# Unified Management of Applications on Heterogeneous Clouds

PhD Thesis

Jose Manuel Carrasco Mora
Programa de Doctorado de Tecnologías Informáticas
Departamento de Lenguajes y Ciencias de la Computación
ETS Ingeniería Informática
Universidad de Málaga

Supervised by
*Dr. Francisco Javier Durán Muñoz*
*Dr. Ernesto Pimentel Sánchez*

May 2021

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña JOSE MANUEL CARRASCO MORA
Estudiante del programa de doctorado TECNOLOGÍAS INFORMÁTICAS de la Universidad de Málaga, autor/a de la tesis, presentada para la obtención del título de doctor por la Universidad de Málaga, titulada: UNIFIED MANAGEMENT OF APPLICATIONS ON HETEROGENEOUS CLOUDS

Realizada bajo la tutorización de ERNESTO PIMENTEL SÁNCHEZ y dirección de FRANCISCO JAVIER DURÁN MUÑOZ Y ERNESTO PIMENTEL SÁNCHEZ (si tuviera varios directores deberá hacer constar el nombre de todos)

DECLARO QUE:

La tesis presentada es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, conforme al ordenamiento jurídico vigente (Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), modificado por la Ley 2/2019, de 1 de marzo.

Igualmente asumo, ante a la Universidad de Málaga y ante cualquier otra instancia, la responsabilidad que pudiera derivarse en caso de plagio de contenidos en la tesis presentada, conforme al ordenamiento jurídico vigente.

En Málaga, a 27 de MAYO de 2021

[illegible]

Fdo.: JOSE MANUEL CARRASCO MORA

Sobre la tesis doctoral con título "Unified Management of Applications on Heterogeneous Clouds", realizada por Jose Manuel Carrasco Mora, los profesores Ernesto Pimentel Sánchez y Francisco Javier Durán Muñoz, del Departamento de Lenguajes y Ciencias de la Computación d ela Universidad de Málaga:

- Confirman la idoneidad de la tesis para su presentación por compendio de publicaciones.

- Afirman que ni las publicaciones que avalan la tesis, ni las que forman parte de la misma, han sido utilizadas en tesis anteriores.

- Autorizan su lectura.

Málaga, 27 de mayo de 2021

[illegible] [illegible]

Ernesto Pimentel Sánchez
(tutor y co-director de la tesis)

Francisco Javier Durán Muñoz
(co-director de la tesis)

## Acknowledgements

We are grateful to our partners in the SeaClouds project, and in particular to our colleagues Alex Heneveld, Andrea Turli, and the rest of Cloudsoft Inc., and Francesco D'Andria and Roi Sucasas from Atos Spain. This work has been partially supported by EU project FP7-610531 SeaClouds; Spanish MINECO/FEDER projects TIN2014-52034-R and TIN2015-67083-R; Andalusian Gov project P11-TIC-7659; and Univ. Málaga, Campus de Excelencia Internacional Andalucía Tech.

# Special Acknowledgements

Antes de comenzar me gustaría mencionar a las personas que de alguna manera me han ayudado a llegar hasta aquí.

La tesis comenzó formalmente hace ya bastantes años, seis para ser exactos, y la verdad es que llegados a este punto a mis directores, Paco y Ernesto, solo puedo ofrecerles mi más sincero agradecimiento por haberme dedicado todo el tiempo, la energía , la confianza y la paciencia que he necesitado para completar este trabajo. Me gusta decir que ha sido un placer trabajar con vosotros, pero en realidad lo que ha sido es un privilegio. Gracias.

La verdad es que esta tesis, supongo que como todas las demás, ha sido un camino largo con muchos altibajos. Aquí quiero agradecer una cosa a mis padres, Jose y Loli, porque además de todo lo que me han dado, creo que me han enseñado una de las armas más útiles que he tenido para enfrentarme a este trabajo, la constancia. Sin vuestro apoyo esto no habría sido posible. Esto va también por el resto de mi familia, Jesús, María, Pedro y Rocío.

Hay una persona que ha vivido muy de cerca este proceso, María Jesús, ha visto cada paso, ha sufrido los quebraderos de cabeza que suponían muchas de las decisiones, me ha dado el mejor ánimo, el apoyo más tenaz, y los mejores consejos que se pueden esperar. No sé cuánto de esto es tuyo, pero creo que no es poco.

Llegado este punto no puedo olvidar a mi grupo de amigos, Paco, Esme, José Francisco, Maricarmen, Pepe, Adriana, Alonso, Mari, Pedro, Rubén, Mary, Álvaro, Alberto, Óscar e Inmi. Por suerte, para mí, han aguantado estoicamente todas las horas que he pasado contándoles como iba con la tesis, ya sea porque ellos me hubiesen preguntado o porque yo recurrentemente acabara hablando del tema.

Afortunadamente mi inicio en el camino de la investigación no fue en solitario, esta aventura comenzó con mis amigos, Adrián, Antonio y Miguel, y a día de hoy puedo decir que trabajar con ellos fue toda una suerte. También me gustaría agradecer al resto de miembros del equipo de SCENIC toda la ayuda, y el soporte personal que me han prestado.

Mis amigos de la universidad, Javi, Alejandro, Antonio, Estefanía, José, Curro, Marcos, Javi Espinar, Hugo y Damián. Vuestra ayuda a lo largo de los años ha sido indispensable para llegar hasta aquí.

Durante todo este tiempo he encontrado otras tantas personas que se han preocupado y me han enseñado muchísimo en lo técnico y en lo personal, Toni, Portero, Víctor, Joaquín, Pedro, Pedraza, Rafa, Carlos, Iván, Antonio Jesús, Edu, Castor y Julio. Gracias.

No me puedo despedir sin agradecer a Sergio Gálvez todo el apoyo, los consejos y el ánimo que me ha dado a lo largo de los años.

*"A hombros de gigantes."*

— *I. Newton*
*Carta a Robert Hooke, 1675*

# Contents

# Resumen (in Spanish)

El concepto *Cloud Computing* o Computación en la Nube se ha descrito hasta la fecha en términos de la evolución de la tecnología que lo conforman (por ejemplo, la computación en Grid y Clustering), de los paradigmas en los que se basa (como *Virtualization*, *Client-Server-Model* and *Peer-to-Peer*), sus características (por ejemplo, elasticidad y escalabilidad), o sus ventajas logísticas y económicas (como el uso y pago bajo demanda de los recursos y servicios). Sin embargo, *ubicuidad* podría ser la palabra más ilustrativa para describir este concepto. La Computación en la Nube no es solo uno de los temas más significativos y controvertidos en entornos empresariales y académicos, sino que la influencia del *Cloud* transciende en la sociedad hasta tal punto que esta tecnología se utiliza cada día en el mundo entero mediante miles de aplicaciones, modificando las relaciones entre empresas y clientes, la forma en que las personas se conectan y comparten información, y cómo esta información se produce, gestiona, procesa y consume.

Es innegable que la Computación en la Nube ha supuesto una revolución de las Tecnologías de la Información y un cambio radical para la industria, que en muy poco tiempo ha abandonado las soluciones del tipo cliente-servidor para usar y desarrollar soluciones basadas en el *Cloud*. Como resultado, en los últimos años muchos proveedores como Amazon, Google, Microsoft e IBM han construido plataformas *cloud*. Además, de estos proveedores también han surgido un conjunto de plataformas de código abierto como OpenStack[1], Open Nebula[2] y Cloud Foundry[3]. Aunque se ofrecen muchas descripciones distintas y decenas de proveedores ofrecen sus propios servicios, es cierto que en términos de forma, conceptualización, capacidades y uso, todos los proveedores y las plataformas *cloud* comparten los mismos principios: el *Cloud* promueve el acceso bajo demanda a una cantidad masiva de recursos que pueden ser aprovisionados y liberados rápidamente desde cualquier parte del mundo a través de tres modelos de servicios principales, *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS).

Mientras que SaaS permite a los usuarios finales utilizar aplicaciones que se están ejecutando en el *cloud*, IaaS y PaaS permiten desarrollar, configurar y ejecutar sistemas, y aplicaciones. El nivel de IaaS proporciona acceso a la computación virtualizada, el almacenamiento y los recursos de red. El nivel de PaaS va un paso más allá y simplifica

[1]OpenStak: `https://www.openstack.org/`.
[2]Open Nebula: `https://opennebula.io/`.
[3]Cloud Foundry: `https://www.cloudfoundry.org/`.

el consumo de la infraestructura bajo demanda para apoyar la gestión de aplicaciones. IaaS parece más flexible desde el punto de vista de los desarrolladores, debido a que ofrece un entorno sin restricciones en el que se puede instalar y configurar cualquier tipo de software, mientras que PaaS puede llegar a ser más restrictivo, por ejemplo, los vendedores proveen un *Software Development Kit* (SDK) específico para diferentes lenguajes y tecnologías para desarrollar y ejecutar aplicaciones.

No obstante, la instalación, configuración y mantenimiento de la tecnología que se ejecuta en IaaS puede llegar a ser una tarea tediosa, mientras que PaaS permite el uso de toda la potencia de las plataformas *cloud*, monitorización, elasticidad y escalabilidad, etc., con una configuración mínima y asistida. Además, el mantenimiento de los entornos suelen ser transparentes, ya que la propia plataforma se encarga del mantenerlo actualizado. Esto significa que la elección del *cloud* con la que trabajar va más allá de elegir en qué proveedor se va ejecutar una aplicación, ya que tiene implicaciones en el desarrollo, despliegue y mantenimiento de las aplicaciones y el entorno de ejecución. Por ejemplo, en IaaS los mecanismos como el auto-escalado o el balanceo de carga se tienen que añadir como parte de la arquitectura de las aplicaciones, incluyendo tecnología específica. Por otro lado en PaaS estas funcionalidades son inherentes a la propia plataforma y se ofrecen de manera nativa, así que las aplicaciones pueden utilizarlas de manera casi transparente con configuración mínima. Esto, por supuesto, tiene un impacto directo en cómo se definen y ejecutan las tareas operacionales para desplegar y mantener el entorno y las aplicaciones. Por ejemplo, el despliegue o actualización de una aplicación en IaaS puede requerir operaciones específicas para ocuparse de la orquestación de un *cluster*. Sin embargo, en PaaS la gestión del *cluster* vienen dada por la plataforma. De modo, los clientes pueden utilizar las operaciones que este nivel de abstracción ofrece, por ejemplo, para actualizar las aplicaciones, simplificando los procesos y minimizando las posibilidades de error.

En esta tesis se estudia el impacto que la Computación en la Nube tiene en el ciclo de vida de las aplicaciones que va desde cómo se desarrollan hasta cómo se ejecutan y mantienen en el *cloud* utilizando los diferentes servicios.

## Motivación

Debido a la tendencia y las expectativas de la Computación en la Nube, la tecnología ha evolucionado muy rápido durante los últimos años. Varios proveedores han implementado sus propias soluciones, construyendo capas de servicios personalizados para exponer sus recursos. Algunos de ellos se han centrado en proporcionar y optimizar diferentes tipos de servicios, como Heroku[1] y Open Shift[2], que se centran en el nivel PaaS, ofreciendo diferentes mecanismos de interacción, como bibliotecas, APIs REST y clientes de consola. Otras plataformas han sido fusionadas o incluidas en soluciones más grandes, como SoftLayer[3] e IBM Cloud[4]. Además, muchos de estos proveedores proporcionan de forma

[1]Heroku: `https://www.heroku.com/`.
[2]Red Hat OpenShift: `https://www.openshift.com/`.
[3]Softlayer: `http://www.softlayer.com/`.
[4]IBM Cloud: `https://www.ibm.com/cloud`.

nativa soluciones integradas en sus plataformas, denominadas *add-ons*, que permiten a las aplicaciones aprovechar servicios críticos, como la mensajería, el almacenamiento o la monitorización, con una configuración mínima, y delegando su mantenimiento en la plataforma.

Como resultado, la mayoría de estos proveedores ofrecen un conjunto de servicios similares, en cuanto a funcionalidades y niveles de abstracción. Sin embargo, cada una de estas soluciones siguen sus propias especificaciones. Por ejemplo, cada proveedor proporciona sus propias APIs, especifica su propio *Service Level Agreement* (SLA), ofrece una *Quality of Service* (QoS) diferente, soporta tecnologías concretas o proporciona servicios *ad-hoc*.

La heterogeneidad y la proliferación de estas soluciones ha aumentado el número de cuestiones que deben abordarse en el contexto de la Computación en la Nube. Los desarrolladores diseñan sus aplicaciones para que se ejecuten en proveedores específicos, pero si suceden cambios en los requisitos de las aplicaciones o en el proveedor *cloud*, los desarrolladores pueden ver sus aplicaciones bloqueadas en las plataformas para las que sus aplicaciones fueron diseñadas, ya que trasladarse a otras plataformas podría no ser factible debido a la complejidad y el coste que podría llevar adaptar sus aplicaciones a las condiciones de un nuevo proveedor. De hecho, los desarrolladores pueden verse bloqueados en un nivel de abstracción concreto de un proveedor, IaaS o PaaS. Esta problemática se denomina *vendor lock-in* o bloqueo del proveedor, y tiene un impacto directo en la portabilidad e interoperabilidad, en la definición de las aplicaciones y en el uso de los servicios.

En esta tesis se analiza los problemas relacionados con el *vendor lock-in* y se ofrece una herramienta para mitigar su impacto en el ciclo de vida de las aplicaciones. Se estudia la descripción portable de las aplicaciones para que se puedan desplegar en proveedores diferentes minimizando el coste de adaptación. Además se plantea un mecanismo de migración que permite a los desarrolladores reaccionar ante cambios en los proveedores *cloud* que permite mover las partes afectadas de las aplicaciones a otros proveedores. Todo esto teniendo en cuenta los posibles errores que pueden suceder en cada fase, como por ejemplo, un error durante el despliegue. A continuación se muestran las cuestiones que han guiado el trabajo que se desarrolla en esta tesis.

– *¿Se pueden desarrollar aplicaciones en la nube independientemente de los proveedores utilizados para ejecutarlas? En caso afirmativo, ¿se pueden desplegar estas aplicaciones en servicios de diferentes proveedores?*

– *¿Se puede gestionar el despliegue de las aplicaciones en función de sus componentes, de modo que cada uno de ellos se despliegue usando servicios de proveedores diferentes? ¿Pueden operar de manera homogénea los componentes desplegados en diferentes tipos de servicios (IaaS y PaaS)?*

– *Si las aplicaciones pueden desplegarse en servicios de diferentes tipos, usando diferentes proveedores, ¿podemos mover estos componentes en tiempo de ejecución? ¿Estas aplicaciones podrían estar operativas mientras se migran? Y si es así, ¿podría minimizarse su tiempo de inactividad?*

– *Si un componente falla, ¿podemos recuperar una aplicación a su estado normal? ¿Y si este fallo ocurre en el momento del despliegue?*

– *¿Cómo se ve afectado el ciclo de vida de una aplicación si algo cambia?*

– *¿Qué pasa si cambian los requisitos de una aplicación del proveedor en el que se está ejecutando o el tipo de servicios utilizados?*

– *¿Qué pasa suceden errores en una aplicación que se está ejecutando?*

A continuación, analizamos los problemas del *vendor lock-in* y cómo afectan al ciclo de vida de las aplicaciones, y resumimos los desafíos que se abordan en esta tesis.

### Portabilidad e interoperabilidad

Como ya se ha mencionado, la heterogeneidad de los proveedores es una de las causas principales de la dependencia con las plataformas *cloud*, y las diferencias entre IaaS y PaaS no hacen sino agravar el problema. Es cierto que ambos tipos de servicios permiten a los desarrolladores desplegar sus aplicaciones y almacenar y gestionar datos, pero ofrecen diferentes mecanismos para la configuración y el consumo de los servicios. En IaaS, se ofrece a los desarrolladores un acceso casi total a la infraestructura, control sobre el aprovisionamiento de la máquina virtual, gestión de las redes y del sistema operativo, etc. En PaaS, los usuarios pierden toda capacidad de configuración de bajo nivel a cambio de obtener entornos preconfigurados en los que pueden desplegar sus aplicaciones y aprovechar bajo demanda características útiles como la elasticidad y la escalabilidad. Así que cada nivel de abstracción proporciona servicios para alcanzar objetivos similares, pero se gestionan mediante mecanismos e interfaces diferentes.

Con el fin de mitigar esta heterogeneidad y encontrar una solución agnóstica para la gestión de los diferentes proveedores *cloud*, han surgido herramientas y *frameworks* independientes que integran bajo una sola interfaz los servicios de múltiples proveedores, tanto públicos como privados proporcionando entornos de despliegue descentralizados siguiendo diferentes enfoques. Por ejemplo, algunas soluciones, como jClouds[1] o Nucleus[2] proporcionan interfaces que cubren las API de varios proveedores. Otras soluciones ocultan la interacción directa con los proveedores finales. Un ejemplo son los *cloud brokers* que integran servicios *cloud* diversos en una capa de interacción unificada, y sirven a los clientes de una compatibilidad semántica entre plataformas, permitiendo el uso de servicios de diferentes proveedores mientras que se oculta la heterogeneidad y la complejidad de su uso.

En muy poco tiempo, estas plataformas han evolucionado de formas diferentes, permitiéndole a los usuarios aprovechar los servicios *cloud* de multitud de proveedores para desplegar y hacer funcionar sus sistemas. Términos como *multi-cloud*, *cross-cloud*, *federated clouds*o *inter-clouds*se han utilizado para clasificar las distintas formas en las que se integran los servicios de las plataformas y cómo se distribuyen las aplicaciones,

[1]Apache jClouds: `https://jclouds.apache.org/`.
[2]Nucleus: `https://github.com/stefan-kolb/nucleus`.

ofreciendo una solución a muchos de los problemas relacionados con la portabilidad e interoperabilidad entre proveedores. Las principales diferencias entre estos enfoques radican en cómo se integran los servicios de terceros y en cómo se gestionan los módulos desplegados en diferentes plataformas. Sin embargo, solo permiten operar simultáneamente con un único nivel de abstracción para desplegar las aplicaciones, es decir, todos los componentes de una aplicación se despliegan en servicios IaaS o todos en servicios PaaS.

Qué servicio elegir entre la multitud de servicios que existen en el *cloud* sigue siendo un desafío para los usuarios. Además, una vez que se selecciona un servicio, son necesarios mecanismos para asegurar que el proveedor elegido ofrece los recursos de acuerdo a la especificación de requisitos definidas en el SLA y el QoS. La decisión no es, en efecto, trivial, y el contexto y conocimiento pueden cambiar con el paso del tiempo, por lo que puede tener un impacto inesperado. Por ejemplo, hoy podemos decidir que se va a utilizar un proveedor de PaaS concreto para un módulo de una aplicación porque es más rentable, o porque requiere menos esfuerzo de gestión, pero mañana nuestras necesidades o modelo de negocio puede requerir más control sobre nuestras máquinas virtuales (VM), por ejemplo, para mejorar la integración con la infraestructura de la empresa, o porque necesitamos aumentar el nivel de seguridad de los servicios. Esto es problemático, ya que este tipo de cambios requieren un esfuerzo de desarrollo. Cambiar de un proveedor de PaaS a otro puede requerir un esfuerzo significativo, y adaptar una aplicación y los procesos operacionales para que funcione en diferentes niveles, de IaaS a PaaS o viceversa, puede ser simplemente prohibitivo. Desafortunadamente, este tipo de problemas puede ser inevitable con el tiempo, debido a los cambios en los servicios ofrecidos, los precios, las políticas de seguridad, o simplemente porque un proveedor deja de prestar sus servicios[1].

Además, la mayoría de las soluciones actuales, denominadas orquestadores, usan modelos de topologías de las aplicaciones que describen los componentes de de las aplicaciones, cómo estos se relacionan entre ellos y los recursos utilizados. De esta manera, ofrecen un entorno portátil e interoperable en el que los desarrolladores pueden describir sus sistemas y seleccionar los recursos que mejor se adapten a sus necesidades, sin preocuparse por los detalles técnicos relacionados con el uso de los servicios. Sin embargo, muchas de estas soluciones proporcionan su propia especificación para la topología que, además, suelen estar orientadas por la definición de sus propias APIs, por lo que normalmente estas descripciones de topología no suelen ser compatibles entre ellas. Desafortunadamente, podemos ver esto como un problema recurrente. El uso de estas especificaciones no hace más que trasladar el problema del *vendor lock-in* de los proveedores finales a las capas intermedias de orquestación que se encargan de integrar y consumir los servicios de los diferentes proveedores. Por ejemplo, supongamos que una aplicación está modelada para ejecutarse en un entorno compuesto por más de un pro-

[1] Varios servicios, incluso plataformas completas, como DotCloud o CloudBees, han sido clausurados por sus proveedores. DotCloud, el servicio *cloud* que dio origen a Docker, se cerró en febrero de 2016 (`https://www.datacenterknowledge.com/archives/2016/01/26/dotcloud-the-paas-cloud-provider-that-birthed-docker-sets-closing-date`).

veedor, lo que se denomina *multi-cloud*, utilizando un orquestador *cloud* concreto, como RoboconfAhora, supongamos que debido a cambios en los requisitos de la aplicación (tecnología, QoS, etc.), se necesita llegar a nuevos servicios en la nube que no están soportados por el orquestador actual. Lamentablemente, el uso de nuevos proveedores *cloud* pueden requerir adaptaciones que no sean sencillas de llevar a cabo. Por ejemplo, las descripciones de las aplicaciones —la topología— tendrían que ser modificadas para adaptar los componentes al uso de los nuevos servicios. Además de las descripciones de las aplicaciones, algunas soluciones también podrían requerir desarrollo de nuevos scripts o extensión de las APIs para que las tareas de despliegue y mantenimiento soporten el nuevo proveedor, lo que sería necesario para que las aplicaciones sean gestionadas por el ciclo de vida del orquestador.

Aquí entran en juego los estándares que tratan de armonizar el *Cloud* proporcionando especificaciones concretas para normalizar las topologías de aplicación y definiciones de los recursos *cloud* utilizados. Además, los estándares también definen protocolos de uso que detallan mecanismos de conexión e interacción con las plataformas. Han aparecido estándares que especifican mecanismos para gestionar entornos *multi-cloud*, lo que permite a las aplicaciones aprovechar más de un proveedor, por ejemplo los estándares TOSCA[1] y OCCI[2]. Como resultado, algunas de las soluciones de integración de proveedores mencionadas anteriormente han basado en estándares las especificaciones de sus topologías, lo que permite que las aplicaciones sean compatibles entre diferentes orquestadores, minimizando el esfuerzo de adaptación necesario. Sin embargo, como ya se ha dicho, muchas de estas soluciones sólo pueden gestionar un único nivel de abstracción.

Debido a todo esto, en este trabajo se argumenta que los mecanismos para describir aplicaciones portátiles e interactuar con diferentes niveles de abstracción, IaaS y PaaS, de forma homogénea es un reto de investigación que permitiría a los desarrolladores ejecutar aplicaciones utilizando los recursos *cloud* que mejor se adapten a sus necesidades. Además, las soluciones orientadas a estándares pueden ayudar a mitigar los problemas de portabilidad y facilitar el análisis de las topologías de las aplicaciones para automatizar los despliegues en el *cloud*.

### Migración en tiempo de ejecución

Lamentablemente, el problema del *vendor lock-in* también afecta a otros aspectos del mantenimiento de las aplicaciones. Incluso si existiese una solución perfecta para el despliegue automático de aplicaciones, en la que cada componente se desplegara utilizando el mejor servicio posible para que su calidad de servicio fuera la óptima, de acuerdo a unos criterios, todavía sería necesario tratar uno de los problemas de bloqueo de proveedores más importantes para la ejecución de aplicaciones: *los cambios*. Como ya se ha mencionado, pueden producirse distintos tipos de cambios que afectan a las aplicaciones *cloud*. Por ejemplo, actualizaciones de las aplicaciones desplegadas, alteraciones impredecibles en las cargas de trabajo, o en los servicios utilizados, como por ejemplo,

[1]TOSCA: `https://www.oasis-open.org/committees/tosca/`.
[2]OCCI: `https://occi-wg.org/`.

cambios en los servicios ofrecidos, precios, políticas de seguridad o incluso proveedores que dejan de ofrecer servicio. Como resultado, habría que mover las aplicaciones, o parte de ellas, para utilizar otros servicios que pueden estar en otros proveedores o incluso en un nivel de abstracción distinto. De hecho, según la tasa de cambio del *cloud* y las tecnologías utilizadas, la migración de componentes individuales o aplicaciones completas es inevitable a lo largo del tiempo.

Teniendo en cuenta las cuestiones relacionadas con el bloqueo de proveedores, *¿Es posible migrar aplicaciones o algunos de sus componentes si es necesario? ¿Cuál es el impacto? ¿Puede realizarse esa migración de manera que se garantice la fiabilidad de las aplicaciones y al mismo tiempo se reduzcan al mínimo el inevitable tiempo de inactividad?*

Cambiar los proveedores sobre los que desplegar una aplicación ya se ha estudiado anteriormente. Sin embargo, desde el punto de vista de la migración, estas soluciones se limitan en su mayor parte a mover aplicaciones completas de un proveedor a otro. En algunas de estas propuestas, como por ejemplo, en las que se soporta el *cross-cloud*, sí soportan el redespliegue orientado a componentes.

No obstante, lo que proponen estas soluciones no puede llamarse migración en tiempo de ejecución. Por ejemplo, con el fin de optimizar el coste de una aplicación en ejecución, puede que queramos mover algunos de sus componentes a diferentes proveedores *cloud*. Aplicando un proceso de migración en tiempo de ejecución, estos componentes deberían moverse mientras la aplicación sigue funcionando, minimizando el impacto en el sistema en ejecución y deteniendo solo las partes necesarias de la aplicación mientras se mantiene su rendimiento tanto como sea posible. Sin embargo, en las soluciones comentadas anteriormente la aplicación tendría que detenerse completamente y volver a desplegarse, lo que tiene un impacto significativo en su rendimiento. Además, la mayoría de las soluciones existentes solo se ocupan de un único modelo de servicio, típicamente IaaS.

La migración en tiempo de ejecución es un tema que ha sido y está siendo estudiado tanto por la academia como por la industria, pero que aún no se ha resuelto. De hecho, hay varias cuestiones claves relacionadas con este tema que se deben tener en cuenta. Por ejemplo, además de la heterogeneidad en el *cloud*, la gestión en tiempo de ejecución de los componentes requiere un conocimiento exhaustivo de la topología de la aplicación. Para migrar los componentes es necesario orquestar todo el contexto de ejecución, como los servicios y los recursos vinculados, para realizar el movimiento esperado de los componentes, además de tener en cuenta los posibles problemas de interoperabilidad y portabilidad. Además, los componentes no pueden operarse de manera aislada porque el rendimiento de otras partes de la aplicación puede verse afectado. Solo una descripción completa de la topología de la aplicación permite analizar sus componentes y su estructura como parte del proceso de migración para poder determinar las operaciones necesarias para la migración. Ya se han propuesto algunas soluciones para la *migración en vivo* de los componentes de aplicaciones en ejecución que usan su topología y se ocupan del movimiento de los componentes entre diferentes proveedores. Sin embargo, estas soluciones se limitan al nivel IaaS, y se basan en descripciones de aplicaciones que no son compatibles entre proveedores ni orquestadores.

El desarrollo de una solución para el problema de la migración en tiempo de ejecución por componentes parece una cuestión compleja que se agrava aún más si se consideran los movimientos entre servicios de niveles de abstracción diferentes entre proveedores. Sin embargo, esto permitiría a los desarrolladores reaccionar ante los cambios en los requisitos de las aplicaciones, cambios inesperados en los proveedores y servicios utilizados. Además, el análisis de la topología permitiría optimizar el proceso de migración y minimizar el impacto de los movimientos en el rendimiento de los otros componentes que continúan en ejecución.

### Gestión robusta del ciclo de vida

Ya se ha mencionado cómo el *vendor lock-in* afecta a la elección de los recursos *cloud* para una aplicación, el despliegue e incluso a la migración para reaccionar ante cambios. Sin embargo, el impacto del *vendor lock-in* no se termina aquí. Los errores en el *cloud* pueden ocurrir durante todas las fases del ciclo de vida de una aplicación. Por ejemplo, algo podría salir mal al aprovisionar recursos durante el despliegue de la aplicación, como por ejemplo, fallos en la instanciación de las máquinas virtuales.

Los errores no solo ocurren durante el despliegue. Una vez que las aplicaciones se están ejecutando, pueden ocurrir problemas tanto en las aplicaciones como en los servicios *cloud* utilizados, como la sobrecarga de recursos o problemas de conectividad. Además, los errores inesperados pueden desde parar una parte del sistema hasta detener la aplicación por completo. Por lo tanto, los mecanismos de *self-healing* o auto-reparación deben incluirse en las soluciones que gestionan los ciclos de vida de las aplicaciones en entornos *multi-cloud* para detectar y gestionar errores.No obstante, para desarrollar estos mecanismos, también es necesario abordar la heterogeneidad del *cloud* en lo que respecta a la observabilidad para conocer el estado de los componentes y detectar los errores, así como la ejecución de las operaciones necesarias para restaurar las partes de las aplicaciones afectadas.

Desafortunadamente, el soporte actual de la gestión de fallos no está totalmente automatizado. Por ejemplo, cuando un componente de una aplicación deja de dar servicio, en una máquina virtual, el administrador de la aplicación normalmente tiene que analizar manualmente y bajo demanda el sistema para encontrar la causa principal del incidente y resolverlo. Esto puede requerir el reinicio los servicios afectados en la máquina, o incluso el reinicio de la propia máquina. Puede darse el caso de que ocurra algún error de infraestructura que no pueda ser recuperado, lo que requerirá la reconstrucción de la máquina virtual.

En la mayoría de los casos, estas soluciones consisten en desarrollos hechos a medida para arquitecturas de aplicaciones específicas que deben actualizarse constantemente, de acuerdo a los cambios en las aplicaciones y el entorno de ejecución, utilizando herramientas de gestión, como por ejemplo, Chef[1] o Puppet[2]. La automatización de estas tareas

[1]Chef es una herramienta de automatización para definir la infraestructura como código (`https://www.chef.io`).

[2]Puppet es una herramienta de gestión de configuración y despliegue de software de código abierto (`https://puppet.com/`).

conllevaría una mejora significativa, por ejemplo en lo referente a las pruebas, y además evitarían operaciones manuales de configuración y mantenimiento de sistemas, que son propensas a errores. Varios sistemas ofrecen soluciones automatizadas para gestionar incidentes básicos comunes, aunque requieren de la intervención de expertos para resolver cuestiones más complicadas.

De hecho, el *self-healing* sigue siendo un desafío que está siendo investigado tanto por la academia como por la industria. Actualmente, algunas plataformas proporcionan cierto apoyo a algunos mecanismos básicos de *self-healing*. Por ejemplo, AWS[1], Google[2] o Azure[3] ofrecen soluciones basadas en agentes que permiten a un servicio de análisis monitorizar el estado los servicios de las aplicaciones y definir políticas para reemplazar las instancias con errores si es necesario. Sin embargo, la capacidad de estas acciones son limitadas: después de algún tiempo sin comunicación, los agentes marcan las instancias con un estado desconocido, y se inicia un procedimiento de recuperación. Dependiendo del tipo de servicios, se pueden aplicar diferentes planes de recuperación del error, pero normalmente las operaciones soportadas son únicamente parar, reiniciar o recrear. Los mecanismos de *self-healing* pueden aplicarse simplemente a una lista de tipos de recursos reducidos, incluso puede que solo sean aplicables si se han desplegado y configurado siguiendo algunas restricciones. Por ejemplo, las máquinas virtuales decoradas con funciones de *self-healing* en AWS deben ser operadas a través de la consola de AWS OpsWorks Stacks[4]. Además, como ya se ha mencionado, solo pueden realizarse operaciones orientadas a la infraestructura, como el reinicio o la recreación de la máquina virtual, por defecto no permite ejecutar operaciones específicas para gestionar los errores concretos en las aplicaciones.

Como se ha visto en las secciones anteriores, la mayoría de los orquestadores que intentan mitigar los problemas relacionados con el *vendor lock-in* utilizan sus propios modelos para especificar la topología de las aplicaciones, y utilizan esta información para orquestar el despliegue y el mantenimiento en entornos *muti-cloud*. En cuanto a la detección de errores muchos de ellos incluyen mecanismos para comprobar el estado de la aplicación. Otros van un paso más allá y ofrecen algún tipo de análisis del comportamiento de la aplicación y sugieren algunos cambios, pero no proporcionan ninguna capacidad de *self-healing*.

La extensibilidad de muchos orquestadores podría soportar el desarrollo de las operaciones de *self-healing*, como Brooklyn[5], Terraform[6] y Roboconf. No obstante, como en escenarios anteriores, sería necesario emplear recursos para desarrollar estos mecanismos. Además, en muchos casos podrían no ser soluciones agnósticas y requerirían

[1]La información sobre las capacidades de *self-healing* de los servicios web de Amazon se puede encontrar en `https://docs.aws.amazon.com/opsworks/latest/userguide/workinginstances-autohealing.html`.

[2]La información sobre las capacidades de *self-healing* de los servicios web de Google se puede encontrar en `https://cloud.google.com/compute/docs/instance-groups/autohealing-instances-in-migs`.

[3]La información sobre las capacidades de *self-healing* de los servicios web de Azure se puede encontrar en `https://azure.microsoft.com/en-us/blog/service-healing-auto-recovery-of-virtual-machines`.

[4]AWS OpsWorks Stacks: `https://aws.amazon.com/opsworks/`.

[5]Apache Brooklyn: `https://brooklyn.apache.org/`.

[6]Terraform: `https://terraform.io/`.

modificaciones de la topología, impactando en la portabilidad de las aplicaciones.

Otras soluciones se centran en la automatización basada en eventos, como por ejemplo, StackStorm[1] o RunDeck[2], ofreciendo un amplio conjunto de sensores, que permiten definir disparadores de eventos sofisticados. Estas soluciones soportan una cantidad muy variada de herramientas y tecnologías de ejecución, incluyendo la ejecución de servicios REST, o incluso pueden conectarse a una máquina y ejecutar comandos, por ejemplo, usando SSH. Sin embargo, como otras soluciones mencionadas anteriormente, el mantenimiento de estas tareas no está automatizado, y por lo tanto se necesita un esfuerzo de adaptación y mantenimiento si algo cambia en la aplicación o en el entorno.

## Desafíos de la investigación

De acuerdo a las cuestiones descritas hasta ahora, el reto principal de esta tesis es ofrecer una gestión homogénea de los servicios de IaaS y PaaS, y promover una metodología para describir las aplicaciones y los recursos *cloud* de los proveedores, proporcionando a los desarrolladores mecanismos para mejorar la portabilidad e interoperabilidad de sus aplicaciones. Además, estos mecanismos son las bases sobre las que se construyen las soluciones para la migración y *self-healing*. Una descripción exhaustiva de la topología permite analizar la estructura de la aplicación a fin de orquestar un proceso de migración en tiempo de ejecución para las aplicaciones en funcionamiento, mientras que una API común hace transparente la gestión de los servicios *cloud*. Esto permite la optimización orientada a componentes durante todo el ciclo de vida de la aplicación, incluyendo tanto el despliegue como su ejecución. Los usuarios pueden elegir los recursos *cloud* cuyas características se adapten mejor a los requisitos de sus aplicaciones independientemente de su nivel de abstracción, IaaS o PaaS. Por las mismas razones, una gestión robusta de las aplicaciones es también factible. La API común permitirá conocer el estado de las aplicaciones en ejecución mediante la unificación de los mecanismos de supervisión, de manera que se puedan detectar los errores cuando se produzcan. Así pues, las técnicas de control de fallos y recuperación de errores pueden integrarse directamente en el proceso de orquestación de aplicaciones, de modo que su gestión se base completamente en el conocimiento inferido de la topología, evitando desarrollos *ad-hoc* para reaccionar ante los errores.

Por lo tanto, con el fin de minimizar los efectos del *vendor lock-in*, este trabajo pretende que los desarrolladores abstraigan sus aplicaciones de la complejidad del *cloud*, proporcionando herramientas agnósticas para construir aplicaciones portátiles que faciliten la reacción ante cambios tanto en el *cloud* como en el de las aplicaciones, además de ofrecer mecanismos para detectar y reparar errores automáticamente. A continuación, se detallan los desafíos principales que se abordan en esta tesis.

**Descripción agnóstica de la topología.** Definir un marco agnóstico de modelado basado en estándares para permitir la descripción completa de las aplicaciones y

[1]StackStorm es una herramienta de automatización basada en eventos (`https://stackstorm.com/`).

[2]Rundeck permite la configuración, monitorización, configuración de trabajos en centros de datos (`https://rundeck.com/`).

los servicios y recursos *cloud* utilizados (IaaS y PaaS). Concretamente, se propone usar los estándares actuales, CAMP y TOSCA para permitir la portabilidad.

**Heterogeneidad semántica del *cloud*.** Desarrollar una API común que unifique los servicios *cloud* independientemente de su nivel de abstracción, para IaaS y PaaS.

**Gestión del ciclo de vida de las aplicaciones.** Integración del modelado y la API unificada para construir un marco que permita modelar y desplegar aplicaciones portátiles de manera estandarizada. Proporcionar una orquestación completa de las aplicaciones durante los despliegues utilizando tanto servicios IaaS como PaaS.

**Portabilidad.** Minimizar el impacto en la adaptación de las aplicaciones a nuevos proveedores tanto en la descripción de las aplicaciones como en el uso de los recursos *cloud*.

**Migración en vivo.** Proporcionar mecanismos para realizar operaciones de reconfiguración en tiempo de ejecución, minimizando el impacto en los sistemas en ejecución.

**Gestión robusta de las aplicaciones.** Permitir una gestión robusta del ciclo de vida de las aplicaciones, proporcionando mecanismos de detección de fallos y de recuperación.

**Despliegue de un prototipo funcional.** Desarrollar un prototipo funcional de experimentación para alcanzar los objetivos anteriores y analizar el rendimiento de la interacción con la nube.

## Publicaciones principales

Esta tesis viene avalada por una serie de trabajos publicados en lo que se estudian las cuestiones anteriores. A continuación se listan dichos trabajos en los que se analizan cómo se relaciona el ciclo de vida de las aplicaciones con los recursos *cloud*, la resistencia a los cambios de las aplicaciones *cloud* y cómo este se puede abordar minimizando el impacto durante los despliegues o en aplicaciones que ya se están ejecutando.

– Jose Carrasco, Javier Cubo, Francisco Durán, and Ernesto Pimentel. "Bidimensional cross-cloud management with TOSCA and Brooklyn". In 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pages 951–955. IEEE, San Francisco, California, EE.UU, 2016.
DOI: 10.1109/CLOUD.2016.0143.

– Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Trans-cloud: CAMP/TOSCA-based bidimensional cross-cloud". Computer Standards & Interfaces, 58:167–179, 2018. DOI: 10.1016/j.csi.2018.01.005.

– Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Component migration in a trans-cloud environment". In 7th International Conference on Cloud Computing and Services Science (CLOSER), *Revised Selected Paper*, pages 286–307. Springer, Oporto, Portugal, 2017. DOI: 10.1007/978-3-319-94959-8_15.

– Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Runtime migration of applications in a trans-cloud environment". In Adaptive Services-Oriented and Cloud Applications (ASOCA) - Workshops of 15th International Conference on Service-Oriented Computing (ICSOC), pages 55–66. Springer, Málaga, España, 2017. DOI: 10.1007/978-3-319-91764-1_5.

– Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Live migration of trans-cloud applications". Computer Standards & Interfaces, 69:103392, 2020. DOI: 10.10-16/j.csi.2019.103392.

– Antonio Brogi, Jose Carrasco, Francisco Durán, Ernesto Pimentel, and Jacopo Soldani. "Robust management of trans-cloud applications". In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pages 219–223. IEEE, Milán, Italia, 2019. DOI: 10.1109/CLOUD.2019.00046.

## Conclusiones y contribuciones

De acuerdo a las cuestiones mencionadas anteriormente, en esta tesis se estudia cómo el *vendor lock-in* afecta al mantenimiento de las aplicaciones durante su ciclo de vida. Con el fin de mitigar estos problemas, proponemos *trans-cloud* como una abstracción de la gestión del *cloud* que, mediante una API unificada, ofrece a los desarrolladores una única forma de operar de forma simultánea servicios *cloud* en IaaS y en PaaS de proveedores diferentes, extendiendo así el concepto de *cross-cloud*. Esto permite construir aplicaciones y procesos operacionales agnósticos que no dependen de plataformas *cloud* concretas, lo que ofrece a los desarrolladores mecanismos que permiten reaccionar ante cambios tanto en el *cloud* como en las aplicaciones. Además, la propuesta de *trans-cloud* se utiliza como la base de soluciones que se proponen para migrar las aplicaciones en tiempo de ejecución y para garantizar la solidez de la orquestación de las aplicaciones durante su gestión.

A continuación se resumen las contribuciones principales de esta tesis:

1. Una descripción del entorno *trans-cloud*, que proporciona la base de una nueva abstracción de los recursos *cloud*, incluyendo niveles de servicio de IaaS y PaaS. Además se define un entorno para la gestión de las aplicaciones y los recursos asociados basado en el modelado de su topología.

2. Una API de unificación de IaaS y PaaS basada en CAMP. Se ofrece a los usuarios un uso agnóstico y sencillo de diferentes servicios *cloud* permitiéndoles centrarse en sus funcionalidades, mientras que la complejidad de usar e integrar sus interfaces se abstrae mediante la API unificada.

3. Descripción de aplicaciones y servicios *cloud* portables basados en el estándar TOSCA que permite que los usuarios elaboren descripciones completas de sus aplicaciones, incluido todo el conocimiento sobre las capacidades, requisitos, tipos de servicios para ejecutar las aplicaciones, etc., independientemente de los proveedores concretos sobre los que finalmente se desplegará la aplicación.

4. Un *framework* de *trans-cloud* basado en el orquestador de aplicaciones de Apache Brooklyn. El *framework* proporciona un entorno que permite construir y desplegar aplicaciones portátiles utilizando la API unificada de forma estandarizada, proporcionando una gestión completa del ciclo de vida de la aplicación. La principal contribución de esta parte es la portabilidad, ya que los servicios soportados por la API unificada estarán disponibles para desplegar las aplicaciones modeladas sin requerir ningún conocimiento sobre las interfaces concretas de los proveedores.

5. Una herramienta para migrar componentes de aplicaciones en ejecución que orquesta el ciclo de vida de los componentes y los recursos *cloud* utilizando la información de la topología.

6. Una extensión del *framework* de *trans-cloud* para incluir un orquestador de migración en tiempo de ejecución para automatizar un proceso de migración fiable, eficiente y orientado a los componentes de las aplicaciones. Las migraciones pueden iniciarse con solo indicar los nuevos servicios de destino de los componentes, ya sean IaaS o PaaS, sin necesidad de ninguna modificación o interacción con la topología, lo que tiene por objeto desligar las aplicaciones de los proveedores en los que se ejecutan.

7. Mecanismos agnósticos de monitorización, como parte de la API basada en CAMP, permiten comprobar el estado de la aplicación independientemente de los recursos *cloud* sobre los que se ejecutan.

8. Una metodología que soporta la gestión automatizada de fallos en el entorno *trans-cloud*, mediante técnicas de detección de fallos y un procedimiento de recuperación de errores. De esta manera, proporcionamos a los entornos *trans-cloud* la capacidad de construir una orquestación robusta de aplicaciones y la interacción con los proveedores *cloud*.

9. Una extensión del *framework trans-cloud* que integra un protocolo de gestión de errores. Esto contribuye a la capacidad del usuario de automatizar las operaciones para hacer frente de manera agnóstica a los fallos en el *cloud*. Además, ofrece una visión aislada de las tareas operacionales, desacoplándolas de los recursos *cloud* utilizados, contribuyendo a la automatización del proceso de migración de aplicaciones.

10. Se proporcionan casos de estudios complejos que se han utilizado para probar y evaluar las soluciones desarrolladas y su aplicabilidad a diferentes aplicaciones y entornos *cloud*.

# Chapter 1. Introduction

Several definitions have been written of *Cloud Computing* in terms of evolution of technologies (as Grid and Clustering), based paradigms (as Virtualization, Client-Server-Model, and Peer-to-Peer), its capabilities (as elasticity and scalability), or its economic and logistic advantages. However, *ubiquity* may be the most illustrative word to describe the concept. Cloud Computing is nowadays not only one of the most popular topics in the enterprise and academia environments, but the term *Cloud* has transcended to such an extent in society that people uses it every day by means of hundreds of applications, modifying the relations between enterprises and customers, the way in which people connect and share information, and how this information is produced, managed, processed and consumed. It is undeniable that the Cloud has meant a revolution of IT (Information Technology) and a radical change for the industry, which has moved from client-server to cloud-based solutions in a very short time (cf. [36]). As a result, in recent years many cloud platforms have been built by vendors such as Amazon, Google, and Microsoft, and a set of open-source platforms, such as OpenStack, Open Nebula, and Cloud Foundry, have also emerged.

Although several definitions have been written and dozens of cloud providers offer their own services, in terms of shape, conceptualization, capabilities, and usage, all of them share the same principles: the cloud promotes on-demand access to a massive number of resources that can be rapidly provisioned and released from anywhere around the world throughout three main service models, namely, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [61].

While SaaS offers running applications in the cloud to end-users, IaaS and PaaS enable the capability of developing, configuring and running systems and applications in the cloud. IaaS provides access to virtualized computing, storage, and network resources. PaaS goes one step further and simplifies the consumption of on-demand infrastructure, to support the management of cloud applications. On the one hand, IaaS may seem more flexible from the developers' point of view, since any software can be installed and configured, while for PaaS the application development can be more restrictive, since, for example, vendors provide specific SDKs, for different languages, to build and deliver applications. On the other hand, the configuration on IaaS can be tedious, while PaaS allows to take advantage of powerful cloud capabilities, as monitoring, scalability, and elasticity, with a minimal and assisted configuration, therefore simplifying the maintenance and improving their robustness.

Then, the choice of cloud goes beyond where an application is run, but it has consequently implications on how applications are developed, deployed, and maintained. For example, in IaaS, mechanisms such as auto-scaling or load balancing require to be added on-demand as part of the architecture of applications, by including specific technology, considering these functionalities part of the infrastructure. On the other hand, in PaaS, these functionalities are natively offered and can be used in applications after a little configuration. This of course has a direct impact on how operational tasks are defined and performed. The deployment or updating of an application in IaaS would require specific operations to deal, for example, with cluster orchestration. However, PaaS allows to delegate the cluster management to the platform, so clients can use some available operations to update the applications, thus simplifying the process. Moreover, it allows to simplify the maintenance for running applications. For example, for a simple restart on PaaS users delegate to the platform the management of the resources, to maintain the cluster integrity and the service availability, what has implications on the application readiness and error recovery tasks.

In summary, Cloud Computing has significant effects on the lifecycle of applications, from how applications are built to how they are run and maintained in the cloud using different services. The main research questions that have led our research are the following ones:

***Question 1*** – *Can the development of cloud applications be carried out independently of the cloud providers used to deploy them? If so, can these applications be deployed on services of different providers?*

***Question 2*** – *Can this deployment be managed component-wise, so that each component is deployed in services of different providers? Can components deployed on different service types (IaaS and PaaS) be interoperated in a homogeneous way?*

***Question 3*** – *If applications can be deployed on services of different types, by different providers, can we move these components at runtime? Could these applications be operational while migrated? And if so, could their downtime be minimized?*

***Question 4*** – *If a component fails, can we recover the application to its normal state? What if this failure occurs at deployment time? And if failures occur while recovering from a previous failure?*

***Question 5*** – *How is the application lifecycle affected if something change?*

***Question 6*** – *What if application requirements, target cloud, or the kind of used services change?*

These are the questions we try to deal with in this work. In [25], [31], [30], [29], [32], and [20] we analyze how applications' lifecycle relates to cloud resources, the applications' resilience to cloud changes, and how these changes can be addressed, minimizing the impact on cloud applications during deployments and already running applications.

## 1.1 Motivations and challenges

Due to the aforementioned cloud hype, the landscape has evolved very fast during the last years. Several vendors have implemented their own solutions, building their custom service layers to expose their resources. Some of them focused on providing and optimizing different kinds of services, such as Heroku[1] and Open Shift,[2] which focus on the PaaS level, offering different mechanisms for interaction, such as libraries, REST APIs, and shell clients. Other platforms have been merged or included into bigger solutions, such as SoftLayer[3] and IBM Cloud.[4] Moreover, many of them natively provide solutions that are integrated in their platforms, called add-ons, which allow applications to take advantage of critical services, such a messaging, storage or monitoring, with a minimal configuration, and delegating their maintenance to the platform.

As a result, most of these providers offer a set of similar services, regarding functionalities, on top of similar abstraction levels. However, these solutions follow their own specifications. For example, each one provides its own APIs, specifies its own Service Level Agreement (SLA), offers different Quality of Service (QoS), supports concrete technologies, or provides ad-hoc services. Moreover each provider applies to these agreements a different pricing model and billing customizations, such as described in the Customer Agreement (CA) of each platform [52].

The proliferation of these solutions and their heterogeneity has also increased the number of issues to be addressed in Cloud Computing. Developers design their applications to work with specific providers, and, if after some time, application requirements or cloud capabilities change, they are locked with it because it is not feasible to move to other platforms due to the high complexity and cost to adapt their applications to the conditions of a new vendor. Indeed, developers can see themselves locked in a specific abstraction level of a provider. These issues are known as *vendor lock-in* problems (see, e.g., [3, 67]), which hamper the portability and interoperability in the definition and the usage of services. As a result, developers lose their agility to adapt their applications to the cloud resources that best satisfy their requirements.

Next, we analyze vendor-lock issues and how they impact the applications' lifecycle and summarize the challenges addressed in this work.

### 1.1.1 Portability and interoperability

As discussed in the previous section, the heterogeneity of providers is one of the main causes of vendor lock-in, and the differences between IaaS and PaaS only aggravate the problem. Both kinds of services allow developers to deploy applications and store and manage data, but they offer different mechanisms for the configuration and the consumption of services. In IaaS, developers are offered almost total access to the infrastructure, with control over VM provisioning, management of networks and operating systems, etc.

[1]Heroku: `https://www.heroku.com/`.
[2]Red Hat OpenShift: `https://www.openshift.com/`.
[3]Softlayer: `https://www.softlayer.com/`.
[4]IBM Cloud: `https://www.ibm.com/cloud`.

With PaaS, users lose any capability for low-level configuration in exchange for gaining pre-configured environments where they can deploy their applications and easily take advantage of useful features as on-demand elasticity and scalability. Thus, each abstraction level provides services to reach similar goals, but they have to be managed through distinct mechanisms and interfaces, which users must know and understand.

In order to mitigate this heterogeneity and find a vendor-agnostic solution, independent tools and frameworks have emerged with the goal of integrating, under a single interface, the services of multiple public and private providers (see, e.g., [100], [109], [57], and [95]), or providing decentralized deployment environments (see, e.g., [105] and [86]) following different approaches. For example, some solutions, like jClouds[1] or Nucleus [69], provide interfaces that cover APIs of several providers. Other solutions hide the direct interaction with final clouds. For example, cloud *brokers* integrate diverse cloud services by means of a layer that offers to customers semantic compatibility among vendors, allowing them to deal with heterogeneity and reach services of different providers.

In a very short time, these platforms have evolved to adjust to different ways in which users can take advantage of integrated cloud services to expose and run their systems. Terms such as *multi-cloud* [75], *cross-cloud* [45], *federated clouds* [97], or *inter-clouds* [58] have been used to describe the ability to distribute modules of an application using services from different providers, addressing a significant part of the portability and interoperability issues between providers. The main differences between these approaches lie on the different ways of handling the connections between modules deployed on different platforms. However, in all these attempts, platforms allow operating simultaneously with a single level of service to deploy applications, i.e., all the components of an application are deployed either at the IaaS level or all at the PaaS level (see, e.g., [60], [119], and [45]).

Which service to select from among the multitude of cloud services is still a challenge for users (cf. [3], [86], [70], and [102]). Furthermore, once a service has been selected, we need mechanisms to ensure that the chosen cloud provider is delivering the promised computing resources (see, e.g., [102] and [121]). The decision is indeed non-trivial, and the context and knowledge may change as time passes, and it could have an unexpected impact. For example, we may decide today to use a PaaS provider for a particular module because it is more cost-effective, or because it requires less management effort, but tomorrow our needs or business model may require more control over our virtual machines (VMs), e.g., for a better integration with our enterprise's infrastructure, or because we need to increase the security level of our services. This is problematic, since changes in these decisions require development effort (see, e.g., [98] and [38]). Changing from a PaaS provider to another may already require a significant amount of effort, and adapt an application to run over different levels, from IaaS to PaaS or vice versa, may be simply prohibitive. However, it may be unavoidable over time, because of changes in the offered services, prices, security policies, or simply because a provider just stops

[1]Apache jClouds: `https://jclouds.apache.org/`.

providing its services.[1]

Furthermore, most of the aforementioned solutions support the building of models of the application topologies, including dependencies and used resources, independently of the providers in which services will be executed. Thus, they offer a portable and interoperable environment where developers can describe their systems and select the resources that better fit their requirements, without worrying about technical details of the services use, and focusing on the required features. However, many of these solutions provide their own topology specification bounded by the definition of their own APIs, and normally these topology descriptions are not compatible between different integration solutions. We can see this as a recurrent problem. The usage of these specifications just moves the lock-in problem from final providers to intermediate orchestration layers that are in charge of dealing with cloud consumption. For example, suppose an application is modeled to run in a multi-cloud environment using a concrete cloud orchestrator, such as Roboconf [100]. Then, suppose that, due to changes of the requirements in the application (technological, QoS, etc.), it was needed to reach new cloud services which are not supported by the current orchestrator. Unfortunately, the usage of new cloud providers may require some adaptation that would not be straightforward. For example, applications' descriptions — topology — would have to be operated to adapt the components to use new services. In addition to the applications' descriptions, some solutions would also require some scripting effort to configure the delivery tasks and enable the applications to be managed by the orchestrator's lifecycle (see, e.g., [100]).

Standards try to harmonize cloud context by providing concrete specifications to normalize from application topologies to cloud resource definitions, and provide specific details on the way to perform the interaction between them. Moreover, several standards specify mechanisms to manage multi-cloud environments, allowing applications to take advantage of more than one cloud provider (see, e.g., [91] and [92]). As a result, some of the solutions rely on standards to specify agnostic topologies and allowing applications to be portable between different orchestrators, minimizing the needed adaptation effort. However, as already said, many of these solutions can only manage one abstraction level (see, e.g., [21], [72] and [90]).

Our claim is that mechanisms to describe portable applications and interact with different abstraction levels, IaaS and PaaS, in a homogenous way is a research challenge that would allow developers to run applications using the cloud resources that best fit their requirements. Moreover, standard-compliant solutions can help to mitigate portability issues and facilitate the analysis of application topologies to automatize deployments over the cloud.

[1] Several services (indeed complete platforms), such as DotCloud or CloudBees, have been shut down by their providers. DotCloud, the cloud service that gave birth to Docker, shut down in February 2016 (`https://www.datacenterknowledge.com/archives/2016/01/26/dotcloud-the-paas-cloud-provider-that-birthed-docker-sets-closing-date`).

### 1.1.2 Runtime migration

Unfortunately, the vendor lock-in issue also impacts other aspects of application maintenance. Even if we had a perfect solution for the automatic portable deployment of applications, where each component of our applications is deployed using the best possible service so that its quality of service is the optimal one, according to your chosen criteria, it is still needed to deal with one of the most important vendor lock-in problems for running applications: *change*. As already mentioned, there can be changes of very different nature that affect cloud applications. For example, there can be updates in the deployed applications [84], unpredicted changes in the applications' workloads or contexts, or on the used services, namely, changes in the offered services, prices, security policies, or discontinued providers. As a result, applications, or part of them, should have to be migrated to use different providers and/or abstraction levels. Indeed, according to the change rate of cloud and used technologies the migration of individual components or entire applications is unavoidable over time.

Taking into account vendor-lock-in-related issues, *Can applications or some of their components be migrated in case of need? What about the impact? Can such migration be performed so the reliability of the applications is ensured while minimizing the inevitable downtimes?*

As already mentioned, the possibility of changing decisions about what cloud providers to chose to deploy an application has been widely studied. However, from the migration point of view, these solutions are mostly limited to moving full applications from one provider to a different one. In some of these proposals, when cross-cloud deployment is supported, the re-deployment may be performed component-wise. Indeed, additional problems have to be considered (and solved) when different components of the same application have to be moved to diverse datacenters, or even different cloud providers [55, 12].

Nonetheless, what these solutions propose cannot be called runtime migration. For example, in order to optimize the cost of some running applications, we may want to use a different cloud for some of its components. By applying a runtime migration process, these components should be moved while the application is still running, minimizing the impact on the running system, and stopping only the required application parts while maintaining application readiness as much as possible. However, in the above-mentioned works, the applications must be completely stopped and re-deployed, what has a high impact on the application's performance. Furthermore, most of the existing solutions only deal with one service model, typically IaaS.

As we discuss in Section 2.3.6, on related work, currently only container-based solutions provide some support for runtime application migration. However, these proposals move complete execution environments, and only containerized components. The only currently-available way to handle these situations is using script languages, like Ansible,[1] to manually control the actions to be performed. Of course, this task is error-prone, and requires a significant amount of time and great expertise. It has to be maintained

[1]Ansible: `https://www.ansible.com/`.

together with the application's topology, and it is very dependent on the source and target vendor and the kind of cloud resources used to deploy each of the application's components.

In summary, runtime migration is still an unresolved topic, which has been widely studied by both academia and industry (see, e.g., [62, 120]).

Indeed, there are several key issues related to this topic that must be taken into account. For example, in addition to cloud heterogeneity, the runtime handling of components requires exhaustive knowledge about the application's topology. To migrate components, it is necessary to orchestrate the entire cloud context, such as services and bound resources, to perform the expected movement of the components' services, but taking into account the possible interoperability and portability problems. Moreover, components cannot be operated in an isolated way because the performance of other parts of the application can be affected. Only a complete description of the topology of the application would allow us to analyze the application's components and their relations as part of the migration process to determine the operations required to perform the migration. We can find several proposals for the *live migration of cloud applications' components* in the literature (see, e.g., [42, 43, 14]). These proposals take into account the application's topology and deal with the movement of running application components between different vendors. However, these solutions are limited to the IaaS level, and are based on non-portable application descriptions.

The enabling of component-wise runtime migration seems challenging, and becomes even more complicated if cross-abstraction-level movements are to be considered. However, it would allow developers to react to changes in the applications' requirements or unexpected changes in cloud scenarios. Moreover, the topology analysis would allow us to optimize the process and minimize the impact of component movements on running systems.

### 1.1.3 Lifecycle robust management

We have already discussed on how the vendor lock-in problem affects the choice of cloud resources for an application and to the deployment — and migration — process. However, the impact of the vendor lock-in problem does not stop here yet. Cloud failures can happen during the different phases of an application's lifecycle. For example, something could go wrong when provisioning resources during application's deployment, as e.g., failures in the instantiation of VMs.

Furthermore, failures and other issues do not only happen during deployment. Once applications are running, several problems can happen on both applications and the cloud services they are deployed on (see, e.g., [16, 110]), such as resource overload or connectivity issues. Also, unexpected errors can affect an application, from stopping one part of the system to getting down the application entirely. Therefore, self-healing mechanisms, to detect and manage errors, have to be included on included in solutions that manage applications' lifecycles on multi-cloud environments [101]. However, to develop these mechanisms, it is also needed to deal with cloud heterogeneity regarding observability, to know the components' status and to detect the errors, and the performance of the needed operations to restore the impacted applications' parts.

Moreover, the current support of failure management is not fully automated. For example, when an application component 'dies' in a VM, the application administrator typically has to operate on-demand to find the root cause of the incident and solve it. Such a situation may require restarting the affected workloads on the machine, or even restarting the machine itself. It may even be the case that some infrastructure error happens that cannot be recovered, requiring the re-provisioning of the machine.

In most cases, these solutions are custom developments for specific application architectures, so they have to be constantly updated according to application and environment changes using management tools, as, e.g., Chef[1] or Puppet.[2] The automatization of these tasks would mean a significant improvement, since they involve a considerable amount of effort of configuration and maintenance. Several systems provide automatized solutions to manage common basic incidents, although expert intervention should be required to solve more complicated issues.

In fact, self-healing is still a challenge, being researched by both academia and industry (see, e.g., [101, 76, 59]). Currently, some platforms provide some support for some basic self-healing mechanisms, but they are very basic. For example, AWS,[3] Google[4] or Azure[5] offer an agent-based solution to communicate services with a monitor instance health service and define policies to replacing failed instances if needed. However, these actions are limited: after some time without communication, agents flag an unknown instance status, and a recovery procedure is performed. Depending on the kind of services, different plans can be applied to recover the failure, but normally the supported operations are just stop, restart or recreate. Self-healing mechanisms can just be applied to a reduced list as resource types, and only if they were deployed and configured following some restrictions. For example, virtual machines provisioned with self-healing features in AWS must be operated through the AWS OpsWorks Stacks console.[6] Moreover, as already mentioned, only infrastructure-oriented operations can be performed, such as VM restart or recreate, but specific operations to manage failures in running software cannot be used by default.

As seen in previous sections, different approaches deal with the vendor lock-in problem by delegating the interaction with the cloud to an external orchestrator that concentrates the capability to operate with different providers. Most of them use their own models to specify the applications' topology, and they use this information to orchestrate the deployments and migrations to different providers. Several of them also include

[1]Chef is an automation tool that provides a way to define infrastructure as code (https://www.chef.io).

[2]Puppet is an open source software configuration management and deployment tool (https://puppet.com/).

[3]Information on the self-healing capabilities of Amazon Web Services can be found at https://docs.aws.amazon.com/opsworks/latest/userguide/workinginstances-autohealing.html.

[4]Information on the self-healing capabilities of Google Cloud can be found at https://cloud.google.com/compute/docs/instance-groups/autohealing-instances-in-migs.

[5]Information on the self-healing capabilities of Microsoft Azure can be found at https://azure.microsoft.com/en-us/blog/service-healing-auto-recovery-of-virtual-machines.

[6]AWS OpsWorks Stacks: https://aws.amazon.com/opsworks/.

mechanisms to check the application's status (see, e.g., [63]). Others go one step further and offer some kind of analysis of the application's behavior and suggest some changes (see, e.g., [18]), but they do not provide any self-healing capacity.

The extensible capabilities of other approaches could support the development of recovery tasks, such as Brooklyn,[1] Terraform,[2] and Roboconf [100]. However, as in previous attempts, an important effort would be needed to develop recovery mechanisms. Moreover, in any case, they would not be very suitable in terms of portability and maintenance, since they would be attached to the application's architecture, to its topology, and to the kind of components and targeted cloud resources.

Other solutions focus on event-driven automation, as, e.g., StackStorm[3] or RunDeck,[4] offering a rich set of sensors, which allow to define elaborated triggers. These solutions support an important amount of management tools, including the execution of REST services, or even to step into a machine and run some commands, e.g., using SSH. However, as other solutions discussed above, the maintenance of these tasks is not automatized, and therefore an adaptation effort is needed if something changes in the application.

### 1.1.4 Research challenges

Given the issues described in the previous sections, the main challenge of this work is to offer a homogeneous management of IaaS and PaaS services and enable a methodology to describe applications and their required target cloud resources. This setting provides developers with mechanisms to improve the portability and interoperability of their applications. In addition, in this work, these mechanisms are considered as the base upon which to address the rest of the stated problems. An exhaustive topology description allows the application structure to be analyzed in order to orchestrate a runtime migration process for running applications, whereas a common API makes transparent the cloud management. This allows the component-wise optimization during the entire application's lifecycle, including both the application deployment and execution. Users can choose the cloud resources whose features best adapt to their applications' requirements, using the PaaS or IaaS services that better fit their needs. Because of the same reasons, robust application management is then also feasible. The common API will allow to know the status of the running applications by the unification of monitoring mechanisms, so that failures can be detected when they happen. Thus, failure control and error recovery techniques can be integrated directly into the application orchestration process, so that the management of applications is based on their topology, and therefore no ad-hoc development is needed to react to errors.

In summary, to minimize the effects of the vendor lock-in problems, this work aims at challenging developers to abstract their applications from cloud complexity by providing

[1]Apache Brooklyn: `https://brooklyn.apache.org/`.

[2]Terraform: `https://terraform.io/`.

[3]StackStorm is an event-driven automation tool for auto-remediation (`https://stackstorm.com/`).

[4]Rundeck is runbook automation for incident management, business continuity, and self-service operations (`https://rundeck.com/`).

agnostic tools to build portable applications and facilitate the reaction to changes in both the cloud and the application sides. In the following, we elaborate on the descriptions of the main challenges addressed in this thesis.

***Challenge 1* – Agnostic topology description.** Define a standard-based agnostic modeling framework to enable the full-detailed description of applications and the used cloud (IaaS and PaaS) services and resources. Specifically, we propose using current standards, CAMP and TOSCA to enable portability.

***Challenge 2* – Cloud semantic heterogeneity.** Develop a common API that unifies cloud services independently of their abstraction level, for IaaS and PaaS.

***Challenge 3* – Application lifecycle management.** Integration of the modeling and the unified API to build a framework to allow portable applications to be modeled and deployed in a standardised manner. Providing a complete application orchestration during deployments using both IaaS and PaaS services.

***Challenge 4* – Portability.** Minimizing the impact on the adaptation of applications to new providers in both application description and cloud resources usage.

***Challenge 5* – Runtime migration.** Provide mechanisms to perform such reconfiguration operations at runtime, minimizing the impact on the running system.

***Challenge 6* – Robust management of applications.** Enable a robust management of applications' lifecycle, providing failure detection and recovery mechanisms.

***Challenge 7* – Deployment of a functional prototype.** Develop a functional prototype in which to experiment with the accomplishment of the previous goals and benchmarking the performance with cloud interaction.

## 1.2 Contributions

This thesis aims to investigate each of the aforementioned challenges related to the vendor lock-in issues and how they affect the maintenance of applications during their lifecycle. First, the heterogeneity problems are tackled by defining the *trans-cloud* infrastructure, with which the lock-in problems are minimized by looking at the problems of portability, interoperability, and lack of standardization. Over this framework, the other issues are addressed, and solutions to minimizing the impact of the vendor lock-in problems when migrating applications at runtime and to ensure the robustness of the orchestration of applications during their management are proposed.

In the following, we summarize the main contributions of this work and indicate how the research questions presented in Section  have been addressed:

1. A novel notion of *trans-cloud environment*, providing the basis to consider a new resource abstraction that includes IaaS and PaaS service levels, and a methodology for the management of applications and associated resources based on topology models (*Question 1*).

2. A CAMP-based API for the unification of IaaS and PaaS. It offers users an agnostic and simple usage of different cloud services, allowing them to focus on the functionalities of these services, whilst the complexity of using and integrating their interfaces is hidden behind the unified API (*Question 1*).

3. Description of portable applications and cloud services based on the TOSCA standard to allow users to build full-fledged descriptions of their applications, including all the knowledge about the capabilities, requirements, kinds of services to run applications, etc., regardless of the concrete providers over which the application will be finally deployed (*Questions 1 and 6*).

4. A *trans-cloud framework* is built on the Apache Brooklyn application orchestrator. The framework provides an environment that allows portable applications to be built and deployed using the unified API capabilities in a standardised manner, providing a complete application lifecycle management. The main contribution of this part is to portability, since services supported by the unified API will be available to deploy the modeled applications without requiring any knowledge about the concrete provider interfaces (*Questions 2, 5 and 6*).

5. A tool to migrate components of running applications that orchestrates the lifecycle of applications' components and cloud resources using topology descriptions (*Questions 3, 5 and 6*).

6. An extension of the trans-cloud framework to include a runtime migration orchestrator that automates the reliable, efficient, and component-wise migration of cloud applications. Migrations can be triggered just by indicating the new target resources for components, without requiring any topology modification or interaction, what aims to unlock applications with the providers they run on (*Questions 3, 5 and 6*).

7. Agnostic monitoring mechanisms, as part of CAMP-based API, allows to check the application's status independently of the cloud resources over which they are executed (*Questions 1 and 2*).

8. A methodology to support the automated management of faults in the trans-cloud environment, through failure detection techniques and an error recovery procedure. In this way, we provide trans-cloud environments with the capability to build robust orchestration of applications and interaction with cloud providers (*Questions 1 and 4*).

9. An extension of the trans-cloud framework to integrate a fault-aware management protocol. This contributes to the user's capability to automatize the operations to deal with failures in the cloud in an agnostic way. Moreover, it offers an isolated view of the operational tasks, decoupling them from the used cloud resources and contributing to the automation of the application migration process (*Question 4*).

10. Some non-trivial case studies have been used to test and evaluate the developed solutions and their applicability to different applications and cloud environments (*Questions 1, 2, 3 and 4*).

## 1.3 Outline

The remainder of the thesis is structured in three additional chapters as follows:

**Chapter 2: Published Work.** This chapter contains a list of the contributions related to this work. Moreover, it contains a description of the papers that support this thesis, including a copy of them.

**Chapter 3: Related Work.** This chapter introduces the related work, grouping each topic in a different section. In Section 3.1, the current status of cloud standards is discussed. Section 3.2 presents an analysis of the portability and interoperability issues, including works both from academia and industria that have proposed different solutions to deal with these problems. Section 3.3 introduces different kinds of application migration that can happen in the cloud context. Section 3.4 describes the concept of self-healing, discusses different kinds of solutions, and describes how commercial vendors apply these techniques. Finally, Section 3.5 analyzes the previous problems in the context of the container technologies.

**Chapter 4: Conclusions and Future Work.** This chapter discusses the main contributions and results of this thesis. Conclusions are presented in Section 4.1 and Future Work in Section 4.2.

# Chapter 2. Published Work

During the development of this thesis we have published several research papers, both in journals indexed in the Journal of Citation Report (JCR) and in international workshops and conferences, to present and disseminate our advances and incremental results regarding portability and interoperability, migration, and self-healing of cloud applications.

## 2.1 List of research contributions

In this section we provide references to the publications with more impact authored (or co-authored) by the candidate related to his thesis. From this list, six of them, namely [25, 29, 30, 31, 20, 32], are included in the body of this thesis as mainly contributed by the candidate.

**Articles published in journals indexed in the JCR**

– [32] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Live migration of trans-cloud applications". Computer Standards & Interfaces, 69:103392, 2020. DOI: 10.1016/j.csi.2019.103392.

– [31] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Trans-cloud: CAMP/-TOSCA-based bidimensional cross-cloud". Computer Standards & Interfaces, 58:167–179, 2018. DOI: 10.1016/j.csi.2018.01.005.

**Articles published in non-JCR journals**

– [19] Antonio Brogi, Jose Carrasco, Javier Cubo, Elisabetta Di Nitto, Francisco Durán, Michela Fazzolari, Ahmad Ibrahim, Ernesto Pimentel, Jacopo Soldani, PengWei Wang, and Francesco D'Andria. "Adaptive management of applications across multiple clouds: The SeaClouds approach". CLEI Electronic Journal, 18(1):1-15, 2015. DOI: 10.19153/cleiej.18.1.1.

– [21] Antonio Brogi, Ahmad Ibrahim, Jacopo Soldani, Jose Carrasco, Javier Cubo, Ernesto Pimentel, and Francesco D'Andria. "SeaClouds: a European project on seamless management of multi-cloud applications". ACM SIGSOFT Software Engineering Notes 39(1):1-4, 2014. DOI: 10.1145/2557833.2557844

**Articles published in international conferences**

– [20] Antonio Brogi, Jose Carrasco, Francisco Durán, Ernesto Pimentel, and Jacopo Soldani. "Robust management of trans-cloud applications". In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pages 219–223. IEEE, Milan, Italy, 2019. DOI: 10.1109/CLOUD.2019.00046

– [29] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Component migration in a trans-cloud environment". In 7th International Conference on Cloud Computing and Service Science (CLOSER), *Revised Selected Papers*, pages 286–307. Springer, Porto, Portugal, 2017. DOI: 10.1007/978-3-319-94959-8_15.

– [25] Jose Carrasco, Javier Cubo, Francisco Durán, and Ernesto Pimentel. "Bidimensional cross-cloud management with TOSCA and Brooklyn". In 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pages 951–955. IEEE, San Francisco, California, USA, 2016. DOI: 10.1109/CLOUD.2016.0143.

– [17] Antonio Brogi, Jose Carrasco, Javier Cubo, Francesco D'Andria, Elisabetta Di Nitto, Michele Guerriero, Diego Pérez, Ernesto Pimentel, and Jacopo Soldani. "SeaClouds: An open reference architecture for multi-cloud governance". In 10th European Conference on Software Architecture (ECSA), pages 334–338. Springer, Copenhagen, Denmark, 2016. DOI: 10.1007/978-3-319-48992-6_25.

– [18] Antonio Brogi, Jose Carrasco, Javier Cubo, Francesco D'Andria, Ahmad Ibrahim, Ernesto Pimentel, and Jacopo Soldani. "EU project SeaClouds - adaptive management of service-based applications across multiple clouds". In 4th International Conference on Cloud Computing and Service Science (CLOSER), pages 758–763. SciTePress, Barcelona, Spain, 2014. DOI: 10.5220/0004979507580763.

**Articles published in international workshops**

– [30] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Runtime migration of applications in a trans-cloud environment". In Adaptive Services-Oriented and Cloud Applications (ASOCA) - Workshops of 15th International Conference on Service-Oriented Computing (ICSOC), pages 55–66. Springer, Málaga, Spain, 2017. DOI: 10.1007/978-3-319-91764-1_5.

– [28] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Towards a unified management of applications on heterogeneous clouds". In Advances in Service-Oriented and Cloud Computing - Workshops of 5th European Conference on Service-Oriented and Cloud Computing (ESOCC), *Revised Selected Papers*, pages 233–246. Springer, Vienna, Austria, 2016. DOI: 10.1007/978-3-319-72125-5_19.

– [6] Dionysis Athanasopoulos, Miguel Barrientos, Leonardo Bartoloni, Antonio Brogi, Mattia Buccarella, Jose Carrasco, Javier Cubo, Francesco D'Andria, Elisabetta Di Nitto, Adrián Nieto, Marc Oriol, Ernesto Pimentel, and Simone Zenzaro. "SeaClouds: Agile management of complex applications across multiple heterogeneous clouds". In Projects Showcase - Workshop of Software Technologies: Applications and Foundations 2015 federation of conferences (STAF), pages 54–61. CEUR-WS, L'Aquila, Italy, 2015. URL: `http://ceur-ws.org/Vol-1400/`.

– [27] Jose Carrasco, Javier Cubo, and Ernesto Pimentel. "Towards a flexible deployment of multi-cloud applications based on TOSCA and CAMP". In Advances in Service-Oriented and Cloud Computing - Workshops of 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC), *Revised Selected Papers*, pages 278–286. Springer, Manchester, UK, 2014. DOI: 10.1007/978-3-319-14886-1_26.

**Bibliometric data of the venues of the publications included in the thesis.** In what follows, we summarize the main bibliometric information on the venues on which the papers that are part of the body of this thesis were published. Specifically, we include here the journal Computer Standards & Interfaces, the IEEE International Conference on Cloud Computing, the International Conference on Cloud Computing and Services Science, and the Workshop on Adaptive Service-Oriented and Cloud Applications.

**CS&I:** The journal Computer Standards & Interfaces, published by Elsevier.

– 2019 **Impact factor: 2.809 (Q1 / T1)**
  JCR Category: Computer Science, Software Engineering (25/108)
  JCR Category: Computer Science, Hardware & Architecture (16/53)

– 2018 **Impact factor: 2.441 (Q2 / T1)**
  JCR Category: Computer Science, Software Engineering (29/107)
  JCR Category: Computer Science, Hardware & Architecture (17/53)

**CLOUD:** IEEE International Conference on Cloud Computing.
CLOUD is rated with **A-** in the GII-GRIN-SCIE (GGS) Conference Rating. This rate comes from the following weighted rates: CORE:B, LiveSHINE:A, MA:A-.

**CLOSER:** The International Conference on Cloud Computing and Services Science (CLOSER) is rated as "Work in Progress" in the GII-GRIN-SCIE (GGS) Conference Rating. This rate comes from the weighted rates: LiveSHINE:C, MA:C.

**ASOCA:** The Workshop on Adaptive Service-Oriented and Cloud Applications is not rated in the GII-GRIN-SCIE (GGS) Conference Rating.

## 2.2 Research execution

This section summarizes the papers that support this thesis, and explains how the proposed research challenges are addressed.

The *trans-cloud* approach is presented in [25] and [31] as a step forward on mechanisms related to the management of applications' components on different providers.

The idea behind trans-cloud is that of being able to build applications by using services and resources offered by different providers, at the IaaS or PaaS levels, indistinctly and in combination, according to application requirements. Trans-cloud is based on three main ideas: agnostic topology descriptions, a unified API, and target-service specifications. Regarding the homogenization of cloud services, trans-cloud relies on an agnostic specification of applications' topologies based on the TOSCA standard. This allows full-detailed specifications of applications' components and how they are related between them, indistinctly using IaaS and PaaS services (see *Challenge 1* and [31]).

The trans-cloud approach relies on a CAMP-based single interface that integrates IaaS and PaaS abstraction levels under a common API (*Challenge 2*). The presented prototype is based on Brooklyn, a CAMP-compliant application orchestrator that provides a common API that enables cross-computing features through a unified API for IaaS components. In [31], the Brooklyn API is extended for the management of PaaS services so that both IaaS and PaaS services can be orchestrated as part of the application's lifecycle.

As mentioned in previous sections, the lack of portability of application descriptions is one of the main reasons why developers get lock into platforms. Decoupling the specific vendor and the description of applications requires mechanisms to agnostically target the cloud services to be used through a unified API. The trans-cloud approach uses TOSCA's *placement policies* as add-ons of the topology, so application descriptions can be built in an independent way, without a direct dependency with used resources.

The trans-cloud environment processes application topologies, specified using TOSCA, and uses a homogenization API to orchestrate their deployment and management over IaaS and PaaS services (*Challenge 3*). As a result, only minimal information is needed to specify target resources on which to deploy applications. Indeed, with our approach, each application component may be deployed at one level or the other just by changing its location policy, enabling portability of applications while minimizing the needed effort (*Challenge 4*).

Works [30], [29] and [32] present the evolution of a novel algorithm to migrate components of cloud running applications (*Challenge 5*). In these works, a framework is built over the previously described trans-cloud infrastructure, which has been extended with a migration orchestrator. Taking advantage of the common API and the agnostic topology descriptions, all issues related to resource allocation, credential handling, component interconnection, etc., are handled by the already existing infrastructure. In fact, one of the most promising features of the approach is the capability of analyzing topologies to get architectural information of applications, which is then used to orchestrate the migration process using the common API.

Given an application already deployed using the trans-cloud framework, a migration operation is requested with the set of target locations of the components to be migrated as an input. That is, when a migration of individual components of an application is requested, a number of cloud-agnostic processes are triggered by moving just the necessary components to respective target services, independently of the target providers and abstraction levels, either IaaS or PaaS. By relying on the trans-cloud infrastructure, operations such as *stop*, *re-start*, *move* and *re-connect* are used in the necessary components independently of the service level, the cloud technology or any other dependencies.

In [30], an extension of trans-cloud framework is presented with a first version of the migration orchestrator. This approach enables the reaction to changes in the requirements or cloud environment, and allows carrying out the movement of components to mitigate them. However, this solution has a high impact on the application performance, since, although changes could affect more than one component, this version only provides support for the migration of one component at a time.

Then, an enhanced algorithm is presented in [29]. In this new version, several components may be migrated simultaneously allowing a significant improvement in its efficiency. This paper explains how the migration orchestrator is integrated inside the trans-cloud framework. Its performance is analyzed, and although this new version is able to move components in batches, it uses a blocking-sequential scheduling, so the operations to perform the migration cannot be parallelized and they are executed one after another. As a result, long independent tasks, such as the provisioning of virtual machines, have to wait their turn to be executed. This has a high impact on the migration, and long application downtimes occur due to the lack on the parallelization.

A new iteration of our solution for migration is presented in [32]. The new orchestrator explores the management of the lifecycle of each application component independently and in parallel. In this solution, the trans-cloud observability mechanisms are also improved to ensure the observation of the real component status, which is key to perform the necessary operations on the required components as soon as possible. As experimentation shows (see [32] and *Challenge 5*), this new solution significantly improves the performance of the algorithm and reduces the downtimes during the migration process. This final version also provides a better orchestration of resources during the migration to improve its performance and reduce the impact. For example, to reduce downtimes, it postpones the deletion of old cloud resources until after the migration process is delivered.

Trying to provide a better management of applications' lifecycles we have also explored mechanisms for fault detection and recovery in the trans-cloud context (*Challenge 6*) in [20].

Taking advantage of the common API and agnostic topology, this proposal extends the trans-cloud environment with the capability to react to errors while applications are operated, and recover from them. For example, errors could happen in the resources provisioning phase during an application deployment. In such an event, the error can be detected, and then request the re-provisioning of affected resources. Moreover, interrelations between components can be reestablished, if needed, and failed cloud resources can be deleted. As these mechanisms are developed on our trans-cloud framework, the detection and recovery procedures work no matter the provider or abstraction level of the specific used services.

In this extension for robustness, two new components are added to the trans-cloud ecosystem, namely a model manager and an orchestrator. The model manager is a custom implementation of the fault-aware management protocol presented in [16]. The orchestrator manages applications by means of trans-cloud and handles the process by checking the application's current status in the cloud while they are operated. Then, while the application is being managed, for example during a deployment, this information is compared with the expected state of the system according to its model. When differences are found, the orchestrator assumes that some error has happened, and asks the model manager to calculate a recovery plan to reach the target application status from the current one, and then instructing trans-cloud to achieve it.

From the first trans-cloud approach proposed in [31], every research step has come with a functional prototype that materializes the goals of each phase of the work (*Challenge 7*). Moreover, different motivating examples and case studies have been used in accordance to the requirements of the different solutions. These case studies have been used to analyze the feasibility of each solution and to evaluate the performance and quality of prototypes. Indeed, an extensive experimentation and evaluation of the different prototypes has been carried out, as can be learnt from the different papers that are part of this thesis. For example, more than 2800 deployments were executed to analyze the performance and reliability of the tool for trans-cloud deployment (see [31] and `https://trans-cloud.firebaseapp.com`). Similar procedures were followed to analyze the migration and self-healing mechanisms. In this last case, support for monkey testing was developed, which allowed us to carry on experiments in which running applications were subjected to random failure injections for long periods of time.

## 2.3 Support Papers

This section includes the following research papers, which directly are part of the body of the thesis:

- [25] Section 2.3.1, pages 37–37: Jose Carrasco, Javier Cubo, Francisco Durán, and Ernesto Pimentel. "Bidimensional cross-cloud management with TOSCA and Brooklyn". In 9th International Conference on Cloud Computing (CLOUD), pages 951–955. IEEE, San Francisco, CA, USA, 2016. DOI: 10.1109/CLOUD.2016.0143.

- [31] Section 2.3.2, pages 39–39: Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Trans-cloud: CAMP/TOSCA-based bidimensional cross-cloud". Computer Standards & Interfaces, 58:167–179, 2018. DOI: 10.1016/j.csi.2018.01.005.

- [29] Section 2.3.3, pages 41–41: Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Component migration in a trans-cloud environment". In 7th International Conference on Cloud Computing and Services Science (CLOSER), *Revised Selected Paper*, pages 286–307. Springer, Porto, Portugal, 2017. DOI: 10.1007/978-3-319-94959-8_15.

- [30] Section 2.3.4, pages 43–43: Jose Carrasco, Francisco Durán, and Ernesto Pimentel. "Runtime migration of applications in a trans-cloud environment". In Adaptive Services-Oriented and Cloud Applications (ASOCA) - Workshops of 15th

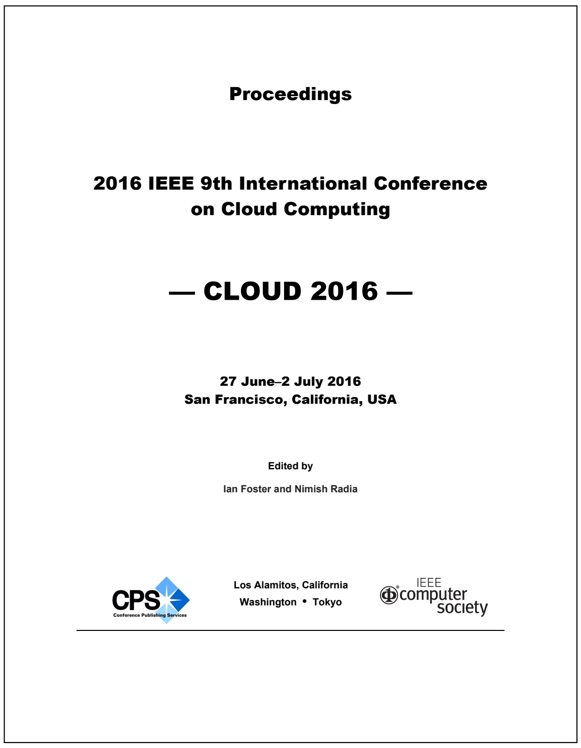
International Conference on Service-Oriented Computing (ICSOC), pages 55–66. Springer, Málaga, Spain, 2017 DOI: 10.1007/978-3-319-91764-1_5.

- [32] Section 2.3.5, pages 45–45: Jose Carrasco, Francisco Durán, and Ernesto Pimentel. “Live migration of trans-cloud applications”. Computer Standards & Interfaces, 69:103392, 2020. DOI: 10.1016/j.csi.2019.103392.

- [20] Section 2.3.6, pages 47–47: Antonio Brogi, Jose Carrasco, Francisco Durán, Ernesto Pimentel, and Jacopo Soldani. “Robust management of trans-cloud applications”. In 12th International Conference on Cloud Computing (CLOUD), pages 219–223. IEEE, Milan, Italy, 2019. DOI: 10.1109/CLOUD.2019.00046.

### 2.3.1 Bidimensional cross-cloud management with TOSCA and Brooklyn

**Title:** Bidimensional cross-cloud management with TOSCA and Brooklyn
**Authors:** Jose Carrasco, Javier Cubo, Francisco Durán, and Ernesto Pimentel.
**Publication:** 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pages 951–955. IEEE, San Francisco, California, USA, 2016.
**DOI**: 10.1109/CLOUD.2016.0143.
**URL**: `https://ieeexplore.ieee.org/abstract/document/7820380`.
**Abstract**: The diversity in the way different cloud providers offer their services, give their SLAs, present their QoS, support different technologies, etc., complicates the portability and interoperability of cloud applications, and favors vendor lock-in. Standards like TOSCA, and tools supporting them, have come to help in the provider-independent description of cloud applications. After the variety of proposed cross-cloud application management tools, we propose going one step further in the unification of cloud services with a deployment tool in which IaaS and PaaS services are integrated into a unified interface. We provide support for applications whose components are to be deployed on different providers, indistinctly using IaaS and PaaS services. The TOSCA standard is used to define a portable model describing the topology of the cloud applications and the required resources in an agnostic, and providers- and resources-independent way. We include in this paper some highlights on our implementation on Apache Brooklyn and present a non-trivial example that illustrates our approach.


Proceedings

2016 IEEE 9th International Conference on Cloud Computing

— CLOUD 2016 —

27 June–2 July 2016
San Francisco, California, USA

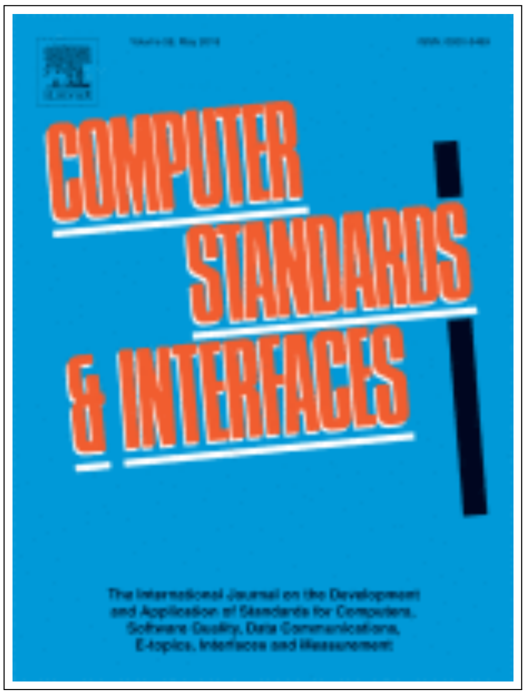
Edited by
Ian Foster and Nimish Radia

Los Alamitos, California
Washington • Tokyo

### 2.3.2 Trans-cloud: CAMP/TOSCA-based bidimensional cross-cloud

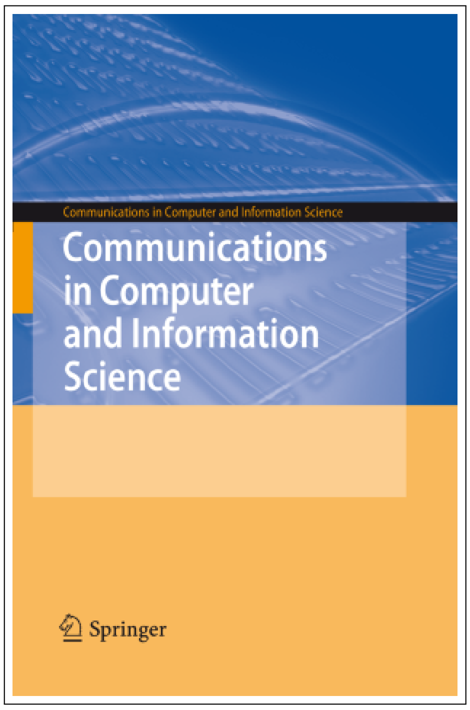
**Title:** Trans-cloud: CAMP/TOSCA-based bidimensional cross-cloud
**Authors:** Jose Carrasco, Francisco Durán, and Ernesto Pimentel.
**Publication:** Computer Standards & Interfaces, 58:167–179, 2018.
**DOI**: 10.1016/j.csi.2018.01.005.
**URL**: `https://www.sciencedirect.com/science/article/pii/S0920548917303185`.

**Abstract**: The diversity in the way in which different cloud providers offer their services, give their SLAs, present their QoS, or support different technologies complicates the portability and interoperability of cloud applications, and favors vendor lock-in. Trying to solve these issues, we have recently witnessed the proposal of unified APIs for IaaS services, unified APIs for PaaS services, and a variety of cross-cloud application management tools. We go one step further in the unification of cloud services, building on the TOSCA and CAMP standards, with a proposal in which the management of IaaS and PaaS services, possibly offered by different providers, are integrated into a unified interface. The TOSCA standard is used for the definition of portable models describing the topology of cloud applications and the required resources in an agnostic, providers-and-resources-independent way. Based on the CAMP standard, we abstract from the particularities of specific providers. Indeed, to change the service on which any of the modules of an application is to be deployed, whether it be IaaS or PaaS, we just need to change its target location by picking from the catalog of supported locations. We provide insights into our implementation on Apache Brooklyn, present a non-trivial case study that illustrates our approach, and show some experimental results.



### 2.3.3 Component migration in a trans-cloud environment

**Title:** Component migration in a trans-cloud environment.
**Authors:** Jose Carrasco, Francisco Durán, and Ernesto Pimentel.
**Publication:** 7th International Conference on Cloud Computing and Services Science (CLOSER), *Revised Selected Paper*, pages 286–307. Springer, Porto, Portugal, 2017.
**DOI**: 10.1007/978-3-319-94959-8_15.
**URL**: `https://link.springer.com/chapter/10.1007/978-3-319-94959-8_15`.
**Abstract**: The *trans-cloud* approach has recently been proposed to simplify the development and operation of cloud applications, and to minimize the *lock-in problem*. The three key ingredients of the *trans-cloud* approach are: agnostic topology descriptions, a unified API, and mechanisms for the independent specification of providers' services. We build on the trans-cloud mechanisms to propose a solution for the migration of stateless cloud components at runtime. In the context of our trans-cloud tool, we propose an algorithm for the migration of cloud applications' components between different providers, possibly changing their service levels between IaaS and PaaS. We present an implementation of our proposed solution, and illustrate it with a case study and experimental results.


Communications in Computer and Information Science
Communications in Computer and Information Science
Springer

### 2.3.4 Runtime migration of applications in a trans-cloud environment

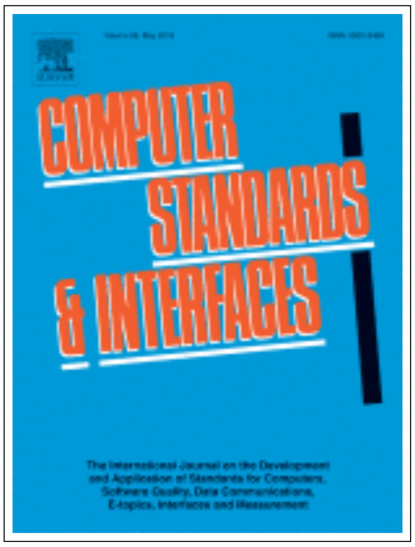
**Title:** Runtime migration of applications in a trans-cloud environment.
**Authors:** Jose Carrasco, Francisco Durán, and Ernesto Pimentel.
**Publication:** Adaptive Services-Oriented and Cloud Applications (ASOCA) - Workshops of 15th International Conference on Service-Oriented Computing (ICSOC), pages 55–66. Springer, Málaga, Spain, 2017.
**DOI**: 10.1007/978-3-319-91764-1_5.
**URL**: `https://link.springer.com/chapter/10.1007/978-3-319-91764-1_5`.
**Abstract**: Making an application independent of the cloud provider where it is going to be deployed is still an open issue. In fact, cloud agnostic software development still presents important challenges to be solved, and one of them is the problem of runtime migration of components already deployed on a given provider to a different one. Even more difficult is dealing with the interoperability issues when the migration also implies a change of service level (*i.e.*, from IaaS to PaaS, or *vice versa*). This paper presents an algorithm for the parallel migration of cloud applications. The migration is performed component-wise, in the sense that each component of the application to be migrated may be deployed on a specific service on a specific provider, and be moved to a different provider, possibly changing the service level between IaaS and PaaS of each of them individually. Since the migration of components with state pose additional difficulties, we only consider stateless components. Our solution relies on three of the key ingredients of the *trans-cloud* approach: a unified API, agnostic topology descriptions, and mechanisms for the independent specification of providers. We show how our approach solves some of the current interoperability and portability issues of cloud environments, and allows us to provide a solution for migration. We present an implementation of our proposed solution and illustrate it with a case study and experimental results.

Lars Braubach · Juan M. Murillo
Nima Kaviani · Manuel Lama
Loli Burgueño · Naouel Moha
Marc Oriol (Eds.)
Services Science
LNCS 10797
Service-Oriented Computing – ICSOC 2017 Workshops
ASOCA, ISyCC, WESOACS, and Satellite Events
Málaga, Spain, November 13–16, 2017
Revised Selected Papers
Springer

### 2.3.5 Live migration of trans-cloud applications

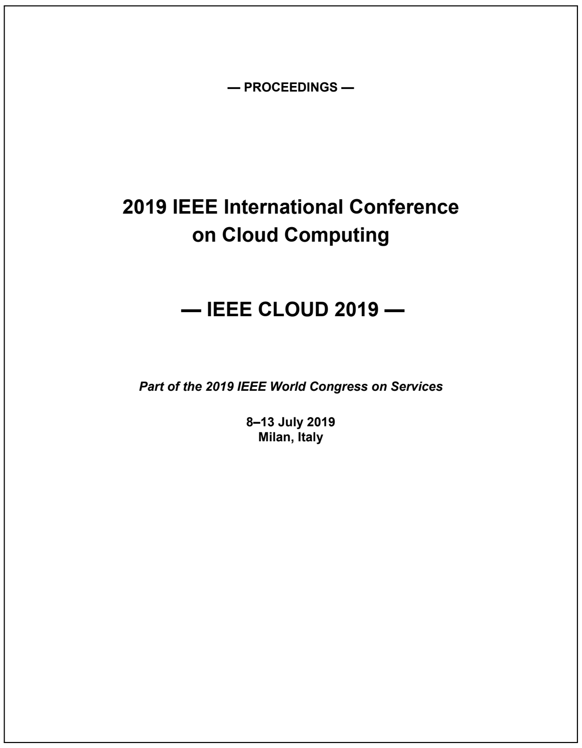
**Title:** Live migration of trans-cloud applications.
**Authors:** Jose Carrasco, Francisco Durán, and Ernesto Pimentel.
**Publication:** Computer Standards & Interfaces, 69:103392, 2020.
**DOI**: 10.1016/j.csi.2019.103392.
**URL**: `https://www.sciencedirect.com/science/article/pii/S0920548919300984`.

**Abstract**: The development of applications independent of the cloud providers where they are going to be deployed is still an open issue. In fact, cloud agnostic software development presents important challenges to be solved. One of these issues is the run-time migration of components. Even more difficult is dealing with the interoperability issues when the migration also implies a change of provider or service level. This paper presents a solution for the component-wise migration of cloud applications. The migration is performed component-wise in the sense that each component of the application to be migrated, which may be deployed on a specific service on a specific provider, may individually be moved to a different one. Our solution relies on the three key ingredients of the *trans-cloud* approach, where the CAMP and TOSCA standards play a central role: a CAMP-based unified API, TOSCA-based agnostic topology descriptions, and mechanisms for the independent specification of target service locations. The effort and the time required for restoring the activity of applications are used as metrics to evaluate the performance of the proposed migration orchestrator. Although the time required for the migration operation is directly related to the topology of the application, the affected components, and their previous and target locations, the downtimes are significantly reduced. Moreover, thanks to the abstraction level at which it operates and the automation provided, the effort needed from the user for a migration operation is almost zero. We present an implementation of our proposed solution and illustrate it with a case study and experimental results.


COMPUTER
STANDARDS
& INTERFACES
The International Journal on the Development and Application of Standards for Computers, Software Quality, Data Communications, E-topics, Interfaces and Measurement

### 2.3.6 Robust management of trans-cloud applications

**Title:** Robust management of trans-cloud applications.
**Authors:** Antonio Brogi, Jose Carrasco, Francisco Durán, Ernesto Pimentel, and Jacopo Soldani.
**Publication:** 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pages 219–223. IEEE, Milan, Italy, 2019.
**DOI**: 10.1109/CLOUD.2019.00046.
**URL**: `https://ieeexplore.ieee.org/document/8814490`.
**Abstract**: The *fault* handling and recovery from runtime failures of cloud applications should be done by taking into account the inter-dependencies occurring among their components, and by dealing with the diverse and heterogeneous cloud offerings used to host them. The latter is even harder in trans-cloud scenarios, i.e., when application components are possibly deployed on different platforms and at different service levels (IaaS or PaaS). In this paper, we propose a methodology to support the automated management and recovery of (un)foreseen failures in a trans-cloud application, which takes into account all interdependencies occurring among its components. We then present a prototype implementation of our proposal, consisting of an orchestrator that exploits a management framework for trans-cloud application deployments, together with management protocols for the automated planning of the fault-aware administration of applications.


— PROCEEDINGS —

2019 IEEE International Conference
on Cloud Computing

— IEEE CLOUD 2019 —

*Part of the 2019 IEEE World Congress on Services*

8–13 July 2019
Milan, Italy

# Chapter 3. Related Work

Cloud application interoperability and orchestration have generated a considerable amount of interest in the literature (cf. [98, 106]). There have been numerous approaches to cloud application orchestration, each with its advantages and limitations. Several approaches focus on the creation and use of standards, others on the use of libraries and intermediate layers, while others exploit the semantics of models. To gain an appreciation for how our approach compares to other related approaches, we present in this chapter an analysis of the state of the art in the field the work lies on, relating the solution proposed in this thesis with other proposals in its context, including standards, other academic and industrial proposals, and open-source proposals.

First, without reviewing the history of cloud computing, let us place cloud computing in context by relating it to grid computing, edge computing, and the Internet of Things (IoT). Both cloud and grid computing provide network-based distributed computing resources. Grid computing aims to accomplish tasks by dividing them into independent subtasks, using a large number of interconnected computers to achieve the maximum computing capacity. Cloud computing targets a similar goal, but in this case resources are accessed via services over the internet, without direct access to such resources, and with mechanisms to flexibly scaling the computing capacity in accordance with users' requirements. The powerful cloud datacenters provide virtually unlimited computation and data storage resources. However, they may suffer from limited bandwidth and network latency. This is the case for IoT applications, where the amount of data generated by mobile devices and sensors does not allow their upload to the cloud for their processing. Edge computing consists in the use of computing and network resources that lay between data sources and cloud datacenters. Thus, edge computing allows the processing of the raw data before it is uploaded to the cloud. The combination of cloud and edge computing enables organizations to store and process the big amount of valuable data generated from IoT devices.

Although cloud computing has many benefits, it also presents several challenges, mainly related to privacy and security concerns in public clouds, failure recovery, poor performance in high-demand situations, and lack of interoperability between services. Since the focus in this thesis is mainly on cloud interoperability, let us focus here on the state of the art on this topic.

Many studies have addressed the service interoperability and the vendor lock-in problem. Solutions have been proposed based on standardization [44, 91, 90, 89, 40, 92], brokering [99, 63], model-driven approaches [4], and semantic-based solutions [82, 63, 99, 3]. In the literature, we can find several surveys of the state of the art on service interoperability (see, e.g., [96, 113, 58, 106, 85, 64, 13]). These surveys focus on different aspects of interoperability. For instance, [113, 64] focus on interoperability related to the vendor lock-in problem, mainly in the context of IaaS. Their conclusions are also quite diverse. For instance, [96, 106] claim that standardization is the solution to the interoperability problems.

In this chapter, we provide a brief summary on the state of the art on approaches in cloud service interoperability, focusing on client-centric solutions, in order to highlight the tendencies and the current trends. In what follows, we analyze work related to issues achieved in this thesis. In Section 3.1 we focus on cloud standards and how they deal with vendor-lock-in issues. We present in Section 3.2 different approaches to deal with the cloud heterogeneity to solve portability and interoperability issues. In Section 3.3 we present migration-related works, and in Section 3.4 we describe different approaches for self-healing. Finally, although the current trans-cloud implementation does not support container-based technologies, since they are a cloud alternative widely used in both academy and industry, in Section 3.5 we analyze vendor-lock-in issues, migration, and self-healing in the context of container-based solutions.

## 3.1 Standards

Cloud standardisation has been very active during the last years (see, e.g., [77, 96]). Given the increased number of new providers, services, functionalities, and technologies, relevant associations, such as IEEE,[1] DMTF,[2] and OASIS,[3] are working on defining standards to tackle portability and interoperability problems, promoting a normalization of the usage of cloud solutions to deal with vendor lock-in-related issues. For example, *The Guide for Cloud Portability and Interoperability Profiles* [94] is among the currently active IEEE projects. In what follows, we summarize the main goals of the most relevant efforts:

**DMTF – Interoperable Clouds.** Different approaches try to deal with different sides of the problem. For example, DMTF (Distributed Management Task Force), an organization participated by groups and companies which develops and promotes standards for IT on industrial environments, in their document "Interoperable Clouds - A White Paper from the Open Cloud Standards Incubator" [39] defines *Interoperable Cloud* and provides an analysis of the interoperability challenge. They list key aspects to consider, such as protocols for the management of the resources, artifacts packaging, and security mechanisms. They define interfaces to homogenize the interaction with services,

[1]Institute of Electrical and Electronics Engineers (IEEE): `https://www.ieee.org/`.

[2]Distributed Management Task Force (DMTF): `https://www.dmtf.org/`.

[3]Organization for the Advancement of Structured Information Standards (OASIS): `https://www.oasis-open.org/`.

and mechanisms to ensure similar functionalities, such as SLA adaptation, facilitating the development and execution of applications.

**CIMI.** Also by DMTF, CIMI (Cloud infrastructure Management Interface) [40] is a proposal that targets to unify the management and interaction among clouds at IaaS level. It models every element of the platform and the interactions using an HTTP-based interface (REST), and it specifies mechanisms to audit and discover available services in the platforms, normalizing the interaction with the providers. Then, users can specify a plan to describe how an application can be deployed on the cloud, in an agnostic way. Moreover, the discovery mechanisms allow to find services with similar capabilities on different providers, ensuring functional and non-functional requirements regardless of the provider where the applications run.

**OCCI.** OCCI (Open Cloud Computing Interface) [92] is a standard managed by OGF (The Open Grid Forum).[1] It offers a full model of the interaction with platforms by means of an HTTP protocol (RESTful), an API, which specifies services, resources, artifacts, and networking. Like TOSCA (and therefore trans-cloud), the OCCI standard offers an exhaustive metamodel to specify the application's architecture, allowing to describe applications' components, their relations, and how they interact with the used services. As already mentioned, services can also be specified: The OCCI standard defines the *action* concept, which defines the functionalities and capabilities of offered services, adding some semantics to the common API. OCCI-compliant solutions can process application models, and use the proposed API to provision the necessary resources and distribute the application. The OCCI standard also provides some technical descriptions about how the specification can be integrated on a platform, in fact, OCCI works with open-code projects such as OpenStack and OpenNebula. About the scope of the reachable services, OCCI defines an interface to reach different kinds of services, IaaS and PaaS, and some orchestration processes, but it does not specify native mechanisms to orchestrate cross-cloud deployments using services at different level.

**OASIS – CAMP.** Cloud Application Management for Platforms (CAMP) [90] is one of the standards that inspired the trans-cloud solution. It is defined by OASIS, an organization with members from both academia and industry with several working groups to study different problems in the IT context. Like CIMI and OCCI, CAMP defines a common API to be implemented by the different platforms, which allows to describe every involved part in the vendor lock-in problem. It defines models for cloud resources, services, and applications, and how all of them are related to the platform. The API also models platforms' services, such as those to deploy, update and delete applications. Moreover, CAMP introduces a set of concepts which have been used in the trans-cloud proposal. For instance, *sensors* and *effectors* [31] allow to audit the status of the applications and the bounded resources and how interacting with them, to

[1]The Open Grid Forum: `https://www.ogf.org/`.

enable the post-deployment management, e.g., the usage of elasticity policies and the development of runtime migration techniques. Regarding portability issues, the standard defines a custom packaging PDP (Platform Deployment Package) whose management is part of the core of the standard. A PDP contains the artifacts to deploy and a description of the required services and resources based on the generic API. PDP enables some form of portability between CAMP-compliant solutions, since this standardized packaging mechanism allows the specification of a deployment engine which is able to orchestrate the deployment of an application by inferring a deployment plan using the PDP specifications. However, unlike trans-cloud, the efforts in the CAMP standard do not focus on getting a heterogeneous multi-cloud deployment, since an application is managed and deployed by a specific platform. Trans-cloud goes one step further, because, despite it also uses a generic API, while CAMP focuses on the PaaS abstraction level, trans-cloud reaches both, IaaS and PaaS. Furthermore, CAMP does not provide a way to specify the topology of applications.

**OASIS – TOSCA.** TOSCA (Topology and Orchestration Specification for Cloud Applications) [91] is one of the most relevant standards in the current cloud context (cf. [81]). TOSCA is a language to describe a topology of cloud-based web services, their components, relationships, and the processes that manage them. The provisioning of comprehensive descriptions of applications' topologies is key in the trans-cloud approach, and TOSCA is used to describe such topologies [27, 31]. The use of TOSCA topology descriptions provides architecture-independent models of applications' components and how they are related between them and with the required cloud resources. Moreover, it also offers a thorough description of capabilities, limits of each topology element, and a description of non-functional requirements, as the minimum and maximum amounts of required resources to work. As above mentioned, this description allows applications to be distributed using different kinds of services and platforms, tackling the challenges of an agnostic delivery system. Like CAMP, TOSCA also provides a normalized packaging mechanism to address portability, the CSAR (Cloud Service ARchitecture), which can be used to infer plans to distribute the applications. The TOSCA standard provides generic interfaces to describe the topology's components and their lifecycle operations, but the standard does not define an API to unify the cloud management, and it delegates the integrity and compatibility of the TOSCA-compliant solution to the providers themselves. TOSCA's complete topology descriptions enable the automatic management of the application distribution, however, in TOSCA, the multi-cloud distribution is a challenging task (cf. [108]), problem that has been tackled in our trans-cloud approach. Moreover, TOSCA defines a generic topology description, but the standard is, like CAMP, mainly focused on one abstraction level, namely IaaS, which hinders having an agnostic multi-level cloud management system.

**Discussion.** Like our trans-cloud proposal, previous solutions based on standardization advocate an homogenization of the usage of platforms, but with some differences. Our trans-cloud solution offers an orchestration process to manage applications which is inferred from their topologies, but other solutions do not specify a topology to define applications' compositions, that we define as a key element of our proposal to have an orchestration process for the deployment of applications [26, 27, 10]. Then, it is necessary to write a plan description for each operation to be executed. For example, the application distribution requires a detailed description about what and how required operations are used. By taking advantage of the common API, the plan is portable, so it allows distributing the application on different providers. Without this description, changes in the application's topology would involve as well modifications on the deployment plan. Moreover, these solutions are not level agnostic. For instance, Interoperable Clouds [39] defines generic and adaptable mechanisms to use different kinds of services, but it does not specify a homogenous management of different abstraction levels. Finally, these solutions have to be integrated directly on the platforms, which would make its implementation very cumbersome. In fact, to date, none of them has a full implementation.

In summary, each standard addresses a different kind of problem inside the more general vendor lock-in context. Different approaches have been proposed. Some of them define their own API, as CAMP and OCCI. Others, like TOSCA, strives on providing an exhaustive and flexible metamodel, to ensure interoperability. However, not only resources and topologies are modeled, some of these standards also define a generic packaging mechanism to allow, together with the topology, the deployable artifacts, and the required services and resources specification. Each of these points solves part of portability and interoperability context. Although there are some problems and limitations, the good news is that some of these standards are in a continuous review and improvement. For example, TOSCA added in its last revisions complete support to container-based technologies. Indeed, as we will see in the following sections, it is difficult to identify and deal with the problems related to cloud heterogeneity, and then provide a final and complete solution. Our trans-cloud solution tries to address several of the aforementioned questions and issues, such as the agnostic-level simultaneous deployment of components on IaaS, PaaS and on-premise infrastructures, or the orchestration of the distribution process, as well as questions on migration, which we analyze in Section 1.1.2. Of course, these standards also try to answer many other questions. Topics such as artifacts packaging or SLA-based service discovery are currently out of the trans-cloud proposal.

## 3.2 Portability and interoperability

Portability and interoperability are the main topics in the study of the vendor lock-in problem. Several causes of this problem can be identified, but possibly the lack of standard mechanisms is the main one. Indeed, dealing with cloud heterogeneity is the most important challenge. As we have seen in the previous section, several standards agree on the need for the unification of providers and normalized management, and on the normalization of application descriptions. These ideas may indeed serve as a basis for solutions for portability, deployment orchestration, discovery and adaptation of services, etc. In most cases, platforms offer resources and services with similar capabilities and functionalities, but the different APIs to use these services and the mechanisms to manage them hinder reaching an integration layer to provide a generic usage of services. These differences prevent us from the possibility of finding solutions for more complex problems, such as cross-deployment.

We can observe in the current state of the art that the problems of portability and interoperability are subdivided into different challenges to solve. Indeed, most of the works in the literature focus on specific subproblems, and try to face them by defining their own scopes and using different mechanisms. For example, some of them unify the management of abstraction levels, others try to address the orchestration of the deployment process, others propose a full implementation of one standard, while others just try to take advantage of concepts or contributions by some standard. In the following sections, we discuss and classify the related works on portability and interoperability in accordance to the approach followed to solve the problem common APIs, federating and brokering clouds, cloud-coupled and -de-coupled orchestrators, commercial solutions, and modeling for applications and platforms.

### 3.2.1 Common API

The unification of cloud providers' interfaces is one of the key aspects of this thesis. The trans-cloud mechanisms are built on the jClouds library to handle IaaS services. Moreover, as part of the development of trans-cloud, support for the unified access to PaaS services has been developed. In this section, we analyze some PaaS unification approaches, namely COAPS, Nucleus, and PaaS-Manager.

**COAPS.** [109] and [60] describe COAPS, a generic API to manage PaaS services of some providers, such as Google App Engine, Cloud Foundry, OpenShift, etc. They define different models to represent application components and cloud services and how they are related. Moreover, they offer a little overview about how the vendors heterogeneity affects model composition. However, deploying an application using the COAPS API requires the developer to provide the application's source archive in the same format that is required by the targeted PaaS platform. In the trans-cloud approach, we provide a uniform interface, independent of the abstraction level, and which indeed allows us to move components from IaaS to PaaS and vice versa just by selecting the target service from the catalog of available services. Moreover, whereas COAPS focuses on the management of PaaS services, we have proposed a solution solving the portability and interoperability issues between multiple providers and abstraction levels.

**Nucleus.** In [69], Kolb and Wirtz provide a conceptual analysis and classification of PaaS trends and contexts. Under the assumption that different PaaS offerings provide different capabilities, they classify platforms' approaches according to their capabilities and the use of IaaS and SaaS services. Like in our trans-cloud proposal, they focus on a concrete kind of generic platforms, providing different perspectives about the portability and migration of systems between platforms. Based on the aforementioned information and classification, they build a PaaS profile to describe, in a comparable and matchable way, the core functional capabilities about offerings, providing a model, and a taxonomy to represent the knowledge about platforms. As in our work, they research about the portability of applications and how their dependencies should be described to ensure such portability. In this aspect, we go one step further in our proposal, since we provide a standard-based application profiling to describe dependencies and ensuring their automatic management. The authors of Nucleus claim that IaaS and PaaS management should be treated separately in terms of portability, whereas we have attempted a uniform solution to the portability management through IaaS and PaaS by means of a common API. Their profile covers a significant number of features and services of many different providers. Our trans-cloud proposal may benefit from their effort if a broader number of providers and services is to be considered in the trans-cloud solution.

In [68], Kolb and Röck present an interface to unify core management functions of cloud platforms, providing mechanisms for the management of application operations and the cloud environment during the application lifecycle. They validate their proposal with a reference implementation, Nucleus,[1] for some cloud platforms (Heroku, Cloud Foundry, etc.). In Nucleus, each provider is represented by a concrete adapter which implements the unified API, and the generic interface allows the application to be represented programmatically, deployed, and managed over the supported vendors. However, in our approach we go one step further and we provide an environment to define standard-based application topology descriptions. Furthermore, as already mentioned, our goal was to unify IaaS and PaaS, providing a unified set of operations and lifecycle management for both contexts in a uniform way.

In [67], Kolb, Lenhard, and Wirtz study the effort of deploying applications across different PaaS providers. They develop an automatic Docker-based deployment system that uses a simple interface for the interaction with a cloud and a set of modules to manage different PaaS vendors. This provides an isolated and reproducible deployment process that allows to measure the interaction with PaaS providers. The integration of each new provider inside their infrastructure requires the deployment of a new module that has to implement the unified API by means of a set of bash files. These files provide the API's generic operations to interact with the cloud, but they have to be added manually to each different application description to define how components are deployed in the cloud. However, in our approach, the trans-cloud infrastructure offers an isolated hierarchy to extend the supported providers (IaaS and PaaS) by adding new locations, thus decoupling application descriptions and target locations.

**PaaS-Manager.** In [34], Cunha et al. also propose an abstraction layer, called PaaS-Manager, to aggregate several relevant PaaS public offerings. As in [68, 67], the authors expose a common API for developers and define a very illustrative modular architecture for PaaS APIs unification. Cuhna, Neves, and Sousa develop an adaptable API architecture based on modules that can be extended to integrate both public and on-promise platforms. Their approach addresses provider interoperability and application porta-

[1]Nucleus: `https://github.com/stefan-kolb/nucleus`.

bility to reduce the vendor lock-in problem, as we propose in our work. However, like previous studies, it only considers PaaS offerings. Moreover, it just offers a programmatical solution, with no application model, and therefore not being able to provide any automatic management of the components. Regarding interfaces, their solution offers APIs (as libraries) that allow the interaction with providers' services. However, they do not provide any logic to manage or orchestrate application's lifecycle-based processing, as the deployment or post-deployment operations. It must be the users of the libraries who define agnostic-provider processes using those generic interfaces.

### 3.2.2 Federated clouds

Cloud Federations have gained momentum in the last years, with the idea of a platform where users can select the infrastructure in which to deploy their software between a set of third-party solutions. Federated clouds offer a solution for provider integration.

**FraSCAti.** In [97], Paraiso et al. define a federated platform which integrates external services at IaaS and PaaS levels. Like in our approach, they believe that a principled definition of heterogeneous services is the first step to deal with vendor lock-in. This work also advocates for the normalization of application descriptions to address portability. They also use an OASIS standard, Service Component Architecture (SCA) [89], to specify applications' components, relations, communications, etc. Then, platforms process the applications and enable the multi-cloud deployment of services in a network of federated distributed nodes on IaaS and PaaS vendors. The framework is in charge of the interconnection of related services according to the application's specifications. As our approach, this work provides mechanisms for the multi-cloud application deployment orchestration based on a standard-based and unified provider management. However, they require the previous deployment of federated nodes, while our solution works directly on providers' APIs, thus allowing more flexibility, for example, to integrate with on-premise infrastructure. Furthermore, we provide a topology-oriented application knowledge, while they use a service-oriented one which focuses on service-level information. This approach makes more difficult the post-deployment management of applications.

**PacificCloud.** In [93], Carvalho et al. propose a decentralized and lightweight architecture based on microservices for multi-cloud interoperability called PacificClouds. In their approach, users may choose the cloud in which to execute each microservice of the application. The solution is based on the usual asynchronous microservices communication mechanisms. With this approach, they are able to address both horizontal and vertical interoperability scenarios in IaaS and PaaS, providing an architectural solution to have multi-cloud in federated clouds.

**The Inter-Cloud Architecture framework.** In [37], Demchenko et al. introduce the Inter-Cloud Architecture (ICA) framework, a multilayer model for addressing interoperability. The ICA solution is based on three main components: (1) a multilayer cloud-services model to define the inter-layer interfaces between the cloud service models, (2) a plan for the inter-cloud management and control, and (3) a framework for inter-cloud federation to enable the federation of independent clouds and their related infrastructure components.

**Cross-Cloud Federation Manager.** In [33], Celesti et al. define a cross-cloud federation model based on three phases to enable the cloud interconnections to establish the cloud federation: discovery of resources, matchmaking of services, and authentication between providers. As implementation of this model they propose Cross-Cloud Federation (CCFM), a module that is added on top of each member federation's cloud infrastructure, which is in charge of the management of local resources, such as provisioning and connecting with other federated resources. Multiple coordinated CCFMs use a distributed strategy to discover services between them based on peer-to-peer connections, enabling elasticity since members can be dynamically added or removed if needed. A matchmaking module is in charge of finding the best set of resources to run an application according to its requirements. Finally, an authentication module creates a secure connection between providers, making transparent the cloud usage to the federation users.

These solutions provide vendor-side mechanisms to deal with different providers by defining a federated cloud that homogenize the cloud management. However, these solutions do not include application modeling, which is very useful to build portable applications and minimize the impact of changes in both the cloud side and the application side.

### 3.2.3 Broker-based solutions

**The mOSAIC project.** The main goal of the mOSAIC project [99] is to offer access to heterogeneous resources from multiple clouds. The mOSAIC open-source platform integrates the management of cloud vendors to assist in the deployment process of applications, addressing portability and interoperability issues. Its authors propose a cloud ontology to detail application knowledge. They specify the required cloud services and resources via a common API that unifies the management of providers. Furthermore, their ontology also allows to specify the non-functional requirements of applications, what, combined with a multi-agent brokering mechanism and SLA service, enables a negotiation which allows searching for services, matchmaking the applications' requests, and possibly composing the requested service if no direct hit is found. The mOSAIC solution can manage only IaaS-based services, which are described using the ontology, and managed by the unified APIs. These services are then offered as services in the mOSAIC PaaS layer, but it does not directly integrate external real-PaaS solutions. The matchmaking capabilities of the trans-cloud solution rely on jClouds, which is currently only available for IaaS resources.

**Cloud4SOA.** Cloud4SOA [63] uses a broker-based architecture to address interoperability between providers and facilitate the multi-PaaS deployment and the management of applications' lifecycles. The solution is based on a SOA architecture, and it can also offer the best matches to their computational needs. Users can describe their *Application Profiles* using a programmatically-oriented DSL that contains the data necessary to carry out the matchmaking of the application component's constraints, such as SLA requirements, and find the best services to enable applications' deployment. Like our proposal, Cloud4SOA reduces the risk of vendor lock-in by taking advantage of a unified management, by means of something that its authors call harmonized API and a semantic layer. By means of this API, Cloud4SOA is able to manage multi-PaaS on public, on-premise and hybrid platforms, and address interoperability between them.

**PaaSport.** PaaSport [7] also defines a cloud broker based on a PaaS ontology to resolve portability issues between PaaS providers. They define three different semantic levels. The first one aims to describe functional and non-functional details of PaaS platforms. The second one allows users to describe their applications and their requirements. The last model allows detailing the SLA requirements for an application. The first and second models are key to allow to infer and perform the needed operations to reach a concrete PaaS platform, by means of a common API to operate with different providers. In addition, PaaSport defines a recommendation algorithm that uses the application information and SLA requirements for a matchmaking process, recommending the best providers and services to optimize the distribution of applications. This project shares some similarities with Cloud4SOA, since both of them try to solve issues in the PaaS context using ontology and semantic models. In fact, PaaSport models are inspired on Cloud4SOA research. However, Cloud4SOA presents a better management of the application's lifecycle, including capabilities to monitor applications and apply scaling techniques if the workload variates. PaaSport however uses a more detailed modeling of providers.

As our proposal, all of these solutions try to mitigate cloud heterogeneity using the modeling of providers and applications, in these cases by means of ontologies, and use this information to translate application's needs to real actions to operate concrete providers. However, they are bound to a concrete abstraction level, IaaS or PaaS, whereas our trans-cloud solution offers an agnostic level environment to operate on both of them. Moreover, they use their own models to describe applications and environments, instead of using standard-oriented solutions.

### 3.2.4 Cloud-coupled orchestrators

We find in the literature several works which present platforms that integrate services and providers via public APIs and allow customers to reach diverse providers dealing with heterogeneity.

To manage heterogeneous services, some projects, as in our solution, use abstracted or common interfaces to support different providers, thus decoupling application descriptions and cloud integration. However, other more coupled solutions use an ad-hoc description of the services management, or requires to modify the application topology to reach new providers. In this section, we analyze different coupled solutions, whereas decoupled approaches will be described in the following section.

**Roboconf.** Roboconf [100] is an open-source distributed-application orchestration framework for multi-cloud platforms. It provides mechanisms based on the modeling of applications and resources using Domain Specific Language (DSL) techniques. This modeling includes descriptions of application distribution and lifecycle management. It is mainly focused on IaaS integration, but Roboconf provides a generic and extensible infrastructure where new providers, including PaaS, would be added by means of a set of configuration and DSL-based description files, to define the interaction with the provider. To deploy applications, developers have to provide some DSL-based configuration files plus some additional files, and artifacts to specify the necessary steps to deploy and execute the application over the target providers. The application description and the target services are very dependent, and therefore it is very complicated to modify the target providers without affecting the application models. This is an important difference with respect to our proposal, where just an agnostic TOSCA-based application description is needed, which remains stable, avoiding laborious behavior and management specification and allowing target providers to be easily managed by means of minor decorations of the modeling. However, probably the most important difference is that our trans-cloud solution offers modules that are designed to be indistinctly deployed on IaaS and PaaS.

**OpenTOSCA.** The OpenTOSCA ecosystem[1] offers a modeling tool for TOSCA-based cloud applications. Specifically, it offers a graphical topology modeling editor for the TOSCA specification of systems, enabling a collaborative development of TOSCA-based application topologies. The OpenTOSCA environment offers a container, namely, OpenTOSCA [8, 71], which can process TOSCA-based applications. It offers a tool and generic mechanisms to take advantage of the standard's capabilities and flexibility. In fact, it could support both IaaS and PaaS deployment levels, but it does not provide yet a predefined set of Node Templates to easily represent and manage providers of different abstraction levels. Therefore, the Service Templates must provide the required implementation artifacts that support the different vendors' services. Furthermore, since the container expects that the topologies use Node Templates to describe cloud resources, it is necessary to modify the node templates of an application to select new target providers, what means that new compatible Node Templates should be used for the application's components. In contrast, our approach takes advantage of statement policies (a.k.a. deployment policies) to describe locations in an independent way (using policies), giving place to stable application topologies, and facilitating the modification of the target cloud providers (IaaS and PaaS) without remodeling any topology element. As prescribed by the TOSCA standard, an explicit plan specified in a workflow service, such as BPEL (Business Process Execution Language) or BPMN (Business Process Model

[1]OpenTosca: `https://www.opentosca.org/`.

and Notation), has to be provided together with the application, so that Open-TOSCA can define the application orchestration. There has been some recent research on the support of declarative plans in the OpenTOSCA ecosystems. In [65], a hybrid solution based on provisioning policies is proposed following the concepts described in [15]. In this approach, application topologies and provisioning policies are analyzed by Open-TOSCA to generate an imperative plan, which is then used to operate the applications and their cloud resources. Like in our trans-cloud approach, the description of locations based on policies offers some flexibility, because the integration of the topology and the used providers is limited to a few lines for the policies' configuration. However, in [65] the plans are coupled to the Node Types and used services because this information is used to generate the custom imperative plans. Furthermore, they are not integrated into the ecosystem yet. In contrast, thanks to its extended agnostic API, our trans-cloud solution uses Brooklyn's engine to manage the orchestration of applications using declarative plans.

In [104], Rafique et al. present a middleware platform for the distribution and management of hybrid-cloud applications which enables, by means of a uniform API, the portability over multiple services, such as data storage, asynchronous task execution or interoperability, between PaaS platforms. The authors of [104] evaluate their middleware using a multi-tenant SaaS application, for which they check the overhead and the performance impact of its deployment on different providers. In all these cases, they need to reimplement the portability driver for the desired platform. However, in our approach, the only change required to target different providers is to change the location in the corresponding policies. Rafique et al.'s middleware can handle PaaS platforms, and although it also supports, indirectly, IaaS clouds through the use of cloud-enabling middleware, our trans-cloud solution offers a combination of services to manage directly both levels, IaaS and PaaS, under a unique API. The authors of [104] do not define a comprehensive and normalized topology, although they already abstract and integrate services of the data-layer, such as NoSQL datastores and BLOB storage.

**SAMOS.** In [48], Fang et al. propose SAMOS, an ontology capable of modeling cloud services regardless of the kind of service, vendor, and abstraction level (IaaS, PaaS or SaaS) allowing both horizontal and vertical interoperability between levels. The ontology provides a comprehensive modeling for operations, datatypes, and services. Moreover, functional and non-functional requirements such as technologies, service level agreements, offering vendors, or capabilities of the services can be detailed, what is very useful to address service discovery. Indeed, with these models the SAMOS tool can be used to reason and assist in the deployment and management of applications. Despite its wide coverage of services, the usability of their approach is limited by the complexity in the definition and use of the ontology classes for both cloud services and providers. For example, using application and provider models, the system can be used to find the best cloud target providers to distribute the application and define a plan to operate providers and orchestrate the application management. Although our solution can also infer deployment plans for the automatic distribution of applications, it is true that the ontological modeling presents certain advantages in this regard, mainly associated to its reasoning engine. However, in comparison with our solution, it is not easy in SAMOS to add new providers and cloud services due to the complexity of asserting new cloud operations as ontology classes. Furthermore, the SAMOS system does not use agnostic application topologies. Instead, they use a programmatic description that formalizes all the knowledge on the application and the required service operations, what can limit its portability, since they must be modified if the application is to be deployed over different providers.

### 3.2.5 Cloud-decoupled orchestrators

In this section we analyze approaches based on decoupled orchestrators, which allow us to separate applications' descriptions from their cloud management using models that are totally agnostic or that only present minimal dependencies with their environment.

**SeaClouds.** SeaClouds [17] is a cloud orchestrator that focuses on the deployment and management of complex multi-component applications over heterogeneous clouds. The approach is based on the concept of service orchestration, and is designed to fulfil the functional and non-functional requirements of applications. In addition, services can be deployed, replicated, and administered using standard harmonized APIs. As our proposal, it uses a custom Brooklyn as deployment engine, and uses a TOSCA-compliant topology to model the application's architecture and its functional and non-functional requirements. As Cloud4SOA and REMICS (see above), SeaClouds provides mechanisms for the matchmaking of cloud offerings, based on the requirements of a given application, and the deployment of applications across multiple clouds. Indeed, the SeaClouds initiative goes one step further since it includes the auditing of SLAs and it uses Brooklyn and MODAClouds' monitoring features to inspect application and services' status to detect violations on the restrictions on the quality of services. When failures to satisfy such SLAs are detected, SeaClouds initiates a reactive repairing process to find a better cloud context according to the performance requirements defined by users. As in our proposal, SeaClouds uses TOSCA for the description of applications and Brooklyn as agnostic deployment orchestrator. However, in our case, Brooklyn has been customized to manage both IaaS and PaaS services. Furthermore, while with trans-cloud we can move application components between abstraction levels just by modifying the corresponding location policy, SeaClouds does not support vertical portability. Moreover, in Seaclouds, applications' components have to be modeled in the topology for a concrete abstraction level, either IaaS or PaaS.

**The PaaSage project.** As SeaClouds, the PaaSage project can also match applications' requirements to platform features and assist in the deployment process by making recommendations that are then run by its orchestrator. Although PaaSage can only deal with PaaS resources, the main difference with our approach is that the application description is neither topology-oriented nor DSL-oriented. The developers of PaaSage propose the use of CAMEL [107] as application profiling language, based on model-driven languages with a strong DSL integration. Although it might seem that this implies an agnostivity loss, authors argue that it brings the language to the target domain, therefore improving its expressiveness and capability, for example, for a better generation and adaptation of application workflows with target services. In addition of agnosticity differences, our work goes one step further since we try to maximize portability by using an open standard to describe applications, regardless of the cloud resources integration.

**CoMe4ACloud.** The aim of the CoMe4ACloud project [23] is a generic and extensible solution for the autonomic management of Cloud services. This approach takes advantage of the most general concept of XaaS model (Anything-as-a-Service or Everything-as-a-Service [41, 24]), to address portability and interoperability on the entire cloud services stack, IaaS, PaaS, and SaaS, while our proposal does not integrate models for SaaS services. However, CoMe4ACloud just offers horizontal portability, and to allow a component to be run using a new kind of service it is necessary to introduce modifications on the topology. CoMe4ACloud proposes an extension of TOSCA to describe XaaS systems, so that deployments in public and on-premise cloud and third party services can be integrated in the topology and in the execution plans. As our trans-cloud solution, CoMe4ACloud uses YAML profiles, but it does not take advantage of location-based policies of the standard, nor any other mechanism to provide agnostic topologies and minimize change when applications require new providers to run.

**MODAClouds.** MODAClouds is presented in [4]. It follows a Model-Driven approach that extends the REMIC project. MODAClouds delivers an open-source IDE for the high-level design, cloud service selection, early prototyping, QoS assessment, semi-automatic code generation, and multiple cloud applications automatic deployment. Like SeaClouds, it monitors applications at runtime and detects error and non-functional-requirements violations. It integrates PaaS management, but it does not provide an agnosticity-level as our proposal.

### 3.2.6 Commercial orchestrators

All previous tools and proposals are from academia, however, there are several commercial solutions worth mentioning, and we can expect more and more powerful solutions in the near future. Flexiant[1] can govern infrastructure-based services from different providers. AppFormix[2] supports several kinds of resources. It can manage public and on-premise IaaS and PaaS services, bare-metal systems in private clouds, virtual machines in OpenStack, and containers in Kubernetes clusters. IBM Cloud Orchestrator[3] offers an orchestrator to manage hybrid clouds by integrating the IBM cloud environment, SoftLayer, which includes both IaaS and bare-metal services, and Bluemix PaaS

[1]Flexiant: `https://www.flexiant.com/`.

[2]AppFormix: `https://www.juniper.net/us/en/products-services/application-management-orchestration/appformix`.

[3]IBM Cloud Orchestrator: `https://www.ibm.com/ie-en/marketplace/deployment-automation`.

services, with OpenStack- and CloudFoundry-based solutions. Morpheusdata[1] supports several IaaS providers, VMWare-based solutions, and container clusters. It is oriented to continuous delivery management, assisting to create deployment pipelines and providing monitoring and analytics features. Rigscale[2] provides and extensible API based on plugins to reach both IaaS and PaaS services via HTTP. All these solutions provide GUI-based dashboards to build applications and select the providers where to run them. Then, an orchestrator can provision resources, deploy applications, and execute policies to carry out workload adaptations. However, this abstraction hides the application topology, which cannot be manipulated by users. Only Rightscale offers an application description, provided using a DSL, to configure needed plugins to deploy and govern applications. However, it does not provide a full description of applications' components. As a result, these solutions allow users to deal with heterogeneous providers, but it just results in moving the cloud lock-in problem to a different level: users would not be able to extract the information concerning their applications if they needed to use other providers.

**Terraform.** Terraform is currently one of the most popular orchestrators. It provides a flexible abstraction of cloud resources, providers, and applications. Its models allow to represent everything related to the deployment of applications, including physical hardware, virtual machines, containers, and cloud resources, by means of an enriched application description that has its own DSL (called *interpolations*). Although it mainly focuses on its IaaS offering (such as AWS, Google Cloud, SoftLayer, and software virtualization platforms such as vSphere), it also provides some support for PaaS services, including AWS BeansTalk, Heroku, and CloudFoundry. It provides mechanisms for the description of applications using a fully resource-based unified high-level syntax, instead of requiring operators to use independent and non-interoperable tools for each IaaS offering and service. Although this representation is adapted for PaaS services, the management of each of these PaaS services may require its own minor customization of its configuration files. Like our proposal, Terraform defines its own internal representation and infers the sequence of the steps to run the application. It supports cross-deployment and takes advantage of the different cloud abstraction levels (IaaS, PaaS, and SaaS). Unfortunately, it does not consider the unification of IaaS and PaaS modeling, and does not offer standard-based application descriptions.

Some commercial solutions have been envisioned like standards implementations. Alien4Cloud[3] offers a GUI for managing blueprints (TOSCA topologies) and monitoring the deployment phase. It provides a drag-and-drop visual editor that helps in the creation of node templates and relationships, which can then be composed for modeling application topologies. Node operations can be defined and implemented by using implementation artifacts, which are completely integrated in the application's deployment lifecycle. It manages several IaaS providers, such as AWS, GCP and OpenStack. It also

[1]Morpheusdata: `https://www.morpheusdata.com/`.
[2]Rightscale: `https://docs.rightscale.com/`.
[3]Alien4Cloud: `https://alien4cloud.github.io/`.

integrates BYON (Bring-your-own-nodes) locations and containers. Alien4Cloud may then be used to deploy an application using its TOSCA definition. As our trans-cloud solution, Alien4Cloud uses TOSCA on top of a generic API to hide the cloud management by means of the policy locations, improving the portability of applications. Alien4Cloud was one of the first to support the use of location policies. Indeed, as pointed out in [31], the topology descriptions used by our trans-cloud solution are compatible with Alien4Cloud's in its version 1.3, since Alien4Cloud supported Brooklyn as an internal orchestrator. Despite of this, Alien4Cloud does not support agnostic node type definitions to be deployed on IaaS and PaaS, and therefore some changes are necessary when components are moved between abstraction levels.

As pointed out in [25, 31], Brooklyn,[1] developed as the first CAMP implementation, also supports the TOSCA standard. In this work, Brooklyn has been extended to support the agnostic cloud management envisioned by trans-cloud.

### 3.2.7 Applications and platforms modeling

The modeling of applications and cloud resources is key to deal with heterogeneity issues. Knowledge regarding applications and cloud resources is key to perform matchmaking and to automatize and orchestrate the needed operations to deploy applications on vendors' services [49]. There are already some standards which provide complete modeling for these aspects. However, after their definition, these specifications have still to become supported by engines or orchestrators that can process the modeled applications and interact with cloud resources to deploy and manage the applications, what is not always straightforward due to differences between the modeling goals and the capability of the orchestrators. As a result, current solutions use their own formal or ad-hoc and non-standardized applications' descriptions, they use different concepts, granularity and capabilities, increasing more the heterogeneity and complicating the portability between solutions.

In [105], Rabanahu et al. present SCALES, an abstraction-driven approach to address cloud application portability. The approach is based on an abstract language to describe applications' behaviors rather than their implementations. They use an agnostic DSL with which to provide full-detailed specifications of applications, including information on components, relationships, used and required services, technological requirements, etc. A model-transformation-based procedure is applied to this specification to translate generic descriptions to software components specific of a required provider. This approach deals with the utilization of vendors' services, what allows developers not to worry about the specific requirements of vendors' APIs and technologies. However, descriptions and transformations are focused on the application's performance, and application-deployable artifacts management, such as packaging mechanisms, are not taken into account. Like in our approach, the use of an agnostic DSL allows users to model their applications in an abstract way, and postpone the selection of target providers. Both IaaS and PaaS abstraction levels are supported, including AWS EC2

[1]Brooklyn: `https://brooklyn.apache.org/learnmore/theory.html`.

and Google App Engine. However, this solution does not support cross-orchestration, since all modules must use the services of the same provider.

AWS CloudFormation[1] allows to describe the composition of AWS resources in a reusable template. AWS CloudFormation allows to describe applications and fully detailed resources, such as zones, kinds, and some custom configurations such as security and networking. These resources can be specified using either a programming language or a simple text file, which is then used both to model and provision, in an automated and secure way. Although services from other providers may be referenced and used, CloudFormation can only manage IaaS AWS resources.

In a similar approach, Nguyen et al. [87] propose Blueprints as abstract descriptions of cross-layer services, XaaS, which can reach IaaS, PaaS and even SaaS. The approach provides a vendor-neutral blueprint templating mechanism in which users can specify the required services, and QoS and policy profiles. QoS profiles can be used to describe QoS characteristics of cloud resources. Moreover, the model is flexible and it can be extended to use concrete functionalities and add-ons of a concrete provider, something similar to what jClouds's API offers. Like our proposal, this solution provides an agnostic management of cloud resources regardless of the abstraction level, but it does not include an orchestrator to operate final providers. They do not provide an agnostic decoupling between application descriptions and the used resources.

Other projects and initiatives have developed their own descriptions. For example, Ubuntu Juju[2] is an orchestrator that includes its own application modeling language to describe applications and the management of resources. Other platforms use well-known solutions and include them in open platforms, such as Heat,[3] which adds support for CloudFormation to OpenStack. Although these solutions are quite popular, none of them provide an agnostic management of cloud resources.

## 3.3 Migration

We can found different ways of understanding the term *migration* in the literature depending on what kind of cloud services are involved and how they are managed (see, e.g., [120, 55]): VM migration, migration of legacy applications, and the migration of application components. The work proposed here clearly falls in this last group. VM migration is clearly out of the scope of this thesis, since on this topic most works typically focus on memory-to-disc data transfer, optimization of CPU, network or energy consumption, and similar low-level issues. We discuss in what follows several works on the other two topics, since they are all related to portability and interoperability issues.

[1]AWS CloudFormation: https://aws.amazon.com/es/cloudformation/.
[2]Ubuntu Juju: https://jaas.ai/.
[3]OpenStack HEAT: https://wiki.openstack.org/wiki/Heat.

### 3.3.1 Migration of legacy applications

Most of the work on migration verse on the *migration of legacy applications to the cloud* [55], where entire applications or some parts of them are moved to a cloud environment in order to take advantage of the cloud features, such as elasticity and scalability, payment strategies, or on-demand provisioning of services [5]. Migration processes and techniques are classified in the literature in accordance to the usage of cloud resources and what and how many parts of the applications are moved to the cloud. For example, in [2], migration types are classified as (i) replacement, (ii) partial migration, (iii) whole-stack migration, and (iv) cloudify. These categories go from the replacement of one legacy component by services in the cloud to the refactoring of entire applications to be cloud-compliant. Similar to this, in [73], Kratzke analyzes migration according to the application maturity to be run in the cloud [74], identifying what kind of configuration is needed to move an application or some of its components to the cloud.

In any of these cases, migration operations require some process to transform applications so that they become cloud-compliant. Several authors have analyzed the challenges of moving legacy applications to the cloud, although typically using cloud patterns to adapt applications to the restrictions of the cloud. Moreover, some of them report on lessons learnt and formal mechanisms to adapt application architectures to cloud environments (see, e.g., [62, 51]). For example, in [2], Andrikopoulus et al. offer a complete guideline to develop and adapt applications to the cloud. They analyze the impact of adapting application layers to different kinds of cloud services, in both IaaS and PaaS. Moreover, they propose some design patterns to facilitate the migration of legacy applications and even the development of new ones.

Other works present manual migration processes (see, e.g., [115, 114]), where they do not only report technical problems regarding the adaptation of application architectures to cloud environments, but they also describe the effort needed to adapt the delivery processes to the cloud model and in the usage of cloud APIs and clients. In this line, in [114], Tran et al. propose a novel formalization of the learned lessons by means of a taxonomy for the tasks performed during the migration, with the goal of helping ongoing migrations. This taxonomy can help to analyze legacy applications and the target cloud to identify the needed transformations to adapt the applications, and the operations to accomplish the migration process, such as how developing delivery services or how to scale the moving of components.

Other works go one step further and propose some kind of automatization in the adaptation of legacy applications. For example, in [12], Borgesa et al. propose Cloud Restriction Solver (CRS), a semi-automatic approach to adapt legacy applications' architectures to PaaS environments to decouple developers of cloud restrictions. For this, authors propose an assisted process with two steps. First, developers specify the restrictions of applications, and choose a target cloud environment. Then, with this information, CRS checks whether the target environment is suitable for the application, and it identifies the pieces of code that violate the restrictions on the chosen PaaS services. In a second phase, once the violations have been found, some refactoring is recommended to modify the code, and concrete cloud services can be suggested if it is necessary to re-implement the affected parts of the application. In this same line, in [50], Frey et al. present CloudMig, a model-driven approach to assist in the migration process. It allows to model applications and evaluate their profile in concrete IaaS and PaaS cloud environments to identify inconsistencies.

Besides the migration capabilities of these solutions, developers can evaluate the needed effort to migrate an application to a concrete environment, what may help in the decision of how to distribute an application in the cloud. Moreover, there are several frameworks to assist in the decision of which cloud providers to use (see, e.g., [119, 54, 53]). However, the choice for the migration of legacy applications is not easy yet, since it is not only necessary to find the cloud resources that best fit our needs, but it is also necessary to adapt delivery processes to deploy them into de cloud.

In summary, migration assistants and design patterns try to provide users with common techniques to move applications to the cloud, facilitating the matchmaking with cloud resources and hiding the cloud complexity as much as possible. It allows developers to decouple the model and the environment, identify the parts of the legacy system which present problems for the cloud migration, and then propose solutions to manage them in order to ensure the movement to the cloud. This is exactly the goal of solutions described in Section 3.2. Common APIs such as jClouds, COAPS, Nucleus, etc., offer a way to interact with providers during migration processes, as described in [60, 67], minimizing the impact of changes and providing some capability to react to unforeseen problems and reach new providers if needed. Standards like TOSCA, OCCI or CIMI allow to describe applications and adapt to cloud requirements [9], and apply patterns to facilitate the migration [88, 2]. Orchestrators such as Brooklyn, Terraform, SeaClouds, Alien4Cloud, and CLOUD4SOA would be the best choices for accomplishing the migration. Developers can model every application part and delegate the management of the application's lifecycle and cloud resources to the engine, minimizing the impact on applications. However, these solutions just assist on the deployment and maintenance of applications, but they do not carry out the adaptation and refactoring process needed to make applications cloud-compliant as previously described. Moreover, a common difference between our proposal and some of these solutions, such as SeaClouds, CLOUD4SOA, and [53], is that they include some kind of analysis of application requirements and cloud capabilities analysis to apply a matchmaking process to find the most suitable distribution of the applications. However, this is out of the scope of this thesis, since it is focused on providing a unification of the management of different service levels and in the task of the migration itself and the problem related with it, such as the optimization of resources during the migration and avoid wrong application status and unexpected behavior.

In this line, CMotion [11] proposes an agnostic and holistic migration process to move legacy application components to different providers. It provides an application modeling to describe applications' components and relations, containing all the needed application artifacts. Then, the framework operates them to deploy the application on different kinds of services, including both IaaS and PaaS. The idea behind CMotion is to use adapters to transform application artifacts to be compatible with a concrete technology. For example, let us suppose a Java application has to be migrated to AWS Beanstalk. This would require specific configuration or adaptation of artifacts, what is also known as vendor lock-in effect. Then, CMotion uses adapters to automatize these transformations, making transparent the adaptation of applications for a concrete cloud, hiding the vendor's complexity. This follows the same principle that other works described in previous sections. In summary, CMotion expedites the legacy migration process, reducing the effort needed to adapt the applications to specific providers. It supports Java, SQL-based dialects, and some on-premise infrastructure. However, creating adapters is a difficult task, since they require significant intrusive operations to enable new artifacts, what is an important challenge [67].

### 3.3.2 Runtime migration

Runtime migration is the capability to operate on a running application to move one or more of its components to a new environment, minimizing the impact on the application. All the works discussed in the previous sections offer mechanisms to move an application to the cloud, they allow to describe, transform, refactor and adapt applications' architectures, technologies and delivery systems. However, only some of the reviewed works support the migration of components of live applications. In fact, only a few works have attempted the runtime reconfiguration of applications.

SeaClouds (see Section 3.2.5) offers a very novel approach to move one Java application component. As our trans-cloud migration, users only need to point to a new provider and request to start the migration process. However, the migration can only happen in an abstraction level, between IaaS or PaaS providers, whereas our trans-cloud solution aims to agnostic levels management, allowing the transitions between different kinds of services. Moreover, the kind of migration supported by SeaClouds has a high impact on the application performance, practically stopping the application, and operating each component one by one. Our works on trans-cloud present an evolution of the migration techniques to, finally, identify the minimum part of the application that needs to be operated to carry out the migration task and operate on nodes as soon as possible.

In [47], Erbel et al. present an OCCI-based model and an engine to apply modifications to application topologies at runtime. Starting from a running application, whose topology is modeled using an extended-OCCI model, they propose to modify it according to a new target status specification (which is also modeled using OCCI). Then, developers can use the updated application description to request an application reconfiguration. The runtime adaptation process extracts the current running application model and compares it against the new application specifications, and then resources and relations are analyzed in order to define the adaptation plan. The proposal by Erbel et al. consists of three phases to operate the application to reach the new configuration: *(i)* deprovisioning, *(ii)* updating, and *(iii)* provisioning. The first one allows the decommission of resources that will not be used in the new configuration. In the second phase, resources are updated. Finally, during the last phase, new cloud resources are provisioned and components are redeployed.

The approach by Erbel et al. shares some features with trans-cloud runtime migration, such as the capability to delimit what application parts need to be modified to reach the new configuration, information that is then used to minimizing the needed operations. However, our trans-cloud solution presents a different management of resources' lifecycles that has a significant impact on the migration performance. The OCCI-based solution deprovisions resources at the beginning of the process, so the depending application's components have to also be stopped, to avoid failures. Only when the new resources have been provisioned, the stopped dependencies can be restarted. As described in [32], this means that some application parts will be stopped during the most timewise-expensive and error-prone operations, resulting in big delays to the migration process in which the migrated application does not provide any service. Our trans-cloud approach starts the migration by provisioning the new resources in background. Once this task is finished, components' connections are re-routed to use new resources. This means that downtimes only happen during the reconnection phase, what has a significant improvement on the process performance. Finally, when the application is back to normal operation, the old resources are also released in background tasks, without affecting the running application. Moreover, the solution proposed in [47] does not support either multi-cloud, component-wise migration of applications, nor agnostic level mechanisms, what is one of the key features of our trans-cloud approach.

Terraform also allows some kind of runtime reconfiguration. As described in Section 3.2.6, this orchestrator has its own models to describe applications and cloud resources. Based on these descriptions, Terraform can infer declarative plans to provision cloud resources and deploy application components. Thus, like [47], Terraform allows users to modify these application descriptions and request them for a running reconfiguration. Then, Terraform compares the application models with the current one to find differences and trace a plan to operate the application to reach the target configuration. Like our trans-cloud solution, Terraform tries to identify the minimum application topology region the migration process needs to operate on, to improve the performance of the process. However, they do not provide either agnostic-level migration nor a standardized approach. Moreover, like [47], Terraform deprovisions resources at the beginning of the process, whereas our proposal deletes old resources at the end of the migration when they are no longer useful. Despite these limitations, Terraform goes one step further on its reconfiguration capabilities, since it allows the application topology structure to be updated and modified, something that none of the works described above, nor our proposal, can perform, since they can modify the target cloud resources, but not the number of components or their relations. The Terraform's reconfiguration mechanism allows to add new components, to remove old ones, and can also manage the connections between them.

In [43], Durán and Salaün specify a protocol to automatize reconfiguration tasks. Like Terraform, this solution is based on an orchestrator, called Cloud Manager (CM), which receives a migration request, and uses an application model to trace the needed operations to perform the reconfiguration. However, it carries out a resource management similar to the one of the previous solutions, removing cloud resources in early stages of the protocol. Moreover, it only supports changes on IaaS, and the application profile is not based on any standard. The protocol proposed in [43] is however robust and fault-tolerant, something that none of previous solutions can perform. The reconfiguration process is not centralized in the CM orchestrator, but the components interact between them to ensure that the connections are properly managed, stopped and reestablished, during the removing, provisioning, and restarting operations. This approach allows the agile reaction to failures during the reconfiguration process, and fixing them by applying the needed operations.

In this context, in [14], Boyer et al. also provide a centralized robust configuration algorithm to reconfigure applications. Although this solution is quite similar to the one in [43], it does not allow to modify the configuration of cloud resources, as our trans-cloud solution does, since it is based on a centralized orchestrator, that only can manage the resources inside a unique virtual machine, where the application components are running.

One of the things our proposal has in common with the mentioned solutions is all of them propose their own mechanisms to manage the applications' lifecycles and operate them. This is key to allows manage multi-cloud applications and synchronize the requested operations during a runtime migrations process. In fact. decouple the application and its management of the clouds is one of the goals of this work. Cloud-native applications[78] have been gaining importance during last years, since they allow to take advantage of the cloud paradigm capabilities by applying different technologies, architectures, and patters [111]. Indeed, several of the work mentioned in Section 3.3.1, such as [51, 2], discuss how to transform legacy systems to cloud-native applications during the migration to the cloud. Although this kind of applications presents several advantages, such as the transparent usage of scalability, they would increase the coupling with the cloud, not only for the definition of the application itself, such as for the usage of add-ons, but the management of application lifecycle, for example, the pipelines for deployment and the definition of operational tasks. As result, as mentioned it would lock the management of the application to a concrete platform, limiting the capability to react to changes and move components if needed.

## 3.4 Self-Healing

In the last years the interest on self-healing in the cloud has been growing in both academy and industry. This kind of systems were characterized by Dai et al. in [35] as follows: *"systems designed to be self-healing are able to heal themselves at runtime in response to changing environmental or operational circumstances"*. In this section, we analyze different alternatives for self-healing proposed from both the infrastructure and the application's points of views. We close this section describing mechanisms to achieve self-healing in public clouds.

### 3.4.1 Architecture and infrastructure

Several works propose architecture-oriented solutions to include self-healing mechanisms into the cloud infrastructure.

Dai et al. were the first to formalize the terms self-diagnosed and self-healing in the cloud computing context [35]. They propose a system based on consequence-oriented diagnosis in which failures effects are diagnosed (predicted) from the failures in the system (symptoms). A self-diagnostic system is able to detect errors and determine their severity level. Then, according to these parameters, the self-healing system can decide to apply different healing procedures for recovering. For example, minor errors would require a report providing the diagnostic and the consequences of current errors. Major or catastrophic failures would require actions with a bigger impact on the system, such as a full reboot. Moreover, using Naïve Bayes Classifiers, Dai et al. propose a system that can learn and predict errors before they occur, avoiding the degradation of the system in the event of failures. This is the principal difference with our proposal, which cannot predict system behaviors. However, Dai et al. do not clarify how the healing mechanisms can be implemented on IaaS and PaaS services, or how intrusive it would be in the application description to allow this infrastructure to register and to learn and analyze applications.

In this same line, in [56], Singh Gill et al. present RADAR. RADAR is a complete architecture that integrates cloud infrastructure with a comprehensive monitoring system to detect and predict failures in the resources, together with mechanisms for the execution of corresponding recovery processes. RADAR also includes mechanisms for the monitoring of the behavior, the performance, and workload of an application and its environment, detecting violations of QoS and SLA requirements. Thus, the self-configuring system adapts the resources to the application's requirements. RADAR can analyze the workload of applications to reconfigure them in case of need and minimize usage to optimize the costs. This is similar to what orchestrators such as SeaClouds (see Section 3.2.5) can do. However, it is necessary to detail the characteristics and restrictions of the application, which need to be updated when modifications of the application, such as the usage of new resources or the migration of some components, take place. This means that the automatic changes by the self-healing system and the manual updates of these restrictions need to be synchronized. The decoupling of these two parts in one of the main challenges addressed by our proposal.

In [112], Satck et al. present self-healing as a useful technique to achieve continuous availability, since it requires failures detection, diagnostic, and recovering to maintain working systems. They propose a distributed solution based on a master-slave architecture to enable flexibility and high availability for a cloud system. Low layers inspect the status of the associated local components, such as cloud resources, and gather and send their status (KeepAlive messages) using an asynchronous messaging system (e.g., RabbitMq). Then, the system status is received by a higher layer that analyzes this information to detect failures. If errors are diagnosed, then recovery operations are applied, such as destroying or reprovisioning VMs. This solution focuses on what its authors call self-organization and self-management (SOSM), but this approach can only manage information and process the resources from an infrastructure's point of view, it cannot operate to solve errors maintaining the integrity of applications. Their proposal is therefore limited to IaaS approaches.

These solutions typically come with evaluations in which cloud environments are simulated, and using different workloads to evaluate their efficiency, reliability, and resilience, and how recovery plans allow them to react to foreseen failures. However, they do not evaluate unforeseen errors in the infrastructure or application layers. We have done so with no difficulty, since we use our trans-cloud monitoring system to automatically trigger a recovery. Moreover, their system can only observe and detect failures once applications have been deployed and they are running. Conversely, our proposal covers the entire applications lifecycle, allowing to react and repair failures during the deployment process as well, improving the robustness and fault-tolerance of application management.

Apache Mesos [59] provides a scalable and efficient system to build large clusters and deploy distributed and scalable frameworks, such as Hadoop, Kafka or even Kubernetes, using a master-slave architecture. Master processes manage slave daemons running on each cluster node, and frameworks that run tasks on these slaves. Thus, a fault-tolerance system is critical to maintain the masters' health and reliability, since all the frameworks running depend on them. Mesos creates hot-standby masters managed with Zookeeper, which are ready to recover the status of the cluster's master when one of them fails. When a failure occurs, Mesos notifies the framework's scheduler to react to it. For example, Kafka could require to update a slave process to ensure its own fault-tolerance system. Like masters, slave resources are also constantly monitored, to detect and react to failures. In the latest available version of Mesos, some periodic health-check messages are broadcasted, and if failures are found, repairing policies are applied. The repairing available operations go from the restarting to the re-provisioning of infrastructure resources. Mesos goes one step further than previous solutions since it can manage foreseen and unforeseen failures in the infrastructure, like our proposal. Moreover, it presents mechanisms to notify infrastructure errors to the application process. This allows frameworks to become preventive systems, reacting to external failures and prevent errors, instead of being passive approaches that wait for infrastructure anomalies that will cause failures inside their own systems. Then, Mesos can react to errors in running resources and even applications, since it provides support for frameworks to carry out their own self-healing policies. However, this requires applications to be customized to adapt to Mesos' lifecycle.

Other solutions, such as the ones proposed in [117, 80], follow a less intrusive approach. Instead of improving the resource providers or managers and instrumentalize the resources to check and share their status, they apply a passive inspection based on log analysis. These solutions are not predictive and they can only react to errors that have already happened, like our proposal, to apply some recovery operations. However, logs analysis reduces the effort necessary for system analysis, since it does not require adding to applications or to infrastructure custom status producer or collector to have an overall view of the current system's status. Moreover, it can delegate on well-known systems, such as Splunk[1] or Fluentd,[2] to efficiently collect and index logs in distributed systems. These solutions present a different scope than our proposal, since they are designed to work with IaaS infrastructures, given that they have to inspect machines to extract logs traces. In fact, they are suitable to be applied to on-premise infrastructures, but it seems difficult to deal with a multi-cloud environment, since a centralized log collector would have to receive the log traces of the cross-cloud system, increasing the traffic network and having a significant impact on its cost.

### 3.4.2 Application self-healing

In this section we discuss solutions that focus on providing self-healing mechanisms at the application level.

In [1], Alhosban et al. argue that the generation of recovery plans at runtime is a challenge due to the lack of capabilities of systems for self adaptation. They propose a solution based on two techniques. The first one is what authors call *pre-recovery*, in which behavior, reliability and utility of each service are intercepted, stored, and analyzed to assess the likelihood of fault occurrences. It allows to predict failures and unexpected situations, and then generate plans to prevent and recover possible errors. The second technique is *post-recovery*. Before running, applications are analyzed to detect their failure points, and BPEL recovery plans are generated and stored. When failures happen, exception handlers capture the errors and their context and they apply the static plans. If the fault is not fixed, dynamic recovery plans are generated. These plans may vary from ignoring the error to replacing the service or creating a passive replication in which a replication of the failing component is created and its connections are moved to the new copy step by step, until the old component can be stopped and released. The application of these plans is evaluated depending on the degree of criticality and the priority of the impacted parts of the system. As a result, the system can self-repair different situations of running applications, like our proposal. However, this solution presents a better capability to predict and identify the different kinds of errors to evaluate the impact of the recovery process and run more or less intrusive operations according to the need of the application. It can even use predefined static plans, something that our trans-cloud self-healing mechanisms do not support. On the other hand, our proposal is able to repair errors during the deployment of application services, what is an error-prone phase, since it is when cloud resources are provisioned. Moreover, it presents level-agnostic recovery mechanisms, to deal with failures in both IaaS and PaaS, whereas the proposal by Alhosban et al. can only handle IaaS environments.

In [79], Li et al. propose a framework to decouple the failures management of infrastructure and application layers. When an application is deployed, their component and resources are registered in a Health Manager, which decides the necessary monitoring rules to configure the monitor mechanisms. According to these rules, the Health Manager receives the notifications regarding applications and their status, using this information

[1]Splunk: `https://www.splunk.com/`.

[2]Fluentd: `https://www.fluentd.org/`.

to find errors. When failures are detected, the Recovery Service is in charge of finding and running a recovery plan. Following a general scheme, a solution for OpenStack is provided, which uses a well-known enterprise monitor product to analyze and notify the status of applications and their components. This architecture is similar to our proposal in which the orchestrator is in charge of deploying and managing applications, what allows us to observe the status of applications by means of our trans-cloud monitoring mechanisms. Then, this information is processed by the orchestrator, and, if failures are detected, a recovery plan is requested to the Analyzer Manager, which then sends the needed repairing operations to the orchestrator. Taking advantage of the trans-cloud's monitoring capabilities, our proposal can observe the earliest stages of the applications' lifecycle, such as provisioning, what allows to detect and then recover from failures even before applications are completely deployed. Again, this is one of the key differences with the proposal in [79]. Moreover, the proposed solution for OpenStack focuses on management and recovery in IaaS, without considering PaaS, whereas our trans-cloud solution covers both of them.

In [83], Magalhães et al. focus on self-healing in the web-application domain. They present SHõWA, a framework that can detect performance anomalies in web-based applications and execute operations to repair them. Aspect-Oriented-Programming-based sensors are added to the applications to intercept transactions and collect information at two different granularity levels. The first one is what authors call *user-transaction*, and it offers measures of the response times of servers and information on the status of resources, memory, open files, and current threads. The second one is the *profiling*, and it offers an abstract view of the behavior of applications, including a tracing profile of transaction call-paths, transaction chaining, and interactions between the application components. This is similar to the information provided in AppDynamics,[1] where it is called Business Transactions. Then, this information is collected and used to analyze and find workload variations and performance anomalies in order to detect both failures and performance-faulty scenarios, in which application components can provide correct outputs but with a low performance. The detection and reparation of the performance degradation cannot only prevent failures, but it can improves the elasticity of the system. This solution can also be applied to other architectures, such as micro-services-oriented applications, since transactions can be captured with non-intrusive mechanisms, as in our trans-cloud approach. Magalhães et al. use on-premise infrastructure based on Open Nebula in their test scenarios, but our guess is that the approach could also be applied to multi-cloud environments. However, even if monitoring mechanisms were added to the application, it seems complicated to extend the approach to PaaS because the use in PaaS of push-oriented solutions makes difficult to provide information on transactions. Moreover, the information on user transactions could be spread, since the PaaS infrastructure uses its own dynamic mechanisms to provision resources, e.g., container-based solutions. This makes, a priori, very complicated the provision of reliable information on user transactions.

The above mentioned approaches provide valuable solutions for self-healing that share

[1]AppDynamics: `https://docs.appdynamics.com/display/PRO45/Business+Transactions`.

some features with our trans-cloud solution. However, none of them is able to manage both IaaS and PaaS services as our trans-cloud solution does. None of them uses profiles to describe applications, what means that applications have to stick some specific structure or requirements set to allow self-healing solutions to be aware of applications and to extract the needed knowledge to manage and repair them, which can reduce the applications portability. Our trans-cloud approach requires the use of a TOSCA description, which allows avoiding vendor lock-in problems, and offers a transparent and agnostic monitoring mechanisms. Furthermore, it also allows covering the complete application lifecycle since our solution can apply its self-healing mechanisms at its earliest phases, e.g., during application deployment.

We wrap up this section with a final comment on the orchestrator-based solutions discussed in Section 3.2. For example, SeaClouds allows users to describe applications and their SLA and QoS requirements using TOSCA. Then, the orchestrator tries to discover the best distribution in cloud environment to deploy applications. Moreover, observability mechanisms inspect the performance of applications and resources, and if requirement violations or failures are detected, a new distribution is proposed to re-deploy the application to fit resources to requirements or solve errors. Similar mechanisms are proposed in MODAClouds, since applications are monitored to detect anomalies in their behavior. However, these solutions cannot be considered as self-repairing, since they require of some human intervention to decide and approve the changes proposed by the orchestrators. However, definitively they can deploy applications and monitor them to detect failures and then proposing some repairing operations.

The last versions of TOSCA standard have included specifications for the definition of reactive interfaces that can emit messages. Taking advantage of this, Node Types can include observability-oriented interfaces to share the current status of components and throw different kinds of messages when failures happen. Then, reactive and asynchronous systems may receive these notifications and they can react to failures and unexpected situations. As a result, imperative and declarative management plans can identify and operate affected topology parts to recover applications. These features are key to normalize self-healing mechanisms in a TOSCA-compliant orchestrator. However, this requires the instrumentalization of the application's components to provide the expected event behavior. Moreover, this technology has to be compatible with the different cloud resources where the application's components will run. The application's topology also needs to be modified, by including the Node Types' extensions to support the monitoring interfaces, which have to be compatible with the orchestrator's notification manager. The self-healing mechanisms proposed in this thesis try to deal with some of these challenges by means of a decoupled and modular architecture. Monitoring tasks are delegated to our trans-cloud's observability infrastructure, which is able to inspect resources in both IaaS and PaaS environments. Moreover, the orchestrator is in charge of inspecting the application's status, and if failures are detected, a recovery plan is requested via HTTP to an external and agnostic manager, which only needs the agnostic application model and the current status to plan the needed repairing operations. These instructions are then sent to the orchestrator, which applies them to recover the application. As conclusion, our proposal allows to define agnostic applications which do not need to be aware of the adaptation facilities to take advantage of the self-healing features.

### 3.4.3 Commercial solutions

Some of the current commercial clouds include mechanisms to detect failures in their hardware, such as issues related to the network, disk or solid storage, and memory. The different platforms use self-healing solutions to solve these problems, keeping users unaware of these problems, whose systems should not be affected. Furthermore, as already mentioned, during recent years public clouds have incorporated self-healing mechanisms to their public catalogs, what allows users to incorporate self-healing techniques to monitor and repair their own applications running on such cloud platforms.

AWS offers several recovering mechanisms. It allows to set up CloudWatch[1] alarms based on a set of predefined EC2 instance metrics. With this mechanism active, users are notified when events happen, or they can directly trigger recovery plans to repair applications. However, the operations that can be performed are limited to stopping, terminating, and rebooting the corresponding machines. Moreover, only specific kinds of instances can be repaired, and they can only be operated through the AWS OpsWorks console, what limits the possibility of external systems, like orchestrators or delivery tools, to take advantage of such self-healing features. A new kind of self-repairing mechanism, called *Host Recovery*, was recently announced. It allows Amazon EC2 systems to automatically restart instances on new hosts when unexpected hardware failures happen. However, this functionality is only available for a concrete kind of machines called Dedicated Hosts.

Azure uses health checks to detect when instances fail, and then can apply recovery operations. Moreover, Azure does not only react to failures, errors are stored and used to learn about the system's behavior. Then, heuristics can be applied to predict imminent failures. This allows users to adopt more proactive approaches to deal with unforeseen failures and repair them even before happening. Furthermore, diagnosis tools can also help users to analyze the behavior of their systems over time in order to detect system degradations and identify their root causes.

Google Cloud also uses health check mechanisms to determine if VM instances are responding as expected. They are used to maintain the high availability of instance groups, but they also allow detecting failures in VM instances and apply some repairing operations. A similar solution is offered by Alibaba Cloud.

## 3.5 Containers, the vendor lock-in's chimera

Container-based technologies are not supported by the current trans-cloud implementation. However, it is worth including them in the present discussion on related work.

[1]CloudWatch: `https://aws.amazon.com/es/cloudwatch/`.

Although containers have emerged very recently, it is nowadays one of the most popular technologies, and offer significant progress on some of the issues related to the vendor lock-in problem. In what follows, we discuss on how portability, migration, and self-healing are handled on both domains.

### 3.5.1 Portability of container-based applications

Containers allow isolating applications and their functional dependencies as portable pieces using a layered strategy over containers. They can be executed in any system supporting the container technology, for example, Docker, ensuring that both the application and its dependencies will be installed and properly configured. This is a real and effective breakthrough in terms of portability, since container technologies are supported practically by every computing resource or service. For example, Docker can be installed in bare-metal or virtual machines in on-premise and cloud infrastructure (IaaS); even some PaaS providers, such as Heroku, also support the deployment of Docker.

Moreover, its encapsulation and portability capabilities offer developers a reproducible execution environment, which is an effective opportunity to build, run and test applications as in production environments. As a result, the configuration effort to adapt applications to different environments is reduced, simplifying the definition and maintenance of applications' lifecycle procedures such as the deployment. Indeed, some of the portability solutions described in previous sections integrate support for Docker, as a way to maximize their portability — this is the case of jClouds and Brooklyn,[1] and OpenTOSCA.

A container represents a runnable unit, but normally cloud applications are not monolithic systems (see, e.g., [108, 93]). This perfectly matches with containers, since the idea behind this technology is to have each component, or a small set of them, isolated in each container, what can improve the maintenance and performance of applications [118]. For instance, it allows to maintain a relation between delivery tasks and the different parts of the system, simplifying the deployment and management processes. Moreover, the different parts of the system can be configured according to their own requirements. For example, some components or modules of the system would require more resources or scalability than other parts, or they may need some specific security configuration to provide their services.

As already said, container-based approaches might be a key step in the way to deal with portability problems. However, current solutions still present some limitations since the revisit problems regarding application's components orchestration, as some authors have already pointed out [22, 95, 103]. For example, Docker is able to run portable application components, as containers that can be configured with complete specifications, indicating, for example, open ports, volumes, and even commands to be run. However, it is not easy to operate the component or components that a container contains while it is being deployed. Moreover, although each container contains a part of the application, the information on which components are allocated in each container is

[1]Clocker is a Brooklyn distribution to work with containers (`http://www.clocker.io/`).

not explicit, which makes difficult to identify relations between containers to preserve the relations of the application's components. As Brogi et al. pointed out in [22], containers are *black-boxes*. As a result, managing applications as a whole to deploy, update, audit and configure them is still a challenge that is being currently studied.

Docker Compose[1] allows an application to be specified as a set of containers that are bounded by on-demand virtual network resources. However, although Docker Compose analyzes the application's descriptions and creates the containers that form an application respecting their dependencies, its orchestration capacity is limited, since the relations between the application's components can only be expressed in terms of containers' bindings, what makes difficult the orchestration of application's components.

Elastic container platforms solutions as Docker Swarm and Kubernetes offer solutions for the management of containers, facilitating the orchestration and clustering of applications. For example, Docker Swarm[2] allows Docker containers to be created and distributed along clusters of Docker hosts. Moreover, Swarm automatizes the replication and coordination of Docker containers and it offers network and load balancing capabilities to connect containers and expose services to make them available externally to the Swarm cluster.

Kubernetes also offers network and load balancing and exposition features, but it goes one step further than Swarm by offering a complete ecosystem for the orchestration, automated scheduling, and management of application containers. Kubernetes defines a set of new concepts, such as *Pod*, *Deployment* and *Jobs*, to group and run containers in different ways. For example, a pod is the minimal unit in Kubernetes, which represents one or more containers that are deployed and managed together. Deployments allow pods to be run and replicated. A deployment can be exposed as a network service by means of a *Service*, which also includes other features such as load balancing, and it can centralize the handling of network issues as network policies. Kubernetes defines a comprehensive model to describe different elements, which are written in files called *manifests*. Manifests are key for container portability: a manifest can contain one or more Kubernetes objects, and they can be deployed on any Kubernetes instance. Then, since Kubernetes is available in practically every cloud vendor and it can be installed over on-premise infrastructures, using for example OpenStack or Apache Mesos, applications expressed and adapted to Kubernetes can be deployed practically everywhere.

Nonetheless, Kubernetes does not provide a way to orchestrate the deployment of manifests, to maintain applications or other systems' relations or dependencies. This has to been handled by deployment tools such as Ansible or delivery pipelines such as Bamboo[3] and Jenkins[4].

Different proposals try to solve these issues. For example, Helm[5] defines a templating and dependencies system to generate manifests and package them as *Charts*. So each applications' components can create its own charts to public their manifests. Moreover,

[1]Docker Compose: `https://docs.docker.com/compose`.

[2]Docker Swarm: `https://docs.docker.com/engine/swarm`.

[3]Bamboo: `https://www.atlassian.com/software/bamboo`.

[4]Jenkins: `https://www.jenkins.io/`.

[5]Helm: `https://helm.sh/`.

charts can be related between them, so a chart can explicitly declare dependencies with others. However, these relations are managed in terms of inclusions, similar to dependencies between libraries: when a component has a dependency with another chart, the manifest of the target chart is added as part of the manifest of the source component. At the end, they only represent a potentially deployable application artifact that contains all needed manifests of an application's components, such as Deployments and Services.

Helm can run manifests on Kubernetes clusters, but charts gather their manifests and their dependencies, and they do not offer an application model. As a result, Helm cannot orchestrate manifests using an application's topology and the components' requirements.

As discussed in works like [22, 95, 103], Swarm, Kubernetes, and Docker Compose have some limitations, since in them software components are packaged as containers, and the extraction of runtime models of the topology of applications is cumbersome. Indeed, it can be even more difficult when manifests are involved, what challenges tasks such as the management of multi-component applications or multi-cloud runtime migration.

Some work is currently being carried out on these challenges. To deal with the lack of topology in container contexts, some authors have proposed solutions relying on the TOSCA standard. In [95], Pahl proposes a solution for the orchestration of complex applications' stacks based on TOSCA, to create topologies independent of service suppliers, cloud providers and agnostic topologies.

In [22], Brogi et al. envision a TOSCA-based combination with Docker like a method to enhance the capability to extract the complete description of complex applications in order to allow orchestrators to have a better management of multi-cloud environments. They define new TOSCA types, nodes, artifacts, and relationships to support the container domain in topology descriptions. All of these elements are supported by their system TOSKER, an engine that can process application topologies to orchestrate applications' components over Docker installations on different target providers. Moreover, using their own management protocol, TOSKER is able to infer the needed operations to generate declarative workflows.

Some other works try to use TOSCA models to improve Docker's orchestration capabilities [66, 116]. Indeed, the TOSCA standard itself has recently incorporated changes in the profiling for the integration of container-based technologies.

None of these solutions supports the orchestration of Kubernetes manifests, which currently is one of the most important Docker-based solutions. In 2019, the OAM standard[1] emerged to try to provide a solution for this. Like TOSCA, OAM defines concepts for components, application topology manifests and relations as part of its agnostic application model, which is based on a YAML profile. In OAM, every Kubernetes cluster is different, since they can use different object versions or even different object implementations, so the agnostic models allow developers to decouple their applications from these environment details. The agnostic models are transformed into concrete application descriptions according to the target Kubernetes cluster. OAM defines *Traits* in its model as the way to specify environment requirements, properties, and characteristics, what

[1]OAM: `https://oam.dev/`.

are used to generate final deployable resources. Although traits are also agnostic from the developers' point of view, they have to be included and supported by infrastructure operators, to ensure that applications can be deployed in a concrete environment.

### 3.5.2 Runtime migration of container-based applications

Although, as discussed in Section 3.3.2, currently there is no solution for the runtime migration of cloud applications, the use of containers may present some advances on this topic.

Most of the current container-based solutions manage containers as isolated pieces that are not part of an application's topology, so they can easily be destroyed and recreated in a Docker-based system. For example, orchestrators such as Kubernetes and Swarm allow the runtime movement of containers, although with some limitations.

Kubernetes clusters are built over a set of hosts, which can be virtualized on physical machines, containers (see Apache Mesos), or instances hosted on different cloud providers. Thus, a cluster installation can be built using different kinds of resources running across different systems in the same time than using on-premise infrastructure to running VM instances on different cloud providers.

Kubernetes allows hosts to be labeled, for example to separate resources according to their different purposes. For, example, hosts could be reserved and labeled depending on whether they are for internal operations of the cluster, for tests, or even for monitoring purposes, while other kinds of hosts could be in charge of running production applications. Then, these labels can be used by the manifests to indicate in which set of hosts of the clusters the Kubernetes objects have to be deployed. For example, a manifest can use these labels to deploy a job in the set of hosts for running production applications. Thanks to this capability, it is possible to use hosts allocated on different locations or even cloud providers. For example, a cluster could be formed by a set of nodes in AWS and others in Azure, or even in on-premise infrastructure. Using labels, users could indicate in their manifests in which cloud to deploy their applications, having something similar to a multi-cloud environment.

Moreover, while pods are running, Kubernetes can be requested to re-create them using different hosts to run the containers. Then, Kubernetes will create new pods and delete old ones progressively, while routing the pods requests and maintaining the load balancing. In this way, Kubernetes allows the migration of application components at runtime. However, the decision is limited to current cluster's hosts. For example, let us suppose a cluster composed by VMs in AWS and Softlayer. If we needed to move some deployed containers to a new provider, such as Google Cloud, some actions would be required. First, new hosts have to be provisioned and configured in Google. Then, these hosts should be added to the cluster, what requires some operational tasks, that has a significant impact on the performance of the cluster. Because of this, an important effort and non-agile process is needed to migrate containers to new providers in Kubernetes.

This possibility is exploited by Kratzke in [73]. This proposal provides a prototype tool to dynamically manage Swarm and Kubernetes clusters running in on-premise OpenStack-based infrastructure to resize them, adding new resources of public vendors, such as AWS EC2 and Google Compute Engine. Kratzke concludes that this process is a complex and challenging engineering task. In fact, what is even more difficult is to manually carry out the process due to the operational complexity of the tasks involved.

Moreover, it seems complicated to justify the need of adding hosts from a new vendor to a Kubernetes cluster due to functional requirements. Kubernetes always offers the same functionalities and the same isolated execution environment regardless of the hosts or vendors the containers are running, decoupling the cluster's behavior to specific functionality of a hosting system, such as clouds' add-ons or other specific features. However, non-functional-requirements could justify the use of a concrete set of nodes, for example, to take advantage of solid-storage.

In [103], Quint et al. propose the use of applications' agnostic topologies in the container scope, separating topology descriptions from the specification of the container-based platforms where they were to be deployed. For it, they define a complete DSL for applications' descriptions and propose an orchestrator to deploy applications on multi-cloud environments independently of the used elastic container platforms. Using this container-based framework, they provide support for the migration of runtime environments, that is, application migration between different providers in the container scope. However, they only consider the migration of entire applications, not being able to migrate individual components.

All previous solutions rely on the capabilities of current clusters such as Kubernetes and Swarm. In [46], Elliott et al. propose a totally agnostic solution based on an external orchestrator, that uses adapters to interact with Docker systems running on cloud providers and form their own orchestrated clusters. The approach offers a custom profiling to describe components inside containers. Once applications are running, the orchestrator allows to perform live migration requests to move components between hosts in the clusters. The migration protocol can move even stateful containers. However, despite the novel approach and interesting ideas, it does not present a real orchestration of containers, and no solution is provided for the addition of elements to clusters, what is a critical part, as described in [73].

### 3.5.3 Self-healing container-based applications

Current container environments present useful features to develop self-healing mechanisms. Containers contain a part of a system and the needed environment to run, and current technologies allow to rapidly recreate them. So, if there would be a mechanism to detect failures in containers, they could be easily recreated in order to fix such issues.

Current orchestrators such as Kubernetes and Swarm include observability hooks to check the container status and to operate on them if needed. For example, the manifest of a Kubernetes deployment allows to describe operations to check when the container deployment has been completed (readiness) and check periodically if the container is running correctly (liveness). Some retry or recreation policies can be specified if these indicators show errors. In this case, Kubernetes will be in charge of checking the containers status and if failures happen during building, or once the application is running, Kubernetes can remove the wrong container and create a new one.

Self-healing actions are part of manifests, what confirms the idea of manifests as self-contained units that have all the needed information for containers to be executed. However, as we have already seen, the lack of an application model to express applications as a whole has also a significant impact here, since it makes difficult to identify what containers or Kubernetes objects are connected and which are the relations between them. As a result, manifests have the needed information to recover failures in a concrete part of the system, but the impact on the rest of the application is not either measured nor managed, what would have more complex implications. Indeed, this is one of the questions we solved in the context of our work.

Once failures are detected, Kubernetes applies policies to easily carry out rolling-restart or even roll-out operations. However, manifests must include the concrete operations to check the containers status, what can go from an HTTP request to an applications' endpoint to call some element inside the container, such as a descriptor file. In summary, containers and applications' components must provide information on the way to be audited.

Orchestrators such as Kubernetes and Swarm can also react to failures in their own infrastructure. These orchestrators check worker node's liveness to monitor the cluster status. Thus, if one of the worker hosts fails, its container may be recreated in another worker node, or even restarted in the same worker after recovering from the failure in the worker. This is a key improvement in the resilience of container orchestrators since they can recover both application components and the environment where they are running. Health check mechanisms are used internally to maintain the Kubernetes clusters, which are offered to the clients in public clouds such as AWS and Google.

Orchestrators are not the only way to achieve recovery mechanisms, some standards also incorporate them in their specifications. As discussed in Section 3.4, the TOSCA standard provides ways to observe components status to react to failures and apply recovery processes. These features can also be used to manage simple containers as part of TOSCA application topologies. However, the problem is the same already mentioned above: the development of recovery plans is difficult and it has to be maintained together with the application's topology. The OAM standard is also developing generic and portable mechanisms to check the health of applications by means of Traits, but no stable solution has yet been added to the specification of the standard.

# Chapter 4. Conclusions and Future Work

## 4.1 Conclusions

In this thesis we have analyzed the vendor lock-in problems and its impact on the development and management of applications in the cloud. First, developers have to choose a cloud, or even a set of them, according to their applications' requirements, what means that they have to drive the design of their applications to the restrictions of the target clouds. Therefore, the heterogeneity of providers and the services they offer make things very difficult, and applications end up being developed in agreement to the restrictions of the concrete services chosen. Furthermore, the lack of services normalization drastically limits the portability between providers. Indeed, portability is a problem even between different abstraction levels inside the same platform, e.g., between EC2 and Elastic Beanstalk, the IaaS and PaaS offerings of AWS.

A tight relationship between applications and the cloud is necessary to run and optimize applications, but it is also a limitation in their operation. Indeed, changes can happen in applications and in the cloud environments they are deployed on, which may require changes in the used cloud resources to restore their functional and non-functional requirements. However, the effort and cost needed to carry out the modification of applications to adapt to the changes required to use different services or new providers can become very high. Sometimes developers cannot spend the needed resources to satisfactorily adapt their applications, and, as a result, users cannot react on time, having their systems locked in the cloud environments where they are running.

For all this, being agile and reacting to changes by using new cloud providers or new kinds of services is not always easy for developers. Moreover, it is even more complicated carrying on such changes when applications are already in production, because then it is necessary to move components and re-deploy them using new services while applications are in operation and offering their services.

To deal with these problems, in this thesis we have presented the trans-cloud approach as an abstraction for cloud management that hides the diversity and complexity of the cloud and offers to developers a unique way to interact with cloud providers and manage their applications. Trans-clouds extends the cross-cloud application deployment and management by supporting the portability and interoperability of application modules between different providers and at different levels. We have done so by developing of a common API to unify the management of IaaS and PaaS cloud services, making their use completely uniform. The trans-cloud infrastructure allows us to isolate the application development from cloud providers, allowing developers to develop portable applications, that can be deployed on the IaaS and PaaS services of different vendors without changing their applications, and without having to deal with the custom delivery process of each specific platform.

We have proposed a TOSCA-based agnostic modeling of applications and cloud services, which allows us to specify the characteristics and requirements of any system to be deployed in the cloud. The have shown how the standardised description of applications and cloud resources, and the homogenous service API, significantly reduce the portability and interoperability issues related to the vendor lock-in, facilitating the reusability of cloud services. Furthermore, having an agnostic model of our system greatly simplifies migration and decision change. Indeed, with our approach, each component may be deployed at one level or the other just by changing its location. It is worth noting that the proposed thesis project is not an implementation exercise on an existing deployment tool, but an innovative general approach to ease the cloud deployment of applications, enforcing the independence of both cloud providers and cloud models.

We have developed an operational prototype built on the well-established Apache Brooklyn tool in order to test our trans-cloud ideas. Brooklyn provides support for a large number of IaaS providers by means of jClouds and a complex engine to manage the applications' lifecycles in cloud contexts. Thanks to our efforts in integrating Cloud Foundry into Brooklyn, it now also provides access to PaaS Cloud-Foundry-based providers such as Pivotal Web Services and Bluemix. Developers can deploy their applications using IaaS and PaaS without modifying any of their delivery process configurations, any cloud integration or any aspect of their applications. They only have to modify the target locations in the TOSCA application descriptions, what means a significant reduction of the effort in the adaptation to reach different providers and abstractions levels.

Having an agnostic model of our system does not only greatly simplifies deployment, but it also simplifies the decision about what providers can be used to run an application, since the limitation and the complexity of using different providers is dramatically reduced. Indeed, the decoupling between the application definition and the target providers to use to deploy it that we have in our approach makes it possible to specify the service type and provider on which each component is to be deployed independently of the rest of the components just by specifying its target location. Then, the underlying management tool is in charge of the required provisioning and interoperation. The development and the results of our trans-cloud proposal are described in [25, 31].

Trans-cloud allows developers to react to change, by designing and deploying their applications using different vendors and abstraction levels. However, being agile and adapting systems become more difficult when applications are in production [29], In these scenarios, we could deploy the new version of the entire application while the old one is offering its services. Once all the components have been deployed and connected, the new application instance would start providing its services. Then, the old application may be released. This procedure minimizes downtimes, because the transition to the migrated application only involves the final switching to use the new services. However, this does not seem an optimal process, since it replicates the application entirely, which implies the duplication of all the resources used by the application, even application components that do not need to be migrated. To optimize its efficiency while minimizing its downtime, we have proposed a procedure in which we detect the minimum part of the application that needs to be handled to apply the required changes. Moreover, we propose a unique generic process to manage applications and cloud providers, thus facilitating the response if something changes. The key to this is that it does not need a custom delivery-adaptation process to operate each application and cloud resources when something changes.

As a solution to these problems, in this work we have developed a migration orchestrator in the context of our trans-cloud framework, taking advantage of its management of IaaS and PaaS. The orchestrator allows the concurrent migration of multiple components of an application, and it only needs the target locations of the components to be migrated, independently of the service level used both in the source and the target providers. Thus, the proposed component-wise migration orchestrator is vendor, technology, and service-level agnostic.

The migration process is fully automated, and is available on an extended version of Apache Brooklyn. As the rest of the trans-cloud framework, the orchestrator relies on a TOSCA specification of the application being managed. Indeed, this specification is the same one used for the initial deployment of the application. The only required external intervention to carry out the migration is just a migration request to initialize the process and the specification of the target locations of the components to be migrated.

As the analysis of the migration algorithm shows, it provides a migration mechanism that fulfils the described goals. The effort of the user to perform a migration operation is almost zero, just selecting the components to move and their target locations from the catalog of available services. Notice that there is, of course, an initial effort for the specification of the TOSCA description of the application.

In [30, 29, 32], we have presented the progression of our work, in consecutive steps enabling parallelization to generalize and optimize the migration process. Moreover, these works show that the application downtimes have been significantly reduced. Although we cannot say that it has been reduced to its minimum, we have argued that the algorithm was designed so that all operations that could be carried out in background, either before or after the downtime, are. Furthermore, the migration algorithm ensures the integrity of the application during its migration. At the same time, the generalization and optimization of the process reduces the impact on the application's performance, allowing developers to be agile and react to changes in the cloud, minimizing the needed effort in the operational tasks to reach new providers and kinds of services.

Neither the trans-cloud infrastructure nor its extension for migration were designed to provide robustness to the applications it manages. However, the system was further extended in [20] to provide self-healing mechanisms for trans-cloud applications with the additional goal of doing so while minimizing downtimes and instability. This work provides to users facilities that support the automated management of foreseen and unforeseen failures, in scenarios where applications components may be running on different platforms and abstraction levels (IaaS or PaaS).

The trans-cloud extension presented in [20] analyzes the status of the application along its entire lifecycle, checking if its progression is the expected one. Upon the occurrence of a failure, the trans-cloud infrastructure is able to recognize the error, and recover the needed information about the current application status. Then, using this information, the orchestrator is able to compose a plan with the needed operations to fix the process to accomplish the expected goals.

This ensures that the different phases of the application lifecycle will be executed with the expected results. For example, the deployment of an application always will end with all the components running in the cloud, independently of the failures that happen during the process, or if they are working on IaaS or PaaS. Moreover, the improvement in fault tolerance has a positive impact on the predictability and quality of systems, because it cannot only react to failures in the infrastructure, but it can also manage errors and unexpected situations related to the application itself. For example, developers would see how their systems react and they are operated to fix concrete failures, such as system breakdowns due to overloads, or even cyclic errors related to design problems, such as memory leaks in which the application will breakdown recurrently until the error is fixed.

All of the aforementioned characteristics of trans-cloud have been added to the prototypes that have been used to check our work and analyze its viability and performance. The implementation has been extensively tested, as you can see in the papers that support this thesis.

In conclusion, trans-cloud provides functionalities and capabilities to deal with many issues related with vendor lock-in, providing users robust and flexible mechanisms to deploy and manage their applications, react to changes, and ensure their systems' behavior in a heterogeneous cloud environment, decoupling them from vendor details and complexity. Therefore, this decoupling offers to developers a way to deal with QoS and SLA questions, and the flexibility of changing their applications to reach new providers if some improvements or optimizations of the non-functional requirements are needed.

The unification of the management of cloud providers seems the correct way to deal with vendor lock-in issues, what allows developers to get the most out of the cloud capabilities. As discussed, this is something that has already been studied and carried out in different proposals such as jClouds, Nucleus, etc., but each of them focuses on a concrete abstraction level. We believe that the most important innovation of this thesis is the unification of different abstraction levels to manage services and to describe applications. This is possible thanks to the unification of the lifecycle's of applications' components, what is the basis on which the work on migration and self-healing is built.

## 4.2 Future work

In this work, we have presented the trans-cloud concept and its implementation as an extension of Brooklyn. Indeed, there were several extensions: first to support the trans-cloud deployment of applications, and then to support their migration and the definition of robust processes. This progression is reflected in the papers that are part of the body of this thesis, since all of them built on the previous ones. However trans-cloud is still far from being finished, and much work remains ahead.

The first step of our plan is to analyze new providers in order to extend the supported PaaS services and technologies. As a consequence, the current model will be extended in order to integrate PaaS levels of new providers, such as Heroku and OpenShift. Due to the providers heterogeneity, the new providers have to be carefully analyzed in order to elaborate on how they should be added to our approach.

The current version is mainly focused on back-end components based on pure Java technologies. It would be interesting to support other languages, such as Python or Kotlin, or even going a step further and managing technologies such as PHP or C#.

We also plan to study the possibility of using the flexibility and scalability mechanisms available for PaaS to develop management policies to react to applications' events. It would allow users to decouple the definition of the flexibility of their systems from that of the cloud specification.

We have shown that the effort of changing location targets is almost zero if the proposed CAMP interface offers the location in its catalog. Of course, if we wanted to use a location not in the catalog, it should be added before using it. Its addition would require some effort, either by the owners of the interface or their clients. It is a limitation that we cannot avoid. As a result, some development is needed to add new providers. We plan to explore mechanisms to alleviate the required effort.

One of the main drawbacks of our proposal is that it requires a TOSCA specification of the application to be managed, which requires having a deep knowledge about the CAMP interface and TOSCA. A wizard for the development of topology specifications may greatly help users.

The possible improvements of our proposal are not only possible extensions, some processes of our system could also be enhanced. For example, one of the most important goals of this work has been to reduce downtimes as much as possible, since downtimes, however small, always have an impact on the performance and the behavior of applications. The last version of the migration orchestrator drastically reduces the downtimes during the process, however, different techniques could be used to further reduce them. For example, once new instances of the application's components are created, load balancers or proxies could be used to route the traffic of the connections from old components to the new ones, further reducing the downtimes. However, this may have a direct impact on the migration process, since it would require dealing with the management of these new traffic orchestrators.

Another limitation of our work is related to the mutability of the application topology. Our process allows the movement of the application's components between different providers and levels, but the orchestrator assumes that the topology of an application does not change. Components cannot be added to or deleted from the original topology. We plan to explore the possibility of handling topology change.

Our migration solution also has limitations The main one is that it can only migrate stateless components. The initial load of data and the transfer of data to and from the cloud is a hot research topic. Our solution assumes that no state needs to be transferred, what allows us to focus on the stop, re-connection, and start/re-start of components. As future work, we plan to study the migration of components with state. However, it does not seem easy since the management of statefull components requires ensuring data integrity. Data are normally very close to the application which produces and manages them, and therefore, the migration of the data would need to know the behavior of applications, and, depending on the nature of the data and the applications, different kinds of processes would be required. Indeed, downtimes can become unavoidable. Since the decoupling of data and applications do not seem to be a straightforward improvement, this hinders the creation of a generic process for migration of statefull components. We plan to analyze this kind of problems, and we also plan to study the migration of distributed cloud databases, such as Cassandra, and the mechanisms it already provides to support such operations.

The self-healing extension of trans-cloud also has some possible improvements. Regarding the architecture of the solution, the interaction between the management planner and the trans-cloud deployer can be greatly improved. The integration of two pre-existing components through an orchestrator allowed us to show the feasibility of the proposal, but a tighter integration would improve the response times. For example, the use of reactive technologies would simplify the orchestrator and their interaction, as the analysis of the current status system and their reaction to perform the plans. This change would possibly have a minimal improvement on the system's behavior, on the delay of the operations, and on the data recollection, but it would be useful to carry out future extensions of this module.

Possibly, one of the most interesting extensions of our proposal is on the robustness of the migration process. Currently, when an error occurs during a migration process, the task is aborted. Since components are provisioned during migration, and most errors related to the handling of cloud applications occur during the provisioning of resources, the current orchestrator has a high failure rate. We plan to extend the support for self-healing applications to the migration process.

Although not a perfect solution, container technologies present several improvements on portability and interoperability that allow to integrate the management of multiple providers. Moreover, these technologies present several mechanisms, for example, to collect information about container status, what can be useful to design robust systems. However, as described, these technologies have some limitations regarding topology descriptions, what limits the possibility to have flexible mechanisms to accomplish a stable and generic migration process. We plan to carefully analyze the use of container-based technologies to improve the possibilities of our work.

# References

[1] Amal Alhosban, Khayyam Hashmi, Zaki Malik, and Brahim Medjahed. Self-healing framework for cloud-based services. In *ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 1–7. IEEE, 2013.

[2] Vasilios Andrikopoulos, Tobias Binz, Frank Leymann, and Steve Strauch. How to adapt applications for the cloud environment. *Computing*, 95(6):493–535, 2013.

[3] Darko Androcec, Neven Vrcek, and Peep Kungas. Service-level interoperability issues of platform as a service. In *11th IEEE World Congress on Services (SERVICES)*, pages 349–356. IEEE, 2015.

[4] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, et al. MODAClouds: A model-driven approach for the design and execution of applications on multiple clouds. In *4th International Workshop on Modeling in Software Engineering (MISE)*, pages 50–56. IEEE, 2012.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[6] Dionysis Athanasopoulos, Miguel Barrientos, Leonardo Bartoloni, Antonio Brogi, Mattia Buccarella, Jose Carrasco, Javier Cubo, Francesco D'Andria, Elisabetta Di Nitto, Adrián Nieto, Marc Oriol, Ernesto Pimentel, and Simone Zenzaro. SeaClouds: Agile management of complex applications across multiple heterogeneous clouds. In *Projects Showcase - Workshop of Software Technologies: Applications and Foundations 2015 federation of conferences (STAF)*, pages 54–61. CEUR-WS.org, 2015.

[7] Nick Bassiliades, Moisis Symeonidis, Panagiotis Gouvas, Efstratios Kontopoulos, Georgios Meditskos, and Ioannis Vlahavas. PaaSport semantic model: An ontology for a platform-as-a-service semantically interoperable marketplace. *Data & Knowledge Engineering*, 113:81–115, 2018.

[8] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. OpenTOSCA–a runtime for TOSCA-

based cloud applications. In *11th International Conference on Service-Oriented Computing (ICSOC)*, pages 692–695. Springer, 2013.

[9] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Migration of enterprise applications to the cloud. *it Information Technology*, 56(3):106–111, 2014.

[10] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. TOSCA: Portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.

[11] Tobias Binz, Frank Leymann, and David Schumm. CMotion: A framework for migration of applications into and between clouds. In *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–4. IEEE, 2011.

[12] Marcos Borges, Erick Barros, and Paulo Henrique Maia. Cloud restriction solver: A refactoring-based approach to migrate applications to the cloud. *Information and Software Technology*, 95:346–365, 2018.

[13] Nour El Houda Bouzerzour, Souad Ghazouani, and Yahya Slimani. A survey on the service interoperability in cloud computing: Client-centric and provider-centric perspectives. *Software: Practice and Experience*, 50(7):1025–1060, 2020.

[14] Fabienne Boyer, Olivier Gruber, and Damien Pous. Robust reconfigurations of component assemblies. In *35th International Conference on Software Engineering, (ICSE'13)*, pages 13–22. IEEE, 2013.

[15] Uwe Breitenbücher, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger. Combining declarative and imperative cloud application provisioning based on TOSCA. In *2014 IEEE International Conference on Cloud Engineering*, pages 87–96. IEEE, 2014.

[16] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Fault-aware management protocols for multi-component applications. *Journal of Systems and Software*, 139:189–210, 2018.

[17] Antonio Brogi, Jose Carrasco, Javier Cubo, Francesco D'Andria, Elisabetta Di Nitto, Michele Guerriero, Diego Pérez, Ernesto Pimentel, and Jacopo Soldani. SeaClouds: An open reference architecture for multi-cloud governance. In *10th European Conference Software Architecture (ECSA)*, pages 334–338, 2016.

[18] Antonio Brogi, Jose Carrasco, Javier Cubo, Francesco D'Andria, Ahmad Ibrahim, Ernesto Pimentel, and Jacopo Soldani. EU project SeaClouds - adaptive management of service-based applications across multiple clouds. In *4th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 758–763. SciTePress, 2014.

[19] Antonio Brogi, Jose Carrasco, Javier Cubo, Elisabetta Di Nitto, Francisco Durán, Michela Fazzolari, Ahmad Ibrahim, Ernesto Pimentel, Jacopo Soldani, PengWei Wang, and Francesco D'Andria. Adaptive management of applications across multiple clouds: The SeaClouds approach. *CLEI Electronic Journal*, 18(1), 2015.

[20] Antonio Brogi, Jose Carrasco, Francisco Durán, Ernesto Pimentel, and Jacopo Soldani. Robust management of trans-cloud applications. In *12th IEEE International Conference on Cloud Computing (CLOUD)*, pages 219–223. IEEE, 2019.

[21] Antonio Brogi, Ahmad Ibrahim, Jacopo Soldani, José Carrasco, Javier Cubo, Ernesto Pimentel, and Francesco D'Andria. SeaClouds: A European project on seamless management of multi-cloud applications. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–4, 2014.

[22] Antonio Brogi, Luca Rinaldi, and Jacopo Soldani. TosKer: A synergy between TOSCA and docker for orchestrating multicomponent applications. *Software: Practice and Experience*, 48(11):2061–2079, 2018.

[23] Hugo Brunelière, Zakarea Alshara, Frederico Alvares, Jonathan Lejeune, and Thomas Ledoux. A model-based architecture for autonomic and heterogeneous cloud systems. In *8th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 201–212. SciTePress, 2018.

[24] Zhicheng Cai, Xiaoping Li, and Jatinder ND Gupta. Heuristics for provisioning services to workflows in XaaS clouds. *IEEE Transactions on Services Computing*, 9(2):250–263, 2016.

[25] Jose Carrasco, Javier Cubo, Francisco Durán, and Ernesto Pimentel. Bidimensional cross-cloud management with TOSCA and Brooklyn. In *9th IEEE International Conference on Cloud Computing (CLOUD)*, pages 951–955. IEEE, 2016.

[26] Jose Carrasco, Javier Cubo, and Ernesto Pimentel. Propuesta de metodología de despliegue de aplicaciones en nubes heterogéneas con TOSCA. In *19th Spanish Conference on Software Engineering and Databases (JISBD)*, pages 321–334. Sistedes, 2014.

[27] Jose Carrasco, Javier Cubo, and Ernesto Pimentel. Towards a flexible deployment of multi-cloud applications based on TOSCA and CAMP. In *Advances in Service-Oriented and Cloud Computing - Workshops of 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC), Revised selected papers*, pages 278–286. Springer, 2014.

[28] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. Towards a unified management of applications on heterogeneous clouds. In *Advances in Service-Oriented and Cloud Computing - Workshops of 5th European Conference on Service-Oriented and Cloud Computing (ESOCC), Revised Selected Papers*, volume 707, pages 233–246. Springer, 2016.

[29] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. Component migration in a trans-cloud environment. In *7th International Conference on Cloud Computing and Services Science (CLOSER), Revised Selected Papers*, pages 286–307. Springer, 2017.

[30] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. Runtime migration of applications in a trans-cloud environment. In *Adaptive Services-Oriented and Cloud Applications (ASOCA) - Workshops of 15th International Conference on Service-Oriented Computing (ICSOC)*, pages 55–66. Springer, 2018.

[31] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. Trans-cloud: CAMP/TOSCA-based bidimensional cross-cloud. *Computer Standards & Interfaces*, 58:167–179, 2018.

[32] Jose Carrasco, Francisco Durán, and Ernesto Pimentel. Live migration of trans-cloud applications. *Computer Standards & Interfaces*, 69:103392, 2020.

[33] Antonio Celesti, Francesco Tusa, Massimo Villari, and Antonio Puliafito. How to enhance cloud architectures to enable cross-federation. In *4th IEEE International Conference on Cloud Computing (CLOUD)*, pages 337–345. IEEE, 2010.

[34] David Cunha, Pedro Neves, and Pedro Sousa. A platform-as-a-service API aggregator. In *Advances in Information Systems and Technologies (WorldCIST'13)*, pages 807–818. Springer, 2013.

[35] Yuanshun Dai, Yanping Xiang, and Gewei Zhang. Self-healing and hybrid diagnosis in cloud computing. In *1st International Conference on Cloud Computing (CloudCom)*, pages 45–56. Springer, 2009.

[36] C. Davis. Realizing software reliability in the face of infrastructure instability. *IEEE Cloud Computing*, 4(5):34–40, 2017.

[37] Yuri Demchenko, Canh Ngo, Cees de Laat, Marc X. Makkes, and Rudolf J. Strijkers. Intercloud architecture framework for heterogeneous multi-provider cloud based infrastructure services provisioning. *International Journal of Next-Generation Computing*, 4(2):1–18, 2013.

[38] Beniamino Di Martino. Applications portability and services interoperability among multiple clouds. *IEEE Cloud Computing*, 1(1):74–77, 2014.

[39] DMTF. Interoperable clouds - A white paper from the open cloud standards incubator. Standard, DMTF, 2009.

[40] DMTF. Cloud infrastructure management interface (CIMI) - model and RESTful HTTP-based protocol - an interface for managing cloud infrastructure. Standard, DMTF, 2016. `https://www.dmtf.org/sites/default/files/standards/documents/DSP0263_2.0.0.pdf`.

[41] Yucong Duan, Guohua Fu, Nianjun Zhou, Xiaobing Sun, Nanjangud C Narendra, and Bo Hu. Everything as a service (XaaS) on the cloud: Origins, current and future trends. In *8th IEEE International Conference on Cloud Computing (CLOUD)*, pages 621–628. IEEE, 2015.

[42] Francisco Durán and Gwen Salaün. Robust reconfiguration of cloud applications. In *International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'14)*, pages 179–184. ACM, 2014.

[43] Francisco Durán and Gwen Salaün. Robust and reliable reconfiguration of cloud applications. *Journal of Systems and Software*, 122:524–537, 2016.

[44] Andy Edmonds, Thijs Metsch, Alexander Papaspyrou, and Alexis Richardson. Toward an open cloud standard. *IEEE Internet Computing*, 16(4):15–25, 2012.

[45] Yehia Elkhatib. Defining cross-cloud systems. *ArXiv e-prints*, 2016.

[46] David Elliott, Carlos Otero, Matthew Ridley, and Xavier Merino. A cloud-agnostic container orchestrator for improving interoperability. In *11th IEEE International Conference on Cloud Computing (CLOUD)*, pages 958–961. IEEE, 2018.

[47] Johannes Erbel, Fabian Korte, and Jens Grabowski. Comparison and runtime adaptation of cloud application topologies based on OCCI. In *8th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 517–525. SciTePress, 2018.

[48] Daren Fang, Xiaodong Liu, Imed Romdhani, and Claus Pahl. An approach to unified cloud service access, manipulation and dynamic orchestration via semantic cloud service operation specification framework. *Journal of Cloud Computing*, 4(1):1–20, 2015.

[49] Yong-Yi Fanjiang, Yang Syu, Shang-Pin Ma, and Jong-Yih Kuo. An overview and classification of service description approaches in automated service composition research. *IEEE Transactions on Services Computing*, 10(2):176–189, 2015.

[50] Sören Frey and Wilhelm Hasselbring. The CloudMIG approach: Model-based migration of software systems to cloud-optimized applications. *International Journal on Advances in Software*, 4(3 and 4):342–353, 2011.

[51] Sören Frey, Wilhelm Hasselbring, and Benjamin Schnoor. Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software: Evolution and Process*, 25(10):1089–1115, 2013.

[52] José María García, Octavio Martín-Díaz, Pablo Fernández, Carlos Müller, and Antonio Ruiz-Cortés. A flexible billing life cycle for cloud services using augmented customer agreements. *IEEE Access*, 9:44374–44389, 2021.

[53] Jesús García-Galán, Pablo Trinidad, Omer Farooq-Rana, and Antonio Ruiz-Cortés. Automated configuration support for infrastructure migration to the cloud. *Future Generation Computer Systems*, 55:200–212, 2016.

[54] Radhika Garg, Marc Heimgartner, and Burkhard Stiller. Decision support system for adoption of cloud-based services. In *6th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 71–82. SciTePress, 2016.

[55] Mahdi Fahmideh Gholami, Farhad Daneshgar, Graham Low, and Ghassan Beydoun. Cloud migration process—A survey, evaluation framework, and open challenges. *Journal of Systems and Software*, 120:31–69, 2016.

[56] Sukhpal Singh Gill, Inderveer Chana, Maninder Singh, and Rajkumar Buyya. Radar: Self-configuring and self-healing in resource management for enhancing quality of cloud services. *Concurrency and Computation: Practice and Experience*, 31(1), 2019.

[57] Fotis Gonidis, Iraklis Paraskakis, and Anthony James Howard Simons. A development framework enabling the design of service-based cloud applications. In *Advances in Service-Oriented and Cloud Computing - Workshops of 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC), Revised Selected Papers*, pages 139–152. Springer, 2014.

[58] Nikolay Grozev and Rajkumar Buyya. Inter-cloud architectures and application brokering: Taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390, 2014.

[59] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Howard Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 22–22. USENIX Association, 2011.

[60] Eman Hossny, Sherif Khattab, Fatma Omara, and Hesham Hassan. A case study for deploying applications on heterogeneous PaaS platforms. In *2013 International Conference on Cloud Computing and Big Data (CloudCom-Asia)*, pages 246–253. IEEE, 2013.

[61] IEEE - C/CCSC - Cloud Computing Standards Committee. The NIST definition of cloud computing. Technical report, 2011.

[62] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Cloud migration research: A systematic review. *IEEE Transactions on Cloud Computing*, 1(2):142–157, 2013.

[63] Eleni Kamateri, Nikolaos Loutas, Dimitris Zeginis, James Ahtes, Francesco D'Andria, Stefano Bocconi, Panagiotis Gouvas, Giannis Ledakis, Franco Ravagli,

Oleksandr Lobunets, and Konstantinos Tarabanis. Cloud4SOA: A semantic-interoperability PaaS solution for multi-cloud platform management and portability. In *2nd European Conference on Service-Oriented and Cloud Computing (ESOCC)*, pages 64–78, 2013.

[64] Kiranbir Kaur, Sandeep Sharma, and Karanjeet Singh Kahlon. Interoperability and portability approaches in inter-connected clouds: A review. *ACM Computing Surveys*, 50(4):49:1–49:40, 2017.

[65] Kálmán Képes, Uwe Breitenbücher, Markus Philipp Fischer, Frank Leymann, and Michael Zimmermann. Policy-aware provisioning plan generation for TOSCA-based applications. In *11th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2017)*, pages 142–149. Xpert Publishing Services, 2017.

[66] Kitti Klinbua and Wiwat Vatanawood. Translating TOSCA into docker-compose YAML file using ANTLR. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 145–148. Springer, 2017.

[67] Stefan Kolb, Jörg Lenhard, and Guido Wirtz. Application migration effort in the cloud. In *8th IEEE International Conference on Cloud Computing (CLOUD)*, pages 41–48. IEEE, 2015.

[68] Stefan Kolb and Cedric Röck. Unified cloud application management. In *12th IEEE World Congress on Services Computing (SERVICES)*, pages 1–8. IEEE, 2016.

[69] Stefan Kolb and Guido Wirtz. Towards application portability in platform as a service. In *8th IEEE International Symposium on Service Oriented System Engineering (SOSE)*, pages 218–229. IEEE, 2014.

[70] Stefan Kolb and Guido Wirtz. Data governance and semantic recommendation algorithms for cloud platform selection. In *10th IEEE International Conference on Cloud Computing (CLOUD)*, pages 664–671. IEEE, 2017.

[71] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. Winery–a modeling tool for TOSCA-based cloud applications. In *11th International Conference Service-Oriented Computing (ICSOC)*, pages 700–704. Springer, 2013.

[72] Fabian Korte, Stéphanie Challita, Faiez Zalila, Philippe Merle, and Jens Grabowski. Model-driven configuration management of cloud applications with OCCI. In *8th International Conference on Cloud Computing and Services Science (CLOSER)*. SciTePress, 2018.

[73] Nane Kratzke. About the complexity to transfer cloud applications at runtime and how container platforms can contribute? In *7th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 19–45. Springer, 2017.

[74] Nane Kratzke. A brief history of cloud application architectures. *Applied Sciences*, 8(8):1368, 2018.

[75] Kyriakos Kritikos and Dimitris Plexousakis. Multi-cloud application design through cloud service composition. In *8th IEEE International Conference on Cloud Computing (CLOUD)*, pages 686–693. IEEE, 2015.

[76] Priti Kumari and Parmeet Kaur. A survey of fault tolerance in cloud computing. *Journal of King Saud University - Computer and Information Sciences*, 2018.

[77] Grace A. Lewis. Role of standards in cloud-computing interoperability. In *46th Hawaii International Conference on System Sciences (HICSS)*, pages 1652–1661. IEEE, 2013.

[78] Frank Leymann, Uwe Breitenbücher, Sebastian Wagner, and Johannes Wettinger. Native cloud applications: why monolithic virtualization is not their foundation. In *6th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 16–40. Springer, 2016.

[79] Xinhui Li, Kai Li, Xudong Pang, and Yiping Wang. An orchestration based cloud auto-healing service framework. In *IEEE International Conference on Edge Computing, (EDGE)*, pages 190–193. IEEE, 2017.

[80] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102–111. ACM, 2016.

[81] Paul Lipton, Derek Palma, Matt Rutkowski, and Damian Andrew Tamburri. TOSCA solves big problems in the cloud and beyond! *IEEE Cloud Computing*, 5(2):37–47, 2018.

[82] Nikolaos Loutas, Vassilios Peristeras, Thanassis Bouras, Eleni Kamateri, Dimitrios Zeginis, and Konstantinos A. Tarabanis. Towards a reference architecture for semantically interoperable clouds. In *2nd International Conference on Cloud Computing (CloudCom)*, pages 143–150. IEEE, 2010.

[83] João Paulo Magalhães and Luís Moura Silva. A framework for self-healing and self-adaptation of cloud-hosted web-based applications. In *5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 555–564. IEEE, 2013.

[84] Tariq Mahmood, Bharath Balasubramanian, Mithuna Thottethodi, Sanjay G. Rao, and Kaustubh Joshi. ACCORD: Automated change coordination across independently administered cloud services. In *11th IEEE International Conference on Cloud Computing (CLOUD)*, pages 770–777. IEEE, 2018.

[85] Ibrahim Ejdayid A. Mansour, Reza Sahandi, Kendra M. L. Cooper, and Adrian Warman. Interoperability in the heterogeneous cloud environment: A survey of recent user-centric approaches. In *International Conference on Internet of Things and Cloud Computing (ICC)*, pages 62:1–62:7. ACM, 2016.

[86] Ahmed Moustafa, Minjie Zhang, and Quan Bai. Trustworthy stigmergic service composition and adaptation in decentralized environments. *IEEE Transactions on Services Computing*, 9(2):317–329, 2016.

[87] Dinh Khoa Nguyen, Francesco Lelli, Yehia Taher, Michael Parkin, Mike P Papazoglou, and Willem-Jan van den Heuvel. Blueprint template support for engineering cloud-based services. In *4th European Conference on Towards a Service-Based Internet (ServiceWave)*, pages 26–37. Springer, 2011.

[88] Alexander Nowak, Tobias Binz, Christoph Fehling, Oliver Kopp, Frank Leymann, and Sebastian Wagner. Pattern-driven green adaptation of process-based applications and their runtime infrastructure. *Computing*, 94(6):463–487, 2012.

[89] OASIS. SCA: Service Component Architecture. Standard, OASIS, 2011. `http://www.oasis-opencsa.org/sca`.

[90] OASIS. CAMP: Cloud Application Management for Platforms (Version 1.1). Standard, OASIS, 2012. `http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html/`.

[91] OASIS. TOSCA: Topology and Orchestration Specification for Cloud Applications (Version 1.0). Standard, OASIS, 2012. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/`.

[92] OASIS. OCCI: The Open Cloud Computing Interface. Standard, The Open Grid Forum (OGF), 2016. `http://occi-wg.org/`.

[93] Juliana Oliveira de Carvalho, Fernando Trinta, and Dario Vieira. PacificClouds: A flexible microservices based architecture for interoperability in multi-cloud environments. In *8th International Conference on Cloud Computing and Services Science, (CLOSER)*, pages 448–455. SciTePress, 2018.

[94] The Guide for Cloud Portability and Interoperability Profiles. Standard, IEEE - C/CCSC - Cloud Computing Standards Committee, 2011.

[95] Claus Pahl. Containerization and the PaaS cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[96] Claus Pahl, Li Zhang, and Frank Fowley. Interoperability standards for cloud architecture. In *3rd International Conference on Cloud Computing and Services Science (CLOSER)*, pages 123–126. SciTePress, 2013.

[97] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. A federated multi-cloud PaaS infrastructure. In *5th IEEE International Conference on Cloud Computing (CLOUD)*, pages 392–399. IEEE, 2012.

[98] Dana Petcu. Portability and interoperability between clouds: Challenges and case study. In *4th European Conference Towards a Service-Based Internet (ServiceWave)*, pages 62–74. Springer, 2011.

[99] Dana Petcu, Beniamino Di Martino, Salvatore Venticinque, Massimiliano Rak, Tamás Máhr, Gorka Esnal Lopez, Fabrice Brito, Roberto Cossu, Miha Stopar, Svatopluk Šperka, and Vlado Stankovski. Experiences in building a mosaic of clouds. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):12, 2013.

[100] Linh Manh Pham, Alain Tchana, Didier Donsez, Noel De Palma, Vincent Zurczak, and Pierre-Yves Gibello. Roboconf: A hybrid cloud orchestrator to deploy complex applications. In *8th IEEE International Conference on Cloud Computing (CLOUD)*, pages 365–372. IEEE, 2015.

[101] Harald Psaier and Schahram Dustdar. A survey on self-healing systems: approaches and systems. *Computing*, 91(1):43–73, 2011.

[102] Lie Qu, Yan Wang, Mehmet A Orgun, Ling Liu, Huan Liu, and Athman Bouguettaya. CCCloud: Context-aware and credible cloud service selection based on subjective assessment and objective assessment. *IEEE Transactions on Services Computing*, 8(3):369–383, 2015.

[103] Peter-Christian Quint and Nane Kratzke. Towards a lightweight multi-cloud DSL for elastic and transferable cloud-native applications. In *8th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 400–408. SciTePress, 2018.

[104] Ansar Rafique, Stefan Walraven, Bert Lagaisse, Tom Desair, and Wouter Joosen. Towards portability and interoperability support in middleware for hybrid clouds. In *IEEE International Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 7–12. IEEE, 2014.

[105] Ajith Ranabahu, E Michael Maximilien, Amit Sheth, and Krishnaprasad Thirunarayan. Application portability in Cloud Computing: An abstraction-driven perspective. *IEEE Transactions on Services Computing*, 8(6):945–957, 2015.

[106] Rajiv Ranjan. The cloud interoperability challenge. *IEEE Cloud Computing*, 1(2):20–24, 2014.

[107] Alessandro Rossini. Cloud application modelling and execution language (CAMEL) and the PaaSage workflow. In *Advances in Service-Oriented and Cloud Computing — Workshops of 4th European Conference on Service-Oriented and Cloud Computing (ESOCC)*, volume 567, pages 437–439, 2015.

[108] Karoline Saatkamp, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Topology splitting and matching for multi-cloud deployments. In *7th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 247–258. SciTePress, 2017.

[109] Mohamed Sellami, Sami Yangui, Mohamed Mohamed, and Samir Tata. PaaS-independent provisioning and management of applications in the cloud. In *6th IEEE International Conference on Cloud Computing (CLOUD)*, pages 693–700. IEEE, 2013.

[110] Jacopo Soldani, Damian A. Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[111] Josef Spillner, Yessica Bogado, Walter Benítez, and Fabio López-Pires. Co-transformation to cloud-native applications - development experiences and experimental evaluation. In *8th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 596–607, 2018.

[112] Paul Stack, Huanhuan Xiong, Dali Mersel, Maxime Makhloufi, Guillaume Terpend, and Dapeng Dong. Self-healing in a decentralised cloud management system. In *1st International Workshop on Next generation of Cloud Architectures*, pages 1–6, 2017.

[113] Adel Nadjaran Toosi, Rodrigo N. Calheiros, and Rajkumar Buyya. Interconnected cloud computing environments: Challenges, taxonomy, and survey. *ACM Comput. Surv.*, 47(1):1–47, 2014.

[114] Van Tran, Jacky Keung, Anna Liu, and Alan Fekete. Application migration to cloud: A taxonomy of critical factors. In *2nd International Workshop on Software Engineering for Cloud Computing (SECLOUD)*, pages 22–28. ACM, 2011.

[115] Quang Hieu Vu and Rasool Asal. Legacy application migration to the cloud: Practicability and methodology. In *8th IEEE World Congress on Services (SERVICES)*, pages 270–277. IEEE, 2012.

[116] Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann. Standards-based devops automation and integration using TOSCA. In *7th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pages 59–68. IEEE, 2014.

[117] Yue Yuan, Wenchang Shi, Bin Liang, and Bo Qin. An approach to cloud execution failure diagnosis based on exception logs in openstack. In *12th IEEE International Conference on Cloud Computing (CLOUD)*, pages 124–131. IEEE, 2019.

[118] Uwe Zdun, Elena Navarro, and Frank Leymann. Ensuring and assessing architecture conformance to microservice decomposition patterns. In *15th International*

*Conference on Service-Oriented Computing (ICSOC)*, pages 411–429. Springer, 2017.

[119] Dimitris Zeginis, Francesco D'Andria, Stefano Bocconi, Jesus Gorronogoitia Cruz, Oriol Collell Martin, Panagiotis Gouvas, Giannis Ledakis, and Konstantinos A. Tarabanis. A user-centric multi-PaaS application management solution for hybrid multi-cloud scenarios. *Scalable Computing: Practice and Experience*, 14(1):17–32, 2013.

[120] Jun-Feng Zhao and Jian-Tao Zhou. Strategies and methods for cloud migration. *International Journal of Automation and Computing*, 11(2):143–152, 2014.

[121] Zibin Zheng, Yilei Zhang, and Michael R Lyu. Investigating QoS of real-world web services. *IEEE Transactions on Services Computing*, 7(1):32–39, 2014.