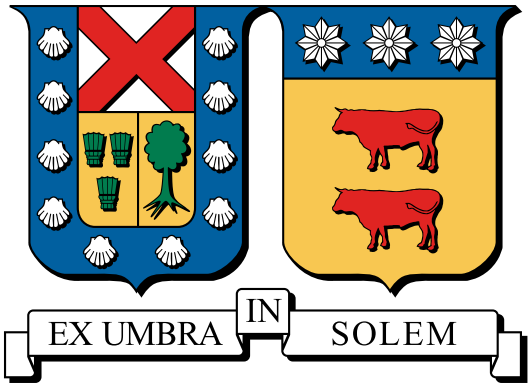
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



REPRODUCIBILITY OF COMPUTATIONAL ENVIRONMENTS FOR SCIENTIFIC EXPERIMENTS USING-CONTAINER-BASED VIRTUALIZATION

MAXIMILIANO FEDERICO OSORIO BAÑADOS

TESIS PARA OPTAR LA GRADO DE
MAGÍSTER EN CIENCIAS DE LA INGENIERÍA INFORMÁTICA

DICIEMBRE 2018

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

Científicos han argumentado sobre la importancia de poder reproducir experimentos científicos debido a la necesidad de corroborar los resultados y conclusiones y a la necesidad de contribuir en el mismo trabajo.

Sin embargo, proceso de reproducibilidad puede ser tan complejo y costoso que ha sido referido como una tarea forense [1] De hecho, estudios controversiales y escándalos han expuesto la necesidad de mejorar los procesos de reproducibilidad de las investigaciones. Algunos ejemplos son: casos clínicos [2], sondeos del área de psicología [3] o estudios en área de la computación [1]

La reproducibilidad del experimento es la capacidad de volver a ejecutar un experimento con o sin la introducción de cambios en él y la obtención de resultados que sean consistentes con los originales. La introducción de cambios permite evaluar diferentes características de ese experimento ya que los investigadores pueden modificarlo gradualmente, mejorando y re- orientando los métodos y condiciones experimentales [4].

Para permitir la reproducibilidad del experimento es necesario proporcionar suficiente información sobre él mismo, permitiendo comprenderlo, evaluarlo y volver a construirlo. Normalmente, los experimentos se describen en flujos de trabajo científicos (representaciones que permiten gestionar cálculos a gran escala) que se ejecutan en sistemas informáticos distribuidos. Para permitir la reproducibilidad de estos flujos de trabajos científicos es necesario, en primer lugar, abordar un problema de conservación del flujo de trabajo. La conservación es un acto de proporcionar información compresible que describa el contexto original del experimento y es por ello que los flujos de trabajo experimentales necesitan garantizar que haya suficiente información sobre los experimentos para que sea posible

construirlos nuevamente por un tercero, replicando sus resultados sin ninguna información adicional del autor original [5].

Para alcanzar la conservación la comunidad se ha enfocado en la conservación de las ejecuciones de workflow conservando los datos, código y la descripción, pero no dejando de lado la infraestructura subyacente (e.g, recursos computacionales y componentes de software). Existe otros enfoques que se han enfocado en conservar el ambiente computacional de un experimento como WICUS [6] o el proyecto TIMBUS [1] [7] que describe los procesos de negocios y el software y el hardware subyacente.

Los autores en [6] identificaron dos enfoques para conservar el ambiente computacional de un experimento científico: la conservación física, en la que los objetos de investigación dentro del experimento se conservan en un entorno virtual (e.g., máquinas virtuales); y la conservación lógica, en la que se describen las principales capacidades de los recursos del ambiente computacional utilizando vocabularios semánticos para que el investigador pueda reproducir un entorno equivalente. Definieron un proceso para documentar la aplicación de flujo de trabajo y su sistema de gestión relacionado, así como sus dependencias.

Sin embargo, este proceso es realizado en forma manual, dejando trabajo restante para los científicos. Además, al igual que la mayoría de los trabajos dejan fuera del alcance la conservación física del entorno computacional del flujo de trabajo (basándose en la infraestructura elegida). Sin embargo, la conservación lógica y física es importante para lograr la reproducibilidad del experimento.

El problema de reproducibilidad no sólo sucede en la comunidad científica. Compañías tecnológicas también enfrenta problemas similares cuando ellos desean distribuir cualquier producto de software a múltiples servidores. Para resolver esto, compañías utilizan virtualización a nivel. Esta tecnología se refiere a una característica del Sistema Operativo (SO) en la que el kernel del SO permite la existencia de múltiples instancias aisladas de espacio de usuario llamadas maquinas virtuales o contenedores. Una de las tecnologías de virtualización más populares en la industria es Docker[2], que implementa la virtualización de software mediante la creación de versiones mínimas de un sistema operativo base (un contenedor). Los Docker Containers pueden ser vistos como máquinas virtuales ligeras que permiten el ensamblaje de un entorno computacional, incluyendo todas las dependencias necesarias, como bibliotecas, configuración, código y datos necesarios, entre otros.

[1] `http://www.timbusproject.net/`
[2] `https://www.docker.com/`

Para reproducir experimentos científicos computacionales de otros investigadores, es obligatorio permitir que los científicos compartan estos experimentos o permitir que los ejecuten de nuevo ambiente computacional reproducido (el mismo o muy similar). De esta manera, un científico tendrá garantías de que el experimento que está ejecutando en el mismo entorno o similar desde el que fue creado. Por lo tanto, se necesita un procedimiento que garantice ambos requisitos: preservar los entornos tanto lógicos como físicos para volver a ejecutar los flujos de trabajo de datos con garantías de reproducibilidad. Para permitir esta reproducibilidad es necesario, por lo tanto, proporcionar una manera de describir primero los experimentos científicos y el entorno computacional en el que se ejecutó el experimento. La conservación lógica permite describir el entorno computacional en el que se ejecutó el experimento y la conservación física permite a los investigadores volver a ejecutar un experimento sin tener que ocuparse de la distribución física del a y sin tener que ocuparse de los problemas de dependencias.

Por lo tanto se una solución para mejorar la solución de conservación física y lógica mediante el uso de contenedores. Se propone en primer lugar utilizar las imágenes Docker como medio para preservar el entorno físico de un experimento. Se utiliza contenedores ya que son ligeros y, lo que es más importante, son más fáciles de describir automáticamente, por lo que mejoramos el proceso de documentación de los flujos de trabajo científicos. Con Docker, los usuarios pueden distribuir estos entornos computacionales a través de imágenes de software utilizando un repositorio público llamado DockerHub. Por lo tanto, nuestro objetivo es abordar la conservación física. Con el fin de lograr una conservación lógica, se construye un sistema de anotador para las imágenes Docker que describe el sistema de gestión del flujo de trabajo, así como sus dependencias mediante el desarrollo de un sistema de anotador para las imágenes Docker antes. De esta manera, se pretende abordar la conservación física. Para validar la solución se reproducen cuatros experimentos computacionales diferentes. Estos experimentos abarcan diferentes sistemas, lenguajes y configuraciones, lo que demuestra que nuestro enfoque es genérico y puede aplicarse a cualquier experimento computacional. Realizamos estos experimentos, los describimos lógicamente y a continuación los reproducimos en base a las descripciones lógicas que obtuvimos anteriormente. Para validar el enfoque, comparamos los resultados de los experimentos.

Capítulo 2

Estado del arte

2.1. Conservación de procedimiento científicos

Distintas áreas de la ciencia han adoptado técnicas y herramientas para conservar el procedimiento. Por ejemplo, investigadores en bio-informática ha incorporado los workflows para distintos análisis: Doblamiento de proteínas [8], secuencias de DNA y RNA [9, 10] y la detección de ondas gravitacionales [11]. Los workflows científicos son métodos que permiten representar un conjunto de pasos computacionales. Estos pasos pueden ser la obtención de los datos de entrada, transformaciones o generación de los resultados. La representación de los workflow se construye en un lenguaje abstracto para simplificar la complejidad. El conjunto de pasos se pueden representar como grafos sin ciclos y dirigidos, donde cada paso computacional es representado por un nodo y las dependencias entre los pasos son representado por los arcos. El uso de sistemas de manejo de workflows científicos *Scientific Workflow management Systems (WMS)* permiten diseñar abstractamente, ejecutar y compartir el procedimiento científicos.

Dado que los workflows formalmente describen la secuencia de tareas computacionales y administración de datos, es fácil encontrar el camino de los datos producidos. Un científico podría ver el workflow y los datos, seguir los pasos y llegar al mismo resultado. En otras palabras, la representación del workflow facilita la creación y administración de la computación y además construye una base en la cual los resultados pueden ser validados y compartidos.

Sin embargo, múltiples estudios han mostrado las dificultades reproducir los resultados de los experimentos.

2.1.1. Sistemas de administración de workflows

Cómo se menciono anteriormente, los workflows científicos permiten a los usuarios expresar fácilmente tareas computacionales de varios pasos, por ejemplo, recuperar datos de un instrumento o una base de datos, formatear los datos y ejecutar un análisis. Un workflow científico describe las dependencias entre las tareas y la mayoría de los casos se describe como un gráfico acíclico dirigido (DAG), donde los nodos son tareas y los bordes denotan las dependencias de las tareas. Una propiedad que define un workflow científico es que gestiona el flujo de datos. Es por ello, que las tareas en un workflow científico varían ampliamente según las necesidades del autor, los tipos pueden ser tanto como tareas cortas en serie o tareas paralelas muy grandes (e.g, utilizando Message Passing Interface - MPI) rodeadas de un gran número de pequeñas tareas en serie utilizadas para el pre y post procesamiento. La interpretación y ejecución de los workflows son manejados por un sistema de manejo de workflows (WMS) que administra la ejecución de la aplicación en la infraestructura. Un WMS puede ser considerado como una capa intermedia necesaria para la abstracción y orquestación de procedimiento científico. A continuación se describe algunos de los WMS más populares.

Actualmente, existen múltiples WMS que han sido generado por diversas comunidades.

2.1.2. Galaxy

Galaxy [12] es un sistema de gestión de almacenes basado en la web que tiene como objetivo llevar las capacidades de análisis de datos computacionales a usuarios no expertos en el campo de las ciencias biológicas. Los principales objetivos del marco de trabajo de Galaxy son la accesibilidad a las capacidades computacionales biológicas y la reproducibilidad del resultado del análisis mediante el seguimiento de la información relacionada con cada paso del proceso.

2.1.3. Taverna

Taverna [13] es un WMS basado en servicios web, ya que todos los componentes del flujo de trabajo deben implementarse como servicios web (ya sea localmente o utilizando un servicio remoto disponible). Taverna es capaz de integrar los servicios web de Soaplab26, REST (Fielding, 2000) y WSDL (Christensen et al., 2001). Ofrece una amplia gama de

servicios para diferentes capacidades de procesamiento, tales como servicios Java locales, servicios de procesador R estadístico, scripts XPath o servicios de importación de hojas de cálculo.

2.1.4. Pegasus

Pegasus [11] es un WMS capaz de gestionar flujos de trabajo compuestos por millones de tareas, registrando datos sobre la ejecución y resultados intermedios. En Pegasus, los flujos de trabajo se describen como flujos de trabajo abstractos, que no contienen información de recursos, o las ubicaciones físicas de datos y ejecutables.

2.1.5. WINGS

WINGS [14] no puede ser considerado como un WMS apropiado por sí mismo, ya que no proporciona características de ejecución y promulgación del flujo de trabajo. Sin embargo, es ampliamente conocido por el diseño del flujo de trabajo. WINGS puede considerarse como una herramienta de diseño de alto nivel y orientada a los dominios, cuyos flujos de trabajo pueden ejecutarse posteriormente en diferentes motores de flujo de trabajo, como Pegasus o Apache OODT29.

2.1.6. dispel4py

dispel4py [15] es una biblioteca Python (Rossum, 1995) para describir flujos de trabajo. Describe flujos de trabajo abstractos para aplicaciones intensivas en datos, que luego se traducen y ejecutan en plataformas distribuidas (por ejemplo, Apache Storm, clusters MPI, etc.).

2.2. Conservación de equipamiento

Comúnmente el equipamiento en otras disciplinas no es un problema a resolver dado que los recursos utilizados son son conocidos, no-variables y estándares. Por ejemplo, la utilización de probetas, microscopios u otros elementos físicos. Consecuentemente, los investigadores pueden nombrarlos e identificarlos de forma manual en los procedimientos de sus cuadernos de laboratorio. Lo que permite que otro investigador conozca cuáles fueron las

herramientas utilizadas en el experimento. Aunque existen excepciones como en la ciencia de biología, donde ciertos recursos son materiales que son utilizados en los procedimientos. En estos casos, los investigadores deben describir los materiales incluyendo información como marcas, composición y otros. En otras ciencias cómo la astronomía se utilizan recursos de alta tecnología, donde también es necesario documentar las características de hardware y configuraciones utilizadas en el proceso experimental. En las ciencias de la computación sucede un caso similar, dado que los recursos computacionales son una componente requerida en la ejecución del sistema. Es por ello, que esta comunidad no puede ser la excepción respecto a la descripción de los recursos. Por lo tanto, los autores deben poder documentar computadores, clusters, servicios web, componentes de software, etc., en el contexto de sus experimentos.

Diversos trabajos han estudiado el estado actual de la reproducibilidad en las ciencias de computación, en [16] se estudia el factor de decaimiento de un conjunto workflows científicos almacenado en myExperiment [1] que fueron diseñados para el WMS Taverna [13] del área de biología. Para ello, los autores utilizaron cuatro conjuntos de paquetes de workflows y clasifican el decaimiento de los workflows en cuatro categorías: recursos de terceros volátiles, datos de ejemplos faltantes, ambiente de ejecución faltante y descripciones insuficientes sobre los workflow. El estudio muestra que casi el 80 % de workflows fallan al ser reproducidos, con un 12 % de esos fallos debido al ambiente de ejecución faltante y 50 % recursos de terceros volátiles. Estas dos últimas categorías están asociadas a la conservación del ambiente computacional del experimento. En [17], los autores describen un procedimiento para preservar el software, argumentando que el software es frágil a los cambios de ambiente ya sea hardware, sistema operativo, versiones de las dependencias y configuración. Los autores afirman que el software no puede ser preservado con la metodología anterior de sólo mantener su código binario ejecutable. Por ello, introducen el concepto de adecuación de la preservación, una métrica para medir si la preservación de conjunto de funcionalidades de componente de software luego de un proceso reproducción.

De la misma manera, editoriales se ha enfocado en intentar resolver los desafíos en la publicación de trabajos científicos. Por ejemplo, Elsevier formó el *Executable Paper Grand Challenge* para abordar la dificultad de reproducibilidad de los resultados en las ciencias de la computación, ellos argumentan que los bloques vitales y necesarios de información para replicar los resultados -por ejemplo, software, código, grandes conjuntos

[1] `https://www.myexperiment.org/home`

de datos- no suelen estar disponibles en el contexto de una publicación académica. Y *Executable Paper Grand Challenge* creó una oportunidad para que los científicos diseñen soluciones que capturen esta información y proporcionen una plataforma para que estos datos puedan ser verificados y manipulados. En 2011, [18] se argumenta que el documento de investigación en su estado actual ya no es suficiente para reproducir, validar o revisar completamente los resultados y conclusiones experimentales de un documento. Esto impide el progreso científico. Para remediar estas preocupaciones, presentan Paper Mâché, un nuevo sistema para crear documentos de investigación dinámicos y ejecutables. La principal novedad de Paper Mâché es el uso de máquinas virtuales, que permite a los lectores y revisores ver e interactuar fácilmente con un documento y poder reproducir los principales resultados experimentales. En la misma línea, CernVM [19] propuso la utilización de máquinas virtuales para resolver problemas de reproducibilidad en la ciencia. CernVM es un sistema para el uso de máquina virtuales capaz de ejecutar aplicaciones físicas de los experimentos relacionados al *Large Hadron Collider* (LHC) en el *European Organization for Nuclear Research* (CERN). Su objetivo es proporcionar un entorno completo y portátil para desarrollar y ejecutar el análisis de datos LHC en cualquier ordenador del usuario final (portátil, de sobremesa), así como en la red, independientemente de las plataformas de sistemas operativos (Linux, Windows, MacOS). La motivación del uso de técnicas de virtualización que permite separar los recursos de computación desde la infraestructura subyacente.

Algunos autores han expuesto la necesidad de capturar y preservar el entorno de ejecución de un ejecución del experimento, proporcionando herramientas para analizar y empaquetar los recursos involucrados en él. ReproZip [20] busca captar el conocimiento sobre una infraestructura e intentar reproducirla en un nuevo entorno. Esta herramienta lee los componentes de infraestructura involucrados en la ejecución (archivos, variables de entorno, etc.) y almacena esta información en una base de datos MongoDB [2]. A continuación se recogen y empaquetan los elementos descritos. Luego, el sistema debe desempaquetar los elementos en otra máquina para reproducir el experimento. Sin embargo, este tipo de enfoque que empaqueta los componentes físicos de una infraestructura determinada presenta limitación en la práctica, debido que los paquetes deben ser ejecutado en una máquina destino similar. Otro ejemplo es TOSCA (Topology and Orchestration Specification for Cloud Applications), TOSCA es un ejemplo de las soluciones que han definido sintaxis

[2] `https://www.mongodb.com/es`

para describir la ejecución de los ambientes computacionales. TOSCA es un lenguaje de código abierto utilizado para describir las relaciones y dependencias entre servicios y aplicaciones que residen en una plataforma de computación. TOSCA puede describir un servicio de computación en nube y sus componentes y documentar la forma en que están organizados y el proceso de orquestación necesario para utilizar o modificar dichos componentes y servicios. Esto proporciona a los administradores una forma común de gestionar aplicaciones y servicios en la nube, de modo que esas aplicaciones y servicios puedan ser portátiles a través de las diferentes plataformas de los proveedores de Cloud Computing. Otro esfuerzo importante relacionado a nuestro trabajo incluye la descripción de los ambientes computaciones utilizando ontologías es TIMBUS. El proyecto se focaliza en preservar procesos de negocios y su infraestructura computacional. Para ello, propusieron un extractor para extraer y anotar los componentes de Software y Hardware, éstas anotaciones son almacenadas según un conjunto de ontologías con el objetivo de gestionar la preservación y ejecución de los procesos de negocio. Sin embargo, el enfoque extractor del Proyecto Timbus no es adecuado para ser utilizado en cualquier sistema de virtualizada ya que aumenta la complejidad del ambiente y generando ruido. Además, exige ejecutar el ambiente computacional lo cual conlleva altos costos computacionales, disminución en la escalabilidad y creación de brechas de seguridad.

En el enfoque de describir los recursos computacionales, los autores en [6] identificaron dos enfoques para conservar el ambiente computacional de un experimento científico: la conservación física, donde los objetos de investigación dentro del experimento se conservan en un entorno virtual; y la conservación lógica, donde las principales capacidades de los recursos en el entorno se describen utilizando vocabularios semánticos para permitir al investigador reproducir un entorno equivalente. Para ello, definieron un proceso para documentar la aplicación de flujo de trabajo y su sistema de gestión relacionado, así como sus dependencias. Además, los autores propusieron *The Workflow Infrastructure Conservation Using Semantics ontology* (WICUS). WICUS es una red de ontologías OWL2 (Web Ontology Language) que implementan la conceptualización de los principales dominios de una infraestructura computacional. Como: Hardware, Software, Workflow y Recursos Informáticos. Los autores argumentan este trabajo debido que a que el workflow científico requiere un conjunto de componentes de software, y los investigadores deben saber cómo desplegar esta pila de software para lograr un entorno equivalente. Sin embargo, este proceso se realiza de forma de manual, dejando mucho trabajo a los científicos. Además,

los autores afirman que la conservación de los ambientes computacionales comúnmente se logra utilizando un enfoque físico dado que la conservación física permite compartir fácilmente un ambiente computacional con otros investigadores y ellos pueden reproducir el experimento utilizando en el mismo ambiente. Sin embargo, los esfuerzos necesarios para mantener la infraestructura son altos y no hay garantías que no sufran un proceso de decaimiento [21]. Consecuentemente, la mayoría de los trabajos dejan fuera del ámbito de aplicación la conservación física del entorno informático del workflow (basándose en la infraestructura elegida). Pese a que la conservación lógica y física son deseadas para lograr la reproducibilidad del experimento.

En diversos trabajos [22, 23, 24, 25] se ha propuesto la utilización de Docker como un reemplazo al uso de máquinas virtuales como ambiente computacionales científicos, los trabajos argumentan que Docker presenta beneficios de portabilidad, documentación precisa de la instalación y configuración, manejo de control de versiones de las imágenes y fácil adopción por desarrolladores. Un ejemplo de uso de Docker para la reproducibilidad es Bio Containers [22]. Bio Containers es un framework de código abierto y orientado a la comunidad que proporciona entornos ejecutables independientes de la plataforma para el software de informática. Bio Containers permite a los laboratorios instalar fácilmente software de bio-informática, mantener múltiples versiones del mismo software y combinar herramientas de análisis. Bio Containers se basa en los populares proyectos de código abierto Docker [3], Singularity [4] y rkt [5], que permiten que el software sea instalado y ejecutado bajo un entorno aislado y controlado. Sin embargo, en [24, 26] los autores exponen que Docker no controla que paquetes instalados en las imágenes y no existe una descripción completa de los componentes de la imagen y consecuente del contenedor. Por lo tanto, las imágenes Docker funcionan como una caja negra, lo que significa que los usuarios saben cuál es el paquete que se ejecuta dentro del contenedor pero no conocen las versiones o los otros paquetes necesarios para ejecutarlo.

Respecto a la descripción de los componentes de una imagen Docker, en [27] analizó más de 300.000 imágenes de Docker almacenadas en el repositorio oficial de Docker. Los autores encontraron que en promedio las imágenes que contiene el Docker Hub (sistema de almacenamiento de imágenes) son más de 180 vulnerabilidades, siendo la raíz de tal

[3] `https://www.docker.com/`
[4] `https://www.sylabs.io/singularity/`
[5] `https://coreos.com/rkt/`

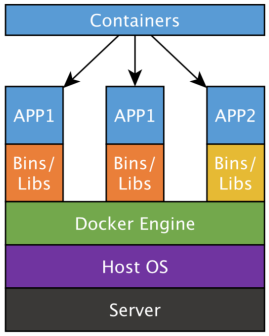
cantidad de vulnerabilidades el hecho de que muchas imágenes no han sido actualizadas en varios días y que muchas de estas vulnerabilidades se propagan de imágenes de padres a hijos. Los autores encontraron correlaciones entre las imágenes más influyentes y los paquetes vulnerables mejor clasificados, lo que sugiere que la fuente de esa cantidad de vulnerabilidades era probablemente el resultado de la propagación de un pequeño número de imágenes populares (debido a la falta de actualización de las imágenes principales). Los autores utilizaron el software Clair [6] de la empresa CoreOS[7]. En términos de ingeniería ontológica, los autores en [28] presentan la ontología Smart contenedor que extiende DOLCE [29] y modela Docker en términos de sus interacciones para desplegar imágenes. Otro trabajo relacionado, [30] describe cómo usar RDF para representar archivos de construcción de Docker.

2.3. Docker

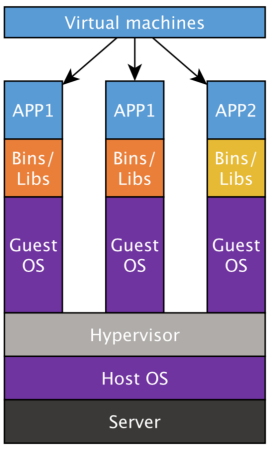
Durante los últimos veinte años el uso de tecnologías de virtualización ha aumentado rápidamente, esta tecnología permite dividir el sistema de un computador en múltiples ambientes virtuales.

Uno de los usos comunes para esta tecnología es la virtualización de servidores en *datacenters*. Con la virtualización de servidores, un administrador de sistemas puede crear una o más instancias virtuales o máquinas virtuales (VMs) en un servidor. Este enfoque hoy se utiliza comúnmente en *datacenters* y también en plataformas de *cloud* como Amazon EC2, RackSpace, Dreamhost y otros [31]. El crecimiento del uso de la virtualización ha hecho necesario la búsqueda de una solución que permita tener un ambiente escalable y seguro. Un gran numero de soluciones han nacido en el mercado y se pueden clasificar en dos tipos: *contenedor-based virtualization* y *Virtualización basada en hipervisores*. *contenedor-based virtualization* es una virtualización liviana a nivel de software usando el kernel de host para correr múltiples ambientes virtuales. Estos ambientes son nombrados con contenedores (contenedores). Y hoy en día Linux-VServer, OpenVZ, libcontenedor y Linux contenedor (LXC) son las principales implementaciones para utilizar contenedores. En la figura 2.1a se puede observar que la arquitectura de *contenedor-based virtualization*. Los contenedores utilizan el sistema operativo compartido del ambiente virtualizador (host)

[6] `https://github.com/coreos/clair`
[7] `https://coreos.com/`



(a) Virtualización basada en contenedores: los contenedores comparten el sistema operativo del host y por lo tanto no necesitan una copia.



(b) Virtualización basada en máquinas virtuales: las máquinas virtuales requieren un sistema operativo para cada uno.

Figura 2.1: Dado que un contenedor comparte recursos, esto reduce significativamente el uso de almacenamiento en comparación a las máquinas virtuales.

y por lo tanto no es necesario nuevamente el sistema operativo para cada contenedores que se está ejecutado. Para el sistema operativo del *host*, los procesos del contenedor no son especiales y se tratan como cualquier otro proceso que ejecuta sobre el kernel. Sin embargo, los contenedores son ambientes aislados y con recursos limitados y estas características se lograr a través de herramientas del Kernel [32].

Las diferencias entre la virtualización basada en contenedores y máquinas virtuales presentan características interesantes para la ejecución de aplicaciones y workflows. Dado que un contenedor comparte recursos con el sistema operativo, como las bibliotecas, esto reduce significativamente la necesidad de reproducir el código del sistema operativo, y significa

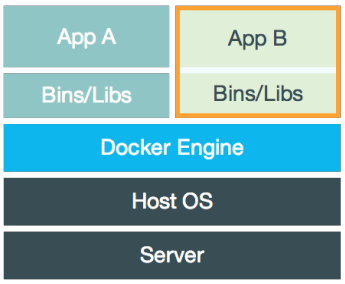


Figura 2.2: Arquitectura de Docker

que un servidor puede ejecutar varios ambientes con una sola instalación del sistema operativo. Por lo tanto, los contenedores son excepcionalmente ligeros: sólo tienen un tamaño de mega-bytes y sólo tardan unos segundos en arrancar. En comparación con los contenedores, las máquinas virtuales tardan minutos en funcionar y son un orden de magnitud mayor que un contenedor equivalente. [33, 34, 31]. A partir de esto nace la motivación de la utilización de contenedores con una alternativa para alcanzar la conservación física de los ambientes. Docker es un proyecto open-source que utiliza la tecnología de los contenedores (libcontenedor) para "construir, migrar y correr aplicaciones distribuidas". Actualmente utilizado por Yelp, Spotify, entre otros [35] Docker es una solución que simplifica el uso de los contenedores que han estado presente durante más de una década. Primero provee una interfaz simple y segura para crear y controlar contenedores [36], segundo permite a los desarrolladores empaquetar y correr sus aplicaciones de manera sencilla y además se integra con herramientas terceras que permiten administración y despliegue como Puppet[8], Ansible [9] y Vagrant[10]. Y diversas herramientas de orquestación como Mesos [11], Shipyard [12], Kubernetes [13], RancherOS [14] y Docker Swarm [15].

Docker puede separarse en dos grandes componentes Docker Engine y Client. Docker Engine es una herramienta liviana y portable para administrar la virtualización. Y Docker Client, provee una interfaz para interactuar con los contenedores con los usuarios a través de RESTful APIs[36].

[8] https://puppet.com
[9] https://ansible.com
[10] https://vagrant.com
[11] http://mesos.apache.org/
[12] https://shipyard-project.com/
[13] https://kubernetes.io/
[14] http://rancher.com/rancher-os/
[15] https://docs.docker.com/swarm/

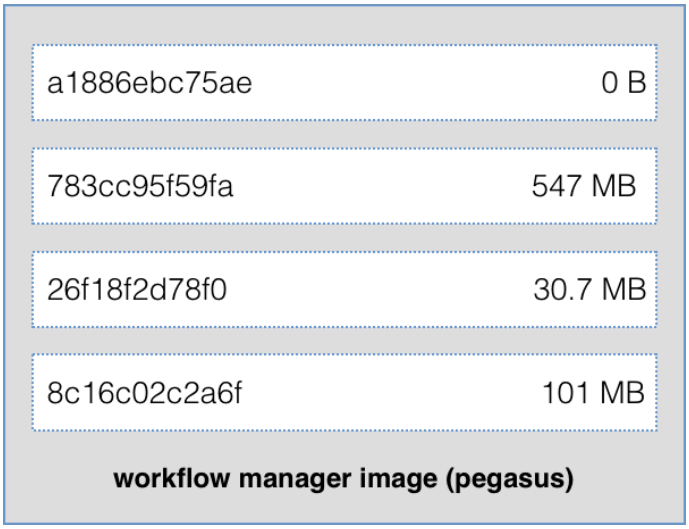


Figura 2.3: Las imágenes Docker con compuestas por capas. La última capa (a1886ebc75ae) es la única capa en modo escritura.

Docker utiliza una arquitectura de cliente-servidor, Docker Client interactúa con Docker Daemon y este construye, maneja y corre los contenedores. Docker Client y Docker Daemon pueden correr en el mismo host o se puede conectar el cliente desde un host remoto. El cliente y el Daemon se comunican en forma de sockets o RESTful API [16]. Una imagen de Docker (*Docker Image*) es una plantilla de solo lectura. Por ejemplo, una imagen puede contener las herramientas básicas de Ubuntu con Apache y una aplicación web instalada o simplemente el sistema operativo. Las imágenes son usadas para construir *Docker contenedores*. Cuando el usuario crea cambios en el contenedor, este cambio no se realiza en la imagen, sino que Docker añade una capa adicional con los cambios de la imagen[36]. Por ejemplo, si el usuario utiliza una imagen base de Debian, luego añade el paquete emacs y luego añade el paquete apache, el estado de capas estaría representado por la figura 2.3. Esto permite tener un proceso de distribución de imágenes más eficiente dado que solo es necesario distribuir las actualizaciones [36]. El sistema de archivos descrito anteriormente se denomina una sistema de archivo basado en capas.

Docker Hub es un repositorio que almacena dos tipos de repositorios públicos, oficiales y comunitarios. Los repositorios oficiales contienen imágenes públicas y verificadas de empresas y proveedores de software de renombre, como Canonical, Nginx, Red Hat y el propio Docker. Al mismo tiempo, los repositorios comunitarios pueden ser repositorios públicos o privados creados por usuarios y organizaciones individuales, que comparten sus

[16] `https://docs.docker.com/introduction/understanding-docker/`

aplicaciones y resultados.

Usando ese repositorio y una línea de comandos, es posible descargar e implementar imágenes Docker localmente, ejecutando el contenedor en un entorno host, y ejecutando así el software dentro de la imagen. Los usuarios pueden crear y almacenar imágenes en el registro del Docker Hub, creando un archivo descriptor llamado Dockerfile o ampliando uno existente. Este descriptor describe cuáles son los paquetes de software que estarán dentro de la imagen, construye la imagen y finalmente la sube al Docker Hub. Sin embargo, Docker Hub no controla qué paquetes hay en las imágenes, si la imagen se desplegará correctamente o si las imágenes pueden tener algún problema de seguridad. Así, las imágenes Docker funcionan como una caja negra, lo que significa que los usuarios saben sólo paquetes principales que se ejecutan dentro del contenedor pero no conocen los otros paquetes necesarios para ejecutarlo. Hay dos maneras de subir imágenes a un repositorio de usuario, ya sea enviar la imagen desde un host local o automatizando ese proceso desde un repositorio Github. Para enviar una imagen al Docker Hub, los usuarios necesitan nombrar sus imágenes locales usando su nombre de usuario Docker Hub, y el nombre del repositorio que han creado. Después, los usuarios añaden múltiples imágenes a un repositorio añadiéndole la versión `:<tag>`. Esta es toda la información que normalmente tienen las imágenes Docker en DockerHub, siendo por tanto casi imposible reproducir el entorno de ejecución si se modifica alguno de los paquetes de software utilizados dentro de las imágenes.

DockerHub permite a las comunidades científicas almacenar y compartir, conservando el contenido de los contenedores, así como comprobar la identidad del editor y recuperar los contenedores de interés. Sin embargo, la información del contenido de cada repositorio no siempre es accesible de forma clara. Como la mayoría de las veces sólo se proporcionan descripciones cortas de los contenedores, no es fácil para el usuario entender qué componentes están instalados en ellos. Incluso cuando los archivos Dockerfile están disponibles, no son lo suficientemente intuitivos de entender cuáles paquetes están siendo desplegados por cada comando. Además, es posible que en el contenedor existan algunos componentes que no estén especificados por el propio Dockerfile. Para abordar este problema proponemos un enfoque automático para analizar el contenido de un contenedor Docker y extraer la información sobre los componentes de software instalados en él. Esta información se convierte en datos semánticos, codificados como RDF bajo el conjunto de ontologías.

Los contenedores Docker consiste de un ambiente virtual con: con archivos de usuarios

Listing 2.1: Ejemplo de la ejecución de un contenedor utilizando la imagen Docker

```
docker run -i -t ubuntu /bin/bash
```

y metadatos, cada contenedor es construido a partir de una imagen base como se mencionó anteriormente. Esta imagen indica la base del contenedor y se asocia un proceso inicial, él cual debe correr cuando el contenedor es iniciado.

La figura 2.1describe los pasos incluidos en la creación de un contenedor.

- *Docker Client* le informa al Docker que debe correr un contenedor.
- En este caso el comando `/bin/bash` será el proceso `init` o 0 del contenedor
- **Traer la imagen:** Docker verifica la existencia de la imagen Ubuntu y sino existe en el host, entonces Docker descarga desde el repositorio ya sea privado o publico. Si la imagen existe entonces crea el contenedor.
- **Asignar el sistema de archivos y montar la capa de escritura:** El contenedor es creado en el sistema de archivos y se añade una capa en modo escritura en la imagen.
- **Crear la red y conectar con la interfaz puente:** Crea la interfaz de red que permite que contenedor pueda enviar y recibir paquetes con el host a través del puente (docker0).
- **Asignar un dirección IP:** Asigna una dirección IP del pool al contenedor
- **Capturar la salida estándar y de errores:** Dependiendo de la configuración de contenedor, Docker enviará a los registros de errores hacia el mismo servidor u otro externo.

Docker utiliza dos funcionalidades de Linux *namespaces* y *cgroups* para crear ambientes virtuales para los contenedores. *cgroups* o *control groups* proveen un mecanismo de contabilidad y limites de recursos que pueden utilizar los contenedores[36]. Los *namespaces* proveen del aislamiento que es llamado contenedor. Cuando sea crea un contenedor, Docker crea un conjunto de *namespaces* para el contenedor. Los *namespaces* utilizados por Docker: mount (mnt) para el manejo del montaje, PID para el aislamiento de los procesos, net para el manejo de interfaces y IPC para acceder a recursos de IPC. Docker logra el aislamiento

de los procesos separando los procesos en *namespaces* y limitando los permisos de los procesos a otros contenedores. *PID namespaces* (añadido en el kernel $\geq$ 2,6,3,2) es el mecanismo utilizado, logrando que un proceso que se encuentra en el contenedor solo pueda ver procesos que se encuentra en ese contenedor. Por lo tanto un atacante no puede observar procesos de otro contenedor, lo que aísla al contenedor este nivel [36]. Docker usa *mount namespaces* o *filesystem namespaces* para aislar los *filesystems* asociados a los contenedores. De la misma forma que ocurre con los procesos, los eventos del sistema de archivo que ocurre en el contenedor sólo afectan a ese contenedor.

Cómo se menciono anteriormente, Docker utiliza *cgroups* que permite especificar que *device* puede ser utilizado con el contenedor. Esto bloquea la posibilidad de crear y usar *device nodes* que puedan ser utilizados para atacar el host. Los *device nodes* que son creados para cada contenedor por defecto: son: /dev/console, /dev/null, /dev/zero, /dev/full, /dev/tty*, /dev/urandom, /dev/random, /dev/fuse. IPC (*inter-process communication* es un conjunto de objetos para el intercambio de datos a través de los procesos, como semáforos, colas de mensajes, segmentos de memoria compartida. Los procesos corriendo en los contenedores utilizan *IPC namespaces* que permite la creación de un *IPC* separado y independiente para cada contenedor, con esto se previene que procesos en un contenedor interfieran con otros contenedores o el host. Para cada contenedor, Docker crea una red independiente usando *network namespaces*, compuesta de su propia IP, rutas, *network devices*. Esto permite que el contenedor pueda interactuar con otro host a través de su propia interfaz. Por omisión, la conexión se realiza gracias al host que provee un *Virtual Ethernet bridge* en la máquina host, llamado docker0 que automáticamente realiza un *forward* de los paquetes entre las interfaces. Cuando Docker crea un nuevo contenedor, esto establece una interfaz de red virtual con un nombre único que se conecta con el *bridge (docker0)* y con la interfaz *eth0* del contenedor [36]. Además para controlan la cantidad de recursos como CPU, memoria, *disk I/O* que el contenedor puede utilizar se utiliza *Cgroups*. A partir de esto se protege contra ataques de DoS.

2.4. Clair

Clair es un proyecto de código abierto para el análisis estático de vulnerabilidades en contenedores de aplicaciones (actualmente incluyendo appc y docker).

En intervalos regulares, Clair ingiere metadatos de vulnerabilidad de un conjunto

configurado de fuentes y los almacena en la base de datos. Para analizar las imágenes, los clientes utilizan la API de Clair para indexar sus imágenes; esto crea una lista de características presentes en la imagen y las almacena en la base de datos.

Clair define su propia terminología:

Feature: cualquiera sea que éste presente en el sistema de archivos que sea una indicación de una vulnerabilidad (e.g. presencia de un archivo o paquete)

Feature Namespace: contexto de *feature* o vulnerabilidades (e.g. un sistema operativo o lenguaje)

Vulnerability Source (vulnsr): el componente de Clair que rastrea los datos de vulnerabilidad y los importa a la base de datos de Clair

Vulnerability Metadata Source (vulnmdsrc): el componente de Clair que rastrea los metadatos de vulnerabilidad y los asocia con las vulnerabilidades en la base de datos de Clair

Clair se componente de distintos controladores (*drivers*).

featurefmt: identifica el formato de las funcionalidades de una capa (e.g. apt).

featurens: identifica cuáles son los *namespaces* que son aplicables a la capa (e.g. ubuntu 16.04).

imagefmt: determina la ubicación del sistema de archivos raíz de la capa (e.g. Docker).

versionfmt: determina y compara los strings de la versión (e.g. rpm).

vulnmdsrc: descarga los metadatos de las vulnerabilidades para ser procesados (e.g. National vulnerability database - NVD [17]) .

vulnsrc: descarga las vulnerabilidades para un conjunto de *namespaces* (e.g. *Alpine Security Database of Backported fixes*)

La figura 2.1 muestra los drivers implementados por Clair.

[17] `https://nvd.nist.gov/`

Fuente de datos	Datos recolectados	Formato	Licencia
Debian Security Bug Tracker	*Namespaces:* Debian 6, 7, 8 y inestable	dpkg	Debian
Ubuntu CVE Tracker	*Namespaces:* Ubuntu 12.04, 12.10, 13.04, 14.04, 14.10, 15.04, 15.10, 16.04	dpkg	GPLv2
Red Hat Security Data	*Namespaces:* CentOS 5, 6, 7	rpm	CVRF
Oracle Linux Security Data	*Namespaces:* Oracle Linux 5, 6, 7	rpm	CVRF
Alpine SecDB	*Namespaces:* Alpine 3.3, Alpine 3.4, Alpine 3.5	apk	MIT
NIST NVD	Metadatos genéricos de vulnerabilidades	N/A	Public

Tabla 2.1: Fuente de datos implementadas por Clair

2.5. Sistemas de paquetes

Un gestor de paquetes es un conjunto de herramientas de software que automatiza el proceso de instalación, actualización, configuración y eliminación de programas informáticos para el sistema operativo de un ordenador de forma coherente.

Un gestor de paquetes se ocupa de paquetes, distribuciones de software y datos en ficheros de archivo. Los paquetes contienen metadatos, como el nombre del software, la descripción de su propósito, el número de versión, el proveedor, la suma de comprobación y una lista de dependencias necesarias para que el software funcione correctamente.

Tras la instalación, los metadatos se almacenan en una base de datos de paquetes local. Esta base de datos se guarda en un archivo en el sistema de archivos. Por lo tanto, la información de los componentes del software de una imagen se encuentra en cada una de sus capas. Dependiendo del sistema de paquetes, se puede utilizar otras herramientas para obtener mayor información utilizando la información que se encuentra en la base de datos

2.5.1. rpm

RPM Package Manager (RPM) es un sistema de empaquetado abierto, que se ejecuta en Red Hat Enterprise Linux así como en otros sistemas Linux y UNIX. El archivo de base de datos de rpm se encuentra ubicado en `/var/lib/rpm/Packages`. La información disponible en ese archivo es: Nombre, Versión, Lanzamiento, Arquitectura, Fecha de instalación, Grupo, Tamaño, Licencia, Firma, RPM de origen, Fecha de construcción, Host de reconstrucción, Proveedor del paquete, URL, Resumen y Descripción.

2.5.2. dpkg

Debian Package Manager (dpkg) es un sistema de gestión de paquetes en el sistema operativo libre Debian y sus numerosos derivados. dpkg se utiliza para instalar, eliminar y proporcionar información sobre los paquetes .deb.

El archivo de base de datos de dpkg se encuentra ubicado en `/var/lib/dpkg/status` utilizando formato texto plano. La información disponible es: Paquete, Estado, Prioridad, Sección, Tamaño de la instalación, Encargado del mantenimiento, Arquitectura, Fuente, Versión, Reemplaza, Dependencias y Recomendaciones.

2.5.3. apk

apk es un sistema de gestión de paquetes en el sistema operativo Alpine. apk se utiliza para instalar, eliminar y proporcionar información sobre los paquetes .apk. El archivo de base de datos de apk se encuentra ubicado en `/var/lib/rpm/Packages`. La información disponible se basa la especificación de apk [18]: Arquitectura, Suma del pull, Dependencias, Ruta de archivos, Tamaño, Licencia, Nombre del paquete, Descripción, URL, git commit del paquetes, tiempo de construcción, entre otros.

2.5.4. Conda

Conda es un gestor de paquetes, dependencias y entornos para lenguajes Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, FORTRAN y que es ampliamente utilizado en entornos de *Jupyter notebook*.

El archivo de base de datos de Conda se encuentra ubicado en `conda/history` según la posición del ambiente. La información disponible se basa la especificación de apk [19]: Nombre, versión, versión de construcción, número de construcción, dependencias, arquitectura, plataforma y archivos.

[18] `https://wiki.alpinelinux.org/wiki/Apk_spec`

[19] `https://conda.io/docs/user-guide/tasks/build-packages/package-spec.html`

2.6. Web Semántica

La web semántica es un extensión *World Wide Web* añadiendo un conjunto de estándares diseñado por la *World Wide Web Consortium* con el objetivo de la creación de tecnologías para publicar datos legibles por aplicaciones informáticas. Para lo lograr lo anterior, se añaden metadatos semánticos y ontologías para describir el contenido y generar relaciones entre los datos, esta representación se realiza de una manera formal. Consecuentemente, la información disponible pueden ser procesadas de manera automática mejorando la inter operatividad entre distintos sistemas informáticos que realizan búsquedas sin operadores humanos.

El término fue acuñado por Tim Berners-Lee para una red de datos que puede ser procesada por máquinas es decir, una red en la que gran parte del significado es legible por máquinas. En [37] se afirma la necesidad de que los datos se transformen desde objetos legible por personas a información semántica diseñada para las máquinas.

Es por ello que la web semántica ha definido diversos estándares para construir una descripción formal de los conceptos, términos y relaciones. Estos estándares definidos por W3C incluyen: RDF, RDFS, SPARQL, Notation3 (N3), N-Triples, Turtle y OWL.

2.6.1. RDF

RDF (del inglés *Resource Description Framework*) es un modelo estándar para el intercambio de datos en la Web. Fue adoptado como una recomendación del W3C en 1999, mientras que la especificación 1.0 fue publicada el 2004 y la 1.1 el 2014[38].

RDF extiende la estructura de enlace de la Web para usar URIs para nombrar la relación entre las cosas así como los dos extremos del enlace (esto se conoce como un "triple"). Un triple tiene la forma de expresión $\langle sujeto, predicado, objeto \rangle$. Donde el $sujeto$ indica el recurso, el $predicado$ nombra la relación entre el $sujeto$ y $elobjeto$

2.6.2. RDFS

RDFS (de las siglas del inglés *Resource Description Framework Schema*, también llamado RDF Schema) es un vocabulario que extiende RDF proveyendo un conjunto de clases y propiedades que mejoran la creación de modelos como son: `Class` para declarar clases, `subClassOf` para denotar herencia, `range` y `domain` para el rango y dominio

de cierta propiedad (`rdf:Property`), entre otras.

RDF fue presentado en 1998 e introducido finalmente como recomendación del W3C el 2004[38].

2.6.3. OWL

OWL (del inglés *Web Ontology Language*) es una familia de lenguajes para la creación de ontologías complejas. Agrega lógica computacional para que las relaciones hechas con este lenguaje puedan ser procesadas con el fin de verificar la consistencia de la información o generar información implícita.

La versión actual de OWL se conoce como "OWL 2" y fue publicada el 2009 como una revisión y extensión de la versión inicial publicada el 2004[38]. Generalmente cuando se habla de "OWL" nos referimos a la versión del 2004.

Capítulo 3

Objetivos de trabajo

El principal objetivo de este trabajo es complementar enfoques existentes para la reproducibilidad científica en el área de las ciencias de la computación. Para ello, se propone un nuevo enfoque para conservar el ambiente de ejecución del experimento científico. Hemos identificado problemas abiertos, en orden de definir los objetivos de trabajo. Luego, estos objetivos se ven formalizados por un conjunto de hipótesis. Y además, se define un conjunto de hechos que se asumen para restringir el campo de aplicación de la propuesta.

3.1. Problemas de investigación abierto

- Problema 1: La infraestructura computacional utilizada por un workflow científico se encuentra predefinida. Por lo tanto, no existe una definición de los recursos de la infraestructura para ejecutar el experimento. Consecuentemente, pueden existir dificultades para lograr la reproducción del experimento.

- Problema 2: La conservación física de los ambientes computaciones permite mantener y compartir fácilmente el ambiente con la comunidad. Sin embargo, se ha descartado debido a tres problemas: 1. Alta utilización de almacenamiento por parte de las máquinas virtuales, 2. El acceso a los datos almacenados está sujetos a políticas de la organización y 3. Existe un proceso de decaimiento en tiempo. Sin embargo, no se ha estudiado el uso de containers para solucionar el problema.

- Problema 3: Los enfoques actuales anotan los pasos de construcción de los ambientes

computacionales de forma indirecta, por lo tanto, recaen en el científico

3.2. Hipótesis

En función a los problemas abiertos detectados, se definen las siguientes hipótesis:

- Un proceso automático puede describir los requerimientos de ambiente computacional y codificarlo en un formato compartible utilizando modelos semánticos.
- La descripción de containers utilizando modelos semánticos permite la reproducción del ambiente de un experimento científico.

3.3. Objetivos

Para enfrentar los problemas abiertos se definen los siguientes objetivos. Estos objetivos permiten la verificación de la hipótesis y ser una guía para el desarrollo.

- Lograr conservación física y lógica de los ambientes computaciones de un experimento usando Containers
- Implementar un proceso automático capaz de leer la descripción del ambiente y especificar uno nuevo.
- Integrar un sistema que permita el despliegue de estos ambientes computacionales en proveedores de infraestructura e instalar el software apropiado basado al plan de despliegue.

3.3.1. Objetivos específicos

- Adaptar y mejorar modelos estándares que describen ambientes computacionales científicos para incluir virtualización basada en containers.
- Designar una framework para anotar los componentes de ambiente del experimento usando modelos semánticos.

3.4. Suposiciones

- Las técnicas de virtualización basadas en contenedores son una tecnología estable. Por lo tanto, las herramientas utilizadas estarán disponibles a largo plazo. Como sitios de almacenamiento de repositorio o soluciones de software que soportan la gestión de recursos virtualizados.
- Los componentes de software anotados será los cuáles sean instalados por sistemas de paquetes anteriormente nombrados. Los componentes de software que sean binarios serán solamente anotados en los pasos de construcción.

Capítulo 4

Conservación del ambiente de ejecución

En este trabajo, se argumenta que las descripciones de los ambientes computacionales es necesaria para la reproducción del experimento. Además, la información debe ser la suficiente para comparar y detectar las diferencias entre el ambiente original y el reproducido.

Dado que las imágenes Docker son ambientes aislados y independientes, los componentes de software dentro del contenedor si están relacionado al experimento y no existen componentes relacionados a otros experimentos. Por esta razón, se asegura que las anotaciones no presentarán ruido de otras herramientas o dependencias asociadas. Para realizar una anotación automática de los paquetes instalados, se propone e implementa un sistema de anotación automático. El sistema requiere el nombre de una imagen existente en un repositorio de DockerHub y opcionalmente los datos del repositorio Git que almacena el archivo Dockerfile.

Los pasos asociados del sistema anotación:

1. Consultar a repositorio de imágenes Docker público o privado los metadatos de la imagen. Anotar los metadatos utilizando la ontología propuesta.

2. Descargar cada una de las capas asociadas a la imagen Docker.

3. Sistema anotador consulta a Clair, Clair monta cada una de capas de la imagen, detecta los componentes de software instalados utilizando el sistema de paquete.

4. Obtener la información de Clair, anotar los datos obtenidos por Clair según la ontología.

5. Guardar los datos en una base de datos RDF que permita consulta SPARQL.

4.1. Modelos semánticos

En [6], los autores proponen *The Workflow Infrastructure Conservation Using Semantics ontology (WICUS)*. WICUS es ontología OWL2 (Web Ontology Language) que implementa la conceptualización de los principales dominios de la infraestructura computacional. Estos son: Hardware, Software, Workflow y recursos de computo Los autores definen que los workflows científicos requieren un conjunto de componentes de software, y los investigadores deben conocer cómo desplegar estos componentes para lograr ambiente equivalente. Considerando que la ontología está relacionada con nuestro trabajo, se utiliza algunas clases y relaciones desde la ontología WICUS:

DeploymentPlan: Un plan de despliegue está compuesto de todos los pasos. Y el plan de despliegue permite entender al investigador cuáles fueron los pasos requeridos para construir el ambiente computacional.

DeploymentStep: Cada paso de despliegue es representado por una línea de comando que realiza la instalación, descarga o configuración del ambiente. La información se obtiene desde el archivo DockerFile

SoftwareComponent: Es un componente de software instalado sin una versión específica.

Se define el recurso `dockerpedia:SoftwarePackage` como una subclase de `wicus:SoftwareComponent` para definir los componentes instalados por el gestor de paquetes del sistema operativo u otro. Cada `wicus:SoftwareComponent` tiene una relación `dockerpedia:hasVersion` o `dockerpedia:isVersionOf` que muestra la versión del software instalado. El recurso `dockerpedia:PackageVersion` puede estar afectado por vulnerabilidades representadas por el recurso `dockerpedia:SoftwareVulnerabi` Y las versiones que reparan la vulnerabilidad se encuentran representadas por `dockerPedia:Security` Una imagen Docker se representa por `dockerpedia:SoftwareImage` y sus capas por el recurso `dockerpedia:ImageLayer`. Los paquetes instalados en la imagen están relacionado por `vocab:ContainsSoftware` con su imagen. Finalmente, el sistema operativo es representado por recurso `dockerpedia:OperatingSystem` y cada imagen y capa de la imagen está asociado al recurso.

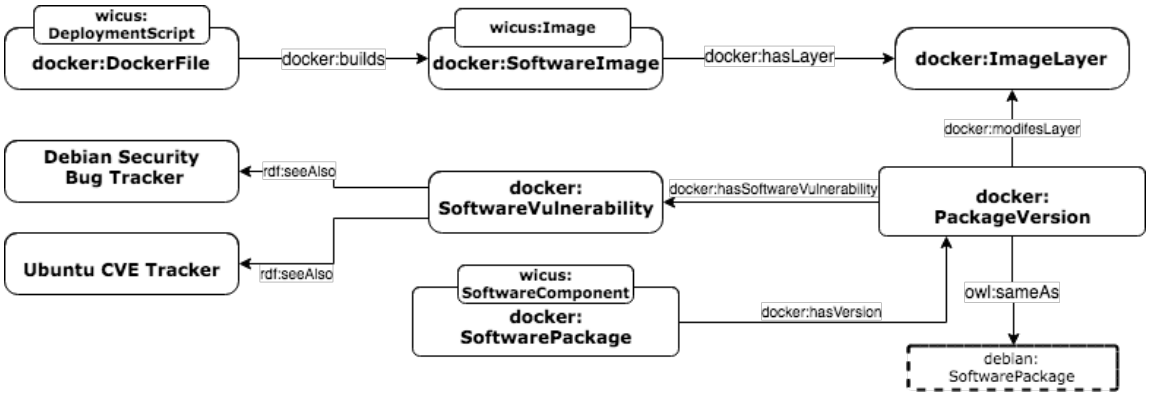


Figura 4.1: Ontología resumida para DockerPedia.

Una versión completa de la ontología se encuentra disponible en nuestros repositorios [1] y la versión resumida está representada en la figura 4.1.

4.2. Anotador

El servicio de anotación propuesto cuenta con una interfaz REST para recibir el nombre y la versión de la imagen Docker relacionada con el experimento científico. Para almacenar y interactuar con los datos el sistema de anotación utiliza a Apache Jena. Apache Jena es un framework Java gratuito y de código abierto para la construcción de aplicaciones de Web Semántica y Datos Enlazados. El sistema anotador construye los triples relacionado con el experimento, imagen, capas, componentes de software y vulnerabilidades y se envían usando con la API de Apache Jena que serializa los triples utilizando formatos populares como RDF/XML o Turtle. Esta decisión permite realizar cambios a otra herramienta en caso que Apache Jena no cumpla con los requerimientos de escalabilidad.

El sistema anotación realiza dos tipos de anotaciones: Componentes de software y pasos de construcción.

4.2.1. Anotaciones de los pasos de construcción

Anotamos el plan de despliegue (Deployment Plan) usando dos métodos. El primero método obtiene los pasos desde el archivo Dockerfile entregado por el usuario, esto permite obtener los pasos y la ubicación de archivo en el repositorio. Usando este método se asegura la reconstrucción del ambiente debido a que cualquier archivo necesario por Dockerfile se

[1] `https://github.com/dockerpedia/ontology/`

encuentra en el repositorio Git. Sin embargo, un usuario puede construir una imagen sin compartir el archivo Dockerfile. En [39] se reporta que el 30 % de las imágenes Docker en DockerHub vincula su archivo Dockerfile. Por lo tanto, sino existe información del Dockerfile el sistema anotación utiliza el manifiesto de la imagen Docker. Según [2] para conocer los pasos, el manifiesto de la imagen provee la configuración y el conjunto de las capas de la imagen

Algunos atributos importantes son:

name: *string* nombre de la imagen

tag: *tag* versión de la imagen

architecture: *string* arquitectura del servidor en cuál la imagen ha sido construido. Esta información actualmente no es utilizada por Docker.

fsLayers: *array* lista de las capas que componente la imagen. La estructura contiene los siguiente campos:

blobSum: es un identificador utilizando una función de hash sha256 para cada capa de la imagen

history : *array* Es una lista de datos históricos no estructurados para la compatibilidad con la v1. Contiene el ID de la capa de imagen y el ID de las capas principales de la capa. El historial es una estructura que consta de los siguientes campos:

v1Compatibility : *string* V1Compatibilidad es la información de compatibilidad de V1 en bruto. Esto contiene el objeto JSON que describe la V1 de esta imagen. Una V1Compatibilidad es una estructura que consta de los siguientes campos:

Id: *string* ID de la capa utilizando hash sha256

Parent: *string* ID de la capa madre utilizando hash sha256

ContainerConfig: *string* El comando que construyó la capa

4.2.2. Anotaciones de componentes de software

Para lograr la anotación de los componentes de software se utiliza los gestores de paquetes del sistema, un gestor de paquete es una colección de herramientas de software

[2] `https://docs.docker.com/registry/spec/manifest-v2-2/`

que automatizan la instalación, actualización, configuración y eliminación de componentes de software. Los gestores que paquetes se clasifican en dos tipos: gestor de paquetes del tipo sistema y general:

Gestor de paquetes de sistema: son aquellos vinculados al sistema operativo (e.g., apt de familia Debian, yum de familia RedHat).

Gestor de paquetes generales: son aquellos externos que normalmente son utilizados para instalar un componentes de software de tercero ó un lenguaje especifico (e.g., pip, conda, npm). Conda es un sistema paquete frecuentemente utilizado por investigadores al estar relacionado con Jupyter Notebook.

En la sección 2.4, se introdujo la descripción de sistema Clair, éste permite la detección componentes de software y vulnerabilidades dentro de una imagen Container. En este trabajo se utiliza Clair para detectar los componentes de software debido a que (i) no necesita ejecutar el container para detectar los componentes de software, (ii) no necesita instalar componentes de software extras dentro de la imagen y por lo tanto no se añade ruido al sistema, (iii) permite la extensión a otro tipo de sistema de contenedores como Singularity [40][3] y (iv) el análisis es realizado por capas, lo que permite re usar análisis de otras capas ya analizadas.

Además, como prueba de generalidad se extiende Clair para detectar los paquetes instalados por Conda en la imagen. Conda es un sistema de paquetes altamente utilizando por la comunidad científica dado a su fuerte relación con Jupyter Notebook. Sin embargo, Clair no cumple totalmente las necesidades para solucionar los problemas de anotación dado que Clair no considera múltiples detectores en la misma imagen. Por ejemplo, una imagen está compuesta de dos capas: A y B donde A es padre de B y Clair presenta dos detectores `Alpha` y `Beta` donde el detector `Alpha` detecta componentes de software en los archivos `/etc/a` y `Beta` detecta componentes de software en los archivos `/etc/b`

Si en la capa A, `Alpha` lista el software `1` en `/etc/a` y `Beta` lista `2` en `/etc/b`. Y luego, en la capa B, `Beta` lista `2` y `3` en `/etc/b`

El resultado de las diferencias será:

1. A añadió a `1` y `2`. Dado que observa el componentes de software en `/etc/a` y `/etc/b`

[3] `https://www.sylabs.io/docs/`

Listing 4.1: Ejemplo de instalación de dependencias para la imagen TensorFlow

```
apt-get install -y --no-install-recommends \
        build-essential \
        curl \
        libfreetype6-dev \
        libhdf5-serial-dev \
        libpng12-dev \
        libzmq3-dev \
        pkg-config \
        python \
        python-dev \
        rsync \
        software-properties-common \
        unzip
```

2. B añadió a `3` y se mantuvo `2` dado que observan en `/etc/b`. Y se removió `1` porque no hay archivo `/etc/a`

Pero si `/etc/a` no tuvo cambios, el archivo no se ve refleja en la capa hija. Y no significa que el `1` fuese removido. La casual del problema es que Clair añade y remueve paquetes sin considerar los detectores que detectaron los componentes de software. Consecuentemente, en esta propuesta se utiliza una copia del proyecto Clair que discrimina según el sistema de paquete que instaló el componente para permitir múltiples sistemas de paquetes de la imagen. El proyecto se encuentra disponible en el repositorio de DockerPedia [4].

4.2.3. Desafíos del sistema de anotación

La descripción de los componentes de software es fundamental para realizar cuantificar la similitud entre dos o más ambientes computacionales. Un enfoque común para detectar los componentes de software guardar o detectar los comandos que realizan la instalación de software. Por ejemplo, la figura 4.1 muestra los comandos para instalar los componentes de la imagen TensorFlow.

El enfoque detectaría los componentes: `build-essential`, `curl`, `libfreetype6-dev`, `libhdf5-serial-dev`, `libpng12-dev`, `libzmq3-dev`, `pkg-config`, `python`, `python-dev`, `rsync`, `software-properties-common` y `unzip`. Sin embargo,

[4] `https://github.com/dockerpedia/clair`

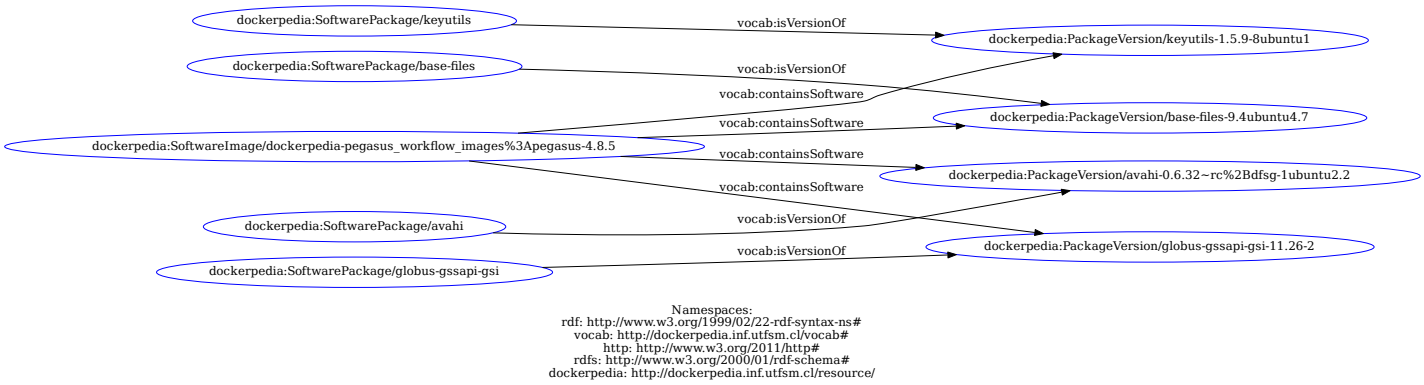


Figura 4.2: Ejemplo de triples que muestran los componentes de software de la imagen Pegasus

este enfoque no obtiene información sobre las versiones o las dependencias del software. Utilizando el enfoque propuesto, se puede terminar que el línea anterior instala 184 paquetes.

En la figura se muestra algunos de los paquetes instalado en la imagen de un workflow construido con Pegasus 4.2, en la figura 4.3 se muestra una vulnerabilidad de paquete glibc instalado en la imagen Pegasus.

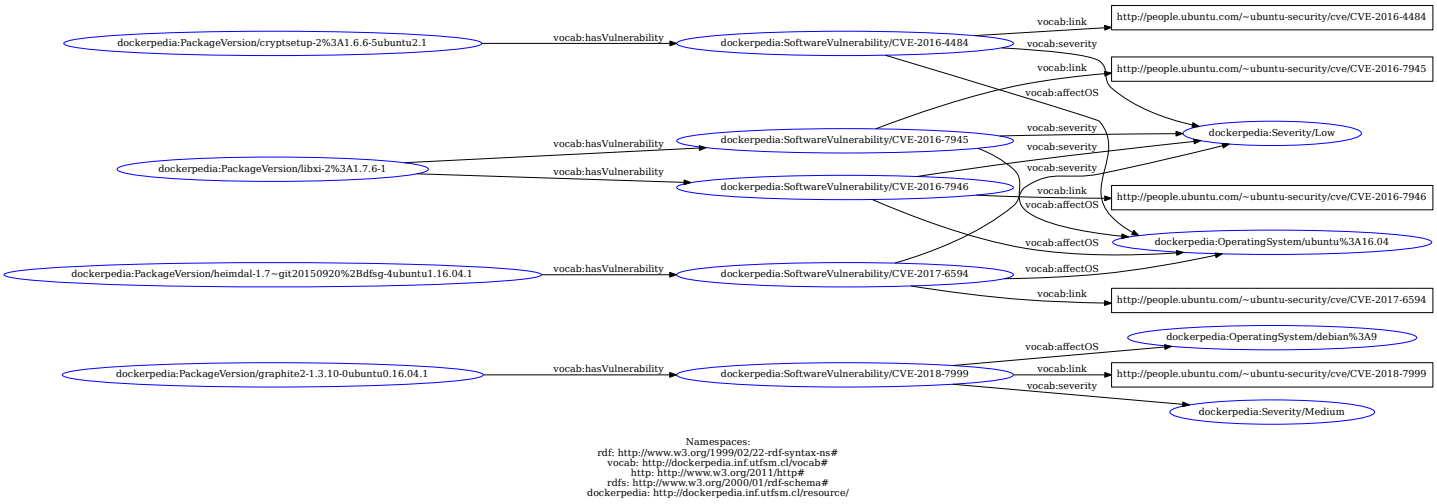


Figura 4.3: Vulnerabilidades y paquetes vulnerables de la imagen Pegasus

Capítulo 5

Experimentación y evaluación

5.1. Experimentos computacionales

Para validar nuestra propuesta y su aplicación en el contexto de workflows científicos, se ha seleccionado un subconjunto de workflows y WMSs. El subconjunto ha sido seleccionado bajo los criterio de: nivel de utilización de WMS, la diversidad de lenguajes utilizados en el conjunto, y disponibilidad de los materiales del workflow. Es requisito que los materiales deben incluir los datos de ingresos que permitan reproducir de forma completa el workflow, documentación, resultados o anotaciones. Además nos deben permitan comparar los resultados de nuestro ambientes resultantes.

Para cada WMS, hemos construimos una imagen estándar. En consecuencia, un investigador puede importar esta imagen e instalar los componentes de software relacionados con el workflow. Esto se puede conseguir utilizando la instrucción `FROM` de Dockerfile. Por ejemplo, el workflow SoyKB utiliza Pegasus como WMS. Por lo tanto, la imagen SoyKb importa la imagen Pegasus. Para entender los requisitos del workflow, nosotros inspeccionamos la documentación disponible y las anotaciones generadas por WICUS [6]. En caso que el WMSs no distribuya su software utilizando Docker, se construye las imágenes, los archivos necesarios como configuraciones y el DeploymentPlan (Dockerfile). Para cada flujo de trabajo los archivos están disponibles en nuestros repositorios. [1]. Además, cada DeploymentFile incluye información sobre la imagen utilizando el estándar de la Open Container Initiative [2] Esta información es:

[1] https://github.com/dockerpedia

[2] `https://www.opencontainers.org/`

org.opencontainers.image.created: fecha y hora en la que se construyó la imagen (string, fecha y hora según la definición de RFC 3339).

org.opencontainers.image.authors datos de contacto de las personas u organizaciones responsables de la imagen (lista de forma libre)).

org.opencontainers.image.url URL para encontrar más información sobre la imagen (string).

org.opencontainers.image.documentation URL para obtener documentación sobre la imagen (string).

org.opencontainers.image.source URL para obtener el código fuente para construir la imagen (string).

org.opencontainers.image.version versión del software empaquetado La versión puede coincidir con una etiqueta o tag en el repositorio de código fuente versión podría ser compatible con el versionado semántico.

org.opencontainers.image.revision Identificador de revisión de control de origen para el software empaquetado.

org.opencontainers.image.vendor Nombre de la entidad distribuidora, de la organización del artículo o del individuo.

org.opencontainers.image.licenses Licencia(s) bajo la(s) cual(es) el software contenido se distribuye.

org.opencontainers.image.ref.name Nombre de la referencia de un objetivo (string).

org.opencontainers.image.title Título de la imagen legible por el ser humano (string)

org.opencontainers.image.description Descripción legible por un ser humano del software empaquetado en la imagen (string)

5.1.1. Pegasus

Pegasus [21] es un WMS capaz de gestionar flujos de trabajo compuestos por millones de tareas, registrando datos sobre la ejecución y resultados intermedios. Cuando se producen

errores, Pegasus intenta recuperarse reintentando tareas, reintentando todo el flujo de trabajo, proporcionando puntos de control a nivel de flujo de trabajo, re-mapeando partes del flujo de trabajo, probando fuentes de datos alternativas para la puesta a disposición de los datos y, cuando todo lo demás falla, proporcionando un flujo de trabajo de rescate que contiene sólo una descripción del trabajo que queda por hacer. Limpia el almacenamiento a medida que se ejecuta el flujo de trabajo, de modo que los flujos de trabajo de datos intensivos tengan suficiente espacio para ejecutarse en recursos limitados por el almacenamiento]. Pegasus lleva un registro de lo que se ha hecho (procedencia), incluyendo las ubicaciones de los datos utilizados y producidos, y qué software se utilizó con qué parámetros. Pegasus lee las descripciones del flujo de trabajo de los archivos DAX. El término DAX.[es] la abreviatura de "Directed Acyclic Graph in XML". DAX es un formato de archivo XML que tiene sintaxis para expresar trabajos, argumentos, archivos y dependencias. Para crear un DAX es necesario escribir código para un generador de DAX. Pegasus opcionalmente utiliza en HTCondor como administrador de tareas. Por lo tanto, construimos dos versiones para la imagen de Pegasus; una versión tiene ins talado el paquete condor y otra sin él. La justificación de decisión recae en permitir a los científicos utilizar una imagen simple si es necesario. El paquete Pegasus ha sido obtenido del repositorio oficial [3]. Los principales requisitos de pegasus son: Java (la versión Java depende de la versión pegasus), Python y Perl. La figura 5.1 muestra las dependencias especificadas tanto por el sistema de paquetes y documentación.

Las imágenes se encuentran disponibles en DockerHub [4]

Soybean Knowledge Base

El workflow SoyKB (Soybean Knowledge Base) [41] permite realizar un proceso de re-secuencia de germoplasma de la soya, con el objetivo de estudiar rasgos como aceites, proteína, resistencia de los nematodos del quiste de la soya y resistencia al estrés. Pegasus entrega este workflow que implementa tres operaciones: polimorfismo de nucleótido único (SNP), la operación insertar/remover (indel) de la base del genoma del organismo y una análisis utilizando el software GATK [5] La figura 5.2 muestra una representación gráfica del workflow, donde se realiza un análisis en paralelo de las muestras. Primero son alineadas

[3] `http://download.pegasus.isi.edu/wms/download/debian`
[4] `https://hub.docker.com/r/dockerpedia/pegasus_workflow_images/`
[5] `https://software.broadinstitute.org/gatk/`

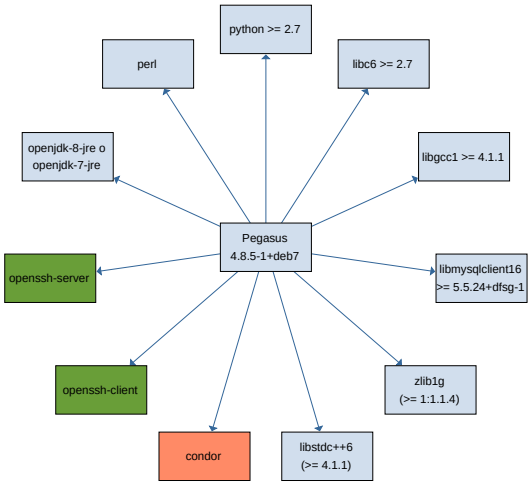


Figura 5.1: Dependencias de Pegasus. En azul: dependencias necesarias y especificadas en el sistema de paquetes. En verde: necesarias pero no especificadas en el sistema de paquetes. En naranjas: dependencias recomendadas

con el genoma de referencia, se identifica los indels y SNPs y luego se fusiona y filtra los resultados.

Los componentes de software utilizados por SoyKB se clasifican en dos tipos: propio (en amarillo) y de terceros (en verde). Los componentes principales son: bwa, gatk y picard y el software de tercero java-1.7.0

La evaluación de los resultados se realizó de forma manual al igual que en [6] debido a los pasos aleatorios del workflow. La metodología de la verificación fue la revisión de la estructura de los resultados, el tamaño de los archivos, número de líneas y la inexistencia de errores.

Montage

Montage es un conjunto de herramientas creadas por *NASA Infrared Processing and Analysis Center (IPAC)*, permitiendo generar bajo demanda mosaicos de imágenes astronómicas personalizadas, se utilizan archivos de entrada que cumplen con el estándar del Sistema de Transporte Flexible de Imágenes (FITS) y que contienen datos de imágenes registrados en proyecciones que cumplen con los estándares del Sistema Mundial de Coordenadas (WCS).

La figura 5.4 entregada por Pegasus ilustra el workflow Montage. Cada uno de los nodos de la figura es un software binario que generan la imagen final.

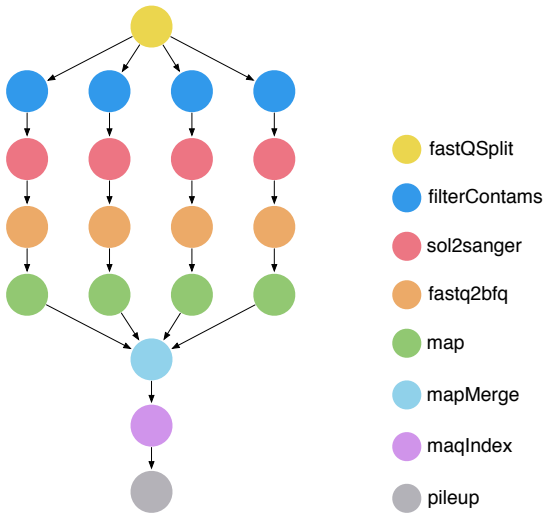


Figura 5.2: Representación entregada por Pegasus del workflow SoyKB.

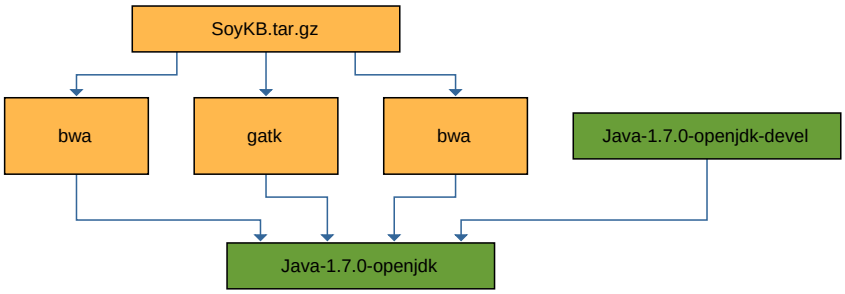


Figura 5.3: Dependencias de SoyKb. En amarillo: software propio y en verde software de terceros

Debido a que el software es un binario, no se encuentra empaquetado. Por lo tanto fue descargado de la fuente. La dirección de descarga se encuentran en el plan de despliegue [6].

Dado que los resultados de Montage son imágenes se utiliza la herramienta de hash perceptual [7] para realizar la comparación entre la imagen reproducida (imagen del cielo de 0,1 grados) frente a la original. Como resultado, se obtiene un factor de similitud de 1,0 (de 1,0) con un umbral de 0,85. Las figuras 5.5a y 5.5b muestran las imágenes resultantes y los archivos resultantes en formato FITS se encuentran en nuestro repositorio [8].

[6] https://github.com/dockerpedia/montage/blob/master/Dockerfile
[7] http://phash.org
[8] https://github.com/dockerpedia/montage_results

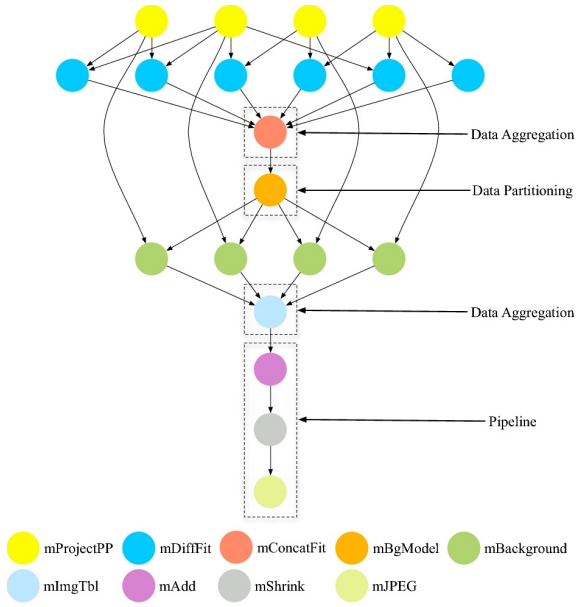


Figura 5.4: Representación entregada por Pegasus del workflow Montage. Cada uno de los nodos es una operación y la leyenda indica el nombre de la operación.

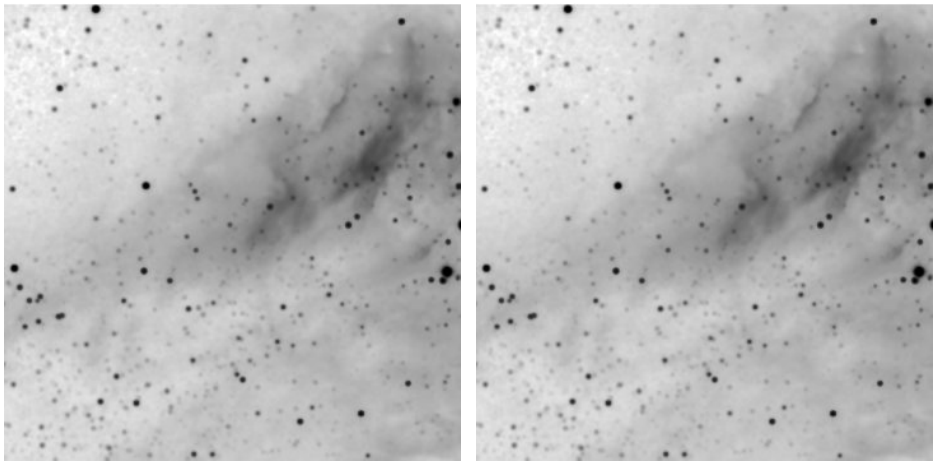
5.1.2. dispel4py

dispel4py [15] es una biblioteca Python para describir workflow para aplicaciones intensivas, que luego se traducen y ejecutan en plataformas distribuidas (por ejemplo, Apache Storm, clusters MPI, etc.). El paquete dispel4py ha sido obtenido del repositorio oficial [9]. Para la instalación el paquete, utilizamos Conda, un gestor de paquetes, dependencias y entornos para lenguajes Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, FORTRAN y que es ampliamente utilizado en entornos de *Jupyter notebook*. Los principales requisitos de dispel4py obtenidos de la documentación son: Python2.7, git y Python-setuptools Las imágenes se encuentran disponibles en DockerHub [10]

Internal extinction

Internal Extinction of Galaxies workflow calcula la extinción interna de galaxias desde el catalogo Amiga. Estos datos son obtenidos a partir de un servicio llamado Observatorio Virtual, que es una red de herramientas y servicios que implementan estándares publicados por la *International Virtual Observatory Alliance* (IVOA). El workflow calcula una propiedad, que representa la extinción de polvo dentro de las galaxias y que es un coeficiente de

[9] `https://github.com/rosafilgueira/dispel4py_workflows`
[10] `https://hub.docker.com/r/dockerpedia//`



(a) Resultados originales obtenidos desde WICUS

(b) Resultados reproducidos por nuestra propuesta

Figura 5.5: Los resultados obtenidos con el nuevo ambiente son iguales

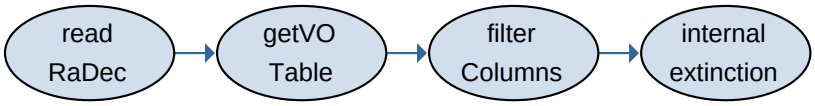


Figura 5.6: Pasos necesarios para el workflow Internal Extinction

corrección necesario para calcular la luminosidad óptica de una galaxia.

El workflow primero lee el archivo de entregada que contiene la declinación y ascensión recta de 1051 galaxias. Luego, utiliza estos valores para realizar consultas al observatorio virtual. Los valores resultantes de las consultas son filtrados y se seleccionan sólo los valores que correspondan al tipo morfológico (Mtype) y al rango de los ejes del isófito 25 $mag/arcsec^2$ (logr25) de las galaxias. Finalmente, se calcula su extinción interna. La figura 5.6 muestra los pasos del workflow

Nuestra investigación inicial sobre las dependencias para la ejecución de Internal extinction muestra que el software principal requeridos es el siguiente: `requests==2.20.0`, `python=2.7.15=h33da82c_4`, `numpy=1.15.0=py27h1b885b7_0` y `astropy==2.0.9`

Seismic Ambient Noise Cross-Correlation

Seismic Ambient Noise Cross-Correlation workflow (o xcorr workflow) es parte del proyecto *Virtual Earthquake and seismology Research Community e-science environment*

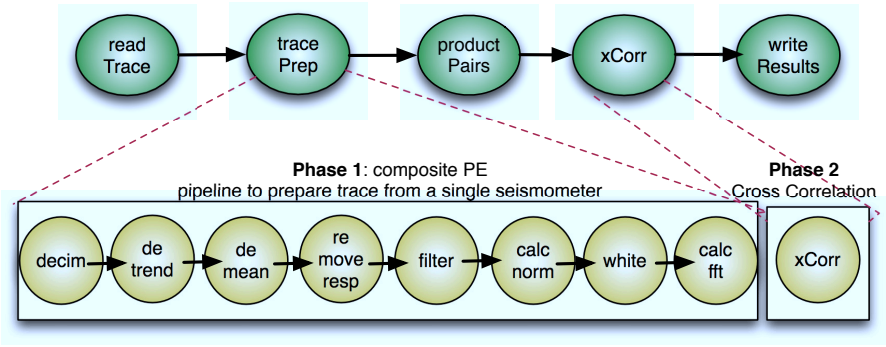


Figura 5.7: Representación workflow: Seismic Ambient Noise Cross-Correlation

in Europe (VERCE). El objetivo del workflow es la previsión de riesgos producidos por terremotos y erupciones volcánicas. Estos eventos en ciertos casos van precedidos o acompañados de cambios en las propiedades geofísicas de la Tierra, como la velocidad de las olas o la velocidad de los eventos.

Para lograr desarrollo de métodos fiables de evaluación de riesgos para estas amenazas requiere un análisis en tiempo real de los datos sísmicos y un pronóstico verdaderamente prospectivo y pruebas para reducir los sesgos.

`xcorr` logra lo anterior a través de dos etapas, la primera etapa es un pre-procesamiento de series de tiempo de una estación sísmica llamadas trazas, en esta etapa se realiza una serie de tratamientos y el procesamiento de cada traza es independiente del procesamiento de otras trazas, haciendo que esta fase sea paralela. Luego la segunda etapa empareja todas las estaciones y calcula la correlación cruzada para cada par. La figura 5.7 muestra los pasos del workflow.

Nuestra investigación inicial sobre las dependencias para la ejecución indica que los componentes de software necesarios son: `python=2.7.15=h33da82c_4`, `obspy=1.1.0=py27h3` y `numpy=1.15.0=py27h1b885b7_0`

5.1.3. WINGS

WINGS es un sistema de flujo de trabajo semántico que ayuda a los científicos en el diseño de experimentos computacionales. En WINGS, se especifica cómo deben ser procesados los conjuntos de datos por una serie de componentes del software en una configuración particular. WINGS a diferencia de los otros sistemas de workflows utiliza imágenes Docker para su distribución. Nuestra investigación inicial sobre las dependencias

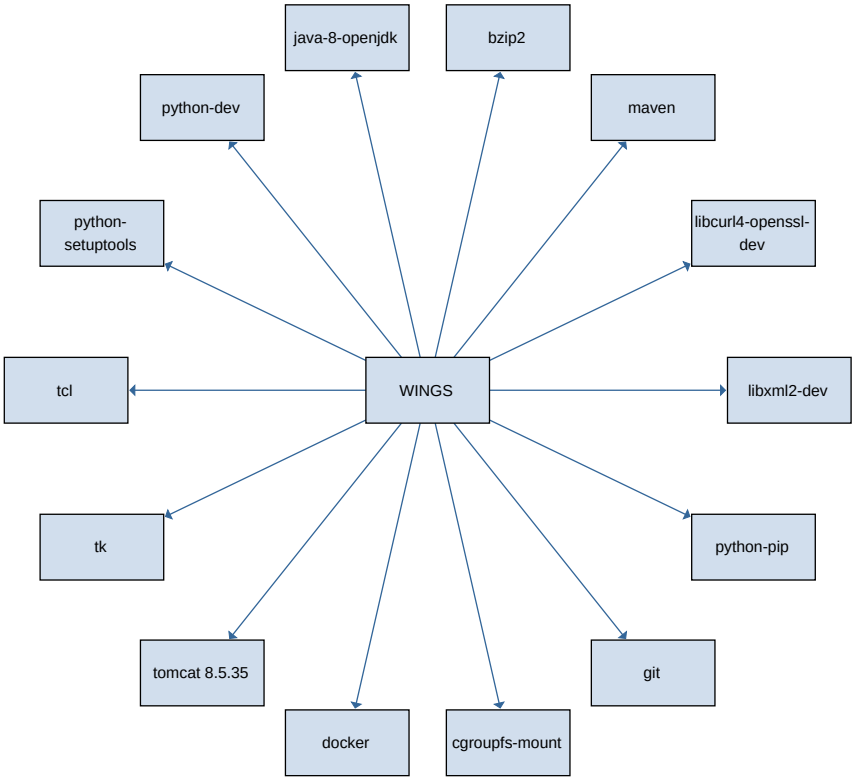


Figura 5.8: Dependencias de WINGS

indica que los principales requisitos de WINGS son: Java 1.8, Tomcat 8.5, Docker. La figura 5.8 muestra en mayor detalles las dependencias especificadas tanto por el sistema de paquetes o documentación.

Las imágenes se encuentran disponibles en DockerHub [11]

MODFLOW-NWT

MODFLOW es el modelo hidrológico modular del *United States Geological Survey* (USGS). MODFLOW se considera una norma internacional para simular y predecir las condiciones de las aguas subterráneas y las interacciones entre las aguas subterráneas y superficiales. El USGS MODFLOW-NWT es una formulación de Newton-Raphson para MODFLOW-2005 con el objetivo de mejorar la solución de problemas de flujo, secado y humectación de aguas subterráneas no confinadas.

La figura 5.9 muestra la estructura principal del workflow, desarrollado con pipeline

[11] `https://hub.docker.com/r/dockerpedia/wings/`

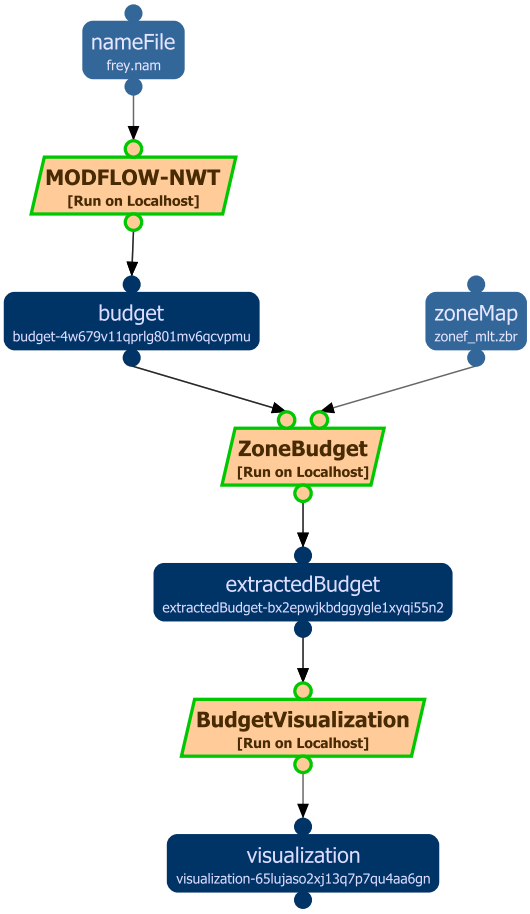


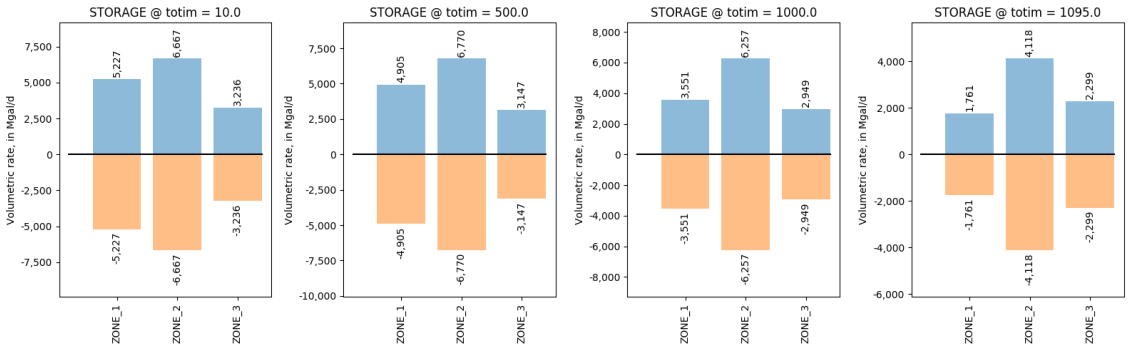
Figura 5.9: Representación workflow: MODFLOW-NWT

compuesto por tres pasos. El proceso inicia leyendo el modelo a utilizar. Luego, el archivo de zona se usa para especificar arreglos de zonas que van usarse y finalmente se genera una visualización que muestra la cantidad de millones de galones por día en zona. Las figuras 5.10a y 5.10b muestran los resultados generados por el ambiente original y reproducido respectivamente.

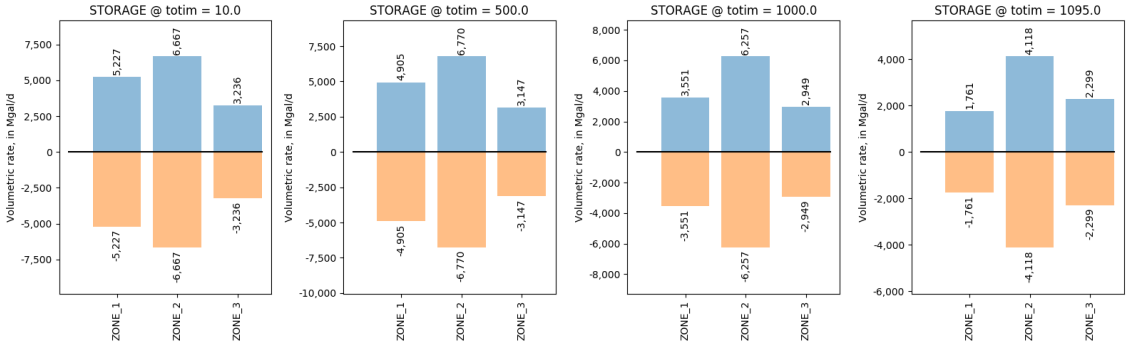
5.2. Conservación física

Todas las imágenes son publicada con su correspondiente Dockerfile y cualquier usuario puede inspeccionarlas y mejorarlas. Las imágenes puede ser encontradas en DockerHub [12] y GitHub [13].

[12] https://hub.docker.com/u/dockerpedia/
[13] `https://github.com/dockerpedia`



(a) Resultados originales entregados por el Information Sciences Institute



(b) Resultados reproducidos por nuestra propuesta

Figura 5.10: Los resultados obtenidos son idénticos

Para evaluar la conservación física se utiliza tres proveedores diferentes: DigitalOcean, Google Cloud y un local. La figura de 5.1 presenta una descripción de las características de los ambientes

El ambiente debe tener instalado Docker, cada imagen en su manifiesto describe la versión de Docker con cuál fue construida. Sin embargo, Docker asegura idempontencia para los ambientes CentOS7, Debian 10/9/8/7.7, Fedora 26/27/28, Ubuntu 14.04/16.06/18.04, Windows 10, macOS El Capitan 10.11 o nuevas versiones. El proceso de instalación puede ser encontrado en la documentación oficial [14].

Cada imagen de Docker tiene archivo README con las instrucciones para correr el experimento. Las figuras 5.1 5.2 y 5.3 muestran los pasos necesarios para correr el experimento computacional.

El primer paso es correr el experimento y descargar la imagen:

Luego, el usuario debe entrar al container. El usuario puede confirmar que se encuentra dentro del container por el cambio de símbolo de la terminal (prompt).

[14] `https://docs.docker.com/install/`

Resource	Digital Ocean	Google Compute	Local
RAM (GB)	8	8	4
Disk (GB)	100	100	70
CPU (GHz)	2.0	2.5	2.8
CPU (Cores)	4	2	4
CPU (Arch)	64	64	64
OS	Centos 7	Debian 9	Fedora 27

Tabla 5.1: Características de hardware de pruebas.

Listing 5.1: Descargar y correr la imagen disponible en DockerHub mosorio/pegasus_workflow_images:soykb

```
docker run -d --rm -it --name soybean \
 mosorio/pegasus_workflow_images:soykb
```

Finalmente, correr el workflow.

Para evaluar si las imágenes Docker son livianas y almacenables, se construyó dos imágenes, una utilizando Docker y otra utilizando máquinas virtuales. La imagen de Docker se construyó bajo nuestro enfoque y la imagen de la máquina virtual basada en el trabajo de [6]. Luego se compara el uso de disco de ambas.

5.3. Conservación lógica

A través de las anotaciones realizadas, se busca describir los ambientes computacionales en forma automática, comparar las diferencias entre dos ambientes y construir un ambiente computacional similar que permita reproducir la ejecución del workflow.

Para ello, se anota de automática los workflows con nuestra herramienta, las anotaciones están agrupadas por: pasos de construcción y componentes de software.

Para evaluar la calidad de las anotaciones se utilizan dos experimentos.

- Reproducir el ambiente utilizando los pasos de construcción representados por DeploymentPlan.
- Detectar las similitudes y diferencias entre dos ambientes computacionales.

Para obtener las anotaciones, se propone e implementa usar Clair y construir un sistema de anotador. La figura 5.11 muestra los pasos principales del sistema.

Listing 5.2: Entrar al ambiente computacional utilizando bash

```
root@docker-instance:~# docker exec \
-ti -u workflow:workflow soybean bash
workflow@a0f861e6fbc4:~
```

Listing 5.3: Run the workflow

```
workflow@a0f861e6fbc4:~/soykb \
./workflow-generator --exec-env distributed
```

1. El investigador pregunta sobre la información de una imagen al sistema anotador. El sistema anotador se encuentra escrito en GoLang y disponible en nuestro repositorio.

2. El sistema anotador pregunta a Clair sobre software y sus vulnerabilidades de la imagen. El sistema de Clair es una versión propia que puede detectar componentes de software instalado por Conda.

3. Para obtener los pasos de construcción, etiquetas, arquitectura y más información. El anotador obtiene el manifiesto de la imagen desde DockerHub

4. El sistema anotador guarda la información usando RDF y ontología propuesta.

Para reproducir el entorno, el sistema obtiene el repositorio asociado a la imagen y construir la nueva imagen. La URL del repositorio y el cambio asociado (representado por un VCS commit) se pueden obtener por dos métodos: consultando nuestras anotaciones o usando un comando Docker. El listado 5.4 muestra las etiquetas de la imagen de flujo de trabajo SoyKB.

La figura 5.5 muestra algunas de las etiquetas de la imagen basado en *Open Container Initiative*. Y todas las imágenes disponibles en nuestros repositorios presentan esas etiquetas.

Para evaluar si los entornos son similares, comparamos ambas imágenes utilizando el lenguaje de consulta SPARQL 1.1. El experimento fue un caso real en el que una imagen podía ejecutar el flujo de trabajo y la otra no. La figura 5.6 muestra la consulta para identificar los componentes diferentes de software entre dos imágenes y la figura 5.7 muestra muestra la consulta para identificar los componentes iguales de software entre dos imágenes.

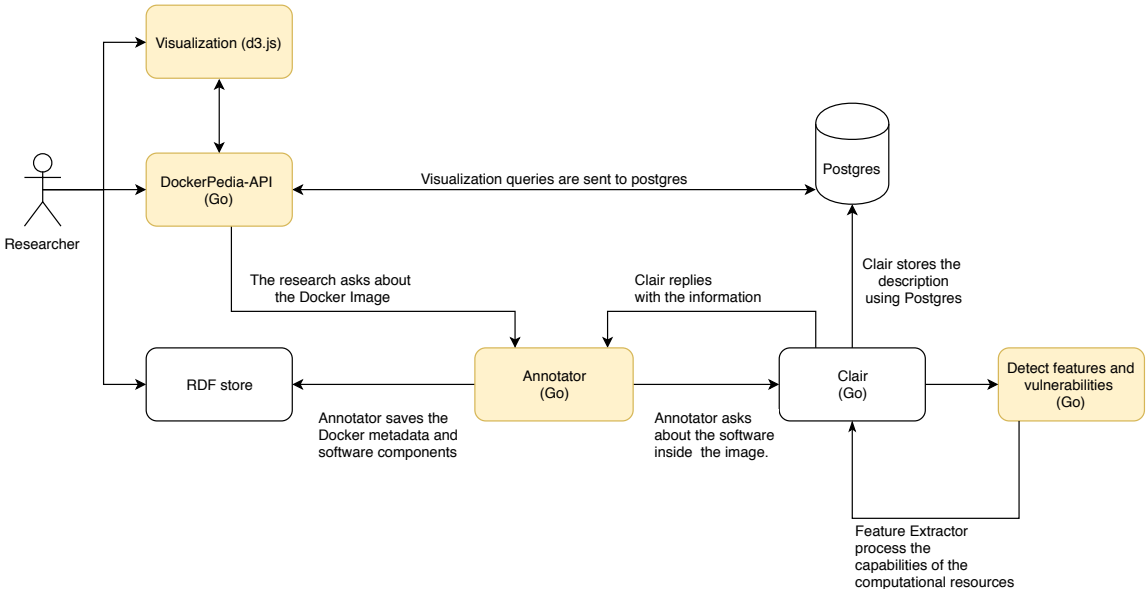


Figura 5.11: El sistema anotador permite recibir la información de la imagen, consultar los componentes de software a nuestra versión de Clair, obtener metadatos desde el manifesto y almacenar la información en forma de triples en Apache Jena.

Listing 5.4: Inspect image annotations

```
root@docker-instance:~# docker inspect \
    --format='{{json .Config.Labels}}' \
    dockerpedia/soykb:latest
```

5.4. Resultados y discusión

Se logró ejecutar los workflows utilizando imágenes Docker sobre sus plataformas correspondientes. Todas las ejecuciones se compararon con la imagen VM predefinida, donde ya existía un entorno de ejecución. Los resultados muestran que los ambientes de ejecución de contenedores son capaces de ejecutar completamente sus workflow relacionados. Se comprobó que no sólo los workflows se ejecutan con éxito, sino también que los resultados son correctos y equivalentes a través de los datos de salida producidos.

Por otra parte, los resultados experimentales muestran que nuestra propuesta puede detectar automáticamente los componentes del software, las vulnerabilidades relacionadas, los pasos de construcción y los metadatos específicos de los experimentos científicos en forma de imágenes Docker. Además, los resultados muestran que es posible extender Clair para anotar a otros gestores de paquetes.

Listing 5.5: Inspect image annotations

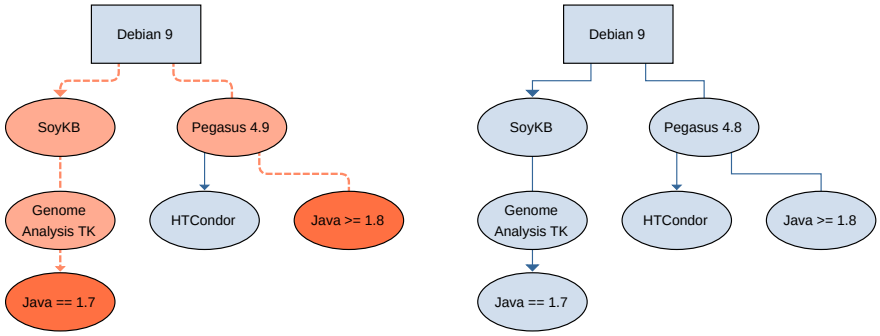
```
{
"maintainer": "Maximiliano Osorio <mosorio@inf.utfsm.cl>"
   ,
"org.label-schema.build-date": "2018-11-10T21:11:28Z",
"org.label-schema.name": "Soybean Knowledge Base",
"org.label-schema.schema-version": "1.0",
"org.label-schema.url": "http://www.soykb.org/",
"org.label-schema.vcs-ref": "15955b0",
"org.label-schema.vcs-url": "https://github.com/
   dockerpedia/soykb",
"org.label-schema.vendor": "DockerPedia",
"org.label-schema.version": "1.0"
}
```

Listing 5.6: ¿Cuáles son los diferentes componentes entre dos imágenes?

```
SELECT * WHERE {
 pegasus_workflow_images%3Alatest
  vocab:containsSoftware ?p .
 MINUS{
 pegasus_workflow_images%3Apegasus-4.8.5
  vocab:containsSoftware ?p
 }
}
```

Listing 5.7: ¿Cuáles son los componentes que comparten entre dos imágenes?

```
SELECT * WHERE {
 pegasus_workflow_images%3Alatest
      vocab:containsSoftware ?p .
 pegasus_workflow_images%3Apegasus-4.8.5
      vocab:containsSoftware ?p
    }
```



(a) Gráfico de dependencias Pegasus 4.9 y SoyKB

(b) Gráfico de dependencias Pegasus 4.8 y SoyKB

Figura 5.12: Análisis de dependencias imágenes Pegasus

Las anotaciones generadas por nuestro enfoque permiten comparar los componentes de software entre dos o más entornos. Esta función se puede utilizar como herramienta de depuración cuando un entorno reproducido no funciona. Por ejemplo, en agosto de 2018, se construyó la imagen del workflow SoyKB, y se pudo ejecutar el flujo de trabajo con éxito. Sin embargo, se reconstruyó una nueva imagen en noviembre con el mismo DeploymentPlan y no se pudo ejecutar el workflow con éxito.

Se utilizó las anotaciones de los componentes de software dentro de ambas imágenes. Y se encontró las siguientes diferencias:

- Agosto: Pegasus 4.8 y Java 1.7
- Noviembre: Pegasus 4.9 y Java 1.8

Luego se analizó el código y la documentación de SoyKB y las dependencias de Pegasus 4.9. Como resultado, se obtuvo las gráficas de dependencia mostradas en la figura 5.12a. Los gráficos de dependencia muestran que el paquete Pegasus 4.9 y SoyKB no son compatibles debido a sus requerimientos de versión Java.

Y construyó una nueva imagen instalando Pegasus 4.8, y se obtuvo los gráficos de dependencia que se muestran en la figura 5.12b. Aquí, el gráfico no tiene un conflicto y se pudo ejecutar el workflow con éxito.

La nueva imagen fue nombrada como: `pegasus_workflow_images:4.8.5`

La razón principal de los trabajos anteriores para evitar la conservación física era la gran demanda de almacenamiento de máquinas virtuales. Los resultados de la comparación

Enfoque	Pegasus (MB)	dispel4py (MB)
Virtualization	1929	3509
Container	690	2500

Tabla 5.2: Uso de disco de las imágenes pegasus y dispel4py utilizando máquinas virtuales y contenedores

del uso del disco muestran que hay una disminución del 64,2 % para la imagen Pegasus y del 41,5 % para la dispel4py. La tabla 5.2 muestra la diferencia para el sistema de flujo de trabajo Pegasus y dispel4py.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Main 1

m at, molestie in quam. Aenean rhoncus vehicula hendrerit.

Bibliografía

[1] K. A. Baggerly and K. R. Coombes, "Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology," *The Annals of Applied Statistics*, pp. 1309–1334, 2009.

[2] J. P. Ioannidis, D. B. Allison, C. A. Ball, I. Coulibaly, X. Cui, A. C. Culhane, M. Falchi, C. Furlanello, L. Game, G. Jurman, *et al.*, "Repeatability of published microarray gene expression analyses," *Nature genetics*, vol. 41, no. 2, p. 149, 2009.

[3] O. S. Collaboration *et al.*, "Estimating the reproducibility of psychological science," *Science*, vol. 349, no. 6251, p. aac4716, 2015.

[4] V. C. Stodden, "Reproducible research: Addressing the need for data and code sharing in computational science," *Computing in science & engineering*, vol. 12, no. 5, pp. 8–12, 2010.

[5] D. Garijo, S. Kinnings, L. Xie, L. Xie, Y. Zhang, P. E. Bourne, and Y. Gil, "Quantifying reproducibility in computational biology: the case of the tuberculosis drugome," *PloS one*, vol. 8, no. 11, p. e80278, 2013.

[6] I. Santana-Perez, R. F. da Silva, M. Rynge, E. Deelman, M. S. Pérez-Hernández, and O. Corcho, "Reproducibility of execution environments in computational science using semantics and clouds," *Future Generation Computer Systems*, vol. 67, pp. 354–367, 2017.

[7] A. Dappert, S. Peyrard, C. C. Chou, and J. Delve, "Describing and preserving digital object environments," *New Review of Information Networking*, vol. 18, no. 2, pp. 106–173, 2013.

[8] T. Craddock, P. Lord, C. Harwood, and A. Wipat, "E-science tools for the genomic scale characterisation of bacterial secreted proteins," in *All hands meeting*, pp. 788–795, 2006.

[9] D. Blankenberg, G. V. Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor, "Galaxy: a web-based genome analysis tool for experimentalists," *Current protocols in molecular biology*, pp. 19–10, 2010.

[10] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, *et al.*, "Galaxy: a platform for interactive large-scale genome analysis," *Genome research*, vol. 15, no. 10, pp. 1451–1455, 2005.

[11] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientific workflows onto the grid," in *Grid Computing*, pp. 11–20, Springer, 2004.

[12] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome biology*, vol. 11, no. 8, p. R86, 2010.

[13] T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[14] Y. Gil, V. Ratnakar, J. Kim, P. A. González-Calero, P. T. Groth, J. Moody, and E. Deelman, "Wings: Intelligent workflow-based design of computational experiments," *IEEE Intelligent Systems*, vol. 26, no. 1, pp. 62–72, 2011.

[15] R. Filgueira, A. Krause, M. P. Atkinson, I. A. Klampanos, A. Spinuso, and S. Sanchez-Exposito, "dispel4py: An agile framework for data-intensive escience," in *11th IEEE International Conference on e-Science, e-Science 2015, Munich, Germany, August 31 - September 4, 2015*, pp. 454–464, IEEE Computer Society, 2015.

[16] J. Zhao, J. M. Gómez-Pérez, K. Belhajjame, G. Klyne, E. García-Cuesta, A. Garrido, K. M. Hettne, M. Roos, D. D. Roure, and C. A. Goble, "Why workflows break - understanding and combating decay in taverna workflows," in *8th IEEE International*

Conference on E-Science, e-Science 2012, Chicago, IL, USA, October 8-12, 2012, pp. 1–9, IEEE Computer Society, 2012.

[17] B. Matthews, E. Conway, J. Woodcock, C. Jones, J. Bicarregui, and A. Shaon, "Towards a methodology for software preservation," in *Proceedings of the 6th International Conference on Digital Preservation, iPRES 2009, San Francisco, CA, USA, October 5-6, 2009*, 2009.

[18] G. R. Brammer, R. W. Crosby, S. Matthews, and T. L. Williams, "Paper mâché: Creating dynamic reproducible science," in *Proceedings of the International Conference on Computational Science, ICCS 2011, Nanyang Technological University, Singapore, 1-3 June, 2011* (M. Sato, S. Matsuoka, P. M. A. Sloot, G. D. van Albada, and J. J. Dongarra, eds.), vol. 4 of *Procedia Computer Science*, pp. 658–667, Elsevier, 2011.

[19] P. Buncic, C. A. Sanchez, J. Blomer, L. Franco, A. Harutyunian, P. Mato, and Y. Yao, "Cernvm–a virtual software appliance for lhc applications," in *Journal of Physics: Conference Series*, vol. 219, p. 042003, IOP Publishing, 2010.

[20] F. S. Chirigati, D. E. Shasha, and J. Freire, "Reprozip: Using provenance to support computational reproducibility," in *5th Workshop on the Theory and Practice of Provenance, TaPP'13, Lombard, IL, USA, April 2-3, 2013* (A. Meliou and V. Tannen, eds.), USENIX Association, 2013.

[21] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, and R. K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Comp. Syst.*, vol. 46, pp. 17–35, 2015.

[22] F. da Veiga Leprevost, B. A. Grüning, S. Aflitos, H. L. Röst, J. Uszkoreit, H. Barsnes, M. Vaudel, P. Moreno, L. Gatto, J. Weber, M. Bai, R. C. Jimenez, T. Sachsenberg, J. Pfeuffer, R. V. Alvarez, J. Griss, A. I. Nesvizhskii, and Y. Pérez-Riverol, "Biocontainers: an open-source and community-driven framework for software standardization," *Bioinformatics*, vol. 33, no. 16, pp. 2580–2582, 2017.

[23] B. K. Beaulieu-Jones and C. S. Greene, "Reproducibility of computational workflows is automated using continuous analysis," *Nature biotechnology*, vol. 35, no. 4, p. 342, 2017.

[24] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, pp. 71–79, Jan. 2015.

[25] M. E. Aranguren and M. D. Wilkinson, "Enhanced reproducibility of sadi web service workflows with galaxy and docker," *GigaScience*, vol. 4, no. 1, p. 59, 2015.

[26] M. Osorio, C. B. Aranda, and H. Vargas, "Reproducibility of computational environments for scientific experiments using container-based virtualization," in *Proceedings of the Second Workshop on Enabling Open Semantic Science co-located with 17th International Semantic Web Conference, SemSci@ISWC 2018, Monterey, California, USA, October 8-12th, 2018.* (D. Garijo, N. Villanueva-Rosales, T. Kuhn, T. Kauppinen, and M. Dumontier, eds.), vol. 2184 of *CEUR Workshop Proceedings*, pp. 43–51, CEUR-WS.org, 2018.

[27] R. Shu, X. Gu, and W. Enck, "A Study of Security Vulnerabilities on Docker Hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, (New York, NY, USA), pp. 269–280, ACM, 2017.

[28] D. Huo, J. Nabrzyski, and C. Vardeman, "Smart container: an ontology towards conceptualizing docker.," in *International Semantic Web Conference (Posters & Demos)*, 2015.

[29] A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider, "Sweetening ontologies with dolce," in *International Conference on Knowledge Engineering and Knowledge Management*, pp. 166–181, Springer, 2002.

[30] R. Tommasini, B. De Meester, P. Heyvaert, R. Verborgh, E. Mannens, and E. Della Valle, "Representing dockerfiles in RDF," in *ISWC2017, the 16e International Semantic Web Conference*, vol. 1931, pp. 1–4, 2017.

[31] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *technology*, vol. 28, p. 32, 2014.

[32] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[33] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin, *et al.*, "Performance evaluation of virtualization technologies for server consolidation," *HP Labs Tec. Report*, 2007.

[34] N. Regola and J.-C. Ducom, "Recommendations for virtualization technologies in high performance computing," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 409–416, IEEE, 2010.

[35] V. Marmol, R. Jnagal, and T. Hockin, "Networking in containers and container clusters," *Proceedings of netdev 0.1, February*, 2015.

[36] T. Bui, "Analysis of docker security," *arXiv preprint arXiv:1501.02967*, 2015.

[37] T. J. Berners-Lee, "The world-wide web," *Computer Networks and ISDN Systems*, vol. 25, no. 4, pp. 454–459, 1992.

[38] Bikakis, Tsinaraki, Gioldasis, Stavrakantonakis, and Christodoulakis, "The xml and semantic web worlds: Technologies, interoperability and integration. a survey of the state of the art," in *Semantic Hyper/Multi-media Adaptation: Schemes and Applications*, Springer, 2013.

[39] M. Osorio, C. B. Aranda, and H. Vargas, "Dockerpedia: a knowledge graph of docker images," in *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - 12th, 2018.* (M. van Erp, M. Atre, V. López, K. Srinivas, and C. Fortuna, eds.), vol. 2180 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018.

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

[41] T. Joshi, K. Patil, M. R. Fitzpatrick, L. D. Franklin, Q. Yao, J. R. Cook, Z. Wang, M. Libault, L. Brechenmacher, B. Valliyodan, *et al.*, "Soybean knowledge base (soykb): a web resource for soybean translational genomics," in *BMC genomics*, vol. 13, p. S15, BioMed Central, 2012.