2016 IEEE 9th International Conference on Cloud Computing

Model-Driven Management of Docker Containers

Fawaz Paraiso, Stéphanie Challita, Yahya Al-Dhuraibi, Philippe Merle

University of Lille & Inria Lille - Nord Europe
CRIStAL UMR CNRS 9189, France
Email: firstname.lastname@inria.fr

***Abstract*—With the emergence of Docker, it becomes easier to encapsulate applications and their dependencies into lightweight Linux containers and make them available to the world by deploying them in the cloud. Compared to hypervisor-based virtualization approaches, the use of containers provides faster start-ups times and reduces the consumption of computer resources. However, Docker lacks of deployability verification tool for containers at design time. Currently, the only way to be sure that the designed containers will execute well is to test them in a running system. If errors occur, a correction is made but this operation can be repeated several times before the deployment becomes operational. Docker does not provide a solution to increase or decrease the size of container resources in demand. Besides the deployment of containers, Docker lacks of synchronization between the designed containers and those deployed. Moreover, container management with Docker is done at low level, and therefore requires users to focus on low level system issues. In this paper we focus on these issues related to the management of Docker containers. In particular, we propose an approach for modeling Docker containers. We provide tooling to ensure the deployability and the management of Docker containers. We illustrate our proposal using an event processing application and show how our solution provides a significantly better compromise between performance and development costs than the basic Docker container solution.**

***Keywords*-Cloud Computing; Container; Docker; Model Driven Engineering; Models@runtime**

I. Introduction

Nowadays, with the emergence of Docker[1], lightweight containers are gaining in popularity and they are adopted by cloud providers such as Google, Azure, Amazon, and Digital Ocean. Containers are a lightweight solution that developers can use to deploy and manage applications. Indeed, compared to hypervisor-based virtualization where each Virtual Machine (VM) runs its own operating system (OS), which increases the use of system resources, the container technology has several advantages [10], [11]:

- *Low resource consumption*: containers share resources with host operating system, which make them more efficient. The stop and start actions on a container take a few seconds while they take minutes for VMs. Applications running in a container have small overhead compared to those running natively on the host OS.
- *Portability*: the portability of containers has the potential to eliminate whole kinds of bugs caused by subtle changes in the running environment and the vendor lock-in problem.
- *Lightweight*: the lightweight nature of containers permits developers to run dozens of containers at the same time, making it possible to emulate a production-ready distributed system. Operation engineers can run many more containers on a single host machine rather than using VMs alone.

Despite the advances in using Docker, the verification process related to the deployability of containers remains a challenging task. Currently, the only way to be sure that the designed containers will run correctly is to execute them in a running system. In this context, when errors occur, a correction is made and this operation can be repeated several times before the deployment becomes operational. Docker does not provide a solution to increase or decrease the size of container resources in demand. For example, at peak periods the container should scale resources up, and similarly on off-peak periods the container should release unneeded resources by scaling down.

Moreover, Docker is still lacking of supporting runtime systems evolution. In particular, how to affect changes on a deployed containers system? Besides the deployment of containers, Docker lacks of synchronization between the designed containers and those deployed. Container management with Docker is done at low level, and requires users to focus on low level system issues. Docker is lacking an explicit model representation of underlying containers and the relation between them. This makes it difficult to fix design errors, to fold new decisions into a running system in order to support controlled ongoing design. Obviously, there is a need to represent a human understandable description of some aspects of a running container. This can be represented in a form that can be mechanically analysed is relevant for many applications [16], as not only the content but also the context in which they were created, determine its value.

In this paper we propose a model to manage Docker containers. Our approach represents views of all aspects of Docker containers and are thus abstractions of executed phenomena. The model is not only used to design Docker containers architecture but also used to represent the con-

[1] http://docker.io

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.98

tainers deployed in the target systems. The designed and deployed containers need to provide views that are consistent with each other. Our approach therefore provides a high-level abstraction for Docker containers that is used for reasoning and managing large container deployments in the cloud.

The reminder of this paper is structured as follows. In Section II we give an overview of some background concepts we use in our proposal. Next, Section III presents the motivation of our work. Then, in Section IV we describe our model. Section V presents the validation of our solution. In Section VI we discuss some related work. Finally, we conclude our work in Section VII.

II. Background

In this section we give a brief introduction to some of the concepts and technologies about Docker that we use throughout this paper, in order to facilitate the understanding of this work.

A. Docker architecture

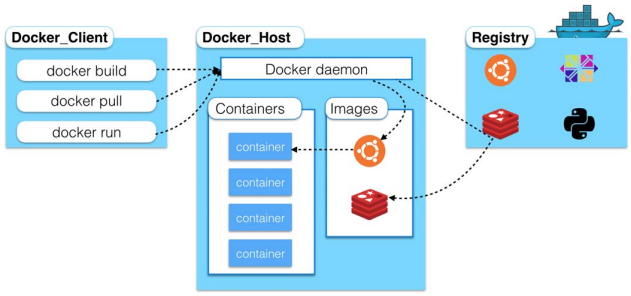

Figure 1. High-level overview of Docker architecture.

Basically Docker is a technology used for developing, deploying and executing applications packaged into containers. Docker defines a client/server architecture. In Figure 1, we can see the major components of a Docker installation.

- At the center, the Docker host represents the physical machine or VM in which Docker daemon and containers are deployed (cf., Figure 1). The Docker daemon is responsible for creating, running, and monitoring containers, as well as building and storing images. The launch of Docker daemon is normally handled by the host OS.
- The Docker client is on the left-hand side in Figure 1. It communicates with the Docker daemon via sockets through a RESTful API. The purpose of Docker client is to control the host, create images, publish, execute and manage containers corresponding to the instantiation of these images. Communication via HTTP makes it easiest the remote connections to Docker daemons. The combination of Docker client and Docker daemon is called Docker engine.
- Docker registry is on the right-hand side in Figure 1. It stores and distributes images. The default registry is the Docker Hub, which hosts thousands of public images. Docker containers are created using base images. A Docker image can include just the OS fundamentals, or it can consist of a sophisticated pre-built application stack ready for launching. To create an image, the most convenient option is to write a script file composed of various commands (instructions) named Dockerfile and then execute it. Many organizations run their own registry that can be used to store private images. The Docker daemon will download images from registry in response to requests.

B. Underline technologies

In this section, we provide some underline technologies tied to Docker.

Docker [10], [11] uses the existing Linux container technology and extended it through portable images. The Docker daemon uses an execution driver to create containers. The Docker containers are running by using a tool called RunC. RunC is very closely tied to the following kernel features:

- *cgroups*, which are responsible for managing resources used by a container (e.g., CPU and memory usage).
- *namespaces* are responsible for isolating containers; making sure that a container's filesystem, hostname, users, networking, and processes are separated from the rest of the system.

C. Surrounding technologies

In this section we provide some surrounding technologies supplied by Docker:

- *Compose* is a tool for building and running applications composed of multiple Docker containers.
- *Swarm* is a clustering solution. Swarm can group together several Docker hosts, allowing the user to manage them as a unified cluster.
- *Machine* provides and configures Docker hosts on local or remote resources.

III. Drawbacks of Docker technology

Despite the benefits [7], [10] Docker brings to teams, its adoption across enterprises has not been without issues. The problems with container adoption have little to do with the technology itself. Rather, they relate to organizational aspects that have not caught up to the technology [3]. The adoption of Docker in production is a real concern and was identify in [3], [5]. This section describes four key challenges that Docker containers should be faced: *Lack of verification*, *Resources management at runtime*, *Synchronization between design and execution environment*, and *Inconsistency use of containers across organization*.

- **Lack of verification**: Docker provides tools such as Docker Compose or Docker Swarm (cf. Section II) used

to design a set of containers connected together. However, once designed, the deployment of the containers can face several problems such as misconfiguration of links between containers, lack of resources on the hosts in which the containers are deployed, human errors, etc. Given the executable mechanism the Docker containers are related, the only way to be sure that the containers deployed will run or fail is to deploy them on the target executing environment. Moreover, there is no way to verify that deployed containers are conform with those designed. The lack of verification tool can become quickly painful and expensive when the deployment task is repeated several times.

- **Resources management at runtime**: when creating containers, Docker gives the possibility to set the resources (cpu, memory, disk, network) limits. In other words, Docker provides the possibility to set containers resources at design time. In the cloud environment, the container resources consumption fluctuates according to their embedded application workload. In order to provision the appropriate resources, if the workload grows or shrinks, the containers resources should be increased or decreased as required at runtime. Docker does not provide a mechanism to reconfigure the containers resources at runtime.
- **Synchronization between design and execution environment**: in Docker context, the execution environment consists of the Docker engine and the containers deployed. Conceptually, the deployed containers represent a predefined architecture. Thus, a major challenge is how to synchronize the predefined architecture of containers with the containers deployed in the execution environment. When modifications occur in an existing architecture, the update should be done in the executing environment. Conversely, when changes occur in the executing environment they should affect the existing architecture. A modification can be the addition of a new container, the retrieval of an existing container, the addition of a link between a new container with an existing one, etc.
- **Inconsistency use of containers across organization**: container adoption has not been a carefully planned and executed by companywide understanding and belief in its virtues [3]. Instead, individuals or small teams of developers have started using containers because they are fast and convenient, enabling them to respond to the increased pressure for quick turnaround coming from their business units and thus making their jobs easier. Each user relies on its familiar tools (e.g., Chef, Puppet, and Ansible) which are used for building and deploying containers. In this context, the problem of maintainability remains unsolved due to the heterogeneity of tools used by users.

This paper brings forward a solution to address these challenges, using a model-driven approach. This approach will allow Docker technology to have a complementary tool to take better advantage of containers in production environments.

IV. Approach

In this section we present our solution. We begin by giving an overview of the solution architecture and then we present how we model Docker containers. We also describe how the generation of the appropriate artifacts are done.

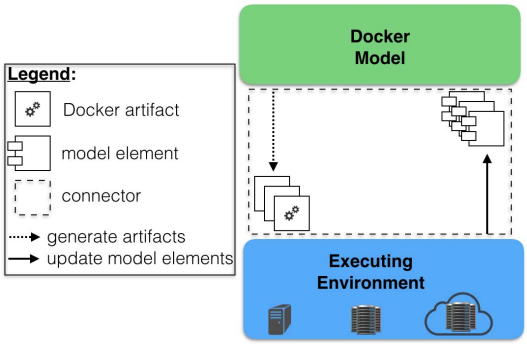

Figure 2. Architecture overview.

A. Architecture overview

To understand the concepts that rely under our architecture, we begin by giving an illustration of it in Figure 2. This architecture is composed of three parts: *Docker Model*, *Connector*, and *Executing Environment*. Conceptually, the architecture depicted in Figure 2 presents a *Docker Model* which provides an expressive model for containers. This model provides an appropriate abstraction of Docker containers (cf. Section IV-B for more details). The *Connector* defines the relationship between the *Docker Model* and *Executing Environment*. This *Connector* provides tools that are used not only to generate necessary Docker artifacts corresponding to the model actions (create, start, stop, restart, pause, unpause, kill), but also to operate efficiently to make online update for the *Docker Model* elements according to the changes in *Executing Environment*. Every artifact is handled in a seamless way thanks to the homogeneity provided by modeling principles. Finally, the generated artifacts are executing in the *Executed Environment*.

Our approach employs Model-Driven Engineering (MDE) techniques [1], in order to handle and analyze Docker containers at a higher level of abstraction compared to low level that the actual Docker solution provides. Using MDE techniques, our *Docker Model* describes explicitly certain concerns or certain views on an *Executing Environment* required to face the challenges discussed in Section III.

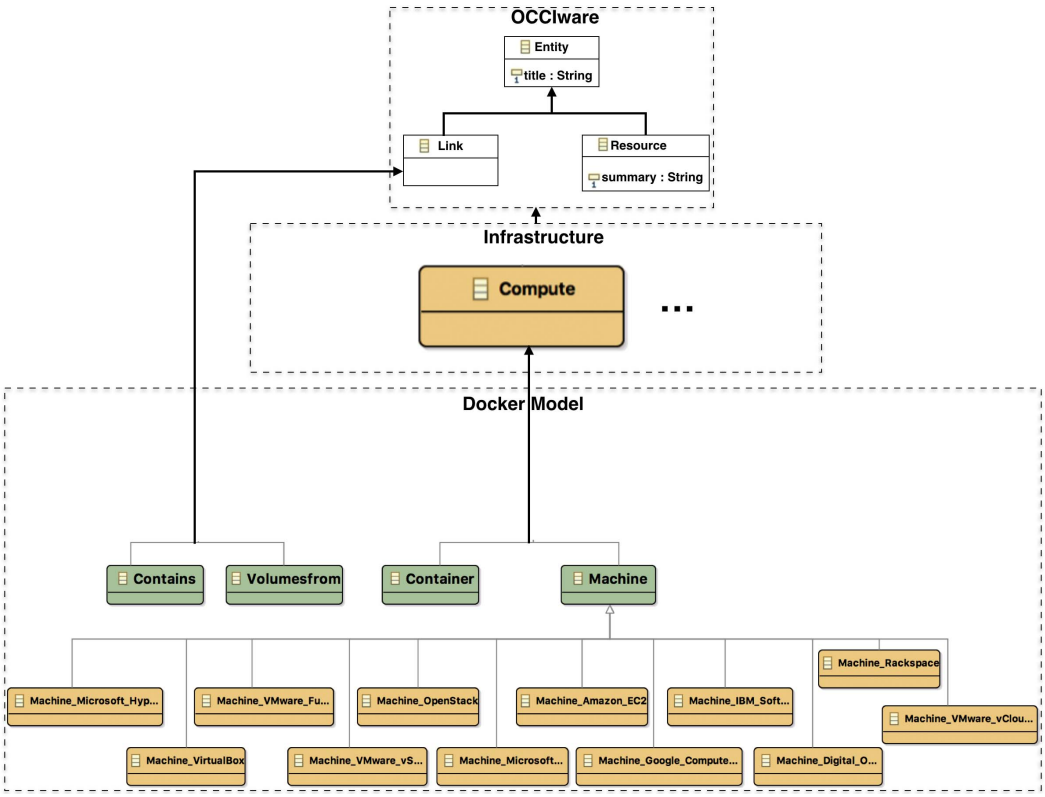

Figure 3. Docker Model.

B. Modeling Docker containers

This section describes how the modeling of Docker containers is achieved. Before we set out to design the *Docker Model*, we investigate to identify the requirements. We begin by examining the Docker containers with consideration of the main concepts, its structure and the relationship between each other. In this context, our model captures all necessary information related to the characteristics and management of Docker containers. This model is designed to be compliant with Docker containers. As depicted by Figure 3, our model is conceptually divided into three levels.

The top level represents the **OCCIware**[2] metamodel which is a precise metamodel of Open Cloud Computing Interface (OCCI)[3] [8], an OGF's specification defining an open interface for managing any kind of cloud computing resources (IaaS, PaaS and SaaS). The **OCCIware** metamodel is encoded with Eclipse Modelling Framework (EMF) [13].

The middle level named **Infrastructure** is based on our **OCCIware** metamodel. The **Infrastructure** model abstracts the cloud infrastructure resources (i.e, Compute, Network and Storage).

The bottom level represents our Docker Model, which extends the **Infrastructure** Model. In Figure 3, our Docker Model is simplified as it does not show, among others, attributes, and enumerations. Based on the **Infrastructure** Model, our Docker Model provides a comprehensive view on Docker containers. Building a Docker Model means thinking about structure of containers, their relationships with each other, and the hosts in which they are deployed. In our model, we explicitly provide a rich abstraction for describing, composing, and manipulating structured information related to the containers and the hosts in which they are deployed.

In the following, we present briefly the main concepts of our Docker Model:

- *Container* represents a Docker container. *Container* has a set of properties (name, image, command, etc.) related to a Docker container.
- *Link* is a relation between two container instances. *Link* references both source, and target containers, e.g, when containers are linked, information about source container can be sent to target container.
- *Machine* represents any physical or cloud VM that hosts containers. Here, MDE allows to factorize common pattern and reuse them. For instance, the class diagram of *Machine* is extended to describe the specificities of targets VM, e.g, *Machine_OpenStack* is an extension of

[2]http://occiware.org
[3]http://occi-wg.org/

Machine used to define the specificities (location, key, type of machine, etc.) of VM belongs to OpenStack.
- *Volumesfrom* represents a block storage that is attached to one or more container instances to persist data.
- *Contains* is used to define the relationships between *Machine* and *Container*, e.i, a machine contains zero to more container instances.

Our Docker Model is stored into a file in order to facilitate its reusability anytime and everywhere. The *Docker Model* provides the support for reasoning on architectural constraints of containers. In fact, to analyze architectural constraints, the Object Constraint Language (OCL) and checkers like EMF OCL[4] are used to define and check constraints that are attached to the model elements. For instance, among the constraints defined for the *Docker Model*, one constraint states that bidirectional or closed loop link is not permitted, etc. This is translated in OCL rule as:

```
context Container
inv NoCycleBetweenContainerLinks:
links->select(oclIsTypeOf(Link)).target->closure
(links->select(oclIsTypeOf(Link)).target)->excludes(self)
```

Listing 1. Descriptor for MyApp.

Unlike Docker solution, our model uses a constraint validator at **design time** to validate the constraints defined before the deployment. This validation guarantees the coherence of the containers and their relationships with each other. By allowing the use of constraint validator, our *Docker Model* provides a solution to the **Lack of verification** challenge identified in Section III. On the other hand, the use of an explicit model to represent containers allows us to address the **Inconsistency use of containers across organization** challenge identified in Section III.

C. Design tool for Docker

Our *Docker Model* comes with a tool to help designing, editing, and building custom views. We have implemented[5] this tool as Eclipse plug-ins, that can be downloaded from here[6]. We provide a friendly user graphical interface to assist users in modeling and deploying any containerized applications with Docker. This tool is called *Docker Designer*. *Docker Designer* abstracts all Docker concepts for designing, editing, validating, transforming, and deploying containers. Once an application embedded in a container is designed and validated, a user can register a cloud provider to automate the provisioning of VMs on ten different cloud providers: VMWare vSphere, OpenStack, Amazon Web Services, Rackspace, Microsoft Azure, DigitalOcean, HP Public Cloud, IBM SoftLayer, Google Compute Engine or local hypervisors such as Virtual Box and VMWare Fusion. This tool also allows us to import a running containers into our *Docker Model*, where model elements are represented graphically. A screenshot of the current *Docker Designer* is depicted in Figure 4. Frame (a) in Figure 4 shows the Eclipse Model Explorer used to navigate through a Docker project containing a *Docker Model*. Frame (b) in Figure 4 gives a perspective or a global view of the modeled containers. Obviously, this view can be adjusted to provide the most optimal perspective. Frame (c) displays the design area that provides a graphical representation of *Docker Model*. As shown in Frame (c), the model elements are **green** or **red**. The green color of machine or container elements shows the **started** state of containers and host machines. The red color shows the **stopped** state of containers and the host machines. Frame (d) in Figure 4 contains the Eclipse properties editor for visualizing and modifying attributes of a selected modeling element. All *Docker Model* elements displayed in Frame (c) can be setted through their properties. Frame (e) in Figure 4 displays the configuration pallet that represents the *Docker Model* elements such as: *container*, *link*, *volumefrom*, and *machine*.

Overall, our Docker Modeler assists users to design and deploy large containerized applications. This tool can be easily integrated into existing Docker environment. This integration can be done easily by importing the deployed containers as a well designed model on which users can reason and interact.

D. Synchronization tool for Docker

As described in the previous section, our *Docker Designer* tool is provided in order to assist users for modeling containers. Once modeled and validated, the containers will be deployed in an executing system. *Docker Connector* is a tool used to deploy and synchronize *Docker Models* with the *Executing Environment*.

Our synchronization mechanism is bidirectional . More precisely, the synchronization is done from *Docker Model* to *Executing Environment* and conversely. In the first direction (from *Docker Model* to *Executing Environment*), the *Docker Connector* updates the model changes into the running system by generating corresponding Docker artifacts (Docker Client commands, Docker Compose file, Docker Swarm configurations). These generated artifacts are used for online deployment. In the second direction, when the synchronization is performed from *Executing Environment* to *Docker Model*, the *Docker Connector* updates the model elements according to the *Executing Environment* changes. This connector performs this update by means of a partial reflection of the containers architecture using model introspection.

Our *Docker Connector* is implemented using Docker-Java API[7] to interact with Docker daemon though HTTP. This connector allows users to introspect an *Executing Environment* in order to build corresponding *Docker Model*, or

[4]https://wiki.eclipse.org/OCL
[5]http://tinyurl.com/dockermodeler, accessed at February 22th 2016
[6]http://www.obeo.fr/download/occiware
[7]https://github.com/docker-java/docker-java

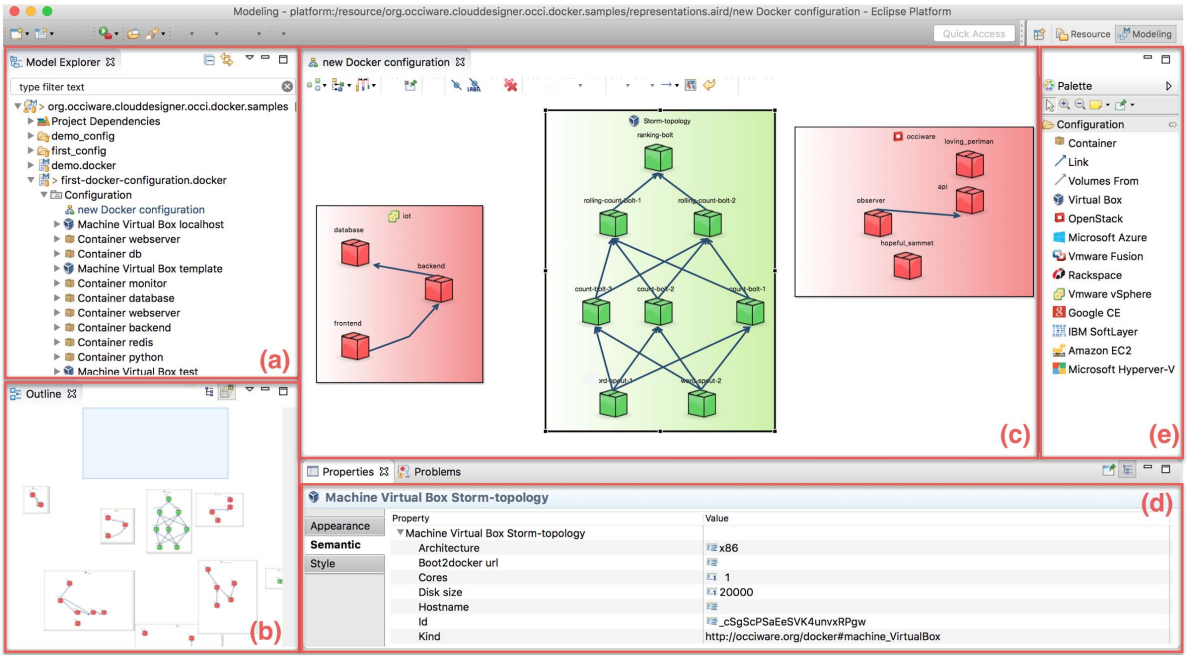

Figure 4. Docker Designer.

update an existing model and send changes back to the *Executed Environment*. To achieve appropriately the synchronization between the *Docker Model* and the *Executing Environment*, the *Docker Connector* checks first if the *Docker Model* elements are still consistent with the corresponding *Executing Environment* by navigating efficiently inside the model. If this is not the case, the *Docker Connector* re-establishes consistency by synchronizing containers states (started, stopped, etc.), container attribute values, adjusting links between container, deleting existing containers by new ones. Thus, our *Docker Connector* synchronizes the *Docker Model* and the *Executing Environment* incrementally. This addresses the **Synchronization between design and execution environment** challenge presented in Section III.

E. Connecting Docker Model online

To detect online modifications efficiently, we need to provide a mechanism that monitors both the *Docker Model* and the *Executed Environment*. To detect model modifications, the connector relies on a notification mechanism that reports events when a model element has been changed. To this, we use the EMF observer/listener design pattern. Concerning the changes which occur in the *Executed Environment*, the *Docker Connector* interacts directly with the Docker daemon which provides a callback hook mechanism that gets called whenever changes occur. In the online context, the *Docker Connector* only reacts to change notifications.

As discussed in Section III, Docker does not provide the possibility to modify the container resources (cpu, memory, disk, network) at runtime. To achieve this, our *Docker Connector* manages the Docker resources by manipulating their corresponding **Cgroups**. In fact, as described in Section II, **Cgroups** is a powerful tool for managing resources used by a container. Thus, this addresses the **Resource management at runtime** challenge presented in Section III.

V. Validation

In this section we evaluate our approach with respect to both performance and online model manipulation achievement.

A. Performance evaluation

To evaluate our *Docker Model* in cloud environments, we run a distributed containerized application composed of 8 containers. This distributed application is a computation system for processing large volume data. To focus on the real performance of our *Docker Model*, all our experiment were performed using Scalair[8] private cloud provider with twenty virtual machines (VMs), that uses VMware to build their cloud infrastructure. The configuration of each VM is: 1 VCPU, 20 G0 of Disk, 1Go of RAM, Ubuntu 12.04 Linux 3.13.0-66-generic. Our Docker Designer is running using a Macbook Pro workstation with 2,2 GHz Intel Core i7 processor, 16 Go 1600 MHz DDR3, OSX version 10.11.2 (15C50), and Oracle Java 1.7.

To evaluate our solution, we answer the following questions: (i) Does *Docker Model* introduce overhead? (ii) How much time is taken to manipulate *Docker Models* online? (iii) Does our *Docker Model* scale?

[8]http://www.scalair.fr

1) Overhead introduced by the Docker Model: To determine the overhead introduced by our *Docker Model*, we evaluate two of the scenarios where our distributed containerized application were created, started, and stopped: i) natively with Docker, and ii) with Docker integrating our model. The scenario was executed hundred times on each of the both implementations.

In Tables I, II and III, we present the results of the average time for creating, starting, and stopping of height containers for each implementation, as well as the mean overhead introduced by the *Docker Model*.

The overhead introduced by our model when creating containers is **1.11%**, this creation phase consists of pulling image once and the creation of the containers. Next, when starting the containers, the overhead introduced by our model is **2.12%**. Then, the overhead introduces by our model when stopping containers is **2.25%**. The small overhead fluctuation of the start and stop actions compared to the creation action is due to the model elements manipulation.

Table I
CONTAINER CREATING TIME AND OVERHEAD.

Start action	Avg. start time	Docker Model overhead
Docker	168.509 sec	-
Docker with Model	170.382 sec	1.11%

Table II
CONTAINER STARTING TIME AND OVERHEAD

Start action	Avg. create. time	Docker Model overhead
Docker	5.033 sec	-
Docker with Model	5.04 sec	2.12%

Table III
CONTAINER STOPPING TIME AND OVERHEAD

Stop action	Avg. stop time	Docker Model overhead
Docker	84.12 sec	-
Docker with Model	86.01 sec	2.25%

This experiment shows that there is an overhead introduced by adding the *Docker Model*. The overhead is negligible regarding all advantages provided by our approach: verification of containers, increasing resource at runtime (cf. Section III).

B. Online model manipulation

When manipulating our model, we have evaluated the time taken by our Docker Model to detect and integrate the changes inside the model. In this context, we have modified the model and evaluate the time taken by the Docker connector to propagate changes in the *Executing Environment*. Conversely, we operate modifications in *Executed Environment* and evaluate the time taken by Docker Connector to operate the changes in the model.

When creating a new container from *Executed Environment*, our Docker Connector takes about **12** milliseconds to detect the changes and spends about **290** milliseconds to graphically represent a container into the model. Next, when creating a new container in the model, it took about **10** milliseconds for the Docker Connector to detect the changes. This experiments shows that our Docker Connector reacts quickly to changes.

C. Scalability of model-handling at runtime

Model manipulation at runtime, as opposed to design time, is subject to the same efficiency requirements as the rest of the system because the execution of model operations impacts overall system performance. To validate whether our proposed approach scales to large systems, we quantified this overhead for randomly updated and created large model elements. These models started with fifty elements (containers) and were populated with 50 new elements in each iteration. After model population, we evaluated the time taken to generate model elements. Even with 50,000 elements in the model, the average time taken to generate each iteration of 50 elements was **14.30** seconds, which is acceptable compared to the minimum time which is **12.02** seconds and the maximum time which is **16** seconds. As shown in Figure 5, the generation time of 50 elements (containers) is deterministic using our **Docker Model**.

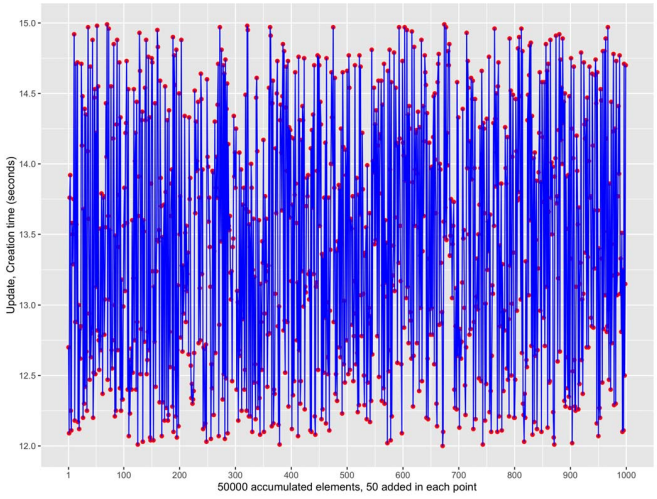

Figure 5. Time taken to update and create model elements.

VI. RELATED WORK

In this section we will present some of the related work from different fields of research that are relevant to our approach.

Authors in [6] proposed a control architecture that dynamically and elastically adjust VMs and containers provisioning. However, their work did not provide any solution to verify the deployability of Docker. In this work, we tackle the issue of deployability verification by using an approach based on Model Driven Engineering. In contrast to the existing solutions, our solution checks constraints

expressed in OCL which allows users to verify and guarantee containers deployability at design time.

In the cloud computing context many works [12], [15], [2], [14] have addressed resource management at runtime with hypervisor-based solutions. A major difference between their solution and ours is the baseline containerization technology adopted.

Model-driven approaches [16], [9] considering runtime models, in contrast to ours, do not work incrementally to maintain those models or they provide only one view on a managed system.

The runtime model in [4] is updated incrementally. However, it provides a view focused on the configuration and deployment of a system, but no other information, e.g., regarding management. All these approaches do not consider the transformation of models specified by different metamodels [16], [9].

VII. Conclusion

This paper presented our approach for the model-driven management of Docker containers. It enables the verification of containers architecture at design time. The synchronization between Executed Environment and our model can be done incrementally. Our approach leverages the use of MDE for managing Docker containers combined with model synchronization techniques at runtime.

We designed a graphical model-driven tool chain called **Docker Designer** to design, reason, and deploy containers. We also provide model-based generators that target Docker commands, Compose, Swarm using Docker Connector.

Most existing solutions do not allow to verify the deployability of containers at design time. Instead of performing the verifications of deployability of the containers at execution time, we propose the use of constraint verifications using our Docker Model at design time which avoid to lose time and save development cost.

As future work, we will investigate the adaptation of managed containers architecture, as a set of atomic changes might have to be performed. Moreover, extending our approach to other container solutions will be considered.

Acknowledgment

This work is supported by OCCIware (www.occiware.org) research and development project funded by French Programme d'Investissements d'Avenir (PIA).

References

[1] J. Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.

[2] J. Cáceres, L. M. Vaquero, L. Rodero-Merino, Á. Polo, and J. J. Hierro. Service scalability over the cloud. In *Handbook of Cloud Computing*, pages 357–377. Springer, 2010.

[3] J. F. Consulting. Maximize Container Benefits With A Top-Down Approach. Website http://tinyurl.com/redhadContainers, April 2015.

[4] J. Dubus and P. Merle. Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-based Systems. In *Workshop Models @ Runtime, in conjunction with MoDELS / UML 2006*, pages 242–251, Gênes, Italy, Oct. 2006.

[5] A. Gerber. The State of Containers and the Docker Ecosystem. Website http://offers.ruxit.com/rs/987-BEQ-874/images/State_of_Containers_Ruxit_compressed_V2.pdf, 2015.

[6] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete. Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers. In *Service-Oriented Computing*, pages 316–323. Springer, 2015.

[7] M. Janakiram and M. Caroline. Is Docker a threat to the Cloud ecosystem? Website https://research.gigaom.com/2014/08/is-docker-a-threat-to-the-cloud-ecosystem, 2014.

[8] P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, and S. Tata. A Precise Metamodel for Open Cloud Computing Interface. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pages 852–859, 2015.

[9] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@run.time to support dynamic adaptation. *Computer*, 42(10):44–51, Oct. 2009.

[10] A. Mouat. *Using Docker Developing and Deploying Software with Containers*. O'Reilly Media Pub., 2015.

[11] H. Philipp, W. Ingo, S. Stefan, Z. Liming, and F. Alan. Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers. In *Service-Oriented Computing - 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings*, pages 316–323, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[12] B. P. Rimal, A. Jukan, D. Katsaros, and Y. Goeleven. Architectural requirements for cloud computing systems: an enterprise cloud approach. *Journal of Grid Computing*, 9(1):3–26, 2011.

[13] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[14] H. N. Van, F. D. Tran, and J. M. Menaud. Performance and power management for cloud infrastructures. In *2010 IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pages 329–336. IEEE, July 2010.

[15] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[16] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker. Incremental Model Synchronization for Efficient Run-time Monitoring. In *Proceedings of the 2009 International Conference on Models in Software Engineering*, MODELS'09, pages 124–139, Berlin, Heidelberg, 2010. Springer-Verlag.