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
ESB	Enterprise Service Bus
GUI	Graphical User Interface
MSA	Micro-services Architecture
MOM	Message-oriented Middleware
OASIS	Organization for the Advancement of Structured Information Standards
OS	Operating System
PID	Process Identifier
REST	Representational State Transfer
SOA	Service-oriented Architecture
WFMC	Workflow Management Coalition
WfMS	Workflow Management System
WSDL	Web Services Description Language

1   Introduction and Motivation

RQ1: How can Docker leverage the deployment and execution of workflows in a distributed system? RQ2: Which decisions in software architecture and software design of a WFMS are complemented by Docker's functionality?

- 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 [5]. 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 evaluated 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, 15]. The WFMC defines workflows as the computerized facilitation or automation of a business process, in whole or part [15].

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 [15]. 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 [15]. This model is usually called *process definition* and stored in form of some high-level programming language construct [15, 28]. 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 [15]. They further may contain references to other processes, which are treated as a single activity in the process definition [15, 5].

- 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. [5]. If only the automated parts of such an instance are meant, the WFMC advocates for the term *workflow instance* [15].

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 [15].

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 [15].

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 [15].

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 [15].

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 [15, 5].

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 [15, 5].

Russel et al differentiate seven commonly used forms of data visibility in workflow management systems [23, p. 6-15]:

- **Activity Data**

  Data which is defined within an activity and which is accessible within the instance of this activity.

- **Sub-workflow Data**

  Data which is defined within a sub-workflow activity and is accessible from everywhere within this sub-workflow.

- **Scope Data**

  Data which is accessible within a subset of activities in a worfklow instance.

- **Multiple Instance Data**

  Data which is defined within an activity that can be instanciated multiple times. Each instance can access its own version of that data.

- **Workflow Instance Data**

  Data which is specific to a process instance of a workflow and which can be accessed by all components of that workflow during its execution.

- **Workflow Data**

Data elements which are accessible to all components all instances of a workflow and are controlled by the WfMS.

- **Environment Data**
  Data which exists in the operating environment and which can be accessed by components of any workflow during execution.

They further identified six types of data interaction between the various hierarchy levels in workflows [23, p. 16-24]:

- **Activity – Activity**
  Data is passed between two activity instances which belong to the same workflow instance.

- **Sub-workflow Activity – Sub-workflow Components**
  Data is passed from a sub-workflow activity instance to the corresponding sub-workflow.

- **Sub-workflow Components – Sub-workflow Activity**
  Data is passed back from a sub-workflow instance to the corresponding sub-workflow activity instance.

- **Activity – Multiple Instance Activity**
  Data is passed from an activity instance to a successor activity which may be instanciated multiple times. It may be passed to all instances of the multiple instance activity or distributed among them according to specific rules.

- **Multiple Instance Activity – Activity**
  Data is passed from an activity which may be instanciated multiple times to a successor activity instance.

- **Workflow Instance – Workflow Instance**
  Data is passed from one instance of a workflow during its execution to another workflow instance that is being executed in parallel.

Workflow data may be either made available from a common data store, get passed along with the control flow of a workflow, or be explicitly passed to the receiving component [23, pp. 16-21].

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 [15, 5].

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 [15, 5].

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 [17]. 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" [15].

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 [15, 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 [15].
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 [15].

2.2.2   System Components

TODO: Low-level The WFMC identified four high-level groups of software components that most WfMSs have in common: *Process Definition Tools*, *Administration and Monitoring Tools*, *Workflow Client Applications*, and *Workflow Enactment Service* [15].

Process Definition Tools

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 [15].

Administration and 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* [15].
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* [15]. The user interface may be part of the workflow client applications or exist as a separate software component.

Workflow Engine

In order to enact workflows, instances of them are created. This happens based on the interpretation of previously created process definitions. Workflow instances are usually managed by a component which is called workflow engine. The workflow engine decides which activities and sub-workflows of a workflow can be started, determines suitable participants, invokes external applications and it updates the users' worklists accordingly. It further manages the storage and flow of workflow control data and workflow relevant data [15].

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 [15].

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 [12]. These goals can be fulfilled by both virtual machines and software containers [22]. 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 and to identify possibilities it gives. 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 [22].

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 [22].
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* [22].

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 Identifiers (PIDs), and other components, in order to achieve a fine grained control over the respective isolation [22].
Besides Docker, there are several solutions that are all based on the aforementioned kernel features, e.g. LXC, LXD, lmctfy, systemd-nspawn, etc [22]. There are ongoing efforts to create a common container standard [16].

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 [8, 20]. 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 [20, 8].

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 [8].

- *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 [9]. They are never deleted automatically and therefore must be cleaned up manually when they are not needed anymore [8].

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 [9].

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 [9].

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 [8].

Within a registry, images may be organized in collections, which are called *repositories* [9].

3.1.6 Docker Networking

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 [8].

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 [8].
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 [8].

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 [8].
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 [10]. 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) [8].

3.3   Docker Ecosystem

Around the Docker Engine, several other solutions have evolved to cope with different special-
ized 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 spe-
cific host based on given rules [7].

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 [8].

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 [8].

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 [7, 8]. 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 [8]. 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 [7].

3.3.4   Docker Hub

4 Conceptual Development of the WFMS

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 4.3.

The benefit WfMSs could obtain from using the Docker ecosystem is twofold. On the one hand, the distribution and execution of workflows could be improved. On the other hand, the infrastructure of the system itself might be improved by the use of Docker.

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 Objectives

In the following, expectations towards the functionality of the resulting WfMS are motivated in a structured manner. These functionalities are grouped by the aspects and tasks of a WfMS, which are described in 2.2.1 and 2.2.2. The resulting objectives and the requirements that need to be met in order to fulfill them are summarized in Table 4.2.

<-><-> split to typicla wfms feature subsection?

Infrastructure and Infrastructure Management

As the IT environment of an organization changes over time, the WfMS should be structured in a way that allows the adaption to such changes with the least possible system downtime. Further, it should be possible to add servers to the system during execution time, which then should be usable with a minimum of manual configuration.

If an organization is unable to perform its business processes, it is likely to suffer from financial losses. The failure of a WfMS which enables those business processes can thus cause severe problems for an organization. The WfMS developed in this thesis should thus be able to be resilient towards failures, i.e. provide as much functionality as possible if a part of it fails when and try to recover autonomously. This requires well separated modules.

Objective	Requirements
Resilience in case of failures	• Non-failed components continue to provide their functionality • Failed components are restarted
Dynamic addition of enactment servers	• Suitable servers are discovered • User can add servers during execution time
Environment of arbitrary services	• Required services for a workflow can be selected • Required services are started on respective nodes • Workflows are connected to their required services • Services are restarted if necessary
Third-party containers as workflow components	• Graphical User Interface (GUI) for browsing Docker Hub images exists • Modeling GUI has a "Container" element • User can specify start parameters and commands
Input and output validation	• User can specify validation schemata • WfMS performs validity checks
Resource usage management	• User can prioritize/demote activities and workflows • WfMS enforces respective resource usage
Property-based scheduling of containers	• Properties of servers can be described • Workflows and activities can require server properties • Containers are run on suitable servers
Reduction of administrative work	• Added servers are configured automatically • All execution related containers are started automatically • Saved/updated workflows and activities are deployed automatically

Tab. 4.1:   Objectives and their respective requirements

To enable workflows to be run in a specified environment of services, the execution environment of the WfMS should have a mechanism that allows the specification of such services in form of containers which are made available to workflow components when they are being executed. It also should be able to ensure that these containers are restarted in case of a failure to ensure their presence in the execution environment.

Workflow Modeling

One benefit of Docker containers is, that full application stacks can be bundled with all their dependencies and pre-configured regarding their invocation [3, p. 82]. The result can be considered as a black box, which provides some specific functionality that could be used without further configuration. In combination with the facilitated sharing of images through repositories, this provides a good foundation for modular reuse and combination [4, p. 6]. In order to reap this advantage, WfMS should thus enable modeling developers to incorporate the invocation of third party images from within their workflows. This includes the specification of parameters, with which the image should be run.

To ensure the correctness of data along the execution, the modeler should be able to pass along information with both activities and workflows that makes the validation of their input and output data possible. The WfMS should provide a mechanism which performs this validation in the course of the workflow enactment.

In case that an execution node is working to full capacity, a means should be provided to support the swift finalization of time-critical tasks before those that are not, e.g. the temperature analysis of a cold storage with sensitive goods which is prioritized over the automated reorder of tacks for the office. The modeling environment should thus enable the user to put restrictions on the resource usage of specific activities in order to prioritize or demote them.

Workflow Distribution

In the course of an WfMS's life cycle, many workflows are modeled and many activities are created, and both are likely to be updated once in a while. In order to reduce administrative work, workflows and their activities should be distributed to their correct execution servers after these events in an automated way.

Workflow Execution

All containers that are related to the execution of workflows should be started by the WfMS without user interaction.

The IT infrastructure in an organization may be heterogeneous in terms of machine capabilities and environment, e.g. (among many others) the amount of memory that is available or the geographic location of the machine. These factors may be of interest when it comes to performance objectives or legal regulations. The scheduling of workflows or activities to nodes for execution should thus be possible based on a structured description of said properties.

In 2.1.5, various forms of data visibility and interaction were presented. Russel et al examined the capabilities of various workflow engines with regards to these characteristics [23]. In order to obtain a WfMS with useful functionality, it should support at least those forms of data visibility and interaction that are common among existing solutions. As a rough estimation for this, each capability shall be deemed as required if a majority of solutions examined in that study supports it.

4.1.2   Intangible Objectives

Besides the rigid functional objectives there are also less palpable ones. Although they are harder to quantify, they are likely to have an impact on the value of the produced artifacts. The functionalities that were fleshed out in 4.1.1 lose value if using them is cumbersome.

While some complications may be tolerable during the setup of the solution, as it (hopefully) is a one-time procedure for an organization, recurring interactions should be facilitated by an appropriate graphical user interface.

- browsing docker hub images - modeling workflows - adding servers - accessing audit data

4.2   Docker in Workflow Execution

- image = activity/workflow, container = activity instance / workflow instance

There are several possibilities how Docker can be utilized for the execution of workflows. Each combination of variants (abbreviated as depicted in Table **??**) has its own advantages and disadvantages, which are elaborated in this chapter.

The first aspect is whether one wants to spread the containers associated with one workflow

instance across various machines for execution (S) or constraint them to run on the same node as a group (G).

Second, one can differentiate to which extend workflow components are wrapped in their own containers. One could encapsulate only activities in containers ($*_{AC}^{*}$) and distribute workflow information on another way, let each workflow and activity reside in a different container ($*_{SEPC}^{*}$), or wrap them in one container together ($*_{1C}^{*}$). Since this container is an atomic unit, it cannot be spread across many nodes for execution. One also could abandon the idea of a one-to-one mapping between the Docker and workflow concepts and establish worker containers with specialized behavior (according to a workflow/activity type) which perform suitable tasks on request ($*_{Type}^{*}$).

Third, it is important to define the way data is exchanged between containers. One possible solution could be a data volume that is shared by all containers in need to exchange data with each other ($*_{*}^{DV}$). Data could also be passed between containers via some system service, e.g. a database, ($*_{*}^{SER}$), or on a direct connection between the containers ($*_{*}^{D}$).

Finally, rather independent from the previous variants and hence discussed in isolation, one might want to choose the mechanism that decides which containers are run on which machines, i.e. the execution scheduling.

TODO: add the "how's", table consistent with text?

4.2.1 Grouped execution using common data volume (G_{*}^{DV})

The idea behind this concept is, that all containers involved in the execution of one workflow instance have access to a common working directory, in which the data visibility scopes can be established using a file system structure. This working directory resides in a data volume owned by a container that is specific to the respective workflow instance and whose only purpose is to ensure the existence of said volume.

G_{*}^{DV} natively supports all identified types of data visibility by providing respective directories and access to them, with the limitation that workflow data and environment data has to be copied to the data volume before execution and is thus restricted to the state it had at that time. Altering this data is possible, e.g. by replacing the respective files via `docker copy` or altering it in a `docker exec` session, but extra measures have to be taken to request such an up-to-date version.

Data interactions between activity instances, a sub-workflow activity and its related components and vice versa are supported by G_{*}^{DV}. Exchanging data between two running workflow

Objective	Requirements
Support of data visibility	• Activity Data • Sub-workflow Activity Data • Multiple Instance Activity Data • Workflow Instance Data • Workflow Data • Environment Data
Support of data interactions	• Activity $\rightarrow$ activity • Sub-workflow activity $\rightarrow$ sub-workflow • Sub-workflow $\rightarrow$ sub-workflow activity • Workflow instance $\rightarrow$ workflow instance

Tab. 4.2: Objectives and their respective requirements

Data Exchange / Containerization	Common Data Volume	Service	Direct
Grouped execution on one node			
Activities in containers	G_{AC}^{DV}	G_{AC}^{SER}	G_{AC}^{D}
Workflows and activities in separate containers	G_{SEPC}^{DV}	G_{SEPC}^{SER}	G_{SEPC}^{D}
Workflow and activities in one container	G_{1C}^{DV}	G_{1C}^{SER}	G_{1C}^{D}
Type based workers	G_{Type}^{DV}	G_{Type}^{SER}	G_{Type}^{D}
Spread execution over available nodes			
Activities in containers	✗	S_{AC}^{SER}	S_{AC}^{D}
Workflows and activities in separate containers	✗	S_{SEPC}^{SER}	S_{SEPC}^{D}
Workflow and activities in one container	✗	✗	✗
Type based workers	✗	S_{Type}^{SER}	S_{Type}^{D}

Tab. 4.3: Containerization/Grouping/Communication Solution Pairings

instances is not possible, though, as mounting volumes to running containers is not supported by Docker yet. This kind of interaction thus requires some additional tools. It would be possible to grant access to workflow instances running on the same machine – at the price of loss of control over data visibility – by mounting the top-level working directory of that machine in all containers.

As long as no requests to external sources are made within the workflow, data has to be transferred over the network only twice in this approach – for the input and output of the workflow. All subsequent data transfers are either implicit, e.g. by accessing the respective working directory or a symbolic link to it, or take place on the local machine, e.g. copying one or more files. Because of transfer rates *QUOTE*, G_*^{DV} has an advantage over message-based approaches when it comes to processing of large and/or many data sets, i.e. log files, genome research data or images. Also, unless required by the activities themselves, data conversion is less of an issue since relying on the file system allows arbitrary file types to be used.

Sharing a data volume using only Docker tools requires all containers to be on the same machine, which is why there is no spread version S_*^{DV} of this concept. Approaches exist to utilize **NFS!** (**NFS!**) or **P2P!** (**P2P!**) file sharing for distributed access to data volumes, but given the limited scope of this thesis they shall not be regarded further here [18].

G_{1C}^{DV} could theoretically work on its own file system in the editable layer of the container without a dedicated data volume, since all workflow components would have access to it. This would make the container self contained on the one hand – and thus easier to export or migrate – but would on the other hand couple the data life cycle to the processing container's life cycle. That is, if the container is removed, the data is removed with it.

4.2.2   Grouped or spread execution using an intermediate service (G_*^{SER} /s_*^{SER})

$*_*^{SER}$ represents the concept of providing some service that is able to store and serve data on request. This implies, that the docker containers have to feature a mechanism to communicate to this service.

The service could either be running as a single instance on some node in the network or on each node in the network. While the former avoids having to deal with synchronization between service instances and inconsistencies resulting from race conditions, the latter could balance the load. (shorten repsonse times? hops)

Since the storage of workflow related data is decoupled from the execution in $*_*^{SER}$, the coverage of data visibility and data interaction capabilities depends on the chosen underlying service.
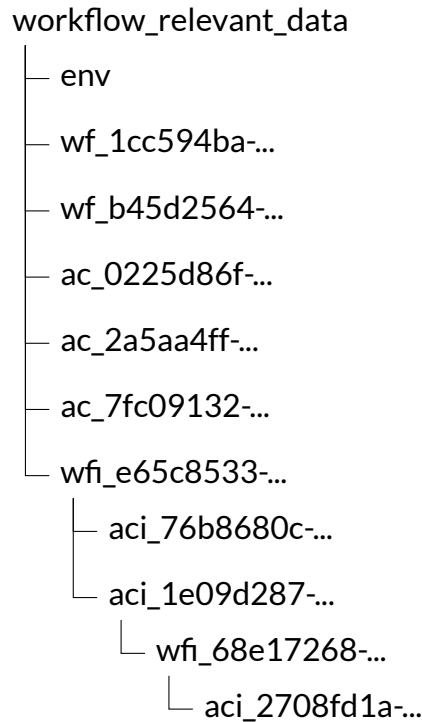
```
workflow_relevant_data
├── env
├── wf_1cc594ba-…
├── wf_b45d2564-…
├── ac_0225d86f-…
├── ac_2a5aa4ff-…
├── ac_7fc09132-…
└── wfi_e65c8533-…
        ├── aci_76b8680c-…
        └── aci_1e09d287-…
                └── wfi_68e17268-…
                        └── aci_2708fd1a-…
```

Fig. 4.1:   Exemplary directory structure for G_*^{DV}

Theoretically, all forms of data visibility and interactions should thus be possible. Also, the solution exhibits the same properties for the grouped and spread execution, for the same reason.

In this variant, data is transferred before and after each step in the workflow, i.e. when the execution starts, when it ends, whenever an activity is instantiated or an instance finished its work. This is only economical if the amount of data is sufficiently small or if the workflows consist of few activities.

In order for the workflow execution to function correctly, the service must be reliably available to all containers.

- service must be present

- think about hosting multiple services (1 per node) - probably superfluous network roundtrip
- less efficient for large files than using a file system

4.2.3   Grouped or spread execution using direct connections (G_*^D/S_*^D)

In this scenario, the containers communicate with each other directly in order to exchange workflow relevant data. It can be split again in two sub-variants, according to the data passing patterns noted in 2.1.5. On the one hand, one where the workflow relevant data is passed along

the control flow ($*_*^{D_j}$). On the other hand, one where containers can query each other for their data($*_*^{D_s}$).

Joined data and control flows ($*_*^{D_j}$)

In $*_*^{D_j}$, the data flow is is directly coupled to the control flow, i.e. all data is passed on on invocation. This requires all data that might be used in another activity instance to be passed along, no matter whether the succeeding activity uses them or not [23]. While this allows to pass (and update) workflow and environment data, it may be problematic for larger amounts of data.

Separate data and control flows ($*_*^{D_s}$)

$*_*^{D_s}$ requires all containers that shall be queried for data to provide some communication mechanism and to be running. Thus, in the worst case every container related to a workflow instance has to be running. Even though the containers' use of processor time can be reduced by pausing them until they are needed, this approach could impose a considerable strain on the host machine's memory, as pausing containers has no effect on their memory consumption.

Workflow data and environment data in $*_*^{D_s}$ may be provided to the first activity, which then could be queried for a (static) version of it. Because workflow instances are not represented as containers in $*_{AC}^{D}$, $*_{AC}^{D_s}$ provides no means to store and access sub-workflow data, multiple instance data and workflow instance data.

$*_{Type}^{D_s}$ is not a feasible solution, as keeping the worker containers running in order to make them queryable would block their use for other workflow instances, thus rendering the WfMS incapable of processing further workflow instances once all workers are invoked – unless further workers are spawned.

As all workflow components reside in one container in G_{1C}^{D}, no communication between containers is necessary – all data can be exchanged on an arbitrary way within that single container. This variant thus represents a special case.

4.2.4 Promising combinations of characteristics

Due to its shortcomings regarding the supported types of data visibility and interactions, direct communication between instance-related containers ($*_*^{D}$) can be ruled out as a candidate for the prototype. The remaining combinations may be eligible depending on the intended use

	Data Visibility						Data Interactions			
	Ac Data	SubWF Ac Data	MultInst Ac Data	WFInst Data	WF Data	Env Data	Ac→Ac	SubWF Ac→SubWF	SubWF→SubWF Ac	WFInst→WFInst
G_{AC}^{DV}	✓	✓	✓	✓	✓(†)	✓(†)	✓	✓	✓	○
G_{SEPC}^{DV}	✓	✓	✓	✓	✓(†)	✓(†)	✓	✓	✓	○
G_{1C}^{DV}	✓	✓	✓	✓	✓(†)	✓(†)	✓	✓	✓	○
G_{Type}^{DV}	✓	✓	✓	✓	✓(†)	✓(†)	✓	✓	✓	○
G_{AC}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G_{SEPC}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G_{1C}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G_{Type}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$G_{AC}^{D_j}$	✓	✓	○	✓	✓	✓	✓(‡)	○	○	○
$G_{AC}^{D_s}$	✓	○	○	○	○	○	✓(‡)	○	○	○
G_{SEPC}^{D}	✓	✓(‡)	✓(‡)	✓(‡)	○	○	✓(‡)	✓(‡)	✓(‡)	✓(‡)
G_{1C}^{D}	✓*	✓*	✓*	✓*	○	○	✓*	✓*	✓*	✓*
G_{Type}^{D}	✓	✓(‡)	✓(‡)	✓(‡)	○	○	✓(‡)	✓(‡)	✓(‡)	✓(‡)
S_{AC}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_{SEPC}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_{Type}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_{AC}^{D}	✓	○	○	○	○	○	✓(‡)	○	○	○
S_{SEPC}^{D}	✓	✓	✓	✓	○	○	✓(‡)	✓(‡)	✓(‡)	✓(‡)
S_{Type}^{D}	✓	✓	✓	✓	○	○	✓(‡)	✓(‡)	✓(‡)	✓(‡)

✓ natively supported | ✓* natively supported, a direct connection within the container is assumed | ✓(†) can be passsed on instantiation, real-time access requires additional tools
✓(‡) natively supported, assuming that all containers are left running for the time of workflow execution | ○ not natively supported, requires additional tools

Ac = Activity | SubWF = Sub-workflow | MultInst = Multiple instance | WFInst = Workflow instance | WF = Workflow | Env = Environment

Tab. 4.4: Containerization/Grouping/Communication Solution Pairings

case and desired properties of the WfMS, e.g. the kind of data that is processed, the targeted infrastructure, and nature of the workflows.

If the data volume approach is chosen, one is committed to grouped execution on one node. Considering the different containerization solutions, G_{SEPC}^{DV} is favorable over G_{AC}^{DV}, because the explicit existence of containers which represent (sub-)workflow instances makes it possible to both track their state using Docker mechanisms and manage their respective working directories. Embedding workflow engine logic into these containers could enable workflows to be exported and used in a stand-alone fashion.

G_{SEPC}^{DV} is also favorable over G_{1C}^{DV}, because the modularity of G_{SEPC}^{DV} permits to update single activities within the workflow by distributing new versions of their respective image. In G_{1C}^{DV}, the whole image would have to be updated due to the way layering in Docker images works. Also, G_{1C}^{DV} does not provide the means to track the activity instances' life cycles with Docker mechanisms, only that of the workflow instance. If the workflow is not meant to be updated but to be distributed to third parties as a stand-alone solution, e.g. an automated batch process for photos that is sold to photographers, it is a viable option, though.

The main benefit of $*_{Type}^{*}$ is that it allows to distribute the workload among several workers. Since G_{*}^{DV} requires all involved workers to be on the same machine, this advantage is voided.

Altogether, this makes G_{SEPC}^{DV} the variant of choice for data-centric WfMSs.

When comparing the service-based variants with each other, G_{*}^{SER} and S_{*}^{SER} share most their advantages and drawbacks. S_{*}^{SER} permits the containers to run on different nodes, though. As this allows containers to be assigned to machines according to their resource requirements and the momentary workload of a machine, it is the preferred variant of those two.

Because the decoupling of data and containers is well supported by $*_{*}^{SER}$, it is a good fit for the worker-based approach of $*_{TYPE}^{*}$, which in turn facilitates load balancing.

- AC vs SEPC?

** DECISION TREE**

4.3 System 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 4.3.1. With increasing level of detail, the other aspects are then fleshed out gradually.
(Sections go here)

4.3.1   Architecture Styles

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 [24, 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 [25, p. 35]. They communicate through shared memory and direct function calls. Monolithic applications are typically written using one programming language [24, p. 14]. In order to cope with increasing workload on a monolithic system, multiple instances of it are run behind a load balancer [25, p. 35].

The strengths of monolithic architecture lie mostly in its comparably simple demands towards the infrastructure. As the application is run as one entity, deployment and networking are rather simple [25, 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 [24, p. 14].
Also, as the interaction between the application's components happens XYZ, the complexity of this interaction is lower compared to interaction between distributed components [24, p. 14].

The weaknesses of monolithic architecture stem from its cohesive nature. As its components are usually tightly coupled, changes to one component can affect other parts of the application, which complicates the introduction of new components and the refactoring of existing ones [25]. Components cannot be deployed individually, which hinders reuse of functionality across several applications more difficult and makes scaling of single bootleneck components impossible [25]. Also, if the application runs in a single process, the failure of one component may bring down the whole application [19, p. 5].

In combination with Docker, a possible solution could look like this: ...

Service-oriented Architecture

SOA is based on the idea that code which provides related business functions can be bundled into one component which offers said functionality to other systems *as a service*, thus avoiding duplicated implementation of the functionalities among these systems [14, p.8]. An application may then use several services in order to fulfill its own business function [21, p. 390]. 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 [13]. Each service provider exposes its offered services in a standardized way, e.g. using Web Services Description Language (WSDL), which can then be utilized by *service consumers* [21, p. 390], [24, p. 17].

Messages between services in SOA are either of direct nature, which is called point-to-point connection, or backed by a message bus, the Enterprise Service Bus (ESB) which incorporates the integration logic, e.g. on transport and transformation of messages, between services and supports asynchronous messages [21, p. 393]. While the former leads to tight coupling between the components, which becomes impractical with increasing numbers of endpoints, the latter manages this scenario better [21, p. 393].

On the one hand, SOA has some advantages in comparison to monolithic architecture. Service consumers do not have to make assumptions - or know - how services work, they only have to rely on the invocation of a service and its result to be formed as expected [21, p. 390]. As long as the interface and the output of existing services do not change, a service provider may thus be altered or its capabilities be extended without affecting its services' consumers [21, p. 390]. SOA thus enhances an organization's ability to respond quickly to changes [21, p. 390], [6, p. 254]. Since legacy applications can be provided with appropriate interfaces, SOA can help to integrate and extend them [21, p. 390].

On the other hand, SOA has some drawbacks, too. For example, the failure of a single service provider may bring down multiple applications that consume its services, if no fallback measures are in place [21, p. 408f]. Also, the overall performance of an application with SOA depends on the aggregated performances of the services it uses and their respective interactions [21, p. 408f].

In combination with Docker, a possible solution could look like this: ...

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 unambiguity whether MSA is actually a concept on its own – or rather a specialized application of SOA [25, p. 35], [24, p. 17]. Stubbs et al describe MSA as a distributed system that consists of independent services which are narrowly focused and thus considered "lightweight" [25, p. 35]. That exact principle has been described as a version of SOA before [21, p. 395].

Strîmbei et al created a differentiation between SOA and MSA as a distinct concept based on several sources. They come to the conclusion, that while the communication in SOA is synchronous and "smart but dependency-laden", MSA usually relies on asynchronous, "dumb, fast messaging" – meaning that there is few information on the participating services contained in the messaging infrastructure. Further, they perceive applications in SOA to be typically imperative in their programming style, while MSA would be in an event-driven programming style [24, pp. 17-20]. They see SOA applications as being usually stateful and MSA applications as stateless. Finally they characterize the databases in SOA as large relational databases and the databases in MSA as small, often non-relational databases.

One benefit of MSA, which it shares with SOA is that each service can be developed in a language and with a toolset that suits its specific needs, e.g. a lower-level language for time-critical but simple tasks or a high-level language with some framework for complex ones, instead of having to find a compromise that suits most of the application [25, p. 35], [19, p. 4], [26, p. 113]. The narrow focus of each service makes it less specialized to certain uses, which should theoretically enable better reuse of code [25, p. 35]. Another positive aspect is the *resilience* of micro-services when it comes to service failures, that is, a single failing service does not render the whole system incapable of working [19, p. 5]. Due to the properties of the MSA, micro-services may be deployed, upgraded and scaled individually [26, p. 116].

Researchers also see disadvantages and problems that may go hand in hand with the use of MSA. While MSA facilitates deploying parts of an application individually, the overall amount of work required for the deployment of all services is higher and the coordination of the deployments is complexer than the deployment of a monolithic application. As services may be unavailable at times, a mechanism has to be in place that allows the discovery of services (such as the Message-oriented Middleware (MOM)) [25, p. 35]. Unless a dedicated service is introduced, there is no central access to the services' logs. Aggregation, analysis and fixing errors is thus more complicated in comparison to monolithic architecture [25, p. 35]. In order to define the different services, it is necessary to find the right size for each service, i.e. the appropriate

scope of its functionality. This process is difficult and not needed to this extend in monolithic architectures or SOA.

In combination with Docker, a possible solution could look like this: ...

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

4.3.2   Choice of an Architecture Style

One central requirement for the stated objectives is the modularization of the application. It enables the containment of failures, the replacement or upgrading of components at runtime, and the individual scaling of parts of the WfMS. The concept of SOA and MSA inherently requires the modularization of code, while it is optional – yet, advisable – in a monolithic architecture.

While measures can be taken in monolithic applications to cope with failure of components to some degree, if the underlying machine fails or the process dies for whatever reason, the whole application is rendered inoperative [19, p. 55]. SOA and MSA both urge to account for the possibility of a non-responding service – in the first place because of the unreliability of network communication, but that works or any other reason, too. They hence inherently support the objective of resilience better than monolithic architecture.

Upgrading or replacing components of an application at runtime is possible in each of the presented architectures. In SOA the service may be replaced at will by directing requests to an instance of the new version of that service, given that the previously exhibited behavior does not change. The same applies for MSA, but as there is no direct messaging, the replaced components just have to adhere to the expected messaging scheme. Monolithic applications may introduce patterns such as dependency injection and dynamic loading to make changes at runtime possible.

Even though scaling individual parts of an application is a non-trivial task in SOA and MSA, it is possible. With a monolithic architecture, scaling the whole application is usually easier than in SOA and MSA – in most cases, another instance of the application may be started for that purpose –, but it is not possible to scale only those parts of an application where performance bottlenecks arise.

Facing these considerations, monolithic architecture is ruled out as the favored application structure. In the following, only the decision between SOA and MSA thus has to be made.

The Docker Ecosystem facilitates the setup of the infrastructure for a MSA. As stated in 4.3.1, the MOM itself contains little to no knowledge about the system using it. Thus, the MOM and

all application components may simply be started in separate containers and connected using an overlay network.

Docker permits the configuration of restart policies for specific containers. In case that one container crashes, it is restarted with its previous settings, if configured so.

This reasoning leads to the overall conclusion, that micro-services are the architecture of choice with regards to the chosen objectives.

4.3.3 User Interaction with the System

TODO: implemented as REST + MQ for simplicity

- in monolithic architecture - unified frontend feels natural - with micro services - two options - separate frontends per service - more flexibility - client somehow has to know the services - API gateway pattern to consolidate services the user can interact with - The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services. - + single entry point for all clients. - + decouples internal structure from clients / encapsulates the internal structure of the application - - new single point of failure - - has to be maintained [] - options:

4.3.4 Inter-component Communication

Since all services reside in separate containers, i.e. are isolated through namespaces for processes, networks etc, they have to be connected to each other somehow.

The naïve approach would be to let each container expose its required ports on the host's network interface. In order to communicate with a service in another container, an application would then address the host machine **IP!** (**IP!**) with the respective port. While this solution appeals because of its simplicity, it comes with considerable drawbacks. First, a port can only be used by one application at a time. This poses a problem as soon as a container is run more than once simultaneously, e.g. if multiple services require the same database application or a service is started several times for scaling. Second, this exposes the services in doubt to requests from any computer that can communicate with the host machine. Unless this is desired behavior, it creates an unnecessary attack surface.

Another approach is the use of a Docker feature called *links*, which allows to specify direct connections between containers based on their names. Docker then creates a secure tunnel between the specified containers and provides information on how the link source container may be ad-

dressed to the recipient container. This happens in two ways: it passes along all environment variables of the link source container to the targeted container and updates the `/etc/hosts` file, which is responsible for manual resolution of hostnames to **IP!** addresses. Links can be specified when a container is created by referring to one or more already running containers. Linked containers can be contacted via their hostname from within the started container. While links – in contrast to the first approach – allow the same port to be used in different containers and do not expose the containers per se, they too have disadvantages. First, and most important, they are static. That is, restarting one container breaks the link functionality. This is problematic, as the re-deployment of a service that relies on links or is linked to may require a domino-like chain of container restarts to restore the linking behavior. Second, they do only work on the same host. This solution thus does not support the distributed execution of the WfMS micro-services, unless directly related containers, e.g. a service and its database, are placed on the same node and all other communication takes place via exposed ports on the respective host machines. Third, **all!** (**all!**) environment variables that Docker created within a container are passed to any container that links to it, which could post problems regarding security if they contain sensitive data like passwords.

To cope with these disadvantages, the ambassador pattern was introduced [11]. The idea behind that pattern is the introduction of a container that acts as an intermediator between two services. This container is linked to both original containers and forwards their requests. In case that one of them needs to be restarted, this container is restarted to restore the connection – in place of the other container. By using two ambassador containers which point to each other, a multi-host setup can be achieved. An obvious drawback of this pattern is, that it does not scale well, since each connection requires at least one additional container to be added to the setup. In a clique-like connection setup between five containers *on the same host*, the ambassador pattern would already require ten additional containers.

While they are still supported under the name *legacy links*, they have been deprecated with the introduction of Docker networks due to their drawbacks.

Another solution is based on the Docker networking feature set, which was introduced in the end of 2015. To be more specific, it utilizes overlay networks, which were briefly presented in 3.1.6. During the development of this feature set, the *Container Network Model* was added, which allows containers to become member of multiple networks [27]. This enables the creation of purpose-oriented networks, e.g. a "backend" and a "frontend" network. Containers that are members of both networks can communicate with all containers, while containers which are exclusively connected to the frontend network can only see other containers of this network. This way, one could for example force access to a database to be routed through a container that filters malicious requests.

This concept can be adapted to suit the needs *TODO WHAT ARE THE NEEDS MY FRIEND* of the prototype: one overarching network is created, which allows all micro-services to communicate with each other. Their containers may are also be members of smaller networks that connect them to their support containers, e.g. databases, while keeping these containers isolated from the rest of the WfMS. In order to limit access of (probably untrustworthy) third party containers in the workflow to the WfMS, a second network may be created in which only containers required for the enactment of workflows are members. ** TODO drawbacks? **

As the last approach has the fewest drawbacks, it should be favored over the other presented ones.

4.4   System Design

4.4.1   Components

As noted in 4.3.1, one of the downsides of MSA is that suitable service boundaries have to be determined. Some sources advise to first build a monolithic application and then analyze the result to single out services that can be extracted. In case of WfMSs, the identification of system components by the WFMC for their reference model can be interpreted as such an analysis. Based on those components further micro-services for the prototype are then identified.

As presented in 2.2.2, the WFMC identified the following components:

- Software Components
  - Definition Tool
  - Organization Modeling Tool
  - User Interface
  - Workflow Engine(s)
  - Worklists Handler
- Data Components
  - Organization Data
  - Process Definitions Data
  - Workflow Control Data
  - Workflow Relevant Data
  - Worklists Data

- ui adapted to suit the API gateway pattern - user interface split per purpose (dev/user)

Besides the major components, independent functionalities which are frequently used may be

singled out to own services. To demonstrate this exemplarily, a service which deals with the validation of input and output data is extracted for this prototype.

In the following, the resulting set of services is presented.

Workflow Definition Service

The workflow definition service encompasses the functions envisioned by the WFMC as Process Definition Tools, i.e. it is concerned with the analysis, modeling, description and documentation of business processes in form of workflow models and their process definitions. It further manages the assignment of activities to roles.

With regards to that functional scope, this is also the component that should handle the transformation of workflows into their distributable format, e.g. a self-contained description file or Docker images. In case of the latter, the workflow definition service would require access to a Docker daemon in order to perform the export. Once a workflow is transformed, the service should publish it.

This components needs to incorporate the concepts required for its task. That is, workflow, activity , process definition, and control flow. The roles assigned to activities have only to be dealt with as unique identifiers, relying on the assumption that components which have to use them may resolve them themselves.

According to the concept of MSA, the workflow definition service component does not bring its own user interface, but rather exposes only an API. This allows workflow definitions to be created and altered by arbitrary other services, e.g. some converting service which translates other process definition formats, some feedback mechanism that alters workflows based on their execution performance, or different gateways which provide suitable user interfaces for various user devices.

Organization Management Service

The organization management service is part of the "administration and monitoring tools". As its name suggests, its functionality is aimed at the management of actors within an organization and their mutual relationships. Based this information, it may be queried for users or roles, or authenticate users for the use of the WfMS. Just like the workflow definition service, this service only exposes an API.

Worklist Service

The sole responsibility of this service is the management of users' worklists. It should create and delete worklist items on request and publish the data users submitted to it. If an user is deleted, it should reschedule the worklist item to a new provided user.

Workflow Engine Service

In wide parts, the workflow engine service is congruent to the "workflow engine" as identified by the WFMC in terms of functionality. It is As its responsibilities encompass the provision of the runtime environment, the workflow engine service should instanciate a provisioner on a node when it is added to the swarm. It also should ensure, that

- choose participants - add to execution networks

The chosen option from the solutions presented in 4.2 has an impact on the range of functionalities that this service has.

The extent to which the workflow engine service controls the instanciation of workflow components depends on .

Developer Gateway

- provides access to - infrastructure - definition - organization

User Gateway

- provides access to - worklists - forms

Organization Data

Process Definitions Data

Workflow Control Data

Workflow Relevant Data

Worklists Data

Validation Service

One micro-service that might be extracted from the above services concerns the validation of data is the validation service. Given a dataset and a set of rules on how to validate this dataset, the service is able to perform its task autonomously. As validation is a frequently recurring action in the execution of workflows – before and after each activity and workflow – it could thus be beneficial to be able to scale the execution of this task independently.

Support Services

There are three services which are required for the infrastructure in order to achieve the specified objectives. First, some MOM, as it enables the communication between all components. Second, a service that allows the configuration of the execution environment. And third, a component that takes care of provisioning all machines with the images they require.

Message Oriented Middleware

- must be reachable by all other services

Infrastructure Management Service

- see machines in the swarm - see their (labelled) properties - see (running) containers and available images

Registry

All solutions presented in Section 4.2 feature custom Docker images, be it workers or containerized activities or workflows. These images presumably contain information on business

processes and other information whose disclosure should be avoided. In order to store and distribute these images, a private registry is thus required.

A possible alternative for less sensitive images could be the utilization of a private remote repository on the Docker Hub.

Provisioning Service

The objectives include the reduction of administrative work. In order to prevent the user from having to distribute the Docker images required for the execution of workflows manually, a service should perform this task. This service should provision each machine with said images whenever such an image is created or updated. If a workflow or an activity is deleted, the service should remove its respective image from all machines.

The service could either run as an instance on each machine, performing the required Docker operations locally, or run on the Docker Swarm master machine as one instance and perform the operations remotely on all machines.

4.4.2   Communication

While the previous considerations were targeted at finding a model for the low-level communication, a way how the services communicate with each other

- likeliness of scaling - REST between non-scaling services - MOM for broadcasts / asynchronous jobs

Fig. 4.2: Deployment Diagram of the Architecture
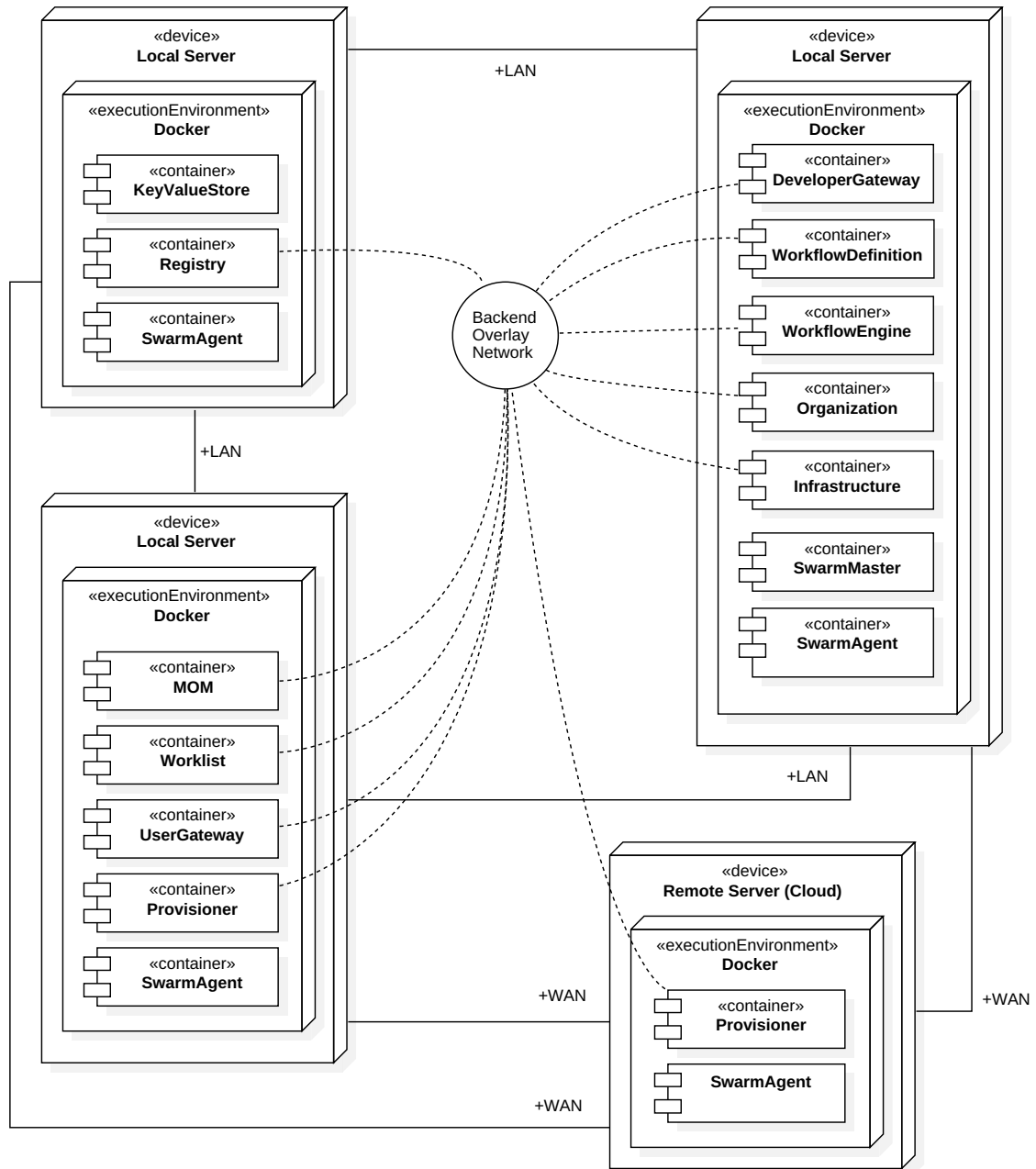
Note: the depicted distribution of containers to nodes is just exemplarily. Most of them could run on any node in the swarm. The only mandatory assignments are the swarm agents, of which each node needs one, and the provisioners, of which each node that is intended to execute workflows on needs one.

Also, the databases and their respective data volumes were omitted for the sake of clarity.

5   Prototypical Implementation

5.1   Design decisions

- GDVSEPC - why? - wf engine partially in wf container

- ruby / ruby on rails - for readability - because i know it, - easy to build restful APIs - docker library exists - drawback: speed, runtime

- extra validator - defeats purpose of not sending data in gdvsepc but, just POC

- for sake of simplicity: - no compiled language

- activity types

- export request synchronous

5.2   Execution Images

5.2.1   Workflow Image

- application - process instance - process definition - activity instance - file helper - configuration

5.2.2   Activity Image

- application - configuration - file helper

- application wrapper - manual input wrapper - custom code wrapper

5.3   System Components

5.3.1   Workflow Definition Service

The workflow definition service is composed of three components: one container running a Ruby on Rails application which is configured to expose a JSON API, one container running a PostgreSQL database and a data volume container which provides persistent storage to that database.

PostgreSQL was chosen as database solution for the definition service, because it supports both relational data, which is useful for storing the workflow, its elements and their relations, and the document-store-like JSONB format, which allows the schema-less storage of configuration information. - useful for queryable conf with dynamic structure In order to keep the stored data during container restarts or migrations across nodes, the database makes use of a data volume container which provides its working directory.

The application container is granted access to the Docker daemon of its host node in form of a mounted volume in order to build and push images.

As planned in **??**, the service has the model classes `Activity`, `ControlFlow`, `Process-Definition` and `Workflow`, which on the one hand act as object-relational mappers for persistence to the database, on the other hand ensure some validity constraints. Further, there exists a controller class for each of the aforementioned models, which provides **CRUD!** (**CRUD!**) actions for the respective model. In order to have more fine-grained control over the serialization of a workflow, the `WorkflowFullSerializer` is used if a specific workflow is requested. It nests all relevant workflow elements into the workflow model before it is serialized.
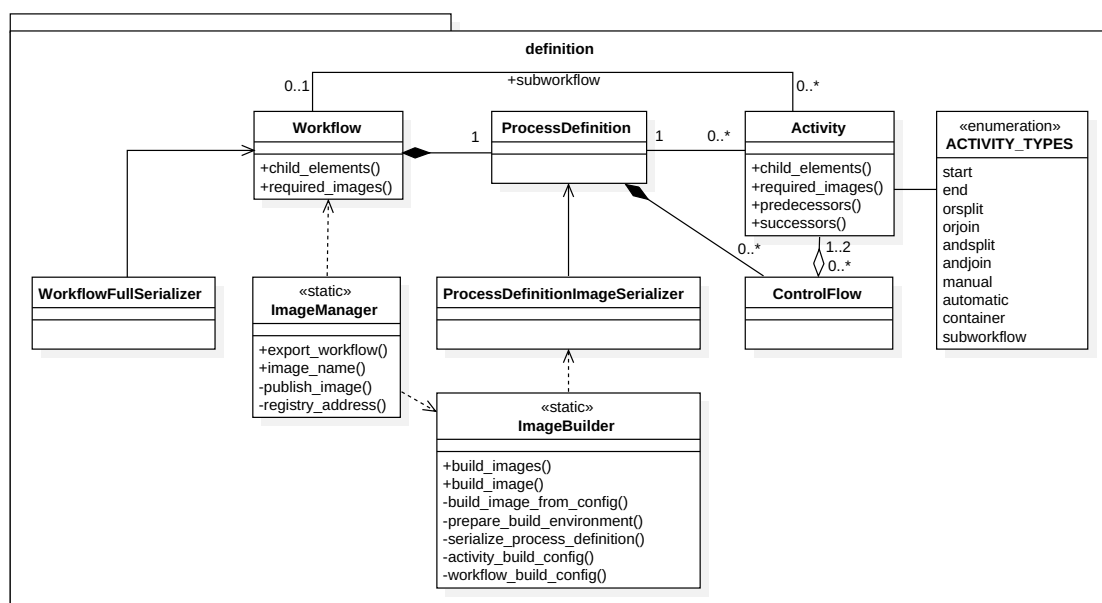
While the modeling logic is partially explicitly contained in the model and controller classes and implicitly in the underlying data schema, the export logic resides in the classes `ImageManager` and `ImageBuilder`, which are supported by the class `ProcessDefinitionImageSerializer`.

The `ProcessDefinitionImageSerializer` provides the means to create the consolidated JSON representation of a process definition with all necessary information to be interpreted in order to execute the corresponding workflow.

Export of a Workflow

If the user requests the export of a workflow, the request is forwarded to the `ImageManager`. In order to export a workflow, the first step is to identify all of its elements that require to be wrapped in an image. Obviously, one of them is the workflow itself. The other required images are determined by traversing the workflow's process definition recursively. Each activity has to be exported, too, and each sub-workflow has a method that exposes the Docker images it requires beyond that. It does so by passing the call for required images on to the referenced workflow, which collects its required images analogously, and returning the result.

The `ImageBuilder` then iterates over all these elements and creates a correspondent image for each. Starting with the respective base image – `ac_base` or `wf_base` – it adds additional

Fig. 5.1: UML Class Diagram for the Definition Service

layers to do so. The `ImageBuilder` creates the files which are specific to the current element from the activity's or workflow's configuration, i.e. the input/output validation schemata, the element's description file, and in case of a workflow the serialized process definition, in a temporary directory. The `ImageBuilder` then copies these files to the image and names it after the workflow element that it represents.

The `ImageManager` then uploads all images that were successfully built to the private repository and publishes a notification of the successful build via the message broker.

5.3.2   Organization Management Service

The organization management service is built similar to the workflow definition service. Just like it, the service consists of three Docker containers: a **RoR!** (**RoR!**)-based application that is backed by a PostgreSQL database with a data volume.

As visible in Figure 5.2, the application itself is simpler than that of the workflow definition service, though. It merely consists of the classes User and Role and their respective controller classes. The RoleController offers an additional method to draw one user form all users with a certain role at random, which can be used as a rudimentary way to distribute tasks among users.

The organization management service publishes successful CRUD events on the models to the message broker, such that the other services can react to it.
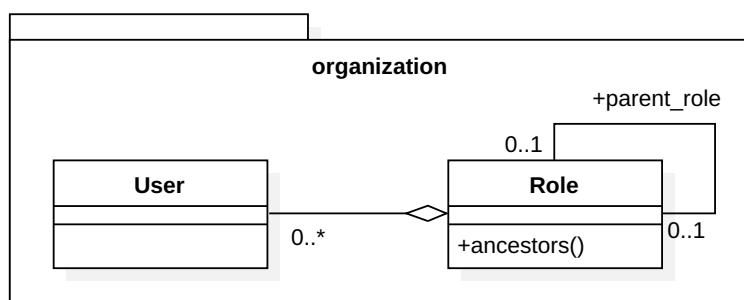
Note: Controllers omitted for the sake of simplicity. User and Role both have a controller with the respective pluralized name plus a 'Controller' suffix.

Fig. 5.2: UML Class Diagram for the Organization Service

5.3.3 Worklist Service

The worklist service is even simpler than the organization management service. It consists of three Docker containers of the same kind, too. It only features one model class and its corresponding controller: the `WorklistItem` and the `WorklistItemController`.

Worklist items may be queried by the identifier of the assigned user. They contain a JSON schema that describes the expected data and, after they have been worked on, the data itself.

TODO

5.3.4 Workflow Engine Service

- application

5.3.5 Developer Gateway

Since it only offers a unified user interface for the WfMS' services and does not store any data itself, the developer gateway does not require any database. Hence, it consists of a single Docker container that wraps a **RoR!** application. This application is realized in a two-tier architecture, with a backend part, which fetches data from the various services of the WfMS, and a frontend part, which presents that data to the developer and accepts his or her input.

- must expose ports

Backend

The only task of the backend is forwarding requests to the appropriate services and their responses to the user. The model and controller classes are thus very lean. The model classes are built upon `ActiveResource`, a **RoR!** component that provides object-relational mapping for REST web services. The workflow definition service, the organization management service and the infrastructure service all serve a RESTful API, which enables a drop-in use of `ActiveResource` possible.

Requests to the workflow engine are realized as messages sent to respective queues via the MOM. An example for such a request is the instantiation of a workflow.

Due to the Docker networking services, each service can simply be addressed by its container's name. As long as their required ports are not exposed on the host's network interface, any number of containers may be reachable on the same port. This is especially beneficial for running multiple containers that offer the same application, since they all can use their default ports this way.

Frontend

- forms for data manipulation - visual workflow modeling - infrastructure: - display available nodes + ip - display installed images and running containers

5.3.6   User Gateway

Just like the developer gateway, the user gateway does not require any storage mechanism. It thus also consists of only one Docker container, which contains the **RoR!** application.

- must expose ports

- application - backend: rails - frontend: angular app

5.3.7   Message Oriented Middleware

For this prototype, RabbitMQ was chosen as message oriented middleware, because it is well documented and has various ruby clients, e.g. *Hutch* and *Bunny*.

RabbitMQ exists as a pre-configured Docker image on the Docker Hub and can thus be utilized easily. The configuration of RabbitMQ in this image takes place when the respective con-

tainer is run, which allows its configuration in the `docker-compose` configuration file. For the sake of simplicity, no authentication mechanism was introduced besides the simple default username/password combination.

5.3.8   Infrastructure Management Service

The data that this service offers – the state of the Docker swarm and its nodes – should be up-to-date. It is thus gathered ad-hoc when a request arrives. This proceeding makes a database obsolete, that is why the infrastructure management service solely consists of one Docker container which contains the application.

The `EnvironmentManager` queries the Docker daemon and processes the response. It is supported by the `DockerHelper`, which provides the means to point the local Docker client at arbitrary members of the swarm or the swarm manager.

The infrastructure management service - application - app logic - docker helper - environment manager - controllers - servers controller - models - server

5.3.9   Registry

- one docker image - port exposed on host - deployed on coordination machine

5.3.10   Provisioning Service

- application
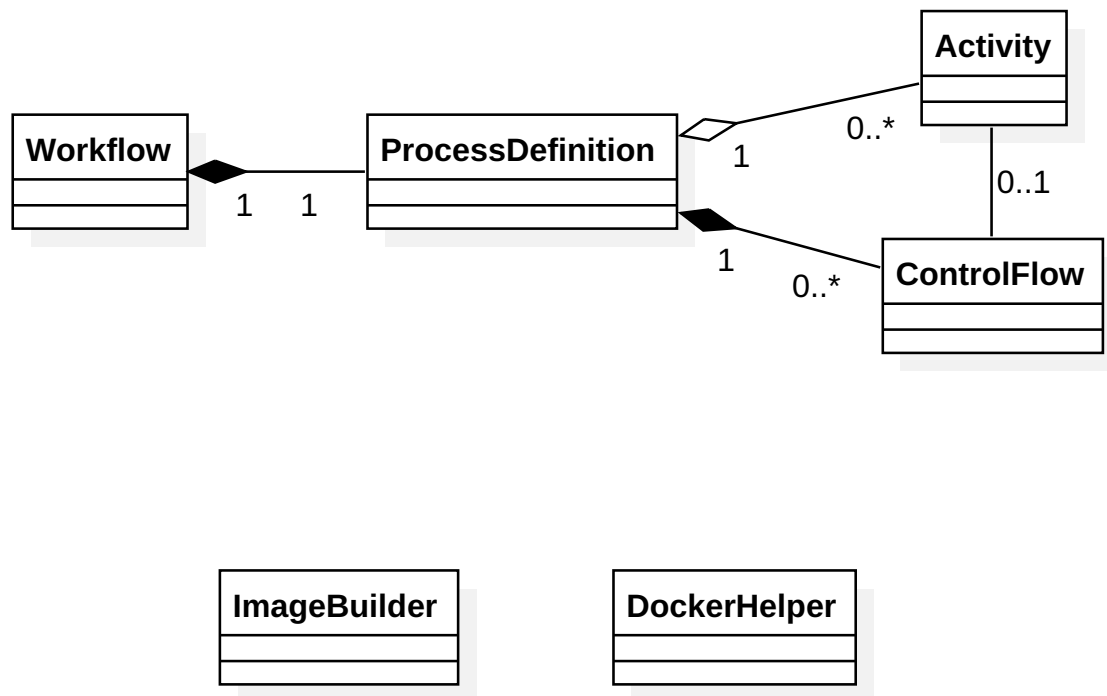
5.4   Implementation Issues

Fig. 5.3: UML Class Diagram of the Workflow Definition Service

6   Evaluation and Discussion

- other two models would be interesting to see - choice of ruby impacted performance, could be better with the worker model -

7   Conclusion

results: - three promising combinations for execution - capability table table - wfmc model translated to docker-enabled microservices - statically compiled activity/workflow would be faster

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] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *Cloud Computing, IEEE*, 1(3):81–84, 2014.

[4] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[5] 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.

[6] J. Choi, D.L. Nazareth, and H.K. Jain. Implementing service-oriented architecture in organizations. *Journal of Management Information Systems*, 26(4):253–286, 2010.

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

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

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

[10] Inc. Docker. Docker.com.

[11] Inc. Docker. Docker documentation, 2016.

[12] 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.

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

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

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

[16] Open Container Initiative. Open containers initative.

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

[18] Ian Miell and Aidan H. Sayers. How to share docker volumes across hosts, 2015.

[19] Sam Newman. *Building microservices: [designing fine-grained systems].* O'Reilly, Beijing, 1. ed edition, 2015.

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

[21] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

[22] 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.

[23] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Lois Delcambre, Christian Kop, Heinrich C. Mayr, John Mylopoulos, and Oscar Pastor, editors, *Conceptual Modeling - ER 2005*, volume 3716, pages 353–368, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

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

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

[26] Johannes Thones. Microservices. *IEEE Software*, 32(1):116–116, 2015.

[27] Dave Tucker. Docker networking takes a step in the right direction, 2015.

[28] 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.