Container-Based Module Isolation for Cloud Services

Stefan Kehrer, Florian Riebandt, Wolfgang Blochinger
Reutlingen University
{stefan.kehrer, wolfgang.blochinger}@reutlingen-university.de

```
@INPROCEEDINGS{KEHRER2019_SOSE,
      author={Stefan Kehrer and Florian Riebandt and Wolfgang Blochinger},
      title={Container-Based Module Isolation for Cloud Services},
      booktitle={2019 IEEE International Conference on Service-Oriented System
      Engineering (SOSE)},
      year={2019},
      pages={177-186},
      doi={10.1109/SOSE.2019.00032},
      ISSN={2642-6587},
}
```

Container-based Module Isolation for Cloud Services

Stefan Kehrer, Florian Riebandt, Wolfgang Blochinger
Parallel and Distributed Computing Group, *Reutlingen University*, Germany
firstname.lastname@reutlingen-university.de

***Abstract*—Due to frequently changing requirements, the internal structure of cloud services is highly dynamic. To ensure flexibility, adaptability, and maintainability for dynamically evolving services, modular software development has become the dominating paradigm. By following this approach, services can be rapidly constructed by composing existing, newly developed and publicly available third-party modules. However, newly added modules might be unstable, resource-intensive, or untrustworthy. Thus, satisfying non-functional requirements such as reliability, efficiency, and security while ensuring rapid release cycles is a challenging task. In this paper, we discuss how to tackle these issues by employing container virtualization to isolate modules from each other according to a specification of isolation constraints. We satisfy non-functional requirements for cloud services by automatically transforming the modules comprised into a container-based system. To deal with the increased overhead that is caused by isolating modules from each other, we calculate the minimum set of containers required to satisfy the isolation constraints specified. Moreover, we present and report on a prototypical transformation pipeline that automatically transforms cloud services developed based on the Java Platform Module System into container-based systems.**

***Index Terms*—continuous delivery, DevOps, non-functional requirements, container virtualization, deployment automation**

I. Introduction

DevOps [1] and continuous delivery [2] have been introduced to bridge the gap between development and operations. Automated delivery pipelines support frequent releases of new software versions. Technically, container virtualization provides a solid foundation to operate cloud services based on state-of-the-art container runtime environments [3]–[7]. By following this approach, each service can be delivered, deployed, and managed independently in line with the microservices architectural style [6], [8]. At the same time, many other benefits result from the isolated context established by container virtualization, such as improved security, fault containment, and fine-granular resource control.

With respect to the internal structure of dynamically evolving cloud services, modular software development has become the dominating paradigm to ensure flexibility, adaptability, and maintainability for each individual service [8]. By means of modern development frameworks (e.g., OSGi[1], JPMS[2]), cloud services can be designed and developed as a set of small and loosely coupled modules, which can be replaced fast, shared, and reused in another context. Thus, modular software development enables the rapid construction of cloud services by composing existing, newly developed and / or publicly available modules. Combining the microservices architectural style for the external architecture and modular software development for the internal architecture of cloud services is a powerful means to tackle both the increasing complexity of software-based systems and the need for rapid adaptation due to frequently changing requirements.

However, whereas composing newly developed and publicly available modules enables rapid software development, satisfying non-functional requirements such as reliability, efficiency, and security in this context is a hard task. Newly added modules might be unstable, resource-intensive, or untrustworthy. In fact, we would like to establish an *isolated context* for these modules to benefit from improved security, fault containment, and fine-granular resource control, but without the need to create, deploy, and manage a separate service. Building and maintaining too many (too fine-grained) services has been identified as one of the major issues of the microservices architectural style and leads to performance issues [9], [10].

In this paper, we discuss how to employ container virtualization to isolate modules of a single cloud service from each other according to a specification of isolation constraints. Our approach satisfies several non-functional requirements by automatically transforming a cloud service comprised of modules into a container-based system. To deal with the increased overhead that results from isolating modules by means of containers, we calculate the minimum set of containers required to satisfy the isolation constraints specified. In particular, our contributions are the following:

- We introduce the concept of *container-based module isolation* for evolving cloud services, which complements the operational principles of existing container runtime environments (and corresponding cloud offerings).
- We present the architecture of a transformation pipeline for transforming a cloud service comprised of modules into a container-based system with respect to the isolation constraints specified.
- We discuss and report on a prototypical implementation of the transformation pipeline that automatically transforms cloud services developed with the Java Platform Module System (JPMS) into container-based systems.
- We discuss the applicability of container-based module isolation in the context of continuous delivery.

[1]https://www.osgi.org.
[2]http://openjdk.java.net/projects/jigsaw/spec.

The remainder of this paper is structured as follows. In Section II, we motivate our work by describing several scenarios where container-based module isolation can be beneficially employed. Section III introduces the concept of container-based module isolation for cloud services. In Section IV, we present the architecture of a transformation pipeline that automatically transforms a cloud service into a container-based system with respect to the isolation constraints specified. Our prototypical implementation is discussed in Section V and evaluated in Section VI. Moreover, we show that our concepts are well-suited to enhance existing continuous delivery pipelines in Section VII. Section VIII reviews related work. In Section IX, we conclude our work.

II. Problem Statement and Motivation

Both modules and (micro)services are well-known components in the tradition of Component-based Software Development [11] from which more complex systems can be built. Whereas (micro)services have been proven to benefit from independent deployment and management as well as interoperability [10], modules are well-suited to establish code-level structure and enable code reuse and sharing [12]. Today, modern frameworks support modular software development (e.g., OSGi, JPMS), which introduced novel features such as service abstraction[3]. This enables loosely coupled modules and thus fosters flexibility, adaptability, and maintainability [12]. Modules can easily be added, replaced, refactored, or removed depending on the current context and application-specific requirements. Moreover, modules can be shared across teams and with the public by employing private / public code repositories. By following this approach, developers typically enhance their services with existing modules rather than developing required functionality from scratch. Making use of public code repositories and open source software (OSS) dramatically increases the speed of development and has become a major competitive advantage in recent years.

Besides, continuous delivery provides a foundation for shortening software release cycles. Continuous delivery pipelines have been introduced that automate the delivery of new software releases [13]–[15]. Moreover, container runtime environments are employed to operate and manage cloud services [4], [10]. These environments make use of container virtualization to ease the transition from development to production and substantially reduce the risk of faults due to environmental changes. To make use of these benefits, developers are required to provide their service in form of one or more containers [16].

As modules can be shared and reused in another context, they might be unstable, resource-intensive, or untrustworthy. Thus, operating all modules in a single container might impair non-functional requirements such as reliability, efficiency, and security. To deal with this problem, we argue that *container virtualization can be employed to isolate selected modules from one or more other modules*. Containers create an isolated context with benefits such as improved security, fault containment, and fine-granular resource control based on established Linux kernel features (such as namespaces and control groups) [5], [17]. Fig. 1 summarizes the benefits of modules and services as well as the benefits gained from container-based isolation of modules.

[3]Note that service abstraction in the context of modules is a programming-level concept not to be confused with web services.

In the following, we discuss several scenarios, in which isolating modules, i.e., limiting their interference, by means of container virtualization and thus deploying a cloud service as a set of containers can be beneficially employed.

Security: Modules imported from third parties contain potentially untrustworthy or malicious code. Whereas known vulnerabilities can be detected by vulnerability scanners in an automated manner, some (yet unknown) vulnerabilities cannot be detected by such scanners. Finding these vulnerabilities is a manual and time-consuming task resulting in delayed software releases. Moreover, developers might want to use third-party modules even if they are untrustworthy, because there is no alternative available. Implementing the provided functionality from scratch is often no option, e.g., in the context of rapid prototyping. Well-known examples of security threats are memory safety issues including buffer overflow attacks and dangling pointer bugs as well as file system manipulations. Whenever timely releases are an important requirement, operating third-party modules in isolation from security-sensitive modules avoids these threats.

Fault containment: Newly developed, prototypical, and potentially unstable software deteriorates the reliability of a service. Isolating potentially unstable modules by means of container virtualization enables fault containment [18] and thus successfully avoids consequential errors that affect the core functionality of a service [19].

Resource allocation and control: Isolating a module allows the fine-grained specification of resource requirements for the corresponding container [10]. Thus, modules can be restricted with respect to their CPU cycles or memory requirements

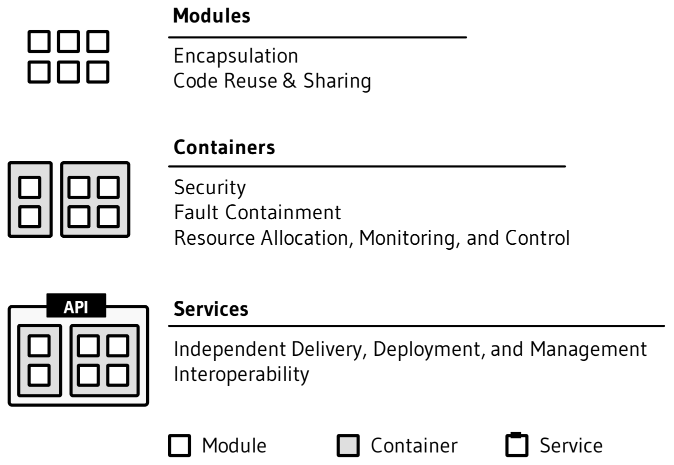

Fig. 1. The benefits of modules, isolating modules by means of container virtualization, and services can be easily combined.

by simply leveraging resource management features of the underlying operating system.

Testing and monitoring: Testing and monitoring multiple modules operated in a single container requires instrumentation at the programming level. Isolating a module in a container, however, eases monitoring. By following this approach, modules can be monitored by logging the resource consumption and runtime behavior of the corresponding container. Newly added modules can be isolated this way, monitored easily, and might become part of the core functionality over time, e.g., when they have been implemented more efficiently with respect to their resource consumption.

Whereas existing container runtime environments facilitate the deployment and operation of multi-container cloud services [16], we identified a lack of concepts and tools that allow developers to employ container virtualization to isolate modules from each other. On the other hand, manually constructing a set of containers is a frustrating and error-prone task, subject to frequent changes as a service evolves, and thus impedes rapid release cycles.

III. Methods and Concepts of Container-based Module Isolation

In this section, we introduce the concept of container-based module isolation for dynamically evolving cloud services. First, we describe the requirements identified and discuss how our approach satisfies these requirements. Then, we show that our concepts can be mapped to the operational principles of existing container runtime environments.

A. Requirements

We identified the following requirements with respect to container-based module isolation for evolving cloud services:

R1 Software developers require a simple means to specify isolation constraints, which ensure that a module runs in isolation from one or more other modules.

R2 Shortening software release cycles is a fundamental requirement. Hence, an automated transformation of a modular cloud service to a container-based system has to be ensured. Similarly, automated deployment must be facilitated by generating the deployment artifacts required.

R3 Isolating modules by means of containers leads to various sources of overhead, which should be minimized. Thus, it is essential to find the minimum set of containers required for deployment purposes while satisfying all isolation constraints specified.

R4 To access isolated modules, inter-container communication is required. Access and location transparency have to be achieved to avoid code changes (by developers).

B. Container-based Module Isolation

The core idea of our approach is that isolating modules by means of container virtualization can be beneficially employed for a variety of use cases (cf. Section II). Therefore, we have to provide simple yet expressive constructs to specify isolation constraints among one or more modules (R1).

In this context, a modular cloud service is represented by a set of modules $\mathcal{M} = \{m_1, ..., m_n\}$. We define a set of isolation constraints as a binary relation $\perp \subseteq \mathcal{M} \times \mathcal{M}$, where an isolation constraint $(m_k, m_l) \in \perp$ states that the modules m_k and m_l have to be operated in different containers. This simple formalization also allows the construction of higher-level isolation constraints, which can be automatically transformed to a binary relation. We discuss several examples in Section III-C.

The generation of a container-based system based on the isolation constraints specified has to be automated (R2). Therefore, we propose an automated transformation pipeline that transforms a given cloud service and a set of isolation constraints to a container-based system (cf. Fig. 2). Technically, the transformation pipeline generates deployment artifacts that can be used for automated deployment. Details of the transformation pipeline are described in Section IV. In the following, we describe the core concepts of the transformation by means of a formal specification. We define the Container-based Module Isolation Problem (CMIP) and the Minimum Container-based Module Isolation Problem (Minimum-CMIP) based on [20].

Let $C = \{c_1, ..., c_m\}$ be a set of containers in each of which one or more modules can be operated.

Definition 1. ***Container-based Module Isolation Problem (k-CMIP)*** *Given a cloud service with a set of modules $\mathcal{M}$ and a set of isolation constraints $\perp$ over $\mathcal{M}$, k-CMIP is the decision problem whether there is a set C of containers with $|C| \leq k$ and a mapping $f : \mathcal{M} \to C$ such that all isolation constraints in $\perp$ are satisfied.*

k-CMIP is equivalent to the graph (vertex) coloring problem [21] and thus NP-complete. This can be shown by mapping the modules of a cloud service to the vertices of a graph and the isolation constraints specified to the graph's edges. In this context, a valid coloring of the graph represents a mapping of modules to containers. Note that a graph only has a valid coloring if the edge relation is irreflexive. Analogously, an

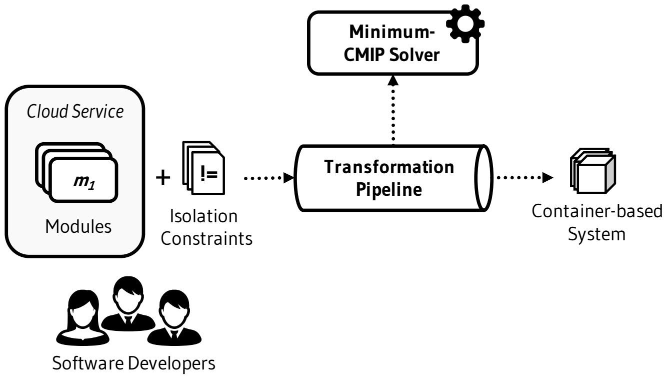

Fig. 2. Our approach for container-based module isolation considers isolation constraints specified by software developers and automatically transforms a given modular cloud service into a container-based system by solving the underlying optimization problem.

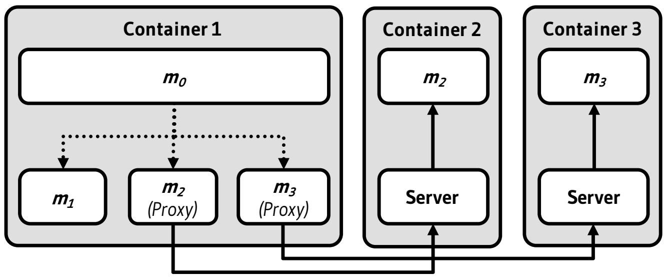

Fig. 3. Exemplary container-based system that satisfies a set of isolation constraints specified.

isolation constraint $(m_k, m_k) \in \perp$ cannot be satisfied because a module cannot be isolated from itself. Therefore, we assume $\nexists(m_i, m_i) \in \perp$.

To minimize the overhead resulting from container-based module isolation, we have to find the minimum set of containers required to satisfy the isolation constraints specified (R3). This can be accomplished by solving the corresponding optimization problem.

Definition 2. ***Minimum Container-based Module Isolation Problem (Minimum-CMIP)*** *Given a cloud service with a set of modules $\mathcal{M}$ and a set of isolation constraints $\perp$ over $\mathcal{M}$, Minimum-CMIP is the optimization problem of finding a minimum set of containers C, i.e., finding a minimum k, such that a solution to k-CMIP exists for $\mathcal{M}$ and $\perp$.*

A Minimum-CMIP solver thus computes a valid and optimized container assignment for the modules of a given cloud service. Based on the calculated container assignment, the proposed transformation pipeline builds a container-based system (cf. Fig. 2). However, isolating modules by means of containers leads to inter-container communication. To ensure access and location transparency (R4), additional proxy mechanisms are required that serve local method / function calls by routing them to isolated modules. Fig. 3 shows an exemplary container-based system with four modules. As we can easily see, m_2 and m_3 have been isolated from module m_0 and each other. Nevertheless, m_0 accesses the isolated modules m_2 and m_3 in the same manner as the local module m_1. The required proxy mechanisms have to be generated automatically. This is accomplished by the transformation pipeline, which is described in detail in Section IV.

C. Specification of Isolation Constraints

In the following, we show that the formalization of isolation constraints as defined above provides a powerful means to specify higher-level isolation constraints, which can be automatically transformed to a binary relation. Therefore, we discuss several examples based on the application scenarios described in Section II.

A simple higher-level isolation constraint can be textually described as *isolate module m_k from all other modules in $\mathcal{M}$*, e.g., for monitoring purposes. This isolation constraint can be transformed by adding isolation constraints (m_k, m) to $\perp$ for all $m \in \mathcal{M} \setminus \{m_k\}$. Modules might also be organized according to security levels from which isolation constraints can be deduced. For instance, to ensure that all modules with security level x cannot be operated together with modules of a lower security level: For all $m_k \in \mathcal{M}$ with $secLevel(m_k) = x$, add isolation constraints (m_k, m) to $\perp$ for all $m \in \mathcal{M} \setminus \{m_k\}$, where $secLevel(m) < secLevel(m_k)$. Here, a hierarchy of security levels is only an exemplary use case for any other form of higher-level isolation constraints that can be deduced from module meta data.

To improve fault tolerance, one might isolate at least two modules that implement the same interface. In this case, there is a higher probability that we can access the provided functionality even if one module is currently not available. For creating the corresponding isolation constraints, the source code of the cloud service has to be scanned.

On the other hand, existing isolation constraints can also be removed. For instance, when a module has been proven to be stable in production or after its source code has been checked comprehensively and security issues have been eliminated.

Technically, isolation constraints can be specified in a simple text file (for an example see Section V).

D. Deployment to Container Runtime Environments

Existing container runtime environments support the deployment and management of multi-container services [16]. In Kubernetes[4] for example, a so-called *pod*[5] is defined as a group of one or more containers with shared resources such as storage and network. The containers of a pod are guaranteed to be co-located and are given a shared context. In particular, containers of a pod share an IP address and port space, which eases the generation of proxy mechanisms for inter-container communication. The pod concept has been adopted by other container runtime environments. For example, Marathon, a container orchestrator for Apache Mesos, supports pods with the version 1.4[6]. Nomad supports so-called groups[7] with similar semantics. Our prototypical implementation (cf. Section V) generates deployment artifacts for Kubernetes and thus facilitates the automated deployment of a given cloud service.

These container runtime environments can be hosted in a public or private cloud setting. Furthermore, they are also offered as a service by cloud providers. For instance, Amazon Web Services (AWS) offers Kubernetes as a service with Amazon EKS[8]. By following this approach, customers benefit from an automated setup with multiple availability zones and cluster auto-scaling.

IV. Transformation Pipeline

In this section, we describe the architecture of a transformation pipeline for transforming a given cloud service represented by a set of modules and a set of isolation constraints

[4]https://kubernetes.io.
[5]https://kubernetes.io/docs/concepts/workloads/pods.
[6]https://mesosphere.github.io/marathon/docs/pods.html.
[7]https://www.nomadproject.io/docs/job-specification/group.html.
[8]https://aws.amazon.com/de/eks.

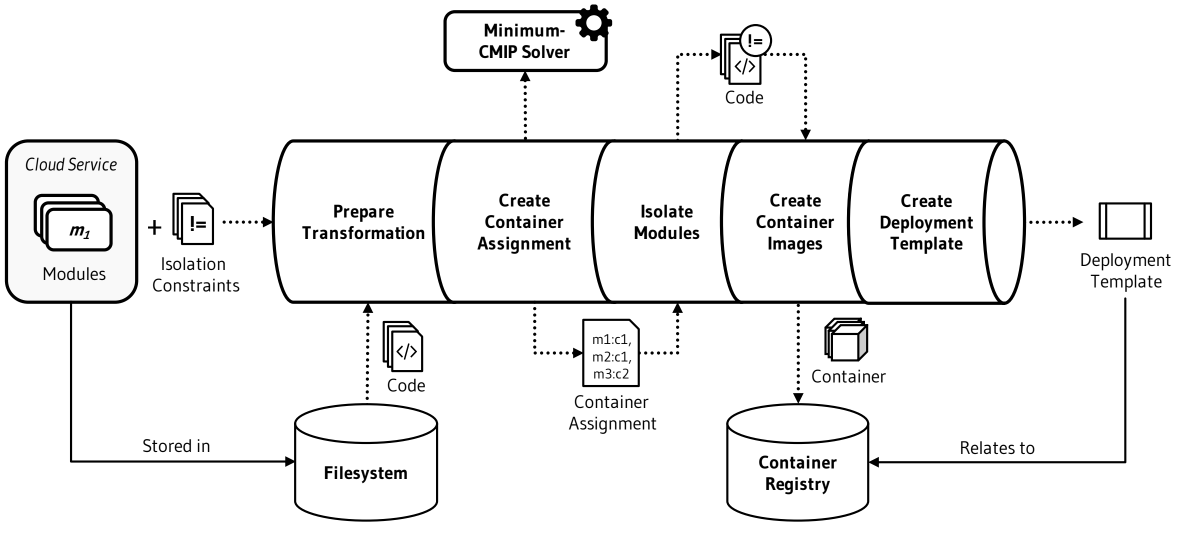

Fig. 4. The transformation pipeline is comprised of five steps and finally produces a container-based system according to the isolation constraints specified.

into a container-based system. Different technologies can be used to implement this transformation pipeline (for an example see Section V). The transformation pipeline is essentially comprised of five steps, which are depicted in Fig. 4 and explained in the following.

Prepare transformation: The transformation pipeline performs code-level transformations to isolate modules of the cloud service based on the isolation constraints specified. Typically, the source code of the corresponding cloud service is required. Source code files have to be provided by developers and are retrieved from a filesystem or a code repository (version control system) such as Subversion[9], Git[10], or Mercurial[11]. Additional validity checks can be applied to ensure that the source code provided and the specification of isolation constraints can be processed by the transformation pipeline.

Create container assignment: To create a valid container assignment, i.e., an assignment of modules to containers, the underlying Minimum-CMIP has to be solved. Based on the given cloud service, a corresponding Minimum-CMIP instance is derived. Therefore, the provided source code files have to be scanned to identify modules. As a result, an undirected graph with the modules as vertices and the isolation constraints as edges is generated (cf. Section III). Consequently, algorithms that are able to solve the graph coloring problem can be applied to solve the constructed Minimum-CMIP. However, note that solving such an NP-optimization problem requires an approximation algorithm because large problems cannot be solved exactly (with acceptable runtimes). In Section V, we describe a prototypical implementation of the Minimum-CMIP solver. The solution of this optimization problem provides the number of containers required as well as the assignment of modules to containers. The container assignment is the construction plan according to which the following steps build a container-based system.

Isolate modules: In this step, several code-level transformations are applied to isolate modules according to the container assignment provided. To isolate modules by means of containers, inter-container communication is required. Inter-container dependencies are resolved (1) by generating proxy modules, each of which replaces an original module with a proxy and serves local calls by routing them to the isolated module and (2) by generating a corresponding server per container that dispatches inter-container calls to the modules hosted by the container (cf. Fig. 3). Inter-container calls can be implemented based on any remote procedure call (RPC) technology. However, special emphasis has to be put on ensuring access and location transparency (related to requirement R4).

Create container images: Container images are the artifacts from which containers can be instantiated. To specify the construction of a container image, a build specification (e.g., a Dockerfile) is employed. In this step, a build specification is generated per container listed by the container assignment. A build specification contains the modules assigned to the container, the generated proxy mechanisms, as well as dependencies required for execution (e.g., interpreter, runtime system). In this context, it might be required to compile developer-supplied and generated modules. The build specifications are employed to build container images. Finally, these container images are pushed to a container registry from which they can be retrieved during deployment (cf. Fig. 4).

Create deployment template: For deployment purposes, a deployment template is required that specifies the container images related to the service. A deployment template can be processed by a container runtime environment in an automated manner. The syntax of the deployment template depends on the target runtime environment. For example, our prototype described in Section V creates a deployment template for Kubernetes.

[9]https://subversion.apache.org.
[10]https://git-scm.com.
[11]https://www.mercurial-scm.org.

V. Prototypical Implementation

To validate the concept of container-based module isolation, we implemented a prototype of the proposed transformation pipeline in Java. We chose the Java Platform Module System as modular development framework, Docker[12] to build containers, and Kubernetes as target runtime environment.

A. *Java Platform Module System (JPMS)*

The novel JPMS, introduced with Java 9, provides the basic constructs for modular software development of Java-based applications and thus aims at solving several well-known dependency management problems [22]. Therefore, it proposes modules for encapsulating code, which explicitly declare their dependencies to other modules, exported packages, and provided services (implemented interfaces) in the module descriptor file (`module-info.java`). By design, dependencies are static and thus can be verified at compile time. Modules can be distributed in form of modular JAR (Java Archive) files.

The JPMS does not only enable application-level modularity, but has been used to modularize the Java Development Kit (JDK) itself. This has a major advantage over former monolithic Java Runtime Environments (JRE): By resolving explicit dependencies, only required platform modules are packaged into a so-called *modular runtime image* [23]. A novel JDK-tool called `jlink`[13] can be used to create these runtime images. This is particularly beneficial in the context of container virtualization because the size of container images is significantly reduced.

B. *Implementation*

In the following, we describe how we implemented each step of the transformation pipeline depicted in Fig. 4. The expected input is a Java-based cloud service developed with the JPMS. Each module of the service contains a `JSON` file that specifies its isolation constraints.

Prepare transformation: We retrieve the source code files provided by developers from the local filesystem. Each module contains a simple text file describing a list of modules from which it should be isolated. We check that the specification of isolation constraints complies with our `JSON` format.

Create container assignment: We identify modules by means of the Java Module API and parse the corresponding `JSON` files to identify isolation constraints. Based on this information, a Minimum-CMIP is constructed. We generate an undirected graph with the identified modules as vertices and the identified isolation constraints as edges and check if $\nexists(m_i, m_i) \in \perp$ holds (cf. Section III). To solve the Minimum-CMIP, we rely on the Welsh-Powell algorithm [24], which is a heuristic (greedy) algorithm for graph coloring. Our Minimum-CMIP solver produces a valid container assignment according to which we isolate modules in the following.

[12]https://www.docker.com.

[13]https://openjdk.java.net/jeps/282.

Isolate modules: To isolate modules according to the container assignment, we apply several code-level transformations. First, we replace modules that are required locally but hosted in another container with generated proxy modules. Such dependencies can be derived from the module descriptor file (cf. Section V-A). Proxy modules provide the same interface as the replaced module (via the module descriptor) and internally perform a remote procedure call to the original module (via localhost). Next, we create a server module per container that dispatches inter-container calls to the modules hosted by the container. Technically, remote procedure calls are implemented based on the Java bindings[14] of ZeroMQ[15], a high-performance messaging library. Access and location transparency is thus ensured by encapsulating the inter-container communication required inside local proxy modules. Proxy and server modules are generated with JavaPoet[16], a source code generation library. Note that we assume all containers to be co-located by means of the pod concept (cf. Section III-D). Thus, discovery mechanisms are not required.

Create container images: We create a temporary folder per container on the local filesystem, which contains the build specification of and all artifacts required by the corresponding container. First, we compile the developer-supplied and generated modules resulting in a bunch of JAR files. Then, a Dockerfile per container has to be created. The generation of Dockerfiles is implemented based on Apache FreeMarker[17], which is an open-source template engine. The Dockerfiles generated are based on the Alpine Linux base image[18]. To build container images, we assume a Docker Engine running on the host. We connect to the Docker Engine by using Docker-Client[19]. Docker-Client connects to the Docker Engine via the default UNIX domain socket provided to control Docker-specific functionality. We build the container images described by the generated Dockerfiles and push them to a private Docker Registry[20] from which they can be retrieved for deployment (cf. Fig. 4).

Create deployment template: In Kubernetes, deployment templates are specified in `YAML`. As mentioned in Section III-D, we create a pod per cloud service. Thus, the generated `YAML` file defines a single pod and lists all the container images related to this pod that have been built before. The generation of deployment templates is also implemented based on Apache FreeMarker.

VI. Evaluation

In the following, we evaluate our prototypical implementation and discuss several sources of overhead resulting from container-based module isolation. We also show that our transformation pipeline is well-suited in the context of continuous

[14]https://github.com/zeromq/jeromq.

[15]http://zeromq.org.

[16]https://github.com/square/javapoet.

[17]http://freemarker.org.

[18]https://hub.docker.com/_/alpine.

[19]https://github.com/spotify/docker-client.

[20]https://docs.docker.com/registry.

delivery, i.e., that it does not introduce limitations on fast and frequent releases.

Testbed. Our testbed consists of a Kubernetes cluster hosted in our OpenStack[21]-based cloud environment. The Kubernetes master is operated on a CentOS 7 virtual machine (VM) with 2 vCPUs clocked at 2.6 GHz, 4 GB RAM, and 40 GB disk. Kubernetes nodes are operated on a CentOS 7 VM with 8 vCPUs clocked at 2.6 GHz, 8 GB RAM, and 40 GB disk.

Communication Overhead. As our implementation transforms method calls to isolated modules into inter-container calls (crossing container boundaries), communication latency is expected to be higher compared to intra-container calls. For evaluation purposes, we measured the latency of an intra-container and an inter-container method call. The method employed for our measurements requires a `String` object as argument and simply returns this `String` object. The serialized size of this object is only 8 bytes. It thus establishes a baseline for more complex, arbitrarily implemented methods. We executed 150 measurements for both the intra- and the inter-container method call. All measurements were performed in our Kubernetes testbed described above. Based on our measurement, we calculated an average intra-container latency T_{intra} of 60.21 ± 20.4 microseconds and an average inter-container latency T_{inter} of 8405.15 ± 4977.5 microseconds. Thus, the average inter-container latency T_{inter} is about 140 times higher. Depending on the actual execution time of the method itself and how frequently it is called, the performance penalty might be negligible. Note that our measurements evaluate a worst case scenario. The execution time of methods is usually much higher. Nonetheless, trade-offs are required with respect to conflicting non-functional properties. Isolation constraints provide a means to trade security for response time.

Deployment Overhead. We performed a simple experiment to compare the deployment time of a single-container cloud service with the deployment time of several multi-container cloud services. We measured the deployment time T_{deploy} for a single-container service and three multi-container services with 5, 10, and 20 containers per service. The container image employed for the measurements is based on the Alpine Linux 3.7 base image and OpenJDK 11. It has a total image size of 253 MB. Our measurements were performed in our Kubernetes testbed. We calculated an average deployment time T_{deploy} of $13.0 \pm 0.7\,\mathrm{s}$ (1 container), $20.7 \pm 1.1\,\mathrm{s}$ (5 containers), $32.0 \pm 1.9\,\mathrm{s}$ (10 containers), and $59.0 \pm 6.5\,\mathrm{s}$ (20 containers) based on three deployment runs for each service. Note that the deployment of multiple containers requires sublinear time when compared to a single-container service. For instance, the deployment time of a service with 20 containers only increases by a factor of roughly 4.5.

Minimum-CMIP Solver Overhead. In the context of continuous delivery, we have to ensure that solving the Minimum-CMIP (which is part of the presented transformation pipeline) does not negatively affect the overall transformation time. To evaluate the performance of the Minimum-CMIP solver, we measured the execution time T_{solve} that is required to create a valid container assignment. In the following, we describe an experiment that shows how our implementation performs for a wide range of different problems with different characteristics (such as number of modules and isolation constraints as well as constraint densities).

We generated 1400 prototypical Minimum-CMIP instances and measured the corresponding execution time T_{solve}. Minimum-CMIP instances were constructed for 28 different sets of modules $\mathcal{M}_i, i \in \{1, 2, ..., 28\}$, with

$$|\mathcal{M}_i| = \begin{cases} 10 & \text{for} \quad i = 1 \\ |\mathcal{M}_{i-1}| + 10 & \text{for} \quad 1 < i < 10 \\ 100 & \text{for} \quad i = 10 \\ |\mathcal{M}_{i-1}| + 100 & \text{for} \quad 10 < i < 19 \\ 1000 & \text{for} \quad i = 19 \\ |\mathcal{M}_{i-1}| + 1000 & \text{for} \quad 19 < i < 28 \\ 10000 & \text{for} \quad i = 28 \end{cases} \tag{1}$$

Isolation constraints are generated by means of the GENERATECONSTRAINTS procedure shown in Algorithm 1. For each module $m_k \in \mathcal{M}_i$ it adds a random amount of isolation constraints to $\perp_i$. GENERATECONSTRAINTS can be parameterized by means of the weighting factor w to control the constraint density. For each $\mathcal{M}_i$, we employed 10 different weighting factors $w \in \{0.1, 0.2, ..., 1.0\}$ for each of which we generated 5 random configurations of isolation constraints leading to $28 \cdot 10 \cdot 5 = 1400$ Minimum-CMIP instances in total.

Fig. 5 shows the measured execution time T_{solve} for all 1400 Minimum-CMIP instances generated. As we can easily see, all container assignments have been created in less than 23 seconds, even in a scenario with 10000 modules and more than 50 million isolation constraints. Consequently, these

Algorithm 1 Generation of Isolation Constraints

```
1: procedure GENERATECONSTRAINTS($\mathcal{M}_i, w$)
2:     $\perp_i \leftarrow \{\}$
3:     $\xi \leftarrow |\mathcal{M}_i| \cdot w$
4:     for each $m_k \in \mathcal{M}_i$ do
5:         $x \leftarrow$ a random integer in $\{0, ..., \xi\}$
6:         $k \leftarrow$ GETINDEX($m_k$)
7:         for $j = 0,\ j < x,\ j{+}{+}$ do
8:             $l \leftarrow k$
9:             while $l = k$ do
10:                $l \leftarrow$ a random integer in $\{1, ..., |\mathcal{M}_i|\}$
11:            end while
12:            $m_l \leftarrow$ GETELEMENT($\mathcal{M}_i, l$)
13:            if $(m_l, m_k) \notin \perp_i$ and $(m_k, m_l) \notin \perp_i$ then
14:                $\perp_i \leftarrow \perp_i \cup \{(m_k, m_l)\}$
15:            end if
16:        end for
17:    end for
18:    return $\perp_i$
19: end procedure
```

[21] https://www.openstack.org.

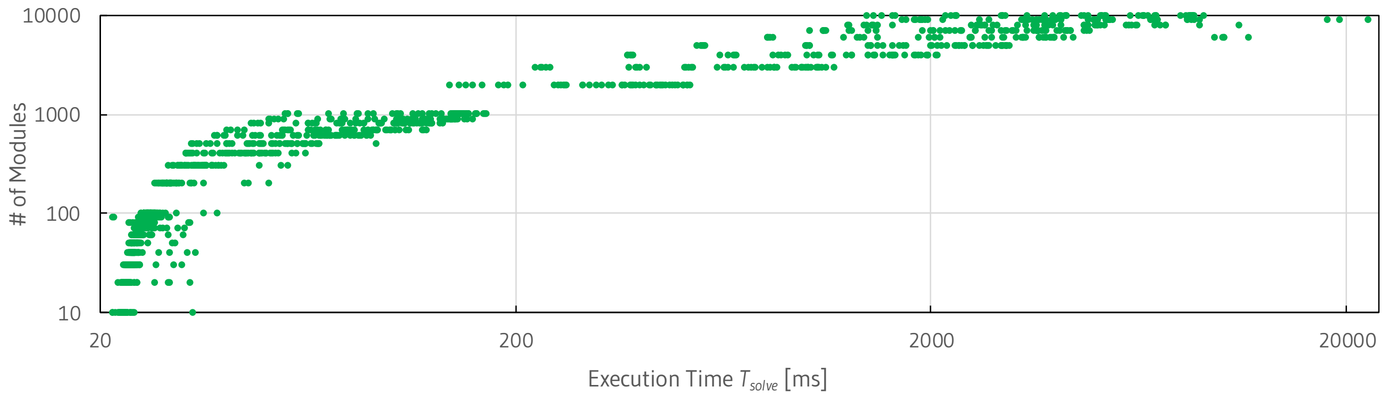

Fig. 5. We measured the execution time T_{solve} of 1400 Minimum-CMIP instances by running our Minimum-CMIP solver implemented in Java on a CentOS 7 VM with 8 vCPUs clocked at 2.6 GHz, 8 GB RAM, and 40 GB disk in our OpenStack-based cloud environment. Note that our implementation of the Minimum-CMIP solver is a sequential one, i.e., it does not make use of parallel processing. Both axes of the plot are scaled logarithmically to base 10.

measurements show that our Minimum-CMIP solver does not introduce major limitations on fast and frequent releases.

Limitations. Technically, because serialization is required for inter-container communication, we can only isolate modules if serialization of input and output objects is applicable. Security benefits of container-based module isolation as described in Section II depend on the container virtualization technique employed. Furthermore, resource allocation and control as well as monitoring depend on the container runtime environment. However, note that the Kubernetes ecosystem, for instance, provides many tools that can be used in conjunction with our concepts. Moreover, multi-container services and inter-container communication also introduce additional challenges for conventional debugging tools.

VII. Applicability and Context

Continuous delivery is used to shorten software release cycles [2]. To reduce manual and error-prone steps, continuous delivery heavily relies on automation. A system that automates the delivery process of a single independently deployable service is called *continuous delivery pipeline*. A continuous delivery pipeline streamlines the delivery process to reduce the time from development to release and deployment in the production environment [14]. Such a pipeline typically includes multiple stages with unit and integration as well as performance or user acceptance tests. Whereas most of these tests are performed in an automated manner, also manual ones can be included in the pipeline. One might also design a pipeline with multiple branches to run several tests in different test environments in parallel. Furthermore, a continuous delivery pipeline also includes feedback cycles because the essential idea is to get feedback (such as failed tests) as fast as possible [2].

Fig. 6 illustrates an exemplary continuous delivery pipeline and shows how the concept of container-based module isolation can be integrated into this setting by means of the described transformation pipeline (cf. Section IV). To benefit from container-based module isolation, we simply have to insert our transformation pipeline as a new stage into an existing continuous delivery pipeline. The generated deployment template can be used to deploy the corresponding service to the production environment. Whereas the continuous delivery pipeline itself heavily relies on automation, the final deployment to the production environment is often designed as a manual step. However, this manual step should be facilitated by automated deployment scripts or templates [2]. In our case, the generated deployment template can be used to trigger either an automated deployment or a manual one by means of a self-service tool. Alternatively, the deployment template also provides a means to deploy the service to one or more test environments to execute performance, security, or user acceptance tests. The feedback loop enables developers to rapidly react to failed tests, e.g., in form of code changes, service redesign, or the specification of new isolation constraints. Feedback from the transformation pipeline might be related to conflicting isolation constraints (thus describing a Minimum-CMIP that cannot be solved) or the specification of module names that cannot be located in the source code.

Another option to benefit from the concepts presented in this work is the isolation of modules only for testing purposes as described in Section II. By following this approach, developers are able to isolate modules for monitoring purposes or running stress tests on release candidates. Performance issues can be identified by simply logging the resource consumption and runtime behavior of the corresponding container, finally resulting in a better understanding of newly developed or added third-party modules. We deem this specifically useful in the context of rapid prototyping.

The use of open source software and the need to change rapidly pose hard challenges for satisfying non-functional requirements [25]. By integrating the concepts presented in this work into an existing continuous delivery pipeline, one is able to benefit from both fast software releases as well as security, fault containment, and fine-granular resource control.

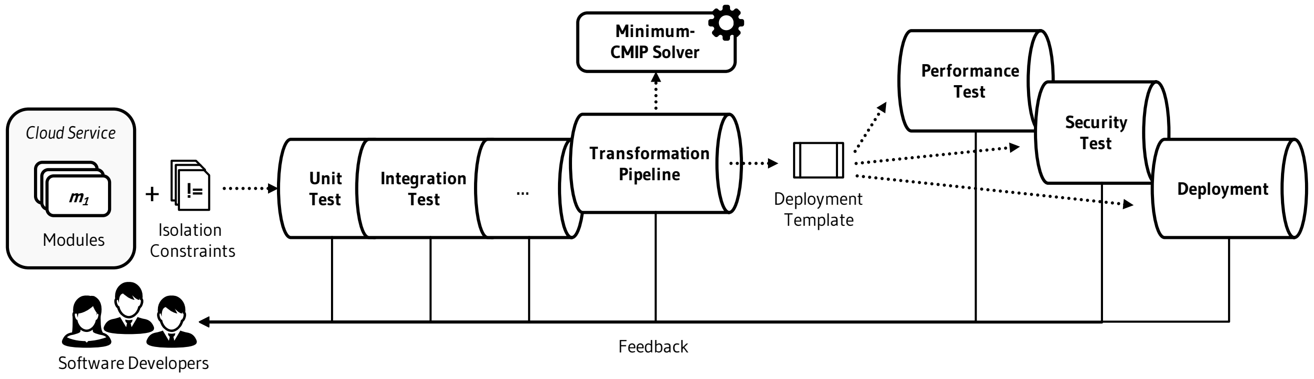

Fig. 6. Our transformation pipeline can be integrated into existing continuous delivery pipelines to benefit from container-based module isolation.

VIII. Related Work

It has been widely recognized that continuous delivery is required to address the need for fast and frequent software releases [1], [2], [26]. Whereas configuration management approaches provide a means to automate the delivery and deployment of cloud services [8], [14], [27], more recently containers have been used to package, deliver, and deploy services [3], [4], [17], [19], [28].

OSGi is another well-known framework for developing Java-based modular software. Modules are called bundles in OSGi jargon. An additional service layer provides a publish-find-bind model for Java interfaces, which can be dynamically bound by means of a service registry mechanism. Whereas both JPMS and OSGi offer code-level encapsulation, they fail to offer isolation in terms of resource allocation and security [29]–[31]. In [29], process-level isolation is employed to isolate modules based on the OSGi framework. The main motivations given are improved security and stability. The authors mainly consider use cases in the field of home automation, whereas we employ container-based module isolation for evolving cloud services deployed to state-of-the-art container runtime environments. The authors of [32] describe a customized JVM that provides isolation and resource accounting for OSGi-based applications. The authors of [20] provide a formal definition of the so-called Module Isolation Problem (MIP) and the corresponding Minimum Module Isolation Problem (Minimum-MIP). Whereas the original concept has been described in the context of distributed computing infrastructures, we apply the idea of module isolation to dynamically evolving cloud services. Our formulation of the CMIP and the Minimum-CMIP represent adaptations of the ones presented in [20], which we tailored to our problem setting (cf. Section II). Fundamental differences of both approaches are that the authors of [20] do not consider container virtualization, support hot-deployment of modules, and employ an online algorithm for solving the underlying optimization problem. We do not support hot-deployment of modules, which counteracts the principles of container-based systems. Consequently, Minimum-CMIP has to be solved only once per software release, which is ensured by the Minimum-CMIP solver in our transformation pipeline.

Service-level isolation is a well-known principle in the design and development of cloud services [33]. In this work, we show that containers can be beneficially employed not only for service-level isolation but to isolate modules of a single service from each other, which has been shown to be beneficial for a variety of use cases (cf. Section II). Current container runtime environments (cf. Section III-D) establish a shared context for all containers of a single pod by means of the same kernel features that are employed to isolate each container. Thus, they support both container-based service isolation and container-based module isolation (as proposed in this work) out of the box.

In [16], the authors argue that containers are the fundamental objects from which distributed systems should be built and discuss several design patterns for container-based systems. Specifically, the *single-node, multi-container application patterns* described are relevant to our work. In this context, three patterns namely the sidecar, ambassador, and adapter pattern have been described. All these patterns describe a specific composition of containers, which can be reused in other contexts. An example for the ambassador pattern is a container that acts as a proxy for communication from / to a main container. This proxy container can be shared across teams and / or with the public, thus other applications can easily reuse the proxy mechanism provided. In contrast to this approach, we employ containers to enforce isolation among modules. Whereas our approach is restricted to a specific programming language, it enables fine-grained control over module composition and non-functional requirements while minimizing the resulting overhead.

IX. Conclusion

In this work, we introduce the concept of container-based module isolation, which enables software developers to deal with non-functional requirements in the context of dynamically evolving cloud services. We present an automated transformation pipeline, which has been validated and evaluated by means of a prototypical implementation. Moreover, we discuss the applicability of our concepts to enhance existing continuous delivery pipelines. As a result, we show how to combine the benefits of modular software development, container virtualization, and continuous delivery to deal with several non-functional requirements in the context of dynamically evolving and rapidly changing cloud services.

The concepts presented in this work pave the way for future research on new service development and engineering methods. We plan to consider container-based module isolation in the context of cloud migration. More specifically, we plan to apply our concepts for migrating monolithic legacy applications to the cloud. Another idea is the automatic deduction or recommendation of isolation constraints based on testing results. Generally speaking, one might think of whole test suites for testing cloud services based on OSS or third-party components with respect to their non-functional requirements. Going even one step further, we envision an intelligent controller based on an automated feedback loop that analyzes testing results, deduces the causes of faults, specifies new isolation constraints, and executes another test series without human intervention.

Acknowledgment

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program *Services Computing*.

References

[1] J. Humble and J. Molesky, “Why enterprises must adopt devops to enable continuous delivery,” *Cutter IT Journal*, vol. 24, no. 8, p. 6, 2011.

[2] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley, 2010.

[3] R. Peinl, F. Holzschuher, and F. Pfitzer, “Docker cluster management for the cloud - survey results and own solution,” *Journal of Grid Computing*, vol. 14, no. 2, pp. 265–282, 2016.

[4] N. Kratzke and P.-C. Quint, “Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study,” *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[5] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, “Cloud container technologies: a state-of-the-art review,” *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[6] C. Pahl and P. Jamshidi, “Microservices: A systematic mapping study,” in *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, ser. CLOSER 2016. Portugal: SCITEPRESS - Science and Technology Publications, Lda, 2016, pp. 137–146.

[7] F. Leymann, U. Breitenbücher, S. Wagner, and J. Wettinger, *Native Cloud Applications: Why Monolithic Virtualization Is Not Their Foundation*. Cham: Springer International Publishing, 2017, pp. 16–40.

[8] J. Wettinger, U. Breitenbücher, and F. Leymann, “Dyn tail - dynamically tailored deployment engines for cloud applications,” in *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing*, ser. CLOUD ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 421–428.

[9] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017, pp. 195–216.

[10] J. Soldani, D. A. Tamburri, and W.-J. V. D. Heuvel, “The pains and gains of microservices: A systematic grey literature review,” *Journal of Systems and Software*, vol. 146, pp. 215 – 232, 2018.

[11] T. Vale, I. Crnkovic, E. S. de Almeida, P. A. da Mota Silveira Neto, Y. a. C. Cavalcanti, and S. R. de Lemos Meira, “Twenty-eight years of component-based software engineering,” *Journal of Systems and Software*, vol. 111, pp. 128 – 148, 2016.

[12] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[13] J. Wettinger, U. Breitenbücher, M. Falkenthal, and F. Leymann, “Collaborative gathering and continuous delivery of devops solutions through repositories,” *Computer Science - Research and Development*, vol. 32, no. 3, pp. 281–290, Jul 2017.

[14] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, “Streamlining devops automation for cloud applications using tosca as standardized metamodel,” *Future Generation Computer Systems*, vol. 56, no. C, pp. 317–332, 2016.

[15] S. Kehrer and W. Blochinger, “Tosca-based container orchestration on mesos,” *Computer Science - Research and Development*, vol. 33, no. 3, pp. 305–316, Aug 2018.

[16] B. Burns and D. Oppenheimer, “Design patterns for container-based distributed systems,” in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016.

[17] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[18] T. Saridakis, “Design patterns for fault containment,” in *Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2003, pp. 493–520.

[19] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, omega, and kubernetes,” *ACM Queue*, vol. 14, pp. 70–93, 2016.

[20] S. Schulz and W. Blochinger, “Adjustable module isolation for distributed computing infrastructures,” in *2011 IEEE/ACM 12th International Conference on Grid Computing*, Sept 2011, pp. 98–105.

[21] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, “Introduction to algorithms (3rd ed.),” 2009.

[22] A. Jecan, *Java 9 Modularity Revealed: Project Jigsaw and Scalable Java Applications*. Apress, 2017.

[23] N. Black, “Nicolai parlog on java 9 modules,” *IEEE Software*, vol. 35, no. 3, pp. 101–104, May 2018.

[24] D. J. A. Welsh and M. B. Powell, “An upper bound for the chromatic number of a graph and its application to timetabling problems,” *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[25] C. Esposito, A. Castiglione, and K.-K. R. Choo, “Challenges in delivering software in the cloud as microservices,” *IEEE Cloud Computing*, vol. 3, no. 05, pp. 10–14, 2016.

[26] L. Chen, “Continuous delivery: Huge benefits, but challenges too,” *IEEE Software*, vol. 32, no. 2, pp. 50–54, Mar 2015.

[27] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, and T. Spatzier, “Integrating configuration management with model-driven cloud management based on tosca,” in *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013, 8-10 May 2013, Aachen, Germany*. SciTePress, 2013, pp. 437–446.

[28] S. Kehrer and W. Blochinger, “Autogenic: Automated generation of self-configuring microservices,” in *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, INSTICC. SciTePress, 2018, pp. 35–46.

[29] T. Wegner, “A secure multi-provider osgi platform enabling process-isolation by using distribution,” in *Proceedings of the 2009 International Conference on Security & Management, SAM 2009, July 13-16, 2009, Las Vegas Nevada, USA, 2 Volumes*, 2009, pp. 340–345.

[30] K. Gama and D. Donsez, “Towards dynamic component isolation in a service oriented platform,” in *Component-Based Software Engineering*, G. A. Lewis, I. Poernomo, and C. Hofmeister, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 104–120.

[31] N. Geoffray, G. Thomas, B. Folliot, and C. Clément, “Towards a new isolation abstraction for osgi,” in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, ser. IIES ’08. New York, NY, USA: ACM, 2008, pp. 41–45.

[32] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frenot, and B. Folliot, “I-jvm: a java virtual machine for component isolation in osgi,” in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, June 2009, pp. 544–553.

[33] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, “Service isolation vs. consolidation: Implications for iaas cloud application deployment,” in *2013 IEEE International Conference on Cloud Engineering (IC2E)*, March 2013, pp. 21–30.