Analyzing and Improving the Performance of Continuous Container Creation and Deployment*

Ahmad Alamoush, Holger Eichelberger
{alamoush,eichelberger}@sse.uni-hildesheim.de
University of Hildesheim, Hildesheim, Germany

Abstract

Continuous Deployment automates the delivery of new versions of software systems. To ease installation and delivery, often container virtualization is applied. In some applications, container images may be subject to variants, as, e.g., device-specific software is needed on Edge devices in Industry 4.0. Here, model-driven approaches can prevent human errors and save development efforts. However, employing a naive approach, creating one container image per variant can be time-consuming. In this paper, we discuss the impact of different (Docker) container image creation techniques for variant-rich Industry 4.0 applications. Our results show that a combination of techniques like container image stacking or semantic fingerprinting can save up to 59% build time and up to 89% deployment time, while not affecting the container startup time.

1 Introduction

Continuous Deployment automates and reduces the time of software systems delivery. Using container virtualization in continuous (incremental) deployment eases the installation and delivery of software systems to different environments. Depending on the application, the images[1] may be subject to variants to consider customizations, user decisions or device requirements, e.g., for Industry 4.0 edge devices. To handle variants efficiently while preventing human errors and saving development efforts, a model-driven generative approach can be a solution. However, using a naive approach may create an image for each variant which can be time and resource consuming.

In this context, we ask: *Can we improve and measure the development time performance of creating (Docker) containers by advanced image creation techniques?*. In [1], Shahin et al. present a systematic literature review on Continuous Integration, Delivery, and Deployment. The authors identified 30 approaches that facilitate the implementation of Continuous Integration. Non of those use containerization. Park et al. [2] add a replaceable dynamic layer in an image to reduce boot-up time during updates. Instead of redeploying a new container, they pause the container, exchange the dynamic layer and resume the container. We also rely on layering, but to improve build time. Moreover, we are not aware of comparable approaches to model-driven container creation.

In this paper, we analyze the effects of two techniques, namely image stacking to increase the reuse among similar containers and semantic fingerprinting to avoid unnecessary image creations. To identify performance effects, we conduct an experiment with two service-based applications from the IIP-Ecosphere project [4], where we create container images in a model-driven fashion. In particular, we show that a combination of the two techniques can save up to 59% build time and up to 89% deployment time.

In Section 2, we present image stacking and semantic fingerprinting as techniques to improve container creation. In Section 3, we detail our experiment and discuss the results. Finally, we conclude with some outlook on future work.

2 Approach

In this section, we discuss the context of our work, i.e., usual and model-driven container creation and introduce two techniques that we apply to improve container creation time, namely image stacking through base images and semantic fingerprinting.

Usually, (Docker) images are created by stacking container layers on top of each other. Each layer adds software, libraries, dependencies, files, etc. A **naive approach** just adds all layers of a container to a single image, in the extreme case without placing reusable elements into a lower layer. Changing a single file, e.g., as part of an update, forces the re-creation of the image, and, potentially of almost all contained layers. Although Docker detects file changes and triggers the creation of only needed layers, it may perform unneeded operations if only timestamps, e.g., in a JAR file or a Python library installation have been changed, while the actual software remained the same. Thus, a naive approach usually leads to superfluous builds and unnecessary time consumption.

A **model-driven approach** to container creation

*IIP-Ecosphere is partially supported by the German Federal Ministry of Economic Affairs and Climate Action (BMWK) under grant 01MK20006D

[1] We use "image" as short form of "container image" and denote with "container" a running container.

employs a detailed model of the applications to be virtualized, e.g., including required software dependencies or operating system tools, to create containers automatically. In our case, we focus on service-based applications for an AI-enabled Industry 4.0 platform [3, 4], where the model specifies the required Maven/Python dependencies per Java/Python service, the needed platform services to start services remotely, or the required Java or Python version. Moreover, as Industry 4.0 edge devices are rather heterogeneous in nature, we also capture device specific information, e.g., monitoring plugins or whether a custom build of libraries are already such as TensorFlow are required. We focus on static ahead-of-time images for each application to detect build and dependency errors before applications are deployed to a shop floor. In other words, we avoid (risky) time- and bandwidth intensive on-site builds per individual (edge) device.

We apply the following two techniques to reduce the build time:

- **Base image / image stacking:** Create base images with common layers shared among images and applications. Base images are stacked on the operating system as start image, while further application-specific layers are stacked on top. Thus, images for common layers are created once, saving time in building, publishing / pushing, and deploying a container.

- **Semantic fingerprinting:** Employ additional knowledge from the model regarding the files used in a container to identify, whether relevant content has been changed. We store MD5 checksums as fingerprints of the files used during the last build of an image. Relevant files may be source files including Python dependency lists, binary files or archive files (based on their contained files). This approach avoids image rebuilds if only timestamps in Java JAR files or installations of Python libraries have changed.

3 Experiment and Results

We are interested, whether the two techniques introduced in the last section can outperform a naive build with respect to build, publication, deployment/installation and container startup time. In this section, we discuss an experiment based on two application and measure the respective execution times. Then, we discuss the results.

Experimental environment: We used two VMware virtual machines (VM), each with 2 virtual CPU cores, 4 GB of RAM, and Linux Ubuntu Server 20.04 installed. For containerization, we use Docker version 20.10.7. One VM acts as build/distribution Server and hosts a local docker registry, the other VM plays the role of a device, i.e., a deployment target.

Experimental subjects: We use the following two representative service-based applications from the

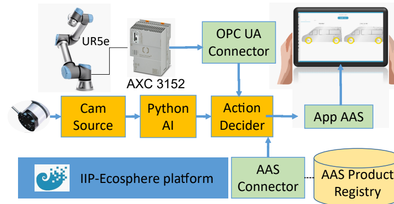

Figure 1: HM'22 application services [3]

IIP-Ecosphere project to measure and compare our approach. Each application is defined in a separate configuration model.

- The *Test application (TA)* defines a simple application consisting of with three linearly connected Java services. We use this application for regression tests of our platform.

- The *HM'22 application (HA)* was used as a platform demonstrator at the Hannover fair 2022 [3, 4]. As shown in Figure 1, HA consists of five Java services and a Python AI service and performs a visual quality inspection of model cars.

In addition to the application services, a container may include a Java VM, a Python installation, required dependencies as well as IIP-Ecosphere platform services to manage the application services [4] as well as a (local) communication broker. Both applications are specified in terms of a topological configuration model, which also defines the base characteristics of the involved (edge) devices.

Depending on the settings in the configuration model, the container creation either employs the naive approach or the base image technique, which may be combined with the semantic fingerprinting. In general, the container creation for HA stacks python:3.8-slim-buster[2] as start image, a JDK, the IIP-Ecosphere platform services, the Python dependencies of the application, the application services, device-specific services and a startup script. For TA, the container creation replaces start image with Alpine:3.18[3] as no Python is needed. If the base image technique is enabled, for the HM'22 application a reusable image with Java, Python, IIP-Ecosphere services and Python dependencies is created. The base image for TA is similar, but without Python dependencies. As we use separate models for the applications, there images are not shared among the applications. The configuration models specify two types of devices and thus, cause, the creation of two (differing) images per application.

Experimental procedure: We apply five treatments: naive approach, base image without fingerprinting and fingerprinting triggering the creation of three different layers with different impact. The layer

[2] https://hub.docker.com/_/python
[3] https://hub.docker.com/_/alpine

triggers are in order of decreasing impact a python dependency change, a Java platform service change and a configuration setting change just affecting some resource files. In the experiment, we disable the Docker build cache to avoid any mutual influence.

On the server VM, we measure the sums of the building and publishing time for all images per application. On the device VM, we measure the installation / pulling time of the images, the time needed to create containers from the images as well as the startup time of the containers. We ran the experiment five times and calculated the average.

Test application (3 services)					
	Server Side		Client Side		
Treatment	*Build*	*Publish*	*Pull*	*Create*	*Start*
Naive	235	32	13	0,620	0,280
Stacked	302	20	8	0,391	0,224
Python	0	0	0,157	0,101	0,244
Java	190	9	4	0,378	0,237
Model	192	10	5	0,436	0,240
HM'22 (7 services)					
	Server Side		Client Side		
Treatment	*Build*	*Publish*	*Pull*	*Create*	*Start*
Naive	425	87	41	0,607	0,279
Stacked	394	51	20	0,397	0,246
Python	389	46	21	0,385	0,233
Java	176	10	4	0,384	0,240
Model	175	10	4	0,400	0,242

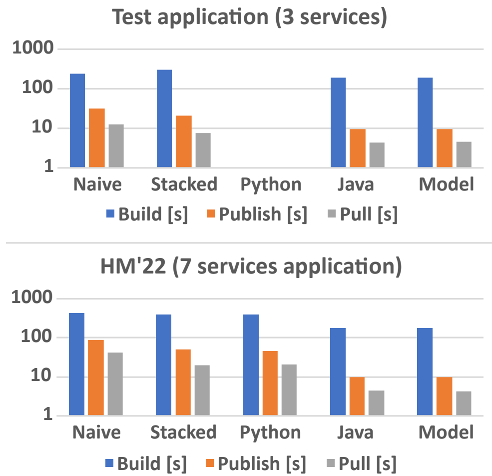

Figure 2 & Table 1: Experimental results in seconds.

Results: At a glance, the naive approach is faster for the TA than the base image technique as shown in Figure 2. This is due to the small image size where creating two images instead of one causes measurable overhead. Here, base images improve publishing and pulling time by 35% and 39%, respectively. For HA, container building, publishing and pulling time are improved by 7%, 41% and 51%, respectively.

The following treatments trigger different forms of layer re-creation for the respective stacked image. As the TA does not contain Python services, the container creation is skipped here. For HA, the container creation re-builds the base image including the shared Python dependencies, i.e., the measures are similar to the base image technique. For the remaining two changes, in particular the re-creation of the base image is skipped and only the required layers are rebuilt. For the TA, this leads to an improvement of 19%, 70%, and 65% improvement for build, publishing and pulling time, respectively. For HA, we measured a speedup of 59% improvement for build time and 89% for publishing and pulling time.

The differences for container creation and startup for all treatments are relatively small compared with container building, publishing, and pulling. Here, the improvement in container creation and startup time is around 0,2 and 0,04 seconds, respectively.

Limitations: Our results are coupled with the notion of service-based applications in IIP-Ecosphere and that reuse effects may only be apparent for multiple applications in the same model/platform.

4 Conclusion

Container virtualization eases the delivery of software in Continuous Deployment. Variant-rich software applications may require multiple containers, which can be efficiently realized by a model-drive approach.

In this paper, we present a model-driven approach to container creation and compare techniques to reduce the build, publishing, installation time. In applications with a larger number of common container layers, our techniques can improve the build time by up to 59% and deployment time by up to 89%. In applications with a smaller size of common layers, our techniques may affect the build time while still improving the deployment time up to 39%.

In the future, we plan to refine the techniques, e.g., to create multiple base images to facilitate the reuse among applications specified in the same model.

References

[1] M. Shahin, M. Ali Babar, and L. Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5 (2017), pp. 3909–3943.

[2] J. Park et al. "A Method of Dynamic Container Layer Replacement for Faster Service Providing on Resource-Limited Edge Nodes". In: *International Conference on Electronics and Communication Engineering (ICECE)*. 2019, pp. 434–437.

[3] H. Eichelberger, G. Palmer, and C. Niederee. "Developing an AI-enabled Industry 4.0 platform - Performance experiences on deploying AI onto an industrial edge device". In: *Softwaretechnik-Trends* (2023), pp. 35–37.

[4] H. Eichelberger et al. "Developing an AI-Enabled IIoT Platform - Lessons Learned from Early Use Case Validation". In: *Software Architecture. ECSA 2022 Tracks and Workshops*. 2023, pp. 265–283.