UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Elastic, interoperable and container-based cloud infrastructures for High Performance Computing

May 2021

Author: Sergio López Huguet
Advisor: Dr. Ignacio Blanquer Espert

"Lo que importa no es aquello que miras, sino lo que ves"

Henry David Thoreau

Para ti, Rocío,

porque juntos somos capaces de cualquier cosa.

Abstract

Scientific applications generally imply a variable and an unpredictable computational workload that institutions must address by dynamically adjusting the allocation of resources to their different computational needs. Scientific applications could require a high capacity, e.g. the concurrent usage of computational resources for processing several independent jobs (High Throughput Computing or HTC) or a high capability by means of using high-performance resources for solving complex problems (High Performance Computing or HPC). The computational resources required in this type of applications usually have a very high cost that may exceed the availability of the institution's resources or they are may not be successfully adapted to the scientific applications, especially in the case of infrastructures prepared for the execution of HPC applications. Indeed, it is possible that the different parts that compose an application require different type of computational resources. Nowadays, cloud service platforms have become an efficient solution to meet the need of HTC applications as they provide a wide range of computing resources accessible on demand. For this reason, the number of hybrid computational infrastructures has increased during the last years. The hybrid computation infrastructures are the combination of infrastructures hosted in cloud platforms and the computation resources hosted in the institutions, which are named on-premise infrastructures. As scientific applications can be processed on different infrastructures, the application delivery has become a key issue. Nowadays, containers are probably the most popular technology for application delivery as they ease reproducibility, traceability, versioning, isolation, and portability. The main objective of this thesis is to provide an architecture and a set of services to build up hybrid processing infrastructures that fit the need of different workloads. Hence, the thesis considered aspects such as elasticity and federation. The use of vertical and horizontal elasticity by developing a proof of concept to provide vertical elasticity on top of an elastic cloud architecture for data analytics. Afterwards, an elastic cloud architecture comprising heterogeneous computational resources has been implemented for medical imaging processing using multiple processing queues for jobs with different requirements. The development of this architecture has been framed in a collaboration with a company called QUIBIM. In the last part of the thesis, the previous work has been evolved to design and implement an elastic, multi-site and multi-tenant cloud architecture for medical image processing has been designed in the framework of a European project PRIMAGE. This architecture uses a storage integrating external services for the authentication and authorization based on OpenID Connect (OIDC). The tool *kube-authorizer* has been developed to provide access control to the resources of the processing infrastructure in an automatic way from the information obtained in the authentication process, by creating policies and roles. Finally, another tool, *hpc-connector*, has been developed to enable the integration of HPC processing infrastructures into cloud infrastructures without requiring modifications in both infrastructures, cloud and HPC. It should be noted that, during the realization of this thesis, different contributions to open source container and job management technologies have been performed by developing open source tools and components and configuration recipes for the automated configuration of the different architectures designed from the DevOps perspective. The results obtained support the feasibility of the vertical elasticity combined with the horizontal elasticity to implement QoS policies based on a deadline, as well as the feasibility of the federated authentication model to combine public and on-premise clouds.

Resumen

Las aplicaciones científicas implican generalmente una carga computacional variable y no predecible a la que las instituciones deben hacer frente variando dinámicamente la asignación de recursos en función de las distintas necesidades computacionales. Las aplicaciones científicas pueden necesitar grandes requisitos. Por ejemplo, una gran cantidad de recursos computacionales para el procesado de numerosos trabajos independientes (High Throughput Computing o HTC) o recursos de alto rendimiento para la resolución de un problema individual (High Performance Computing o HPC). Los recursos computacionales necesarios en este tipo de aplicaciones suelen acarrear un coste muy alto que puede exceder la disponibilidad de los recursos de la institución o estos pueden no adaptarse correctamente a las necesidades de las aplicaciones científicas, especialmente en el caso de infraestructuras preparadas para la ejecución de aplicaciones de HPC. De hecho, es posible que las diferentes partes de una aplicación necesiten distintos tipo de recursos computacionales. Actualmente las plataformas de servicios en la nube se han convertido en una solución eficiente para satisfacer la demanda de las aplicaciones HTC, ya que proporcionan un abanico de recursos computacionales accesibles bajo demanda. Por esta razón, se ha producido un incremento en la cantidad de clouds híbridos, los cuales son una combinación de infraestructuras alojadas en servicios en la nube y en las propias instituciones (on-premise). Dado que las aplicaciones pueden ser procesadas en distintas infraestructuras, actualmente la portabilidad de las aplicaciones se ha convertido en un aspecto clave. Probablemente, las tecnologías de contenedores son la tecnología más popular para la entrega de aplicaciones gracias a que permiten reproducibilidad, trazabilidad, versionado, aislamiento y portabilidad. El objetivo de la tesis es proporcionar una arquitectura y una serie de servicios para proveer infraestructuras elásticas híbridas de procesamiento que puedan dar respuesta a las diferentes cargas de trabajo. Para ello, se ha considerado la utilización de elasticidad vertical y horizontal desarrollando una prueba de concepto para proporcionar elasticidad vertical y se ha diseñado una arquitectura cloud elástica de procesamiento de Análisis de Datos. Después, se ha trabajo en una arquitectura cloud de recursos heterogéneos de procesamiento de imágenes médicas que proporciona distintas colas de procesamiento para trabajos con diferentes requisitos. Esta arquitectura ha estado enmarcada en una colaboración con la empresa QUIBIM. En la última parte de la tesis, se ha evolucionado esta arquitectura para diseñar e implementar un cloud elástico, multi-site y multi-tenant para el procesamiento de imágenes médicas en el marco del proyecto europeo PRIMAGE. Esta arquitectura utiliza un almacenamiento distribuido integrando servicios externos para la autenticación y la autorización basados en OpenID Connect (OIDC). Para ello, se ha desarrollado la herramienta *kube-authorizer* que, de manera automatizada y a partir de la información obtenida en el proceso de autenticación, proporciona el control de acceso a los recursos de la infraestructura de procesamiento mediante la creación de las políticas y roles. Finalmente, se ha desarrollado otra herramienta, *hpc-connector*, que permite la integración de infraestructuras de procesamiento HPC en infraestructuras cloud sin necesitar realizar cambios en la infraestructura HPC ni en la arquitectura cloud. Cabe destacar que, durante la realización de esta tesis, se han utilizado distintas tecnologías de gestión de trabajos y de contenedores de código abierto, se han desarrollado herramientas y componentes de código abierto y se han implementado recetas para la configuración automatizada de las distintas arquitecturas diseñadas desde la perspectiva DevOps. Los resultados obtenidos avalan la idoneidad de la elasticidad vertical combinada junto con la elasticidad horizontal para implementar políticas de Calidad de Servicio basadas en plazos de ejecución, así como la viabilidad del modelo de autenticación federada para combinar clouds públicos y on-premise.

Resum

Les aplicacions científiques impliquen generalment una càrrega computacional variable i no predictible a què les institucions han de fer front variant dinàmicament l'assignació de recursos en funció de les diferents necessitats computacionals. Les aplicacions científiques poden necessitar grans requisits. Per exemple, una gran quantitat de recursos computacionals per al processament de nombrosos treballs independents (High Throughput Computing o HTC) o recursos d'alt rendiment per a la resolució d'un problema individual (High Performance Computing o HPC). Els recursos computacionals necessaris en aquest tipus d'aplicacions solen comportar un cost molt elevat que pot excedir la disponibilitat dels recursos de la institució o aquests poden no adaptar-se correctament a les necessitats de les aplicacions científiques, especialment en el cas d'infraestructures preparades per a l'avaluació d'aplicacions d'HPC. De fet, és possible que les diferents parts d'una aplicació necessiten diferents tipus de recursos computacionals. Actualment les plataformes de servicis al núvol han esdevingut una solució eficient per satisfer la demanda de les aplicacions HTC, ja que proporcionen un ventall de recursos computacionals accessibles a demanda. Per aquest motiu, s'ha produït un increment de la quantitat de clouds híbrids, els quals són una combinació d'infraestructures allotjades a servicis en el núvol i a les mateixes institucions (on-premise). Donat que les aplicacions poden ser processades en diferents infraestructures, actualment la portabilitat de les aplicacions s'ha convertit en un aspecte clau. Probablement, les tecnologies de contenidors són la tecnologia més popular per a l'entrega d'aplicacions gràcies al fet que permeten reproductibilitat, traçabilitat, versionat, aïllament i portabilitat.

L'objectiu de la tesi és proporcionar una arquitectura i una sèrie de servicis per proveir infraestructures elàstiques híbrides de processament que puguen donar resposta a les diferents càrregues de treball. Per a això, s'ha considerat la utilització d'elasticitat vertical i horitzontal desenvolupant una prova de concepte per proporcionar elasticitat vertical i s'ha dissenyat una arquitectura cloud elàstica de processament d'Anàlisi de Dades. Després, s'ha treballat en una arquitectura cloud de recursos heterogenis de processament d'imatges mèdiques que proporciona distintes cues de processament per a treballs amb diferents requisits. Aquesta arquitectura ha estat emmarcada en una colÂ·laboració amb l'empresa QUIBIM. En l'última part de la tesi, s'ha evolucionat aquesta arquitectura per dissenyar i implementar un cloud elàstic, multi-site i multi-tenant per al processament d'imatges mèdiques en el marc del projecte europeu PRIMAGE. Aquesta arquitectura utilitza un emmagatzemament integrant servicis externs per a l'autenticació i autorització basats en OpenID Connect (OIDC). Per a això, s'ha desenvolupat la ferramenta *kube-authorizer* que, de manera automatitzada i a partir de la informació obtinguda en el procés d'autenticació, proporciona el control d'accés als recursos de la infraestructura de processament mitjançant la creació de les polítiques i rols. Finalment, s'ha desenvolupat una altra ferramenta, *hpc-connector*, que permet la integració d'infraestructures de processament HPC en infraestructures cloud sense necessitat de realitzar canvis en la infraestructura HPC ni en l'arquitectura cloud. Es pot destacar que, durant la realització d'aquesta tesi, s'han utilitzat diferents tecnologies de gestió de treballs i de contenidors de codi obert, s'han desenvolupat ferramentes i components de codi obert, i s'han implementat receptes per a la configuració automatitzada de les distintes arquitectures dissenyades des de la perspectiva DevOps.

Els resultats obtinguts avalen la idoneïtat de l'elasticitat vertical combinada junt amb l'elasticitat horitzontal per implementar polítiques de Qualitat de Servei basades en terminis d'execució, així com la viabilitat del model d'autenticació federada per combinar clouds públics i on-premise.

Contents

List of Figures

Acknowledgements

Esta tesis es el resultado del esfuerzo que hemos estado haciendo durante los últimos años. Digo “hemos” porque no ha sido un proceso que haya hecho solo y sin ayuda, al contrario, ha sido posible gracias a todos vosotros.

En primer lugar, quiero agradecer a mis compañeros del grupo de investigación GRyCAP: Miguel Caballer, Amanda Calatrava, Germán Moltó, J. Damià Segrelles, Carlos de Alfonso, Andy Alic, Alfonso Pérez, José Miguel Alonso, Pau Lozano, David Arce, Conrado J. Calvo, Diana Naranjo, Vicent Giménez y Sebas Risco. Durante estos años, siempre habéis estado ahí para ayudarme en todo lo posible, apoyarme y animarme. En los peores momentos de la tesis, siempre habéis estado ahí para decirme “tranquilo, es normal, no te agobies”. Nos ha tocado vivir esta época dónde no se nos permite juntarnos como de costumbre, pero, cada día queda menos para poder volver a disfrutar de esos momentos en los que siempre habéis conseguido que me sienta como en casa. ¡Muchas gracias a todos!

En 2016 tuve la suerte de tenerte como profesor, Nacho. Unos meses después, un poco como carambola, comencé mi andadura en el grupo. Desde ese mismo momento, siempre has estado ahí, entendiéndome, apoyándome, enseñándome este mundo tan desconocido en el que trabajamos y ayudándome a conseguir todas las cosas que me he propuesto, siempre cuidando tanto de mí como del resto del grupo). Nacho, gracias por la paciencia (en especial durante este último año), por tu mano izquierda, por tu optimismo, por conseguir que exista este ambiente en el grupo, por la oportunidad de trabajar contigo, por cuidarme en todos los aspectos, pero, sobre todo, por ser cómo eres, un referente para mí. Ahora, en 2021, podría decir que he tenido la suerte como director de tesis, pero, no es del todo cierto, he tenido la suerte de encontrar a un amigo que siempre va a ocupar un lugar especial en mi corazón.

Durante esta etapa he tenido la suerte de poder colaborar con muchas personas, a las cuales me gustaría dedicar unas palabras.

Por un lado, quiero agradecer a la gente con la que he colaborado en QUIBIM. En especial, quiero destacar a Ángel Alberich, Fabio García, Ismael González, Alejandro Mañas y Rafa Hernández. Siempre habéis tenido la puerta abierta de vuestras oficinas y me habéis ayudado en todo, ¡muchas gracias!

I also want to thank the people from Onedata, ACC Cyfronet, and the AGH University of Science and Technology in Krakow. Marian Bubak, Marek Kasztelnik, and Jan Meizner: thanks for your support and assistance during my research stay. Łukasz Opioła and the rest of the Onedata team: thanks for the time we spent together, the pizza days and all the coffee breaks, you were a great support during my time in Poland.

También en el ámbito profesional, pero, esta vez, durante mi etapa como docente, quiero agradecer a todos los profesores con los que he tenido el placer de compartir asignatura. Muchas gracias, María José Vicent, Juan Sánchez por toda la ayuda prestada. En especial, quiero dar las gracias a Sole Valero y Estefanía Argente, por tener siempre la puerta (o el Teams) abierto para hablar de todo, por preocuparos por mí y por ser unas grandes profesoras.

Esta tesis ha sido posible gracias a muchas otras personas que no pertenecen al ámbito laboral, aquellas que han estado siempre y aquellas que han ido entrando en mi vida.

Gracias a todos los amigos de Nules, aquellos que he conocido gracias a Rocío. David, Ángel, Raquel, Judith, Mario, Rubén, Pau, Katia: habéis hecho que sienta como si os conociera desde siempre y, aunque durante esta etapa no hemos podido juntarnos tanto como deseáramos, siempre habéis estado animándonos con la tesis.

Aunque te conocí en el grupo, Amanda, tanto tú como David habéis entrado de lleno en nuestras vidas. Todo comenzó en el momento en que te pregunté, Amanda, por juegos de mesa en uno de esos descansos que tanto se echan de menos. Desde ese momento, siempre habéis estado ahí apoyándonos, tanto a Rocío como a mí, con la tesis y con el resto de los problemas. Muchas gracias por los consejos, las noches "pescando" en Animal Crossing, por los días de juegos de mesa y por las aventuras de Dungeons & Dragons pero, sobre todo, gracias por estar ahí.

A mis amigos, habéis estado a mi lado desde siempre. Gracias por estar ahí cuando lo he necesitado, gracias por entender que durante esta etapa he tenido que estar más desconectado y gracias por los ánimos que me habéis mandado. Doñate, Miriam, Pedro, Pelino, Toni y Miriam, se echa de menos las cenitas y ponernos al día. Durante esta etapa también ha aumentado la familia: Álvaro y Marc. Aunque ellos no hablen, han hecho feliz a su "tío" solo con mirarlo y tengo muchas ganas de pasar más tiempo con ellos. Nunca se me olvida aquello que siempre hemos dicho "da igual cuanto tiempo pase, siempre es como si nos hubiéramos visto ayer, pero, con más cosas que contarnos". Pronto, cuando podamos volvernos a juntarnos con normalidad, nos cobraremos todo el tiempo que nos debemos.

Alba y Adri, entre la tesis y la pandemia, tampoco hemos podido pasar todos los momentos que desearíamos, especialmente, durante este último año. Pronto volverán las cenitas los 4, los vinitos, los Zombicide o Massive (aunque sea con la cámara cenital) y nuestros viajes. Gracias por el "power", por apoyarnos en los peores momentos, pero, sobre todo, por todos los ratitos juntos.

Este camino no lo podría haber recorrido sin el apoyo de la familia, la de siempre y la política.

A Koba y Merlín, gracias por haber entrado tan “fácilmente” en nuestras vidas y por todo el cariño y la paz que aportáis. Estoy seguro de que vamos a pasar grandes momentos durante los próximos años y espero que consigamos haceros muy felices. A Peluseta, gracias por los momentos de sofá, de persecución de pelota con la lengua fuera, por la paz que nos has dado, por tus movimientos de cola “efusivos”, por trasnochar junto a nosotros mientras trabajábamos, y por el resto de los momentos que nos has permitido disfrutar junto a ti. Fuiste un ejemplo de lucha hasta el final, demostrando lo grande que se puede ser siendo tan pequeñita. Espero haber podido hacerte feliz desde que empecé a formar parte en tu vida.

A Javier y Pili, habéis sido un gran apoyo durante esta tesis tanto para mí como para Rocío. Desde el primer momento en el que os conocí, habéis conseguido hacerme sentir como en casa. Gracias por los consejos, por las sorpresas, por comprenderme aún sin estar de acuerdo, por las reflexiones, por vuestra experiencia, por la tranquilidad que aportáis, por escucharnos, por animarnos y por haber convertido al amor de mi vida en lo que es. Perdonad por los malos momentos, los ratos de quejas y los nervios que os hacemos pasar. En definitiva, muchas gracias por todo, habéis conseguido que sienta que tengo otro padre y otra madre, espero poder devolver todo lo feliz que me hacéis ser.

A Lucía, gracias por siempre apostar por mí y por tus ánimos. En muchos momentos no estamos de acuerdo o discutimos, pero, lo más importante, es que siempre lo arreglamos. Durante esta tesis hemos tenido que separarnos para que fueras a Albacete a trabajar, pero eso no ha conseguido que me hayas ayudado cuando lo he necesitado. Carlos, gracias por cuidar de mi hermana en todo este periodo y aportarle la paz que le hace falta. Perdonad porque no he podido ir todavía a visitaros y porque muchas veces soy un despistado. Cuando todo esto baje, os prometo que iremos a visitaros y a recuperar todo el tiempo que no hemos podido pasar juntos durante esta etapa y la pandemia.

A mis padres, gracias por luchar para conseguir que Lucía y yo seamos lo que somos ahora. Durante todos estos años me habéis enseñado a pelear por lo que quiero, a levantarme cuando fallo y que siempre hay que estar dispuestos a cambiar

para mejorar. Tenemos formas distintas de recorrer la vida, pero siempre hacemos por encontrarnos en el sendero. Perdonad por todos los nervios que os hacemos pasar, por no estar tanto tiempo como querríamos y por ser tan difícil en algunos momentos. Gracias por siempre tener un abrazo disponible, por preparar una "paelleta" y "la picaeta' que tanto disfrutamos juntos, por todos los esfuerzos que habéis hecho por nosotros y por permitirme desde muy joven, aunque a veces no estuvierais de acuerdo, hacer lo que yo mejor consideraba.

A ti, Rocío, gracias por hacerme sentir el hombre más afortunado del mundo. Sin tu apoyo, esto no hubiera sido posible. Gracias por creer en mí aún cuando ni yo mismo lo hago, por quererme cuando menos lo merezco, por tu paciencia, por tus locuras, por escucharme, entenderme y mimarme. La escritura de la tesis al mismo momento ha sido en ciertos momentos complicada por muchos motivos, pero eso no ha impedido que sigamos "viviendo un sueño juntos". Ahora comienza una nueva etapa para ambos en la que debemos estar separados, pero, como siempre te digo, "lo haremos lo mejor posible". No tengo ninguna duda que seremos felices descubriendo y disfrutando Valladolid tanto como lo hemos hecho con Barcelona y Burdeos. Perdona por todas las veces que no he estado a la altura y gracias por luchar para seguir siendo nosotros, te quiero.

Chapter 1

Introduction and objectives

Scientific applications generally imply a variable and unpredictable computational workload that researchers must estimate when requesting resources to Data Processing Centres, with little or no capability of dynamically adjusting the allocation of resources. Therefore, overestimation is habitual, so institutions end up with an enormous request of computational resources. Moreover, scientific applications typically comprise parts that require processing many independent jobs (High Throughput Computing or HTC) or high-performance resources for solving complex problems (High Performance Computing or HPC), which could fit different types of resources.

In a flexible scenario, researchers could request computing resources that fit the common workloads, but when conditions change, the infrastructure must be updated and resized. The acquisition of equipment that only is necessary to face up punctual workload peaks or when different workloads appear could be able for big institutions or large consortia. For this reason, it is very interesting to take profit of the cloud platforms to address this problem. Cloud service platforms provide a wide range of computing resources and services that have made them an efficient solution to get the resources needed on demand.

Elasticity is a key issue in cloud computing because it allows to dynamically resize the infrastructure depending on certain aspects, for example, to guarantee a minimum level of Quality of Service (QoS) or to reduce the waste of unused resources. There are two types of elasticity depending on the problem: horizontal and vertical. The horizontal elasticity is used when the problem can leverage concurrency (for example, in the HTC case) to be completed earlier. In some cases, applications can increase their performance by adjusting the resources allocated to individual instances, such as memory and number of cores. Vertical elasticity focuses on increasing or decreasing the allocation of such resources to the running instances. Occasionally, vertical elasticity could complement horizontal elasticity.

The advances in the study of elasticity and the increase in the number of cloud computing platforms offers have contributed to the arise of hybrid infrastructures, which combine computing resources hosted by different providers (such as public cloud platforms and on-premise institutional resources).

The management of hybrid infrastructures is a complex task because there are many aspects to consider: the synchronization of several processing infrastructures; distributed storage; coherent authentication and authorization; application delivery, etc. Configuring and operating such infrastructures is complex and requires nontrivial system administration skills.

Nowadays, the growth of hybrid and cloud infrastructures has contributed to consider application delivery also a key issue. Container technologies have become the most popular solution for application delivery in cloud environments. Regarding HPC infrastructures, popular container technologies require administrative privileges that are incompatible with the access restrictions that HPC centres impose to their users. However, they are starting to use them thanks to the improvements of user-space container technology. For example, Singularity containers [94] are currently used [152] in the world's fastest supercomputer (RIKEN's Fugaku [153] in Tokio, Japan) according to the November 2020 TOP500 list [172].

One of the aims of this thesis is to provide elastic cloud architectures that address different workloads (for example, HPC or HTC). Other objective of this thesis is to take profit of the DevOps perspective to contribute with configuration recipes for the implementation of hybrid infrastructures, as well as the development of services and tools for the integration of on-premises and cloud infrastructures. These contributions could facilitate the adoption of cloud infrastructures in commercial or academia environments for leveraging the benefits of cloud platforms. This dissertation is composed of different articles that have been published or have been submitted to different conferences and journals.

1.1 Objectives

The main objective of this thesis is to design, implement and validate an architecture and a set of services to build up hybrid processing infrastructures that can address different workloads. This general objective can be decomposed into several (sub-)objectives that will be tackled in this thesis:

- **Analyse the requirements of scientific applications dealing with medical image processing.** Institutions have different computing necessities depending of their scientific applications. Hence, there are some aspects that must be considered to provide cloud infrastructures that cover all of their necessities. This thesis will study the typical computing requirements of trend research fields, such as data analysis or medical imaging processing, to design cloud architectures that efficiently address their necessities.

- **Design and implement a mixed vertical and horizontal elasticity framework.** The cloud architectures designed in this thesis will make use of automatic horizontal elasticity to optimize the resource allocation in cloud platforms and to address workload peaks or different types of workloads

(HTC or HPC). The implementation of these cloud infrastructures must be performed following the DevOps perspective to provide automatic management of computing resources. Furthermore, the vertical elasticity will be studied to guarantee a minimum of QoS defined as a time execution deadline.

- **Support hybrid infrastructures by developing services for federated authentication and interconnection between different processing infrastructures.** Hybrid architectures enable fitting different workload types by the combination of cloud and on-premise infrastructures. There are some aspects that must be considered and will be studied in this thesis. Authorization is a key aspect and, in this dissertation, a service to provide federated authorization must be implemented. The management of different job queues or multiple workload manager also are important aspects that will be studied during this thesis.

- **Identify the most appropriate container technology and orchestrator tool according to the architecture requirements.** Containers have become the most popular technology for application delivery. During the research phase, different container technologies and container orchestrator solutions will be analysed to identify which are the most appropriate depending on their architecture necessities. Although container technologies commonly have similar features, they work with containers with different strategies or security requirements, so one of them can fit better than the rest for certain scenarios.

- **Develop use cases to validate the architectures and the services implemented**. Despite the architectures developed are for general purpose, a set of use cases will be implemented in the context of medical imaging and data analytics to validate their concept and development. The first case is focused on validating a cloud architecture for data analytics that address different types of workloads (long-running jobs and batch jobs) leveraging the horizontal and vertical elasticity. The second case is the design of an elastic cloud architecture that focuses on demonstrating the integration of different tools and workloads, focusing on medical imaging analysis processing for a business company named QUIBIM. The third case focuses on demonstrating that the cloud architecture designed and software developed provide a hybrid infrastructure, multi-tenant and multi-site in the framework of the European project PRIMAGE.

Open Science improves the effectiveness and productivity of the whole research community and avoids “reinventing the wheel”. Thus, the whole set of services, tools and configuration recipes that will be developed during this thesis must be open source.

1.2 Summary of the state of art

This section summarizes the state of the art in the framework of this dissertation. This dissertation is structured as a compilation of research papers that have been published or submitted to indexed journals and conferences. Therefore, each chapter includes a detailed section regarding the state of the art in the particular problem addressed by the chapter.

The use of cloud infrastructures by institutions and enterprises is increasingly widespread [14]: renowned research centres such as CERN [174] use private cloud platforms; European initiatives and projects provide hybrid cloud platforms such as Helix Nebula Science Cloud [53] or Gaia X [63]; and multinational companies as Philips [44] or Spanish companies such as Mahou San Miguel [75] are using cloud platforms.

Cloud computing has proven to be effective to tackle e-Science challenges [182][181] due to the fact that cloud platforms enable the ability to adapt the computing infrastructure to the application. For example, the analysis of large amounts of data can be processed to obtain very useful information. There have been many programming models for big data that leverage cloud computing [180]. The MapReduce programming model has provided an efficient way to process large amounts of data on cloud service platforms due to the access to large clusters on demand. The computational paradigm MapReduce is implemented by Apache Hadoop [59], a framework to process applications on large clusters with the data stored in its own distributed file system named Hadoop Distributed File System. Another important programming model is functional programming. In this type of programming model, the most popular framework is Spark [62], which allows to process data storage in RAM memory. The use of Structured Query Language (SQL) is the classical way to interact with data warehouses. Apache Hive [60] is an open-source data warehousing solution built on top of Hadoop that supports queries expressed in a SQL-like declarative language that will be compiled into MapReduce jobs executed on Hadoop. The Actor model [72] is a programming model for concurrent computation, which considers "Actor" as the universal primitive unit for computation. An actor is responsible for reacting to a set of messages to trigger specific processing logics for different contexts. Apache Storm [173] is an example of this programming model. The dataflow programming paradigm [87] is designed to model programs as directed graphs with operations and dependencies as nodes and edges. An example of this programming model is Microsoft Dryad [82].

Another field that has relied on cloud services is biomedicine. In [78] [115] the authors describe a large number of bioinformatic applications as well as scientific workflow tools that support bioinformatics applications leveraging cloud services. On the other hand, there are works that use other preconfigured platforms with a large number of tools for bioinformatic analysis. For example, a web platform named Galaxy Project [64] that can be deployed on public and on-premises Cloud offerings.

Regarding the deployment and configuration of cloud computing clusters, there are several works that provide tools to configure a Load Resource Management System (LRMS) and the associated resources in cloud service providers. On the one hand, StarCluster [169] (from the Massachusetts Institute of Technology) is an open-source cluster-computing toolkit for Amazon's Elastic Compute Cloud that has a plugin [170] to manage the elasticity according to the length of the cluster's job queue. On the other hand, Elastic Cloud Computing Cluster (EC3) [119], a tool that manages elastic virtual clusters from computational resources in cloud platforms that was developed in the research group where this thesis is performed.

It should be pointed out that EC3 is also able to manage the horizontal elasticity according to customisable policies thanks to a previous work: CLUster Energy Saving (CLUES) [36]. EC3 is able to deploy and destroy VMs according to the workload of the cluster. Elasticity is a research field widely studied in cloud computing. For example, in [23] [122] the authors present a method to manage the elasticity in a HTC infrastructure. In [123], the authors design and study a generic and elastic architecture for hybrid infrastructures. The authors in [90] propose a framework to manage the elasticity according to the resource usage (CPU, memory, etc.). In[161] the authors present Kingfisher, a system to manage the elasticity according to the pricing models and transition strategies that optimize the incurred cost.

The creation of scientific applications has different stages [154] and institutions may have different computing environments for their scientific applications. Furthermore, it is possible that scientific applications could be used in different computing resources. For these reasons, it is crucial to consider the application delivery aspect. Containers are not a new concept (they come from the idea of FreeBSD jails [42] in 2000) but they have become the most popular technology for application delivery during the last years thanks to the reproducibility, isolation, provenance and portability. A container is an executable unit of software in which an application code is packaged, along with its libraries and dependencies. Container technology encapsulates processes and their execution environment in isolated filesystems and namespaces that enable the execution of processes in a restricted environment. As they are actually system processes, there is no overhead like that caused by the hypervisor layer introduced by virtual machines in CPU, memory, and storage [52] because containers do not virtualise the entire OS. Going one layer deeper, container concept can be split in various concepts: container image format, container engine, container runtime and container orchestrator. Historically, there were different Container Image formats: Docker [147], Appc [9] and LXC/LXD [105] [106]. Nowadays, with the formation of the Open Container Initiative (OCI) [128], almost all container engines and runtimes are compliant with the specification defined by the OCI [129]. Furthermore, container images can be made up of one or more layers. A container engine can be defined as “a piece of software that accepts user requests, including command line options, pulls images, and from the end user’s perspective runs the container” [150]. Almost all container engines are OCI compliant (Docker and RKT [33]) but LXC/D has its own format. Regarding the container runtime, OCI also defines a Runtime specification [130]. The reference implementation of this standard is runC [81] but there exist other OCI compliant runtimes: crun [32], RailCar [142], Katacontainers [57] and CRI-O[56]. The container orchestrators are in charge of managing the execution of containers along the different computing nodes. Commonly, the container execution implies several system calls and functions that could require privileged permissions. All container technologies employ one of the following security approaches: only allow executing containers to users that belong to the sudoers group, have a privileged root daemon that runs the containers, use SetUID [158] to enable privilege escalation, use Linux capabilities [98], or employ user namespaces [175].

Linux Containers (LXC or LXD) take profit of kernel features (kernel namespaces [124], chroots [26], cgroups [20], capabilities [98], etc.) to create an environment as near as possible to a standard Linux but without the need for a separate kernel.

Docker has reached the maximum popularity due to its convenient and rich ecosystem of tools and container orchestrators. Docker manages containers using *containerd* [31], which relies on runC library the container execution. Docker and runC commonly depend on privileged daemons to manage containers, although both can run in user namespaces (Docker rootless mode is an experimental feature [39]). Docker images are formed by several layers that typically are stored in a Container Registry, which is a public or private image repository.

The launch of Docker in 2013 placed containers as the most popular technology for application delivery and, therefore, other alternatives emerged during the recent years trying to facilitate the acquisition of containers in other environments such as HPC infrastructures: Singularity [94], Shifter [125], CharlieCloud [146] [95], udocker [67] or Podman [151]. Most of them inherit the Docker image ecosystem for building and storing the images and even permit to use pure Docker images or convert them to their specific formats. Singularity is the most popular container technology for HPC environments. Singularity is designed to allow users to run containers as themselves. For this reason, there are some calls that require privileges and, by default, Singularity employs a SetUID binary to perform these tasks, although it supports Linux capabilities and user namespaces [163]. Shifter and CharlieCloud are other alternatives designed for HPC environments developed by the National Energy Research Scientific Computing Centre (NERSC) and Los Alamos National Laboratory (LANL), respectively. On the other hand, Podman and uDocker are user-space and daemonless container technologies for generic use. uDocker is designed for executing Docker containers in user space and it provides three alternative modes to run containers: user namespaces (using runC), *PTRACE* execution method (using PRoot [149]) and *LD_PRELOAD* method (using fakechroot [50]). Podman manages OCI compliant containers and pods (groups of containers) but it only supports running containers on Linux. On the other hand, if an institution employs Docker, the authors state that the adoption of podman is easy because the only requirement is to create the “alias” command `alias docker=podman`.

There are some excellent container orchestration tools and job schedulers that are able to manage the different container technologies. Docker Swarm [41] was the native job scheduler of Docker. It has designed to be used in small and medium clusters. When this thesis started, Docker Swarm was more used that Kubernetes but, nowadays, it is being replaced by Kubernetes (even Docker Engine has included a standalone version of Kubernetes). The container runtime used in Docker Swarm is *containerd*. Kubernetes [171] (also known as K8s) is an open-source orchestration system for Docker containers. *Pods* are the minimum unit of scheduling in Kubernetes and they are groups of containers that are deployed and scheduled together. Kubernetes has a big community that enhances the tool with different plug-ins and modules, such as Helm [71] (package manager to configure applications or services) or Kubernetes Ingress Controller [92] (it exposes HTTP and HTTPS routes from outside the cluster to services within the cluster). It should be pointed out that Kubernetes has a module [93] that is able to manage the horizontal elasticity of *pods* based on the CPU usage. Kubernetes also used *containerd* as a container runtime, but Kubernetes recently announced that it will be using CRI-O by default. Apache Mesos [117] is not a job scheduler, it is a large-scale resource management system that partitions and assigns computing resources across various job schedulers, which are also called frameworks (such as Spark [62], Hadoop [59], etc.). It should be pointed out that Docker Swarm and Kubernetes also can run on top of a Mesos cluster. Nomad [126] is an open-source job scheduler that allows running non-containerised and container-based jobs with different container technologies (natively, Docker and Podman; and by using community-developed plugins, it can run Singularity, rkt and LXC). It can be remarked that Kubernetes can also run other container technologies but it is required to change the container runtime, and Apache Mesos relies on its frameworks to be able to run several container technologies.

1.3 Organization of this document

This dissertation is presented as a compendium of research articles which were published or submitted during the research phase. Chapters 2, 3, 4, and 5 correspond to works presented in already published articles in conferences and journals, and Chapter 6 is associated to a submitted article. As each chapter is a whole paper, all of them include the sections introduction and state of art. Thus, these chapters can be read independently and it is possible that there exists some redundant information in the different related work sections. The remaining chapters are a discussion of the results obtained and the conclusion of the thesis. The following paragraphs describe each chapter in detail.

Chapter 2, “Vertical Elasticity on Marathon and Chronos Mesos frameworks”, presents an open-source mechanism that provides a way to manage the vertical elasticity in Chronos and Marathon framework in an Apache Mesos cluster. This work demonstrates that, thanks to the checkpointing feature and the redeployment capabilities of the Mesos frameworks, it is possible to dynamically vary the resources assigned to an application according to its progress and considering a specific Quality of Service (QoS). This work [101] was published in the Journal of Parallel and Distributed Computing, which is a JCR-indexed journal that belongs to the second quartile (Q2).

Chapter 3, “A self-managed Mesos cluster for data analytics with QoS guarantees”, presents an elastic cloud architecture for data analytics with vertical elasticity. This work combines the previous work with the tools developed in the research group with a set of open-source recipes that allow to build a cloud architecture from scratch in the majority of cloud providers. This work [102] was published in the Future Generation Computer Systems journal, which is a JCR-indexed journal that belongs to the first quartile (Q1).

Chapter 4, “A Cloud Architecture for the Execution of Medical Imaging Biomarkers”, presents an elastic cloud architecture designed for the execution of medical imaging biomarkers. This architecture is composed of a heterogeneous computing infrastructure to deal with different types of jobs. Moreover, this architecture includes a workflow to ease the building, testing, delivery, and version management of the containers thanks to the adoption of a continuous integration tool. This work [100] was published in the conference International Conference on Computational Science, which is rated as CORE A and Class 2 in the GII-GRIN-SCIE conference ranking.

Chapter 5, “Seamlessly managing HPC workloads through Kubernetes”, introduces an open-source tool named *hpc-connector* to ease the combination of two existent computing infrastructures. The main idea behind this tool is to create an extra layer to permit the combination without modifying the computing infrastructure. Thus, it is possible to interconnect different computing infrastructures that have incompatible job schedulers or different authentication methods. In this work, the authors combine a Kubernetes cluster hosted in Spain with the supercomputer Prometheus [34] hosted in Poland, which uses the job scheduler Slurm. This work [103] was published in the conference ISC High Performance, which is rated as CORE C and has not defined a class yet in GII-GRIN-SCIE conference rating.

Chapter 6, “Automated isolation management of processing workflows in a multi-tenant and multi-site Kubernetes clusters - a medical imaging use case”, presents *kube-authorizer*, which is a service to automate the creation of namespaces, service accounts, and permissions of users authenticated by OpenID Connect (OIDC) in a Kubernetes cluster. This work has been submitted to the IEEE Access journal, which is a JCR-indexed journal that belongs to the first quartile (Q1).

Chapter 7 is a discussion of the achievements obtained during the research phase of this thesis. Finally, Chapter 8 summarizes the main results, concludes this dissertation and points to future work.

Chapter 2

Vertical Elasticity on Marathon and Chronos Mesos frameworks

Published as:
Sergio López-Huguet et al. "Vertical elasticity on Marathon and Chronos Mesos frameworks". In: Journal of Parallel and Distributed Computing *133 (2019), pp. 179 –192.* ISSN*: 0743-7315.* DOI*:* *https://doi.org/10.1016/j.jpdc.2019.01.002.* URL*:* *http://www.sciencedirect.com/science/article/pii/S0743731519300085*

2.1 Introduction

Apache Mesos [117] is a large-scale Resource Management System (RMS) that can partition and assign the resources of computing infrastructure (CPU, memory, I/O, disk and special resources) across several job schedulers, namely frameworks. Apache Mesos is not used to execute applications, but to allocate resources to those frameworks. Apache Mesos is becoming very popular due to a large number of supported frameworks. Among them, Marathon [109] and Chronos [25] provide means for deploying reliable services and periodic batch applications.

Applications deployed on Marathon can be scaled up and down according to specific triggers. However, this horizontal elasticity is appropriate when the problems that are solved are inherently parallel (e.g. stateless web services scaling on a variable access workload), but when the problem cannot benefit from an increase of the number of resources, another type of elasticity must be considered. Vertical elasticity consists of resizing dynamically the resources assigned to each application to meet varying workload demands or a previously Quality of Service (QoS) agreed. Some scenarios will benefit more from vertical elasticity rather than horizontal elasticity [155]. Studies propose proactive elasticity [104] as an effective mechanism to improve QoS.

This work uses the RMS Apache Mesos, whose objective is to execute applications by using distributed frameworks and controlling resources such as storage, CPU and memory in a collection of computational nodes. Depending on the framework features, it is possible to scale-out or scale-in worker nodes (horizontal elasticity) or adjust the CPU share (vertical elasticity) associated with each execution instance. There are some frameworks among Apache Mesos, but this work focuses on Marathon and Chronos. Marathon is a production-grade container orchestration platform with high availability and fault-tolerant framework. Furthermore, it is designed for managing long-running applications, which is a common type of application that executes on Cloud infrastructure. Besides, Marathon provides methods to perform horizontal and vertical elasticity but only for stateless applications. Chronos is a distributed and fault-tolerant scheduler designed as a replacement for Linux Cron [99]. Furthermore, it supports the execution of Docker containers and also provides a mechanism for varying their assigned resources. Both frameworks are complementary and widely used together. There are many applications in Cloud computing that run several iterations and, with this framework, it is possible to manage such class. More precisely, this work addresses two use cases, each one with one of the frameworks:

- Chronos can schedule a set of application iterations with a given periodicity and start them at a given time. It is not targeted at the bag-of-tasks execution model, in which applications are independent and embarrassingly parallel, which could be controlled by simpler horizontal elasticity mechanisms. Chronos can execute applications defined as a workflow, in which each stage requires the completion of a previous stage. We consider the simple case of an iterative workflow with a fixed number of iterations of one application. Therefore, vertical elasticity is essential to define the most appropriate number of resources to execute each iteration in the desired timeframe.

- Marathon can run a multithreaded application whose computational cost is known previously, so it is expected to finish once a certain amount of CPU time has been consumed. We cannot apply horizontal elasticity as it runs on a single node. If vertical elasticity is not utilised, then the resource allocation must ensure that the deadline is met. Given a scenario of nodes with 4 cores each, if, for example, an application requires 1.5 CPUs for a given time, and this reservation is static, the remaining 2.5 cores will be wasted if no matching request comes to the system. In our approach, the application will use the maximum CPU share (4 cores) if there are no competing applications in the same node. If a new request comes to the system asking for 2.5 cores, the system could allocate it in the same node, reducing the CPU share according to the 1.5 / 2.5 ratio. Therefore, the application could advance part of the computation, anticipating load to earlier time slots and leading to a better load balancing.

Mesos, Marathon and Chronos are only examples of the tools that appear to foster the usage of containers. Containers enable the developer to encapsulate the applications and their libraries winning more flexibility, scalability, boot up time and resource efficiency than when using Virtual Machines (VM) [12] [52]. Thus, the containers become the most popular way for packaging and deploying applications on Cloud infrastructures. Docker containers, in turn, have become the most popular container platform. For this reason, the development proposed in this work is based on Docker containers instead of Mesos native containers.

Thereby, the work is focused on designing and implementing a mechanism that, given an application specification and a QoS target, can deploy it in Mesos framework (Chronos or Marathon) using a Docker container, monitor such application and vary the assigned resources with the objective of achieving the agreed QoS.

This paper is organised as follows. Section 2.2 describes the problem in detail and then defines the two possible scenarios according to the above use cases and the frameworks. Section 2.3 describes the past works of elasticity in VM and Docker containers. Section 2.4 describes the technologies mentioned in this paper. Section 2.5 presents the mechanism developed. Section 2.6 describes the tests for evaluating this mechanism and then discusses the obtained results. Finally, Section 2.7 presents the conclusions of the work explained in the paper.

2.2 Description of the problem

The motivation of the work is to provide a mechanism for adjusting the allocation of vCPU resources to a running application inside a Docker container managed by a Mesos Framework (Marathon or Chronos). The design principles are the following:

- There is a known, feasible deadline for a given execution of an application or, if not, there is a Quality of Service target, expressed as the CPU time, that must be allocated to the application in a given timeframe.
- The application can benefit from multiple cores.
- Multiple applications are competing for the resources – otherwise, the QoS guarantees would not have any meaning.

- Applications are multithreaded but can only use one node (shared memory). The number of threads an application will use will depend on the number of vCPUs of the node, which will be the same for all the nodes.

Therefore, we identify the following requirements:

- Deadline should be provided at submission time, so this information is associated with the application.
- A fine-grain monitoring of the task (e.g. consumed CPU time for the individual task) is needed. Considering that the tasks run in a distributed environment, this may require identifying the physical resources where the task runs.
- The progress of the application must be computed and defined. We will use three states: *under-progress*, if the application progress is below a given threshold of the expected progress or CPU time allocation; *ontime*, if the application progress is between the acceptable, expected progress interval, given a threshold; and *overprogress*, if an application has advanced more than a given threshold with respect to the expected progress. The user can configure these thresholds at submission time.
- An actuator will trigger a new resource allocation if needed. An *underprogress* state will lead into an increase of resources and an *overprogress* state into a reduction of such resources. The user can configure these increments or decrements at submission time.
- In case of the application state needs to be frozen and restored, checkpointed is needed.

The motivation for these requirements can be seen in the following scenarios:

- Applications which perform a known number of iterations (with a predictable execution time each one) that should be finished before meeting a deadline. This scenario is implemented using Chronos: The user submits an application to Chronos that iterates multiple times through independent executions, which can not take place concurrently due to flow dependencies, external data dependencies, etc. The user wants to guarantee that the application completes all of the individual executions before a given deadline. Each iteration starts with a specific resource reservation and, according to the application progress at the end of each iteration, the developed mechanism decides to modify the assigned vCPU share to meet the QoS agreed. This modification is decided by the developed mechanism but is performed by Chronos. There is no need to monitor the vCPU usage but only the completion of the iterations.

- Applications whose progress cannot be monitored but have an estimation for the CPU time. They are implemented using long-living Marathon applications: The user submits a long-living application to Marathon with the guarantee that a minimum CPU time has been allocated to that application in a given time frame. The developed mechanism queries the monitoring system periodically to compute the application progress. The actuator system will vary the vCPU share assigned if needed, using Marathon. The application progress must be preserved because, unlike the first scenario, the modification of the assigned resources is performed during the execution. This scenario assumes that other tasks have been scheduled in the same resources with different QoS requirements and submission times, so temporarily speeding up or decelerating other application may lead to a lower overall QoS violation ratio. It should be noted that the mechanism is designed for deciding to modify the assigned vCPU share of an application based only on its progress (not considering the progress of all applications). Similarly to the Chronos scenario, the framework modifies the allocation of resources. In this case, the mechanism requires to monitor the CPU time to each application in the distributed infrastructure.

2.3 Related work

Elasticity is an active research area in Cloud Computing. There are many works focused on horizontal elasticity and vertical elasticity. Horizontal elasticity consists of providing additional resources (more VMs or Docker containers) dynamically to meet an increase or decrease of service demand (e.g. growth of requests in a Cloud service). This approach is appropriate when the problems can be resolved in a parallel way. Vertical elasticity consists of modifying the assigned resources of a Virtual Machine (VM) or a Docker container to meet an increase of demand (e.g. a scientific application that needs more RAM memory for finalizing its execution). The work of this paper is focused on vertical elasticity.

Most hypervisors and Cloud IaaS support vertical elasticity in VMs. OpenNebula and OpenStack provide with resize functions to provide a stopped VM with a higher or lower number of virtual CPUs (or vCPUs) and RAM. However, there is no support in OpenNebula and OpenStack for dynamic resizing of VMs, as they will require acting both at the level of the virtualisation hypervisor (varying the assigned resources to the VM) and at the VM's guest Operating System (updating the resources available in the OS).

One approach for providing vertical elasticity of vCPU for VM's can be leveraging the CPU CAP (the maximum number of CPU resources a VM can use) and the physical memory allocated by the virtualisation hypervisor. These techniques do act only at the level of the virtualisation hypervisor. This way, the internal configuration of the VM's OS remains the same, but it is provided with a higher share of physical resources, so the vCPU can run faster or slower, and have more RAM mapped on physical RAM. [162] presents a mechanism named CloudScale, to automate fine-grained elastic resource scaling for multi-tenant Cloud computing infrastructures. In contrast, other approaches such as CPU hot-plugging requires that both the guest Operating System supports dynamic plugging of CPUs.

From [51], vertical elasticity approaches can be categorised into performance-based, capacity-based, and hybrid approaches. The virtualisation hypervisor supports these approaches with two mechanisms: add or remove memory, also named hot memory plugging, and memory ballooning ([121], [51]). However, in such cases, the allocation of resources is performed at the level of the whole Virtual Machine, which will affect all the applications running on it.

Few works deal with vertical elasticity in Docker containers. [143] provides a tool to perform vertical elasticity in Cloud infrastructures manually. [2] presents Elasticdocker, which is a mechanism that modifies the allocated resources (CPU time, vCPU cores and memory) of a Docker container according to the workload demand. Elasticdocker monitors the CPU time, vCPU utilisation, number of vCPUs and memory usage to take the elasticity decisions and implements these decisions modifying the `cgroups` pseudo-files of the Docker container directly.

There is also some related work on mechanisms to address elasticity aspects in Mesos frameworks. [4] provides a mechanism to scale in and out stateless Marathon services (horizontal elasticity) but considering that services do not store a persistent state so they can be restarted. This approach takes into account both memory and vCPU. Regarding vertical elasticity, a Mesos executing and monitoring framework called Makeflow [185] is proposed for a bag of tasks, which adjusts the number of vCPUs of a series of independent jobs according to their actual utilization.

To the best of our knowledge, there are currently no tools to provide vertical elasticity for applications embedded inside Docker Containers executed by Mesos frameworks (Marathon and Chronos). Thus, the work described in this paper aims to design and implement a flexible mechanism for applications that vary the vCPU share to a specific application in a highly loaded infrastructure to meet the agreed QoS.

2.4 Underlying technologies

This work relies on a set of well-known technologies for containers, resource orchestration, checkpointing, application submission and monitoring.

2.4.1 Application Delivery

We use Docker containers to run applications. Formally, a container is a group of processes executed in an isolated and controlled way on the host kernel on an independent and virtualised filesystem. It provides a convenient mechanism to embed software dependencies for application delivery. Containers hold the whole execution context for an application. In this sense, CRIU [176] is a project for Linux operating system that enables to freeze a running application as a collection of files called checkpoint. Using checkpointing a user can freeze and restore the application at the same execution point as it was when the checkpoint was made, even in another machine. Currently, CRIU is included into many Linux distributions and it is integrated into containers platforms like OpenVZ [177], LXC [105] or Docker (with the experimental release).

2.4.2 Resource management

Apache Mesos is a middleware comprising a set of applications and services for cluster management that provides efficient resource isolation and sharing across distributed applications (frameworks) or roles (users). Mesos can run on top of Virtual Machines, bare metal or Docker containers. Mesos are composed of two main components: the master and a set of agents (or worker nodes).

The Mesos master manages agent daemons running on the worker nodes. Worker nodes run Mesos agents that register with the master and offer their resources (CPU, disk, ports, special hardware, etc.). Mesos provides two mechanisms for reserving the resources discovered in the worker nodes for the execution frameworks: static and dynamic reservation. Static reservation consists of specifying the resources reserved for a certain role in the agent start-up. Dynamic reservation enables roles and authorised frameworks to reserve or liberate resources after agent start-up.

Execution frameworks, in turn, are composed of two main components: scheduler and executor. The framework scheduler registers with the Mesos master for receiving resource offers. When an offer is available, the framework scheduler

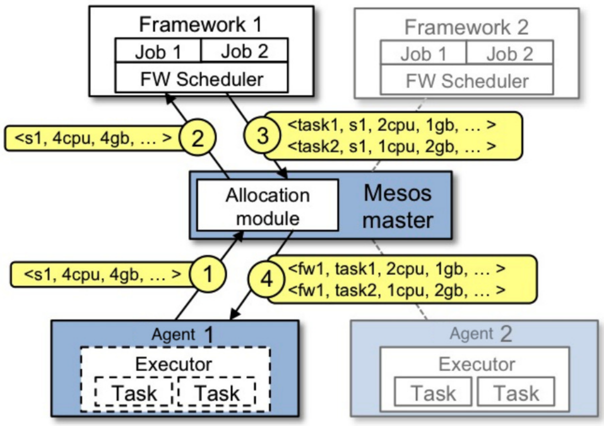


Figure 2.1: Example of resource offer from Mesos agent and how it is assigned to a task of a framework's executor. Image source: [74].

sends the information of the number of resources that it will be assigned to the framework's tasks to the Mesos master. Then, the Mesos master sends the framework's tasks to the corresponding Mesos agent, which allocates appropriate resources to the framework's executor. The framework's executors are in charge of launching the framework's tasks in the Mesos agent. The Figure 2.1 illustrates this procedure.

There are some frameworks [8] for Mesos to execute long-running services (such as Aurora or Marathon), Big Data processing (Spark, Hadoop, Storm, etc.), batch scheduling (Chronos, Cook, Jenkins, etc.), data storage (Cassandra, Ceph, etc) and machine learning (e.g. TFMesos).

Marathon is high-availability and fault-tolerant Mesos framework that acts like a *Platform as a Service* (*PaaS*). This framework is designed for managing long run services, i.e., services that must be permanently running. Marathon obtains high availability thanks to periodic checks of the task health. Activating the checkpointing feature in Marathon enables tasks to continue running during Mesos-slave restarts and Marathon scheduler failovers. The users can interact with Marathon using the REST protocol or graphical web interface.

Chronos is a distributed and fault-tolerant Mesos framework used for job orchestration. This framework is a scheduler that can manage tasks that need to be triggered periodically. Besides, it supports the creation of dependent jobs chains. This type of applications will run when a list of depending applications have been completed before at least once. The users can interact with Chronos using a REST API or a graphical web interface.

2.4.3 Monitoring

Monitoring is a crucial issue when managing elasticity. Monitoring should be fine-grained to focus on the real consumption of resources of the specific task that is monitored, and lightweight to add the minimum overhead and disturbance to the system. For this purpose, OpenStack Monasca [141] has been selected, which is an open-source, highly-scalable, multitenant and fault-tolerant monitoring tool integrated with OpenStack [165].

In Monasca, a *Metric* defines a type of monitored resource. A metric is defined by a name, often representing a hierarchy, and a set of dimensions. The unit of monitoring is a measurement, which contains values for the dimensions and a timestamp. When a *Metric* is stored in Monasca, the user can perform queries or define *Alarms*, which are a composed by a boolean expression and a *Notification* method. This expression compares the result obtained by evaluating a *Metric* with a threshold. In turn, each expression can be formed by one or more expressions.

The data is stored in Monasca by two entities: the users and the Monasca Agent [134]. The users can send *Metrics* to Monasca, once they are authenticated by OpenStack KeyStone [139], using the command-line interface or a REST API [140]. The Monasca Agent [134] is a Python tool that gathers *Metrics* using available plug-ins [136]. There are default plugins for standard resources (such as IO, network, CPU usage) and even for monitoring higher-level services and applications (such as CEPH, MongoDB, Kafka, among others). In this work, Monasca data is stored by the Monasca Agent, and the component Supervisor (described in Section 2.5) of the mechanism developed.

2.5 The proposed system architecture

As mentioned before, the two scenarios considered are a Chronos application with multiple independent iterations and long-living Marathon applications. According to the scenario, the architecture varies in some aspects, which will be explained in this section. Figures 2.2 and 2.3 depict the design of the architecture for both scenarios. Applications are specified using JavaScript Object Notation (JSON) format.

The mechanism architecture is composed of three main components: Launcher, Supervisor and Executor. These components are implemented in Python.

The Launcher is a command-line tool in charge of the submission of the application. The user runs the Launcher specifying the application in JSON format with the QoS information, the parameters to connect and configure the Supervisor, the address of Supervisor and the credentials of the Mesos frameworks (Chronos and Marathon). First, the Launcher provides each application with a unique identifier (UUID). Then, the Launcher creates a new application specification based on the user application specification. The changes (that are detailed in Section 2.5.1) for creating the modified specification are done according to the selected scenario. Afterwards, the Launcher submits via REST the new application specification to the appropriate framework. Once submitted, the Launcher sends through a REST call the relevant information (Section 2.5.1 details this information) to the Supervisor.

The Supervisor is a REST service that receives the information for monitoring from the Launcher and the Docker containers (applications) that are running on the working nodes. Besides, it computes the application performance state and decides if scaling is needed. If the assigned vCPU share must be modified, the Supervisor obtains the application specification from the framework scheduler (Marathon or Chronos), changes it and re-submits it to the framework. Then, the framework relaunches the application with the new vCPU share assignation (it should be pointed out that, in Marathon case, the framework terminates the current execution losing the execution progress).

The Supervisor monitors the applications with two approaches: passive and active. The passive approach is chosen for Chronos scenario and the active one for Marathon scenario. In the first approach, when the Supervisor receives the notification from the application that the execution is complete, it computes the progress state and decides if it is necessary to scale. In the active approach, when the Supervisor receives a notification from the Executor (this component is described below), it starts monitoring the application state. This monitoring is periodical and, for each instant of monitoring, the Supervisor decides if an update in the assigned resources is needed. If it is required to update the assigned resources, the Supervisor notifies it to the framework (Marathon or Chronos). Once the execution of the application is completed (the Supervisor receives a notification from the Executor), the Supervisor stops monitoring the application. Meanwhile, the Supervisor sends metrics to Monasca so that progress can be displayed.

All the applications are embedded in Docker containers. In the Chronos scenario, the scaling decisions are implemented between iterations, so checkpointing is not required to maintain the execution state. In the Marathon scenario, the scaling decisions are implemented during the execution of the Docker container. The container will be forced to restart by Marathon and checkpointing will preserve the execution state. An additional component, the Executor, is required to start and resume the execution of a container from a Checkpoint. This requirement is a consequence of the fact that Marathon cannot initiate Docker containers based on checkpoints, so it cannot handle the stop-resume process after an adjustment. It should be noted that, when applications are not embedded in a Docker container in Chronos or Marathon, these applications are embedded in Mesos Native container, so the Executor runs inside a Mesos Native container.

First, the Executor performs several preprocessing tasks: registering initial data on Monasca, checking the existence of a previous checkpoint to be resumed and notifying that the application execution will be started through a REST call to the Supervisor. Then, it starts or resumes the execution of the Docker container from a checkpoint stored in an accessible directory for all worker nodes. Afterwards, the Executor waits until the Docker container's execution finishes. For this purpose,

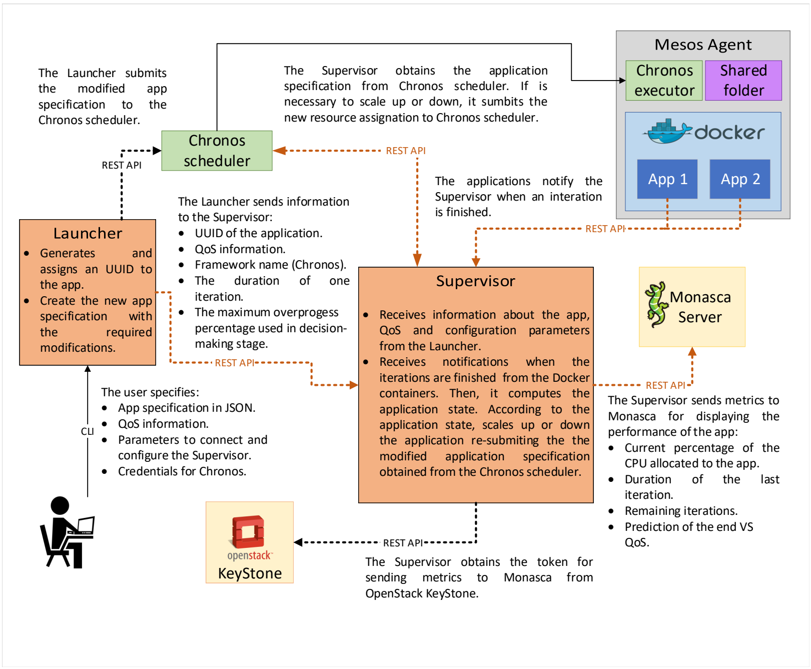

The Launcher submits the modified app specification to the Chronos scheduler.
The Supervisor obtains the application specification from Chronos scheduler. If is necessary to scale up or down, it sumbits the new resource assignation to Chronos scheduler.
Mesos Agent
Chronos executor
Shared folder
docker
App 1
App 2
Chronos scheduler
REST API
REST API
The applications notify the Supervisor when an interation is finished.
The Launcher sends information to the Supervisor:
UUID of the application.
QoS information.
Framework name (Chronos).
The duration of one iteration.
The maximum overprogess percentage used in decision-making stage.
Launcher
Generates and assigns an UUID to the app.
Create the new app specification with the required modifications.
REST API
REST API
Supervisor
Receives information about the app, QoS and configuration parameters from the Launcher.
Receives notifications when the iterations are finished from the Docker containers. Then, it computes the application state. According to the application state, scales up or down the application re-submiting the the modified application specification obtained from the Chronos scheduler.
Monasca Server
REST API
The Supervisor sends metrics to Monasca for displaying the performance of the app:
Current percentage of the CPU allocated to the app.
Duration of the last iteration.
Remaining iterations.
Prediction of the end VS QoS.
The user specifies:
App specification in JSON.
QoS information.
Parameters to connect and configure the Supervisor.
Credentials for Chronos.
CLI
openstack
KeyStone
REST API
The Supervisor obtains the token for sending metrics to Monasca from OpenStack KeyStone.

Figure 2.2: Design of the architecture for the Chronos scenario.

it runs a command (`docker logs -f container_id`) that ends when the Docker container's execution is finished.

When the Supervisor indicates to the Marathon scheduler that the assigned vCPU share must be modified, the Marathon scheduler notifies that the current execution will be terminated to the Marathon executor. If the Executor receives a signal from the Marathon executor, it makes a checkpoint and stores it to a directory shared by all Mesos agents. Once the execution of the Docker container is finished, the Executor notifies it through a REST call to the Supervisor and cleans the shared directory. As mentioned above, the Executor detects when the Docker container ends its execution because the Executor is running a command that ends when the Docker container is finished.

The architecture flow can be divided into four stages: application submitting and execution, monitoring, decision-making, and adjustment.

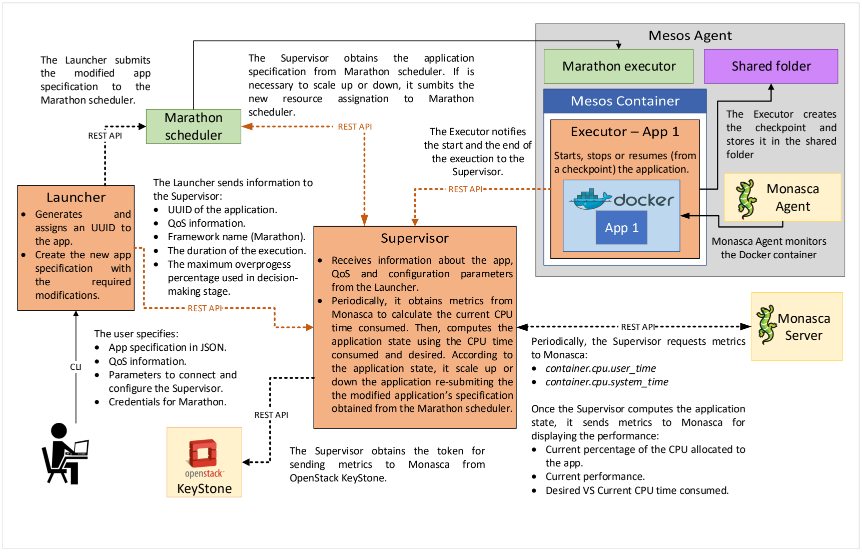

Mesos Agent
Marathon executor
Shared folder
Mesos Container
Executor – App 1
Starts, stops or resumes (from a checkpoint) the application.
docker
App 1
The Executor creates the checkpoint and stores it in the shared folder
Monasca Agent
Monasca Agent monitors the Docker container
The Launcher submits the modified app specification to the Marathon scheduler.
The Supervisor obtains the application specification from Marathon scheduler. If is necessary to scale up or down, it sumbits the new resource assignation to Marathon scheduler.
Marathon scheduler
REST API
REST API
The Executor notifies the start and the end of the exeuction to the Supervisor.
REST API
Launcher
Generates and assigns an UUID to the app.
Create the new app specification with the required modifications.
The Launcher sends information to the Supervisor:
UUID of the application.
QoS information.
Framework name (Marathon).
The duration of the execution.
The maximum overprogess percentage used in decision-making stage.
Supervisor
Receives information about the app, QoS and configuration parameters from the Launcher.
Periodically, it obtains metrics from Monasca to calculate the current CPU time consumed. Then, computes the application state using the CPU time consumed and desired. According to the application state, it scale up or down the application re-submiting the the modified application's specification obtained from the Marathon scheduler.
REST API
REST API
Monasca Server
Periodically, the Supervisor requests metrics to Monasca:
container.cpu.user_time
container.cpu.system_time
The user specifies:
App specification in JSON.
QoS information.
Parameters to connect and configure the Supervisor.
Credentials for Marathon.
CLI
REST API
openstack
KeyStone
The Supervisor obtains the token for sending metrics to Monasca from OpenStack KeyStone.
Once the Supervisor computes the application state, it sends metrics to Monasca for displaying the performance:
Current percentage of the CPU allocated to the app.
Current performance.
Desired VS Current CPU time consumed.

Figure 2.3: Design of the architecture for the Marathon scenario.

2.5.1 Application submission and execution

The Chronos scenario deals with several executions of an application, which must be completed according to the given deadline. As described above, checkpointing is not required for this scenario. Therefore, the Chronos executor directly manages the Docker container. The modifications needed to create the new application specification for this scenario focus only on the monitoring process. The Launcher adds a REST request to the Supervisor in the original command field of the application specification. This REST call is inserted in the command field after the initial command's value, making the Docker container to invoke a REST request to Supervisor once the execution is completed. The request's message is a JSON object formed by the start and the end of the application execution, the application name and UUID.

The Marathon scenario aims to guarantee that the mechanism allocates to the application (at least) enough vCPU share for meeting the targeted QoS, which is defined as the number of vCPU time that has to be allocated. As described above, the application specification must be modified to manage the

Docker container, i.e., to run the component Executor. The modifications in the application specification consist of three parts. First, the Launcher obtains, from the application specification, all the information required for launching the Docker container. As the Executor will manage the Docker container, the Launcher removes the Docker container definition of the application specification. With this changes, the task launched by Marathon will assign the same resources as the original application but embedded in a Mesos Native container instead of a Docker container. Finally, the Launcher changes the command of the original application specification by the command that runs the Executor with the information required for notifying Supervisor, in order to launch the container (obtained at the start of the modification of the application specification), and the application UUID.

Once these changes are done, the Launcher submits the application through a REST call the application to the appropriate framework (Chronos or Marathon). Besides, for both scenarios, the Launcher sends through a REST call the relevant information to the Supervisor. The relevant information comprises the application name, the application UUID, the framework name, the duration of one application execution, the targeted QoS, the maximum overprogress percentage, and, in Chronos case, also the number of iterations is sent. The framework name can be Chronos or Marathon.

It should be noted that the user only has to define two parameters about the application in the specification of each application: the duration of one iteration and the maximum overprogress percentage. The duration of one execution parameter is the most complex to define by the user. In Chronos case, it is an intermediate value between the estimated duration of the execution using one vCPU and the maximum number of vCPUs in one worker node. It should be pointed out that, if this value is closer to the estimated time using one vCPU, the Supervisor increments the assigned resources more easily than in the opposite case, because the prediction calculated by the Supervisor is more pessimistic. In Marathon case, the duration parameter is the estimated amount of CPU time (in seconds) that the system must guarantee to the application before meeting the deadline. The definition of this amount of time can be estimated by the study of the CPU time consumed (user and system) by the Linux processes of the application. These values are located in the `cgroups` files assigned to each Linux process. Besides, it should be noted that the goal of the QoS mechanism is to guarantee that enough vCPU time is assigned to a process to complete its execution before meeting the deadline. This fits those applications whose progress cannot be evaluated during its execution. The maximum overprogress percentage is the percentage in which the user considers that the application's performance is higher enough and it can be resized. For example, if the maximum overprogress percentage is 0.2, the user considers that the system will decrement the assigned resources if the application's performance is 20% higher than the required for completing the execution on time.

2.5.2 Monitoring stage

Once that the application is launched in the Mesos agent by the framework executor, the mechanism monitors it to collect its performance to update (if necessary) its assigned resources. As described in Section 2.5, this work considers two approaches for monitoring: passive and active.

The passive approach is used in the Chronos scenario. In this approach, monitoring is performed by the Supervisor. The Supervisor receives a message from the application (Docker container) each time an iteration finishes its execution. This message is the result of the modification done by the Launcher in the application specification (the Launcher puts a REST request in command field after the original command) described in Section 2.5.1. When this message arrives to the Supervisor, it starts the decision-making stage to meet the QoS agreed. In addition, the Supervisor decrements the remaining iterations of the application.

As the aim of the Marathon scenario is not to run a determined number of iterations, monitoring is done during the execution of the application. For this purpose, the active approach is used in the Marathon scenario. As mentioned above, one of the first tasks of the Executor is to notify the Supervisor using a REST request (the body of its message is formed by the application UUID and the timestamp of the start of the execution) that the application has been started. Once the Supervisor receives this message, it starts to monitor periodically. As the Supervisor does not have access to the Mesos agents, it uses the metrics of the Docker container's performance from Monasca. The Monasca Agent collects these metrics for each container. It should be noted that a modified Docker plug-in for the Monasca Agent (available in the GitHub repository of this work [157]) is required to enable monitoring of restored containers from checkpoints. Whenever the Supervisor obtains the collected data from Monasca, it starts the decision-making stage to meet the QoS agreed. This procedure is periodically repeated until the Executor notifies to the Supervisor that the application execution is finished. The body of this notification message is formed by the application UUID and the timestamp of the end of the execution.

2.5.3 Decision-making stage

Once the data collection in monitoring stage is completed, the next step consists in determining the application performance state. As discussed in Section 2.2, there are three states for the application: *underprogress*, *ontime* and *overprogress*. The Supervisor uses the Algorithm 1 to determine the state of the application. This algorithm has the same input in both scenarios: the application performance percentage ($performance$), the over-progress threshold (threshold$_{overprogress}$) and the under-progress threshold (threshold$_{underprogress}$). The threshold$_{underprogress}$ is set by default to 0.9, which means that an application is considered in the under-progress state when its performance is a maximum of 10% below the required performance. The threshold$_{overprogress}$ is the obtained using the Equation 2.1 with the maximum over-progress percentage ($max\_overprogress$), which is a parameter sent by the Launcher to tune the Supervisor's configuration for each application launched. If $max\_overprogress$ is set to 0.2 by the user, the threshold$_{overprogress}$ is 1.2, which means that an application is considered in the over-progress state when its performance is at least of 20% above the required performance.

$$threshold_{overprogress} = 1.0 + max\_overprogress \tag{2.1}$$

where *max_overprogress* is the maximum overprogress percentage sent by the Launcher in 2.5.1.

Algorithm 1: Algorithm used by the Supervisor to obtain the application state

```
Input : performance, $\text{threshold}_{overprogress}$ and $\text{threshold}_{underprogress}$
Result: application_state
begin
    if $performance > threshold_{overprogress}$ :
        application_state = overprogress ;            /* Decrease assigned
         resources */
    else if $performance < threshold_{underprogress}$ :
        application_state = underprogress ;           /* Increase assigned
         resources */
    else:
        application_state = ontime ;                   /* Nothing to do */
end
```

The $performance$ value is obtained differently according to the scenario. The user specifies the QoS to the Launcher. As it is described in Section 2.5.1, the Launcher specifies the target QoS in two formats: as a deadline in timestamp format or as a time frame in seconds. This action is done for facilitating the computation of the $performance$ to the Supervisor. In the Chronos scenario, as the mechanism guarantees that the application is executed within a determined number of iterations, the QoS is a time limit to complete all the iterations. For example, the QoS of an application establishes that ten iterations of this application must be executed before 9 a.m. (QoS is 9 a.m. but in timestamp format). In the Marathon scenario, as the mechanism guarantees a predefined minimum CPU time to the application according to a given deadline, the QoS is allocating enough vCPU share to finish the application during the time frame. For example, QoS is 300 seconds for an application that needs 275 seconds of

CPU time (the duration of the application) to be executed time before the next five minutes.

In case of the Chronos scenario, the decision-making stage is done when each iteration of the application finalizes its execution. Once an iteration completes its execution, the Supervisor obtains the $performance$ value using the Equation 2.2 for determining the application state using the Algorithm 1. This equation requires the estimated finalization time of all iterations, $t_{prediction}$, which is obtained using Equation 2.3.

$$performance = \frac{t_{prediction}}{t_{qos}} \tag{2.2}$$

where $t_{prediction}$ is the estimated finalization time of all iterations in timestamp format obtained using the Equation 2.3 and t_{qos} is the QoS agreed in timestamp format (obtained from the Launcher in Section 2.5.1).

$$t_{prediction} = t_{current} + rem\_iter * (d_{iteration} + d_{deployment}) \tag{2.3}$$

where $t_{current}$ is the current time in timestamp format, $d_{iteration}$ is the expected duration of one iteration execution (sent by the Launcher), $d_{deployment}$ is the upper bound value of the deployment time of the new iteration in the infrastructure (sent by the Launcher), and rem_iter is the number of remaining iterations.

In the Marathon scenario, the decision-making stage is periodically done (just before the monitoring stage). Once the Supervisor ends each monitoring stage, it immediately obtains the $performance$ at determined time t of the execution using the Equation 2.4. Once the $performance$ is obtained, the Supervisor uses the Algorithm 1 to determine the application state.

$$performance(t) = \frac{cputime_{current}(t)}{cputime_{desired}(t)} \tag{2.4}$$

where $cputime_{current}(t)$ is the real CPU time consumed at determined time t for the application (the computing of this value is detailed below) and

$cputime_{desired}(t)$ the expected CPU time at determined time t for meeting the QoS agreed.

As real progress information is not available, the Supervisor estimates it by computing the CPU time consumed on a given execution time $cputime_{current}(t)$. It can be computed from the information obtained by two queries to Monasca (*container.cpu.user_time* and *container.cpu.system_time*). These values correspond to the total user and system clock ticks consumed by the container in the working node where it is running. These values are transformed into seconds, dividing them by the clock ticks per second constant of the system.

The $cputime_{desired}(t)$ is the CPU time that the application should have consumed at determined time t for meeting the QoS agreed. The Supervisor uses the Equation 2.5 to calculate $cputime_{desired}(t)$ in seconds. As it is described above, in the Marathon scenario, the duration of the application execution ($d_{application}$) is the CPU time that the developed mechanism must allocate in the time frame agreed (t_{qos}). As t_{qos} is the available time frame for allocating $d_{application}$ seconds, it is necessary to calculate the percent of the time frame available (d_{qos}) that the application has consumed at a determined time t. This percentage is calculated with $\frac{t_{current}-t_{start}}{t_{qos}}$. Using this percentage and the CPU time that the application has to consume before to meet to QoS, $d_{application}$, the CPU time that the application should have consumed at determined time t can be calculated using the Equation 2.5.

$$cputime_{desired}(t) = \begin{cases} (\frac{t_{current}-t_{start}}{t_{qos}}) * d_{application} & \text{if } t_{current} \leq t_{start} + t_{qos} \\ d_{application} & \text{otherwise} \end{cases} \tag{2.5}$$

where $t_{current}$ is the current time in timestamp format, t_{start} is the time when the application starts its execution in timestamp format (sent by the Executor), $d_{application}$ is the CPU time in seconds that the developed mechanism must allocate in the time frame agreed (sent by the Launcher) and t_{qos} in seconds is the time frame for allocating $d_{application}$ seconds.

At this point of the architecture's flow, the Supervisor downloads the application specification for obtaining the last value of the vCPU share that was assigned. Then, it sends to Monasca, through a REST call, information about the application performance and the vCPU share, in order to allow displaying it.

2.5.4 Adjustment

Once the Supervisor completes the decision-making stage, it determines, according to the application state, if the application must be resized. If the application state is *overprogress* or *underprogress*, the Supervisor varies vCPU share assigned to the application. It should be remembered that the Supervisor determines the scaling decision (increasing or decreasing the vCPU share), but it is the framework (Chronos or Marathon) executor who implements it. The mechanism implements the variation of the resources by changing and re-submitting the application specification to the framework by means of a REST API. In both scenarios, the framework halts and restarts the application, starting it on a (not necessarily different) working node.

In Chronos case, the Supervisor modifies (increments or decrements the vCPU share and ensures that the remaining iterations value of the application is correct) the application specification, which was downloaded at the end of the decision-making stage. The amount of vCPU share that it is incremented or decremented is set by default at the startup of the Supervisor. In Marathon and Chronos, if an application has assigned 1.0 vCPU, it means that the application has a 100% of the vCPU share of one vCPU. Thus, if an application has 1.5 vCPUs in a node with 2.0 vCPUs, the application has reserved all of the vCPU share of one vCPU and the 50% of the vCPU share of the other vCPU. After several tests, it has been observed that increasing or decreasing the vCPU share by 0.4 leads to the best results. Once the modification is done in the application specification, the Supervisor sends it using a POST request to the REST API of the Chronos scheduler. Therefore, the new iteration of the application will be executed with the vCPU share. As the adjustment stage in Chronos scenario is done between iterations, is not necessary to preserve the state of the execution.

The adjustment stage in the Marathon scenario is more complex than in the Chronos scenario because the mechanism resizes the assigned resources while the application is running. As it is described above, the Executor is in charge of managing the Docker container. According to the state of the application obtained in Section 2.5.3, the Supervisor modifies vCPU share assigned of the application specification downloaded at the end of the decision-making stage. When the Supervisor re-submits the modified application specification with the new value of the percentage of CPU time using a PUT request to the REST API of the Marathon scheduler, the Marathon executor (on the Mesos agent) sends a termination signal to the process that is running the application before removing it. This process is the Executor, which is running embedded in a Mesos container. The termination signal sent by the Marathon executor is captured by the Executor, which creates a checkpoint of the running container and stores it in an accessible location by all worker nodes (Mesos agents) of the infrastructures. As soon as the application with the new vCPU share allocation is running, the Executor restores the checkpoint created previously to resume the execution at the exact point in which the termination signal was captured.

2.6 Results and discussion

In this work, a Mesos cluster has been set up. It comprises five Mesos agents, version 1.2.0, a Marathon (version 1.4.3) and a Chronos (version 2.1.0) framework. Monasca (version 1.6.1) is used for monitoring. Docker (version 17.05.0-ce) running under the experimental mode is used as container engine and CRIU (version 2.6) is used for checkpointing. All machines are virtual machines (VM) deployed on top of an OpenNebula IaaS [137] with 2 CPUs and 4 GB of memory RAM, and they are connected through a private network. Only the front-end has a public IP and all the main services have been deployed there. Figure 2.4 shows an architecture diagram of the solution. It is important to outline that we configure Mesos to enable running jobs to use all the free CPU cycles available in the node, beyond the allocation of resources given by Mesos. Therefore, the CPU allocation will only be limited if other applications are competing in the same node, considering the resource share distribution requested in the application deployment. This feature is desirable in our system, as applications will make use of idle CPU cycles from powered-on resources, over progressing and tentatively releasing more resources in the future for other applications closer to the deadline.

Deployment has been done using the Infrastructure Manager [15] Orchestrator and using the Elastic Cloud Computing Cluster (EC3) [119] client recipes from project EUBra-BIGSEA [47]. This environment is easy to reproduce as the recipes are publicly available in the Docker container *eubrabigsea/ec3client* available in [148]. A Network File System (NFS) is used along all the nodes to share a directory where the CRIU checkpoint images will be stored. No changes have been applied to any of the above components, to facilitate migrating to new versions. We rely on the REST API of Marathon [108], Chronos [24] and Monasca [140] and develop the necessary services at the front end to deal with the application registration and scaling decisions. The code is done in Python and it is publicly available in GitHub [157].

Many applications may benefit from the proposed system, but it is especially useful for applications with high CPU usage. For this reason, the benchmark LINPACK [84] is selected for testing the system. As the worker nodes have 2 vCPUs, the tests use a parallel version of LINPACK benchmark (source code is available in [86]). The parallel version was optimized using OpenMP [132]. Source code of the parallel version and binary executable are available in a Docker image [156].

We evaluated the overhead of the monitoring system by registering the CPU time used (both user and system). The CPU time used of the monitoring probes and services lied below 10% (mostly below 5%) of a single vCPU core. As the monitoring service is needed to register other variables of the system, the monitoring system cannot be entirely switched off. Therefore, this overhead is an upper bound of the effect of introducing the mechanism proposed in the article.

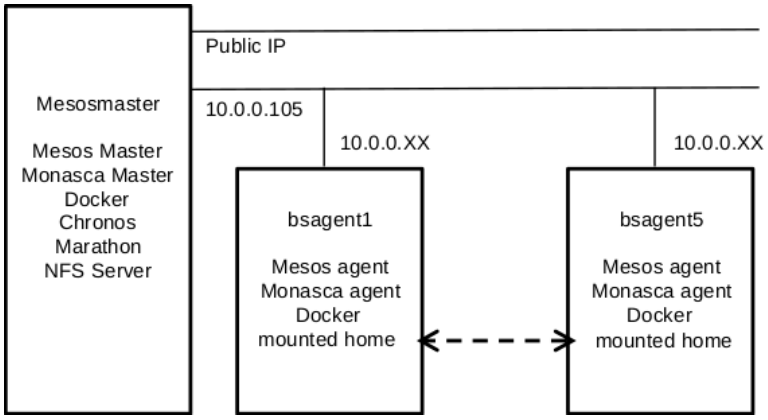


Figure 2.4: Infrastructure and software deployment.

2.6.1 Use case 1: Chronos

The Chronos use case involves the execution of six applications. All applications are the LINPACK benchmark (previously mentioned). Each application will require to execute a different number of iterations and different matrix input size. Table 2.1 shows the information about these applications. In this table, each row corresponds to a different application instance with different requirements in memory and execution time. The column *iteration* denotes the number of Chronos iterations executed per application. The columns *Av. Time* denote the average execution time of a single iteration with 1 or 2 vCPUs respectively. The execution time depends on the size of the problem, shown in the last column (*Matrix dim.*). The use of VMs mainly causes the speedup of the multithreaded application to be very reduced. In the Chronos scenario, the *App duration* column is the expected duration of an iteration in seconds. In the Marathon scenario, the *App duration* column is the CPU time in seconds that the mechanism must allocate in the given time frame. It is important to point out that, instead of asking for specific resource allocation, applications define a feasible time frame in which they will require more than 1 vCPU (and less than 2 vCPUs, which is the per-node limit). The *Prediction* columns depicts the expected total execution time. This value is calculated using the maximum deployment time for each iteration (this value is set to 45 seconds) and the expected duration of one iteration (the column

Table 2.1: Information about QoS, the averaged execution time, the estimated duration defined by the user, the matrix dimension parameter of the application and the results of the experiment (time in seconds).

Name	Iterations	Av. Time 1 iter - 1 vCPU	Av. Time 1 iter 2 vCPUs	App Duration	Prediction 1 vCPU	Prediction 2 vCPUs	Results	Matrix Dim.
app1	4	480.4	362.2	400	2101.6	1628.8	-328	6000
app2	4	480.4	362.2	400	2101.6	1628.8	-340	6000
app3	13	162.75	113.2	130	2700.75	2056.6	36	4000
app4	11	162.75	113.2	130	2285.25	1740.2	-105	4000
app5	12	162.75	113.2	130	2493	1898.4	133	4000
app6	14	92.36	66.32	80	1923.04	1558.48	-290	3500

App duration). Finally, column *Results* is the difference between the QoS agreed and the real execution time of each application in the experiment. The negative values of the column *Results* show that the application ended before the target QoS. Figure 2.5 depicts the difference in seconds between the prediction of the execution finalization of all iterations and the QoS agreed (time limit to finalise all iterations).

This use case considers a highly-loaded infrastructure. If there were always resources for the application to run in the best conditions, vertical elasticity would not be needed. In this use case, all the iterations from each application have a similar completion time. Thus, it is reasonable conclude that the completion time of all executions will grow linearly.

Therefore, it is possible to compute an estimation of the finalization time using the averaged completion time of each execution (the column *App Duration*) described in Table 2.1 and the deployment time, which it is set to 45 seconds in the infrastructure used for the experiment. The deployment time is the maximum time observed deploying applications in the Chronos framework. The same table shows also these predictions and the results obtained in the test.

If the mechanism allocates 1 CPU for each application, the targeted QoS is not achieved. Due to the fact that the infrastructure provides five node with 2 CPUs each and there are six applications, at least one of them will not reach the QoS agreed. Furthermore, according to the column *Prediction*, two applications (*app3*

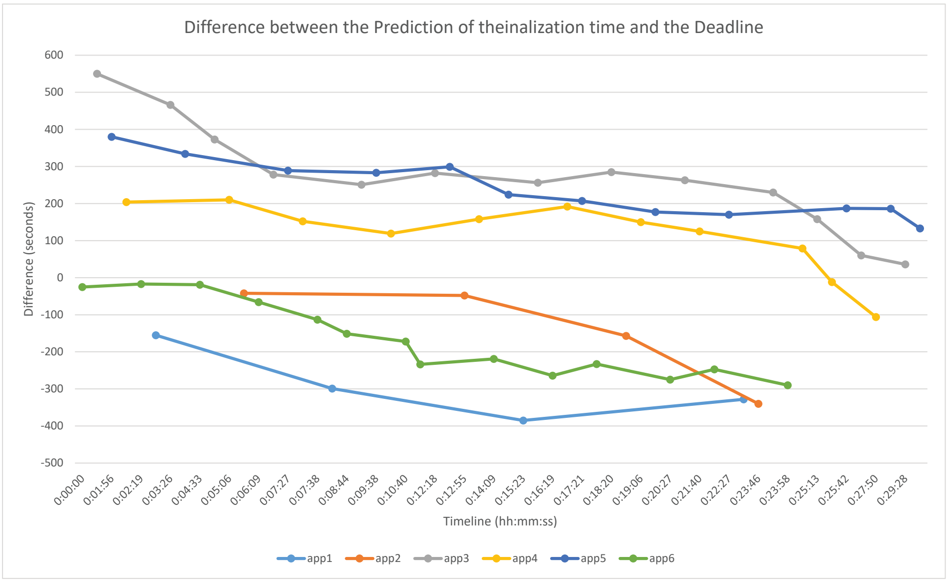


Figure 2.5: Difference between prediction of finalization time and the application deadline for the use case 1.

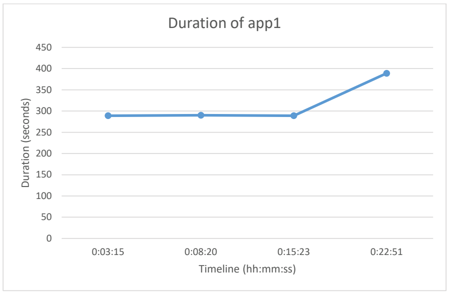
and *app5*) cannot finishing on time. Therefore, the best possible result for the test is to complete four out of six applications on time , which was successfully achieved by the system.

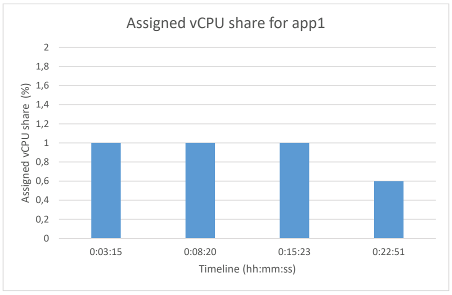
Finally, we can state that no overhead is introduced due to the rescheduling of the Docker containers in Chronos, as the variation of the assigned resources is done between iterations. Chronos always reschedules each iteration as it were a new job. Therefore, changing the reservation of resources does not make any difference.

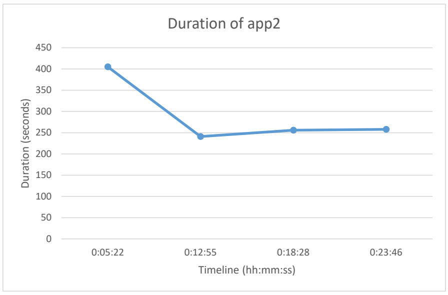
Figure 2.5 depicts the difference between prediction of finalization time and the application time limit. The Y-axis and X-axis represent, respectively, the difference in seconds and the time when the sample was collected in CEST time. The Y-axis coloured ranges in represent the application state: red, green and yellow for, respectively, *underprogress*, *ontime* and *overprogress*.

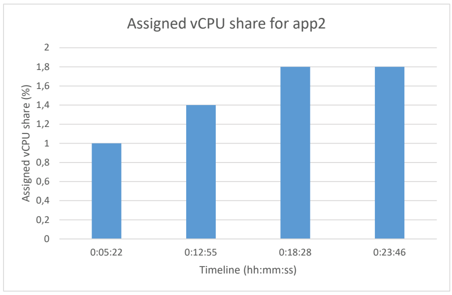
The perfect case for Figure 2.5 would be that an application that begins far from the zero value (in the Y axis) terminates in the zero. If an application begins above the zero mark, means that is into underprogress state. If it is more than the defined threshold below the zero mark, it is into overprogress state. Otherwise, it is into ontime state. If an application concludes its execution at the zero mark means that the application has finished its execution just at the moment before breaking the targeted QoS. Thus, if an application has a downwards tendency, it tendency means that it is improving its performance. This decreasing tendency is common in applications that have been in underprogress state because the mechanisms had increased their assigned resources. Using the same reasoning, when an application has an increasing tendency means that the application is running slower than what is needed to meet the targeted QoS.

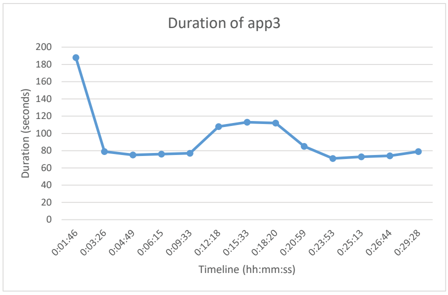
From Figure 2.5 it can be seen that the red and green region applications have a decreasing tendency. In case of the applications of the red region, this tendency is consequence of the increase of assigned resources during the executions. In case of the green region applications, the applications have assigned more resources than needed for meeting the target QoS. In fact, the performance of the application *app1* is so high that the mechanism can reduce its assigned resources. It should be pointed out that, the resizing of the application *app1* causes the change of its tendency.

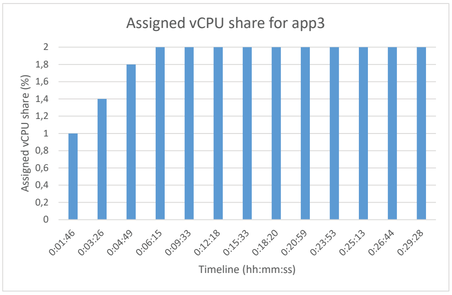


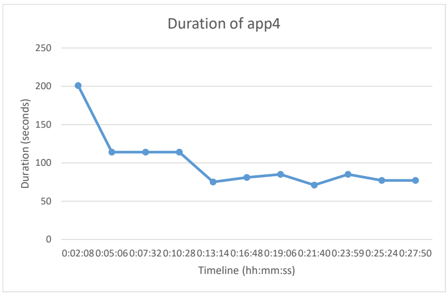


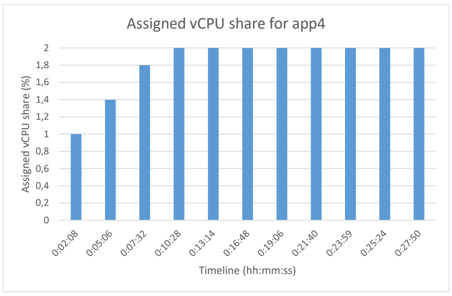

Duration of app2
Duration (seconds)
450
400
350
300
250
200
150
100
50
0
0:05:22
0:12:55
0:18:28
0:23:46
Timeline (hh:mm:ss)

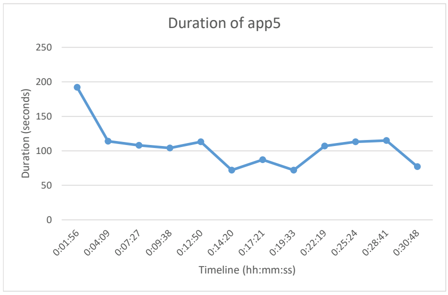

Assigned vCPU share for app2
Assigned vCPU share (%)
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0:05:22
0:12:55
0:18:28
0:23:46
Timeline (hh:mm:ss)

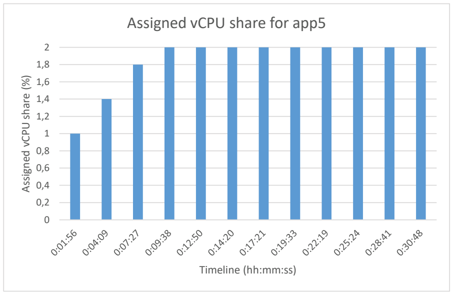

Duration of app3
Duration (seconds)
200
180
160
140
120
100
80
60
40
20
0
0:01:46
0:03:26
0:04:49
0:06:15
0:09:33
0:12:18
0:15:33
0:18:20
0:20:59
0:23:53
0:25:13
0:26:44
0:29:28
Timeline (hh:mm:ss)


Assigned vCPU share for app3
Assigned vCPU share (%)
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0:01:46
0:03:26
0:04:49
0:06:15
0:09:33
0:12:18
0:15:33
0:18:20
0:20:59
0:23:53
0:25:13
0:26:44
0:29:28
Timeline (hh:mm:ss)


Duration of app4
Duration (seconds)
250
200
150
100
50
0
0:02:08
0:05:06
0:07:32
0:10:28
0:13:14
0:16:48
0:19:06
0:21:40
0:23:59
0:25:24
0:27:50
Timeline (hh:mm:ss)


Assigned vCPU share for app4
Assigned vCPU share (%)
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0:02:08
0:05:06
0:07:32
0:10:28
0:13:14
0:16:48
0:19:06
0:21:40
0:23:59
0:25:24
0:27:50
Timeline (hh:mm:ss)


Duration of app5
Duration (seconds)
250
200
150
100
50
0
0:01:56
0:04:09
0:07:27
0:09:38
0:12:50
0:14:20
0:17:21
0:19:33
0:22:19
0:25:24
0:28:41
0:30:48
Timeline (hh:mm:ss)


Assigned vCPU share for app5
Assigned vCPU share (%)
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0:01:56
0:04:09
0:07:27
0:09:38
0:12:50
0:14:20
0:17:21
0:19:33
0:22:19
0:25:24
0:28:41
0:30:48
Timeline (hh:mm:ss)

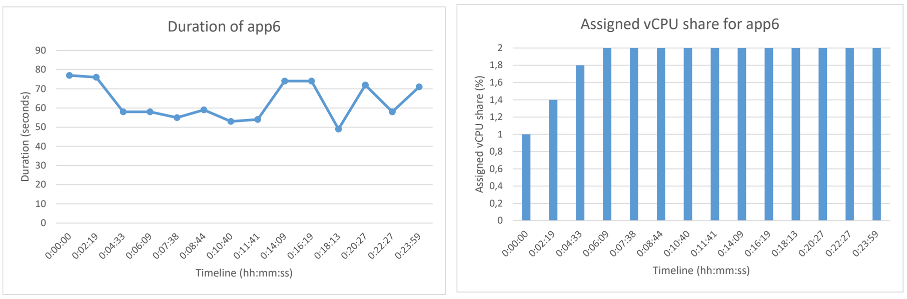


Figure 2.6: Shared CPU time and duration of completion time for each application.

Figure 2.6 shows two graphs for each application. The points in these graphs are collected when an iteration is completed (X-axis) in CEST time. The graphs on the right show the vCPU share assigned for the application and the graphs on the left shows the duration of the execution. As it can be seen in the figure, all applications started with only one vCPU assigned, and the vertical elasticity system varied this resource allocation to correct the evolution of the execution. This fact explains why most of the applications started to be underperforming and progressively improved the performance to fulfil the expected deadline (see figure 2.5). From the results obtained, it seems that the estimation for applications 3, 4 and 5 was too optimistic, which led to a violation of the deadline in two of them.

As the applications are running in multiple iterations (the iterations of each application are independent) there are actually 58 tasks competing for the resources (although only 6 tasks could run concurrently). There is a guarantee through Mesos that there are no over-subscription, so if an iteration requests more resources than available it will be blocked until resources become available. Take into account that this waiting time is added to the total execution time, explaining the different runtime of equivalent iterations (same application and resource assignment).

In Figure 2.6 it can be seen that, for some applications, there is not exists a correlation between the assigned vCPU share and the application duration. Apache Mesos is configured to limit the vCPU share of each containerised task (embedded in a Mesos Native container or Docker container) only when it is competing with other applications (in the same node). Therefore, the real vCPU share used on each container can be higher than the vCPU share assigned.

2.6.2 Use case 2: Marathon

This use case demonstrates the flexibility of the system to update the assignment of resources to an executable task to guarantee that an agreed vCPU time is consumed for a given time frame. This use case analyses:

- The capability of checkpointing a containerised job to resume the execution in a different computing resource at the execution point it was stopped in the origin computing resource. For this purpouse, we evaluate the overheads of checkpointing using CRIU.
- The monitoring an application in a realistic execution environment, sharing the resources with a background workload. For this purpose we will use MONASCA, the Docker plugin and our progress estimation system, which isolates the CPU time spent by a task in the distributed nodes, gathering the accumulated CPU time spent across the different nodes the job has been reallocated to.
- The evaluation of the progress concerning the CPU guarantee and the capability of the supervisor system to update the resource reservation and to fit the new request to the specific node in the infrastructure.

The rest of the section analyses these previous points.

Before starting the experiment, we want to analyse the overhead caused by the use of CRIU. We performed a test consisting of the execution of a parallel Linpack job within a Docker container. The job was checkpointed and immediately restarted to evaluate the following variables:

- The total runtime overhead caused by checkpointing, stopping and restarting a container compared to a run without checkpointing.

Table 2.2: Execution of 5 instances of parallel Linpack with (2) and without (1) checkpointing interruption. In case of (2), the checkpoint was stored in an NFS directory. The checkpointing time is the time taken to create the checkpoint image, which takes place in parallel to the execution (it should not be added or subtracted to the Linpack execution time). Checkpointing in an NFS (3) and a local directory (4) have been measured. Time is in minutes seconds.

	Execution time without checkp. (1)	Execution time with one checkp. (2)	Checkp. time in NFS (3)	Checkp. time in local (4)
#1	06:29,0	06:45,0	00:30,4	0:01,3
#2	06:32,3	06:44,3	00:29,8	0:01,4
#3	06:33,5	06:33,7	00:46,2	0:01,7
#4	06:33,3	06:30,1	00:56,4	0:01,6
#5	06:25,1	06:33,8	00:56,8	0:01,4
Mean	06:30,7	0:06:37	0:00:44	0:01,5
Std. Dev.	00:03,6	0:00:07	0:00:13	0:00,16

- The time requested to create the checkpoint both in an NFS and a local folder.
- The size of the checkpoint image for the experiment job. We repeated the execution of the multithreaded Linpack test for a problem size of dimension 1000.

We repeated the experiment five times, executing one checkpointing and restart per case at different application stages. Table 2.2 presents the results (comparing to the execution without stopping, checkpointing and restarting).

This table shows that the checkpointing only increases (in average) the execution time in 6 seconds. The checkpointing process only considers the layers and memory that are modified in the Docker container, producing a small disk footprint (between 6 and 11 MB). Despite of the fact that the checkpointing in the NFS system takes between 30 and 60 seconds (on the order of 1 second in the case of a local filesystem), the downtime of the application is minimal. Starting the application with the checkpoint takes a negligible cost.

Once the overhead of checkpointing has been analysed, we execute the QoS experiment for the Marathon use case. In this use case, the mechanism aims to guarantee the allocation of a minimum amount of CPU time to the application during a given interval in a congested infrastructure. For this test, the duration of one execution (400 seconds) is a value between the duration in seconds of the application using two CPUs (360 seconds) and the given interval for completion time corresponds with the averaged application duration using one CPU (480 seconds).

The test finalizes successfully because the execution is ended in 393 seconds. This test is very interesting because the application enters into the three possible states. This can be seen in Figure 2.7 where the Y-axis and X-axis represent, respectively, the application *performance* and the time when the sample was collected (in CEST). The Y-axis colored ranges denote the application status: red, green and yellow for, respectively, *underprogress*, *ontime* and *overprogress*.

Computing the real overhead in this case is difficult to evaluate, as several variables are affecting the performance. The previous experiment shown in table 2.2 shows an absolute overhead of 6 seconds per checkpoint. The experiment managed to execute a Linpack test in 393 seconds. The average execution in an isolated two-core working node is around 362 and 480 seconds in a single-core working node, as shown in table 2.1. The test incurred in two checkpoints and restarts, thus leading to three different execution intervals. Although the framework only requested 1 vCPU, the application will try to use all the free resources temporarily unless other jobs run in the system. This has been the case for the middle interval (Figure 2.9 depicts the vCPU reservation). From the execution results, we observe that the resources reserved during the first and last intervals (68% of the time) were between 1 and 2 CPUs. During the middle interval (32% of the total execution), the resources assigned efficiently were between 0.6 and 1 CPU. The overhead of the developed mechanism will be lower than 31 seconds (the difference between the observed execution time and the average time with 2 cores), and higher than the 12 seconds required to perform the 2 checkpoints and restarts.

Figure 2.8 depicts the comparison of CPU time consumed and the CPU time that the system predicts that should be consumed at the moment of which the samples

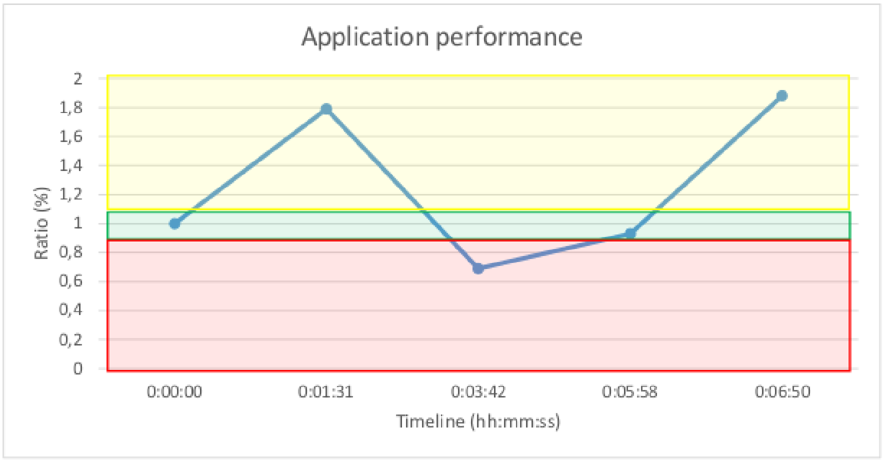


Figure 2.7: Performance value used for compute the application state of the application.

are collected in seconds. These values are used in Equation 2.4 to calculate the performance of the application, whose values are shown in Figure 2.7.

Figure 2.9 shows the assigned vCPU share to the application when the samples are collected. According to application state obtained from ratio progress, the share of CPU time is modified.

The execution trace can be seen from the three graphs. The first sample corresponds to the start of the application. The next sample is collected when the application is in *overprogress* state, thus a decreasing of vCPU share is required. As it can be seen in Figure 2.8, the two following samples of current time consumed are indicating that the application performance is under the expected but, in case of third sample, it is not low enough to trigger an action, thus the state is *ontime*. For this reason, in Figure 2.9 the vCPU share is only incremented once. The last sample is obtained when the execution is completed and, as it can be seen in Figures 2.8 and 2.7, the application ends fulfilling the targeted quality of service with 87 seconds of margin.

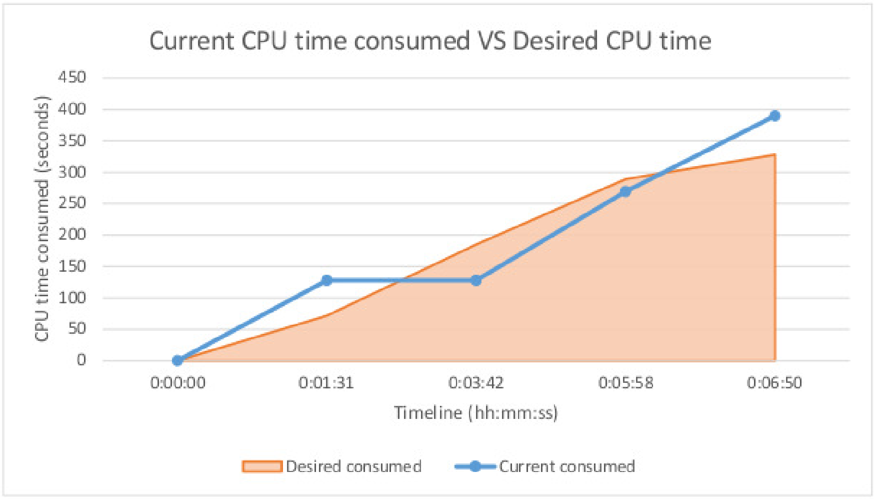


Figure 2.8: Comparison between the current CPU time consumed by the container and the CPU time that should be consumed (both in seconds).

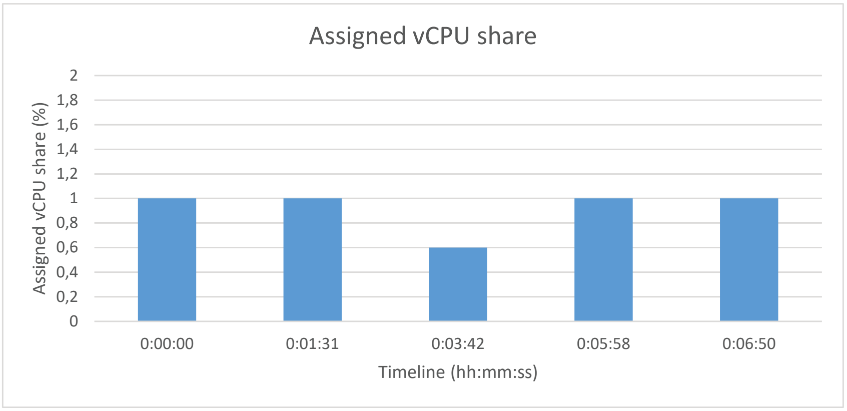


Figure 2.9: Assigned vCPU share for the application.

2.7 Conclusions

Vertical elasticity is the only solution for dynamically speeding up sequential or multicore applications which could not benefit from increasing the number of instances. Vertical elasticity in cloud system usually implies rebooting the services, as hypervisors can resize Virtual Machines only when they are powered off. Memory ballooning and CPU CAP update can be done dynamically without rebooting VMs, but they require privileged access to the hypervisors, which is not the case in most IaaS. The use of container-based applications to dynamically change the resource assignation is a transparent and low-impact technique that can be done in user-space without compromising others executions.

The article demonstrates that the combination of checkpointing and Mesos Frameworks can be used to keep the application progress and even to benefit from the service redeployment capabilities of Marathon and Mesos to find the rightmost resources. This technique has been successfully applied to both iterative jobs (Chronos) and long-run jobs (Marathon) on a busy environment in which workload is dynamic and competitive. The work aimed at ensuring the desired QoS, although it can be used for other policies in the serverless paradigm, enabling the execution and retake of functions.

Chapter 3

A self-managed Mesos cluster for data analytics with QoS guarantees

Published as:
Sergio López-Huguet et al. "A self-managed Mesos cluster for data analytics with QoS guarantees". In: Future Generation Computer Systems *96 (2019), pp. 449–461.* ISSN*: 0167-739X.* DOI*: 10.1016/j.future.2019.02.047.* URL*: http://www.sciencedirect.com/science/article/pii/S0167739X18311087*

3.1 Introduction

The need for data analytics platforms has risen in the recent years, in parallel to the increase in the computing and data storage requirements, in order to tackle the challenges of data processing. Configuring and operating such platforms is not straightforward and requires non-trivial system administration skills. Data analytics platforms involve multiple components and resources, which must be appropriately linked and cross-configured. In addition, dealing with unpredictable workloads is an operationally complex task that requires dynamically readjusting the resources and reconfiguring them on the fly.

In this way, this article presents a set of tools and configuration recipes for deploying a virtual self-managed cluster of computing nodes. The cluster can scale horizontally (in and out), by adding and removing computing resources and reconfiguring them according to the workload, and vertically (up and down), by readjusting the assigned resources to individual jobs dynamically to satisfy a given Quality of Service (QoS).

This paper introduces the problem, the software architecture, the automatic deployment tools and recipes, the elasticity mechanism and the experiments, discussing the results obtained. The reminder of the paper is structured as follows. First, section 3.2 examines the requirements of a data analytics platform and revises the state of the art related to the work presented in the paper. Then, section 3.3 presents the proposed architecture of the platform used to perform data analytics and the mechanisms involved in the elasticity management. Also, a brief analysis of each component involved in the architecture is presented in this section. Section 3.4 describes the most relevant metrics obtained from the deployment of the self-managed virtual cluster and the execution of several test cases to validate the horizontal and vertical elasticity. Section 3.5 discusses the main developments and improvements presented in this work in comparison with the state of the art. Finally, section 3.6 summarizes the main results, concludes the paper and points to future work.

3.2 Requirements & State of the art

This section presents the requirements and reviews the state of the art of the two main areas of research that constitute the basis of this work (cloud orchestration and elastic clusters) as well as other cloud-based processing software architectures that address the requirements identified.

3.2.1 Requirements

In this work, we consider three types of use cases that address three main problems in data analytics [46]. The first use case is data acquisition, which deals with the periodic acquisition of external datasets and the integration with the previously acquired data. The second use case is the development of descriptive models, aiming at deriving additional information and knowledge from raw data. Finally, the third use case concerns predictive models, which build up models for estimating specific variables under new scenarios.

From this analysis, we identified the following technical requirements concerning processing:

1. Running unrestricted batch jobs. This requirement refers to the execution of batch jobs that do not have any QoS guarantee to meet, such as long-running jobs that are not linked to a production service.
2. Running periodic batch jobs. Periodic workloads, such as daily jobs that retrieve the updated data from a public data source, must be regularly executed by the platform.
3. Running batch jobs with QoS restrictions. Tracking the job progress is a complex task that is limited to jobs that use a specific execution framework that supports it (e.g. Spark, Marathon, etc.). We consider in this requirement the guarantee that a given amount of CPU time is assigned to a running job in a given time frame.
4. Self-adapting elasticity. This requirement is strongly linked to requirement 3. The platform should provide enough resources to deal with new jobs and to ensure that jobs with QoS are properly executed.
5. Running parallel Spark jobs. The platform must support the execution of Spark jobs across several nodes in parallel, providing the right amount of resources to each job.

Considering these requirements, we focus on analyzing the available technologies for cloud orchestration, elasticity and cloud services for data analytics.

3.2.2 Cloud Orchestration

Cloud orchestration is the process needed to automate the entire lifecycle of a cloud application. It implies the deployment of all the computational resources, the installation and configuration of the different component parts of the application and their correct interconnection.

To describe cloud applications, the Topology and Orchestration Specification for Cloud Applications (TOSCA) [127] open standard has been defined by the OASIS consortium[1]. It defines the interoperable description of services and applications to be run on the cloud, including their components, relationships, dependencies, requirements, and capabilities; thereby enabling portability and automated management across cloud providers regardless of the underlying platform or infrastructure.

By using TOSCA to model the user's complex application architectures it is possible to obtain repeatable and deterministic deployments. Users can port their virtual infrastructures among cloud providers obtaining the same expected topology.

Several open source orchestration tools and services exist in the market, but most of them come with the limitation of only supporting their own Cloud Management Platforms (CMPs) because they are developed within those project ecosystems. As an example we can cite some of them, such as OpenStack Heat [138] and its YAML-based Domain Specific Language (DSL) called Heat Orchestration Template (HOT) [135], native to OpenStack [165]. OpenNebula [137] also provides its own JSON-based multi-tier Cloud application orchestration called OneFlow [133]. Eucalyptus [168] supports orchestration via its implementation of the Amazon Web Services (AWS) CloudFormation [5] web service.

[1]https://www.oasis-open.org/

In case of other general orchestration tools, we can find Cloudify [29], which provides TOSCA-based orchestration across different cloud providers. Unfortunately, Cloudify is not currently able to deploy on OpenNebula sites, one of the main CMPs used within current science clouds. Apache ARIA [58] is a very recent project, not mature enough and also without support for OpenNebula. Project CELAR [66] uses an old XML-based TOSCA version with SlipStream [164] as the orchestration layer (this project has no activity in the last years and SlipStream has the limitation of being open-core, thus not supporting commercial providers in the open-source version). CompatibleOne [183] provided orchestration capabilities based on the Open Cloud Computing Interface (OCCI). However, the project has not been active in the last years. OpenTOSCA[166] currently only supports OpenStack and the AWS EC2 [3] service.

Our previous work in the field is the Infrastructure Manager (IM) [15], a cloud orchestration runtime that deploys complex and customized virtual infrastructures on multiple back-ends. It supports the TOSCA Simple Profile in YAML version of the standard. It is compatible with a wide variety of Cloud back-ends, both on-premises CMP and public Cloud providers, thus making user applications cloud agnostic. In addition, it features DevOps capabilities, based on Ansible[2] to enable the installation and configuration of all the user required applications providing the user with a fully functional infrastructure.

3.2.3 Cloud-based Data analytics

In recent years, the Data Deluge [7] made it possible to enter an era in which distributed computing is now the new normal, paving the way for Big Data, a term coined for scenarios in which the amount of data (or the speed at which data is generated) can no longer be processed in a feasible time in a single computer. Google, being a large-scale data-oriented enterprise, faced the challenges that involved the processing of huge datasets and, in 2008 unveiled the MapReduce programming model [37], together with an associated implementation for processing large datasets. This was the seed that made possible the Apache

[2]https://www.ansible.com/

Hadoop project [59], an open-source software for reliable, scalable, distributed computing that ended up forming the kernel (as is the case of the Hadoop Distributed File System) of a huge ecosystem of tools aimed at solving Big Data problems. This is the case of Hive [60], a data warehouse software for querying and managing large datasets in a distributed storage or Pig [61], a platform for analyzing large datasets via a high-level language for expressing data analysis programs. Other platforms such as Spark [62], due to its speed related to in-memory processing, are also fundamental for many Big Data scenarios.

In addition, the trend towards lightweight virtualization allowed container technology to considerably evolve, exemplified by the recent advances in Linux containers (LXC) [105] and Docker [147], and the HPC-specific Singularity [94]. LXC enables to run multiple isolated processes in one host without the overhead caused by the hypervisor layer introduced by Virtual machines (VMs) in CPU, memory and storage [52], as if it was a whole new machine. Docker is oriented to applications, and the underlying idea is to run a single application that is isolated and with a tailored environment. Moreover, the ecosystem of tools around Docker has exploded in the last years, with contributions in many areas such as Continuous Integration/Continuous Delivery (CI/CD), application packaging and container orchestration tools. Indeed, there are many applications to manage the execution of containers across multiple hosts (e.g. Kubernetes [171] or Deis [38]) but one of the most advanced tools for computationally challenging problems is Apache Mesos [117], a software that abstracts CPU, memory, storage and other compute resources away from machines to enable fault-tolerant distributed systems to be built. Moreover, Mesos supports several frameworks suitable for resource-intensive computing, as is the case of Chronos [25], for the job fault-tolerant executions, and Marathon [109], for the execution of long-running services. Finally, Singularity is an alternative to Docker that has been developed in the HPC context. Its growing popularity is due to the ability to create containers that run in the user space, and are integrated with the underlying system by mapping the system user ids and important folders (such as home). In order to foster its usage, it is able to get images in Docker format, among others.

3.2.4 Elastic Clusters

Elasticity is the property of an infrastructure to dynamically adapt itself to the current or estimated workload. This is manifested in cloud infrastructures at several levels. In the lower level of on-premises clouds, elasticity represents the ability to dynamically power on and off the nodes of the underlying hardware in order to provision and relinquish physical computing hardware on which the virtualized infrastructure will run. At the level of IaaS (Infrastructure as a Service), these techniques should be integrated within the Cloud Management Frameworks (CMF) so that requests of virtual infrastructure deployment trigger, if necessary, the powering of physical nodes in order to accommodate the virtual infrastructure that will be executed on top of the physical infrastructure. Horizontal elasticity is the ability to dynamically deploy and terminate nodes within a virtual infrastructure according to a set of elasticity rules (scale in/scale out) and this is exemplified by services such as Auto-Scaling [4] for AWS or Heat/AutoScaling for OpenStack, to name a few.

In the literature, we can find several research works regarding horizontal elasticity in virtual clusters. In [110] and [111], the Nimbus toolkit is employed to implement a tool to create elastic sites, so that physical clusters based on a Local Resource Management System (LRMS) such as Torque are supplemented with computational resources provisioned from AWS according to different policies.

A widely used tool is StarCluster [169] which enables the creation of virtual clusters in AWS, that satisfy a user-defined list of required applications (Sun Grid Engine, OpenMPI, NFS, etc.). The Virtual Machines (VMs) are based on predefined Amazon Machine Images (AMI). In addition, a plugin named Elastic Load Balancer [170] is available to add and terminate new cluster nodes taking into account the number of jobs queued up at the LRMS. The main limitation of this plugin is that it requires a permanent connection to the cloud infrastructure from the StarCluster installation in the user's computer in order to deploy and terminate the VMs.

In the last years, horizontal elasticity has also been introduced in well-known Big Data frameworks. This is the case presented in [21], where the authors propose a system called BBQ, which is able to provide elasticity to Hadoop MapReduce. It works with AWS and needs a specific modified implementation of Hadoop to properly work with BBQ, thus, limiting the ability to choose a desired configuration for the users.

Our previous work in the field is Elastic Cloud Computing Cluster (EC3) [17], a tool that creates elastic virtual clusters from computational resources provisioned from IaaS clouds. These clusters scale out to a larger number of nodes on demand, up to a maximum size specified by the user. Whenever idle resources are detected, the clusters dynamically and automatically scale in, according to some simple policies, in order to minimise the costs in the case of using a public cloud provider.

To dynamically manage the clusters, EC3 relies on the CLUster Energy Saving (CLUES)[36] tool, an elasticity manager. CLUES has been already integrated in public and on-premises cloud environments in order to deploy/destroy VMs and it is able to automatically integrate the VMs in the LRMS according to the workload of the cluster.

Horizontal elasticity is appropriate when the problems solved are inherently parallel. In cases where the problems cannot benefit from an increase in the amount of resources, another elasticity strategy must be considered. Vertical elasticity is the ability to dynamically resize the resources of the nodes, such as the number of CPUs, the share of CPU or memory, according to a set of elasticity rules (scale up or scale down).

Most of the hypervisors and cloud IaaS support vertical elasticity. This is the case of OpenNebula and OpenStack, which offer functions to resize the memory or change the number of CPUs of stopped VMs. Nevertheless, dynamic resizing of VMs is not supported because it is necessary to act both at the level of the hypervisor and at the level of the VM's operating system.

There are techniques for providing vertical elasticity leveraging the CPU CAP (the maximum amount of CPU resources a VM can use) and the physical memory allocated by the hypervisor. This strategy only acts at the level of the hypervisor and, thus, it has a better approach than the strategy described above. This way, the internal configuration of the VM remains the same, but it is provided with a higher share of physical resources, so the virtual CPU can run faster or slower, and have more RAM mapped on physical RAM. For example, the work by Shen et al. [162] describes an approach named CloudScale to automate fine-grained elastic resource scaling for multi-tenant cloud computing infrastructures. Other examples of these techniques are described in [45]. It should be pointed that, in this example, the system needs access to the private network to connect with the worker nodes and root privileges for leveraging the CPU CAP and the memory RAM.

Vertical memory elasticity is interesting for some problems (e.g. when the consumed memory grows over time). For this purpose, the virtualization hypervisors provide two mechanisms: add or remove memory, also named hot memory plugging, and memory ballooning ([121], [51]). In any case, the allocation of resources with the aforementioned techniques affects all of the tasks running in the VM.

Providing vertical elasticity to guarantee QoS restrictions requires a more fine-grained approach than the techniques described above because it is necessary to resize the assigned resource for a specific job (not all the running jobs of the VM). For this purpose, it is needed to act at the level of processes of the operating system of the VM.

Nowadays, Docker containers are becoming the new platform for packaging, distribution and deployment of applications in Cloud Computing. Vertical elasticity in Docker containers has few works in the literature compared those available for VMs. The work by Al-Dhuraibi et al. [2] presents a mechanism that modifies the allocated resources (CPU time, vCPU cores and memory) of a Docker container according to the workload demand. This mechanism monitors the CPU time, CPU utilization, vCPUs and memory utilization to take the elasticity decisions and implements these decisions modifying the *cgroups* pseudofiles of the Docker container directly.

The job execution capabilities of the Mesos cluster are provided by their frameworks and, commonly, these jobs are encapsulated in Docker containers or Mesos native containers, which are processes of the VM. In case of Mesos, the level of processes of the operating system of the VM can be seen as the jobs of frameworks that support vertical elasticity (for example, Marathon).

Some techniques modify the assigned resources for Apache Mesos execution frameworks. One example of this approach is [185], a Mesos executing and monitoring framework called Makeflow is designed to adjust the number of vCPUs of a series of independent jobs according to their actual performance. The work [35] proposes a mechanism to provide both horizontal and vertical elasticity according to the share of CPU and memory used. This technique considers that the jobs do not store a persistent state and, thus, they can be easily restarted.

3.3 Architecture Design

The architecture of the infrastructure must enable the execution of a wide variety of workloads, that range from parallel to high-throughput jobs, including short jobs and big data workflows. This section provides information on the proposed architecture for the deployment and the automatic management of both horizontal and vertical elasticity at the level of the framework.

3.3.1 General Architecture

The proposed architecture is depicted in Figure 3.1. In the scenario, the administrator user is in charge of deploying the virtual infrastructure by using the EC3 client. EC3 interacts with the Infrastructure Manager (IM), requesting the needed resources considering the characteristics of the cluster together with its specific configuration. With these data, the IM interacts with the selected cloud provider, requests the VMs that compose the cluster and configures them. Notice that, by using IM, the cluster can be deployed on different on-premises and public clouds. In particular, at least the following providers can be used: OpenNebula,

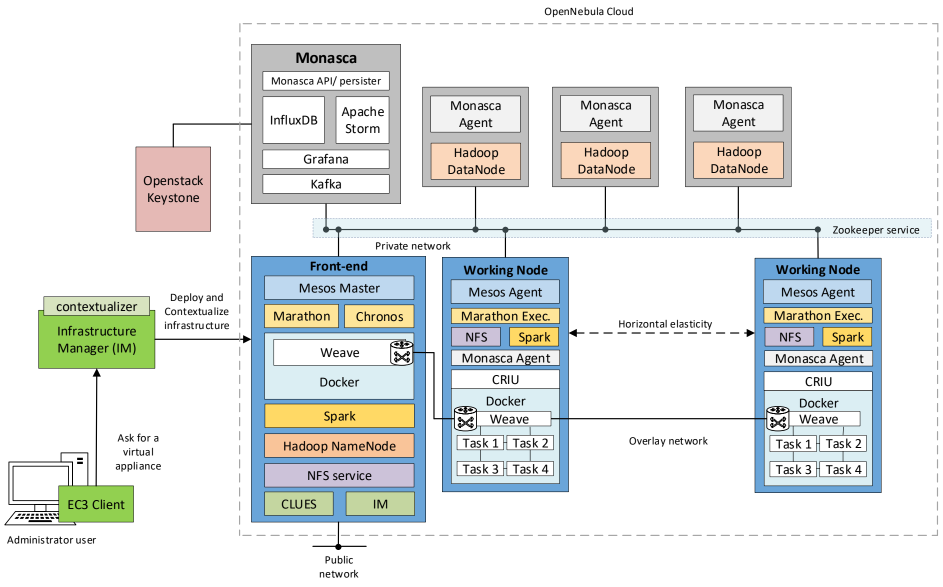


Figure 3.1: Architecture of the required infrastructure to perform data analytics.

OpenStack, Amazon Web Services, Google Cloud Platform [68] and Microsoft Azure [118].

The infrastructure has three main types of nodes:

- The front-end node, that is the master of the cluster. It contains the Mesos master instance together with the Marathon and Chronos frameworks, and also including Docker, Hadoop, Spark and NFS. The front-end also has an instance of the IM that, together with CLUES, is in charge of managing the elasticity of the cluster. This node is also in charge of offering an interface for end users of the infrastructure.

- The Monasca [141] nodes, that include the Openstack Monasca server instance, that also have Apache Kafka [54], Apache Storm [55], InfluxDB [80] and Grafana [96]; three Monasca agents also act as Hadoop datanodes. The architecture also includes a VM that provides the keystone [139] server needed by Monasca. It is not running inside the Monasca server to avoid excessive resource consumption.

- The working nodes, that are the elastic part of the infrastructure. These nodes are deployed on-demand when triggered by CLUES, that monitors the Mesos and Marathon queues and reactively provides the needed Mesos agents. These working nodes also contain Docker and CRIU [176], to have the ability to run jobs inside containers with checkpointing capabilities. NFS is employed to provide a shared file system across the nodes to manage the checkpointed containers. A Spark daemon is also running on each working node and all of them are monitored by Monasca.

Regarding networking, all the components of the infrastructure are interconnected by a private network. Moreover, a dedicated overlay network, managed by Weave [178], is created to interconnect the containers running on different hosts. The front-end is also connected to this overlay network, so that applications running in the containers can interact with the services using the overlay network. This guarantees a bi-directional communication on a wide range of ports without exposing the jobs to the Internet. The front-end of the cluster is the only component that can be accessed via a public network, even though the whole infrastructure has access to the Internet via NAT (Network Address Translation).

3.3.2 Vertical scaling

This section describes the architecture of the developed mechanism for providing vertical elasticity to the batch jobs with QoS restriction. As it is described in section 3.2, the developed mechanism aims to guarantee that the desired amount of CPU time is assigned to a running job in the given time frame of the targeted QoS. These jobs are embedded in Docker containers and are executed using the Marathon framework of Mesos. The architecture is composed of three main components: Launcher, Executor and Supervisor. This architecture is depicted in Figure 3.2, where the green dashed lines represent the interactions between components and the other services (Marathon, Monasca and Keystone).

Vertical elasticity is the ability to resize the assigned resources of a job in order to meet a targeted QoS. In this work, the mechanism varies the assigned share of CPU to the job. Resizing jobs in Marathon requires updating the job specification in the Marathon scheduler via its REST API. Once the job specification is changed, Marathon removes the older version of the job without preserving the execution state. Then, it runs the job with the new resource reservation. Furthermore, it should be pointed out that the new job execution can run on another working node.

Therefore, every time a job is resized its progress state is lost. To avoid this problem, this work uses CRIU [176], which is a project for the Linux operating system that allows to freeze a running application as a collection of files called checkpoint. Checkpointing allows users to stop and resume the job at the same execution point as it was when the checkpoint was made, even in another machine. As Marathon cannot freeze and resume the Docker container using CRIU, it is required that the developed mechanism manages the Docker container execution.

The Launcher is a command-line tool in charge of the submission of the job. Users run the Launcher specifying the job in JSON format with the QoS information, the parameters to connect and configure the Supervisor, and the credentials of the Marathon scheduler. The QoS information is composed of the number of seconds of CPU time that the mechanism should assign for completing the job, the over-progress percentage, and the time frame (in seconds) for executing the job. Users can configure, with the parameter called over-progress percentage, the *overprogress* threshold used by the Supervisor in Algorithm 2 for computing the job *performance state*.

First, the Launcher assigns to each job a unique identifier (UUID). If the Launcher submits to Marathon the job specification provided by the user, then the Marathon executor will manage the Docker container. As the mechanism must manage the Docker container for using the checkpointing feature, an additional component, the Executor, is required. To allow the Executor to manage the Docker container, the Launcher creates a new job specification based on the job specification provided by the user. Tasks in Mesos are isolated because they are executed embedded into Docker containers or Mesos native containers. Thus, the Marathon executor runs the new job (the Executor) created by the Launcher isolated by a Mesos native container. Once the new job specification is generated, the Launcher submits it via a REST API call to the Marathon scheduler. Finally, the Launcher sends a message with information about the job to the Supervisor. This information is formed by the job name, the job UUID, the maximum overprogress percentage, the number of seconds of CPU time that the mechanism should allocate for completing the job, and the time frame for executing the job in seconds.

The Executor performs several tasks. First, it prepares the worker node to enable the Docker container monitoring using a modified Docker plug-in of the Monasca Agent. Afterwards, it checks for the existence of a previous checkpoint of the job in the directory shared by all of the worker nodes. If the Executor does not find a checkpoint, then it starts the Docker container. Once the Docker container is running, the Executor notifies via a REST API call to the Supervisor. Then, it waits until the Docker execution is done or until capture the termination signal sent by the Marathon executor when a scaling decision is implemented by the Supervisor. If the Docker container ends its execution, the Executor cleans the worker node and the shared directory, and notifies the end of the job execution via a REST call to the Supervisor. If the Executor captures a termination signal, then it means that the Docker container will be resized. Thus, the Executor performs the checkpoint of the execution and stores it into the directory shared by all the worker nodes. Once the Marathon executor runs the Executor with the new allocated resources, the Executor resume the Docker container execution from the stored checkpoint.

The Supervisor is a REST service in charge of the decision making. When the Executor notifies the start of the job execution to the Supervisor, it begins to periodically monitor the job to decide if scaling up or down is needed. There are three possible job performance states: *overprogress*, *underprogress* or *ontime*. If the job state is *overprogress* or *underprogress*, then the Supervisor scales, respectively, down or up the assigned resources to the job. The Supervisor implements the scaling decision re-submitting the job specification (which is available on the Marathon scheduler) with the new resource reservation by a REST API call to the Marathon scheduler. The amount of share of CPU that is incremented or decremented is set at the startup of the Supervisor. For this work, empirical observation indicates that 0.4 offers good results. Once the new resource assignation arrives to the Marathon scheduler, it notifies the Marathon executor to send the termination signal to the old version of the job, which is captured by the Executor to create and store the checkpoint.

The Supervisor uses the Algorithm 2 to determine the job *performance state*. This algorithm has three input values: the *performance*, the *overprogress*, and *underprogress* thresholds. The *performance* is obtained using the Equation 3.2. The Equation 3.2 uses the CPU time consumed $cputime_{current}(t)$ and the expected CPU time consumed ($cputime_{desired}(t)$) on a certain time t (both expressed in seconds). The Supervisor estimates the CPU time consumed at certain time t, $cputime_{current}(t)$, requesting via the REST API the gathered information about the Docker container to the Monasca. The information obtained by the Supervisor from Monasca is composed of two metrics (*container.cpu.user_time* and *container.cpu.system_time*). These values correspond to the total user and system clock ticks consumed by the container in the node where it is running. These values are transformed into seconds, dividing them by the clock ticks per second constant of the system. The addition between these values is the $cputime_{current}(t)$. In addition, the Supervisor also estimates the CPU time that the job would have to consume at certain time t, $cputime_{desired}(t)$, by means of Equation 3.1.

$$cputime_{desired}(t) = \begin{cases} \frac{(t_{current} - t_{start}) * seconds_{job}}{seconds_{timeframe}} & \text{if } t_{current} \leq t_{start} + seconds_{timeframe} \\ seconds_{job} & \text{otherwise} \end{cases} \tag{3.1}$$

where $t_{current}$ is the current time in timestamp format, t_{start} is the start time of the execution in timestamp format (obtained when the Executors notifies the start of the execution), $seconds_{job}$ is the number of seconds of CPU time that the mechanism should allocate for completing the job (sent by the Launcher), and

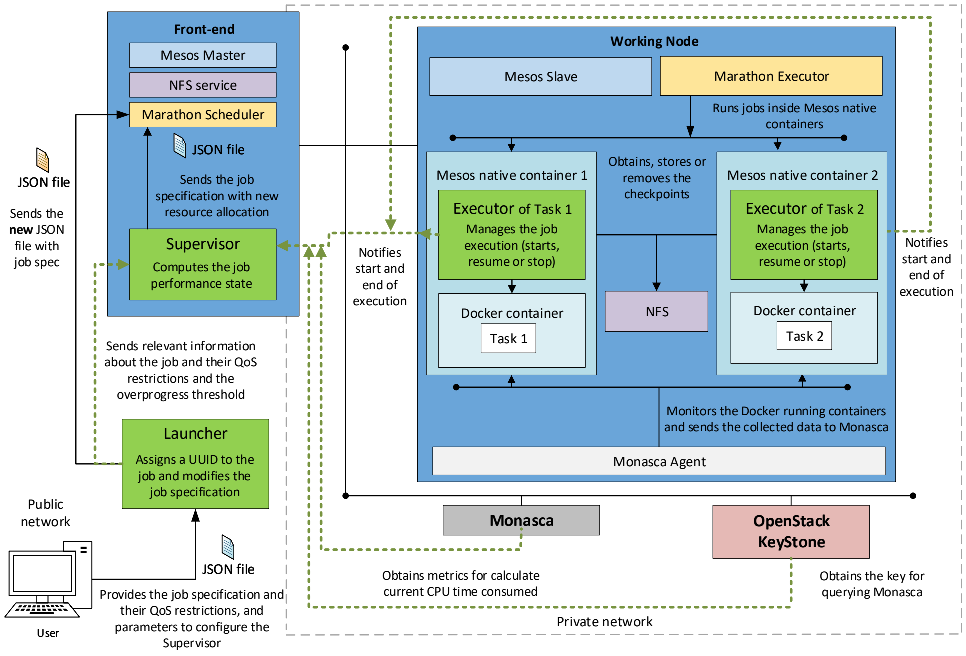

Front-end
Mesos Master
NFS service
Marathon Scheduler
JSON file
Sends the job specification with new resource allocation
Supervisor
Computes the job performance state
JSON file
Sends the new JSON file with job spec
Sends relevant information about the job and their QoS restrictions and the overprogress threshold
Launcher
Assigns a UUID to the job and modifies the job specification
Public network
JSON file
User
Provides the job specification and their QoS restrictions, and parameters to configure the Supervisor
Working Node
Mesos Slave
Marathon Executor
Runs jobs inside Mesos native containers
Mesos native container 1
Executor of Task 1
Manages the job execution (starts, resume or stop)
Docker container
Task 1
Obtains, stores or removes the checkpoints
NFS
Mesos native container 2
Executor of Task 2
Manages the job execution (starts, resume or stop)
Docker container
Task 2
Notifies start and end of execution
Notifies start and end of execution
Monitors the Docker running containers and sends the collected data to Monasca
Monasca Agent
Monasca
OpenStack KeyStone
Obtains metrics for calculate current CPU time consumed
Obtains the key for querying Monasca
Private network

Figure 3.2: Architecture for vertical scaling.

$seconds_{timeframe}$ is the available time interval to complete the execution of the job (sent by the Launcher).

The *underprogress* threshold is 10% by default, so the value used in Algorithm 2 is 0.9. The *overprogress* threshold is customizable by the user because this value is sent by the Launcher. The value used in Algorithm 2 is 1.0 plus the *overprogress* threshold provided by the user to the Launcher.

$$performance(t) = \frac{cputime_{current}(t)}{cputime_{desired}(t)} \tag{3.2}$$

Algorithm 2: Algorithm used by the Supervisor to obtain the job state.

```
Input : performance, threshold$_{overprogress}$ and threshold$_{underprogress}$
Result: state
begin
    if $performance > threshold_{overprogress}$ :
        state = overprogress ;        /* Decrease resource reservation */
    else if $performance < threshold_{underprogress}$ :
        state = underprogress ;       /* Increase resource reservation */
    else:
        state = ontime ;                               /* Nothing to do */
end
```

3.3.3 Description of the components

As shown in the figures above, the proposed architecture is composed of several components. Most of them are well-known software packages and frameworks while others are software tools developed by our research group.

All these components require different configuration files and installation steps, which are customized for the different underlying operating systems supported by the VMs. Therefore, this involves a large number of configuration files. Therefore, to ease the deployment and installation process, Ansible roles and playbooks were used for the sake of maintainability and high reusability. These are configuration files that describe the process of installation, configuration and integration with the selected architecture. Furthermore, to keep the component recipes as generic as possible (to further ease maintenance and reuse), such recipes were coded according to the following principles:

- The production Ansible roles should be stored in public repositories such as GitHub (all the recipes should be open-source and available to the public).
- The variables used inside the Ansible roles should be defined in a way that they can be set up at deployment time. This way, updates can be automatically applied to the roles, so users do not keep outdated

configurations on their systems (except those that could have been explicitly modified by the user).

- The Ansible roles should be added to Ansible Galaxy[3] a public repository[4] of roles so that others can reuse them, thus greatly simplifying the roles definition and composition.
- In addition to the Ansible roles, there must exist high-level installation recipes (also stored in GitHub) for the Infrastructure Manager [5] and EC3[6] that contain all the configuration steps for deploying complete virtual infrastructures.
- The recipes should support different platforms (currently Ubuntu 14, Ubuntu 16 and CentOS 7).

Table 3.1 includes the components that were identified and configured using Ansible roles for the creation of the virtual infrastructure previously described.

The EC3 tool is in charge of deploying the fully configured cluster by using these Ansible recipes. Thus, it provides the required infrastructure with the necessary services to deploy applications by using only a command, that automatically configures and contextualizes all the VMs that compose the cluster infrastructure. As stated above, all the sources of the recipes used to configure the cluster are stored in GitHub[7]. Moreover, for the sake of reproducibility of the results of this contribution, a Docker container image with the EC3 client installed has been released[8].

[3]https://galaxy.ansible.com/
[4]https://galaxy.ansible.com/grycap
[5]https://github.com/grycap/im
[6]https://github.com/grycap/ec3
[7]https://github.com/grycap
[8]https://hub.docker.com/r/eubrabigsea/ec3client/

Node Type	Component	Version	Requirements	Comments
Front/wn	Apache Mesos	1.4.1	All	Main framework, including Mesos-DNS.
Front	Marathon	1.4.3	1,3,4	For deploying long-term and high-availability services on Mesos.
Front	Chronos	2.1.0	1,2,4	Cron-like job scheduler for Mesos.
Front/wn	Spark	1.6.3	1,5	Execution of Spark code through spark-submit from the Front/End or external resources.
Datanode	Hadoop	2.6	All	HDFS storage backend.
Front/Monasca	Zookeeper	3.4.8-1	All	High availability of Mesos and Marathon.
Front/Monasca/wn	Docker	17.05.0-ce	1-4	Containerization of applications launched through Marathon and Chronos.
Front	Docker registry	2	1-4	Mirroring and caching the Docker images to speed-up distribution along the cluster.
Front	CLUES	2.1.0b	4	Manages horizontal elasticity.
Front	Infrastructure Manager	1.6.6	4	Manages the configuration of the internal nodes.
Front/wn	CRIU	2.6	4	Performs container checkpointing
Monasca/wn	OpenStack Monasca	1.6.1	3,4	Monitoring system, including the Docker plugin. Monasca agent installed in WN.
Keystone	OpenStack Keystone	13.0.0	3,4	A service that provides API client authentication, service discovery, and distributed multi-tenant authorization.
Monasca	Apache Kafka + Storm + Grafana + InfluxDB	2.12, 1.0.2, 4.0.1 & 0.9.5	3,4	Components for the Monasca Server
Front/wn	Weave	2.1.3	All	Provides the overlay network to the container infrastructure.
Front/wn	Vertical Elasticity	1.0	3,4	Proactive vertical elasticity mechanism for Marathon using Monasca

Table 3.1: Components of the Mesos cluster architecture.

3.4 Results

The experiments performed in this paper address three main aspects: i) the efficiency of the deployment of a medium-sized virtual infrastructure; ii) the overhead of the horizontal elasticity compared to the execution with resources deployed upfront; and iii) the ability of the vertical elasticity to reconfigure the reservation of resources for a running job to meet a specific QoS.

For the first case, this work measures the deployment time of a cluster with 50 nodes and 100 processing cores. This time includes the deployment of the VMs, the download and installation of the software dependencies, and the configuration of all the services. This process is entirely automatic.

The second case covers the execution of a set of 20 parallel Spark jobs at different time intervals. Initially, there are no processing resources except the front-end node of the cluster and the system automatically starts and reconfigures them as required. The execution intervals have been defined in a way that the job queue is flushed completely, triggering the suspending of idle nodes.

The third case deals with the Quality of Service guarantees for a Marathon job executed in the cluster. The cluster is busy with other jobs competing for the resources, so we aim to assign the required CPU time to meet the targeted QoS. The developed tool[9] dynamically readjusts the share of CPU to reduce the assigned resources to a job if it is over progressing and increases the assigned resources if the job progress is lower than expected. We measure the job performance as the amount of CPU time consumed by a job according the given time frame of the QoS agreed.

For the three experiments, the physical infrastructure used is composed by two type of nodes. The first type of node has two Intel(R) Xeon(R) CPU E5-2683 v3 2.00GHz (14 cores) processors, 64 GB of memory RAM, 240 GB of Solid State Disk, two 1 GB Ethernet network adapter and one 10 GB Ethernet network adapter. The second type of node has two Intel(R) Xeon(R) CPU E5-2660 v4

[9]https://github.com/eubr-bigsea/vertical_elasticity

2.00GHz (14 cores) processors, 128 GB of memory RAM, 250 GB of Solid State Disk, two 1 GB Ethernet network adapter and one 10 GB Ethernet network adapter. The Storage Area Network is a Dell Equallogic PS4210 with 16 TB availables. This hardware is managed by the OpenNebula Cloud Management Framework and the KVM hypervisor.

3.4.1 Deployment metrics

The deployment of the data analytics cluster is done automatically through EC3. The following metrics have been evaluated to show the performance and the impact of the usage of the elasticity on the user experience when interacting with the virtual cluster:

- Deployment of the static components (not managed by CLUES): Mesos master (front-end of the cluster, with 4 CPUs and 16Gb of RAM), Monasca master (with 2 CPUs and 8Gb of memory RAM) and 3 HDFS datanodes (with 2 CPUs and 2Gb of memory RAM). This set of nodes are deployed by the EC3 client in the virtual cluster creation step.

- Deployment of the first working node (with 1 CPU and 2Gb of RAM), including the creation of the *golden image* that will be used to speed up the deployment of the rest of the elastic nodes. This feature consists on creating a VMI from the first working node correctly configured and integrated in the system. Thus, this VMI is used for the next working nodes deployed in the system, accelerating their configuration.

- Deployment of a second working node using the created *golden image* to measure the impact of the usage of *golden images* in subsequent nodes.

- Concurrent deployment of multiple concurrent working nodes (10, 20, 35 and 50 nodes). It will show how the system will react when a large set of nodes are requested.

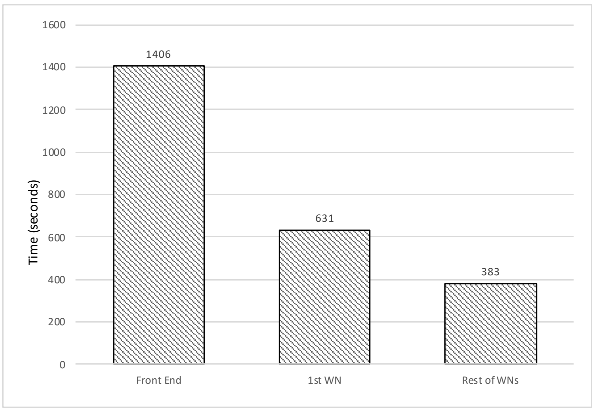


Figure 3.3: Deployment time of the different node types.

These measures give information about the overheads on the deployment of the full operational cluster and their reconfiguration, which serves as basis for defining elasticity mechanisms and suitable applications.

The first step is the deployment of the front-end and the set of static nodes (front + monasca node + 3 datanodes). The average deployment time for the complete initial infrastructure is 23min 26sec (1406 s). Figure 3.3 shows the comparative deployment time of the initial infrastructure and the working nodes. The deployment of a node without golden image plus the creation of the golden image when the node has been configured takes an average time of 10min 31sec. (631 s) whereas the deployment of nodes with golden images takes an average time of 6min 38 sec. (383 s). Clearly, nodes deployed from a *Virtual Machine Golden Image*, created on the fly, show a smaller configuration time.

Finally, for testing the scalability of the system, we present the deployment times for a concurrent deployment of multiple nodes (10, 20, 35 and 50 nodes). Table 3.2 depicts the results of each test, where golden images have been used to accelerate the deployments. The configuration system has been improved in the frame of the EUBra-BIGSEA[10] project to deal with the bottlenecks that appear when a large number of nodes are simultaneously configured. In the original approach a single

[10]http://www.eubra-bigsea.eu/

Number of nodes	**10**	**20**	**35**	**50**
Average time per node (sec.)	513.647	560.349	666.997	900.841
Total time (sec.)	520.472	592.602	751.997	1052.883

Table 3.2: Deployment time for different quantities of nodes (deployed simultaneously).

VM is selected as the "master". Then, Ansible is installed in this VM, which configures all the VMs in parallel. In this new approach (suitable for a large number of simultaneous VM deployments), Ansible is installed in all the VMs and each one configures itself in parallel. This approach increases the latency but reaches higher scalability, being successfully demonstrated in more than 100 machines.

Figure 3.4 shows the latency time (in seconds) from the request of the deployment of 50 simultaneous nodes in the cluster to the actual provisioning of the resources. The graph shows the number of machines deployed at each timestep. The figure shows that most of the nodes (42) are fully configured in less than 980 seconds. The rest of the nodes take a bit more time (72 seconds). It is only 7.3% more than the first groups of nodes. This delay is produced by different bottlenecks of the cloud platform (mainly network) when a large number of nodes are configured in parallel.

3.4.2 Horizontal Elasticity

The second case consisted of submitting 20 parallel data analytics jobs (implemented in Spark) to an infrastructure that initially had only two nodes started (2 vCPUs and 4 GB RAM each). These jobs were submitted at different time frames as shown in Table 3.3. The infrastructure had to detect the registration of a Spark framework, realize that there are not enough resources and deploy an additional node when a job remains queued longer than a given threshold (5 seconds in the experiment), with a cooling time (waiting time to perform a new action) of 5 minutes. Jobs were prepared to run for approximately 11 minutes and were able to use up to 4 cores each and request 0.5GB of memory

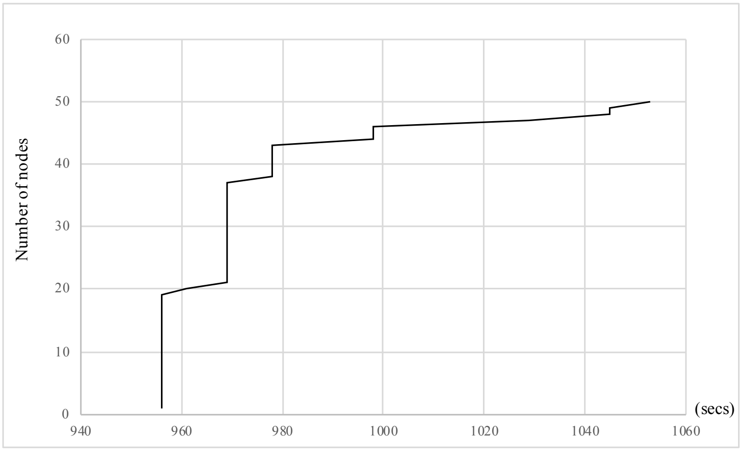


Figure 3.4: Deployment time for 50 simultaneous nodes.

Job	Submit	Job	Submit	Job	Submit	Job	Submit
1	0:00:30	6	0:35:54	11	0:55:14	16	1:31:14
2	0:01:15	7	0:52:34	12	1:28:34	17	1:31:54
3	0:01:53	8	0:53:14	13	1:29:14	18	2:05:14
4	0:18:34	9	0:53:54	14	1:29:54	19	2:21:54
5	0:19:14	10	0:54:34	15	1:30:34	20	2:38:34

Table 3.3: Scheduling of the jobs to be executed.

RAM. It is important to state that even if the job requests for 4 cores and Mesos offers it 2 cores, the job would anyway start. If there are enough resources (4 free cores), the job will take them all.

During the execution of the jobs, we measured the timestamp at the submission, execution start and execution end. We also registered all the changes in the status of the nodes, which could be OFF (not deployed), RESTART (being restarted), IDLE (powered-on and without jobs allocated), USED (executing jobs) or SUSPEND (being suspended). Figure 3.5 shows the results of the evolution of jobs and Figure 3.6 shows the status of the nodes along time.

Figure 3.5 shows the number of jobs (vertical axis) along time (horizontal axis) for five metrics. Submitted, Started and Ended lines denote the accumulated number of jobs that have been submitted, have actually started and have been completed over time. The lines Queued and Running denote the number of jobs that are queued or concurrently running at a given time. The execution profile has been defined to ensure that there are peaks of workload that require starting up new VMs and idle periods long enough to trigger the suspension mode for the VMs. This is used to analyze the behaviour system when adjusting the infrastructure. More details are provided in Figure 3.6.

As depicted in Figure 3.5, the length of the queue does not grow above one job. The delay between the submission and the start of a job (the difference in the horizontal axis between submitted and started lines) is negligible. It is important to remark that this overhead relates mainly to the time required for the VMs to change from suspended to running, as the VMs are suspended on disk rather than destroyed. It should be pointed out that the execution time for the jobs varies according to the resources available at the executing time.

Figure 3.6 shows how the working nodes are started and suspended on demand. The figure shows the number of working nodes (vertical axis) that are in each one of the five possible status (described at the beginning of this section) along time (horizontal axis). It is important to outline that transitions are very short, and the submission pattern of the execution enables emptying the queues and triggering the suspension of idle resources. Moreover, the default amount idle time to switch off a node in CLUES was used (20 minutes) but this value can be modified by user depending on the requirements.

3.4.3 Vertical elasticity

Running batch jobs that need to deal with QoS restrictions on infrastructures with a significant amount of free resources is not a complex task. For this type of infrastructures, it is very easy to accomplish the QoS restrictions because the scheduler only has to assign the maximum resources to all running applications. Thus, vertical elasticity makes sense in congested infrastructures. The QoS

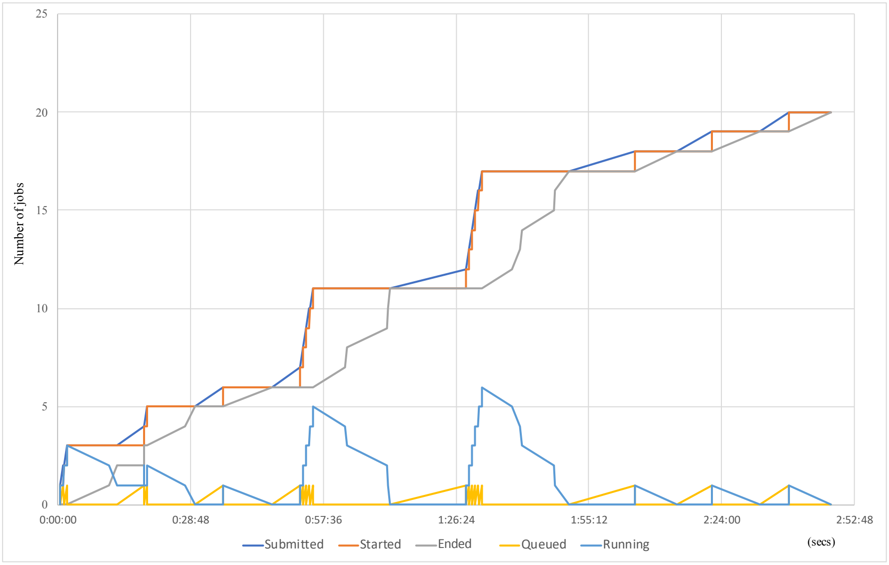


Figure 3.5: Jobs queued vs jobs running in the platform during the experiment and accumulated list of jobs (suspend mode).

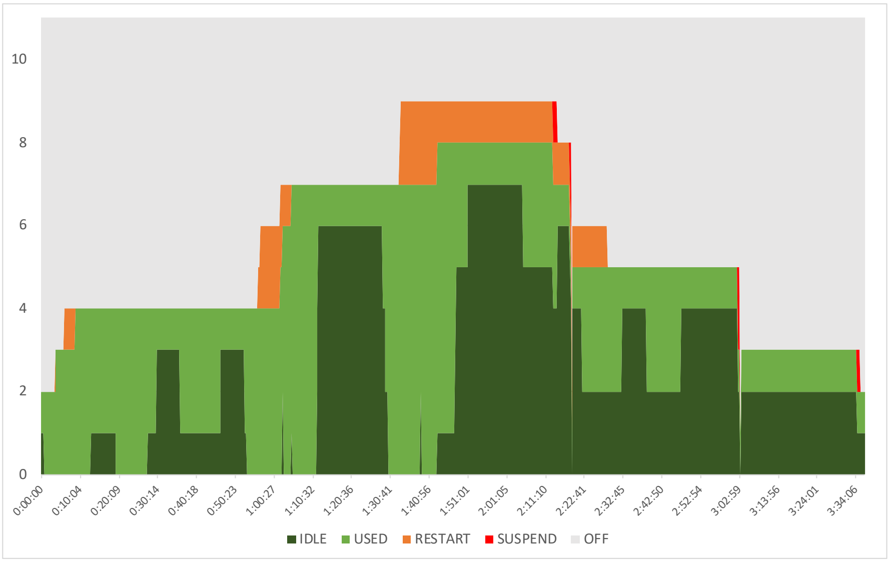


Figure 3.6: Evolution of the state of the cluster nodes, started and suspended on demand.

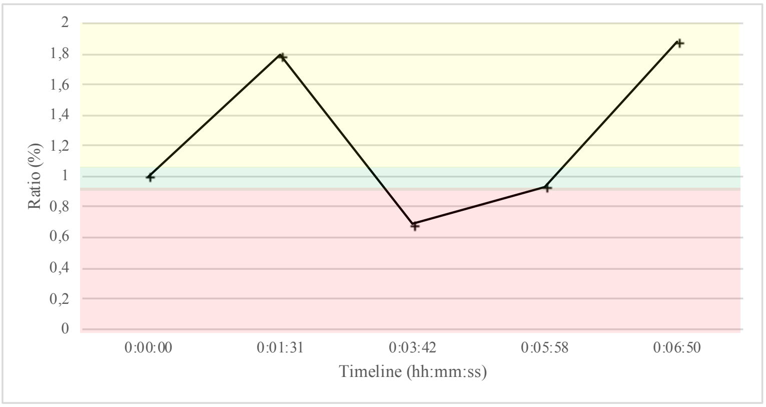


Figure 3.7: *Performance* during the experiment.

restrictions for this type of jobs are defined as the allocation of a minimum CPU time to the job during a given time interval.

The job used for checking the vertical elasticity capabilities of the implemented architecture takes 480 seconds (one CPU) and 360 seconds (two CPUs) to be completed. The number of the seconds (CPU time) that the mechanism must allocate is set to 400 seconds, which is a value between the completion time when using one and two CPUs. If a node has more amount of CPU share available than the job needs, the job uses all amount of CPU share. The QoS restriction on this test is set to the same completion time than when the job is executed using one CPU. Indeed, the developed mechanism for providing vertical elasticity is designed to make all jobs with QoS restrictions of a congested infrastructure to accomplish their agreed QoS. The maximum *overprogress* threshold is 10%, which means the job enters to *overprogress* state when its performance is 10% better than the required for meeting the targeted QoS.

The test has special interest because the job enters into the three possible states (described in section 3.3.2) during the execution. In addition, the experiment ends in 393 seconds, demonstrating that the system can scale up and down to guarantee a minimum CPU time. The execution trace can be observed in Figures 3.7 and 3.8.

Figure 3.7 shows the *performance* value (obtained using 3.2) and the *performance state* (obtained using 2) along the experimentation. The Y-axis and X-axis of the Figure 3.7 represent, respectively, the *performance* and the time when the sample was collected. The Y-axis coloured ranges denote the *performance state*: red, green and yellow represent, respectively, *underprogress*, *ontime* and *overprogress*.

Figure 3.8 shows a comparison between the consumed and the desired CPU time consumed, and the assigned share of CPU at each sample. The X-axis represent the time when the sample was collected. The desired CPU time is the amount of time that the developed mechanism estimates that should be consumed at the moment of which the samples are collected. The left Y-axis represents the amount of CPU time consumed in seconds. The right Y-axis represents the share of CPU assigned to the job during the experiment. If the working node where Marathon executes the job has 1 CPU and the user assigns 1.0 CPU share to the job, it means that the job has the 100% of the CPU share of one CPU. Thus, if the working node has 3 CPU and the CPU share assigned to the job is 2.5 CPUs, the job has reserved all of the CPU share of two CPUs and the 50% of the CPU share of the remaining CPU.

It should be noted that instants when the samples were collected are the same at both figures. The first sample corresponds to the start of the job. When the second sample is collected, it can be observed in Figure 3.7 that the job *performance state* is *overprogress*. In Figure 3.8 it can be observed at the second sample that the difference between the current time of CPU consumed and the desired time of CPU consumed is big enough to make the performance above the *overprogress* threshold. Thus, the Supervisor decrements the assigned share of CPU, which can be seen in the dashed line at the same figure. Then, the Executor performs a checkpoint as soon as it realises that the Supervisor decreased the job assigned resources.

As it can be seen in Figure 3.8, the current CPU time consumed is lower than the desired CPU time consumed in the next two samples but, the job's states are not equal. In case of the third sample, the *performance state* is *underprogress* (the

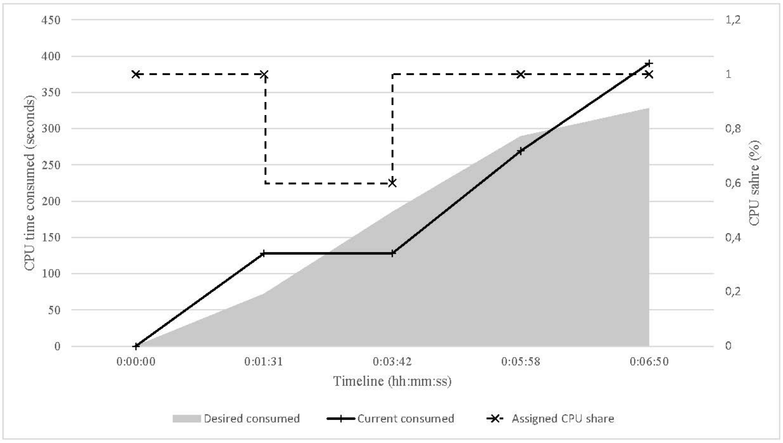


Figure 3.8: The dashed line represents the CPU share assignation during the experiment. The rest of the figure is a comparison between the CPU time consumed and the CPU time (desired) that the job is expected to consume at the time when the sample was collected.

performance value is 69%) so the Supervisor will do an increment of the assigned share CPU and the Executor will create a checkpoint.

Regarding the job performance at the fourth sample, the CPU time consumed is lower than the required CPU time calculated using Eq. 3.1. Due to the *underprogress* threshold is set to 90% and the *performance* value (calculated using the equation 3.2) is 93%, the *performance state* is *ontime*. For this reason, the job does not require to be resized.

The last sample corresponds with the end of the execution. Figures 3.7 and 3.8 show that the application terminates fulfilling the quality of service agreed with 87 seconds of margin.

We measured that the time from the start of the checkpointing to when it is stored in the shared directory (NFS) ranges from 30 to 60 seconds. After several tests, we estimate that executing one checkpointing and restart of the job execution increases the execution time in 6 seconds. Thus, even though the time of checkpointing in NFS is considerable, the downtime of the job execution is negligible. This is because the container continues to be executed while the checkpoint is created, similarly to virtual machine live migration techniques. Indeed, only when the checkpoint is completed the container is stopped and, then, it is immediately rescheduled in a new machine. Therefore, the time required for loading the checkpoint and starting the process is negligible.

It is complicated to estimate the overhead caused by the checkpoint mechanism in the experiment. The duration of the experiment was 393 seconds. The average duration of the job execution with 1 and 2 CPU takes, respectively, 480 and 360 seconds. In this experiment, the mechanism performed two checkpoints, so the overhead of checkpointing is 12 seconds. Thus, the overhead of the mechanism will be lower than 33 seconds and higher than 12 seconds.

3.5 Discussion

This section compares the proposed tools and solution exposed in this work with the already available solutions that can be found in the literature regarding the execution of time critical applications. Concerning cloud orchestration, our analysis of the state of the art revealed that there is no general orchestration tool that enables the deployment of cloud applications in several on-premises and public IaaS deployments using the standard TOSCA specification. Most of them only provide access to a very limited list of cloud providers. Our proposed cloud orchestration solution, the IM tool, supports the TOSCA standard and a big number of public and federated Cloud providers and on-premises CMPs, making the application cloud agnostic. The IM automates the Virtual Machine Image (VMI) selection, deployment, configuration, software installation, monitoring and update of virtual infrastructures.

The deployment of Big Data frameworks such as Mesos or Kubernetes requires an underlying distributed computing and storage infrastructure, that can be provisioned from on-premises clouds, public clouds or even from bare metal. However, there are several limitations that hinder the adoption of these frameworks, especially by Data Scientists that may be well versed in using the frameworks themselves but not specifically on efficiently deploying and scaling them. The framework presented in this paper in combination with the EC3 tool, considerably simplifies the deployment of these Big Data frameworks. EC3 allows to automatically deploy these Big Data frameworks with a single command, and without the need of user interaction.

Most of the already available solutions to automatically deploy clusters provide a virtual cluster with a fixed number of nodes, other solutions are oriented to a specific LRMS or they are tied to Amazon EC2 and, therefore, cannot provision nodes from other public Cloud providers, or even on-premises Cloud deployments (e.g. based on OpenNebula, OpenStack, etc.). The vertical elasticity mechanism presented in this work in combination with EC3 and the IM tools allows the user to deploy virtual clusters offering at the same time both horizontal and vertical auto-scaling capabilities. In addition, the contribution of the presented mechanism to the vertical elasticity capabilities (i.e. executing data analytics jobs embedded in Docker containers using Marathon involving common applications with QoS restrictions) was not found in the literature.

3.6 Conclusions

This paper has presented a software architecture and a set of open-source tools and configuration recipes for deploying a virtual self-managed cluster which offers horizontal (in and out) and vertical (up and down) scalability. Moreover a series of plugins have been developed to offer quality of service capabilities inside the cluster.

Regarding the technical requirements identified at the beginning of the paper, the test cases defined and the results exposed, it can be concluded that all the requirements proposed were fulfilled by the architecture presented. Running unrestricted batch jobs is one of the basic functionalities offered by the standard cluster configuration. In all the test cases it is demonstrated how the cluster admits different types of jobs and executes them without issues.

For running periodic batch jobs, the cluster must be prepared to accept a set of jobs defined to be executed in an specific time. In section 3.4.2 a batch of jobs are programmed to be launched and the cluster executes them by adjusting the resources available. This demonstrates that the defined architecture is not only able to process this kind of scheduled jobs, it is also able to self adapt horizontally depending on the workload. Moreover, the jobs presented in section 3.4.2 are a set of Spark jobs that were executed in parallel thus complying with the last requirement presented which required to execute Spark jobs in parallel.

In addition, the QoS restrictions and the vertical elasticity were tested in section 3.4.3. The execution of batch jobs with QoS restrictions by adjusting the share of CPU assigned is done thanks to a set of plugins developed and deployed automatically in combination with the frameworks available in the architecture presented.

Moreover, and as an extra step towards reusability and community usage, all the code developed for this project is publicly available and any user with access to one of the supported cloud providers (which include the most popular ones) can deploy an elastic cluster and tweak the configuration to fit the needs.

Future work includes testing the cluster with bigger setups, such as several hundred nodes and thousands of jobs during long periods of time but unfortunately and due to all the test being done in real infrastructures with shared resources and real users, the test cases have to be limited.

Chapter 4

A Cloud Architecture for the Execution of Medical Imaging Biomarkers

Published as:
Sergio López-Huguet et al. "A Cloud Architecture for the Execution of Medical Imaging Biomarkers". In: Computational Science – ICCS 2019*. Ed. by João M F Rodrigues et al. Cham: Springer International Publishing, 2019, pp. 130–144.* ISBN*: 978-3-030-22744-9*

4.1 Introduction

Traditionally, medical images have been analysed qualitatively. This type of analysis relies on the experience and knowledge of specialised radiologists in charge of carrying out the report. This entails a high temporal and economic cost. The rise of computer image analysis techniques and the improvement of computer systems lead to the advent of quantitative analysis. Contrary to qualitative analysis, the quantitative analysis aims to measure different characteristics of a medical image (for example, the size, texture, or function of a tissue or organ and the evolution of these features in time) to provide radiologists and physicians with additional, objective information as a diagnostic aid. In turn, the quantitative analysis requires image acquisitions with the highest possible quality to ensure the accuracy of the measurements.

An imaging biomarker is a characteristic extracted from medical images, regardless of the acquisition modality. These characteristics must be measured objectively and should depict changes caused by pathologies, biological processes or surgical interventions [49] [112]. The availability of large population sets of medical images, the increase of their quality and the access to affordable intensive computing resources has enabled the rapid extraction of a huge amount of imaging biomarkers from medical images. This process allows to transform medical images into mineable data and to analyze the extracted data for decision support. This practice, known as radiomics, provides information that cannot be visually assessed by qualitative radiological reading and reflects underlying pathophysiology. This methodology is designed to be applied at a population level to extract relationships with the clinical endpoints of the disease that can be directed to manage the disease of an individual patient.

The execution of medical image processing tasks, such as biomarkers, is a process that sometimes requires high-performance computing infrastructures and, in some cases, specific hardware (GPUs) that is not available in most medical institutions. Cloud service providers make it possible to access specific and powerful hardware that fits the needs of the workload [115]. Another interesting advantage of the Cloud platforms is the capability of fitting the infrastructure capacity to the dynamic workload, thus improving cost contention. The seamless transition from local image processing and data analytic development environments to cloud-based production-quality processing services implies a Continuous Integration and Deployment DevOps problem that is not properly addressed in current platforms.

4.1.1 Motivation and Objectives

The objective of this work is to design and implement a cloud-based platform that could address the needs of developing and exploiting medical image processing tools. In this sense, the work focuses on the development of an architecture focusing on the following principles:

- Agnostic to the platform, so the same solution can be deployed on different public and on-premise cloud offerings, adapting to the different needs and requirements of the users and avoiding lock-in.
- Capable of integrating High-performance computing and storage back-ends to deal with the processing of massive sets of medical images.
- Seamlessly integrating development, pre-processing, validation and production from the same platform and automatically.
- Open, reusable, extendable, secure and traceable platform.

4.1.2 Requirements

A requirement elicitation and analysis process was performed, leading to the identification of 13 requirements, classified into 8 mandatory requirements, 2 recommendable requirements and 3 desirable requirements. The requirements are described in Tables 4.1, 4.2 and 4.3.

4.2 State of the art

Since the appearance of Cloud services, a large number of applications have been adapted to facilitate the access of the application users to Cloud infrastructures. In the field of biomedicine we find examples in [78, 159]. On the other hand, there are works that offer pre-configured platforms with a large number of tools for bioinformatic analysis. An example is the Galaxy Project[64], a web platform that can be deployed on public and on-premises Cloud offerings (e.g. using CloudMan[30] for Amazon EC2 [3] and OpenStack [165]). Another example is

RiD	Name	Description	Level
RI1	Resource provisioning	Resources should be automatically configured in the deployment. Provisioning should be performed with minimal intervention by system administrators and should be able to work in multiple IaaS platforms.	R
RI2	Resource isolation	Jobs should run on the system at the maximum level of isolation. Workload may have different and even incompatible software dependencies, and the failure of the execution of a job should not affect the rest of the executions.	M
RI3	Resource scalability	Virtual infrastructures should be automatically reconfigured when adding or removing nodes (limited by a minimum and maximum number of nodes for each type of resource). Elasticity could be triggered externally.	R
RI4	Manag. of Releases	Software should be easy to update and releases should be easy to deploy. This implies automation, minimal customer intervention, progressive rollouts, roll backs, and version freezing.	M
RI5	User Authentication	Users should be able to log-in the system using ad-hoc credentials or an external Identity Provider (IDP) such as Google or Microsoft LiveID.	D
RI6	User Authorisation	Access to the services should be granted only to authorised users. This implies access to data, services and resources. Only special users would be able to access resources.	D
RI7	High availability (HA)	Services must be deployed in HA to guarantee Quality of Service.	M

Table 4.1: Requirements of the Infrastructure (**M**andatory, **R**ecommended, **D**esirable)

RiD	Name	Description	Level
RE1	Batch execution	The system should run batch jobs. A job will comprise a set of files, software dependencies, hardware requirements, execution arguments, input and output sandbox, job type, memory and CPU requirements.	M
RE2	Workflow execution	A job may include several linked steps that need to be executed according to a data flow. The workflow will imply the automatic execution of the different stages as dependencies are solved.	M
RE3	Job customization	Linked to requirements RI2 and RI4, this requirement poses the need of jobs to run on a customizable environment requiring special hardware, specific software configuration, operating system, and licenses.	M
RE4	Execution triggers	Jobs could also be initiated by means of events. Uploading a file or messages in a queue can spawn the execution of jobs. These reactive jobs will be defined through rules.	D
RE5	Efficient Execution	Jobs should be efficiently executed in the platform. This performance is defined at two levels: a) minimum overhead with respect to the execution on an equivalent pre-installed physical node; b) capability of integrating high-performance resources as GPUs and multicore CPUs.	M

Table 4.2: Requirements for Job Execution.

RiD	Name	Description	Level
RD1	POSIX access	Jobs expect to find the data to be processed in a POSIX file system in a specific directory route.	M
RD2	ACLs access	Storage access authorization based on a coarse granularity (access granted/denied for both read & write).	M
RD3	Provenance and traceability	Traceability for the derived data is key to bound to the GDPR regulations (e.g. a trained model should be invalidated if the permissions for any part of the data used in the training is revoked).	R

Table 4.3: Requirements with respect to the Data.

Cloud BioLinux[28], a project that offers a series of preconfigured virtual machine images for Amazon EC2, VirtualBox and Eucalyptus[48]. Finally, the solution proposed in [120] is specially designed for medical image analysis using ImageJ [79] in an on-premise infrastructure using Eucalyptus.

Before taking a decision on the architecture, an analysis has been done at three levels: Container technologies, Resource Managers and Job Scheduling.

Containers are a set of methodologies and technologies that aim at isolating execution environments at the level of processes, network namespaces, disk areas and resource limitations. Containers are isolated with respect to: the host filesystem (as they can only see a limited section of it, using techniques such as chroot or FreeBSD Jails), the processes running on the host (only the processes derived executed within the container are visible,for example, using namespaces), and the resources the container can use (as the processes in a container can be bound to a CPU, memory or I/O share, for example, using cgroups).

Containers are used in application delivery, isolation and light encapsulation of resources in the same tenant, execution of processes with incompatible dependencies and improved system administration. Three of the most prominent technologies in the market supporting Containers are Docker [147], Linux Containers (LXC) [105] and Singularity [94]. Docker has reached the maximum popularity for application delivery due to its convenient and rich ecosystem of tools. However, Docker containers run under the root user space and do not provide multi-tenancy. On the other side, Singularity run containers on the user-space, but access to specific devices is complex. LxC/D is better in terms of isolation but have limited support (e.g. LxD only works in ubuntu [106]). The solution for container isolation selected will be Docker on top of isolated virtual machines.

Resources should be provisioned and allocated for deploying containers. Resource Management Systems (RMSs) deal with the principles of managing a pool of resources and splitting them across different workloads. RMSs manage the resources of physical and virtual machines reserving a fraction of them for a specific workload. RMSs deal with different functionalities, such as: Resource discovery, Resource Monitoring, Resource allocation and release and Coordination with Job Schedulers. We identify 3 technologies to orchestrate resources are: Kubernetes [171], Mesos [117] and EC3 [15][17].

Finally, Job schedulers manage the remote execution of a job on the resources provided by the RMS. Job Schedulers retrieve job specifications from different interfaces, run them on the remote nodes using a dedicated input and output sandbox for the job, monitor its status and retrieve the results.

Job Schedulers may provide other features, such as fault tolerance, complex jobs (bag of tasks, parallel or workflow jobs) and deeply interact with the RMS to access and release the needed resources. We consider in this analysis Marathon [109] Chronos [25], Kubernetes [171] and Nomad [126]. Marathon and Chronos require a Mesos Resource Management System and can deploy containers as long-term fault-tolerant services (Marathon) or periodic jobs (Chronos). Kubernetes has the capacity of deploying containers (mainly Docker but not limited to it) as services, or running batch jobs as containers. However, any of them deal seamlessly with non-containerised and container-based jobs. In this sense, Nomad can deal with multi-platform hybrid workloads with minimal installation requirements.

4.3 Architecture

The service-oriented architecture is described and implemented in a modular manner, so components could easily be replaced. In Section 4.3.1, the architecture is described in a technology-agnostic way, so different solutions could fit into the architecture. Section 4.3.2 describes all architecture components and how they fulfil the use case requirements identified in Section 4.1.2. Finally, Section 4.3.3 shows the final version of the architecture including the technologies selected and how each one addresses the requirements.

4.3.1 Overview of the Architecture

The system architecture addresses the requirements described in section 4.1.2. The architecture can be divided into two parts: Container Delivery (CD), and Container Execution (CE). It should be noted that although all the components of the architecture can be installed in different nodes, in some cases they could be installed in the same node to reduce costs. As it can been seen in Figure 4.1, the CD architecture is composed of three components: the Container Image Registry, the Source Code Manager (SCM), and the Continuous Integration (CI) tool.

Figure 4.2 depicts the CE architecture. The system consists of four types of logical nodes: Front-end, job schedulers nodes, working nodes and Container Image Registry (this component also appear in CD architecture describe above). The Front-end and job schedulers nodes are interconnected using a private network. The Front-end logical node exposes a REST API, which allows load-balanced communication with the REST API of the job scheduler nodes. Furthermore, it contains the Resource Management Service, which is composed of the Cloud Orchestrator and the horizontal elasticity manager. Job schedulers nodes comprise the master services of the Job Scheduler (JS). Furthermore, as the Front-end is the gateway between users and the job scheduler service, a service for providing authorization and load balancing (between job scheduler nodes) is required. Different working nodes will run the Job scheduler executors. Working

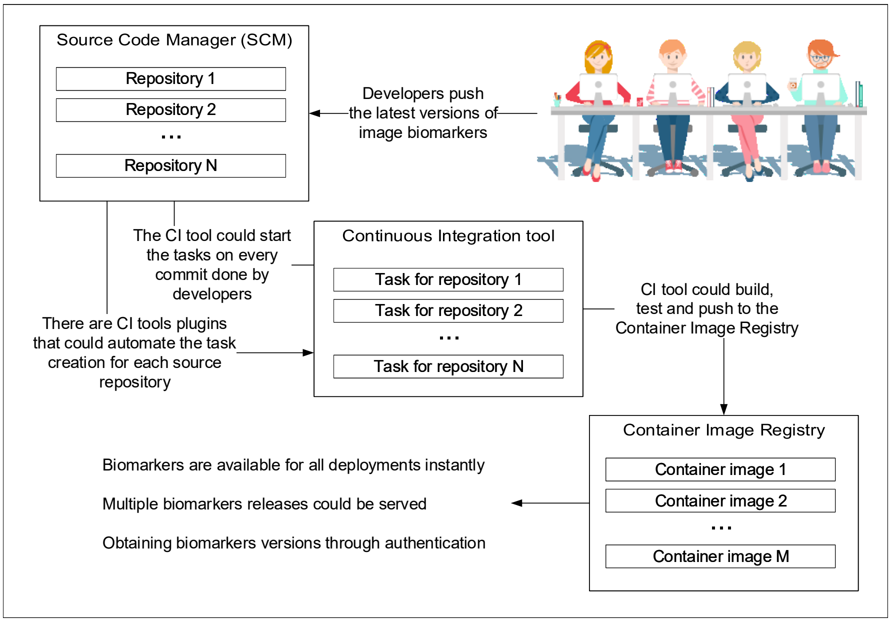

Source Code Manager (SCM)
Repository 1
Repository 2
...
Repository N
Developers push the latest versions of image biomarkers
The CI tool could start the tasks on every commit done by developers
There are CI tools plugins that could automate the task creation for each source repository
Continuous Integration tool
Task for repository 1
Task for repository 2
...
Task for repository N
CI tool could build, test and push to the Container Image Registry
Container Image Registry
Container image 1
Container image 2
...
Container image M
Biomarkers are available for all deployments instantly
Multiple biomarkers releases could be served
Obtaining biomarkers versions through authentication

Figure 4.1: Overview of Container Delivery architecture

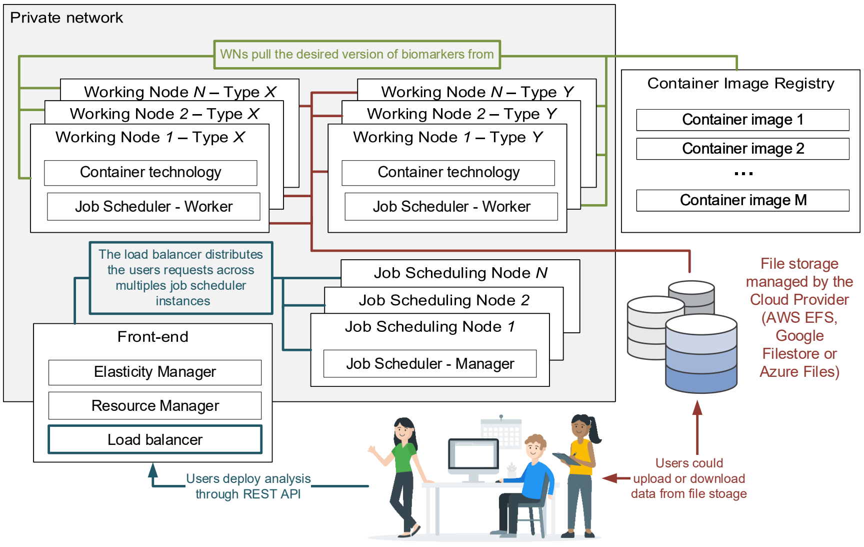


Figure 4.2: Overview of Container Execution architecture

nodes mount locally a volume available from a global external storage. Is should be noted that the set of working nodes can be heterogeneous.

4.3.2 Components

Resource Management service (RMS): It is in charge of deploying the resources, configuring them and to reconfigure them according to the changes on the workload. The requirements stated in Section 4.1.2 focus on facilitating deployment, higher isolation, scalability, application releases management and generic authentication and authorisation mechanisms. RMS may require to interact with the infrastructure provider to deploy new resources (or undeploy them), and to configure the infrastructure accordingly. Furthermore, the deployment should be maintainable, reliable, extendable and platform agnostic.

Job Scheduling service: It will perform the execution of containerised jobs requested remotely by a client application through the REST API. It is required that the job scheduler service includes a monitoring system to provide up-to-date information on the status of the jobs running in the system. Clients will submit jobs through the load balancer service providing a job description formed by any additional information required by the Biomarkers platform, the information related to the container image, the input and output sandbox, and the software and hardware requirements (such GPUs, memory, etc.).

Horizontal Elasticity service: It is necessary to fulfil the Resource scalability requirement, which is strongly related to the Job scheduler and the Resource Manager. Horizontal elasticity tool has to be able to monitor the job scheduler queue and running jobs, and current working node resources in order to scale in or scale out the infrastructure. It is desirable that the horizontal elasticity manager could maintain a certain number of nodes always idle or stopped to reduce the time that the jobs are queued.

Source Code Manager (SCM): It is required to manage the coding source for developers. Due to the release management requirement and the development complexity, it is mandatory to lean on this kind of tools.

Container Image Registry: In order to store and delivery the biomarker applications, it is necessary to use a Container Image Registry. Biomarker applications could be bound to Intellectual Property Rights (IPR) restrictions so Container Image Registry must be private. For this reason, authentication mechanisms are required for obtaining images from the registry. Working nodes will pull the application images when the container image not exists or recent version exists.

Continuous Integration (CI) tool: CI eases the development cycle because it automates building, testing and pushing to the Container Image Registry the biomarker application (with a certain version or tag). Developers or (the CI experts) define this workflow to do it. Furthermore, some of CI tools could trigger these tasks for each SCM commits.

Storage: Biomarker applications make use of legacy code and standard libraries which expect data to be provided in a POSIX filesystem. For this reason, the storage technology must mount the filesystem in the container.

4.3.3 Detailed Architecture

The previous sections describe the architecture and its components. After the technology study done in Section 4.2 and the feature identification for each architecture component in 4.3.2, the technologies selected for addressing the requirements are the following:

- Jenkins as the CI tool due to the wide variety of available plugins (such us SCM plugins). Furthermore, it allows you to easily define workflows for each task using a file named Jenkinsfile. Jenkins provides means to satisfy the requirements RI4, RI5 and RI6.

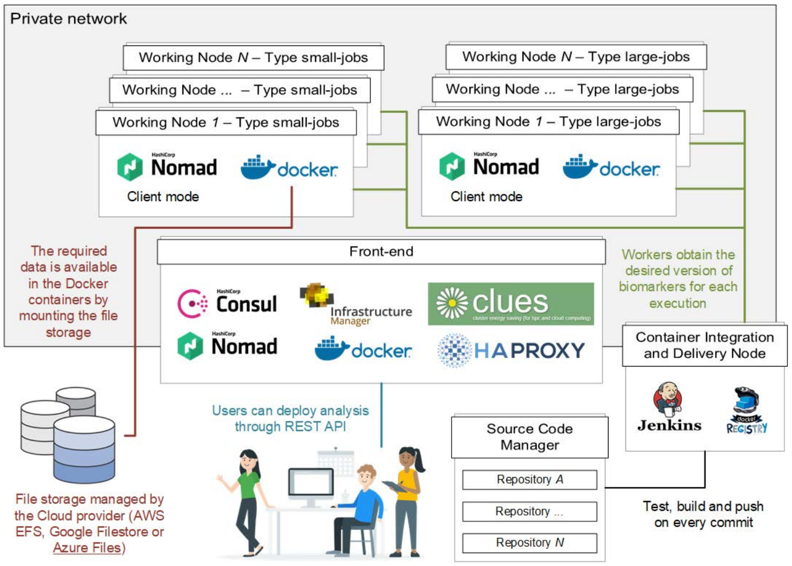

Private network
Working Node N – Type small-jobs
Working Node ... – Type small-jobs
Working Node 1 – Type small-jobs
Nomad
docker
Client mode
Working Node N – Type large-jobs
Working Node ... – Type large-jobs
Working Node 1 – Type large-jobs
Nomad
docker
Client mode
The required data is available in the Docker containers by mounting the file storage
Front-end
Consul
Infrastructure Manager
clues
Nomad
docker
HAPROXY
Workers obtain the desired version of biomarkers for each execution
Container Integration and Delivery Node
Jenkins
REGISTRY
Users can deploy analysis through REST API
Source Code Manager
Repository A
Repository ...
Repository N
Test, build and push on every commit
File storage managed by the Cloud provider (AWS EFS, Google Filestore or Azure Files)

Figure 4.3: Proposed architecture with selected technologies.

- GitHub as SCM because it supports both private (commercial license) and public repositories, which are linked to the CI tool. Furthermore, there is a Jenkins plugin that could scan a GitHub organization and create Jenkins tasks for each repository (and also for each branch) that contains a Jenkinsfile, addressing to requirement RI4. This component meets the requirements RI5, RI6 and RI7.

- Hashicorp Nomad is the Job scheduler. We selected Nomad instead of Kubernetes as Kubernetes can only run Docker containers. Furthermore, Nomad incorporates job monitoring that can be consulted by users. Additionally, it is designed to work in High Availability mode, addressing requirement RI7. Hashicorp Consul is used to resources service discovery as Nomad could use it natively. By using this job scheduler, the architecture meets the requirements RI2, RI4 (job versioning), RE1, RE2, RE3 and satisfies RI5 and RI6 using its Access Control List (ACL) feature.

- Docker is the container platform selected because it is the most popular container technology and it is supported by wide variety of Job schedulers. It provides the resources isolation required by RI2 and support version management (by tagging the different images) of RI4.
- As Docker [147] is the container platform used in this work, Docker Hub and Docker Registry are used as, respectively, public and private container image registry. Requirements RI4 and RI6 are address by using Docker.
- Infrastructure Manager (IM) [15] is the orchestrator chosen because it is open source, cloud agnostic and provides the required functionality to fulfil the use case requirements RI1, RI3, RI5 and RI6.
- CLUES [36] has been chosen for addressing RI3 because it is open source and can scale up or down infrastructures using IM by monitoring the Nomad jobs queue.
- The RMS selected is EC3, which is a tool for system administrators that combines IM and CLUES to configure, create, suspend, restart and remove infrastructures. By using EC3 the system could address the requirements RI, RI3, RI5 and RI6.
- Due to the experimentation will be done in Azure and the current storage solution of QUIBIM is Azure Files, it has been selected as storage. It allows to mount (entirely o partially) the data as a POSIX filesystem, which is the requirement RD1. Also, it provides the mechanisms required to fulfil RI5, RI6 and RD2. Additionally, it allows to mount the same filesystem concurrently.
- HAProxy is used for load balancing because it is reliable, open source and support LDAP or OAuth for authentication.

It should be remarked that the proposed architecture (which is depicted in Figure 4.3) is a simplification of Figure 4.2. For the experiment performed, the job scheduling services are deployed in the Front-end node but users connect with the job schedulers using the load balancer service. So, this simplification does not affect to the users-services communication. Furthermore, in order to avoid costs, the CI tool (Jenkins) and the Docker Private Registry are in the same resource.

4.4 Results

The experiments have been performed on the public Cloud Provider Microsoft Azure. The infrastructure is composed by three type of nodes. The *front-end* node corresponds with the A2 v2 instance, which has two Intel(R) Xeon(R) CPU E5-2660 2.20GHz, 3.5GB of RAM memory, 20 GB of Hard Drive disk and two network interfaces. IM version 1.7.6 and CLUES version 2.2.0b has been installed in this node. HAProxy 1.8.14-52e4d43 and Consul 1.3.0 are running on Docker containers also in that node. The second type of node, *smallwn*, corresponds with the NC6 instance with six Intel(R) Xeon(R) CPU E5-2690 v3 2.60GHz, 56 of memory RAM, 340 GB of Hard Drive disk, one NVIDIA Tesla K80 and one network interface. Finally, the *largewn* node type corresponds with the D13 v2 instance, which has eight Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz, 56GB of RAM memory, 400 GiB of Hard Drive disk and one network interface. The operating system is CentOS Linux release 7.5.1804. Nomad version 0.8.6 and Docker version 18.09.0 build 4d60db4 are installed in all nodes.

4.4.1 Deployment

The infrastructure configuration is coded into Ansible roles[1] and RADL recipes, and they include parameters to differentiate among the deployment. The roles will reference a local repository of packages or specific versions to minimize the impact of changes in public repositories, as well as certified containers. Deployment time is the time required to create and configure a resource. The deployment time of the front-end takes 29 minutes 28 seconds on average. The time required to configure each worker node is 9 minutes 30 seconds.

[1]All Ansible Roles used in this work are available in a GitHub repository. `https://github.com/grycap/`

4.4.2 Use case - Emphysema

The use case selected was the automatic quantification of lung emphysema biomarkers from computed tomography images. This pipeline features a patented air thresholding algorithm from QUIBIM [114] for emphysema quantification and an automatic lung segmentation algorithm. Two versions were implemented. A fast one with rough lung segmentation can be used during the interactive inspection and validation of parameters. Another one with higher segmentation accuracy is implemented for the batch, production case. This brings the need of supporting short and long cases, which take respectively, 4 and 20 minutes.

The small cases are related with executions that take minutes to be completed. In order to provide QoS, CLUES is configured to provide always more resources than required (one node always free). As this type of jobs run very fast, the small-jobs nodes that are IDLE too much time are suspended for avoiding the deployment time. The large cases of QUIBIM biomakers could take many hours and use huge amount of resources, so the deployment time is negligible. For this reason, although the large case of this work takes the same amount of time that the deployment time and the application does not need the all resources of the VM, large-jobs nodes are not suspended and restarted, and only one large-job can run concurrently on the same VM. The "small" Emphysema used in this work consumes 15 GB of memory RAM and 2 vCPUs, so three Emphysema small-jobs could run simultaneously on the same node.

The main goal of the experiment is to demonstrate the capabilities of the proposed architecture. The experiment consists of submitting 35 small-jobs and 5 large-jobs in order to ensure that there are workload peaks that require starting up new VMs and idle periods long enough to remove (in case of large-jobs nodes) or suspend VMs (in case of small-jobs nodes). Table 4.4 shown the time frames were jobs are submitted.

Figure 4.4 shows the number of jobs (vertical axis) along time (horizontal axis) in the different status: SUBMITTED, STARTED, FINISHED, QUEUED and RUNNING. The first three metrics denote the cumulative number of jobs that

Name	Time	Name	Time	Name	Time	Name	Time
small-1	0:01:19	*small-10*	0:47:25	*small-18*	1:00:31	*small-27*	1:10:42
large-1	0:01:20	*small-11*	0:49:25	*small-19*	1:01:32	*large-5*	1:11:44
small-2	0:01:50	*small-12*	0:50:25	*small-20*	1:01:33	*small-28*	1:11:46
small-3	0:05:50	*large-2*	0:51:27	*small-21*	1:02:33	*small-29*	1:14:47
small-4	0:08:51	*small-13*	0:51:27	*small-22*	1:03:34	*small-30*	1:15:48
small-5	0:11:51	*small-14*	0:52:28	*small-23*	1:03:36	*small-31*	1:16:48
small-6	0:16:23	*small-15*	0:55:28	*small-24*	1:08:38	*small-32*	1:17:49
small-7	0:16:24	*small-16*	0:55:29	*small-25*	1:08:38	*small-33*	1:18:49
small-8	0:19:24	*small-17*	0:58:30	*small-26*	1:08:39	*small-34*	1:18:50
small-9	0:22:25	*large-3*	0:59:30	*large-4*	1:09:40	*small-35*	1:21:50

Table 4.4: Scheduling of the jobs to be executed.

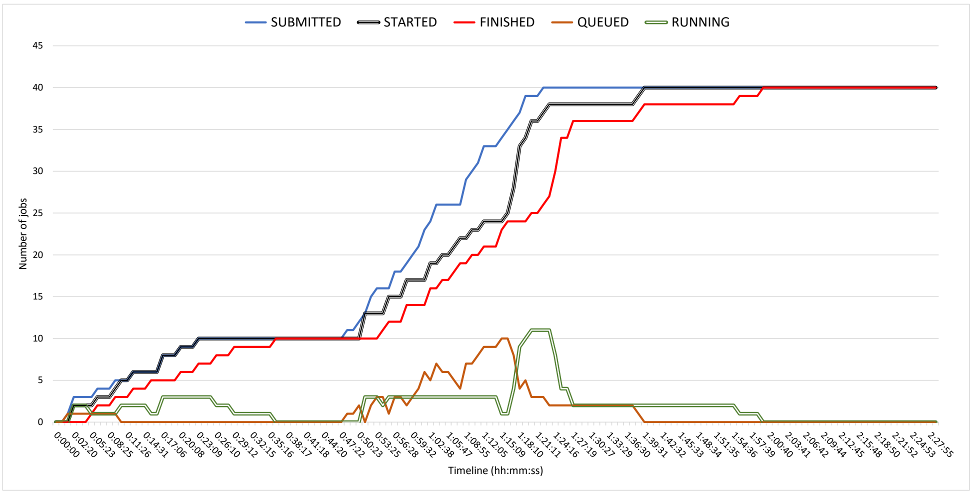


Figure 4.4: Status of jobs during the experiment.

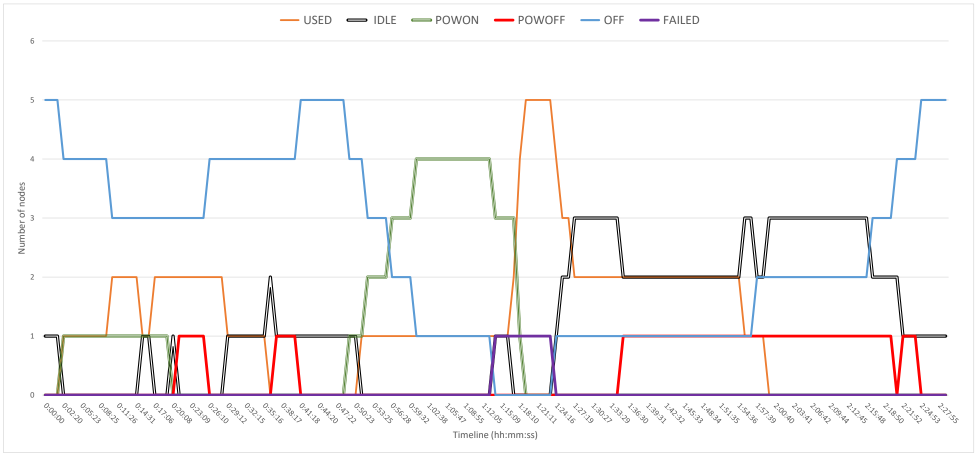


Figure 4.5: Status of working nodes.

have been submitted, have actually started and have been completed over time, respectively. The remaining metrics denote the number of jobs that are queued or concurrently running at a given time.

As depicted in Figure 4.4, the length of the queue does not grow above ten jobs. The delay between the submission and the start of a job (the difference in the horizontal axis between submitted and started lines) is negligible during the first twenty minutes of the experiment. After a period without new submissions (between 00:24:00 and 00:47:00 minutes), the workload grows again due to the submission of new jobs, triggering the deployment of four new nodes (as it can be seen in Figure 4.5).

The largest number of queued jobs (10) is reached during this second deployment of new resources at 01:15:09, although it decreases to four only three minutes later (and to two at 1:23:00). Besides, the four last jobs queued were large-jobs. It should be pointed out that large-jobs have greater delays than short-jobs between submission and starting as they use dedicated nodes and this type of nodes are eliminated after 1 minute without jobs allocated (a higher value will be used in production).

Figure 4.5 depicts the status of the nodes along the experiment, which could be USED (executing jobs), IDLE (powered-on and without jobs allocated), POWON (being started, restarted or configured), POWOFF (being suspended or removed), OFF (not deployed or suspended) or FAILED. CLUES is configured to ensure that a small-node is always active (in status IDLE or USED). It can be seen in Figure 4.5 that two nodes are deployed at the start of the experiment. Then, during the period without new submissions, the running jobs end their executions, so these new nodes are powered off after one minute in the IDLE status. As the workload grows, the system deployed four new nodes between 00:50:00 and 01:13:00. Besides, CLUES tried to deploy one more node (a large-node) but, as the quota limit of the cloud provider's account has reached, the deployment was failed. It should be noted that the system is resilient to this type of problems and successfully ended the experiment. After two hours, all jobs are completed, so CLUES suspends or removes all nodes (except one small-node that has to be active always).

4.5 Conclusions and future work

This paper has presented a agnostic and elastic architecture and a set of open-source tools for the execution of medical imaging biomarkers. Regarding the technical requirements defined in Section 4.1.2, the experiment of a real use case and the results exposed, it can be concluded that all the requirements proposed were fulfilled by the architecture presented. In Section 4.4.2, a combination of 40 batch jobs was scheduled to be executed in a specific time and the cluster achieve to execute all of them by adjusting the resources available. Furthermore, when there are wasted resources too much time, the nodes are suspended (or eliminated).

The proposed architecture are not only related to the execution of batch jobs, it provides to developers a workflow to ease the building, testing, delivery and version management of their application.

Future work includes implementing the architecture on QUIBIM ecosystem, testing other storage technologies as Ceph [179] or OneData [43], the study of Function As a Services (FaaS) frameworks for batch execution (SCAR [145] or OpenFaas [131]) or deploying Nomad, Consul and HAProxy servers as a Kubernetes services, using Kubernetes for ensuring that services are always up.

Chapter 5

Seamlessly managing HPC workloads through Kubernetes

Published as:
Sergio López-Huguet et al. "Seamlessly Managing HPC Workloads Through Kubernetes". In: High Performance Computing*. Ed. by Heike Jagode et al. Cham: Springer International Publishing, 2020, pp. 310–320.* ISBN*: 978-3-030-59851-8*

5.1 Introduction

Most scientific workloads combine requirements that could be efficiently addressed using a combination of *High-Throughput Computing* (HTC) and *High-Performance Computing* (HPC) workloads [16][18][107]. Focusing on Medical Imaging, HPC is extensively used for artificial intelligence model building and simulation. HTC is widely used in image post-processing and applying trained models to new datasets. HTC workloads can be efficiently tackled on cloud computing infrastructures, which fit to massive, coarsely coupled and embarrassingly parallel jobs. HPC workloads typically require infrastructures composed of a large number of highly-coupled computing nodes. HPC infrastructures are typically provided by singular datacenters through specific interfaces.

Cloud computing platforms provide access to a large variety of computing resources on demand and without needing on-premise resources. Therefore, cloud services assist on reducing the cost contention and the ecological impact thanks to the self-adaptive mechanisms that dynamically adjust the cloud infrastructure depending on different aspects. Furthermore, it is possible to build hybrid cloud platforms depending on the institution necessities. Cloud infrastructures are much more flexible than HPC systems. Contrary to Cloud infrastructures, which can be adapted to the application requirements, in HPC systems the applications must be adapted to the execution environment. One important aspect for final users relies on the job management.

On the other hand, an HPC cluster delivers a huge amount of specialized and already configured computation resources to the researchers. Clusters can run free and commercial set of toolboxes which are already prepared to be used efficiently in distributed environments. As a result, running an application in this scenario can be easier than in a pure cloud model for the researcher (who wants to perform calculations and does not want to focus on the hardware and software installation and fine tuning).

The institutions can take benefit of employing a hybrid processing platform composed of HPC and cloud infrastructures. The architecture platform can be complex because there are a lot of aspects to consider: authentication, authorization, data storage, software requirements, special hardware, etc. Furthermore, the majority of the institutions that use HPC infrastructures, use infrastructures that are provided by third parties. Therefore, they must adapt their other processing infrastructures (for example, a public or private cloud platform) to use the different HPC infrastructures. This work presents *hpc-connector*[1], an open-source tool that allows to seamlessly integrate the cloud architecture with the access to the HPC cluster, without administrator privileges.

5.2 Scenario and related work

5.2.1 Architecture

Cloud application architectures typically comprise front-end services and back-end nodes. Front-end services provide external access and manage back-end resources through a job scheduler API or a graphical interface. In some cases, front-end nodes use resource manager tools in order to scale in or out the resources, depending on usage metrics to provide an agreed Quality of Service. The back-end nodes are a set of heterogeneous resources that run the jobs sent by the users through the job scheduler.

There are examples of cloud platforms in the literature that use the previous architecture scheme. In [85], the authors present an architecture to process Internet of Things (IoT) data collected from smart agriculture. In our previous works, we presented a cloud architecture for data analysis [102] and for processing medical imaging [100].

However, there are no examples of hybrid cloud and HPC infrastructures that could provide a seamless interface for both types of workloads. The PRIMAGE project [113] is an ongoing research project that uses artificial intelligence techniques for the processing of medical imaging in paediatric cancer. In this project, the platform architecture combines an HPC infrastructure (the Prometheus supercomputer [34]) and an on-premise cloud platform. The tool presented in this work was designed to solve the problem of combining the execution of some applications in several infrastructures with no administrator privileges.

Job schedulers (or workload managers) manage the remote execution of the applications on the available resources. The most popular job schedulers designed

[1]`https://gitlab.com/primageproject/hpc-connector`

for containers technologies are Docker Swarm, Kubernetes[2], some frameworks of Apache Mesos[3] and Nomad[4]. Regarding workload managers in the HPC environment, there are a lot that are widely used: SLURM [184], Torque[5] or HTCondor[6]. It should be pointed out that *hpc-connector* was designed to integrate any cloud architecture (provided with any job scheduler) with an HPC infrastructure (also provided with any workload manager). In this work, we will use Kubernetes as the cloud job scheduler, and SLURM for the HPC infrastructure.

Application portability and delivery are key issues not only in cloud computing environments but also in HPC. Containers probably are now the most popular technology for application delivery, thanks to the reproducibility, traceability, provenance, isolation, and portability. Docker[7] has reached the maximum popularity as container technology on cloud infrastructures, thanks to its rich ecosystem of tools and great versatility. However, Docker containers run under the root user space and do not provide easily multi-tenancy (it is necessary to create previously the users during the container building stage). Singularity containers [94] are widely used in the HPC environments because they run under user space, support multi-tenancy and provide mechanisms to use Message Passing Interface (MPI). There are other container technologies and runtimes, such as Podman[8], Charliecloud[9], Shifter[10], etc., but we will use Docker and Singularity as container technologies for cloud and HPC infrastructures, respectively. It should be noted that the commonly used HPC workload managers (such as SLURM) can run unprivileged containers without requiring changes or installing a plugin (as containers are processes that are executed in the user space).

[2]`https://kubernetes.io`
[3]`http://mesos.apache.org/`
[4]`https://www.nomadproject.io/`
[5]`https://adaptivecomputing.com/cherry-services/torque-resource-manager/`
[6]`https://research.cs.wisc.edu/htcondor/`
[7]`https://www.docker.com/`
[8]`https://podman.io/`
[9]`https://hpc.github.io/charliecloud/`
[10]`https://www.nersc.gov/research-and-development/user-defined-Chapters/5_ISC_Figures/`

5.2.2 Objectives and requirements

The goal of this work is to provide a tool that permits the combination of a job scheduler that runs applications embedded in Docker containers (for example, Kubernetes) on the cloud environment, with the HPC workload managers that run applications bare metal or in unprivileged containers. Considering the scenario described in the previous section, the identified requirements and assumptions needed to fulfill are:

(R1) Users must use the same method to submit or cancel jobs to the HPC workload manager, as the job scheduler on the cloud environment does.

(R2) The job scheduler used on the cloud environment must manage the full life cycle of a job in the HPC infrastructure. For example, the job scheduler should be able to submit, cancel, get execution state, get the logs, etc.

(R3) The data required to run the job must be accessible in the HPC infrastructure. There are several options: users upload it in advance, the job downloads it before starting, or there is a shared storage between the HPC infrastructure and any other environments that could use the data.

(R4) Any user authorised to access the cloud and the HPC resources must be able to execute jobs without requiring special privileges in neither the HPC nor the cloud infrastructures.

(R5) The solution should be extensible to deal with different job schedulers and workload managers.

5.2.3 State of the art

The combination of the potential of High-Performance Computing for simulation and Big Data and cloud computing for massive data processing has become a driving forces for complex disciplines such as Brain science [22]. The relevance of High-Performance and Cloud Computing for addressing challenges related to medical imaging has boosted with the take off of the application of Artificial Intelligence [116][73]. A revision study [70] highlights 83 articles applying some kind of HPC techniques in Medical Imaging [27], many of them also suitable of being addressed using cloud computing.

Although there are authors that propose hardware specific configurations based on FPGAs and GPUs [77] [89], current tendencies propose the use of cloud computing platforms, especially public offerings [83] with special examples on solutions provided directly by main industry players in cloud [10]. However, in most of the cases, the use of clouds is limited to the storage and access of medical imaging data with low processing capabilities [27][88]. Recently, solutions proposing the combination of container-based platform with computing accelerators have arisen [65].

5.3 The proposed solution: *hpc-connector*

Institutions that manage HPC and cloud environments can integrate job schedulers by developing the appropriate plug-ins and extending the current job types to make them compatible between both environments. Large HPC consortia and institutions may follow this approach. However, in most cases users face a situation in which they can acquire both types of resources from different providers.

Another approach is to adapt the job scheduler on the cloud platform to be able to communicate with the workload manager, as cloud infrastructures are widely accessible and much more flexible than an HPC infrastructure for a regular user. This option could be complex and cumbersome, as it would require continuous work as new updates to the job scheduler arise. If the job scheduler is released under open-source licenses, the institution must extend it with the desired functionality following the rules from the project developers. It should be pointed out that, if the institution wants to use more than one HPC infrastructure with different configurations (for example, in one case you only can interact using a REST API but, in the other case, you can only interact using ssh commands), the software extension could be even more complex.

Adapting the job scheduler to the workload managers could be complex and, besides, it forces to keep using the adapted job scheduler in the future for making the adaption effort profitable. For this reason, we propose a solution by creating an external tool (*hpc-connector*) that manages the jobs in the HPC infrastructure from the cloud infrastructure, as any other job without special privileges.

The key point of the proposed solution is the following: once a user submits a job to the job scheduler that wants to be executed in the HPC infrastructure (fulfilling R1), a special job is executed in the cloud infrastructure. The special mirror job is a an instance of *hpc-connector*, which will manage the job in the HPC infrastructure (R2). Thus, the mirror job updates the job scheduler as the execution of the HPC job progresses in the HPC infrastructure. As *hpc-connector* does not need any special privilege (like mounting a directory or accessing special kernel directives), the special mirror job can run in the user space (fulfilling R4). After submitting the job, *hpc-connector* will monitor it (R2) until the end or its cancellation by the HPC workload manager. Once the job ends, *hpc-connector* can retrieve, if the HPC infrastructure allows it, the job output (R2). Furthermore, *hpc-connector* is able to catch the SIGINT signal that the job scheduler can send to the job before killing it. Therefore, if an appropriate cloud job configuration is performed (the cloud job receives a SIGINT signal and it has a grace period before killing it after the signal is received) and the job still running, *hpc-connector* will cancel the job in the HPC infrastructure (R2).

The tool presented is designed to be running in any environment (even in a local machine) because it is implemented in Python, so it can be running embedded in a any type of container or bare metal. Regarding the support of HPC infrastructures, as each HPC back-end has its own methods for managing the jobs and the data, it is necessary to implement some specific functionality for each back-end in *hpc-connector* to interact with the job submission, information retrieval about the job execution, cancelling, or deleting jobs. Furthermore, if the institution wants to retrieve the logs, it is required to implement how to operate with files (like upload, download, or remove) and operations with directories (list, create, or delete). The tool uses the super-class `Backend` so, for each new HPC infrastructure, a new subclass from the `Backend` class must be created with the name of the HPC infrastructure. For example, let's consider two different HPC infrastructures: *cluster1* and *cluster2*. The infrastructure *cluster1* uses SLURM as workload manager with a REST API. On the other hand, *cluster2* also uses SLURM but only provides a REST API to manage the files, so the users must interact with SLURM via ssh. Thus, *cluster2* implementation is different from *cluster1* because, although both use SLURM, the job operations must be performed using ssh for *cluster2*. Therefore, `cluster1` and `cluster2` sub classes from `Backend` class must be implemented. Thus, *hpc-connector* fulfils R5 because it is designed to be running in any environment and it can be extended for new HPC infrastructures.

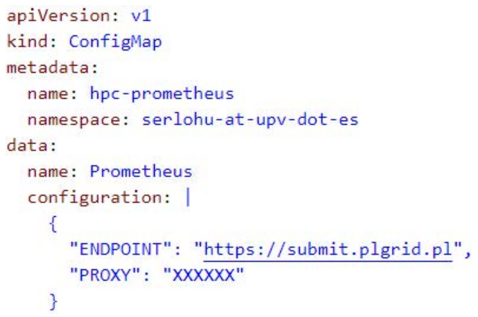
5.4 Use case: Segmentation of neuroblastoma tumours

To validate the usefulness of the *hpc-connector*, we performed a test case. The scenario uses a private cloud platform and the Prometheus HPC infrastructure. The cluster deployed on the private cloud infrastructure is composed of 3 virtual nodes with 4 vCPUs, 32 GB of memory RAM, 80 GB of Solid State Disk and 1 NVIDIA Tesla V100 each. The job scheduler used is Kubernetes (version v1.15.9) and the container technology is Docker (version 18.06.2-ce).

The HPC infrastructure is the Prometheus supercomputer [34], which is located in the 289th position of the TOP500 list (June 2020). Prometheus cluster provides REST API to interact with SLURM (version 19.05.5) called Rimrock (Robust Remote Process Controller)[11] and PLGData service[12] to interact with the file system.

The selected use case is a training of a neural network using Tensorflow for performing an automated segmentation of neuroblastoma tumours. Neuroblastoma is the most frequent solid cancer of the early childhood [13]. This use case belongs to the PRIMAGE project [113].

[11]`https://submit.plgrid.pl/`
[12]`https://data.plgrid.pl/?locale=en`



```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: hpc-prometheus
  namespace: serlohu-at-upv-dot-es
data:
  name: Prometheus
  configuration: |
    {
      "ENDPOINT": "https://submit.plgrid.pl",
      "PROXY": "XXXXXX"
    }
```

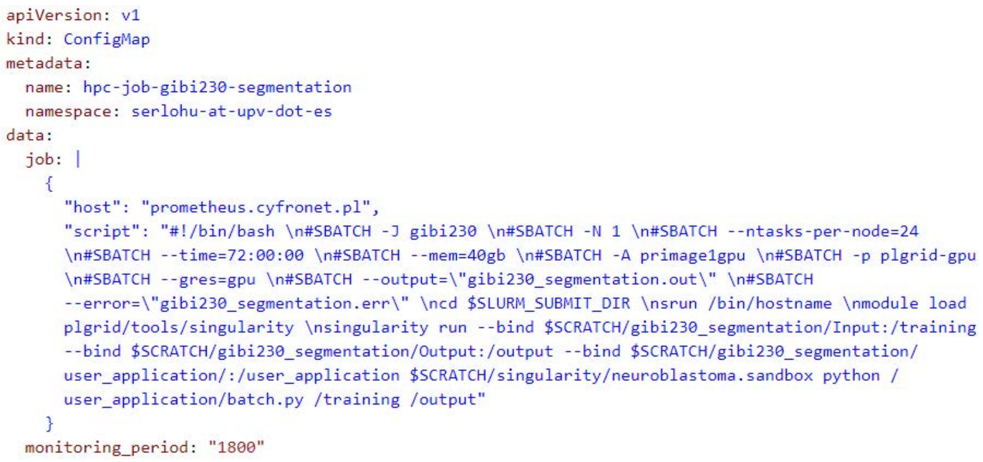
Figure 5.1: ConfigMap definition to specify the Prometheus configuration required by *hpc-connector*.

First, we define two ConfigMap objects, which store non-confidential data in key-value pairs). The Figure 5.1 shows the definition (in YAML[13] format) of the ConfigMap that contains the information required by *hpc-connector* to use the back-end Prometheus. This ConfigMap will be used for all jobs that want to connect with Prometheus. If we were using another HPC infrastructure, the configuration value would maybe contain other dictionary keys.

Once the ConfigMap for accessing properly to the HPC infrastructure is defined, the users must define the job configuration. As we are using Rimrock service from Prometheus cluster, the required parameters are at least, the host and the SLURM script in plain text. In this script, we specify the amount of resources, the special hardware (GPUs), and the batch queue (plgrid-gpu). Then, we implement the tasks: show the hostname, load modules and run the Singularity image (available at the directory `$SCRATCH/singularity/neuroblastoma.sandbox`). Figure 5.2 shows the ConfigMap definition for the job configuration. This new ConfigMap is specific for each job in Prometheus cluster.

Figure 5.3 shows the Kubernetes batch job definition. As it is described in Section 5.3, the job executed in the cloud infrastructure consists of running *hpc-connector* to managing the job in Prometheus cluster. It should be noted that his job is configured with a termination grace period and, if the users cancel this Kubernetes job, it has 30 seconds to execute the command `kill -SIGINT 1`. If this occurs,

[13] `https://yaml.org/`



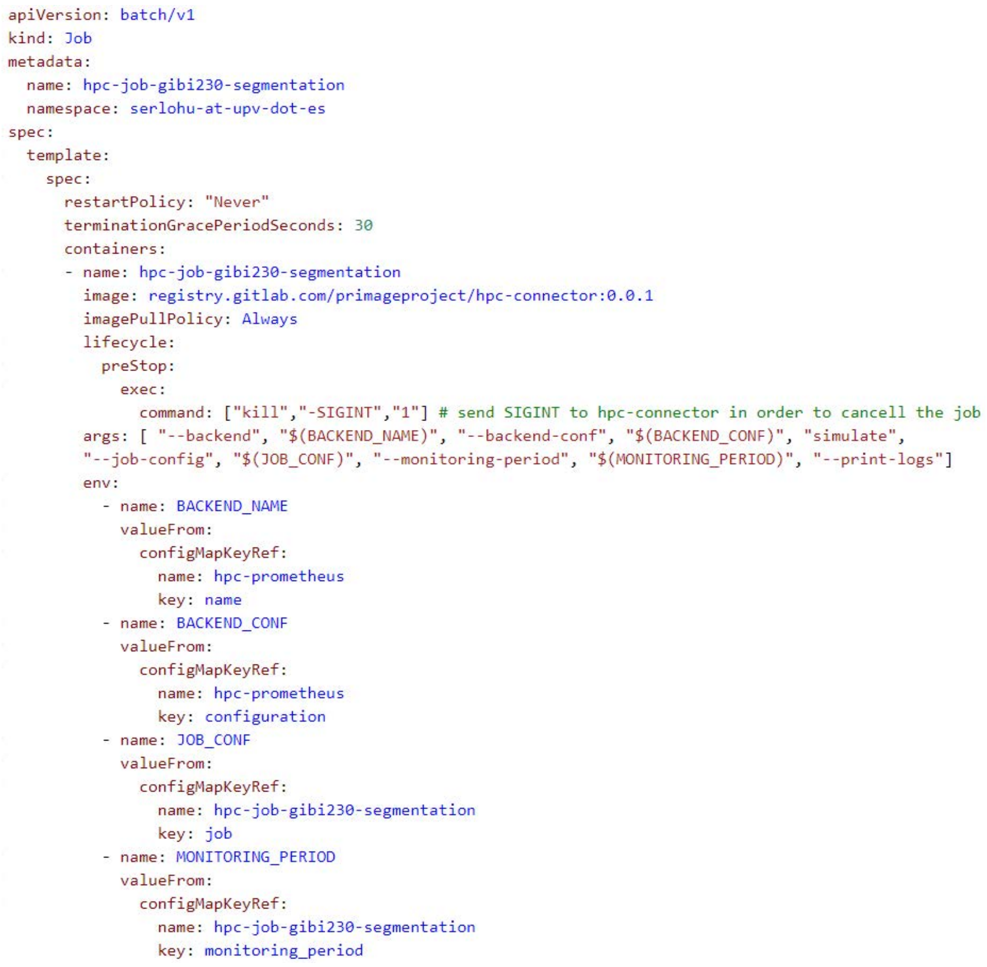
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: hpc-job-gibi230-segmentation
  namespace: serlohu-at-upv-dot-es
data:
  job: |
    {
      "host": "prometheus.cyfronet.pl",
      "script": "#!/bin/bash \n#SBATCH -J gibi230 \n#SBATCH -N 1 \n#SBATCH --ntasks-per-node=24
      \n#SBATCH --time=72:00:00 \n#SBATCH --mem=40gb \n#SBATCH -A primage1gpu \n#SBATCH -p plgrid-gpu
      \n#SBATCH --gres=gpu \n#SBATCH --output=\"gibi230_segmentation.out\" \n#SBATCH
      --error=\"gibi230_segmentation.err\" \ncd $SLURM_SUBMIT_DIR \nsrun /bin/hostname \nmodule load
      plgrid/tools/singularity \nsingularity run --bind $SCRATCH/gibi230_segmentation/Input:/training
      --bind $SCRATCH/gibi230_segmentation/Output:/output --bind $SCRATCH/gibi230_segmentation/
      user_application/:/user_application $SCRATCH/singularity/neuroblastoma.sandbox python /
      user_application/batch.py /training /output"
    }
  monitoring_period: "1800"
```

Figure 5.2: Job definition to specify the job configuration required by *hpc-connector* to launch the job using, in this HPC backend, Rimrock.

hpc-connector will catch the signal and immediately cancel the job in the HPC infrastructure. Once the job is submitted to Kubernetes, it is possible to check (in real time) the progress of the execution consulting the logs of *hpc-connector*. Figure 5.4 is a screenshot of the Kubernetes dashboard showing the logs of the created job.

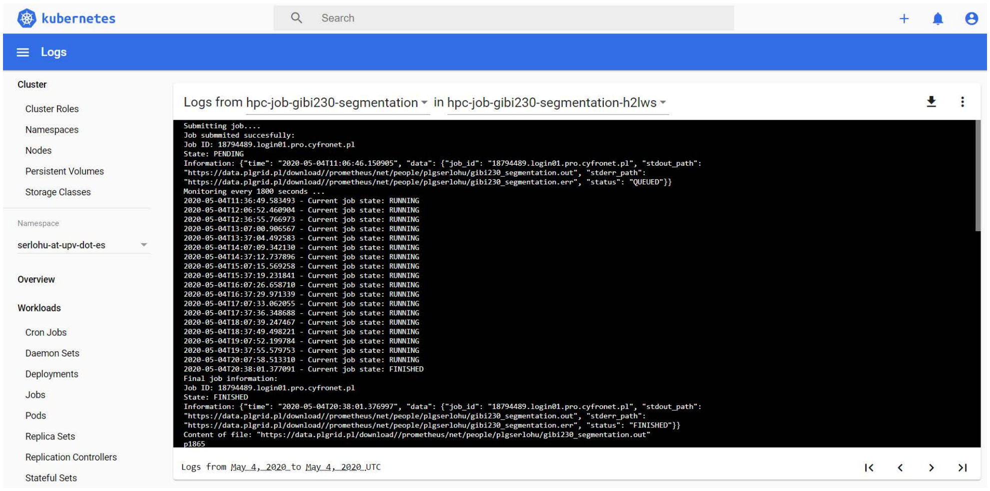
5.5 Conclusions

This paper has presented *hpc-connector*, which is an open-source tool for seamlessly integrating HPC workloads in cloud infrastructures without requiring administrator privileges or changes on the workload manager, providing the users with the same user interface even across different HPC infrastructures. The tool implemented fulfils the requirements identified in Section 5.2.2: running in the user space, and agnosticism of the workload manager (it is implemented as a Python tool easy extendable to other HPC infrastructures) to manage the jobs in an HPC infrastructure.



```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hpc-job-gibi230-segmentation
  namespace: serlohu-at-upv-dot-es
spec:
  template:
    spec:
      restartPolicy: "Never"
      terminationGracePeriodSeconds: 30
      containers:
      - name: hpc-job-gibi230-segmentation
        image: registry.gitlab.com/primageproject/hpc-connector:0.0.1
        imagePullPolicy: Always
        lifecycle:
          preStop:
            exec:
              command: ["kill","-SIGINT","1"] # send SIGINT to hpc-connector in order to cancell the job
        args: [ "--backend", "$(BACKEND_NAME)", "--backend-conf", "$(BACKEND_CONF)", "simulate",
        "--job-config", "$(JOB_CONF)", "--monitoring-period", "$(MONITORING_PERIOD)", "--print-logs"]
        env:
          - name: BACKEND_NAME
            valueFrom:
              configMapKeyRef:
                name: hpc-prometheus
                key: name
          - name: BACKEND_CONF
            valueFrom:
              configMapKeyRef:
                name: hpc-prometheus
                key: configuration
          - name: JOB_CONF
            valueFrom:
              configMapKeyRef:
                name: hpc-job-gibi230-segmentation
                key: job
          - name: MONITORING_PERIOD
            valueFrom:
              configMapKeyRef:
                name: hpc-job-gibi230-segmentation
                key: monitoring_period
```

Figure 5.3: *hpc-connector* job definition.



```
Submitting job....
Job submmited succesfully:
Job ID: 18794489.login01.pro.cyfronet.pl
State: PENDING
Information: {"time": "2020-05-04T11:06:46.150905", "data": {"job_id": "18794489.login01.pro.cyfronet.pl", "stdout_path":
"https://data.plgrid.pl/download//prometheus/net/people/plgserlohu/gibi230_segmentation.out", "stderr_path":
"https://data.plgrid.pl/download//prometheus/net/people/plgserlohu/gibi230_segmentation.err", "status": "QUEUED"}}
Monitoring every 1800 seconds ...
2020-05-04T11:36:49.583493 - Current job state: RUNNING
2020-05-04T12:06:52.460904 - Current job state: RUNNING
2020-05-04T12:36:55.766973 - Current job state: RUNNING
2020-05-04T13:07:00.906567 - Current job state: RUNNING
2020-05-04T13:37:04.492583 - Current job state: RUNNING
2020-05-04T14:07:09.342130 - Current job state: RUNNING
2020-05-04T14:37:12.737896 - Current job state: RUNNING
2020-05-04T15:07:15.569258 - Current job state: RUNNING
2020-05-04T15:37:19.231841 - Current job state: RUNNING
2020-05-04T16:07:26.658710 - Current job state: RUNNING
2020-05-04T16:37:29.971339 - Current job state: RUNNING
2020-05-04T17:07:33.062055 - Current job state: RUNNING
2020-05-04T17:37:36.348688 - Current job state: RUNNING
2020-05-04T18:07:39.247467 - Current job state: RUNNING
2020-05-04T18:37:49.498221 - Current job state: RUNNING
2020-05-04T19:07:52.199784 - Current job state: RUNNING
2020-05-04T19:37:55.579753 - Current job state: RUNNING
2020-05-04T20:07:58.513310 - Current job state: RUNNING
2020-05-04T20:38:01.377091 - Current job state: FINISHED
Final job information:
Job ID: 18794489.login01.pro.cyfronet.pl
State: FINISHED
Information: {"time": "2020-05-04T20:38:01.376997", "data": {"job_id": "18794489.login01.pro.cyfronet.pl", "stdout_path":
"https://data.plgrid.pl/download//prometheus/net/people/plgserlohu/gibi230_segmentation.out", "stderr_path":
"https://data.plgrid.pl/download//prometheus/net/people/plgserlohu/gibi230_segmentation.err", "status": "FINISHED"}}
Content of file: "https://data.plgrid.pl/download//prometheus/net/people/plgserlohu/gibi230_segmentation.out"
p1865
```

Logs from May 4, 2020 to May 4, 2020 UTC

Figure 5.4: Consulting the logs of *hpc-connector* using the Kubernetes dashboard.

The experiment performed in Section 5.4 demonstrated that this approach can address a wider range of complex problems in a convenient way. In this experiment, we successfully trained a neural network using GPUs in an HPC supercomputer (Prometheus) using *hpc-connector* from a Kubernetes job hosted in a private cloud infrastructure.

Future work includes improving the functionality of *hpc-connector*: upload the data (from a repository or from a directory in the Docker container) before submitting the job (if necessary) or consider retrieving the results and storing them in an external repository or in the Docker container itself (for example, if the container has mounted a shared filesystem). Another possible enhancement could be the ability to refresh the HPC credentials when they expire.

Chapter 6

Automated Isolation Management of Processing Workflows in a Multi-Tenant and Multi-Site Kubernetes clusters - A Medical Imaging Use Case

Submitted to IEEE Access as:
Sergio López-Huguet, Pau Lozano, J. Damià Segrelles, and Ignacio Blanquer "Automated Isolation Management of Processing Workflows in Multi-Tenant and Multi-Site Kubernetes clusters - A Medical Imaging Use Case".

6.1 Introduction

The need of platforms for running distributed data analytics [97] [167] [160] and computing-intensive scientific applications has been increasing over the years, as well as the computational requirements to deal with the storage and processing of those applications. These types of platforms are normally shared by multiple users [19] [91], even beyond the limits of a single organization. In those cases, it is necessary to consider multi-tenant authentication and authorization and the different relationships among the teams. In addition, the processing infrastructures may need to provide, on the same platform, different environments with different needs for different groups of users. Moreover, production execution environments also require an efficient distribution and reconfiguration of resources to adapt to its workload and development environments require more flexibility to deal with different configurations and more interactive access. Designing, configuring and maintaining a multi-tenant and elastic infrastructures is not a simple task [144], especially in a multi-site scenario.

Since the appearance of Docker in 2013 [40], containers have grown in popularity to become one of the most widely used technologies for application delivery because they provide an isolated execution environment that has a faster startup than virtual machines (VM). Most commercial cloud computing systems use containers to manage applications in isolated environments through container orchestrator services, which occupy a key part in cloud architectures. Kubernetes [171] is positioned as the standard solution for container orchestration (according to the 2019 Cloud Native Computing Foundation survey [1], it is used in production mode in 78% of the cases). Most commercial cloud computing platforms offer administrated Kubernetes clusters, such as Amazon Elastic Kubernetes Service (Amazon EKS) [6], Azure Kubernetes Service (AKS) [11] or Google Kubernetes Engine (GKE) [69]. As well as other solutions for managing on-premise Kubernetes clusters such as Rancher[1], EC3 [119], and OpenShift[2]. Docker offers isolation of different container executions, but it does not isolate

[1]`https://rancher.com/`
[2]`https://www.openshift.com/`

users, so users can access and manage any container deployed in the system, even by other users who share the access to the resources. For this purpose, container orchestrator tools provide or use additional services to guarantee the isolation of the executions of different users. In Kubernetes, isolation can be providing by creating namespaces, which are an abstraction that provides support for multiple and isolated partitions of the computing resources on the same physical cluster, and assigning quotas to those namespaces, where users can be bound. Thus, managing user authentication and authorization and namespace isolation is key but can be complex, even more so if it is considered on distributed scenarios where this has to be automated. In any case, managing the authorization in a multi-site environment will require some degree of coordination among the providers, which could be cumbersome for users and system administrators.

This article presents two works. On the one hand, the *kube-authorizer* tool, which is an open-source service that automates the creation of namespaces, service accounts, and permissions of users authenticated by OpenID Connect[3] (OIDC) in a Kubernetes cluster. On the other hand, the design of a multi-site, multi-tenant and an elastic cloud container-based architecture with distributed storage that leverages *kube-authorizer* to automate the creation of authorization policies in Kubernetes. The architecture was designed considering a scenario for processing medical imaging data, which involves several Kubernetes clusters, although the work can be applied to other use cases by defining general architecture requirements.

The reminder of the paper is structured as follows. First, section 6.2 details the objectives of the article. Section 6.3 revises the state of the art related to the work presented in the paper. Then, Section 6.4 present the service *kube-authorizer*. Section 6.5 a federated Kubernetes cluster detailing the architecture of their sites. Section 6.6 correspond with the study of the security thread model. Finally, Section 6.7 summarizes the main results, concludes the paper and points to future work.

[3] `https://openid.net/connect/`

6.2 Motivation and Objectives

In the recent years, Kubernetes has become the standard solution tool for container orchestration. Kubernetes provides abstractions to describe, isolate, and share computational resources among different users or entities. The user concept in Kubernetes can be divided into two categories: regular users or service accounts. Regular users map to the underlying operating system users and Service accounts are intended to be used by processes rather than humans. Service accounts are namespaced (in contrast with normal users that exist in the whole cluster). Kubernetes provides different methods[4] for authenticating users: X509 Client certificates, static tokens per users, bootstrap tokens, service account tokens, and OIDC Tokens.

As the computing infrastructure must be shared among a large number of users, the utilization of OIDC as an authentication method in Kubernetes is the most appropriate because, every time a user enters the cluster, Kubernetes creates a user and, if it is configured for that, Kubernetes assigns the user to a certain group that depends on the claims obtained from the ID token. This allows to apply Role-based access control (RBAC) policies to different groups of users. However, Kubernetes does not provide a way to automatically generate isolated environments (namespaces) for each tenant, so it is necessary to create a new service to take care of it. Furthermore, it does not provide an OpenID Connect Identity Provider such as Google, Github[5], dex[6], Keycloak[7], EGI Check-in[8], etc. Regarding multi-tenancy in Kubernetes, the community is working on multi-tenancy models for extending Kubernetes. These models present limitations (for example, some of them are not mature projects) and benefits (for example, one of them enables the ability to create hierarchical namespaces). Depending on the needs of the workload, the multi-site architecture, and the desired level of the isolation between tenants, their utilisation could be recommended. It should

[4]`https://kubernetes.io/docs/reference/access-authn-authz/authentication/`
[5]`https://github.com/`
[6]`https://dexidp.io/`
[7]`https://www.keycloak.org/`
[8]`https://www.egi.eu/services/check-in/`

be pointed out that none of them offer the creation of isolated environments for each tenant automatically.

The second objective of the work focuses on one of the essential features of cloud computing. Cloud computing platforms provide a wide range of computing resources and services that have made them an efficient solution to get the resources needed by institutions on demand. Thus, the institutions can take benefit of them to address their necessities. The elasticity is a key aspect in cloud platforms as they allow to adapt infrastructures according to the workload, avoiding unnecessary costs. These infrastructures can be complex to manage because they must consider aspects such as elasticity, reconfiguration of computational resources, shared storage, as well as authentication and authorization.

Moreover, our scenario applies to multi-site and multi-tenant infrastructures, where the management of the authentication is performed on a central and trusted service which is used by the different sites to perform the authentication and authorization. By the use of this approach, we can automatically create a homogeneous and trustworthy environment where accounts are created on the fly only when resources are needed.

The aims of this work are twofold: 1) the creation of a tool that facilitates the management of authorization in Kubernetes clusters where users authenticate through OIDC; and 2) the design of an elastic and multi-tenant cloud architecture based on Kubernetes with a distributed file system. Fulfilling both objectives implies meeting the requirements described in the sections 6.4 and 6.5, respectively.

6.3 Related work

Cloud computing enables institutions to deploy fully customised computing environments on demand. For example, in the field of Bioinformatics, there are some examples of tools that have been adapted to cloud environments in [78]. Additionally, there are works that are focused on providing pre-configured platforms that contain a large number of tools. An example of this is Galaxy Project[9], a web platform that can be deployed on public and on-premises Cloud offerings (e.g. using CloudMan[10] for Amazon EC2[11] and OpenStack[12]). Focused on medical imaging processing, the authors in [120] presents a on-premise infrastructure using Eucalyptus[13] that offers ImageJ[14] to analyse medical images. In [100], the authors present an elastic cloud architecture for medical imaging processing that can be deployed in the majority of cloud provider platforms. It should be pointed out that this architecture is not a multi-site architecture, it does not consider multi-tenancy and it uses non-distributed storage.

Containers have become the most popular technology for application delivery during the last years thanks to the reproducibility, isolation, provenance, and portability. Nowadays, there are several container technologies that are designed to provide different features that are more appropriate than the rest of them for certain scenarios. Almost all container engines and runtimes are compliant with the specifications defined by the Open Container Initiative[15] (OCI).

Regarding container technologies, Docker[16] has reached the maximum popularity due to its convenient and rich ecosystem of tools and the support by the majority of container orchestrator tools. Docker manages containers using *containerd*[17], which relies on runC[18] library the container execution. Although Docker and runC can run containers in user namespaces (Docker rootless mode is an experimental feature [39]), they commonly require privileged daemons to manage containers. Other alternatives to Docker emerged during the recent years trying to facilitate the acquisition of containers in other environments such as HPC infrastructures: Singularity [94], Shifter [125], CharlieCloud [146], udocker

[9]`https://galaxyproject.org`
[10]`https://galaxyproject.org/cloudman`
[11]`https://aws.amazon.com/ec2/`
[12]`https://www.openstack.org/`
[13]`https://github.com/eucalyptus/eucalyptus`
[14]`https://imagej.nih.gov/ij/`
[15]`https://opencontainers.org/`
[16]`https://www.docker.com/`
[17]`https://containerd.io/`
[18]`https://github.com/opencontainers/runc`

[67] or Podman[19]. The Docker ecosystem is inherited by the majority of them for building and storing images, and even some of them enable users to use pure Docker images or convert to their own formats. Those solutions could minimize the impact and security concerns of Docker in Computing platforms, although they are insufficient by themselves to conveniently address multi-site and multi-tenant authorisation.

Regarding container orchestration tools, Kubernetes[20] (also known as *k8s*) is currently the most popular solution used to run OCI compliant containers. In fact, Kubernetes announced in 2020 that its default container runtime will be CRI-O[21] instead of supporting by default Docker containers (by using the container runtime *dockershim*). *Pods* are the smallest scheduling unit in Kubernetes, consisting of groups of containers that are deployed and scheduled together. There are other tools to manage containers such as Nomad[22], Docker Swarm[23] or Apache Mesos[24]. Docker Swarm is the native job scheduler of Docker but it is being replaced by Kubernetes (even Docker Engine has included a standalone version of Kubernetes). Nomad is an open-source job scheduler that allows running non-containerised and container-based jobs with different container technologies. Apache Mesos is not a container orchestration tool, it is a large-scale resource management system that partition and assign computing resources across various job schedulers, which are also called frameworks (such as Spark[25], Hadoop[26], etc.). For this reason, it is possible to run Docker Swarm and Kubernetes also can run on top of a Mesos cluster.

Authentication in Kubernetes can be performed using different methods[27]: X509 Client certificates, static tokens per users, bootstrap tokens, service account tokens, and OIDC Tokens. Managing user authentication can be complex in the case of there are a considerable number of users. For this reason, it is interesting

[19]`https://podman.io/`
[20]`http://kubernetes.io/`
[21]`https://cri-o.io/`
[22]`https://www.nomadproject.io/`
[23]`https://docs.docker.com/engine/swarm/`
[24]`http://mesos.apache.org/`
[25]`http://spark.apache.org`
[26]`http://hadoop.apache.org`
[27]`https://kubernetes.io/docs/reference/access-authn-authz/authentication/`

the use of OIDC as a authentication method. Kubernetes does not provide a OIDC Identity Provider for managing users and groups so it is required to use an external one and either an OIDC client. Kubernetes solutions for managing on-premise Kubernetes clusters such as Rancher allow to access Kubernetes clusters by including an OIDC client as authentication method of its own service. This way, when a user is authenticated in Rancher by OIDC, he or she can access to his or her clusters. There are some OIDC Identity Provider available that are compatible with Kubernetes such as Keycloak[28] or EGI Check-in[29], to name a few.

EGI Check-in is a proxy service that operates as a central hub to connect federated IdPs to other services. It allows combining user attributes from various sources (IdPs and attribute provider services) in a transparent way. Users in Check-in are divided in *Virtual Organization* (VO), which it is a group of users that besides allow to define different roles for each user.

Kubernetes has a big community that enhances the whole ecosystem by adding different tools and new features. For example, Helm[30] (package manager to configure applications or services), Kubernetes Ingress Controller[31] (it exposes HTTP and HTTPS routes from outside the cluster to services within the cluster) or the Kubernetes Dashboard[32], to name a few.

Kubernetes Cluster Federation[33] (also known as *KubeFed*) is an open-source extension of the Kubernetes API by using *CustomResourceDefinitions*[34] (CRD). *KubeFed* was designed focused on managing life cycle of multi-cluster applications (for example, a web service that must be deployed in multiple regions) easily. Basically, *KubeFed* enables to propagate Kubernetes API objects (Jobs, Deployments, Namespaces, etc.) to multiple clusters. In a *KubeFed* federated cluster, the propagation of the selected Kubernetes objects is in charge of a

[28]`https://www.keycloak.org/`
[29]`https://www.egi.eu/services/check-in/`
[30]`https://helm.sh/`
[31]`https://kubernetes.io/docs/concepts/services-networking/ingress/`
[32]Source code repository: `https://github.com/kubernetes/dashboard`
[33]`https://github.com/kubernetes-sigs/kubefed`
[34]`https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/`

centralised federated control-plane, so one Kubernetes cluster acts as a master of the rest of them. The principal limitation of this approach is that all the federation is centralised in one cluster. Another aspect to take in account is the maturity of the project (nowadays is in alpha state). Although *KubeFed* is a very good solution for scenarios such as microservices that must be running in multiple sites, it does not fit the scenario we are addressing, as objects have to be created in the central service and artifacts and objects are propagated even in sites that may not be used in the future. Our approach address an automatic creation of the namespaces, different policies at each site, and creation on the fly.

The Kubernetes Multi-Tenancy Working Group focuses on defining tenancy models for Kubernetes. Multi-tenancy models in Kubernetes can be categorized in three types, each one with its benefits and limitations.

The clusters as a service tenancy model consists of each tenant gets their own Kubernetes cluster. An example of the tenancy model is the project Cluster API[35] (CAPI). In this scenario, the tenant clusters are provisioned by a management cluster. This tenancy model allows full isolation of the Kubernetes between the tenants and also full control over the cluster resources. On the other hand, it is required to provide an computing resources infrastructure to deploy the clusters and, as the infrastructure is not shared, it can cause that nodes have lower utilization.

The control planes as a service model is a variation of the cluster as a service. In this scenario, each tenant gets their own dedicated Kubernetes control planes, but unlike the previous tenancy model, they share the computing resources. An example of this type of tenancy model is *Virtual Cluster*[36], which extends the namespace-based Kubernetes multi-tenancy model by providing each tenant a cluster view. Thus, the core Kubernetes components are not modified in virtual cluster. In this tool, the physical nodes are managed by a Kubernetes cluster called a *supercluster*. Using this type of tenancy model provides a great isolation

[35]`https://cluster-api.sigs.k8s.io/`

[36]`https://github.com/kubernetes-sigs/multi-tenancy/tree/master/incubator/virtualcluster`

as the cluster as a service but, thanks to the cluster is being shared among the different tenant clusters, the efficiency is better.

The namespace as a service tenancy model consists of isolating tenants at the namespace level. In this scenario, the Kubernetes cluster is shared between tenants. Using this type of multi-tenancy implies configuring in a proper way *RoleBinding* (managing access to Kubernetes objects and namespaces), *NetworkPolicy* (accept or deny network traffic between tenants), and *ResourceQuota* (for limiting usage between tenants). These configurations enable fine-grain policies but also they can be an attack vector between tenants if they are not well configured. Besides, manually creating namespaces and these policies for each tenant cannot be a solution for big teams or institutions. For other hand, using this type of multi-tenancy the same cluster is shared between tenants. One example of this type of model is the Hierarchical Namespace Controller[37] (HNC) project, which enables the creation of additional namespaces under parent namespace propagating resources within the hierarchy. By default, the resources propagated are RBAC policies and RoleBinding but is also possible to propagate the Kubernetes objects: *NetworkPolicy*, *ResourceQuota*, *LimitRange*, *Secret*, and *ConfigMap*. HNC is useful when many users have similar policies or when users belong to a team, but it is required to isolate their own services between the rest of the team.

The utilization of these multi-site and multi-tenant tools described above is recommended depending on how restrictive must be the isolation of the tenants, the architecture of the infrastructure, and workloads that the infrastructure must face. The utilization of *KubeFed* is appropriate for microservices that must be up in multiple clusters, but it is not fitting very well for batch execution. Regarding multi-tenancy, the namespace as a service enables a good level of isolation in an efficient way. HNC is an interesting option to facilitate the propagation of policies but it does not provide a way to automatically generate namespaces for each tenant, so another service must create them. Additionally, HNC is not a mature extension of Kubernetes, indeed, the last version is 0.7.0[38].

[37]`https://github.com/kubernetes-sigs/multi-tenancy/tree/master/incubator/hnc`
[38]`https://github.com/kubernetes-sigs/multi-tenancy/releases/tag/hnc-v0.7.0`

6.4 kube-authorizer

The first aim of this article is the creation of a tool that automates, for each user, the creation of its own Kubernetes namespace, the creation of a service account for the user, and the application of the desired RBAC policies to both entities (user and service account). Service accounts are designed to be used by processes, but they also provide service tokens that can be used by users who wants to access without using OIDC authentication. The requirements of this tool are:

(R.1) The service must be accessible though REpresentational State Transfer (REST) or Simple Object Access Protocol (SOAP). Preferably, it must be a REST service.

(R.2) The tool must create a namespace and a service account inside it, and apply the desired ClusterRole and Role policies (these policies must be available in the Kubernetes cluster).

(R.3) The solution must provide a way to apply Role policies (which are namespaced) in its own user namespace and in other namespaces.

(R.4) The tool is not a OpenID Connect Identity Provider, it will receive the username and the email. For example, if the Kubernetes authentication is through OIDC, the username can be the claim *subject* (*sub*). Then, the tool will perform the actions described in (R.2).

(R.5) The solution requires a service account token that can manage the Kubernetes objects described in (R.2) at both cluster and namespace levels.

(R.6) The tool must perform their actions with the minimum intervention of the administrators in a secure way. Preferably, it must be autonomous.

(R.7) Kubernetes API endpoint must be accessible by the developed tool.

(R.8) The service must be transparent to end users.

(R.9) The service must provide a method to automatically update the applied RBAC policies.

(R.10) The solution must be open source and it should be available as a Docker container.

To fulfil the requirements described above, the authors present *kube-authorizer*[39][40] tool. *kube-authorizer* is a REST API service that, for each REST request received, composed of the email and the username, it creates (if not exists) a namespace and, inside it, a service account. Then, *kube-authorizer* applies the RBAC policies and create the *generic_resources* (this feature will be detailed below). Figure 6.1 depicts the flow diagram actions of *kube-authorizer*. In this figure, it can be seen the difference between the first user access and the following. The name of both Kubernetes entities (namespace and service account) is based on the email to facilitate the differentiation between users when they utilise Kubernetes. As the email has characters that are not allowed for composing the name in Kubernetes objects, the email is modified until it works for Kubernetes. For example, the email "serlohu@upv.es" is transformed to "serlohu-at-upv-dot-es". The username obtained in the REST request correspond to the user in the Kubernetes cluster.

The tool was designed to be used in combination with an *OIDC proxy*[41][42], which is in charge of authentication, but it can run alone if it is required. The combination of both services is useful for putting other services (like Kubernetes Dashboard[43]) behind *OIDC proxy*. In that case, once users are authenticated when are accessing the proxy, the *OIDC proxy* will validate if the user satisfies certain claims (for example, if belongs to a determined group). If the validation is correct, the *OIDC proxy* will perform a request to *kube-authorizer* with the information to trigger the authorization. Finally, it redirects to the Kubernetes Dashboard adding bearer token that contains the ID token of the user. It should be pointed out that the Kubernetes cluster must be configured for enabling OIDC authentication in Kubernetes API. The architecture presented in Section 6.5 uses EGI Check-in as OIDC Identity Provider and its concept of group, so the claim that the user must

[39]Source code repository: `https://gitlab.com/primageproject/kube-authorizer`
[40]Docker image: `https://hub.docker.com/r/primage/kube-authorizer`
[41]Source code repository: `https://gitlab.com/primageproject/oidc-proxy`
[42]Docker image: `https://hub.docker.com/r/primage/oidc-proxy`
[43]Source code repository: `https://github.com/kubernetes/dashboard`

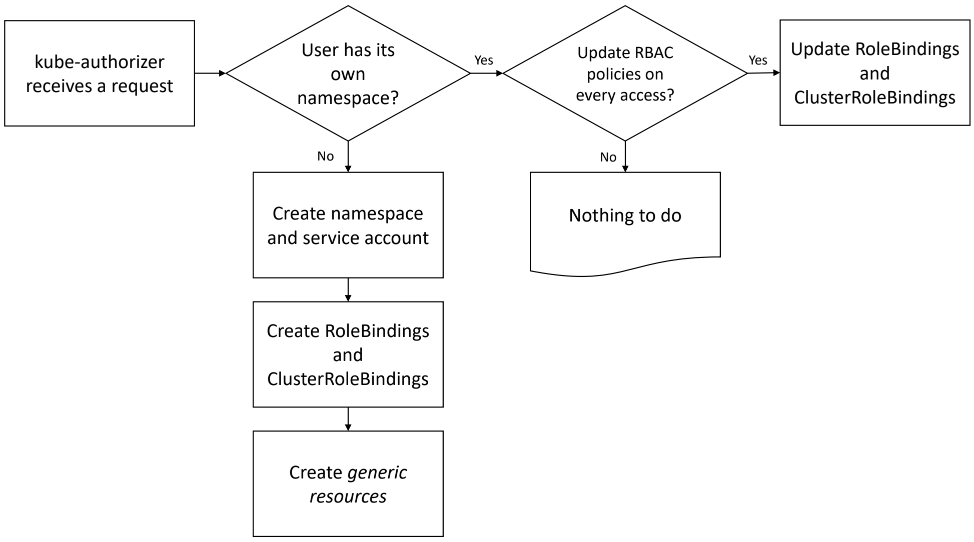

kube-authorizer receives a request
User has its own namespace?
Yes
No
Update RBAC policies on every access?
Yes
No
Update RoleBindings and ClusterRoleBindings
Create namespace and service account
Nothing to do
Create RoleBindings and ClusterRoleBindings
Create *generic resources*

Figure 6.1: Flow diagram that corresponds with the actions performed by *kube-authorizer* on every request.

satisfy correspond with the virtual organization, i.e. the user belongs to certain VO. The actions that are performed on every access through the *OIDC proxy* are depicted in Figure 6.2.

In addition to the requirements already described, *kube-authorizer* has the ability to run a set of containers when the authorization is triggered. This feature, called *generic_resources*, provides system administrators a way to add other actions that are not contemplated in *kube-authorizer*. The actions that administrators can do using *generic_resources* must be defined as a JSON that contains a Kubernetes object (ConfigMap, Job, etc.). These actions will be implemented using the service account token of *kube-authorizer*, so they have enough privileges to create Kubernetes objects in a privileged namespace or in the namespace of the user that trigger these actions. For example, the creation of a ConfigMap that contains information to help users to connect other infrastructure services must be created on each user namespace. Otherwise, a Job that contains tokens to perform administrator actions (such as connecting a database or adding a user in other services like distributed storage) must be created in a namespace that are only accessible by system administrators. In section 6.5, there is an example of the use of *generic_resources*.

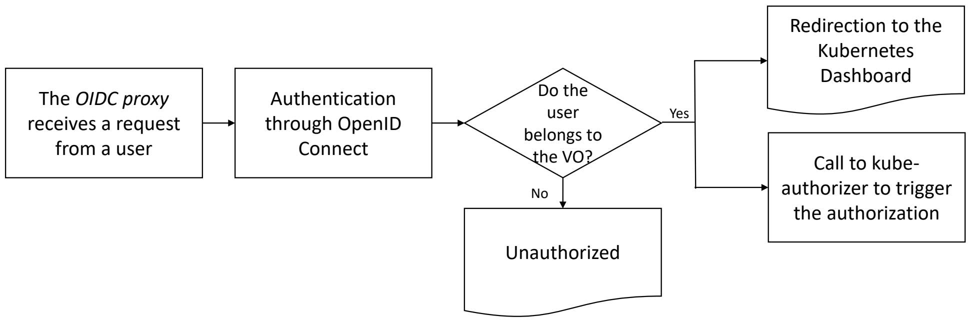

The *OIDC proxy* receives a request from a user
Authentication through OpenID Connect
Do the user belongs to the VO?
Yes
No
Redirection to the Kubernetes Dashboard
Call to kube-authorizer to trigger the authorization
Unauthorized

Figure 6.2: Flow diagram that corresponds with the actions performed by the *OIDC proxy* on every access.

6.5 Architecture design

The second aim of this work is the design of an elastic, multi-site and multi-tenant cloud architecture based on Kubernetes with a distributed file system. A requirement elicitation and analysis process were performed, leading to the identification of the following requirements for the infrastructure, the execution environment, and the distributed storage solution:

(R.1) The architecture must be agnostic to the cloud platform. Thus, the proposed solution must can be deployed on different public and on-premise cloud offerings.

(R.2) Resources should be automatically configured in the deployment with minimal intervention by system administrators.

(R.3) The solution must consider horizontal elasticity of the computing resources, adding or removing nodes depending on the workload.

(R.4) The Identity Server used to enable OpenID Connect should provide a wide variety of Identity providers or a way to connect with them.

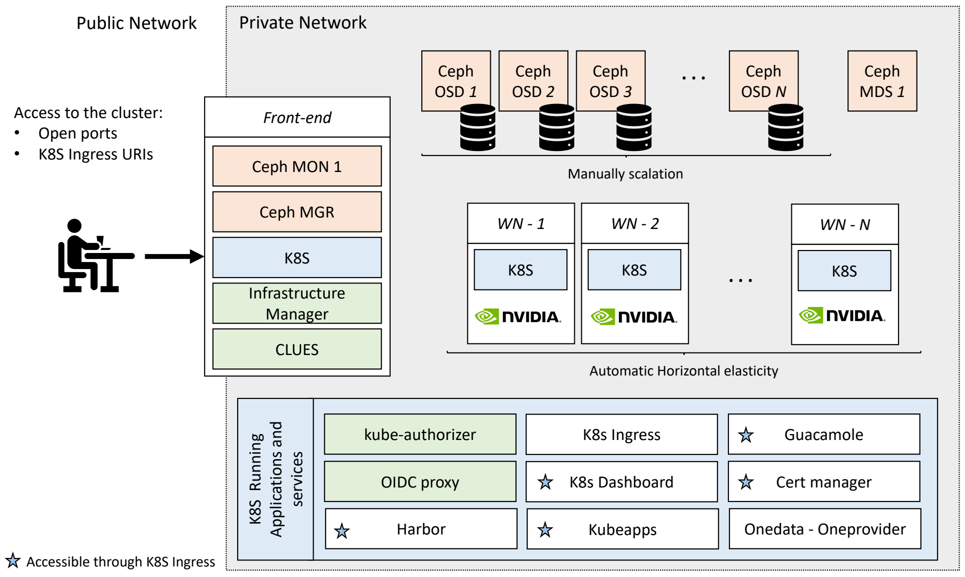


Figure 6.3: Proposed architecture.

(R.5) Users should be able to authenticate in the computing infrastructure and in the distributed storage using OpenID Connect. It is desirable that all services that can be used from tenants allow authentication using OpenID Connect.

(R.6) The solution must consider the authorization of users to access computing resources with minimal intervention by system administrators.

(R.7) The solution provided must ensure that the tenants have isolated environments.

(R.8) The applications used by tenants are embedded in containers.

(R.9) There are different types of jobs that the architecture must address. On the one hand, batch jobs that comprise a set of files (can be mounted), a container image, and a set of computing requirements. These types of jobs are the execution in production mode of the application. On the other hand, the architecture must provide fully-configured workstations that allow users to enter in development environment. Workstation require access through graphical user interface (GUI) or Secure Shell (SSH).

(R.10) Users must be able to customize of the execution environment of the jobs.

(R.11) The architecture should include a high-performance distributed storage solution. Preferably, it must include a solution that allow to combine different storage back-ends.

(R.12) The storage solution must offer access control list (ACL) to manage the permissions of the data.

(R.13) Applications typically require find the data in a specific directory of a POSIX file system.

(R.14) The software used must be open source.

With the study of the features from the different technologies and the literature, the authors propose the architecture depicted in Figure 6.3. The components of this architecture are described in the following paragraphs. The architecture is composed of five types of nodes that can be divided in three categories: the distributed storage cluster nodes, the worker nodes and the *front-end*.

For one hand, the technologies selected to fulfil the requirements (R.1), (R.2) and (R.3) are Infrastructure Manager (IM) [15], CLUster Energy Saving (CLUES)[36], and Elastic Cloud Computing Cluster (EC3) [119]. IM is an open-source tool that deploys complex and customized virtual infrastructures on multiple private and public cloud platforms. CLUES is an elasticity manager tool that has plug-ins to trigger elasticity actions depending on the job queues for a great number of job schedulers, such us Kubernetes, SLURM, Nomad, etc. EC3 is a tool that relies on IM and CLUES for creating elastic virtual clusters on top of Infrastructure as a Service (IaaS) providers.

Regarding resource management and job scheduler, the technology selected is Kubernetes, which is an open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. Kubernetes has a big ecosystem of tools, extensions, and a extensive documentation. Regarding the execution of jobs, Kubernetes provides different type of Kubernetes objects depending on the needs. For example, the object *Job* is often used to run a determined number of times a group of *pods* (running a scientific application one time, i.e. a batch job) but the object *ReplicaSet* is often used to guarantee the availability of a specified number of identical *pods* (for example, multiple instances of a web server). Kubernetes allows to use different type of *volumes* (such as NFS, Ceph, etc.) to obtain data persistence between executions. Kubernetes allows to expose ports from running *pods* to the rest of the cluster or externally through *services*.

Kubernetes has an active community that develops a lot of interesting tools that, at some point, there are integrated natively. One useful tool is Kubernetes Ingress, which is a proxy server that manages external access via HTTP or HTTPS to the internal *services* in a cluster. Certificate management in a multi-tenant environment could be a complex task and the tool *cert-manager*[44] is used for controlling with the certificates. Another tool is Kubernetes Dashboard, which allows users to interact with Kubernetes API through a browser. It should be pointed out that, in the architecture described, the Kubernetes Dashboard can be accessible only by Kubernetes Ingress.

The Kubernetes authentication method selected is OpenID Connect tokens. The OIDC identity provider selected is EGI Check-in because it offers a huge number of IdPs thanks to it is federated in eduGAIN[45] as a service provider. Thank to this, (R.4) is fulfilled.

The multi-tenancy model used in this architecture is a namespace as a service. This is achieved thanks to the use of *kube-authorizer* and a modified OIDC proxy (both are described in Section 6.4). The use of both technologies provides the ability to create namespaces for each tenant, the application of policies and the creation of customised Kubernetes objects. The creation of the tenant namespace is done automatically as soon as the tenant access the cluster at first time using

[44]`https://cert-manager.io/docs/`
[45]`https://edugain.org/`

Kubernetes Dashboard. Kubernetes Dashboard is the only method allowed to access the cluster by the tenants from outside the cluster (from pods they can connect to the Kubernetes API). These limitations is done because it is desirable that clusters that are connected to Internet must have the minimum attack vectors possible.

It should be pointed out that the use of the hierarchical namespaces extension is discarded because it is not able to generate namespaces for each tenant. However, hierarchical namespaces can be used in combination of *kube-authorizer* and the OIDC proxy. Nevertheless, as tenants are single users (not teams), the use of this technology does not provide more features. In case of tenants were teams, it could be interesting to combine all technologies.

The creation of a federated Kubernetes using *KubeFed* is discarded. For one hand, *KubeFed* is useful when there are a lot of microservices that must be running in multiple clusters (for example, a international service that have replication in multiple zones of the world). The services that are running in the sites of this architecture are installed in an automated way when the infrastructure is configured using EC3. For the other hand, *KubeFed* allows to federate namespaces between sites but, in a scenario with a shared distributed storage and a generation of namespaces and policies automated, it is not interesting. Finally, and not least important, *KubeFed* force to select a master Kubernetes cluster it can be a problem for big consortia. Besides, it is not a mature project, indeed it is currently working on a beta version.

Thanks to its features and big ecosystem of tools, the use of Kubernetes, *kube-authorizer* and the OIDC proxy allow to fulfil the requirements (R.5), (R.6), (R.7), (R.8), (R.9), (R.10) and (R.14).

Regarding storage, two technologies are used: Ceph[179] and Onedata[43]. In a multi-site scenario, it is possible that all sites were not managed by the same institution (for example, a big consortium). For this reason, the combination of different storage solutions could be complex. To solve these problems, Onedata is used because it is a multi-tenant solution that offers unified data access across globally distributed environments and multiple types of underlying storage. Onedata is composed of two main components: Onezone[46] and Oneprovider[47]. Onezone is responsible for authentication and authorization of users and makes it possible for users from different Oneprovider to communicate with each other and share data. Oneprovider is in charge of expose the data from a storage solution. In the case of the architecture presented in Figure 6.3, the storage solution is Ceph. Thus, Onedata exposes a virtual storage combining the data from Oneproviders transparently. The access to the data can be through web interface (access like Google Drive), mounting the filesystem using Oneclient[48], and integrated with Jupyter Notebook via the OnedataFS Python library. Onedata ecosystem fulfil requirements (R.5), (R.12), (R.13) and (R.14).

The infrastructure provides fully configured workstations to the tenants. The workstations are containers configured with the proper tools that are accessible through SSH and GUI (R.9). In a multi-site and on-demand scenario, the computing infrastructure must provide users with persistence and shared storage among executions and sites. For this reason, a personal directory will be created in the filesystem for each user thanks to the feature *generic_resources* of *kube-authorizer*. This personal storage will be available in the all sites and in different executions.

Ceph is an open-source high-performance distributed storage solution that provides object, block, and file storage (CephFS). The decision of select Ceph instead of another high-performance solutions such as GlusterFS[49] or Lustre[50] is motivated by various aspects. For one hand, Ceph has support for most major orchestration and deployment frameworks. For example, Ceph provides with an Ansible playbook[51] to install the all system. It should be noted that Ansible is the technology used by the IM tool, which is in charge of configuring the computing resources in the architecture presented. For the other hand, it is well supported

[46] `https://github.com/onedata/onezone`
[47] `https://onedata.org/#/home/documentation/stable/doc/administering_onedata/provider_overview.html`
[48] `https://github.com/onedata/oneclient`
[49] `https://www.gluster.org/`
[50] `https://www.lustre.org/`
[51] `https://github.com/ceph/ceph-ansible`

by Kubernetes thanks to it is possible to use CephFS as a persistent volume natively, offering persistence to the services deployed in the Kubernetes cluster.

The other tools and services available in the Kubernetes cluster are the following:

- *Apache Guacamole*[52] is a clientless remote desktop gateway that allows to connect with the protocols VNC, RDP, and SSH to workstations deployed in the cluster by the tenants. The service is exposed by Kubernetes Ingress and is a key tool for the multi-tenant requirements because it provides a single and secure end-point for authenticated access to multiple containers deployed by the tenants. It avoids the need of open a port in the firewall for every workstation. As the service is running inside the cluster, it allows to connect containers directly from outside the cluster with the possibility of using a graphical desktop and directly using just the browser, even for transfer files (up and down), there is no need for the user to download or install any client application. With Guacamole, the requirement (R.9) is fulfilled.
- *Harbor*[53] is a multi-tenant and open-source registry and Chart repository that can be configured to ensure that the images are scanned and free from vulnerabilities, and signed. The images can be organized in different *projects* which allows applying resource quotas and controlling the access to certain images only to authenticated users and based in predefined roles (RBAC). Anonymous users only has read access to public *projects*. Logged-in users have more or less privileges in the *projects* where they are included depending on their role. Besides, OIDC authenticated users can be auto-assigned to a project and role depending on their group.
- *Kubeapps*[54] is a web-based user interface for deploying and managing applications in Kubernetes clusters. It can be configured to only allow install/deploy applications that are available in a Chart Repository (like Harbor). It can simplify the deployment of jobs and workstations to the

[52]`https://guacamole.apache.org/`
[53]`https://goharbor.io/`
[54]`https://kubeapps.com/`

users of our cluster less familiarized with the edition of Kubernetes manifests (YAML/JSON). They just have to choose from the collection what they want to deploy and visually select and set values for those parameters defined in the chart (for example volumes to mount or daemons to run, like VNC server to connect latter through Guacamole).

It should be noted that all the services that can be used by tenants rely in OpenID Connect to authenticate users fulfilling (R.5). Also they are open source (R.14) and can be deployed with Helm charts.

6.6 Threat model

The architecture has been defined and implemented for supporting the storage and processing of anonymized medical imaging data. Anonymization in medical imaging data has been traditionally argued as it inherently captures unique individual features. Therefore, regardless of dealing with anonymized data, measures to guarantee security, data privacy and isolation are needed.

This section analyses the threat model at the different levels, considering the components involved, the potential effects of a vulnerability and the mitigation measures that should be implemented. For this analysis, we first identify the actors involved and therefore the impact that an adversary for each actor could have if gaining access.

The architecture implemented and presented in the paper focuses on the automatic management of Authorization by trusting on a third-party IdP and a Virtual Organization (VO) service. Users from this VO can only access the platform services and cannot access the resources. Resources are managed by two levels of platform administrators (one at the level of the virtual infrastructure and one at the level of the physical resources). Therefore, we identify four types of actors:

1. Cloud resource system admin. This is the highest privileged user that manages the on-premise cloud management system (for example,

OpenStack) and the physical nodes supporting it. A Cloud resource system admin adversarial could have full access to data and resources.

2. Platform administrator. This is the user authorised to deploy resources as Virtual Machines and Volumes, and who deploys the Kubernetes and all associated services of the platform. A platform administrator adversarial will have access to data and could stop the platform services or replace them by malicious services.
3. VO user. This is a user that can access the Kubernetes resources and is limited to a specific namespace, created on demand. A VO user adversarial will only be able to access the part of the data that the user is authorized and will not be able to delete or replace platform services.
4. Public user. This is a user who can access an application running on the Kubernetes cluster deployed by a VO user. Impact of a Public User adversarial will be quite limited and will be restricted to a subset of data and processing through the application in which she is authorized.

In this schema, the actors defined at one level could perform the actions of the following (higher number) levels. Therefore, an adversary or a vulnerability at the level of a VO user (for example) could not grant access to the underlying Virtual Machines, and could not be able to access the resources or data of a different VO user.

The rest of the section covers the analysis of the threats at the level of the different components of the architecture. We follow a similar approach, starting with the services that have the minimum impact and going down by layers. By exploiting a vulnerability at level i, a malicious user could perform all the actions expressed from this level downwards. It is important to note that this part does not cover gaining access through compromised credentials, but being able to exploit the services and changing the configuration. Despite this may be unlikely in most cases, it helps understanding the criticality of each component.

1. Console access to a processing application

- Guacamole is a remote desktop application for a GUI Linux system. By exploiting the service, an unauthorized user may grant access. As it runs inside a container, this will affect only to the data and applications available inside the Guacamole container.

- Docker Console attach. Services and applications run on Docker containers. Docker containers isolate applications from the host where containers run and from other applications running on other containers. As containers images are restricted, vulnerabilities in Docker containers will be limited. Gaining access to a Docker console will also affect only the data accessible through such console.

2. Container Orchestrator Service level access

   - Kubernetes services are only accessible in the internal network of the Virtual Infrastructure. However, granting access to the K8s services will enable a malicious user to have full control of the platform.

   - IM Service. It acts as Cloud Orchestrator and stores the identifiers of the resources. The IM service does not store resource credentials, although they are provided by the user when interacting with the service. A compromised IM service could obtain infrastructure credentials from the users.

   - Kube-Authorizer provides the creation of the namespaces of the users on demand and the authorization of VO users. Kube-authorizer is totally transparent to the user, who even do not need to explicitly access it. Vulnerabilities at this level could provide access to other user's namespace or even the default one, where the platform services run.

   - Harbor provides a secure repository for Kubernetes Helm Charts and Docker images. This component is key to ensure that only signed and authorized images run in the platform. Vulnerabilities in this level will allow a user to run an arbitrary code.

   - Kubeapps is used to deploy the Helm Charts that describe the applications. By gaining access to the Kuebapps service, the reference repository could be changed, therefore skipping the limitations on the execution of verified images only.

3. Data services

   - Ceph is the main backend for the storage of the data. Data is accessed through the Ceph FileSystem using the access token credentials. By gaining access to the Ceph storage objects, data could be compromised.

   - OneData is the solution for data and metadata federation. Gaining access to OneData services could also compromise data.

4. Fabric access

   - Virtual Machines (VMs) from the Virtual Infrastructure. Only Platform administrator users can access the VMs of the infrastructure. By accessing the VMs, a malicious user could grant access to the services of the platform, potentially accessing the data. Moreover, gaining access to the front-end VM could lead to gain access to any other resource in the virtual deployment.

   - OpenStack Services. OpenStack is the main backend for the management of the Virtual Infrastructure. Gaining access to the OpenStack service could open the possibility to accessing the raw resources, which could end up with accessing the whole platform.

In order to reduce the attach surface, the following measures should be applied:

- Only the services accessible by VO and public users (OneData, Kubeapps, Guacamole) should be externally accessible. The rest of the services will be accessible only through a VPN.

- No SSH port will be openly accessible (only through VPN, and accessing though Kubernetes Dashboard or Guacamole).
- Docker images will be checked, validated and signed prior to be included in the Harbor registry. Users will not have `sudo` privileges within the Docker containers.
- Kubernetes endpoint will not be accessible directly, but only through the Kubeapps application, and Kubernetes Dashboard, which will have only one certified repository (Harbor) of images and charts. Only the custom images that were validated by system administrators should be run in the infrastructure.
- Kube authorizer will only trust users from the official VO. Users will only be allowed to enrol the official VO if they use their institutional credentials and give a proof of membership. VO membership will require adhering to the collaboration Access User Policy and will grant access for a limited period.
- VM access will only be allowed through DSA key pairs and will be securely kept in the VO User resources.
- OpenStack keys should be also kept in the private computers and should follow strong institutional password requirement policies.

The threat evaluation outlines that no arbitrary code is allowed, VO and public users have limited access privileges and network connections are encrypted.

To achieve a higher degree of security, the following actions could be performed:

- Use encrypted storage at the level of Ceph with an external vault storage. This will minimize the risks of vulnerabilities in Ceph and OneData, but not at the level of the other services.
- Use enclaves to run the applications. This will reduce the risk of data leakage even if access to the VM is granted.

6.7 Conclusions

This paper has presented *kube-authorizer*, an open-source service that allows to automate, for each user, the creation of its namespace and service account, the application of RBAC policies and the creation of customised Kubernetes objects. Regarding the technical requirements identified in Section 6.4, it should be pointed out that the service implemented fulfil all of them.

This paper also presented an agnostic, elastic and contained-based software architecture that leverages on *kube-authorizer* and an *OIDC reverse proxy* to automatically manage the authorization in federated Kubernetes clusters. The architecture was built with a scenario in mind: processing medical imaging data through several Kubernetes clusters in a multi-tenant scenario. However, the requirements identified to design the architecture were defined as general as possible in order to provide a general use architecture. Nevertheless, this work can be extended to other use cases redefining or adding new requirements. The configuration recipes to deploy the processing infrastructure are easily reproducible by using the Ansible roles available in the group Github repository[55].

Treating sensitive data implies to carefully analyse the security model of the whole infrastructure at different levels. A study of the thread model of the infrastructure was performed in Section 6.6 in order to identify the risks and vulnerabilities at the different levels.

It should be pointed out that the architecture and *kube-authorizer* were successfully deployed in production in the framework of the European project PRIMAGE[56]. Thus, we conclude by stating that all objectives proposed at the beginning of the research have been successfully achieved.

[55]https://github.com/grycap/
[56]`https://www.primageproject.eu/`

Chapter 7

Discussion of the results

The results of this thesis have been published in several JCR-indexed journals and conference proceedings and are included in this manuscript as chapters. The architectures developed in the thesis (detailed in Chapter 3, 4, and 6) comprise computational infrastructures in cloud platforms. Cloud backends provide the capability of adapting the infrastructure to the actual workload, considering both vertical elasticity (Chapter 2) and horizontal elasticity (Chapter 3), which are used to provide a fair QoS. Chapter 6 presents a tool to 17/manage the authorization in a Kubernetes cluster and an elastic, multi-site and multitenant cloud architecture. HPC backends provide high-end computational capabilities to complete computing-intensive jobs. In Chapter 5, a method to interconnect cloud and HPC architectures was presented. The thesis also shows the evolution of the container-based technologies from Mesos to Nomad and Kubernetes, adapting their developments to the changes in the landscape.

In this chapter, the author analyses the achievements obtained during the research phase as well as other additional contributions developed during the research phase done during the research phase. Then, publications and contributions in international conferences and indexed journals are listed. Finally, the research projects in which the author has participated during the development of this thesis are detailed.

7.1 Contributions

Cloud services provide institutions with a wide variety, flexible and powerful computing resources. These aspects have contributed to the fast adoption of cloud platforms to address the computational needs of many institutions. The contributions of this thesis aim to facilitate the adoption of cloud platforms in the research centres. To do so, the author provides different technological solutions, which were designed considering the requisites identified depending on some particular aspects. Tables 7.1 and 7.2 show a comparison between the related works and the results presented in this thesis.

Elasticity is a key aspect that provides cloud environments. At the level of the infrastructure, horizontal elasticity is used to undeploy idle resources to reduce cost as well as to ensure that the computing infrastructure can manage a workload peak. Vertical elasticity of the nodes usually implies reloading the virtual machines, thus it is not a common mode of elasticity when the workload cannot be anticipated. All the architectures proposed in this dissertation have a mechanism that provides horizontal elasticity depending on the state of the cluster (for example, a combination of the size of the job queue and the node usage), thanks to use the tools developed in the research group to which the author belongs, namely Infrastructure Manager, CLUES, and EC3 (the combination of the two previous tools). Using these tools, the elasticity actions are triggered based on the state of the cluster or some rules, in other words, the elasticity actions are triggered in a reactive way. Thus, one improvement of the architectures described in this thesis is to enhance them with a mechanism that triggers the elasticity actions in a proactive mode anticipating the future workload. The author of this thesis enhanced the CLUES tool with the creation of a plugin for CLUES that allows connecting to Nomad job scheduler. Moreover, the author created some configuration recipes for EC3 and IM, and actively tested the new functionalities of these tools. At the job management level, all the container orchestrator tools support managing multiple instances that belong to the same application, or even provide a mechanism that autonomously scales the number of instances of an application (for example, the Autoscaler of Hashicorp Nomad or the Horizontal Pod Autoscaler of Kubernetes to manage the horizontal elasticity of running applications). Regarding the vertical elasticity of containers, a mechanism was developed that triggers the elasticity actions depending on the quality of service, expressed as a time limit for completing the execution of a batch job. The mechanism can modify the amount of CPU assigned to the job without special privileges due to it acting at the level of the job scheduler. Thus, the mechanism could be used in computing environments without special privileges. The mechanism developed provides two possibilities of use: applications which perform a known number of independent iterations (with a predictable execution time each one) or applications whose progress cannot be monitored. In the first case, the user must provide an estimation of the duration of an individual iteration. In the second case, the user must provide an estimation of the CPU time consumed to complete the job. As the mechanism triggers the elasticity actions in a proactive way (by anticipating if the execution will be finished before the time limit) using the information provided by the user, the principal problem is that the mechanism will work as well as the precision of the information provided by the user. In the second case, as the application execution can be freezed and restored by using checkpointing (CRIU), the application must support it. Besides, this strategy makes sense only when the application can take benefit of the increment of CPU share and the number of cores or memory assigned. Machine learning training applications can leverage this type of elasticity.

The design of a cloud architecture must consider the application properties and necessities (both hardware and software), as well as the institution workload. The cloud architecture presented in Chapter 3 was designed considering the wide variety of workloads that can be part of a big data analysis (data acquisition, parallel spark jobs, etc) and to be elastic and agnostic to the platform. It should be noted that this cloud architecture could be considered when applications consume a lot of computing resources because of the overhead produced by the deployment and the configuration of the nodes instead of having the maximum resources running. In this work, the authors also provide a set of configuration recipes to implement the architecture in the majority of cloud platforms, but they do not consider a continuous integration tool to automatically build and test the Docker containers. Besides, due to the use of certain frameworks of Apache Mesos, this work only considers the utilization of Docker containers.

Chapter 4 presents the evolution of the previous cloud architecture. In this work, the authors designed a heterogeneous cloud architecture for the execution of medical imaging biomarkers that also includes a workflow to ease the building, testing, delivery, and version management of medical imaging biomarkers thanks to the adoption of a continuous integration tool (for example, Jenkins). The authors identified the requirements of QUIBIM, which is a company focused on the development of image processing techniques to measure quantitative information of medical images. After determining the requirements, a cloud architecture considering two types of jobs with different necessities (either short or long-running jobs requiring GPU accelerators) was defined bringing the necessity of different types of computing nodes. Notwithstanding EC3 enables the creation of a hybrid cloud architecture, the architectures presented in Chapters 3 and 4 do not consider the integration with other external computing infrastructures (that may be / might be managed by the same institution).

Sometimes big consortia or partners in research projects need to interconnect different existent computing infrastructures. Scientific problems commonly use applications that have different requirements or hardware architectures. Thus, in some cases, it is required to interconnect different existent computing infrastructures. The developed tool *hpc-connector* is an extra layer to interconnect different architectures. It is required to implement how to do all actions on each new computing infrastructure. It should be pointed out that it is also an advantage because it allows to customize the authentication and the communication methods for each computing infrastructure or even not using containers. The experimentation performed in this work was an extension of a Kubernetes cluster that uses Docker containers with the ability to run jobs in an HPC infrastructure in Poland (the Prometheus supercomputer) embedded in Singularity containers. It should be noted that, although *hpc-connector* can download or upload files, the data required for the computation in the external computing infrastructure was not uploaded by *hpc-connector* (it only downloads the output log). Another limitation can be that *hpc-connector* does not have a mechanism to automate the renew of the credentials.

The tool *hpc-connector* was developed during a research stay of three months in Krakow (Poland) in the framework of a European project named PRIMAGE. This research project uses Onedata as the technology to provide distributed storage among the different storage cluster of the project partners. The author of this thesis also worked on different solutions to make accessible the distributed storage in the supercomputer Prometheus. Those solutions to enable mounting Onedata filesystems in Singularity containers require administrator capabilities to allow containers to mount the filesystem or to install some packages that enable the utilization of Docker in rootless mode. Due to the fact that those solutions require interacting with supercomputer administrators and the Onedata team has in the roadmap to allow mount their filesystems without administrator privileges, this research line was not continued. Other work related to the Singularity technology was the creation of an Ansible role to install Singularity Registry server that is referenced in the official documentation.

Computing infrastructures are commonly used by multiple people. Docker offers isolation of different container executions, but it does not separate the containers of the users of an infrastructure, so containers can communicate with each others. Therefore, container orchestrator tools must provide or use additional services to guarantee the isolation of container executions of different users. Managing user authentication and authorization can be complex. In the previous works, multi-tentancy was not considered as an essential requirement. Chapter 6 presents *kube-authorizer*, a REST service to automate the creation of namespaces, service accounts, and permissions of users authenticated by OpenID Connect (OIDC) in a Kubernetes cluster. It can be used in combination with an OIDC proxy as a gateway to the Kubernetes dashboard or as an external service that will be invocated by other services. The *kube-authorizer* service will update the authorization configuration of the Kubernetes cluster when receiving a REST request with the name of the user (for example, the claim *sub*). In the case of using it in combination with an OIDC proxy, when a new user enters to the cluster through the Kubernetes dashboard, the OIDC proxy makes a request to *kube-authorizer* to guarantee the access to the dashboard. Kubernetes has support for OIDC authentication, but some of OIDC servers are not serving directly the information required to ensure that the user is allowed to access the system (sometimes more than one request to the OIDC server is required). Chapter 6 also presents a multi-site, multi-tentant and elastic cloud architecture with distributed storage. This cloud architecture can be seen as an evolution of the architecture presented in Chapter 4 that includes, on the one hand, the mechanism to manage the authorization, and, one the other hand, the use of the multi-site distributed storage solution Onedata. The cloud architecture is a general architecture; it was not designed focusing on medical image processing platforms. The main limitation of this work is that it only works for Kubernetes. However, Kubernetes is currently the most popular container orchestrator.

Thanks to the predoctoral grant obtained by the author, he taught 12 credits ECTS during the last two years of the Computer Science degree in the Universitat Politècnica de València. Moreover, thanks to the work done during the research phase, the author of this thesis developed Cluster Elasticity Manager (CEM) [76], which is a set of services to provide an elastic and agnostic virtual laboratory that can be accessed by the students via a graphical remote desktop. CEM is composed of three main components: server, agent and web. The web component is the endpoint where the students ask for resources and the teachers can see information of the general state of the system and the resource assignation (current and historical data). The agent component is executed on each working node with the objective of monitoring the users behaviour to detect that they whether they are active or not. The server component is in charge of elasticity actions and the assignation of resources to the students. It should be noted that the author took profit of the experience obtained during the thesis by developing an elastic virtual laboratory mechanism that can be deployed in the most popular computing platforms.

Context - Name	Vertical elasticity	QoS
Related work - Elasticdocker [2]	It monitors CPU and memory usage to modify the vCPU and the memory assigned by modifying the cgroups files directly, so it requires administrator privileges and, if there is a job scheduler, it is possible that it is possible that it tries to allocate resources that are in use.	N / A (Non Applicable)
Related work - Makeflow [185]	It is a workflow engine that is able to monitor and manage jobs in different jobs schedulers that provide mechanisms to automate the vertical elasticity actions to improve the system utilization.	N / A
Related work - BBQ: Elastic MapReduce over Cloud Platforms [21]	N / A	A framework that provides horizontal elasticity to Hadoop MapReduce in AWS.
Related work - DC/OS. Autoscaling with Marathon [35]	This is a proof of concept to autoscale Marathon applications based on the utilization of metrics which Mesos reports. It should be pointed out that this mechanism considers that jobs do not store a persistent state, so it can be restarted at the same point.	N / A

(This table continues on the next page)

Context - Name	Vertical elasticity	QoS
Thesis - Vertical elasticity manager (Chapter 2)	An open-source mechanism has been developed to provide the ability to automate the modification of the vCPU share assigned to the jobs in two scenarios: iterative jobs and long-term jobs. The wrapper acts at job scheduler level, so using it does not force to modify the job scheduler.	The first scenario provides the way to run a certain number of independent iterations before a deadline. The mechanism will modify the vCPU assigned between the different iterations of the job. The second scenario relies on checkpointing techniques (transparent for the user) to manage the vCPU assigned to a long-term job (for example, an HPC job) to fulfil a QoS also expressed as a deadline.
Thesis - Data Analytic Architecture (Chapter 3)	N / A	The combination of a vertical mechanism and an elasticity manager provides this architecture with the ability to reconfigure both the nodes and the jobs in order to fulfil an agreed QoS.

Table 7.1: Comparison between the contributions of this thesis and related works for elasticity and QoS.

Context - Name	Infrastructure As Code	HPC / HTC	Multi-site	Multi-tenant
Thesis - Data Analytic Architecture (Chapter 3)	An elastic cloud architecture was designed. A set of recipes and Ansible roles were developed to implement it in the majority of cloud provider services.	N / A	N / A	The multi-tenancy is supported by the frameworks of Apache Mesos. In this work, only Marathon supports multi-tenancy.

(This table continues on the next page)

Context - Name	Infrastructure As Code	HPC / HTC	Multi-site	Multi-tenant
Related work - StarCluster [169]	A project that allows to create pre-configured virtual clusters in AWS. It has a plugin named Elastic Load Balancer that provides horizontal elasticity taking into account the number of jobs queued up at the job scheduler.	N / A	N / A	N / A
Related work - Galaxy Project [64]	It is a web platform that contains a large number of tools for bioinformatic analysis which can be deployed on public and on-premises Cloud offerings using CloudMan [30].	N / A	N / A	N / A
Related work - Cloud BioLinux [28]	A set of preconfigured virtual machine images for AWS, VirtualBox and Eucalyptus.	N / A	N / A	N / A
Related work - A cloud solution for medical image processing [120]	A cloud architecture for medical image analysis using ImageJ for Eucalyptus.	N / A	N / A	N / A
Thesis - Architecture for medical imaging processing (Chapter 4)	An elastic cloud architecture was designed. A set of recipes and Ansible roles were developed to implement it in the majority of cloud provider services. This architecture was transferred to a medical imaging company (QUIBIM).	N / A	N / A	The multi-tenancy is supported by the chosen job scheduler (Nomad).

(This table continues on the next page)

Context - Name	Infrastructure As Code	HPC / HTC	Multi-site	Multi-tenant
Thesis - hpc-connector (Chapter 5)	N / A	The experimentation of combining a cloud architecture based on Kubernetes in Spain with an HPC infrastructure (Prometheus Supercomputer) in the framework of a European project demonstrates that the developed tool is useful.	An open-source tool was developed to ease the combination of different computing sites that have incompatible job schedulers.	N / A
Related work - KubeFed	N / A	N / A	It is an open-source extension of K8s that was designed for propagating K8s objects to multiple k8s clusters.	N / A
Related work - Cluster API (CAPI)	N / A	N / A	N / A	It is an example of the cluster as a service tenancy model. A management cluster is in charge of manually deploy a complete k8s cluster for each tenant.
Related work - Virtual Cluster	N / A	N / A	N / A	It is an example of the Control plane as a service tenancy model. Each tenant has its own control plane, so the tenants do not share the K8s cluster but are running in the same computing infrastructure.

(This table continues on the next page)

Context - Name	Infrastructure As Code	HPC / HTC	Multi-site	Multi-tenant
Related work - Hierarchical Namespace Controller (HNC)	N / A	N / A	N / A	It is an example of the namespace as a service tenancy model. Each tenant has its own namespace. This K8s extension enables the creation of additional namespaces under parent namespaces propagating resources (roles, policies, etc.) within the hierarchy.
Thesis - kube-authorizer (Chapter 6)	N / A	N / A	N / A	An open-source tool was developed that automates the creation of namespaces and the RBAC policies to each tenant in a K8s cluster.
Thesis - General use architecture (Chapter 6)	An elastic cloud architecture was designed, and a set of recipes and Ansible roles were developed to implement it in the majority of cloud provider services. This architecture was transferred to a European Research Project.	N / A	The architecture described in this work corresponds to the one of their sites.	The multi-tenancy is achieved in multiple sites thanks to the use of EGI CheckIn for authentication and kube-authorizer for the automation of the authorization.

Table 7.2: Comparison between the principal contributions of this thesis and the related works and tools for cloud architectures.

7.2 Publications

The realization of this doctoral thesis has led to the publication of a series of research articles and publications at conferences that are detailed next:

National conferences

- Sergio López Huguet, David de Andrés Martínez, and J. Damián Segrelles Quilis. "Gestión elástica en la nube de recursos computacionales para actividades docentes: caso de uso en el Diseño de Sistemas Digitales". In: *Actas de las Jornadas sobre Enseñanza Universitaria de la Informática* 4.0 (2019). ISSN: 2531-0607. URL: `http://www.aenui.net/ojs/index.php?journal=actas_jenui&page=article&op=view&path%5B%5D=509`

International conferences

- Sergio López-Huguet et al. "A Cloud Architecture for the Execution of Medical Imaging Biomarkers". In: *Computational Science – ICCS 2019*. Ed. by João M F Rodrigues et al. Cham: Springer International Publishing, 2019, pp. 130–144. ISBN: 978-3-030-22744-9. CORE (2018): A, GII-GRIN-SCIE GGS (2018): Class 3.
- Sergio López-Huguet et al. "Seamlessly Managing HPC Workloads Through Kubernetes". In: *High Performance Computing*. Ed. by Heike Jagode et al. Cham: Springer International Publishing, 2020, pp. 310–320. ISBN: 978-3-030-59851-8. CORE (2018): C. GII-GRIN-SCIE GGS (2018): Work in progress.

Journal articles

- Sergio López-Huguet et al. "Vertical elasticity on Marathon and Chronos Mesos frameworks". In: *Journal of Parallel and Distributed Computing* 133 (2019), pp. 179 –192. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2019.01.002`. URL: `http://www.sciencedirect.com/science/article/pii/S0743731519300085`. Q2. Impact factor (2019): 2,296.
- Sergio López-Huguet et al. "A self-managed Mesos cluster for data analytics with QoS guarantees". In: *Future Generation Computer Systems* 96 (2019), pp. 449–461. ISSN: 0167-739X. DOI: `10.1016/j.future.2019.02.047`. URL: `http://www.sciencedirect.com/science/article/pii/S0167739X18311087`. Q1. Impact factor (2019): 6,125.
- Sergio López-Huguet et al. "Automated Isolation Management of Processing Workflows in Multi-Tenant and Multi-Site Kubernetes clusters - A Medical Imaging Use Case". Submitted to: *IEEE Access* (2021). Q1. Impact factor (2019): 3,745.

7.3 Research projects

The thesis has been performed within the framework of different projects sponsored by the university, the regional and national government, as well as by international organizations. In addition, the researcher has participated in a technology transfer project with a company.

The results obtained in this thesis have been reflected in the different projects that will be listed below, although the most important contributions have been made for PRIMAGE, CHAIMELEON, COVID and QUIBIM.

Funded by the university

- *Ejecución de aplicaciones sobre plataformas on-premises elásticas de funciones como servicio (FaaS).* Grant number: SP20180068. Duration: 01/01/19 - 01/01/20. Principal investigator (PI): Miguel Caballer Fernández. Funded by Universitat Politécnica de Valéncia.

National and regional research projects

- *Computación Big Data y de Altas Prestaciones Sobre Multi-Clouds Elásticos (BigCLOE).* Grant number: TIN2016-79951-R. Duration: 01/01/19 - 01/01/20. Principal investigator (PI): Germán Moltó and Ignacio Blanquer. Funded by Ministerio de Economía, Industria y Competitividad of Spain.
- *Bases de datos centralizadas y vinculadas a datos clínicos para soluciones de IA (Inteligencia Artificial) en epidemias tipo Covid-19.* Grant number: 54. Duration: 17/04/2020-16/01/2021. Principal investigator (PI): Dr. Luís Martí-Bonmatí. Funded by Agència Valenciana de la Innovació, Generalitat Valenciana of Spain.

International research projects

- *INtegrating Distributed data Infrastructures for Global ExplOitation. INDIGO-DataCloud.* Grant number: 653549. Duration: 01/04/15 - 01/10/17. Principal investigator (PI): Ignacio Blanquer. Funded by the European Commission.
- *Europe-Brazil Collaboration On Big Data Scientific Research Through Cloud-Centric Application.* Grant number: 690116. Duration: 01/01/16 - 01/01/18. Principal investigator (PI): Ignacio Blanquer. Funded by the European Commission.
- *Designing And Enabling E-Infrastructures For Intensive Processing In a Hybrid Datacloud (DEEP-HybridDataCloud).* Grant number: 777435.

Duration: 01/11/17 - 30/04/20. Principal investigator (PI): Ignacio Blanquer. Funded by the European Commission.

- *Predictive In-Silico Multiscale Analytics To Support Cancer Personalized Diagnosis And Prognosis, Empowered By Imaging Biomarkers (PRIMAGE).* Grant number: 826494. Duration: 01/12/18 - 01/12/22. Principal investigator (PI): Ignacio Blanquer. Funded by the European Commission.

- *Accelerating The Lab To Market Transition Of Ai Tools For Cancer Management (CHAIMELEON).* Grant number: 952172. Duration: 01/09/20 - 01/09/24. Principal investigator (PI): Ignacio Blanquer. Funded by the European Commission.

Technology transfer research projects

- *Research, Development and Implementation of High Performance Computing (HPC) Techniques to Quibim Precision.* Duration: 14/12/2017 - 13/12/2019. Principal Investigator (PI): Ignacio Blanquer. Funded by QUIBIM S.L.

Chapter 8

Conclusions

Nowadays, public and private cloud platforms have become a very interesting option to store and process data on demand. The thesis is focused on the design and implementation of elastic and container-based cloud architectures for high-performance computing.

Firstly, the research was focused on the elasticity. One the one hand, a proof of concept mechanism was developed to provide vertical elasticity on top of an elastic infrastructure for data analytics using Docker containers and Apache Mesos. The mechanism triggers elasticity actions depending on the quality of service (expressed as a deadline) and, by using Apache Mesos frameworks, it modifies the CPU assigned to the application. Moreover, the tool considers two types of executions: applications whose progress cannot be monitored, and applications that perform a known number of independent iterations. On the other hand, horizontal elasticity is obtained by using the tools developed in the research group (IM, EC3 and CLUES).

Another point addressed during this thesis is the design of different container-based cloud architectures. The first architecture presented in this thesis was created to provide an infrastructure for data analytics. Chapter 4 describes a cloud architecture to process medical imaging data. This architecture is an evolution of the first architecture that considers the creation and test of the Docker containers that include medical imaging applications. It should be noted that this architecture is being used in the business environment, concretely at the QUIBIM company.

The combination of existent processing platforms could be complex due to there are a lot of aspects to consider: authentication methods, shared storage between infrastructures, different job schedulers, or even different container technologies. During three-months research stay in Krakow (Poland), the author implemented a tool named *hpc-connector* to ease the connection of different processing infrastructures by adding an extra layer (the own *hpc-connector*) that replicates the job state in one architecture. The tool was implemented to run without requiring special privileges (it runs as an usual job). The author demonstrates that *hpc-connector* allows to interconnect multiple processing platforms (in Chapter 5, a Kubernetes cluster was combined with Prometheus supercomputer); but without sharing any storage between the infrastructures.

Regarding the authorization of federated infrastructures, the author developed a tool named *kube-authorizer* that is able to manage the authorization policies in a single Kubernetes cluster. This tool was designed to work behind of a proxy but can work without it. The aim of the proxy is to authenticate users and to trigger the creation of the security policies using *kube-authorizer*. This ecosystem was successfully deployed in production mode in the framework of the European project PRIMAGE. In the context of this project, the use of both *kube-authorizer* and the proxy enables the access by using OpenID Connect (OIDC) in the distributed storage solution (Onedata) and in the cloud processing platform (Kubernetes cluster deployed at the UPV).

It should be pointed out that all services, tools and configuration recipes developed during this thesis are open-source.

With the results obtained in this thesis, we conclude stating that all objectives proposed at the beginning of the research have been successfully achieved. The works resulting from this thesis are being used both in the business environment (QUIBIM) and in the academia (involved in international and national research projects).

During the research phase of this thesis, the author used a wide variety of trending and open-source technologies to implement the architectures designed. Regarding resource management tools and job schedulers, Apache Mesos, Hashicorp Nomad, Kubernetes, and Slurm were used depending on the use case. The container technologies utilized were Docker and Singularity. Furthermore, the author employed the most popular cloud providers such as Amazon Web Services, Microsoft Azure, Google Cloud, OpenStack, and OpenNebula.

8.1 Future work

Future work involves the design of different architectures or the improvement of the cloud architectures presented during this thesis to address the requirements of the next research projects. The author is being involved in several research projects to provide a processing infrastructure to create AI applications. One of the related projects in which this thesis contributions are utilized and will be thus improved and extended is a national project focused on providing an infrastructure to study COVID patients in the Comunidad Valenciana (Spain). Other international projects that the author is involved in, as well as its dissertation developments, consist of providing infrastructures to create AI models to study cancer diseases, concretely lung emphysema and Diffuse Intrinsic Pontine Glioma (DIPG). The access to a sensible information causes that cloud architectures must consider who uses the data to build AI models or how is the process to obtain the data (if it is stored in other computing infrastructure). Furthermore, it is necessary to identify multi-tenant storage solutions that provide a POSIX to a distributed storage.

Designing cloud architectures also involves considering other technologies. Resource managers, job schedulers and container technologies are constantly evolving. Docker is the most popular container technology but other technologies can replace it in few years. Podman is a container technology that allows to run OCI containers without a privileged daemon (unlike Docker). In contrast to Singularity, Podman is presented not as an HPC container solution, it is designed to be a container technology for general use. Podman uses the same container runtime as Kubernetes and it seems that it will replace Docker in Kubernetes in a few years. However, Docker recently presented rootless Docker, an alternative to regular user that enables Docker containers running in user namespaces (as Singularity or Podman do). Thus, a future research field could consist on adapting the research tools developed by the author and the tools that are maintained by the research group to take profit of the advantages of using unprivileged containers.

Bibliography

[1] *2019 Cloud Native Computing Foundation survey.* https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf. Last accessed: May 2021 (cit. on p. 112).

[2] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. "Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER". In: *IEEE International Conference on Cloud Computing, CLOUD.* Vol. 2017-June. 2017, pp. 472–479. ISBN: 9781538619933. DOI: 10.1109/CLOUD.2017.67 (cit. on pp. 20, 57, 143).

[3] Amazon. *Amazon EC2.* https://aws.amazon.com/ec2/. [Online; accessed April-2018] (cit. on pp. 53, 83).

[4] Amazon. *AWS Auto Scaling.* https://aws.amazon.com/es/autoscaling/. [Online; accessed March-2018] (cit. on pp. 20, 55).

[5] Amazon. *CloudFormation.* https://aws.amazon.com/cloudformation/. [Online; accessed April-2018] (cit. on p. 52).

[6] *Amazon Elastic Kubernetes Service (EKS).* `https://aws.amazon.com/es/eks/`. Last accessed: May 2021 (cit. on p. 112).

[7] Chris Anderson. *The End of Theory: The Data Deluge Makes the Scientific Method Obsolete.* `http://www.wired.com/2008/06/pb-theory/`. [Online; accessed April-2018] (cit. on p. 53).

[8] Apache Mesos. *Mesos frameworks.* `http://mesos.apache.org/documentation/latest/frameworks/`. [Online; accessed May-2018]. 2017 (cit. on p. 22).

[9] *App Container Specification.* `https://github.com/appc/spec/`. Last accessed: Oct 2020 (cit. on p. 7).

[10] *Azure for health.* `https://azure.microsoft.com/en-us/industries/healthcare/#security`. Last accessed: 07 May 2020 (cit. on p. 104).

[11] *Azure Kubernetes Service (AKS).* `https://azure.microsoft.com/es-es/topic/what-is-kubernetes/#azure-kubernetes-service`. Last accessed: May 2021 (cit. on p. 112).

[12] Barik, R.K. and Lenka, R.K. and Rao, K.R. and Ghose, D. "Performance analysis of virtual machines and containers in cloud computing". In: *Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2016* (2017). DOI: `{10.1109/CCAA.2016.7813925}` (cit. on p. 15).

[13] Sushmita Bhatnagar. "An Audit of Malignant Solid Tumors in Infants and Neonates". In: *Journal of neonatal surgery* 1 (Jan. 2012), p. 5 (cit. on p. 106).

[14] Ignacio Blanquer, Goetz Brasche, and Daniele Lezzi. "Requirements of scientific applications in cloud offerings". In: *Proceedings of the 2012*

Sixth Iberian Grid Infrastructure Conference, IBERGRID. Vol. 12. 2012, pp. 173–182 (cit. on p. 5).

[15] Miguel Caballer, Ignacio Blanquer, Germán Moltó, and Carlos de Alfonso. "Dynamic Management of Virtual Infrastructures". In: *Journal of Grid Computing* 13.1 (2015), pp. 53–70. ISSN: 15729184. DOI: `10.1007/s10723-014-9296-5` (cit. on pp. 36, 53, 86, 92, 126).

[16] Luis Cabellos, Isabel Campos, Enol Fernández-Del-Castillo, Michał Owsiak, Bartek Palak, and Marcin Płóciennik. "Scientific workflow orchestration interoperating HTC and HPC resources". In: *Computer Physics Communications* (2011). ISSN: 00104655. DOI: `10.1016/j.cpc.2010.12.020` (cit. on p. 99).

[17] Amanda Calatrava, Eloy Romero, Germán Moltó, Miguel Caballer, and Jose Miguel Alonso. "Self-managed cost-efficient virtual elastic clusters on hybrid Cloud infrastructures". In: *Future Generation Computer Systems* 61 (2016), pp. 13–25. ISSN: 0167739X. DOI: `10.1016/j.future.2016.01.018` (cit. on pp. 56, 86).

[18] Callaghan, Scott and Maechling, Philip and Small, Patrick and Milner, Kevin and Juve, Gideon et. al. "Metrics for heterogeneous scientific workflows: A case study of an earthquake science application". In: *International Journal of High Performance Computing Applications*. 2011. DOI: `10.1177/1094342011414743` (cit. on p. 99).

[19] Enol Fernández del Castillo, Diego Scardaci, and Ãlvaro López García. "The EGI Federated Cloud e-Infrastructure". In: *Procedia Computer Science* 68 (2015). 1st International Conference on Cloud Forward: From Distributed to Complete Computing, pp. 196–205. ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2015.09.235`. URL: `https://www.sciencedirect.com/science/article/pii/S187705091503080X` (cit. on p. 112).

[20] *Cgroups man page.* https://man7.org/linux/man-pages/man7/cgroups.7.html. Last accessed: Oct 2020 (cit. on p. 8).

[21] Nikolaos Chalvantzis, Ioannis Konstantinou, and Nektarios Kozyris. "BBQ: Elastic MapReduce over cloud platforms". In: *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017.* 2017, pp. 766–771. ISBN: 9781509066100. DOI: 10.1109/CCGRID.2017.140 (cit. on pp. 56, 143).

[22] Chen, Shanyu and He, Zhipeng and Han, Xinyin and He, Xiaoyu et. al. *How Big Data and High-performance Computing Drive Brain Science.* 2019. DOI: 10.1016/j.gpb.2019.09.003 (cit. on p. 103).

[23] Jieun Choi, Younsun Ahn, S. Kim, Y. Kim, and Jaeyoung Choi. "VM auto-scaling methods for high throughput computing on hybrid infrastructure". In: *Cluster Computing* 18 (2015), pp. 1063–1073 (cit. on p. 7).

[24] Chronos. *Chronos REST API documentation.* https://mesos.github.io/chronos/docs/api.html. [Online; accessed July-2017]. 2017 (cit. on p. 36).

[25] Chronos. *Chronos website.* https://mesos.github.io/chronos/. [Online; accessed July-2017]. 2017 (cit. on pp. 14, 54, 86).

[26] *Chroot man page.* https://man7.org/linux/man-pages/man8/pivot_root.8.html. Last accessed: Oct 2020 (cit. on p. 8).

[27] *Cloud Access to Mammograms Enables Earlier Breast Cancer Detection.* https://www.itnonline.com/content/cloud-access-mammograms-enables-earlier-breast-cancer-detection. Last accessed: 07 May 2020 (cit. on p. 104).

[28] *CloudBioLinux web site*. http://cloudbiolinux.org/. Accessed: 29-12-2018 (cit. on pp. 85, 145).

[29] Cloudify. *Cloudify*. https://cloudify.co. [Online; accessed April-2018] (cit. on p. 53).

[30] *CloudMan web site*. https://galaxyproject.org/cloudman. Accessed: 29-12-2018 (cit. on pp. 83, 145).

[31] *Containerd website*. https://containerd.io/. Last accessed: Oct 2020 (cit. on p. 8).

[32] Containers. *crun Github repository*. https://github.com/containers/crun. Last accessed: Oct 2020 (cit. on p. 8).

[33] CoreOS. *rkt website*. https://coreos.com/rkt/. Last accessed: Oct 2020 (cit. on p. 8).

[34] Poland Cyfronet. Krakow. *Prometheus Supercomputer*. http://www.cyfronet.krakow.pl/computers/15226,artykul,prometheus.html. Last accessed: 07 May 2020 (cit. on pp. 11, 101, 106).

[35] DC/OS. *Autoscaling with Marathon*. https://dcos.io/docs/1.7/usage/tutorials/autoscaling/. [Online; accessed April-2018] (cit. on pp. 58, 143).

[36] Carlos De Alfonso, Miguel Caballer, Fernando Alvarruiz, and Vicente Hernández. "An energy management system for cluster infrastructures". In: *Computers and Electrical Engineering*. Vol. 39. 8. 2013, pp. 2579–2590. DOI: 10.1016/j.compeleceng.2013.05.004 (cit. on pp. 7, 56, 92, 126).

[37] J Dean and S Ghemawat. "Simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113. ISSN: 00010782. DOI: 10.1145/1327452.1327492. arXiv: 10.1.1.163.5292 (cit. on p. 53).

[38] Deis. *Deis*. `https://deis.com/`. [Online; accessed April-2018] (cit. on p. 54).

[39] Docker. *Docker rootless mode*. `https://docs.docker.com/engine/security/rootless/`. Last accessed: Oct 2020 (cit. on pp. 8, 116).

[40] *Docker Engine release notes version 0.1.0*. `https://docs.docker.com/engine/release-notes/prior-releases/#010-2013-03-23`. Last accessed: May 2021 (cit. on p. 112).

[41] *Docker Swarm*. `https://docs.docker.com/engine/swarm/`. Last accessed: January 2021 (cit. on p. 9).

[42] *Documentation of FreeBSD Jails*. `https://www.freebsd.org/doc/handbook/jails.html`. Last accessed: Oct 2020 (cit. on p. 7).

[43] Łukasz Dutka, Michal Wrzeszcz, Tomasz Lichoń, Rafał Słota, Konrad Zemek, Krzysztof Trzepla, Łukasz Opioła, Renata Słota, and Jacek Kitowski. "Onedata - A Step Forward towards Globalization of Data Access for Computing Infrastructures". In: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015, pp. 2843–2847. ISSN: 1877-0509. DOI: `10.1016/j.procs.2015.05.445` (cit. on pp. 98, 128).

[44] *Elad Benjamin of Philips on Using AWS to Help Improve Healthcare*. `https://aws.amazon.com/es/solutions/case-studies/philips-reinvent/?did=cr_card&trk=cr_card`. Last accessed: April 2021 (cit. on p. 5).

[45] EUBra-BIGSEA. *Deliverable D3.4: QoS infrastructure services intermediate version*. `https://www.eubra-bigsea.eu/sites/default/files/D3.4%20EUBra-BIGSEA%20QoS%20infrastructure%20services.pdf`. [Online; accessed April-2018] (cit. on p. 57).

[46] EUBra-BIGSEA. *Deliverable D7.1: End-User Requirements Elicitation.* `http://www.eubra-bigsea.eu/sites/default/files/D7_1_End-User_Requirements_Final.pdf`. [Online; accessed April-2018] (cit. on p. 51).

[47] EUBra-BigSEA. *EUBra-BigSEA website.* `http://www.eubra-bigsea.eu`. [Online; accessed July-2017]. 2017 (cit. on p. 36).

[48] *Eucalyptus web site.* `https://www.eucalyptus.cloud/`. Accessed: 29-12-2018 (cit. on p. 85).

[49] European Society of Radiology (ESR). "White paper on imaging biomarkers". In: *Insights into Imaging* 1.2 (2010), pp. 42–45. ISSN: 1869-4101. DOI: `10.1007/s13244-010-0025-8` (cit. on p. 82).

[50] *Fakechroot Github repository.* `https://github.com/dex4er/fakechroot`. Last accessed: Oct 2020 (cit. on p. 9).

[51] Soodeh Farokhi, Pooyan Jamshidi, Ewnetu Bayuh Lakew, Ivona Brandic, and Erik Elmroth. "A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach". In: *Future Generation Computer Systems* 65 (2016), pp. 57–72. ISSN: 0167739X. DOI: `10.1016/j.future.2016.05.028` (cit. on pp. 19, 20, 57).

[52] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. "An updated performance comparison of virtual machines and Linux containers". In: *ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software.* 2015. ISBN: 9781479919567. DOI: `10.1109/ISPASS.2015.7095802` (cit. on pp. 7, 15, 54).

[53] Fernandes, João, Jones, Bob, Yakubov, Sergey, and Chierici, Andrea. "HNSciCloud, a Hybrid Cloud for Science". In: *EPJ Web Conf.* 214 (2019),

p. 09006. DOI: 10.1051/epjconf/201921409006. URL: https://doi.org/10.1051/epjconf/201921409006 (cit. on p. 5).

[54] Apache Software Foundation. *Apache Kafka*. https://kafka.apache.org/. [Online; accessed April-2018] (cit. on p. 59).

[55] Apache Software Foundation. *Apache Storm*. http://storm.apache.org/. [Online; accessed April-2018] (cit. on p. 59).

[56] Cloud Native Computing Foundation. *CRIO-O website*. https://cri-o.io/. Last accessed: Oct 2020 (cit. on p. 8).

[57] OpenStack Foundation. *katacontainers website*. https://katacontainers.io/. Last accessed: Oct 2020 (cit. on p. 8).

[58] The Apache Software Foundation. *Apache ARIA TOSCA ORCHESTRATION ENGINE*. http://ariatosca.incubator.apache.org. [Online; accessed April-2018] (cit. on p. 53).

[59] The Apache Software Foundation. *Apache Hadoop*. http://hadoop.apache.org. [Online; accessed April-2018] (cit. on pp. 6, 10, 54).

[60] The Apache Software Foundation. *Apache Hive*. http://hive.apache.org. [Online; accessed April-2018] (cit. on pp. 6, 54).

[61] The Apache Software Foundation. *Apache Pig*. http://pig.apache.org. [Online; accessed April-2018] (cit. on p. 54).

[62] The Apache Software Foundation. *Apache Spark*. http://spark.apache.org. [Online; accessed April-2018] (cit. on pp. 6, 10, 54).

[63] *GAIA X website*. https://www.data-infrastructure.eu/GAIAX/Navigation/EN/Home/home.html. Last accessed: April 2021 (cit. on p. 5).

[64] *Galaxy Platform web site.* https://galaxyproject.org. Accessed: 29-12-2018 (cit. on pp. 6, 83, 145).

[65] *Getting to the heart of the HPC and AI the edge in healthcare.* https://www.nextplatform.com/2018/03/28/getting-to-the-heart-of-hpc-and-ai-at-the-edge-in-healthcare/. Last accessed: 07 May 2020 (cit. on p. 104).

[66] I. Giannakopoulos, N. Papailiou, C. Mantas, I. Konstantinou, D. Tsoumakos, and N. Koziris. “CELAR: Automated application elasticity platform”. In: *2014 IEEE International Conference on Big Data (Big Data).* 2014, pp. 23–25. DOI: 10.1109/BigData.2014.7004481 (cit. on p. 53).

[67] J. Gomes, E. Bagnaschi, I. Campos, M. David, L. Alves, J. Martins, J. Pina, Á. López-García, and P. Orviz. “Enabling rootless Linux Containers in multi-user environments: The udocker tool”. In: *Computer Physics Communications* 232 (2018), pp. 84 –97. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2018.05.021. URL: http://www.sciencedirect.com/science/article/pii/S0010465518302042 (cit. on pp. 8, 117).

[68] Google. *Google Cloud.* https://cloud.google.com/. [Online; accessed April-2018] (cit. on p. 59).

[69] *Google Kubernetes Engine (GKE).* https://cloud.google.com/kubernetes-engine. Last accessed: May 2021 (cit. on p. 112).

[70] Carlos A.S.J. Gulo, Antonio C. Sementille, and João Manuel R.S. Tavares. *Techniques of medical image processing and analysis accelerated by high-performance computing: a systematic literature review.* 2019. DOI: 10.1007/s11554-017-0734-z (cit. on p. 103).

[71] *Helm.* https://helm.sh/. Last accessed: January 2021 (cit. on p. 9).

[72] Carl Hewitt, Peter Bishop, and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence.* IJCAI’73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, 235â245 (cit. on p. 6).

[73] *High Performance Computing and deep learning in medicine: Enhancing physicians, helping patients.* `https://ec.europa.eu/digital-single-market/en/news/high-performance-computing-and-deep-learning-medicine-enhancing-physicians-helping-patients`. Last accessed: 07 May 2020 (cit. on p. 103).

[74] Benjamin Hindman, Andy Konwinski, A Platform, Fine-Grained Resource, and Matei Zaharia. “Mesos: A platform for fine-grained resource sharing in the data center”. In: *Proceedings of the ...* (2011), p. 32. ISSN: 00189456. DOI: `10.1109/TIM.2009.2038002`. URL: `http://static.usenix.org/events/nsdi11/tech/full_papers/Hindman_new.pdf` (cit. on p. 22).

[75] *How Mahou San Miguel Slashes IT Costs with AWS.* `https://aws.amazon.com/es/solutions/case-studies/mahou/`. Last accessed: April 2021 (cit. on p. 5).

[76] Sergio López Huguet, David de Andrés Martínez, and J. Damián Segrelles Quilis. “Gestión elástica en la nube de recursos computacionales para actividades docentes: caso de uso en el Diseño de Sistemas Digitales”. In: *Actas de las Jornadas sobre Enseñanza Universitaria de la Informática* 4.0 (2019). ISSN: 2531-0607. URL: `http://www.aenui.net/ojs/index.php?journal=actas_jenui&page=article&op=view&path%5B%5D=509` (cit. on pp. 142, 148).

[77] Tassadaq Hussain, Amna Haider, Muhammad Shafique, and Abdelmalik taleb ahmed. “A High-Performance System Architecture for Medical

Imaging”. In: July 2019. ISBN: 978-1-83881-188-4. DOI: 10 . 5772 / intechopen.83581 (cit. on p. 104).

[78] Hyungro Lee. *Using Bioinformatics Applications on the Cloud.* http://dsc.soic.indiana.edu/publications/bioinformatics.pdf. Online; accessed 29 December 2018. 2013 (cit. on pp. 6, 83, 115).

[79] *Imagej web site.* https://imagej.nih.gov/ij/. Accessed: 29-12-2018 (cit. on p. 85).

[80] Inc InfluxData. *InfluxDB.* https://www.influxdata.com/. [Online; accessed April-2018] (cit. on p. 59).

[81] Open Container Initiative. *runC Github repository.* https://github.com/opencontainers/runc. Last accessed: Oct 2020 (cit. on p. 8).

[82] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks”. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007.* EuroSys ’07. Lisbon, Portugal: Association for Computing Machinery, 2007, 59â72. ISBN: 9781595936363. DOI: 10.1145/1272996.1273005. URL: https://doi.org/10.1145/1272996.1273005 (cit. on p. 6).

[83] Desislava Ivanova, Plamenka Borovska, and Stefan Zahov. “Development of PaaS using AWS and Terraform for medical imaging analytics”. In: *AIP Conference Proceedings.* 2018. ISBN: 9780735417748. DOI: 10.1063/1.5082133 (cit. on p. 104).

[84] Jack J. Dongarra and Piotr Luszczek and Antoine Petitet. “The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience”. In: *Concurrency and Computation: Practice and Experience* 15 (2003), 2003 (cit. on p. 36).

[85] Saba Jamalian and Hassan Rajaei. "Data-Intensive HPC Tasks Scheduling with SDN to Enable HPC-as-a-Service". In: *Proceedings - 2015 IEEE 8th International Conference on Cloud Computing, CLOUD 2015*. Institute of Electrical and Electronics Engineers Inc., 2015, pp. 596–603. ISBN: 9781467372879. DOI: `10.1109/CLOUD.2015.85` (cit. on p. 101).

[86] John Burkardt. *Sequential and OpenMP versions of Linpack Solver.* `https://people.sc.fsu.edu/~jburkardt/c_src/sgefa_openmp/sgefa_openmp.html`. [Online; accessed July-2017]. 2017 (cit. on p. 36).

[87] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in Dataflow Programming Languages". In: *ACM Comput. Surv.* 36.1 (Mar. 2004), 1â34. ISSN: 0360-0300. DOI: `10.1145/1013208.1013209`. URL: `https://doi.org/10.1145/1013208.1013209` (cit. on p. 6).

[88] Hao Yun Kao, Wen Hsiung Wu, Tyng Yeu Liang, King The Lee, Ming Feng Hou, Hon Yi Shi, and Chih Pin Chuu. "Cloud-based service information system for evaluating quality of life after breast cancer surgery". In: *PLoS ONE* (2015). ISSN: 19326203. DOI: `10.1371/journal.pone.0139252` (cit. on p. 104).

[89] Laszlo Kovacs, Roland Kovacs, and Andras Hajdu. "High Performance Computing in Medical Image Analysis HuSSaR". In: (2018). arXiv: `1806.06171`. URL: `\url{http://arxiv.org/abs/1806.06171}` (cit. on p. 104).

[90] P. Kranas, V. Anagnostopoulos, A. Menychtas, and T. Varvarigou. "ElaaS: An Innovative Elasticity as a Service Framework for Dynamic Management across the Cloud Stack Layers". In: *2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*. 2012, pp. 1042–1049 (cit. on p. 7).

[91] Dieter Kranzlmüller, J Marco de Lucas, and P Öster. “The european grid initiative (EGI)”. In: *Remote instrumentation and virtual laboratories.* Springer, 2010, pp. 61–66 (cit. on p. 112).

[92] *Kubernetes Ingress Controller.* `https://kubernetes.io/docs/concepts/services-networking/ingress/`. Last accessed: January 2021 (cit. on p. 9).

[93] *Kubernetes Pod Autoscaler.* `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`. Last accessed: January 2021 (cit. on p. 9).

[94] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. “Singularity: Scientific containers for mobility of compute”. In: *PLOS ONE* 12.5 (May 2017), pp. 1–20. DOI: `10.1371/journal.pone.0177459`. URL: `https://doi.org/10.1371/journal.pone.0177459` (cit. on pp. 2, 8, 54, 85, 102, 116).

[95] Los Alamos National Laboratory. *CharlieCloud Github repository.* `https://hpc.github.io/charliecloud/`. Last accessed: Oct 2020 (cit. on p. 8).

[96] Grafana Labs. *Grafana.* `https://grafana.com/`. [Online; accessed April-2018] (cit. on p. 59).

[97] B. Langmead and Abhinav Nellore. “Cloud computing for genomic data analysis and collaboration”. In: *Nature Reviews Genetics* 19 (2018), pp. 325–325 (cit. on p. 112).

[98] *Linux Capabilities man page.* `https://man7.org/linux/man-pages/man7/capabilities.7.html`. Last accessed: Oct 2020 (cit. on p. 8).

[99] *Linux Crontab manual website.* `http://man7.org/linux/man-pages/man5/crontab.5.html`. [Online; accessed September-2018]. 2017 (cit. on p. 14).

[100] Sergio López-Huguet, Fabio García-Castro, Ángel Alberich-Bayarri, and Ignacio Blanquer. “A Cloud Architecture for the Execution of Medical Imaging Biomarkers”. In: *Computational Science – ICCS 2019*. Ed. by João M F Rodrigues, Pedro J S Cardoso, Jânio Monteiro, Roberto Lam, Valeria V Krzhizhanovskaya, Michael H Lees, Jack J Dongarra, and Peter M A Sloot. Cham: Springer International Publishing, 2019, pp. 130–144. ISBN: 978-3-030-22744-9 (cit. on pp. 11, 81, 101, 116, 148).

[101] Sergio López-Huguet, Igor Natanael, Andrey Brito, and Ignacio Blanquer. “Vertical elasticity on Marathon and Chronos Mesos frameworks”. In: *Journal of Parallel and Distributed Computing* 133 (2019), pp. 179 –192. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.01.002. URL: http://www.sciencedirect.com/science/article/pii/S0743731519300085 (cit. on pp. 10, 13, 149).

[102] Sergio López-Huguet, Alfonso Pérez, Amanda Calatrava, Carlos de Alfonso, Miguel Caballer, Germán Moltó, and Ignacio Blanquer. “A self-managed Mesos cluster for data analytics with QoS guarantees”. In: *Future Generation Computer Systems* 96 (2019), pp. 449–461. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.02.047. URL: http://www.sciencedirect.com/science/article/pii/S0167739X18311087 (cit. on pp. 11, 49, 101, 149).

[103] Sergio López-Huguet, J. Damià Segrelles, Marek Kasztelnik, Marian Bubak, and Ignacio Blanquer. “Seamlessly Managing HPC Workloads Through Kubernetes”. In: *High Performance Computing*. Ed. by Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief. Cham: Springer International Publishing, 2020, pp. 310–320. ISBN: 978-3-030-59851-8 (cit. on pp. 11, 99, 148).

[104] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. “A Review of Auto-scaling Techniques for Elastic Applications in Cloud

Environments". In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592. ISSN: 15707873. DOI: 10.1007/s10723-014-9314-7 (cit. on p. 14).

[105] Linux Containers (LXC). *Linux Containers website*. https://linuxcontainers.org/. [Online; accessed July-2017]. 2008 (cit. on pp. 7, 21, 54, 85).

[106] *LxD documentation*. https://lxd.readthedocs.io/. Accessed: 29-12-2018 (cit. on pp. 7, 86).

[107] Carlo Manuali, Alessandro Costantini, Antonio Laganà, Marco Cecchi, Antonia Ghiselli, Michele Carpené, and Elda Rossi. "Efficient workload distribution bridging HTC and HPC in scientific computing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2012. ISBN: 9783642311246. DOI: 10.1007/978-3-642-31125-3_27 (cit. on p. 99).

[108] Marathon. *Marathon REST API documentation*. https://mesosphere.github.io/marathon/docs/rest-api.html. [Online; accessed July-2017]. 2017 (cit. on p. 36).

[109] Marathon. *Marathon website*. http://mesosphere.github.io/marathon/. [Online; accessed July-2017]. 2017 (cit. on pp. 14, 54, 86).

[110] Paul Marshall, Kate Keahey, and Tim Freeman. "Elastic site: Using clouds to elastically extend site resources". In: *CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*. 2010, pp. 43–52. ISBN: 9781424469871. DOI: 10.1109/CCGRID.2010.80 (cit. on p. 55).

[111] Paul Marshall, Henry Tufo, Kate Keahey, David La Bissoniere, and Matthew Woitaszek. "Architecting a large-scale elastic environment: Recontextualization and adaptive cloud services for scientific computing". In: *ICSOFT 2012 - Proceedings of the 7th International Conference on*

Software Paradigm Trends. 2012, pp. 409–418. ISBN: 9789898565198. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-84868693626{\&}partnerID=tZOtx3y1 (cit. on p. 55).

[112] Luis Martí-Bonmatí and Angel Alberich-Bayarri. *Imaging biomarkers: Development and clinical integration*. Springer-Verlag GmbH, 2016. ISBN: 9783319435046. DOI: 10.1007/978-3-319-43504-6 (cit. on p. 82).

[113] Luis Martí-Bonmatí, Ángel Alberich-Bayarri, Ruth Ladenstein, Ignacio Blanquer, and et. al. Segrelles J. Damian. "PRIMAGE project: predictive in silico multiscale analytics to support childhood cancer personalised evaluation empowered by imaging biomarkers". In: *European radiology experimental* (2020). ISSN: 25099280. DOI: 10.1186/s41747-020-00150-9 (cit. on pp. 101, 106).

[114] Martí-Bonmatí, L., García-Martí, G., Alberich-Bayarri, A., Sanz-Requena, R.. QUIBIM SL. *Método de segmentación por umbral adaptativo variable para la obtención de valores de referencia del aire corte a corte en estudios de imagen por tomografía computarizada*. ES 2530424B1, 02 September 2013 (cit. on p. 94).

[115] Mbarek Marwan, Ali Kartit, and Hassan Ouahmane. "Using cloud solution for medical image processing: Issues and implementation efforts". In: *2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*. IEEE, 2017. DOI: 10.1109/cloudtech.2017.8284703. URL: \url{http://dx.doi.org/10.1109/CloudTech.2017.8284703} (cit. on pp. 6, 82).

[116] *Medical Imaging Gets an AI Boost*. https://www.hpcwire.com/2019/12/03/medical-imaging-gets-an-ai-boost/. Last accessed: 07 May 2020 (cit. on p. 103).

[117] Apache Mesos. *Apache Mesos website*. `http://mesos.apache.org/`. [Online; accessed July-2017]. 2017 (cit. on pp. 10, 13, 54, 86).

[118] Microsoft. *Azure*. `https://azure.microsoft.com`. [Online; accessed April-2018] (cit. on p. 59).

[119] Miguel Caballer, Carlos De Alfonso, Fernando Alvarruiz and Germán Moltó. “EC3: Elastic Cloud Computing Cluster”. In: *Journal of Computer and System Sciences* 79.8 (Dec. 2013), pp. 1341–1351. ISSN: 0022-0000. DOI: `10.1016/j.jcss.2013.06.005`. URL: `http://dx.doi.org/10.1016/j.jcss.2013.06.005` (cit. on pp. 6, 36, 112, 126).

[120] Ali Mirarab, Najmeh Ghasemi Fard, and Mahboubeh Shamsi. “A cloud solution for medical image processing”. In: *Int. Journal of Engineering Research and Applications* 4.7 (2014), pp. 74–82. ISSN: 2248-9622 (cit. on pp. 85, 116, 145).

[121] Germán Moltó, Miguel Caballer, and Carlos de Alfonso. “Automatic Memory-based Vertical Elasticity and Oversubscription on Cloud Platforms”. In: *Future Gener. Comput. Syst.* 56.C (Mar. 2016), pp. 1–10. ISSN: 0167-739X. DOI: `10.1016/j.future.2015.10.002`. URL: `https://doi.org/10.1016/j.future.2015.10.002` (cit. on pp. 20, 57).

[122] Ruben S. Montero, Rafael Moreno-Vozmediano, and Ignacio M. Llorente. “An elasticity model for High Throughput Computing clusters”. In: *Journal of Parallel and Distributed Computing* 71.6 (2011). Special Issue on Cloud Computing, pp. 750 –757. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2010.05.005`. URL: `http://www.sciencedirect.com/science/article/pii/S0743731510000985` (cit. on p. 7).

[123] R. Moreno-Vozmediano, R. Montero, and I. Llorente. “Elastic management of cluster-based services in the cloud”. In: *ACDC ’09*. 2009 (cit. on p. 7).

[124] *Namespaces man page.* https://man7.org/linux/man-pages/man7/namespaces.7.html. Last accessed: Oct 2020 (cit. on p. 8).

[125] National Energy Research Scientific Computing Center (NERSC). *Shifter: User Defined Images.* https://www.nersc.gov/research-and-development/user-defined-images/. Last accessed: Oct 2020 (cit. on pp. 8, 116).

[126] *Nomad web site.* https://www.nomadproject.io/. Accessed: 29-12-2018 (cit. on pp. 10, 86).

[127] *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA).* https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca. [Online; accessed April-2018] (cit. on p. 52).

[128] *Open Container Initiative.* https://opencontainers.org/. Last accessed: Oct 2020 (cit. on p. 7).

[129] *Open Container Initiative image specification GitHub.* https://github.com/opencontainers/image-spec. Last accessed: Oct 2020 (cit. on p. 8).

[130] *Open Container Initiative runtime specification GitHub.* https://github.com/opencontainers/runtime-spec. Last accessed: Oct 2020 (cit. on p. 8).

[131] *OpenFaas web site.* https://www.openfaas.com/. Accessed: 29-12-2018 (cit. on p. 98).

[132] OpenMP. *OpenMP website.* http://www.openmp.org/. [Online; accessed July-2017]. 2017 (cit. on p. 36).

[133] OpenNebula. *OneFlow.* http://docs.opennebula.org/5.2/advanced_components/application_flow_and_auto-scaling/index.html. [Online; accessed April-2018] (cit. on p. 52).

[134] OpenStack. *Github of Monasca Agent.* https://github.com/openstack/monasca-agent. [Online; accessed July-2017]. 2015 (cit. on p. 23).

[135] OpenStack. *Heat Orchestration Template (HOT) Guide.* https://docs.openstack.org/heat/latest/template_guide/hot_guide.html. [Online; accessed April-2018] (cit. on p. 52).

[136] OpenStack. *Monasca Plugins.* https://github.com/openstack/monasca-agent/blob/master/docs/Plugins.md. [Online; accessed July-2017]. 2015 (cit. on p. 23).

[137] OpenStack. *OpenNebula Project.* https://opennebula.org/. [Online; accessed July-2017]. 2005 (cit. on pp. 35, 52).

[138] OpenStack. *Openstack Heat.* http://wiki.openstack.org/wiki/Heat. [Online; accessed April-2018] (cit. on p. 52).

[139] OpenStack. *OpenStack KeyStone website.* https://wiki.openstack.org/wiki/Keystone. [Online; accessed July-2017]. 2014 (cit. on pp. 23, 59).

[140] OpenStack. *OpenStack Monasca REST API.* https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md. [Online; accessed July-2017]. 2014 (cit. on pp. 23, 36).

[141] OpenStack. *OpenStack Monasca website.* http://monasca.io/. [Online; accessed July-2017]. 2014 (cit. on pp. 23, 59).

[142] Oracle. *Railcar Github repository.* https://github.com/oracle/railcar. Last accessed: Oct 2020 (cit. on p. 8).

[143] Fawaz Paraiso, Stephanie Challita, Yahya Al-Dhuraibi, and Philippe Merle. “Model-Driven Management of Docker Containers”. In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD).* IEEE, 2016,

pp. 718–725. ISBN: 978-1-5090-2619-7. DOI: 10.1109/CLOUD.2016.0100. URL: http://ieeexplore.ieee.org/document/7820337/ (cit. on p. 20).

[144] Gongzhuang Peng, Hongwei Wang, Jietao Dong, and Heming Zhang. "Knowledge-Based Resource Allocation for Collaborative Simulation Development in a Multi-Tenant Cloud Computing Environment". In: *IEEE Transactions on Services Computing* 11.2 (2018), pp. 306–317. DOI: 10.1109/TSC.2016.2518161 (cit. on p. 112).

[145] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. "Serverless computing for container-based architectures". In: *Future Generation Computer Systems* 83 (2018), pp. 50 –59. ISSN: 0167-739X. DOI: 10.1016/j.future.2018.01.022 (cit. on p. 98).

[146] Reid Priedhorsky and Tim Randles. "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126925. URL: https://doi.org/10.1145/3126908.3126925 (cit. on pp. 8, 116).

[147] Docker Open Source project. *Docker website*. https://www.docker.com/. [Online; accessed July-2017]. 2017 (cit. on pp. 7, 54, 85, 92).

[148] EUBra-BIGSEA project. *EC3 client Docker Image*. https://hub.docker.com/r/eubrabigsea/ec3client/. [Online; accessed July-2017]. 2017 (cit. on p. 36).

[149] *PRoot Github repository*. https://github.com/proot-me/PRoot. Last accessed: Oct 2020 (cit. on p. 9).

[150] RedHat. *A Practical Introduction to Container Terminology.* `https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/`. Last accessed: Oct 2020 (cit. on p. 8).

[151] Redhat. *Podman website.* `https://podman.io/`. Last accessed: Oct 2020 (cit. on p. 8).

[152] *RIKEN's Fugaku Utilizes Sylabs' SingularityPRO.* `https://sylabs.io/2020/08/worlds-fastest-supercomputer-runs-singularity-pro?s=09`. Last accessed: Aug 2020 (cit. on p. 2).

[153] *RIKEN's Fugaku website.* `https://www.r-ccs.riken.jp/en/fugaku/project`. Last accessed: Aug 2020 (cit. on p. 3).

[154] Nayan B. Ruparelia. "Software Development Lifecycle Models". In: *ACM SIGSOFT Software Engineering Notes* 33.3 (2010), pp. 8 –13. ISSN: 0163-5948. DOI: `https://doi.org/10.1145/1764810.1764814`. URL: `https://dl.acm.org/doi/10.1145/1764810.17648144` (cit. on p. 7).

[155] Mina Sedaghat, Francisco Hernandez-Rodriguez, and Erik Elmroth. "A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling". In: *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference on - CAC '13.* 2013, p. 1. ISBN: 9781450321723. DOI: `10.1145/2494621.2494628`. URL: `http://dl.acm.org/citation.cfm?doid=2494621.2494628` (cit. on p. 14).

[156] Sergio López Huguet. *Linpack parallelized with OpenMP Docker Image.* `https://hub.docker.com/r/serlophug/openmplinpack/`. [Online; accessed July-2017]. 2017 (cit. on p. 36).

[157] Sergio López Huguet. *Source code of vertical elasticity wrapper.* `https://github.com/eubr-bigsea/vertical_elasticity`. [Online; accessed July-2017]. 2017 (cit. on pp. 30, 36).

[158] *SetUID man page.* https://www.man7.org/linux/man-pages/man2/setuid.2.html. Last accessed: Oct 2020 (cit. on p. 8).

[159] Kashish Ara Shakil and Mansaf Alam. "Cloud Computing in Bioinformatics and Big Data Analytics: Current Status and Future Research". In: *Advances in Intelligent Systems and Computing.* Springer Singapore, 2017, pp. 629–640. DOI: 10.1007/978-981-10-6620-7_60. URL: \url{http://dx.doi.org/10.1007/978-981-10-6620-7_60} (cit. on p. 83).

[160] J. Shamsi, Muhammad Ali Khojaye, and M. Qasmi. "Data-Intensive Cloud Computing: Requirements, Expectations, Challenges, and Solutions". In: *Journal of Grid Computing* 11 (2013), pp. 281–310 (cit. on p. 112).

[161] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. "Kingfisher: Cost-aware elasticity in the cloud". In: *2011 Proceedings IEEE INFOCOM.* 2011, pp. 206–210 (cit. on p. 7).

[162] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. "CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems". In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing.* SOCC '11. New York, NY, USA: ACM, 2011, 5:1–5:14. ISBN: 978-1-4503-0976-9. DOI: 10.1145/2038916.2038921. URL: http://doi.acm.org/10.1145/2038916.2038921 (cit. on pp. 19, 57).

[163] *Singularity security documentation page.* https://sylabs.io/guides/3.6/user-guide/security.html. Last accessed: Oct 2020 (cit. on p. 9).

[164] SixSq. *SlipStream.* http://sixsq.com/products/slipstream/. [Online; accessed April-2018] (cit. on p. 53).

[165] Open Source. *OpenStack website.* https://www.openstack.org/. [Online; accessed July-2017]. 2010 (cit. on pp. 23, 52, 83).

[166] University of Stuttgart. *OpenTOSCA*. http://www.opentosca.org. [Online; accessed April-2018] (cit. on p. 53).

[167] Domenico Talia, Paolo Trunfio, and Fabrizio Marozzo. "Chapter 5 - Research Trends in Big Data Analysis". In: *Data Analysis in the Cloud: Models, Techniques and Applications*. Computer Science Reviews and Trends. Boston: Elsevier, 2016, pp. 123–138. ISBN: 978-0-12-802881-0. DOI: 10.1016/B978-0-12-802881-0.00005-6 (cit. on p. 112).

[168] DXC Technology. *Eucalyptus Cloud Platform*. https://github.com/eucalyptus/eucalyptus. [Online; accessed April-2018] (cit. on p. 52).

[169] Massachusetts Institute of Technology. *StarCluster*. http://web.mit.edu/stardev/cluster/. [Online; accessed April-2018] (cit. on pp. 6, 55, 145).

[170] Massachusetts Institute of Technology. *StarCluster Elastic Load Balancer*. http://web.mit.edu/stardev/cluster/docs/0.92rc2/manual/load_balancer.html. [Online; accessed April-2018] (cit. on pp. 6, 55).

[171] The Cloud Native Computing Foundation (CNCF). *Kubernetes*. http://kubernetes.io/. [Online; accessed April-2018] (cit. on pp. 9, 54, 86, 112).

[172] *TOP500 list November 2020*. https://www.top500.org/lists/top500/2020/11/. Last accessed: Aug 2020 (cit. on p. 3).

[173] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, Krishna Gade, Maosong Fu, J. Donham, Nikunj Bhagat, Sailesh Mittal, and D. Ryaboy. "Storm@twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014) (cit. on p. 6).

[174] *Unveiling CERN Cloud Architecture.* `https://www.openstack.org/videos/summits/tokio-2015/unveiling-cern-cloud-architecture`. Last accessed: Aug 2020 (cit. on p. 5).

[175] *User namespaces man page.* `https://www.man7.org/linux/man-pages/man7/user_namespaces.7.html`. Last accessed: Oct 2020 (cit. on p. 8).

[176] Virtuozzo. *CRIU website.* `https://criu.org/Main_Page`. [Online; accessed July-2017]. 2011 (cit. on pp. 21, 60, 61).

[177] Virtuozzo. *OpenVZ website.* `https://openvz.org/`. [Online; accessed July-2017]. 2005 (cit. on p. 21).

[178] Weaveworks. *Weave.* `https://www.weave.works/`. [Online; accessed April-2018] (cit. on p. 60).

[179] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. "Ceph: A Scalable, High-performance Distributed File System". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation.* OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320. ISBN: 1-931971-47-1. URL: `http://dl.acm.org/citation.cfm?id=1298455.1298485` (cit. on pp. 98, 128).

[180] Dongyao Wu, Sherif Sakr, and Liming Zhu. "Big Data Programming Models". In: *Handbook of Big Data Technologies.* Ed. by Albert Y. Zomaya and Sherif Sakr. Cham: Springer International Publishing, 2017, pp. 31–63. ISBN: 978-3-319-49340-4. DOI: `10.1007/978-3-319-49340-4_2`. URL: `https://doi.org/10.1007/978-3-319-49340-4_2` (cit. on p. 5).

[181] Jie Xiong. *Cloud Computing for Scientific Research.* Scientific Research Publishing, Inc. USA, 2018 (cit. on p. 5).

[182] Xiaoyu Yang, D. Wallom, S. Waddington, Jianwu Wang, A. Shaon, B. Matthews, M. Wilson, Y. Guo, Li Guo, J. Blower, A. Vasilakos, K. Liu,

and P. Kershaw. “Cloud computing in e-Science: research challenges and opportunities”. In: *The Journal of Supercomputing* 70 (2014), pp. 408–464 (cit. on p. 5).

[183] Sami Yangui, Iain James Marshall, Jean Pierre Laisne, and Samir Tata. “CompatibleOne: The Open Source Cloud Broker”. In: *Journal of Grid Computing* 12.1 (2014), pp. 93–109. ISSN: 15707873. DOI: `10.1007/s10723-013-9285-0` (cit. on p. 53).

[184] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2003). ISSN: 03029743. DOI: `10.1007/10968987\_3` (cit. on p. 102).

[185] Chao Zheng, Ben Tovar, and Douglas Thain. “Deploying high throughput scientific workflows on container schedulers with makeflow and mesos”. In: *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017*. 2017, pp. 130–139. ISBN: 9781509066100. DOI: `10.1109/CCGRID.2017.9` (cit. on pp. 20, 58, 143).

Glossary

AI Artificial Intelligence. 155

AKS Azure Kubernetes Service. 112

API Application Programming Interface. 23, 34–36, 61–63, 87, 89, 101, 104, 106, 118, 119, 121, 122, 127, 128

AWS Amazon web Services. 52, 53, 55, 56

CAPI Cluster API. 119

CephFS Ceph File System. 129, 130

CLUES CLuster Elasticity System. 7, 56, 59, 60, 69, 73, 92–94, 97, 126, 138, 153

CMP Cloud Management Platforms. 52, 53, 78

CPU Central Processing Unit. xxi, 7, 10, 36–38, 42, 43, 45

CRD Kubernetes Custom Resource Definitions. 118

CRI-O Open Container Runtime Interface. 8, 10

DIPG Diffuse Intrinsic Pontine Glioma. 155

DSA Digital Signature Algorithm. 135

EC2 Elastic Cloud Computing. 53, 79, 83, 85, 116

EC3 Elastic Cloud Computing Cluster. 6, 7, 36, 56, 58, 66, 69, 79, 86, 92, 112, 126, 128, 138, 140, 153

EKS Amazon Elastic Kubernetes Service. 112

FaaS Functions as a Service. 98, 150

FPGA Field-programmable gate array. 104

GB Gigabyte. 35, 68, 69, 71, 93, 94, 106

GKE Google Kubernetes Engine. 112

GPU Graphics Processing Unit. 82, 84, 89, 104, 107, 110, 140

GUI Graphical User Interface. 126, 129, 133

HNC Hierarchical Namespace Controller. 120

HPC High Performance Computing. vii–x, xiii, xiv, xxiii, 1–4, 8, 9, 11, 54, 102–108, 110, 116, 137, 140, 144, 146, 151, 156

HTC High Throughput Computing. vii, ix, xiii, 1–4, 7, 99

HTTP Hypertext Transfer Protocol. 9, 118, 127

HTTPS Hypertext Transfer Protocol Secure. 9, 118, 127

IaaS Infrastructure as a Service. 19, 35, 48, 55, 56, 78, 126

IdP Identity provider. 118, 127, 131

IM Infrastructure Manager. 53, 58, 59, 78, 79, 92, 93, 126, 129, 133, 138, 153

IoT Internet of Things. 101

IP Internet Protocol. 35

JSON JavaScript Object Notation. 24, 27, 52, 61, 123, 131

K8s Kubernetes. 9, 133

KubeFed Kubernetes Cluster Federation. 118–120, 128

KVM Kernel-based Virtual Machine. 69

LXC Linux Containers. 7, 8, 10, 21, 54, 85

MB Megabyte. 44

MPI Message Passing Interface. 55, 102

NFS Network File System. 36, 44, 55, 59, 60, 77, 127

OCI Open Container Initiative. 7–9, 116, 117, 156

OIDC OpenID Connect. viii, x, xiv, xxiii, 12, 113–115, 117, 118, 121–124, 127, 128, 130, 136, 141, 154

PaaS Platform as a Service. 22

PI Principal investigator. 150, 151

POSIX Portable Operating System Interface. 90, 92, 126, 155

PRIMAGE PRedictive In-silico Multiscale Analytics to support cancer personalized diaGnosis and prognosis, Empowered by imaging biomarkers. viii, x, xiv, 5, 101, 106, 136, 141, 149, 151, 154

QoS Quality of Service. 2, 4, 10, 11, 14, 16, 18, 20, 24, 27–33, 38, 40, 44, 48, 50, 51, 57, 60, 61, 68, 73, 75, 79, 80, 94, 137, 144

RADL Resource Application Description Language. 93

RAM Random Access Memory. 6, 19, 35, 57, 68, 69, 71, 72, 93, 94, 106

RBAC Role-based access control. 114, 120–122, 130, 136

REST Representational State Transfer. 22–29, 34–36, 61–63, 87, 89, 104, 106, 121, 122, 141

SCAR Serverless Container-aware ARchitectures. 98

UPV Universitat Politècnica de València. 154

VM Virtual Machine. 7, 15, 16, 19, 35, 37, 48, 54–59, 65, 66, 68, 71, 73, 94, 112, 134, 135

VMI Virtual Machine Image. 69, 78

VO Virtual Organization. 118, 123, 131–135

XML EXtensible Markup Language. 53

YAML YAML Ain't Markup Language. 52, 53, 107, 131