Westfälische Wilhelms-Universität Münster

Topic:

Prototypical Development of a
Docker-based Workflow Management System

Masterthesis
in the subject
at the Department of Information Systems — Practical Computer Science

Supervisor: Prof. Dr. Herbert Kuchen
Tutors: MScIS Vincent von Hof

Submitted by: Lars Greiving
Dettenstraße 4
48147 Münster

+49-176 704 253 17
l_grei02@uni-muenster.de

Deadline: 2016-02-24

Contents

List of Figures

List of Tables

Abbreviations

API	application programming interface
cgroups	control groups
CoW	copy-on-write
OS	operating system
PID	process identifier
REST	Representational State Transfer
OASIS	Organization for the Advancement of Structured Information Standards
MSA	Micro Services Architecture
SOA	Service-oriented Architecture
WFMC	Workflow Management Coalition
WfMS	Workflow Management System

1 Introduction and Motivation

- wfms are relevant - containers solve things - - what can be gained from combining them?

1.1 Related Work

» Nunamaker, Chen and Purdin

2 Workflow Management Systems

In this chapter, the concepts of workflows and workflow management systems will be briefly introduced and related to each other. There is a plethora of term definitions and deviating understandings of workflows and the concepts related to them [3]. In large parts, the concepts presented here thus rely on specifications published by the Workflow Management Coalition (WFMC), a consortium of workflow management software vendors, researchers in the field of workflow management and Workflow Management System (WfMS) users, as they represent some form of consensus.

The identified use cases and properties will be used in 4.1 to identify objectives for the architecture. Also, they will be the reference to which the final architecture developed in this thesis is compared against.

2.1 Concepts

2.1.1 Workflow

In order to achieve their business goals, organizations perform temporal and logical sequences of tasks that help to interact with business relevant entities. These sequences are known as *business processes*. If the logic that controls the processes is performed in an automated way, e.g. by an information system, one refers to the processes as *workflows* [2, 11]. The WFMC defines workflows as the computerized facilitation or automation of a business process, in whole or part [11].

Process activities are the atomic steps that processes consist of. The WFMC differentiates between *manual activities* and *workflow activities*. The former are activities that involve user interaction in order to be completed, while the latter are automated and require no interaction [11]. As the term “workflow activity” might be misunderstood as “any activity belonging to a workflow”, in the following the term *automated activity* will be used instead.

2.1.2 Process Definition

In order to be able to execute workflows, the underlying business processes must be machine processable and thus have to be formalized from real world to an abstracted model [11]. This model is usually called *process definition* and stored in form of some high-level programming language construct [11, 18]. The process definitions typically consist of a collection of activities with additional metadata such as associated applications or participants, and a set of rules which determine the execution order of these activities [11]. They further may contain references to other processes, which are treated as a single activity in the process definition [11, 3].

- usually directed grpah - how stored?

2.1.3 Process Instance

A *process instance* is an enactment of a process definition. A process definition may be instantiated multiple times, even at the same time. [3]. If only the automated parts of such an instance are meant, the WFMC advocates for the term *workflow instance* [11].

Process instances have several states. When they are created, they are in the *initiated* state. In this state, all relevant data has been provided, but the execution has not yet begun, e.g. because not all requirements are met. When the process is started, it enters the *running* state and it's activities may be started according to the process definition. If it has one or more instanciated activities, a process instance is in the *active* state. Process instances may be suspended, i.e. they enter the *suspended* state and no activities are instanciated until they leave it again. There are two states that a stopped process instance can be in. Either the completion requirements are met and the stopped process instance is in the *completed* state. Or the process instance stopped before its regular end, i.e. because of an error or manual interruption. In this case the process instance is in the *terminated* state [11].

A graphical representation of the state transitions described above can be seen in figure **??**. In this depiction, the allowed transitions between the different states are easy to grasp.

2.1.4 Activity Instance

Just like processes, activities are instanciated during workflow execution and have a set of states that they may be in.

When an activity instance is created, it is in the *inactive* state. From this state, it may enter the *suspended* state, in which it will neither be activated nor assigned a worklist item. If the activity instance is not suspended, it is activated once its entry conditions are fulfilled. It then is in the *active* state. When the execution of the activity has finished, it finally enters the *completed* state [11].

The possible transitions between the activity instance's states can be seen in Figure **??**.

2.1.5 Workflow Data

In a WfMS, several forms of data may occur at diverse occasions. The WFMC differentiates between three types of data: workflow relevant data, workflow application data, and workflow control data [11].

WfMSs use *workflow relevant data* to determine a process instance's status and the next activity to be executed. It is normally available to the WfMS and both process- and activity instances [11].
Applications that are part of an workflow may work on domain specific data, which is called *workflow application data*. In most cases, the WfMS does not interact with this data other that providing it to the respective applications and limit access to it according to some authorization rules [11, 3].
Data that is internally managed by a WfMS is refered to as *workflow control data*. This data usually comprises the states of process- and activity instances and other internal statuses and is per se not interchanged in its default form [11, 3].

2.1.6 Workflow Participant and Worklist

There are workflows that contain activities which require user interaction. A WfMS thus provides the functionality to assign workflows and activities to workflow participants. This assignment can either be a specific one, targeting one single person, or be more general, targeting a set of users from which whe WfMS may choose during execution time. These sets are usually based on an organizational structure that manifests itself in roles, of which an user may have one or more [11, 3].

Each user has a so called *worklist* that consist of activities to which he is assigned to and which are scheduled for execution. Depending on the actual implementation, activites may appear on multiple users' worklists until one of them signals that he/she will work on it [11, 3].

2.2 Typical Architecture

With a growing number of workflows in an organization, the need arises to manage their creation, distribution and execution in a structured manner. An information system is called WfMS, if it is able to define, create and manage the execution of workflows by using software that runs on one or more workflow engines, is able to interpret process definitions, can interact with involved participants, and may invoke external applications [13]. According to the WFMC, a workflow management system is "a system that defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic" [11].

In the following, the typical foundations of WfMSs architectures identified by the WFMC are presented and related to the concepts introduced in Section 2.1.

2.2.1 Functional Areas

The WFMC divides the reponsibilities of a WfMS in three functional areas: *build-time* functions, *run-time process control* functions and *run-time activity interaction* functions [11, 1].

The *build-time* functionalities are concerned with the abstraction of workflows, i.e. the creation of process definitions.
The *run-time process control* functionalities of a WfMS are dealing with instanciating and controlling processes, coordinating the execution of activities within a process instance, initiating (but not performing) both participant interaction and application invocation [11].
Some activites require users to enter data or applications to perform a specific task. The *run-time activity interaction* functions of a WfMS provide the possibilities to do so. They make forms available to users, instruct other applications, and collect the respective outcome [11].

2.2.2 System Components

The WFMC identified four software components that most WfMSs have in common: *Process Definition Tools*, *Administration and Monitoring Tools*, *Workflow Client Applications*, and *Workflow Enactment Service* [11].

Process Definition Tool

Process definition tools are responsible for analysis, modelling, description and documentation of business proceses. The output of process definition tools – process definitions – can be interpreted by workflow engines in order to enact the respective workflow.

The WFMC notes, that process definition tools do not necessarily have to be part of a WfMS, since the definition may take place in another tool as long as it is passed along in a standardized format [11].

Administration & Monitoring Tools

The administration and monitoring tools are responsible for high level monitoring and control of the system. Their functionalities may include user management, role management, logging, performance auditing, resource control, and supervision over running processes.

Workflow Client Applications

The core function of the workflow client applications is to let the user retrieve worklist items that were assigned to him/her. In the WFMC reference model they are thus sometimes referred to as *worklist handlers* [11].
Yet, the WFMC stresses that their functionality may be much broader, e.g. letting him/her enter data that is associated to one worklist item, allow him/her to alter the worklist, signing in or off, or control the processes' statuses. The WFMC thus advocates for the term *workflow client applications* [11]. The user interface may be part of the workflow client applications or exist as a separate software component.

Workflow Engine

Workflow engines provide the runtime control environment for the execution of workflow instances, that is, they interpret the process definition, manage the instances' status, update worklists, determine participants, and invoke external applications. They further manage the storage and flow of workflow control data and workflow relevant data [11].

Workflow Enactment Service

The Workflow Enactment Service groups one or more workflow engines into one logical component that exposes a single coherent external interface to other software [11].

2.2.3 WFMS Implementation Structure

According to the WFMC, the components described in 2.2.2 interlock in order to provide the overall functionality of a WfMS. As visible in **??**, the workflow enactment service plays a central role in wiring the components together.

3 Docker

When multiple applications or application instances are intended to run on one physical machine without interfering with each other, they are usually isolated in terms of execution environments and provided with a controllable share of system resources [8]. These goals can be fulfilled by both virtual machines and software containers [15]. The difference between these two options and the basic principles of software containers are shown in 3.1.

Docker is a tool, that aims at simplifying software container creation and management. In Section 3.1 its underlying concepts will be presented. Based on that, the functionality that Docker provides will be explained in Section **??**. Finally, the Docker ecosystem, i.e. the set of tools that enhance the core docker tool, is introduced in Section 3.3.

3.1 Concepts

First, the concept of software containers will be presented and contrasted against the concept of virtual machines. This is necessary to understand *what* Docker does. Then, internal constructs of Docker – images, containers, data volumes, dockerfiles, registries and repositories – are explained, in order to provide an understanding on *how* Docker does what it does.

3.1.1 Virtualization and Software Containers

The goal of *virtualization* is to simulate the presence of multiple computers on one machine. The use of this is XXX. There are two kinds of virtualization, one that takes place on the hardware level and another that takes place on the operating system (OS) level [15].

Hardware-level virtualization

In most cases when speaking about virtualization, *hardware-level virtualization* is referred to. It is usually driven by a *hypervisor* – a service that manages virtual machines and provides them with abstracted hardware devices to run on. This hypervisor either either runs in the OS of the host machine or directly on its hardware [15].
The virtual machines, i.e. the computers simulated on the host machine, require their own OS to be installed.

OS-level virtualization – or container-based virtualization

The other kind of virtualization, *OS-level virtualization*, is the one that Docker makes use of. It utilizes functions of the host kernel which allow the execution of several isolated userspace instances that share the same kernel, but may differ in terms of their runtime environment, e.g. file system or system libraries. These isolated userspace instances are usually called *software containers* or just *containers*. This type of virtualization is therefore also referred to as *container-based virtualization* [15].

The isolation and resource management in container-based virtualization on Linux systems are mainly achieved by two mechanisms, *control groups (cgroups)* and *namespaces*. While the former allows to group processes and manage their resource usage, the latter can be used on many system components. Namespaces may be introduced for example on network interfaces, the file system, users and user groups, process identifier (PID)s, and other components, in order to achieve a fine grained control over the respective isolation [15].
Besides Docker, there are several solutions that are all based on the aforementioned kernel features, e.g. LXC, LXD, lmctfy, systemd-nspawn, etc [15]. There are ongoing efforts to create a common container standard [12].

Many container solutions rely on a strategy called *copy-on-write (CoW)* to provide a runtime enviroment, which on the one hand lets the containers reuse system libraries and the like while on the other hand limits the container in affecting its surroundings [5, 14]. This strategy is explained in a more detailed fashion in 3.1.2 on the example of Docker.

3.1.2 Docker Images and Containers

CoW is a strategy which makes use of the benefits of both sharing files for read access and copying them to a local version previous to changing them. Processes that require access to a file share the same instance of that file. As soon as one process needs to alter the file, the operating system creates a copy to which only the process has access to. All other processes still use the original file [14, 5].

Docker images (referred to as just *images* from here) are the basis for Docker containers. Each image consists of a sequence of layers, where each layer summarizes one CoW step, i.e. the alterations to the file system that one command causes compared to the previous layer. Each layer is uniquely identifiable, which allows the same layer to be used by several images.

Docker containers are runtime instances of images. In the context of storage, a Docker container

can be considered as an image, i.e. a set of read-only layers, with a writable layer on top of it – the *container layer*. Write operations within a container trigger a CoW operation which copies the targeted file to the container layer, where the write operation is then performed.
Besides reducing the amount of space consumed by containers, the CoW strategy also reduces the time required to start a container. This is because Docker only has to create the container layer instead of providing a copy of all the filed contained in the respective image [5].

- *lifecycle of a docker container here*

3.1.3 Data Volumes

Any data written to the container layer is deleted as soon as its Docker container is deleted. Also, Docker containers that store a lot of data are considerably larger than Docker containers that do not, since the write operations require space in the container layer. This is the reason why data volumes exist – they are designed to persist data. Data volumes are directories or files that are mounted directly into a Docker container and thus bypass the storage driver [6]. They are never deleted automatically and therefore must be cleaned up manually when they are not needed anymore [5].

3.1.4 Dockerfiles

Instead of manually creating a container, running commands on it and then commiting it to create an image, Docker can be instructed by a recipe file – the *dockerfile*. In this file, the user states an image that the new image should be based on and the commands that otherwise would be entered manually [6].
To build an image, Docker is given a Dockerfile and a directory with files required for the build, the *context*, which is usually the directory the Dockerfile is located in. This enables Docker to copy files from the context to some layer within the image, if needed [6].

3.1.5 Registries and Repositories

A registry stores named Docker images and distributes them on request. Each image may be available in different tagged versions in a registry [5].
Within a registry, images may be organized in collections, which are called *repositories* [6].

3.1.6 Docker Networks

As mentioned in 3.1.1, Docker features virtual networks in order to isolate containers in this regard, but at the same time allow containers to communicate with the host, each other and the outside world. These networks are based on virtual interfaces and are managed by the Docker daemon. Containers may be member of multiple networks at the same time [5].

By default, Docker installs three networks: a *bridge* network, a *host* network, and a *none* network. The *bridge* network, titled *docker0*, is a subnetwork that is connected to the host's networks. Docker connects containers to this network if it is not instructed otherwise. Containers that are membsers of this network can communicate with each other by using their respective IP addresses. They also may expose ports that can be mapped to the hosts network, which makes applications in them accessible from the outside.
The *host* network represents the actual hosts network. If containers are assigned to this network, they will be placed in the hosts network stack, i.e. all network interfaces defined on the host are available to the container [5].
The *none* network provides containers with their own network stack. Containers that are only members of the *none* network are completely isolated in regards to network communication, unless futher configuration is undertaken [5].

Besides the network types mentioned above, Docker features another type of network, the *overlay* network. Overlay networks are virtual networks that are based on existing network connections. They are intended to simplify the communication between containers running on multiple hosts which, in turn, run on multiple machines themselves. If a container is member of an overlay network, it is able to communicate with all other containers that are also part of this network, no matter which Docker host (or host machine) they are running on [5].
Docker's overlay network requires a key-value store to be present in order to persist information on its own state, e.g. on lower level networks that it relies on, network members, etc.

3.2 Docker Engine

The Docker Engine forms the core of Docker. Docker uses a client-server architecture: it features a daemon which provides the functionality and a client that controls said daemon [7]. Together, they enable the user to work with Docker containers. Both the client and the daemon may run on the same system, or be connected remotely via sockets or through a Representational State Transfer (REST) application programming interface (API) [5].

3.3 Docker Ecosystem

Around the Docker Engine, several other solutions have evolved to cope with different specialized tasks that are asscociated with building and running containers. In the following, a selection from these solutions will be introduced briefly.

3.3.1 Docker Swarm

Docker Swarm allows applications which rely on several Docker containers to be run on a cluster of machines. It provides an abstraction that lets a set of Docker Engines behave like a single Docker Engine. Further it features a mechanism that automatically assigns container to a specific host based on given rules [4].

A swarm setup typically consists of one or more *swarm managers*, multiple Docker hosts, and, in case that no remote discovery service is used, a local discovery service. By default, every new container is assigned to a swarm-specific overlay network [5].

Docker Swarm provides two kinds of mechanisms for the assignment of containers to Docker hosts, *filters* and *strategies*. Strategies tell Docker how to rank hosts for assignment by some specified criteria, e.g. resource usage or number of deployed containers.
Filters allow to specify rules, which Docker tries to apply when searching for an assignment target. Possible rules could for example be matchers for the host's name or identifier, its OS, or for custom tags, which may describe the host's role or properties like size of attached storage. It is also possible to declare the affinity of certain containers or images for being deployed on the same host [5].

3.3.2 Docker Machine

The goal of the Docker Machine tool is to facilitate the setup of Docker hosts. In order to fulfill this goal, Docker Machine creates one virtual machine per requested host [4, 5]. This has several reasons. First, this proceeding allows several Docker hosts to run on the same machine without having them interfere with each other. Second, it enables machines with OSs, which natively do not support Docker and Docker containers, to act as a host [5]. And third, as the virtual machine image is known, it lets the setup procedure make assumptions on its environment, which simplifies the installation and configuration of the Docker Engine.

3.3.3 Docker Compose

Docker Compose is a tool that enables the user to specify and run applications that consist of many containers. Similar to the way an image is described in a Dockerfile, the user lists the required containers and their respective run configuration in a YAML (?) file. Docker Compose interprets this file and sets the containers up accordingly [4].

4 Application Design

In order to make substanciated decisions in the design process, the intended outcome has to be outlined first. Bearing in mind the concepts presented in Chapter 2 and 3, objectives that together form the intended outcome are thus compiled in Section 4.1. Based on these objectives, the concept of a Docker-based WfMS is then shaped in Section 4.2.

4.1 Determination of Objectives

In this section, the overall objectives are inferred from both requirements imposed on the functionalities of Docker based WfMS as well as intangible ones.

4.1.1 Functional Requirements

In the following, expectations towards the functionality of the resulting WfMS are presented in a structured manner. These functionalities are grouped by the aspects and tasks of an WFMS, which are described in 2.2.1 and 2.2.2.

Infrastructure and Infrastructure Management

The workflow management system should be structured in a way that allows to change of parts of it during execution time.

It should be possible to add machines on the fly.

- execution environment: containers which always run

Workflow Modeling

The WfMS should thus enable modeling developers to incorporate the invocation of third party images into their workflows.

The modeler should thus be able to pass along information which allow the validation of input and output data with both activities and workflows.

- drop in 3rd party containers - specify container params - pass in validation schema for - activity - wf

Workflow Distribution

In order to reduce administrational work, workflows and their activities should be distributed to the servers that they will be run on in an automated way. - produced workflows should be as portable/environment independent as possible

Workflow Execution

- should provide form renderer

4.1.2 Intangible Requirements

Besides the rigid functional requirements there are also less palpable ones. Although they are harder to quantify, they are likely to have an impact on the usefulnes of the produced artifacts.

- modeling workflows (with 3rd p images) should be intuitive

4.1.3 Derived Objectives

4.2 Architecture

After the determination of objectives in Section 4.1, the solution can be developed in this section. Since it has considerable impact on the other aspects, the higher level architecture is chosen first in 4.2.1. With increasing level of detail, the other aspects are then fleshed out gradually. (Sections go here)

4.2.1 Application Structure

Micro services, service discovery [17]

Use Docker only for execution vs docker all the things

Developers of software systems have to cope with factors which impose challenges on them, such as high complexity within their systems, an increased need for integration of internal and external functionality and evolving technologies. Several architectural approaches emerged from the attempt to overcome these challenges. Strîmbei et al consider *monolithic architecture*, *Service-oriented Architecture (SOA)* and *micro services* to be the most relevant [16, p. 13].

Monolithic Architecture

Monolithic software systems are characterized by their cohesive structure. Usually, components in a monolith are organized within one programm, often running in one process [17, p. 35]. They communicate through shared memory and direct function calls. Monolithic applications are typically written using one programming language [16, p. 14]. In order to cope with increasing workload on a monolithic system, multiple instances of it are run behind a load balancer [17, p. 35].

The strengths of monolithic architecture lie in its comparably simple demands towards the infrastructure. As the application is run as one entity, deployment and networking are rather simple [17, p. 35]. Since data can be shared via memory or disk, monolithic applications can access it faster than it would be the case with networked components [16, p. 14].

- lower complexity of interaction between parts when multiple com- ponents or modules can be gathered into a sin- gle unit [16, p. 14] con: - Changes to one component can impact seemingly unrelated areas of the application increasing the risk of new development [17] - Individual components cannot be deployed, making wide reuse across the enterprise more difficult [17] - Components designed for reuse gravitate towards verbosity and readability over performance due to conflicting needs and levels of experience in the development team [17] - At the same time, scaling of individual components is not possible, and together these leads to obvious inefficiencies [17]

Service-oriented Architecture

SOA is based on the idea that code which provides related business functions can be bundled into one component and offer said functionality to other systems *as a service*, thus avoiding duplicated implementation of the functionalities among these systems [10, p.8]. The Organization for the Advancement of Structured Information Standards (OASIS) describes SOA as an architectural paradigm that supports the organization and usage of these services [9]. Each service provider exposes its offered services, which can then be utilized by service consumers [16].

Micro Services Architecture

The concept of Micro Services Architecture (MSA) is closely related to that of SOA, as it also promotes the encapsulation of functionality in standalone services which can be used by other parts of a system. There is disabiguity whether MSA is actually a concept on its own – or rather a specialized application of SOA [17, p. 35], [16, p. 17]. Stubbs et al describe MSA as a distributed system that consists of independent services which are narrowly focused and thus considered "lightweight" [17, p. 35]. * some more here*

«Possible instances of the wfms components in monolith, soa and msa»

Reasoning: monolith: pro: - deployment and networking are trivial [17] - scaling the system is accomplished by running multiple instances of the monolith behind a load balancer [17] - lower complexity of interaction between parts when multiple com- ponents or modules can be gathered into a sin- gle unit [16, p. 14] - highly optimized considering that: (1) data is read directly from disk through the file system and (2) the user has also the possibility to cache and prefetch built-in data [16, p. 14] con: - Changes to one component can impact seemingly unrelated areas of the application increasing the risk of new development [17] - Individual components cannot be deployed, making wide reuse across the enterprise more difficult [17] - Components designed for reuse gravitate towards verbosity and readability over performance due to conflicting needs and levels of experience in the development team [17] - At the same time, scaling of individual components is not possible, and together these leads to obvious inefficiencies [17] soa: pro: con: microservices: pro: - can be developed in disparate languages and tools [17] - They can be deployed, upgraded and scaled individually [17] - in theory leads to greater code reuse [17] con:

Choice: - Docker removes some of the downsides of microservices - microservices it is then

4.2.2 Components

- identified services - components per service

- inter-component communication - RabbitMQ was introduced to the project, which implements the Advanced Message Queuing Protocol (AMQP) - AMQP consists of three structural parts [Ze15]: - Exchanges - are the contact point for incoming messages, where the publishers deliver their messages [Ze15]. - Queues - are used to deliver messages. This is were consumers subscribe, in order to receive all messages which get delivered to a certain queue. If no consumer is registered on the queue, the messages are stored [Ze15] - Routes - describe the mapping between exchanges and routes. They define which requirements a message needs to meet, in order to be placed inside a certain queue [Ze15].

- docker for workflow execution - base images - container as instance w/ writable layer - autonomous vs centrally managed

5 Prototypical Implementation

5.1 Toolchain

5.2 Realization of the Architecture

Any Compromises here?

6 Evaluation and Discussion

7 Conclusion

Bibliography

[1] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. *IEEE Expert*, 12, 1997.

[2] J. Becker, C.V. Uthmann, M. zur Muhlen, and M. Rosemann. Identifying the workflow potential of business processes. In *Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences, 1999. HICSS-32*, volume Track5, pages 10 pp.–, 1999.

[3] Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Guiseppe Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems*, 24(3):405–451, 1999.

[4] Inc. Docker. Docker orchestration product brief.

[5] Inc. Docker. The docker user guide.

[6] Inc. Docker. The docker user guide.

[7] Inc. Docker. Docker.com.

[8] Wes Felter, Re Ferreira, Ram Rajamony, Juan Rubio, Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. *An Updated Performance Comparison of Virtual Machines and Linux Containers*. 2014.

[9] Organization for the Advancement of Structured Information Standards. *Reference Model for Service Oriented Architecture 1.0*. OASIS, 2006.

[10] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: designing, building, and deploying messaging solutions*. The Addison-Wesley signature series. Addison-Wesley, Boston, 2004.

[11] David Hollingsworth. Wfmc: Workflow reference model. Specification, Workflow Management Coalition, 1995.

[12] Open Container Initiative. Open containers initative.

[13] Peter Lawrence, editor. *Workflow Handbook 1997*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

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

[15] C. Ruiz, E. Jeanvoine, and L. Nussbaum. Performance evaluation of containers for hpc. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9523:813–824, 2015.

[16] Catalin Strimbei, Octavian Dospinescu, Roxana-Marina Strainu, and Alexandra Nistor. Software architectures - present and visions. *Informatica Economica*, 19(4/2015):13–27, 2015.

[17] Joe Stubbs, Walter Moreira, and Rion Dooley. Distributed systems of microservices using docker and serfnode. pages 34–39. IEEE, 2015.

[18] Daniel Wutke, Daniel Martin, and Frank Leymann. Model and infrastructure for decentralized workflow enactment. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 90–94, New York, NY, USA, 2008. ACM.