two worlds is being made – and we will surely see some exciting advancement in the future.

5. Conclusions

In this thesis we developed and applied methods for scaling life science analyses on cloud infrastructures. To this extent, apart from showing good scalability for the methods, a strong emphasis was put on improving the accessibility of the technology for life scientists. In summary, the main conclusions of this work are as follows:

- We have shown the applicability of MapReduce-oriented processing to cheminformatics and genomics use cases (Papers I, II and IV).
- We contributed in developing methods for microservice-oriented bioinformatics pipelines and showed their applicability to metabolomics, cheminiformatics and genomics use cases (Papers III and IV).
- We developed high-level APIs for improving the accessibility of MapReduce-oriented and microserivice-oriented processing for life science use cases (Papers I, II, III and IV).
- We improved the accessibility of on-demand cloud infrastructure for researchers (Paper IV).
- We show that our methods scale reasonably well on cloud resources (Papers I, II, III, IV and V).

Throughout the work we learned the importance of repeatable infrastructure and documentation in research software, as well as the tradeoffs and advantages of container-based data processing. Further, I envision that emerging technologies, such as FaaS, unikernels, Kata containers and high-performance cloud computing, could potentially be employed to advance the work that I have presented in this thesis. Finally, the code that I developed during my PhD studies is open source and available on GitHub [160].

6. Acknowledgements

This work was carried out at the Department of Pharmaceutical Biosciences, Division of Pharmacology, Faculty of Pharmacy, Uppsala University, Sweden and at the Department of Information Technology, Division of Scientific Computing, Faculty of Mathematics and Computer Science, Uppsala University, Sweden.

I would like to thank all of the people that supported me throughout my studies and in particular:

My main supervisor **Ola Spjuth** for patiently teaching me how to be a researcher. Thanks for giving me this opportunity and for always giving me the freedom to try out my ideas. I am surely going to miss you.

My co-supervisor **Salman Toor**. Thanks for the insightful discussions on cloud computing and for your positive attitude throughout this years.

My co-supervisor **Lars Carlsson** for the insightful discussions on conformal prediction and machine learning. Also, the dinner and drinks that we had together after the conference in Pattaya were surely memorable.

My fellow KubeNow developers **Anders Larsson**, **Matteo Carone** and **Jon Ander Novella**. Thank you so much for helping me in making the project fly. Also, a special thanks goes to **Noureddin Sadawi** and **Jianliang Gao** for patiently testing KubeNow and reporting issues. Your feedback was really valuable.

My colleague **Phil Harrison** for proofreading this thesis.

My colleague **Samuel Lampa** for some of the best discussions on workflow systems, reproducibility and software development.

Kim Kultima and my fellow PhD students **Payam Emami** and **Stephanie Hermann**. Thank you so much for helping me with metabolomics.

Collegues from Pharmbio and affiliated institutions **Maris Lapins**, **Wesley Schaal**, **Jonathan Alvarsson**, **Tanya Aggarwal**, **Niharika Gauraha**, **Arvid Berg**, **Martin Dahlö**, **Staffan Arvidsson McShane**, **Polina Georgiev**, **Valentin Georgiev**, **Laeeq Ahmed**, **Rikard Nyström**, **Oliver Stein**, **Morgan Ekmefjord**, **Victor Malmsjö**, **Juan Inda** and **Aishvarya Tandon**. Fika and Tuesday's lunches at Yukikos have been some of the best time during my PhD studies.

All of the people involved in PhenoMeNal and in particular **Christoph Steinbeck**, **Kristian Peters**, **James Bradbury**, **Timothy Ebbels**, **Kenneth Haug**, **David Johnson**, **Namrata Kale**, **Pablo Moreno**, **Steffen Neumann**, **Marco Enrico Piras**, **Luca Pireddu**, **Philippe Rocca-Serra**, **Pierrick Roger**, **Antonio Rosato**, **Rico Rueedi**, **Christoph Ruttkies**, **Reza M Salek**, **Michael van Vliet** and **Gianluigi Zanetti**.

Fellow PhD students and professors from the IT department **Andreas Hellander**, **Adrien Coulier**, **Sonja Mathias**, **Fredrik Wrede**, **Ben Blamey**, **Pavol Bauer**, **Lina von Sydow**, **Sverker Holmgren** and **Anna Eckerdal**.

All **co-authors** that I didn't mention previously.

All of the **amazing people** that I've hanged out with throughout the years of my PhD and master studies. It's just impossible to mention everyone and I am sure I'd forget someone if I tried. I am grateful for all the time we've spent together in Uppsala and will surely miss you all when I'll leave.

The radio show **La Zanzara**, which I always listen to in the evening after work, as well as **Germano Mosconi**, **Mario Magnotta** and **Nonno Fiorucci**. Your aphorisms where often in my mouth when experiments didn't come quite right or code didn't compile.

My parents **Carla** and **Artemio**.

Lastly and most importantly **Hedda**, my significant other, for translating the thesis summary to Swedish and for sharing my life.

Marco Capuccini

References

[1] S. Sagiroglu and D. Sinanc, “Big data: A review,” in *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pp. 42–47, IEEE, 2013.
[2] J. H. Faghmous and V. Kumar, “A big data guide to understanding climate change: The case for theory-guided data science,” *Big data*, vol. 2, no. 3, pp. 155–163, 2014.
[3] P. Clarke, P. Coveney, A. Heavens, J. Jäykkä, B. Joachimi, A. Karastergiou, N. Konstantinidis, A. Korn, R. Mann, J. McEwen, *et al.*, “Big data in the physical sciences: challenges and opportunities,”
[4] T. Preis, H. S. Moat, and H. E. Stanley, “Quantifying trading behavior in financial markets using google trends,” *Scientific reports*, vol. 3, p. 1684, 2013.
[5] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, “End to end learning for self-driving cars,” *arXiv preprint arXiv:1604.07316*, 2016.
[6] V. Mayer-Schönberger and K. Cukier, *Big data: A revolution that will transform how we live, work, and think.* Houghton Mifflin Harcourt, 2013.
[7] International Human Genome Sequencing Consortium *et al.*, “Initial sequencing and analysis of the human genome,” *nature*, vol. 409, no. 6822, p. 860, 2001.
[8] F. S. Collins and V. A. McKusick, “Implications of the human genome project for medical science,” *Jama*, vol. 285, no. 5, pp. 540–544, 2001.
[9] N. Ganguly, R. Bano, and S. Seth, “Human genome project: pharmacogenomics and drug development,” 2001.
[10] A. R. Templeton, “Uses of evolutionary theory in the human genome project,” *Annual Review of Ecology and Systematics*, vol. 30, no. 1, pp. 23–49, 1999.
[11] M. Stoneking, “The human genome project and molecular anthropology.,” *Genome Research*, vol. 7, no. 2, pp. 87–91, 1997.
[12] M. Wadman, “Economic return from human genome project grows,” *Nature*, vol. 10, 2013.
[13] “The cost of sequencing a human genome.” `https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost`. Accessed: 2019-07-15.
[14] S. C. Schuster, “Next-generation sequencing transforms today’s biology,” *Nature methods*, vol. 5, no. 1, p. 16, 2007.
[15] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, “Big data: Astronomical or genomical?,” *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.
[16] 1000 Genomes Project Consortium *et al.*, “A global reference for human genetic variation,” *Nature*, vol. 526, no. 7571, p. 68, 2015.

[17] C. E. Cook, R. Lopez, O. Stroe, G. Cochrane, C. Brooksbank, E. Birney, and R. Apweiler, “The European Bioinformatics Institute in 2018: tools, infrastructure and training,” *Nucleic Acids Research*, vol. 47, pp. D15–D22, 11 2018.

[18] S. Goodwin, J. D. McPherson, and W. R. McCombie, “Coming of age: Ten years of next-generation sequencing technologies,” *Nature Reviews Genetics*, vol. 17, no. 6, p. 333, 2016.

[19] UniProt Consortium, “Uniprot: A hub for protein information,” *Nucleic acids research*, vol. 43, no. D1, pp. D204–D212, 2014.

[20] J. A. Vizcaíno, R. G. Côté, A. Csordas, J. A. Dianes, A. Fabregat, J. M. Foster, J. Griss, E. Alpi, M. Birim, J. Contell, *et al.*, “The PRoteomics IDEntifications (PRIDE) database and associated tools: Status in 2013,” *Nucleic acids research*, vol. 41, no. D1, pp. D1063–D1069, 2012.

[21] A. Gaulton, L. J. Bellis, A. P. Bento, J. Chambers, M. Davies, A. Hersey, Y. Light, S. McGlinchey, D. Michalovich, B. Al-Lazikani, *et al.*, “ChEMBL: A large-scale bioactivity database for drug discovery,” *Nucleic acids research*, vol. 40, no. D1, pp. D1100–D1107, 2011.

[22] K. Haug, R. M. Salek, P. Conesa, J. Hastings, P. de Matos, M. Rijnbeek, T. Mahendraker, M. Williams, S. Neumann, P. Rocca-Serra, *et al.*, “MetaboLights–an open-access general-purpose repository for metabolomics studies and associated meta-data,” *Nucleic acids research*, vol. 41, no. D1, pp. D781–D786, 2012.

[23] I. Lappalainen, J. Almeida-King, V. Kumanduri, A. Senf, J. D. Spalding, G. Saunders, J. Kandasamy, M. Caccamo, R. Leinonen, B. Vaughan, *et al.*, “The european genome-phenome archive of human data consented for biomedical research,” *Nature genetics*, vol. 47, no. 7, p. 692, 2015.

[24] A. Brazma, H. Parkinson, U. Sarkans, M. Shojatalab, J. Vilo, N. Abeygunawardena, E. Holloway, M. Kapushesky, P. Kemmeren, G. G. Lara, *et al.*, “ArrayExpress–a public repository for microarray gene expression data at the EBI,” *Nucleic acids research*, vol. 31, no. 1, pp. 68–71, 2003.

[25] C. C. Darie, “Mass spectrometry and its applications in life sciences,” *Australian Journal of Chemistry*, vol. 66, no. 7, pp. 719–720, 2013.

[26] A. Kumar, G. Goel, E. Fehrenbach, A. Puniya, and K. Singh, “Microarrays: the technology, analysis and application,” *Engineering in Life Sciences*, vol. 5, no. 3, pp. 215–222, 2005.

[27] R. T. Hersh, “Atlas of protein sequence and structure,” *Systematic Biology*, vol. 16, no. 3, pp. 262–263, 1967.

[28] C. Ranninger, L. E. Schmidt, M. Rurik, A. Limonciel, P. Jennings, O. Kohlbacher, and C. G. Huber, “Improving global feature detectabilities through scan range splitting for untargeted metabolomics by high-performance liquid chromatography-orbitrap mass spectrometry,” *Analytica chimica acta*, vol. 930, pp. 13–22, 2016.

[29] R. J. Weber, T. N. Lawson, R. M. Salek, T. M. Ebbels, R. C. Glen, R. Goodacre, J. L. Griffin, K. Haug, A. Koulman, P. Moreno, *et al.*, “Computational tools and workflows in metabolomics: An international survey highlights the opportunity for harmonisation through galaxy,” *Metabolomics*, vol. 13, no. 2, p. 12, 2017.

[30] M. Sturm, A. Bertsch, C. Gröpl, A. Hildebrandt, R. Hussong, E. Lange, N. Pfeifer, O. Schulz-Trieglaff, A. Zerck, K. Reinert, *et al.*, "OpenMS–an open-source software framework for mass spectrometry," *BMC bioinformatics*, vol. 9, no. 1, p. 163, 2008.

[31] R. Ihaka and R. Gentleman, "R: A language for data analysis and graphics," *Journal of computational and graphical statistics*, vol. 5, no. 3, pp. 299–314, 1996.

[32] G. Duck, G. Nenadic, M. Filannino, A. Brass, D. L. Robertson, and R. Stevens, "A survey of bioinformatics database and software usage through mining the literature," *PloS one*, vol. 11, no. 6, p. e0157989, 2016.

[33] M. Dahlö, D. G. Scofield, W. Schaal, and O. Spjuth, "Tracking the NGS revolution: managing life science research on shared high-performance computing clusters," *GigaScience*, vol. 7, 04 2018.

[34] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Res.*, vol. 20, pp. 1297–1303, Sep 2010.

[35] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nat. Biotechnol.*, vol. 35, pp. 316–319, 04 2017.

[36] E. Afgan, D. Baker, M. van den Beek, D. Blankenberg, D. Bouvier, M. Cech, J. Chilton, D. Clements, N. Coraor, C. Eberhard, B. Gruning, A. Guerler, J. Hillman-Jackson, G. Von Kuster, E. Rasche, N. Soranzo, N. Turaga, J. Taylor, A. Nekrutenko, and J. Goecks, "The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update," *Nucleic Acids Res.*, vol. 44, pp. W3–W10, 07 2016.

[37] S. Lampa, M. Dahlö, J. Alvarsson, and O. Spjuth, "SciPipe: A workflow library for agile development of complex and dynamic bioinformatics pipelines," *GigaScience*, vol. 8, 04 2019.

[38] J. A. Novella, P. Emami Khoonsari, S. Herman, D. Whitenack, M. Capuccini, J. Burman, K. Kultima, and O. Spjuth, "Container-based bioinformatics with Pachyderm," *Bioinformatics*, vol. 35, pp. 839–846, 08 2018.

[39] G. A. Van der Auwera, M. O. Carneiro, C. Hartl, R. Poplin, G. Del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, E. Banks, K. V. Garimella, D. Altshuler, S. Gabriel, and M. A. DePristo, "From FastQ data to high confidence variant calls: The Genome Analysis Toolkit best practices pipeline," *Curr Protoc Bioinformatics*, vol. 43, pp. 1–33, 2013.

[40] P. A. Ewels, A. Peltzer, S. Fillinger, J. Alneberg, H. Patel, A. Wilm, M. U. Garcia, P. D. Tommaso, and S. Nahnsen, "nf-core: Community curated bioinformatics pipelines," *bioRxiv*, 2019.

[41] F. Giacomoni, G. Le Corguille, M. Monsoor, M. Landi, P. Pericard, M. Petera, C. Duperier, M. Tremblay-Franco, J. F. Martin, D. Jacob, S. Goulitquer, E. A. Thevenot, and C. Caron, "Workflow4Metabolomics: A collaborative research infrastructure for computational metabolomics," *Bioinformatics*, vol. 31, pp. 1493–1495, May 2015.

[42] K. Peters, J. Bradbury, S. Bergmann, M. Capuccini, M. Cascante, P. de Atauri,

T. M. D. Ebbels, C. Foguet, R. Glen, A. Gonzalez-Beltran, U. L. Gunther, E. Handakas, T. Hankemeier, K. Haug, S. Herman, P. Holub, M. Izzo, D. Jacob, D. Johnson, F. Jourdan, N. Kale, I. Karaman, B. Khalili, P. Emami Khonsari, K. Kultima, S. Lampa, A. Larsson, C. Ludwig, P. Moreno, S. Neumann, J. A. Novella, C. O'Donovan, J. T. M. Pearce, A. Peluso, M. E. Piras, L. Pireddu, M. A. C. Reed, P. Rocca-Serra, P. Roger, A. Rosato, R. Rueedi, C. Ruttkies, N. Sadawi, R. M. Salek, S. A. Sansone, V. Selivanov, O. Spjuth, D. Schober, E. A. Thevenot, M. Tomasoni, M. van Rijswijk, M. van Vliet, M. R. Viant, R. J. M. Weber, G. Zanetti, and C. Steinbeck, "PhenoMeNal: Processing and analysis of metabolomics data in the cloud," *Gigascience*, vol. 8, 02 2019.

[43] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Disk-locality in datacenter computing considered irrelevant," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2011.

[44] M. Resch, T. Boenisch, M. Gienger, and B. Koller, *High Performance Computing: Challenges and Risks for the Future: Vol 3: Advanced Intelligent Systems Applied to Environment*, pp. 249–257. 01 2019.

[45] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[46] R. C. Taylor, "An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics," *BMC Bioinformatics*, vol. 11 Suppl 12, p. S1, Dec 2010.

[47] M. Bhandarkar, "MapReduce programming with Apache Hadoop," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–1, IEEE, 2010.

[48] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[49] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson, "ADAM: Genomics formats and processing patterns for cloud scale computing," *University of California, Berkeley Technical Report, No. UCB/EECS-2013*, vol. 207, p. 2013, 2013.

[50] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, *et al.*, "Jupyter notebooks–a publishing format for reproducible computational workflows.," in *ELPUB*, pp. 87–90, 2016.

[51] Y. Cheng, F. C. Liu, S. Jing, W. Xu, and D. H. Chau, "Building big data processing and visualization pipeline through Apache Zeppelin," in *Proceedings of the Practice and Experience on Advanced Research Computing*, p. 57, ACM, 2018.

[52] S. Lampa, M. Dahlo, P. I. Olason, J. Hagberg, and O. Spjuth, "Lessons learned from implementing a national infrastructure in Sweden for storage and analysis of next-generation sequencing data," *Gigascience*, vol. 2, p. 9, Jun 2013.

[53] M. P. Forum, "MPI: A message-passing interface standard," tech. rep., Knoxville, TN, USA, 1994.

[54] "Facebook pushes the limits of Hadoop."

https://www.infoworld.com/article/2616022/facebook-pushes-the-limits-of-hadoop.html. Accessed: 2019-07-15.

[55] "The evolution of Hadoop at Spotify–through failures and pain." https://conferences.oreilly.com/strata/big-data-conference-ca-2015/public/schedule/detail/38595. Accessed: 2019-07-15.

[56] J. Dongarra, "Trends in high performance computing: A historical overview and examination of future developments," *IEEE Circuits and Devices Magazine*, vol. 22, pp. 22–27, Jan 2006.

[57] O. Yildiz and S. Ibrahim, "On the performance of Spark on HPC systems: Towards a complete picture," in *Supercomputing Frontiers* (R. Yokota and W. Wu, eds.), (Cham), pp. 70–89, Springer International Publishing, 2018.

[58] L. D. Stein, "The case for cloud computing in genome informatics," *Genome Biol.*, vol. 11, no. 5, p. 207, 2010.

[59] P. W. Harrison, B. Alako, C. Amid, A. Cerdeno-Tarraga, I. Cleland, S. Holt, A. Hussein, S. Jayathilaka, S. Kay, T. Keane, R. Leinonen, X. Liu, J. Martinez-Villacorta, A. Milano, N. Pakseresht, J. Rajan, K. Reddy, E. Richards, M. Rosello, N. Silvester, D. Smirnov, A. L. Toribio, S. Vijayaraja, and G. Cochrane, "The European Nucleotide Archive in 2018," *Nucleic Acids Res.*, vol. 47, pp. D84–D88, Jan 2019.

[60] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[61] "Django." https://www.djangoproject.com/. Accessed: 2019-07-15.

[62] V. Marx, "Biology: The big challenges of big data," *Nature*, vol. 498, pp. 255–260, Jun 2013.

[63] S. Dzombeta, V. Stantchev, R. Colomo-Palacios, K. Brandis, and K. Haufe, "Governance of cloud computing services for the life sciences," *IT Professional*, vol. 16, pp. 30–37, July 2014.

[64] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[65] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*, vol. 7. University of California, Irvine Doctoral dissertation, 2000.

[66] Open Container Initiative, "The 5 principles of Standard Containers." https://github.com/opencontainers/runtime-spec/blob/master/principles.md, Dec. 2016. Accessed: 2019-07-15.

[67] S. J. Vaughan-Nichols, "Containers vs. virtual machines: How to tell which is the right choice for your enterprise." https://www.networkworld.com/article/3068392/cloud-storage/containers-vs-virtual-machines-how-to-tell-which-is-the-right-choice-for-your-enterprise.html, 2016. Accessed: 2019-07-15.

[68] P.-H. Kamp and R. N. Watson, "Jails: Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, vol. 43, p. 116, 2000.

[69] P. B. Menage, “Adding generic process containers to the linux kernel,” in *Proceedings of the Linux symposium*, vol. 2, pp. 45–57, Citeseer, 2007.

[70] “LinuX Containers.” `https://linuxcontainers.org`. Accessed: 2019-07-15.

[71] M. F. Sanner *et al.*, “Python: A programming language for software integration and development,” *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999.

[72] C. Anderson, “Docker [software engineering],” *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015.

[73] J. Shieber and F. Lardinois, “Docker has raised $92 million in new funding.” `https://techcrunch.com/2018/10/15/docker-has-raised-92-million-in-new-funding`, Oct. 2018. Accessed: 2019-07-15.

[74] “Kubernetes.” `https://kubernetes.io`. Accessed: 2019-07-15.

[75] “Mesosphere.” `https://d2iq.com/solutions/mesosphere`. Accessed: 2019-07-15.

[76] “Docker Swarm.” `https://docs.docker.com/engine/swarm`. Accessed: 2019-07-15.

[77] “Hashicorp Nomad.” `https://www.nomadproject.io`. Accessed: 2019-07-15.

[78] M. Asay, “Why Kubernetes is winning the container war.” `http://www.infoworld.com/article/3118345/cloud-computing/why-kubernetes-is-winning-the-container-war.html`, Sept. 2016. Accessed: 2019-07-15.

[79] A. Khan, “Key characteristics of a container orchestration platform to enable a modern application,” *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017.

[80] Y. Park, H. Yang, and Y. Kim, “Performance analysis of CNI (Container Networking Interface) based container network,” in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 248–250, IEEE, 2018.

[81] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, “Role-based access control models,” *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[82] “GitLab.” `https://about.gitlab.com`. Accessed: 2019-07-15.

[83] “Jenkins.” `https://jenkins.io`. Accessed: 2019-07-15.

[84] G. Shipley and G. Dumpleton, *OpenShift for Developers: A Guide for Impatient Beginners*. "O’Reilly Media, Inc.", 2016.

[85] N. Asthana, T. Chefalas, A. Karve, A. Segal, M. Dubey, and S. Zeng, “A declarative approach for service enablement on hybrid cloud orchestration engines,” in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–7, IEEE, 2018.

[86] “Apache Libcloud.” `https://libcloud.apache.org/`. Accessed: 2019-07-15.

[87] “Ansible.” `https://www.ansible.com`. Accessed: 2019-07-15.

[88] G. Kudla, L. Lipinski, F. Caffin, A. Helwak, and M. Zylicz, “High guanine and cytosine content increases mrna levels in mammalian cells,” *PLoS biology*, vol. 4, no. 6, p. e180, 2006.

[89] “Apache Hadoop.” `https://hadoop.apache.org/`. Accessed: 2019-07-15.

[90] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, “Apache Spark: A unified

engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[91] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima Inc, 2008.

[92] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.

[93] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache Hadoop Yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.

[94] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center.," in *NSDI*, vol. 11, pp. 22–22, 2011.

[95] K. Shvachko, H. Kuang, S. Radia, R. Chansler, *et al.*, "The Hadoop Distributed File System.," in *MSST*, vol. 10, pp. 1–10, 2010.

[96] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[97] "RDD API documentation." `https://spark.apache.org/docs/latest/rdd-programming-guide.html`. Accessed: 2019-07-15.

[98] V. Vovk, A. Gammerman, and G. Shafer, *Algorithmic learning in a random world*. Springer Science & Business Media, 2005.

[99] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[100] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1, pp. 278–282, IEEE, 1995.

[101] A. K. Jain, J. Mao, and K. Mohiuddin, "Artificial neural networks: A tutorial," *Computer*, no. 3, pp. 31–44, 1996.

[102] H. Papadopoulos, "Inductive conformal prediction: Theory and application to neural networks," in *Tools in artificial intelligence*, IntechOpen, 2008.

[103] S. Chonde and S. Kumara, "Cheminformatics: An introductory review," in *IIE Annual Conference. Proceedings*, p. 2316, Institute of Industrial and Systems Engineers (IISE), 2014.

[104] M. Eklund, U. Norinder, S. Boyer, and L. Carlsson, "The application of conformal prediction to the drug discovery process," *Annals of Mathematics and Artificial Intelligence*, vol. 74, no. 1-2, pp. 117–132, 2015.

[105] U. Norinder, L. Carlsson, S. Boyer, and M. Eklund, "Introducing conformal prediction in predictive modeling. a transparent and flexible alternative to applicability domain determination," *Journal of chemical information and modeling*, vol. 54, no. 6, pp. 1596–1603, 2014.

[106] M. Eklund, U. Norinder, S. Boyer, and L. Carlsson, "Application of conformal prediction in QSAR," in *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pp. 166–175, Springer, 2012.

[107] F. Svensson, U. Norinder, and A. Bender, "Modelling compound cytotoxicity using conformal prediction and PubChem HTS data," *Toxicology research*, vol. 6, no. 1, pp. 73–80, 2017.
[108] M. Lapins, S. Arvidsson, S. Lampa, A. Berg, W. Schaal, J. Alvarsson, and O. Spjuth, "A confidence predictor for logD using conformal regression and a support-vector machine," *Journal of cheminformatics*, vol. 10, no. 1, p. 17, 2018.
[109] A. Asuncion and D. Newman, "UCI machine learning repository," 2007.
[110] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, p. 4308, 2014.
[111] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.
[112] "Scala-CP." `https://github.com/mcapuccini/scala-cp`. Accessed: 2019-07-15.
[113] T. Cheng, Q. Li, Z. Zhou, Y. Wang, and S. H. Bryant, "Structure-based virtual screening for drug discovery: a problem-centric review," *The AAPS journal*, vol. 14, no. 1, pp. 133–141, 2012.
[114] G. D. Guerrero, H. E. Perez-S, J. M. Cecilia, J. M. Garcia, *et al.*, "Parallelization of virtual screening in drug discovery on massively parallel architectures," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 588–595, IEEE, 2012.
[115] Y. Fang, Y. Ding, W. P. Feinstein, D. M. Koppelman, J. Moreno, M. Jarrell, J. Ramanujam, and M. Brylinski, "GeauxDock: Accelerating structure-based virtual screening with heterogeneous computing," *PloS one*, vol. 11, no. 7, p. e0158898, 2016.
[116] A. P. Norgan, P. K. Coffman, J.-P. A. Kocher, D. J. Katzmann, and C. P. Sosa, "Multilevel parallelization of AutoDock 4.2," *Journal of cheminformatics*, vol. 3, no. 1, p. 12, 2011.
[117] "Spark-VS." `https://github.com/mcapuccini/spark-vs`. Accessed: 2019-07-15.
[118] M. McGann, "FRED and HYBRID docking performance on standardized datasets," *Journal of computer-aided molecular design*, vol. 26, no. 8, pp. 897–906, 2012.
[119] G. Papadatos, M. Davies, N. Dedman, J. Chambers, A. Gaulton, J. Siddle, R. Koks, S. A. Irvine, J. Pettersson, N. Goncharoff, *et al.*, "SureChEMBL: A large-scale, chemically annotated patent document database," *Nucleic acids research*, vol. 44, no. D1, pp. D1220–D1228, 2015.
[120] J. J. Irwin and B. K. Shoichet, "ZINC–a free database of commercially available compounds for virtual screening," *Journal of chemical information and modeling*, vol. 45, no. 1, pp. 177–182, 2005.
[121] "CityCloud." `https://www.citycloud.com/`. Accessed: 2019-07-15.
[122] "Terraform Apache Spark module for OpenStack." `https://github.com/mcapuccini/terraform-openstack-spark`. Accessed: 2019-07-15.

[123] A. Dalby, J. G. Nourse, W. D. Hounshell, A. K. Gushurst, D. L. Grier, B. A. Leland, and J. Laufer, “Description of several chemical structure file formats used by computer programs developed at molecular design limited,” *Journal of chemical information and computer sciences*, vol. 32, no. 3, pp. 244–255, 1992.

[124] L. Ahmed, V. Georgiev, M. Capuccini, S. Toor, W. Schaal, E. Laure, and O. Spjuth, “Efficient iterative virtual screening with Apache Spark and conformal prediction,” *Journal of cheminformatics*, vol. 10, no. 1, p. 8, 2018.

[125] “Easy HPC clusters on GCP with Slurm.” `https://cloud.google.com/blog/products/gcp/easy-hpc-clusters-on-gcp-with-slurm`. Accessed: 2019-07-15.

[126] “Azure–SLURM cluster.” `https://azure.microsoft.com/en-us/resources/templates/slurm/`. Accessed: 2019-07-15.

[127] “Deploying a Burstable and Event-driven HPC Cluster on AWS Using SLURM.” `https://aws.amazon.com/blogs/compute/deploying-a-burstable-and-event-driven-hpc-cluster-on-aws-using-slurm-part-1/`. Accessed: 2019-07-15.

[128] C. L. Williams, J. C. Sica, R. T. Killen, and U. G. Balis, “The growing need for microservices in bioinformatics,” *Journal of Pathology Informatics*, vol. 7, 2016.

[129] S. Lampa, J. Alvarsson, and O. Spjuth, “Towards agile large-scale predictive modelling in drug discovery with flow-based programming design principles,” *Journal of cheminformatics*, vol. 8, no. 1, p. 67, 2016.

[130] “Amazon elastic file system.” `https://aws.amazon.com/efs/`. Accessed: 2019-07-15.

[131] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, “Object storage: The future building block for storage systems,” in *2005 IEEE International Symposium on Mass Storage Systems and Technology*, pp. 119–123, IEEE, 2005.

[132] “CSC cpouta.” `https://research.csc.fi/cpouta`. Accessed: 2019-07-15.

[133] “Ubuntu Docker Container.” `https://hub.docker.com/_/ubuntu`. Accessed: 2019-07-15.

[134] J. Arnold, *Openstack swift: Using, administering, and developing for swift object storage*. "O’Reilly Media, Inc.", 2014.

[135] “Embassy cloud.” `https://www.embassycloud.org`. Accessed: 2019-07-15.

[136] L. Candela, D. Castelli, and P. Pagano, “Virtual research environments: an overview and a research agenda,” *Data Science Journal*, pp. GRDI–013, 2013.

[137] D. D’Agostino, L. Roverelli, G. Zereik, A. De Luca, R. Salvaterra, A. Belfiore, G. Lisini, G. Novara, and A. Tiengo, “A microservice-based portal for x-ray transient and variable sources.,” *PeerJ Preprints*, vol. 4, p. e2519, 2016.

[138] C. Wu, R. Tobar, K. Vinsen, A. Wicenec, D. Pallot, B. Lao, R. Wang, T. An, M. Boulton, I. Cooper, *et al.*, “Daliuge: A graph execution framework for harnessing the astronomical data deluge,” *Astronomy and computing*, vol. 20, pp. 1–15, 2017.

[139] "Amazon Web Services." https://aws.amazon.com/. Accessed: 2019-07-15.
[140] "Google Cloud Platform." https://cloud.google.com/. Accessed: 2019-07-15.
[141] "Microsoft Azure." https://azure.microsoft.com. Accessed: 2019-07-15.
[142] A.-M. K. Pathan and R. Buyya, "A taxonomy and survey of content delivery networks," *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Technical Report*, vol. 4, 2007.
[143] "Mantl." http://mantl.io. Accessed: 2019-07-15.
[144] "Kubespray." https://github.com/kubernetes-incubator/kubespray. Accessed: 2019-07-15.
[145] "cloud-init." https://cloud-init.io. Accessed: 2019-07-15.
[146] "KubeNow GitHub organization." https://github.com/kubenow. Accessed: 2019-07-15.
[147] F. P. Miller, A. F. Vandome, and J. McBrewster, "Apache Maven," 2010.
[148] H. Mushtaq and Z. Al-Ars, "Cluster-based Apache Spark implementation of the GATK DNA analysis pipeline," in *2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 1471–1477, IEEE, 2015.
[149] "Helix Nebula Science Cloud." https://www.helix-nebula.eu/. Accessed: 2019-07-15.
[150] S. Toor, M. Lindberg, I. Falman, A. Vallin, O. Mohill, P. Freyhult, L. Nilsson, M. Agback, L. Viklund, H. Zazzik, *et al.*, "SNIC science cloud (SSC): A national-scale cloud infrastructure for swedish academia," in *2017 IEEE 13th International Conference on e-Science (e-Science)*, pp. 219–227, IEEE, 2017.
[151] "de.NBI Cloud." https://www.denbi.de/cloud. Accessed: 2019-07-15.
[152] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, pp. 1–20, Springer, 2017.
[153] "Google Cloud Run." https://cloud.google.com/blog/products/serverless/cloud-run-bringing-serverless-to-containers. Accessed: 2019-07-15.
[154] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *Acm Sigplan Notices*, vol. 48, no. 4, pp. 461–472, 2013.
[155] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv—optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 61–72, USENIX Association, June 2014.
[156] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*, pp. 250–257, IEEE, 2015.
[157] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo, "Ebbrt: a framework for building per-application library operating systems," in *12th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 671–688, 2016.

[158] “Kata Containers.” `https://katacontainers.io`. Accessed: 2019-07-15.

[159] G. M. Kurtzer, V. Sochat, and M. W. Bauer, “Singularity: Scientific containers for mobility of compute,” *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[160] “Marco Capuccini’s GitHub repository.” `https://github.com/mcapuccini`. Accessed: 2019-07-15.

// Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1846

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2019

Distribution: publications.uu.se
urn:nbn:se:uu:diva-390666