Topic:

Prototypical Development of a Docker-based Workflow Management System

Masterthesis
in the subject
at the Department of Information Systems — Chair for 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

List of Listings

Abbreviations

API	Application Programming Interface
cgroups	control groups
CoW	Copy-on-Write
CRUD	Create, Read, Update, Delete
ESB	Enterprise Service Bus
GUI	Graphical User Interface
HTML	HyperText Markup Language
ID	Identifier
IP	Internet Protocol
IT	Information Techonology
JSON	JavaScript Object Notation
MOM	Message-oriented Middleware
MSA	Micro-services Architecture
NFS	Network file system
OASIS	Organization for the Advancement of Structured Information Standards
OS	Operating System
P2P	peer-to-peer
PaaS	Platform as a Service
PID	Process Identifier (ID)
REST	Representational State Transfer
RoR	Ruby on Rails
SaaS	Software as a Service
SOA	Service-oriented Architecture
WfMC	Workflow Management Coalition
WfMS	Workflow Management System
WSDL	Web Services Description Language
YAML	Yet Another Markup Language

1 Introduction and motivation

Organizations perform temporal and logical sequences of actions that help to interact with business relevant entities – business processes – with the objective to reach their business goals. If these processes are coordinated in an automated way, they are also called *workflows*. WfMSs are designed to support the definition, execution and monitoring of these workflows.

In the past years, enterprises reacted to their need for increased computational power by making use of Information Techonology (IT) infrastructure and software applications that are offered as services. These offers are known as Platform as a Service (PaaS) and Software as a Service (SaaS) [**?** , p. 606]. To be able run software in an environment enhanced with such services, various approaches have been presented. One of them is the use of software containers. Software containers provide a way of packaging and executing processes that isolates the application from the underlying Operating System (OS) of a computer and other processes that run on it. The concept of software containers is no new notion: an early predecessor was the `chroot` command, which dates back to 1979, *software jails* followed in 1998 [16]. In the second decade of the 21st century, solutions like Rocket, LXD, and Docker emerged, which aim at the introduction of standardized, re-usable software containers, usually in combination with tools for their management. Among these solutions, Docker is very popular. In the beginning of 2016, it was the 20th most "starred" repository on the source code management platform *GitHub* – ranking four positions behind the Linux kernel repository [**?**]. Docker comes with a set of utilities, which extend its main container-related functionality.

With regards to the challenges that heterogenous and distributed IT environments as described above impose on WfMSs – e.g. the distribution of workflows to their location of enactment, the requirement to be able to adapt to increasing workload, or manage the remote execution of tasks – it could be of interest to fathom possible benefits that may arise from the use of the Docker tool set in the context of these WfMSs. The primary objectives of the thesis at hand are thus to address the following questions and to derive artifacts from the findings that may serve as a foundation for the conceptualization and implementation of upcoming WfMSs:

RQ1: How can Docker leverage the deployment and execution of workflows in a distributed environment?

RQ2: Which decisions in software architecture and software design of a WFMS are complemented by Docker's functionality?

The structure of this thesis follows the design science research process suggested by Peffers et al. [**?** , pp. 89-92]. The research problem is identified in this very chapter and the chapters

2 and 3, in which the fundamental concepts of Docker and WfMSs are introduced. Based on considerations drawn from these concepts, the objectives of a solution are inferred and a prototype is designed in Chapter 4. The implementation and usage of the prototype are described in Chapter 5. In Chapter 6, the developed mechanism and the prototype are evaluated. Finally, the findings of this thesis are summarized and suggestions for subsequent research are presented in Chapter7.

** related work

2 Workflow management systems

In this chapter, the concepts of workflows and workflow management systems will be introduced briefly and related to each other. There is a plethora of term definitions and deviating understandings of workflows and the concepts related to them [1]. This Unless noted otherwise, the concepts presented in this chapter thus rely on specifications published by the Workflow Management Coalition (WfMC), a consortium of WfMS vendors, researchers in the field of workflow management and WfMS users, which the authors claim that it "describes a common model for the construction of workflow systems and identifies how it may be related to various alternative implementation approaches" [3].

The identified 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**?**]. The WfMC defines workflows as "the computerized facilitation or automation of a business process, in whole or part" [**?**].

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 [**?**]. 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 to an abstracted model [**?**]. This model is usually called *process definition* and stored in form of some high-level programming language con-

struct [**?** 4]. 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 [**?**]. They further may contain references to other processes, which are treated as a single activity in the process definition [**?** 1].

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. [1]. If only the automated parts of such an instance are meant, the WfMC advocates the term *workflow instance* [**?**].

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 [**?**]. A graphical representation of the state transitions described above can be seen in Figure **??**.

2.1.4 Activity instance

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 enters the *completed* state [**?**]. 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 are distinguished, as they serve different purposes. The WfMC differentiates between three types of data: workflow relevant data, workflow application data, and workflow control data [**?**].

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 [**?**].
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 [**?** 1].
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 [**?** 1].

Russel et al. differentiate seven commonly used forms of data visibility in WfMSs [5, 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 of 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.

Russel et al. identified six further types of data interaction between the various hierarchy levels in workflows [5, 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 either be made available from a common datastore, get passed along with the control flow of a workflow, or be explicitly passed to the receiving component [5, 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. The assignment can either be a specific one, targeting an individual person, or be more general, targeting a set of users from which the WfMS may choose during execution time. These sets are usually based on an organizational structure that manifests itself in roles, of which a user may have one or more [**?** 1].

Each user owns a so called *worklist* that consist of activities to which he or she 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 or she will work on it **ROLE ID in prototpe user fetach their stuff** [**?** 1].

2.2 Typical architecture

For large and complex organizations, the need arises to manage the creation, distribution and execution of workflows in a structured manner. An information system is a 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;
- it is able to interpret process definitions;
- it can interact with involved participants; and
- it may invoke external applications [6].

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" [**?**]. The components of this system 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.

** add this 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 responsibilities of a WfMS in three functional areas: *build-time* functions, *run-time process control* functions and *run-time activity interaction* functions [**?** 7].

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 [**?**].
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 any resulting outcomes [**?**].

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* [**?**].

Process definition tools are designed 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 [**?**].

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.

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

In order to enact workflows, instances of them are created 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 [**?**].

The **workflow enactment service** groups one or more workflow engines into one logical component that exposes a single coherent external interface to other software [**?**].

3 Software containers and 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 [9]. The difference between these two options and the basic principles of software containers are shown in 3.1.1 to give an understanding of the technology.

Docker is a tool that is intended to facilitate the creation and management of software containers. In Section 3.1 its underlying concepts will is 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 functionality of Docker, is introduced in Section 3.2.

3.1 Concepts

First, the concept of software containers will be presented and briefly contrasted against the concept of virtual machines to explain *what* Docker does and to identify the possibilities it offers. Internal constructs of Docker – images, containers, data volumes, dockerfiles, registries and repositories – are then introduced 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 ***. There are two types of virtualization, one that takes place on the hardware level and another that takes place on the OS level [9].

Hardware-level virtualization 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 runs in the OS of the host machine or directly on its hardware [9].
Virtual machines, i.e. the computers simulated on the host machine, require their own OS to be installed.

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* [9].

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 IDs (PIDs), and other components, in order to achieve a fine grained control over the respective isolation [9]. Besides Docker, there are several solutions that are based on the aforementioned kernel features, e.g. LXC, LXD, lmctfy, systemd-nspawn, etc. [9]. There are ongoing efforts to create a common container standard [10]. ** warum docker? -> popular, ecosystem

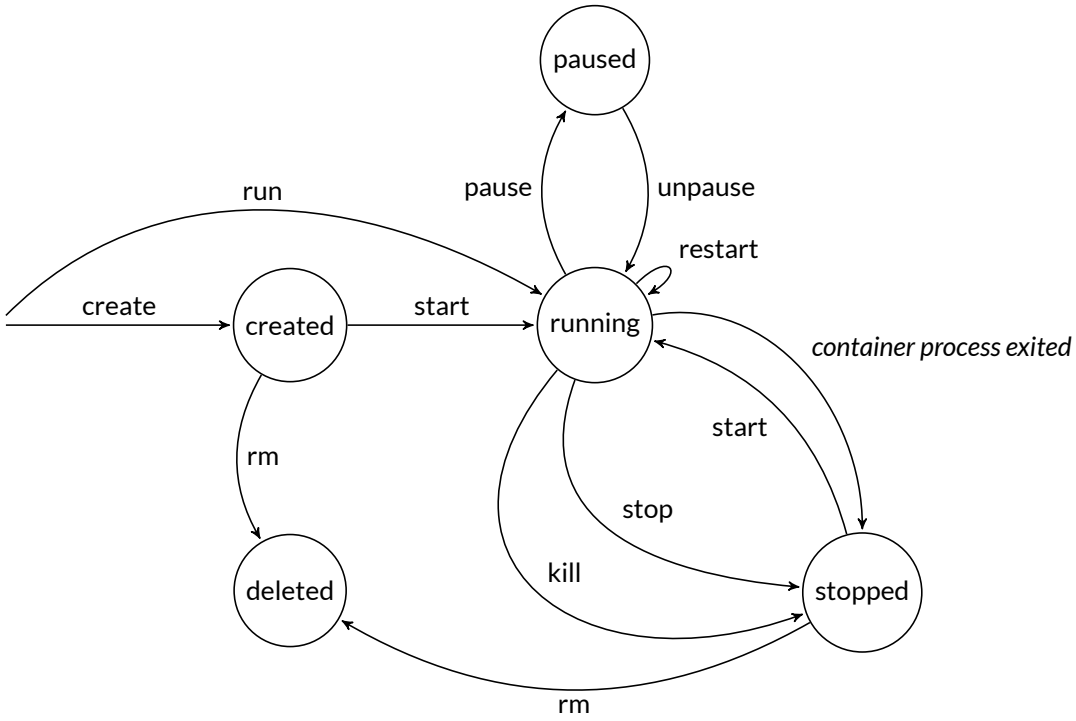
Many container solutions rely on a mechanism called *Copy-on-Write (CoW)* to provide a runtime enviroment, which on the one hand lets the containers reuse system libraries and the like while on the other hand limits the container in affecting its surroundings [11, 12]. CoW is an optimization strategy that 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 that only the process has access to. All other processes still use the original file [12, 11].

3.1.2 Docker Images and Containers

Docker images (referred to as *images* from here on) 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 files contained in the respective image [11].

** lifecycle and commands



Simplified version. Commands should be understood prefixed with `docker`.
Based on https://docs.docker.com/engine/reference/api/docker_remote_api/#docker-events

Fig. 3.1: Docker Container Life Cycle [18]

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 [13]. They are never deleted automatically and therefore must be cleaned up manually if they are not needed anymore [11].

3.1.4 Dockerfiles

Instead of manually creating a container, running commands on it and then commiting it to create an image, a recipe file can be used to instruct Docker to perform these actions – the *Dockerfile*. In this file, the user states the name of an image that the new image should be based on and the commands that otherwise would be entered manually [13]. To build an image, Docker is given a Dockerfile and a directory with files required for the build, the *build 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 [13].

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 [11].
Within a registry, images may be organized in collections, which are called *repositories* [13].

** push pull

3.1.6 Docker Engine

** enables the previously mentioned features The Docker Engine forms the core of Docker. Docker uses a client-server architecture: it features a daemon, i.e. a background process not directly controlled by the user, which provides the functionality and a client that controls said daemon [14]. 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) [11].

3.1.7 Docker Networking

Since version 1.9, which was released in November 2015, Docker features virtual networks in order to isolate containers concerning their network connections, 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 connected to multiple networks at the same time [11].

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 not instructed otherwise. Containers that are members of this network can communicate with each other by using their respective Internet Protocol (IP) addresses. They also may expose ports that can be mapped to the host's network, which makes the applications that run in them accessible from the outside. The *host* network represents the actual host's network. If containers are assigned to this network, they will be placed in the host's network stack, i.e. all network interfaces defined on the host are available to the container [11]. 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 [11].

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 con-

nections. 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 [11].
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 ecosystem

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

3.2.1 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 assigns containers to a specific host for executiion based on given rules [15].

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 [11].

Docker Swarm provides two kinds of mechanisms for the assignment of containers to Docker hosts, *strategies* and *filters*. 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 [11].

3.2.2 Docker Machine

The Docker Machine tool is designed to facilitate the setup of Docker hosts. In order to fulfill this goal, Docker Machine creates one virtual machine per requested host [15, 11]. This has several reasons. First, this proceeding allows several Docker hosts to run on the same computer

while prohibiting them from interfering with each other. Second, it enables computers with OSs that natively do not support Docker and Docker containers, to act as a Docker host [11]. 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.

3.2.3 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 descriptive file. Docker Compose interprets this file and sets the containers up accordingly [15].

** - build w/ image name - build on specifc node - up command - build missing images - start stopped/missing containers - recreate containers with changed images - ignore existing containers

3.2.4 Docker Hub

4 Conceptual development of the WfMS

In order to make sound decisions in the design process for a Docker-based WfMS, 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.

The potential benefit WfMSs could obtain from using the Docker ecosystem is twofold. On the one hand, the distribution and execution of workflows and their components can be enhanced, which is addressed in Section 4.2. On the other hand, the mode of operation of the WfMS itself might be improved by the use of Docker. Based on the determined objectives, the architecture of a Docker-based WfMS is thus shaped in Section 4.3 and subsequently its design in Section 4.4.

4.1 Determination of objectives

In this section, the objectives for the design and implementation are inferred from considerations regarding the desired functionalities of a Docker-based WfMS as well as its intangible properties.

4.1.1 Functional objectives

In the following, expectations towards the functionality of the resulting WfMS are established in a structured manner. These functionalities are grouped by the component groups of a WfMS, which are described in 2.2.2. The resulting objectives and the requirements that need to be met in order to fulfill them are summarized in Table **??**.

** TODO: abgleichen

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. Any failure of a business critical WfMS can thus cause severe problems for an organization. The WfMS developed in this thesis should thus be resilient towards failures, i.e. provide as

Objective	Requirements
Ability to alter components	• Components can be altered on a running system
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
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
Resource usage management ** not implemented**	• 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

Data Source	Criterion Type	Examples
Docker	Node tags	Name ID Storage driver Execution driver Kernel version Operating system
	Qualitative node properties	Geographic location Custom grouping membership Ownership status
	Quantitative node properties	Available memory Available storage Network speed
	Execution environment	Existing containers Existing images Currently running containers
WfMS	Live data	Enactment status
	Activity input/output data	Workflow input/output data
	Activity/workflow properties	Expected data-intensity Compliance requirements

Tab. 4.2: Scheduling Criteria

Requirements	**Feature**
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 → activity • Sub-workflow activity → sub-workflow • Sub-workflow → sub-workflow activity • Workflow instance → workflow instance

Tab. 4.3: Required data visibility and data interaction types

much functionality as possible if a part of it fails and try to recover autonomously. This requires well separated modules.

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 [16, p. 82]. The result can be considered as a black box that provides some specific functionality and that could be used without further configuration. In combination with the facilitated sharing of images through repositories, this provides a foundation for modular reuse and combination [17, p. 6]. In order to reap this advantage, WfMS should 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.

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 a WfMS's life cycle, many workflows are modeled and many activities are created, and both are likely to be updated occasionally. 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. 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 [5]. The 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. The resulting capabilities are presented in Table 4.3.

4.1.2 Intangible objectives

** extra section notwendig?

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 worked out in 4.1.1 lose value if using them is cumbersome and they are avoided in consequence. Concerning the scope of this thesis, the functionalities that should be facilitated are

- the modeling of workflows;
- the export and distribution of workflows;
- the selection of images from Docker Hub for the use in a workflow;

4.2 Docker in workflow execution

There are several possibilities how Docker can be utilized for the execution of workflows. Each combination of variants (abbreviated as depicted in Table 4.4) 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, it can be differentiated to which extent 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 a single container ($*^*_{1C}$). Since such an container is an atomic unit, it cannot be spread across many nodes for execution. The idea of an one-to-one mapping between the Docker and workflow concepts could be abandoned in favor of a solution that features worker containers with specialized behavior which perform suitable tasks on request ($*^*_{WORK}$).

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, the mechanism that decides which containers are run on which machines, i.e. the execution scheduling, can be chosen.

TODO: table consistent with text?

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}
Worker containers	G_{WORK}^{DV}	G_{WORK}^{SER}	G_{WORK}^{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	✗	✗	✗
Worker containers	✗	S_{WORK}^{SER}	S_{WORK}^{D}

Tab. 4.4: Containerization/Grouping/Communication Solution Pairings

4.2.1 Properties and mode of operation of the utilization variants

mehr einleitung hier In the following, the concepts are explained in more detail and some variants on them are given. Further, special interrelations among these variants are highlighted.

Grouped or spread execution

The "grouped and spread execution" variants are about the allocation of containers related to one workflow instance on the available nodes. While 4.2.4 deals with the mechanisms behind the scheduling, the larger concept of grouped or spread execution will be focused here.

On the one hand, the containers could be assigned to different nodes (S^*_*). This could happen at random, with regards to characteristics of the containers or their underlying workflow elements, or based on the current workload of the nodes. Balancing the workload and matching containers to specific nodes could have a positive impact on the performance of the execution. Negative effects might arise from the introduced network latency, though. Also, spread execution requires the images that belong to the workflow elements in question to be present on all nodes.

On the other hand, the first container of a workflow enactment might be assigned to one node and all subsequently started containers are launched on that same node (G^*_*). While this reduces the ability to balance the workload across nodes, it could be beneficial for the speed of communication between containers, since no transfer via network is required.

Element-wrapping Images

The general idea behind this set of concepts is that activities and workflows can be represented by images, whereas containers can represent activity instances and workflow instances.

As described in 2.1, activities can be perceived as self-contained units of work, that is, they contain the information on how to autonomously perform a task on a given set of data. Analogously, Docker images contain the information on how to run processes in an instance of themselves – a container – given a set of parameters. If the work that an activity performs can be manifested in program code, this code and its required runtime environment could be contained in a Docker image. An instance of that image that is created with a specific set of data would then be the counterpart of an activity instance. This notion is the foundation for the $*^*_{AC}$ and $*^*_{SEPC}$ variants.

A similar conception can be applied to workflows: they contain the information that is necessary to perform sequences of tasks in an automated way. The sequence order is usually given in a formalized fashion by a process definition. In the given mindset, this process definition would be a list of Docker images in combination with a formal description of the control flow and data flow between these images. A workflow can thus be seen as a set of appropriate images and a corresponding process definition. A workflow instance, in turn, would consist of the relevant

containers, the data that is used and created by these containers, and the enactment state of the workflow. These conceptual mappings are summarized in Figure 4.1.

Like activities, workflows can thus be represented by images. Such a workflow image could contain the process definition of the respective workflow and other data that is specific to it. Alternatively, the workflow data may be passed to the workflow engine for enactment in another format. The variant in which the workflow information is distributed in a separate image along with the activity images is referred to in the following as $*^{*}_{SEPC}$, while the variant in which the workflow information is passed along otherwise is denoted as $*^{*}_{AC}$. Theoretically, a workflow could be completely encapsulated in an image, i.e. with all its related activities and its configuration. This case is referred to as $*^{*}_{1C}$.

Activity	→	Image
Activity instance	→	Container
Process definition	→	List of images + control flow + data flow
Workflow	→	Activity images + process definition
Workflow instance	→	Activity containers + data + enactment status

Fig. 4.1: Possible Mapping of WfMS and Docker Concepts

Wrapping activities – and optionally workflows – in Docker images creates new possibilities in regard to their distribution and enactment. Images can be uploaded to public or private image registries to provide a standardized way for the deployment to nodes. This offers a way to make the deployment available as a service. Nodes can be instructed to contact these registries in order to be provided with required images or update existing ones.

The layering principle behind Docker images makes updates to activity and workflow images lightweight – as long as only the uppermost layers are changed, the lower ones do not have to be up- and downloaded again. This allows to distribute a complete runtime environment to new nodes while updating only specific layers on existing nodes using one and the same image.

Containers can be paused and unpaused, i.e. all processes within them are suspended. This is useful to save processor capacity for suspended activities or workflows until they are resumed.

** Second, $*^{SEPC}_{*}$ leverages the already existing communication between a container, the local daemon and the swarm master to communicate the status of workflow instances and activity instances, as the creation of a container

more benefits here

One possible variation on G^{DV}_{1C} and G^{DV}_{SEPC} would be the inclusion of engine logic in the workflow instance container. While the scheduling of this container and preparations on the targeted

node would still be in the responsiblity of the main WfMSs' workflow engine, the control over the started activity instances would be handed to a smaller, workflow instance specific engine. This setup facilitates the management of the required directory structure, since the container that issues the commands has direct access to the data volume – no information on the upcoming element instances has to be transferred.

Further, in this case suspension and resumption of workflows and activities can both be realized with the respective pause/unpause Docker commands. In combination with a checkpoint/restore tool, which provides the means to save and restore the memory state of a process, long running workflows could be paused and restored across server restarts, or be migrated to another server, if necessary. Proofs of concept for the feasibility of this procedure have been presented in the past [19, 20].

A drawback of this variant is, that the status of the enactment is then held in the workflow instance container. By following the event stream, started and stopped activity instances could be tracked by the main workflow engine, but low-level enactment details would have to be transferred to the engine separately, if they were needed there.

Worker containers

The $*^{*}_{WORK}$ variants take a different approach at utilizing Docker for workflow enactment, in which many unspecialized containers are running permanently. These containers, when provided with an activity description and input data, perform the required actions and return to a waiting state until they are provided with a task again.

In contrast to the variant outlined above, worker containers bear no one-to-one correlation of an image to an activity or a workflow, and also none of a container to an activity instance or a workflow instance. Hence, there is no further creation and distribution of images required – besides the initial distribution of the workers' images (and invoked third-party images). Since all information that is required for execution is passed to these workers at each invocation, changes to definitions of activities or workflows may immediately show effects. This flexibility comes at the price of a verbose communication, though.

Another benefit of worker containers is that they facilitate swift adaption to workload peaks. In case that scheduled enactments start to accumulate, more worker containers may be run. If the nodes themselves have reached their resource limitations, new nodes may be added to the swarm, which do not have to be provisioned with all activity or workflow images – as it would be the case with the wrapping-images variant – but only with the worker image.

Data exchange via data volume

The idea behind this concept ($*_{*}^{DV}$) 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 belongs exclusively to the respective workflow instance and whose only purpose is to ensure the existence of said volume.

In its simplest form the working directory could be a simple shared directory which is managed cooperatively by all related containers. Activity instances could then read and write files to that directory. **PRO/CON?**

A more elaborate structure could be imposed on the working directory, too. In order to support the data visibility and data interaction types that are chosen in 4.1.1, the following directory structure could be used, which is depicted in Figure 4.2. Data defined at build time, i.e. environment data, workflow data and activity data, could each be stored in a separate subdirectory of the main working directory `workflow_relevant_data`. The directories for workflow data and activity data should have uniquely identifyiable names that can be inferred from the respective workflow element, e.g. `wf_$workflow_id` and `ac_$activity_id`. Data that is shared among multiple instances of one activity could be stored in a subdirectory `shared` of the respective activity data directory.

Also present in the working directory should be a directory `wfi_$workflow_instance_id`, in which the working directories for subsequently started activity instances and workflow instances can be stored. Each activity instance is then assigned a directory `aci_$activity_instance_id` within that that working directory. In case that the activity instance in question is a sub-workflow activity, the respective workflow instance's working directory resides in the sub-workflow activity instance's working directory. This principle can then be applied recursively, as visible in Figure 4.2.

Depending on the desired level of isolation, the instance containers could be instructed to mount either a) the whole `workflow_relevant_data` directory or b) just their own working directory, the directories that contain the data defined at build time, and working directories of activity or workflow instances which they were configured to use. While the former is a much simpler solution, the latter gives more fine-grained control over the data that each instance is allowed to access.

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.

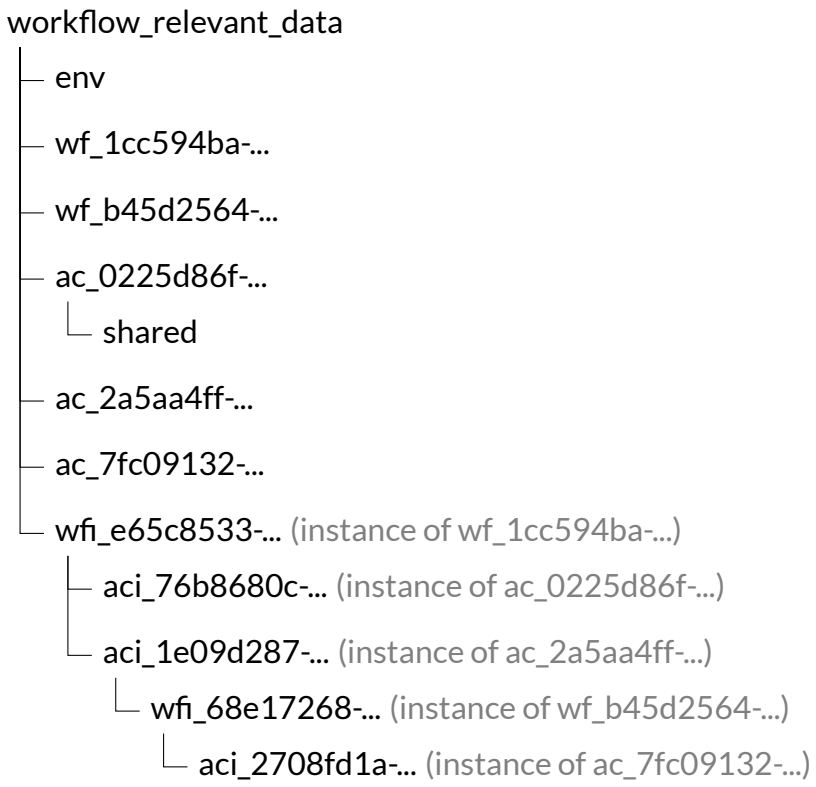


Fig. 4.2: Exemplary directory structure for G_*^{DV}

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 instances is not possible this way, since mounting volumes to running containers is not supported by Docker yet. This kind of interaction thus requires additional means of communication. 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 instance. 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 no spread version S_{*}^{DV} of this concept exists. Approaches exist to utilize Network file system (NFS) or peer-to-peer (P2P) file sharing for distributed access to data volumes, though [21].

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. That would make the container self contained – and thus easier to export or migrate – but would couple the data life cycle to the processing container's life cycle.

In order for G_{WORK}^{DV} to work successfully, it is inevitable that only workers on the same node as the data volume are used for the workflow enactment, since workers on different nodes could not access that data volume.

Data exchange via service

$*_{*}^{SER}$ represents a concept in which a service is provided that is able to store and serve workflow relevant data on request. This implies, that the instance containers have to feature a mechanism to communicate to this service.

The data-providing service could either be running as a single instance on one node in the network or as multiple instances, one on every 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 and decrease the response time, as less network hops would be required to contact a service instance.

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. Theoretically, all forms of data visibility and interactions should thus be possible. Also, $*_{*}^{SER}$ exhibits the same properties for grouped and spread execution alike, 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 data-providing service must be reliably available to all containers.

Data exchange via direct communication

In this variant, 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 ($*_{*}^{D_a}$). On the other hand, one where containers can query each other for their data($*_{*}^{D_b}$).

In $*_{*}^{D_a}$, the data flow is is directly coupled to the control flow, i.e. all data is passed along on invocation. This requires all data that might be used in another activity instance to be passed on the succeeding activity, no matter whether the activity uses them or not [5]. While this variant allows to pass (and update) workflow and environment data, it may be problematic for larger amounts of data.

$*_{*}^{D_b}$ 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 the workflow instance in question has to be kept 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 $*_{*}^{D_b}$ 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 $*_{AC}^{D}$, $*_{AC}^{D_b}$ provides no means to store and access sub-workflow data, multiple instance data and workflow instance data.

$*_{WORK}^{D_b}$ is no useful solution, as keeping the worker containers occupied with one activity 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.

4.2.2 Identification of promising combinations

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

	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_{WORK}^{DV}	✓	✓	✓	✓	✓(†)	✓(†)	✓	✓	✓	◯
G_{AC}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G_{SEPC}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G_{1C}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G_{WORK}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$G_{AC}^{D_a}$	✓	✓	◯	✓	✓	✓	✓(‡)	◯	◯	◯
$G_{AC}^{D_b}$	✓	◯	◯	◯	◯	◯	✓(‡)	◯	◯	◯
G_{SEPC}^{D}	✓	✓(‡)	✓(‡)	✓(‡)	◯	◯	✓(‡)	✓(‡)	✓(‡)	✓(‡)
G_{1C}^{D}	✓*	✓*	✓*	✓*	◯	◯	✓*	✓*	✓*	✓*
G_{WORK}^{D}	✓	✓(‡)	✓(‡)	✓(‡)	◯	◯	✓(‡)	✓(‡)	✓(‡)	✓(‡)
S_{AC}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_{SEPC}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_{WORK}^{SER}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_{AC}^{D}	✓	◯	◯	◯	◯	◯	✓(‡)	◯	◯	◯
S_{SEPC}^{D}	✓	✓	✓	✓	◯	◯	✓(‡)	✓(‡)	✓(‡)	✓(‡)
S_{WORK}^{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. 4.5: Supported types of data visiblity and data interaction by the variants

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

The choice of the data volume approach implies a commitment to grouped execution on one node. Considering the different containerization solutions, G^{DV}_{SEPC} is favorable over G^{DV}_{AC}, 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^{DV}_{SEPC} is also favorable over G^{DV}_{1C} for general use, because the modularity of G^{DV}_{SEPC} permits to update single activities within the workflow by distributing new versions of their respective image. In G^{DV}_{1C}, the whole image would have to be updated due to the way layering in Docker images works. Also, G^{DV}_{1C} 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, which is sold to be used by photographers, this variant could be a viable option, however.

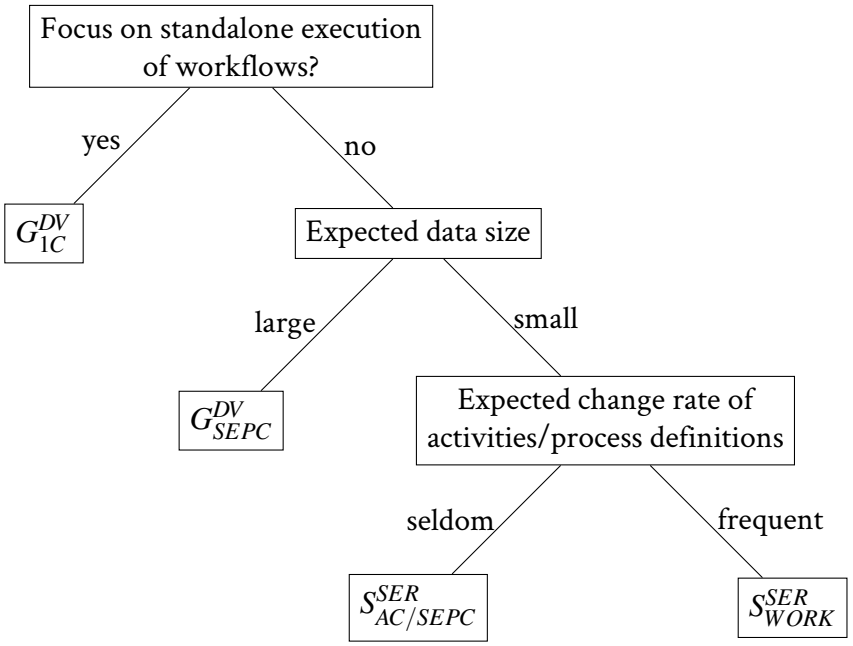
The main benefit of $*^{*}_{WORK}$ is that it allows to distribute the workload among several workers. Since G^{DV}_{*} requires all involved workers to be on the same machine, this advantage is void for the combination G^{DV}_{WORK}.

Altogether, this makes G^{DV}_{SEPC} the variant of choice for data-intense use cases.

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 the two.

Because the decoupling of data and containers is well supported by $*^{SER}_{*}$, it is a good fit for the worker-based approach of $*^{*}_{WORK}$ which in turn facilitates load balancing. A use case for S^{SER}_{WORK} could be WfMS users like market research companies, which create lots of frequently updated form-centric workflows for call center agents to use, i.e. fast-changing workflows with rather small data that has to be passed.

The results of the above reasoning are summarized in a decision tree in Figure 4.3 which is intended to give a hint on the suitable utilization of Docker for workflow enactment for a given use case.



Note: 'Large' and 'small' are rather vague terms. In the end, the suitability is left to be determined based on the size of the data in relation to the achievable transfer rates.

Fig. 4.3: Choice of the right utilization of Docker for workflow enactment

** figure broken!

4.2.3 Utilization of third-party images

The motivation behind enabling the use of third-party images is that any program able to run in a Docker container can be incorporated in a workflow this way. In 2015, over 125.000 public repositories existed [22].

In order for these images to be used by a workflow instance, they have to be present on the node that they will run on. The provision of nodes with the required images could either happen actively, i.e. the node is instructed to pull a specific image, thus qualifying the node for the execution of said image, or passively, i.e. the node is selected for running the image in question and fetches it in the course of running it. While the the former solution limits the number of nodes that are able to run the image – unless every node is provisioned with it – the latter solution is likely to delay the enactment of the workflow at its first use for the time it takes to pull the respective image.

As third-party images are likely to be unaware of their utilization in a workflow, it is not guaranteed that their output suits the needs of the WfMS. Also, these images may require some parametrization for their instantiation. On the one hand, both issues could be approached by the workflow engine, which might transform the output and provide suitable parameters based

on the activity's configuration. On the other hand, an utility activity could be introduced that can be configured such that it is able to instantiate the third-party image and transform its output to a suitable format. The latter solution should be preferred, as it shifts the responsibility for the aforementioned tasks away from the workflow engine, which reduces the engine's complexity. The engine does not need to differentiate between custom and foreign images in this case.

In order to be able to instantiate the third-party image, the adapter image requires access to its host node's Docker daemon.

4.2.4 Execution scheduling

In the course of a workflow enactment, several containers have to be instantiated – unless the worker-based approach ($*^*_{WORK}$) is chosen. Choosing the node on which the instantiation takes place is a task that may impact the performance of the containers and the enactment itself, as the nodes may differ regarding their available resources. It is thus of interest to examine by which rules or criteria the scheduling may take place and whether and how the user should be able to take influence on the scheduling.

Scheduling abilities of Docker Swarm

As described in 3.2.1, Docker Swarm offers two kinds of scheduling mechanisms: filters and strategies.

Filters can be passed on instantiation as an environment variable parameter with the format `<filter-type>:<key><operator><value>`. The value of `filter-type` can either be "constraint" or "affinity" – the two types of filters supported by Swarm. `<key>` can take the values *node* or *container* (which signals a comparison to the respective name or ID), one of the default node tags, or the name of some custom label which can refer to both node labels and container labels. By default, nodes get tagged by Docker with a name, their ID, their storage driver, their execution driver, the kernel version they use and the name of their operating system. Custom labels may be applied to nodes when their daemon is started and to containers as a parameter to the `run` command. In order to avoid conflicting label names, reverse domain name notation is advised by Docker. `<operator>` may either take the value `==` or `!=`, indicating whether a match is desired or should be avoided. It can be followed by a tilde, e.g. `==~` to signal that if the condition cannot be met, the container should be scheduled according to a strategy instead. The `<value>` is a string made of alpha-numeric characters, dots, hyphens,

and underscores. It may be either a regular expression or a globbing pattern, to which names, IDs tags or labels will be matched against.

Constraints focus on the characteristics of nodes for scheduling; either a) some identifier or b) node tags or c) labels:

a) `constraint:node==...`
b) `constraint:operatingsystem==...`
c) `constraint:com.example.label==...`

Affinities can be specified to schedule containers based on the presence of images or containers on the target node. Possible criteria are a) container names or IDs, b) image names or IDs or c) a custom label on a container:

a) `affinity:container==...`
b) `affinity:image==...`
c) `affinity:com.example.label==...`

Further, there are some implicit forms of scheduling. First, containers will not be executed on nodes that Docker Swarm considers *unhealthy*, i.e. that do not respond or otherwise exhibit faulty behavior. Second, some Docker features imply scheduling rules. If the container requires an exposed port, it will not be scheduled on nodes where this port is already occupied. Mounted volumes of, a shared network stack with or a link to another container imply an affinity to that container, which will prohibit the execution if it is not met.

** strategies

Identification of possible scheduling criteria

** move intro + types to objectives chapter?! In order to examine scheduling solutions, goals should be specified to which they can be evaluated against – but an exhaustive list of these goals is not in the scope of this thesis. Thus, two groups of exemplary scheduling criteria are considered, which are depicted in Table **??**.

Scheduling may be performed based on the properties of the nodes, which can be of qualitative or quantitative nature, their tags or the execution environment they have to offer. Examples for qualitative node properties could be its name, present containers or images, geographical location, the nature of its storage device, or its membership in some arbitrary groups. Quantitative properties of interest might be the amount of available memory or storage, the number of currently running containers or the node's network speed.

As an alternative to the above methodology, containers may be assigned to nodes based on WfMS-specific data. This could be for example properties of the underlying activities and workflows, the status and data of their instances or the state of a custom scheduling mechanism of the WfMS. For example, if a user specifies his/her current location in the course of a workflow enactment, all subsequent containers could be scheduled to run on nodes that suit the laws of that country.

Data Source	Criterion Type	Examples
Docker	Node tags	Name ID Storage driver Execution driver Kernel version Operating system
	Qualitative node properties	Geographic location Custom grouping membership Ownership status
	Quantitative node properties	Available memory Available storage Network speed
	Execution environment	Existing containers Existing images Currently running containers
WfMS	Live data	Enactment status Activity input/output data Workflow input/output data
	Activity/workflow properties	Expected data-intensity Compliance requirements

Tab. 4.6: Scheduling Criteria

Both qualitative and quantitative properties of a node can be stored in custom labels on that node. Qualitative values can simply be stored as strings and used in constraints via globbing and regular expressions. Assuming, for example, the nodes had their location stored as a label with the format

```
com.example.location=$country
```

with `$country` being the respective ISO 3166-1 alpha-2 code [23]; and the information whether they internal or third-party servers can be stored as a label with the format

```
com.example.internal=$internal
```

with `$internal` being the string representation of a boolean value. Then, in order to sched-

ule a container to a self-owned node in Germany running any version of the *Ubuntu* operating system, the following combination of tag and label constraints could be used:

```
constraint:operatingsystem==Ubuntu*
constraint:com.example.location==de
constraint:com.example.internal==true
```

Quantitative values stored in a label may be queried with comparisons using regular expressions. Assuming all nodes have a label that specifies their amount of memory, which has the form

```
com.example.ram=$mem
```

with `$mem` being the amount of memory in megabytes. In this setting, a container can be scheduled to a node with at least 650 megabytes of memory by enforcing the constraint

```
constraint:com.example.ram==/(\d\d\d\d+|[7-9]\d\d|[6][5-9]\d)/
```

This regular expression will match a) any number with four or more digits and b) any three digits number that starts with a number in the range of 7-9 and c) any three digits number that starts with 6 and continues with a number in the range of 5-9. Analogously to this, other quantitative measures may be stored and queried.

Scheduling constraints regarding present images and containers can be realized using the appropriate affinity filters. One possible use case is to ensure, that all images and containers required for the enactment of a workflow element are present:

```
affinity:image==required-image-name
affinity:container==required-container-name
```

At the current development state of Docker Swarm, it is not possible to verify that the containers are not only present but also running. If required, this criterion has to be addressed by some custom mechanism of the WfMS, e.g. a method that queries the swarm master's API, selects a suitable node with running instances of the required container and constructs an according node constraint.

Since updating labels and tags of containers and nodes after their creation is not possible at the time of writing, all WfMS related data created during enactment has to be sent to the system and managed internally by it.

Properties of activities and workflows that have been defined at build time may be stored in the form of labels when they are created. This is possible by using the `LABEL` command for desired labels in the respective Dockerfile in combination with passing their value as an argument for the build command:

in Dockerfile:

```
LABEL com.example.expected-data-intensity=$data-intensity
```

in command line:

```
docker build [...] --build-arg data-intensity=high [...]
```

In contrast to other examples given in this thesis, `$data-intensity` should not be understood as a placeholder, as it is the actual Dockerfile syntax for the interpolation of a passed build argument with the respective name.

The workflow engine could then use the image's labels to take the stored properties into account for scheduling. For complex properties, labels containing serialized JSON objects could be used [13].

Implications of other design decisions

WfMSs may let workflow modeling developers take influence on the scheduling of workflow containers by letting them define constraints or affinities on them at build time. This could be beneficial because of the user's potential a priori knowledge on these elements, e.g. whether they are data-intensive and thus require some large storage, or because it enables meeting compliance rules, e.g. concerning the geographical location of data storage and processing.

Since the G_*^{DV} approaches are grouped on one node by definition, the scheduling takes place for the first container only, while subsequent containers are bound to the same node by the implicit affinity created through the mounted data volume.

There is no need for scheduling in the sense of managing container startups in the $*_{WORK}^*$ use case, since the workers are already running. Restrictions could be imposed on the set of workers that are allowed to work on a task, though.

4.3 System architecture

With the objectives determined in Section 4.1 in mind, a Docker-based architecture is developed in this section. First, possible architecture styles are presented in 4.3.1, of which one is then chosen with regards to potential benefits in combination with Docker. Subsequently, the way how users interact with the system is chosen in 4.3.3 and the high-level mode of communication between containers in 4.3.4.

** move to grundlagen?

4.3.1 Architecture styles

Developers of software systems have to cope with challenging factors 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 cope with 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 originate 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 as well as 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 [26, p. 5].

Service-oriented Architecture

SOA is based on the idea that code which provides related business functions can be bundled into one component that offers said functionality to other systems *as a service*, thus avoiding duplicated implementation of the functionalities among these systems [27, p.8]. An application may then use several services in order to fulfill its own business function [28, 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 [29]. 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* [28, 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 [28, 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 [28, 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 [28, 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 [28, p. 390]. SOA thus enhances an organization's ability to respond quickly to changes [28, p. 390], [30, p. 254]. Since legacy applications can be provided with appropriate interfaces, SOA can help to integrate and extend them [28, 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 [28, 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 [28, p. 408f].

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 ambiguity 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 [28, 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], [26, p. 4], [31, 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 [26, p. 5]. Due to the properties of the MSA, micro-services may be deployed, upgraded and scaled individually [31, 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 [25, p. 35]. As services may be unavailable at times, a mechanism has to be in place that allows the discovery of services (such as the MOM) [25, p. 35]. Unless a dedicated logging service is introduced and used, there is no central access to the services' logs. Aggregation and analysis of errors and fixing them 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 poses a challenge that is not needed to this extent in monolithic architectures or SOA.

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 limit the effect of component fail-

ures, the whole application is rendered inoperative if the underlying machine fails or the process dies [26, p. 55]. SOA and MSA both strongly advocate to account for the possibility of a non-responding service – in the first place because of the limited reliability of network communication, but the outcome applies to any other reason for failure, 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 only 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.

SOA and MSA both permit scaling individual parts of an application by duplicating services. 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.

In regards of the above considerations, monolithic architecture is ruled out as the favored application structure. Thus, only SOA and MSA are left for the decision regarding the WfMS' architecture of choice.

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 further 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 MSA is the architecture of choice for the prototype with regards to the chosen objectives.

4.3.3 User interaction with the system

In a monolithic architecture, the use of a single user interface that provides access to the whole functionality of the WfMS would appear to be the obvious choice. In contrast to that, the modular structure of a MSA with clearly defined borders intuitively promotes separate user interfaces for different functionalities. In a MSA, either each service offers its own user interface

or there is a component at some point between the user and the WfMS that consolidates the interaction. As this architectural style is chosen in 4.3.2, the advantages and disadvantages of both options are briefly discussed in the following.

Separate user interfaces increase the flexibility regarding changes in both frontend and backend of a service. While the adaption of a consolidating component to a change within one service would likely require its redeployment and thus affect the availability of other services, this is not the case if a dedicated user interface is in place, which can be restarted without affecting other services.

One disadvantage of separate access to the different services is, that the user needs to know the location of each service in order to address it. Another drawback is that in this setting, tasks such as authentication, load balancing etc. would have to be performed for each service separately. - no of requests - bundle logic in one place, danger of monolith-like heap of logic - with micro services - two options - separate user interfaces per service - + more flexibility - + no single point of failure - - client somehow has to know the services - - auth / load balancing / ... duplicated per service - - multiple clients: need to be updated on service changes - 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 -> can be duplicated behind Load balancer - - has to be maintained API Gateway in this case features message endpoint pattern [27, pp. 95-97] Web-GUI vs application GUI vs CLI

4.3.4 Inter-component communication

Since all services reside in separate containers, i.e. are isolated through namespaces for processes, networks etc, a means must be found to let them communicate with each other.

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 contact the host machine IP address on 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 an instance of the same database application or a service is started several times for scaling. Second, this exposes the services in question to requests from any computer that can communicate with the host machine. Unless this is desired behavior, it creates an unnecessary attack surface.

** TODO: add citations ** 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 addressed to the recipient container. This happens at two places: 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, they have disadvantages, too. 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* environment variables that Docker created within a container are passed to any container that links to it, which could pose problems regarding security if they contain sensitive data such as passwords.

To cope with these disadvantages, the ambassador pattern was introduced [18]. 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 that point at each other, a multi-host setup can be created. 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. In a two-host setup, this number would grow to twenty additional containers.

Another solution is based on the Docker networking feature set, which was introduced in the end of 2015. This solution utilizes overlay networks, which were briefly presented in 3.1.7. During the development of this feature set, the *Container Network Model* was added, which allows containers to become member of multiple networks [32]. 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.

** three networks !!! 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 (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.

With Docker networking, each service can simply be addressed by its container's name and its corresponding port. 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.

As the last approach offers the required capabilities and, at the time of this writing, had no known drawbacks, it should be favored over the other presented ones.

4.4 System design

While high-level decisions are made in Section 4.3, this section is concerned with the more detailed view on the system design. First, the structure and desired behavior of workflow images and activity images is determined in 4.4.1. Then, the mode of communication between the system components is chosen in 4.4.2. Finally, the system's components are identified and designed in 4.4.3.

4.4.1 Workflow and activity images

To reap the benefits of the layer mechanism, the structure of the images should be chosen with care. Layers should be created in a way that enables reusability among the different use cases and they should be ordered by the frequency that they are likely to be changed by.

The proposed structure of workflow and activity images, which is depicted in Figure 4.4 and reflected in the respective Dockerfiles 4 and 5, is thus as follows.

The proposed structure consists on three images which should build on each other consecutively, as they are meant to be increasingly specialized. The first image should provide the

runtime environment. This image could be provided by a third-party vendor that specializes in building such images, i.e. an OS community or framework developers. Based on this image, a generic activity image `ac_base` (and, for $*^*_{SEPC}$, a generic workflow image `wf_base`) should be created. Such an image can be extended with element-specific information for each element of a workflow when that workflow is exported for deployment, to obtain the uppermost images `ac_$activity_id` (and `wf_$workflow_id`). An instance of this last image would then be a container with a suitable name of the form `aci_$activity_instance_id` (and respectively `wfi_$workflow_instance_id`).

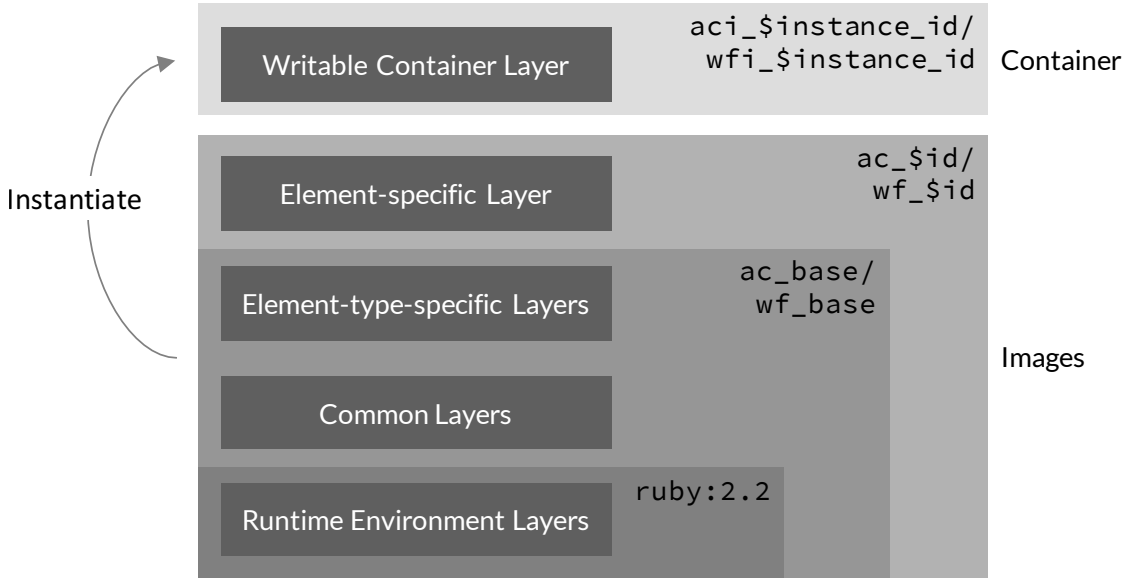


Fig. 4.4: Layer Structure for Activity/Workflow Images

Regarded in a more detail, the images' structure should look as follows. The foundation should be formed by *runtime environment layers*, as they are expected to change seldom and are required by all derived images. Usually, these layers contain an OS, common libraries and utility programs.

The layers that form the `*_base` images can be separated in two groups, *common layers* and *element-type-specific layers*. The *element type* refers to either activity or workflow. The common layers should be created on top of these runtime environment layers. They are intended to contain the effects of invoked commands, added directory structures and files which are required by both activity images and workflow images. Even though these layers are not explicitly named, they will be stored by Docker in its cache and used during the build process of similar images. In the next step, element-type-specific layers should be added. These layers are meant to contain data that is required for the execution of an activity *or* a workflow, for example scripts which perform validation tasks (if not provided by a service) or general-purpose data transformation.

The element-specific layer, which is added in the course of the export of an element (activity

or workflow), contains files that are particular to single activities or workflows. In an activity image, this layer would contain the activity configuration and the schemas for data validation. In a workflow images, for example, it would contain the process definition.

By instantiating the resulting image a container is created at runtime, which owns the upper-most, writable layer. This is where the activity or workflow may store data during execution.

At the time of writing, Docker registries do not reuse layers across repository borders during uploads yet – even though it is a proposed feature [33]. In order to benefit from the layering in the previously described way, it is thus necessary to let all activity images reside in the same repository by tagging them in the format

```
$repository_url/activity
```

and using the respective activity's ID as a version tag to differentiate between them. They can then be referred to as

```
$repository_url/activity:ac_$activity_id
```

Analogously, this is done with workflow images. Since it implies losing the internal image versioning mechanism, this solution should only be used as a workaround until cross-repository sharing of layers is possible.

supported functions: - activity images - start subworkflow - start third-party container - initiate user input - workflow images - ggf wf engine - manage workdirectory (DV approaches)

4.4.2 Communication

While the considerations in 4.3.4 were targeted at finding a model for the low-level communication, a way how the services communicate with each other

- message queue between services - jeweilige protokolle via docker networks network between services publish/subscribe

4.4.3 Components

As noted in 4.3.1, one of the downsides of MSA is that it is crucial to determine suitable service boundaries. Some sources advise to first build a monolithic application and then analyze the result to single out services that can be extracted ** sources **. 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 identified.

As presented in 2.2.2, the WfMC identified the following components [**?** , p. 13]:

- 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

** monitoring and analysis?

** draw communication diagram here **

According to the WfMC's description, the definition tool, the worklists handler and the organization modeling tool each utilize one data component respectively. Combined with its respective datastore, each of them can be considered an autonomous micro-service since each would be able to provide its functionality without any further service.

The workflow control data component can be considered to be part of a workflow engine service. Depending on the chosen mode for the enactment, workflow relevant data is either managed by the workflow engine service, too, or accounted for by a data volume (${}^{DV}_{*}$). Only in the ${}^{SER}_{*}$ variant, a dedicated service for its management and storage is needed.

According to the decision to use the API gateway pattern (4.3.3) to hide the internal system structure from its users, the two contact points – one for administrative work and one for end-user work – are realized using an appropriate gateway. The *developer gateway* enables requests to the definition service, the infrastructure service and the organization management service through a GUI. The *user gateway* emits requests to the worklists service which are also issued through a GUI.

** TODO: add deductions from objectives ** In addition to the services derived above, the need for some additional services originates from considerations regarding the use of Docker for workflow execution in Section 4.2 and the objectives that were stated in 4.1. A Docker image registry was suggested to be used for the distribution of images to all nodes. Unless an external

service such as Docker Hub is used, an own registry service should be part of the WfMS. The decision made in 4.4.2 to use MOM for the communication between services creates the need for a service which acts as such middleware. To meet the requirement of automatically distributed images, a provisioning service should be introduced, which performs the appropriate actions. An infrastructure management service could be used to monitor the status and properties of the available nodes in the swarm.

Besides the major components, independent functionalities which are frequently used could be singled out to separate services. One micro-service that might be extracted could address the validation of input and output data. Given a dataset and a set of rules on how to validate this dataset, such a service would be 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 is beneficial to be able to scale the execution of this task independently. **out of scope!**

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 its functional scope, the workflow definition service is also the service that handles 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. The transformations of workflows is performed by the `ImageBuilder` class, which relies on the `ProcessDefinitionImageSerializer` for the serialization of the process definition. The logic required for publishing the images is defined in the `ImageManager` class.

As depicted in Figure 4.5, the service has the model classes `Workflow`, `Activity`, `ProcessDefinition`, and `ControlFlow`, which provide the object-relational mapping for the respective objects. The roles assigned to activities have only to be dealt with in the form of unique identifiers, relying on the assumption that components which have to use them may resolve these identiefiers themselves.

As a user interface is provided by the developer gateway service; the workflow definition service thus does not offer its own user interface, but rather exposes its functionality via the MOM. This allows workflow definitions to be created and altered by other services, e.g. an conversion service which translates other process definition formats or some feedback mechanism that alters workflows based on their execution performance – or a gateway service that provides a user interface.

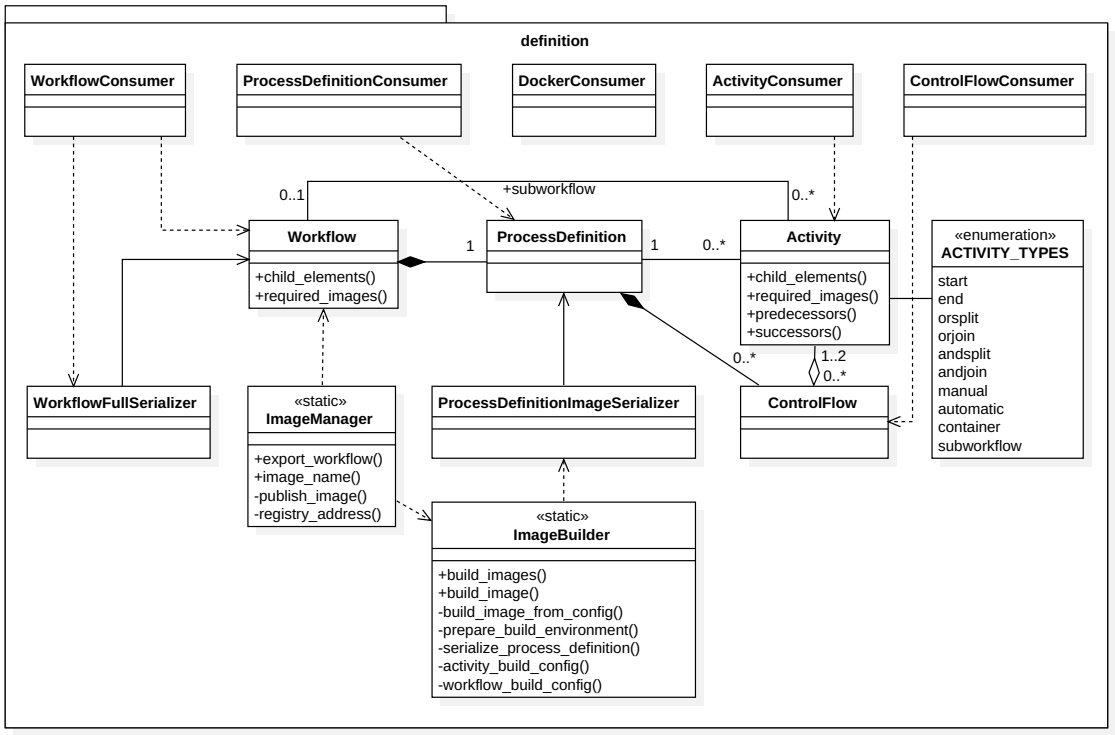


Fig. 4.5: UML Class Diagram for the Workflow Definition Service

** remove methods ** remove automatic activity

In order react to requests of other services, the workflow definition service features consumer classes, which perform the required actions and publish a response, if required. `WorkflowConsumer`, `ActivityConsumer`, `ProcessDefinitionConsumer`, and `ControlFlowConsumer` response to Create, Read, Update, Delete (CRUD) and index requests. The `WorkflowConsumer` additionally provides the means to react to requests for the export of a workflow.

There exist two classes in this service which deal with the serialization of objects. The serialization of a workflow with its components nested inside takes place in the `WorkflowFullSerializer`. Such a serialized version is required to avoid separate requests when the workflow is requested for modeling. The `ProcessDefinitionImageSerializer` is used to generate a serialized version of a process definition which can then be incorporated in workflow images.

Since one of the previously determined requirements for the prototype is that developers should be supported to use third-party images, the workflow definition service further reacts to relev-

ant requests in the `DockerConsumer` by initiating a search for images with a specified name on Docker Hub.

In order to be able to communicate with the MOM, the workflow definition service is connected to the `wmfs_backend` network.

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. The service may be queried for users or roles, or to authenticate users for the use of the WfMS.

As depicted in Figure 4.6, the service consists of the classes `User`, `Rolem` `UserConsumer`, and `RoleConsumer`. The model classes `User` and `Role` provide the object-relational mapping for the database, while the `UserConsumer` and `RoleConsumer` classes enable the service to react to CRUD and index requests that concern these objects.

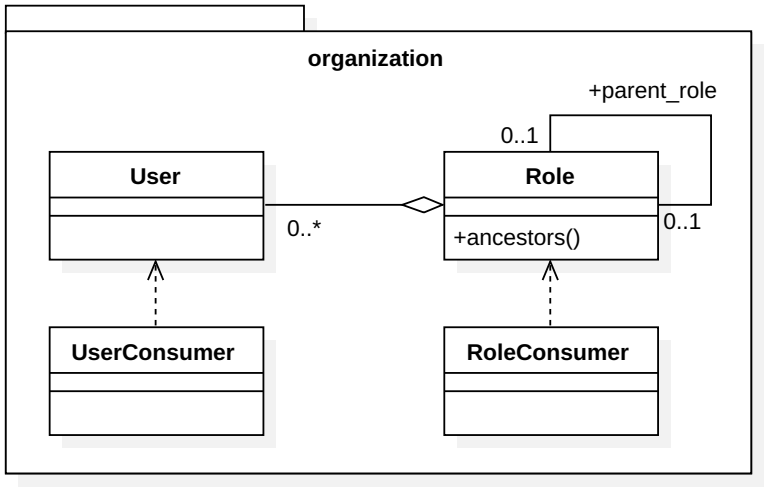


Fig. 4.6: UML Class Diagram for the Organization Management Service

Like the workflow definition service, the organization management service is connected to the `wmfs_backend` network to be able to communicate with the MOM.

Worklist Service

The sole responsibility of this service is the management of users' worklists. It handles CRUD requests for worklist items and publishes the data submitted to it by users to the other services. If a user is deleted, it should remove the worklist item or reassign it to another user. The former tasks are performed by the `WorklistConsumer`, which reacts to related events. The latter

task is in the responsibility of the `UserConsumer`, which reacts to the deletion of a user. The worklist items' object-relational mapping is performed by the `WorklistItem` model class.

Workflow Engine Service

** TODO: incomplete ** In wide parts, the workflow engine service is congruent to the *workflow engine* component identified by the WfMC in terms of functionality, which is described in **??**. The way to utilize Docker for the workflow enactment chosen in 4.2 has an impact on the range of functionalities that this service has, though. ** like, how

- choose participants - add to execution networks

WorkflowConsumer WorkflowInstanceConsumer ServerConsumer WorkflowInstance WorkflowScheduler

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

Developer Gateway

Since it only offers a unified access to the WfMS and does not store any data itself, the developer gateway does not require any database. The service is realized in a two-tier architecture, with a backend part that handles requests to and responses from the various WfMS services, and a frontend part that presents the received data to the developers and accepts their input.

The only task of the backend is forwarding the user's requests to the message queue and the corresponding responses back to the user. The controller classes that are responsible for this (`ActivitesController`, `ControlFlowsController`, `DockerController`, `ProcessDefinitionsController`, `RolesController`, `ServersController`, `UsersController`, `WorkflowsController`) are thus very lean – they only contain the logic to forward requests to suitable routing keys. Each of them inherits from the `ApplicationController`, which manages the messaging logic and gives access to a shared single connection to the MOM.

The `TemplatesController` is different from the other controller classes, as is not involved in forwarding requests, but serves the purpose to render and deliver the HyperText Markup Language (HTML) fragments required by the frontend.

Following the API gateway pattern that is chosen in 4.3.3, this service and the user gateway

service are the only services that can be reached from outside of the WfMS. ** API GATEWAY DEFINED?

User Gateway

Analogous to the developer gateway, the user gateway provides access to those WfMS services that are relevant to its users, that is, in the chosen setup, only the worklist management service.

Due to the few responsibilities of this gateway, there are only two controller classes in this service: `WorklistItemsController`, which forwards CRUD requests that concern worklist items and `WorklistController`, which provides the means to access all existing worklists.

Infrastructure Management Service

The infrastructure management service fetches and refines the information related to the swarms nodes. That is, it lists all nodes and can inform on their properties, (running) containers and available images. Supported by the `DockerHelper`, which provides the connections to the different nodes, the `EnvironmentManager` contains the required logic to fulfill these tasks. The `ServerConsumer` waits for relevant requests via the MOM, instructs the `EnvironmentManager` accordingly, and returns the results. Further, there is a `Server` model class, which is used to structure the obtained information.

Additionally to its role in the information retrieval, the `EnvironmentManager` subscribes to events of new nodes joining the swarm and launches the provisioning service on joining nodes.

Registry

All solutions presented in Section 4.2 feature custom Docker images, be it workers or containerized activities or workflows. These images can contain information on business processes and other information whose disclosure should be avoided. In order to store and distribute such 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 stated in 4.1 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. To do so, the `ImageConsumer` and `ServerConsumer` classes react to relevant events by invoking the appropriate Docker commands.

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. The former variant enables all nodes to react concurrently to the event of a published image, while in the latter variant, the distribution would take place sequentially – unless the service is implemented in a multi-threaded or multi-process way. While the idea of provisioning all nodes at the same time might be appealing, the workload imposed on the registry node by a big swarm should be considered.

5 Prototypical implementation

Based on the considerations in the previos chapter, a Docker-based WfMS prototype was implemented.

** etc**

Since the principle of event-driven service invocation is already demonstrated in the definition service, a detailed documentation of the organization and worklist services would not contribute further relevant insights. They are thus only described regarding their realization with Docker containers and the network configuration of these containers.

5.1 Preliminary decisions

All of the variants highlighted in 4.2.2 represent possible approaches for the utilization of Docker for the enactment of workflows for different use cases. G_{DV}^{SEPC} with a partially integrated workflow engine is the implemented variant for several reasons. First and foremost, this combination enables the use of basic Docker commands for the suspension and continuation of a workflow and single activities, as described in 4.2.1. Second, $*_{*}^{SEPC}$ leverages the already existing communication between the local daemon of a node and the swarm master daemon to publish the status of both workflow instances and activity instances, in the form of events concerning container statuses. Third, G_{DV}^{*} is a promising variant for the demonstration of various scheduling mechanisms introduced in 4.2.4, since this variant includes implicit affinities by default. While partially integrating the workflow engine is mainly interesting for its role in the previously mentioned native pausing, it also facilitates the working directory management, as argued in 4.2.1.

The services identified in 4.4.3 are implemented using *Ruby*, a dynamically typed object-oriented programming language. Ruby provides the means to write concise, well readable code [34, p. 782]. Performance-wise, it is inferior to many other languages [34, p. 786]. However, the focus of this thesis is to explore conceptual possibilities rather than developing efficient implementations. An emphasis is thus put on the expressiveness of the used language to help the reader to grasp the underlying concept quickly.

All services that require access to any Docker API use a gem called `docker-api` which provides a client for these APIs. *Gem* is the name for distributable packages in the Ruby ecosystem. These packages can be managed using *Gemfiles* in which the package dependencies of an application may be declared.

The Ruby-based web application framework Ruby on Rails (RoR) is used for the implementation of the developer gateway and user gateway. It qualifies for this task because it comes with several functionalities that aim at the fast creation of prototypes. For example, so called *scaffolds* permit the creation of model and controller classes and appropriate views based on a specified database schema; the included library *ActiveRecord* supports adding object-relational mapping functionality to model classes [35, p. 5]. ActiveRecord is also used in those services that do not use Rails but need to store objects in databases, namely in the `definition`, `organization`, and `worklist` services.

PostgreSQL was chosen as database solution for the services, because it supports both relational data, which is useful for storing the models and their relations, as well as the document-store-like JSONB format which allows the schema-less storage of configuration information.

RabbitMQ was chosen as message oriented middleware, because it is well documented and provides a web interface for monitoring and administration. The gems `Hutch` and `Bunny` provide different levels of abstraction for the interaction with the RabbitMQ message queue [**?**]. `Hutch` itself is based on `Bunny` and imposes some opinionated conventions for the use of the message queue, as well as automatic (de-)serialization of the messages' payload.

5.2 Execution images

In 4.4.1, activity images and workflow images are designed. In the following, the implementation of these images is described.

5.2.1 Workflow image

The workflow image is implemented as designed in 4.4.1. The runtime-environment layers are provided by inheriting all layers of the `ruby:2.2` image. On top of these, common layers are created, in which the required gems are installed. Then, the `/workflow` directory is created, into which the required Ruby files are copied. The file `run.rb` is set as the default command.

The workflow instance is structured as depicted in ** **??**. It contains the classes `ProcessInstance` and `ActivityInstance` which coordinate the enactment of the workflow instance, and the utility classes `ProcessDefinition`, `FileHelper`, `Configuration`, and `Validator`, which provide auxiliary functionality. `ProcessDefinition` parses the process definition file `process_definition.json` and creates objects for the process definition itself and its contained activities to facilitate accessing their respective properties. `Configuration` wraps the access to relevant environment variables in methods, which may provide defaults for

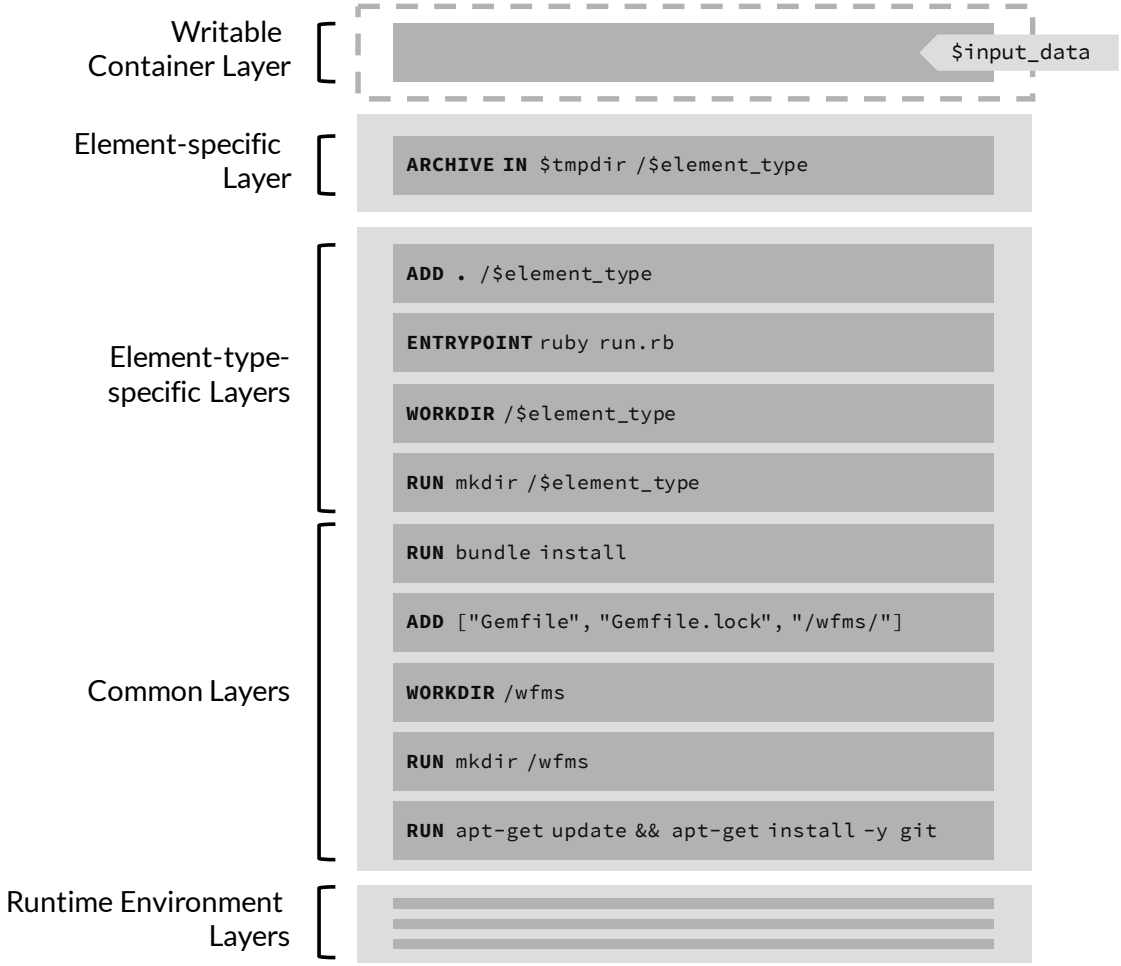


Fig. 5.1: Layer Contents for Element-wrapping Containers

missing values. Everything that is related to file system access, e.g. constructing paths, linking and creating directories etc, is performed by the `FileHelper` class. The `Validator`, is used to validate the workflow instance's input data against the provided JavaScript Object Notation (JSON) schema.

The code and the workflow configuration files reside in the `/workflow` directory. `process_definition.json` contains the exported process definition, `workflow.info.json` contains meta data on the workflow, and `input.schema.json` contains the JSON schema used to validate the workflow instance's input. The instance specific files are copied to `/workflow` before the container is started.

As decided in 5.1, the workflow instance features some functionality that one would usually find in a workflow engine. This functionality resides in the `ProcessInstance` class. On instantiation, the `ProcessInstance` obtains an instance of the helper classes `ProcessDefinition` and `Validator` and instructs the latter to check the validity of the provided input data.

The `run` script loads the required dependencies and ensures, that the container is connected to the `enactment` network. Then, it initiates the enactment by creating an instance of the `ProcessInstance` class and calling its start method. When started, `ProcessInstance` creates a new `ActivityInstance` for the start activity, creates a symbolic link to the workflow instance's input data, and adds the `ActivityInstance` to a queue for incomplete activity instances. Then, it enters a loop in which it carries out the workflow enactment. It leaves this loop which calls the method depicted in Figure 5.2 on each iteration, if no more activities are queued for processing.

```
26  def process_queued_activity_instances
27    uncompleted_instances.each do |instance|
28      next unless instance.required_predecessors_completed?
29
30      instance.run
31
32      instance.activity.successors.each do |successor|
33        successor_instance = find_or_create_activity_instance(successor)
34        successor_instance.completed_predecessors << instance
35        Workflow::FileHelper.link_instance_output_to_successor_input(instance,
    ↪  successor_instance)
36      end
37
38      if instance.activity.type == 'end'
39        @queue.clear
40        Workflow::FileHelper.link_instance_output_to_workflow_output(instance)
41      end
42    end
43  end
```

Fig. 5.2: The processing loop of `ProcessInstance`

The called method, `process_queued_activity_instances`, iterates over the queue of unprocessed activity instances. If the start conditions of such an instance are not met it is skipped for the current iteration. Otherwise, the activity instance is started. As soon as it finishes, an `ActivityInstance` object is created for each of its succeeding activities. The finished activity instance is added to the successors' lists of completed predecessors, and a symbolic link from its output directory to the respective successor's input directory is created. If the activity was an end activity, a symbolic link is created from its output directory to the workflow's output directory and all pending activity instances are removed from the queue. The loop ends and the workflow instance terminates together with the process instance.

The actual creation and start of an activity instance container takes place in the `ActivityInstance` class, which invokes Docker with the command depicted in Figure 5.3. The container is named with an instance ID, labeled with this very ID and also with the IDs of the directly superordinate workflow instance as well as the overall superordinate workflow instance. The container is based on the suitable activity image and launched without an additional command, as the right command is already specified in the image. To enable the instance container to communicate with the local Docker host, the respective socket is mounted on the container's file system. The data volume that contains the workflow relevant data is bound to the activity instance container, too. Environment variables are set on the container in order to pass along configuration: `MAIN_WORKFLOW_ID`, `WORKFLOW_ID`, `WORKFLOW_INSTANCE_ID`, `ACTIVITY_ID`, and `ACTIVITY_INSTANCE_ID` are passed to inform the activity instance container about its activity/activity instance ID a, as well as the directly superordinate workflow/workflow instance, and finally the ID/instance ID of the originally called workflow. It is further passed the path of the directory that is designated to be its working directory in the `WORKDIR` environment variable.

5.2.2 Activity image

Analogous to the workflow image, the activity image is implemented as it is designed in 4.4.1. Instead of a `/workflow` directory, the `/activity` directory is created which contains the necessary code for the activity instance. The file `run.rb` is set as the default command.

The activity image contains the classes `ActivityInstance`, `WorklistClient`, `ContainerInvocation`, `SubworkflowInvocation`, `FileHelper`, `Configuration`, and `Validator`, as depicted in **??**. `FileHelper`, `Configuration`, and `Validator` have the same functionalities as their counterparts in the workflow image.

The code and the activity configuration files reside in the `/activity` directory. These files are `activity.info.json` and `input.schema.json`. `activity.info.json` contains con-

```
40    def container
41      config = Workflow::Configuration
42
43      Docker::Container.create({
44        'name' => "aci_#{@id}",
45        'Labels' => {
46          "main_workflow_instance" => "#{config.main_workflow_instance_id}",
47          "workflow_instance" => "#{config.workflow_instance_id}",
48          "activity_instance" => "#{@id}",
49        },
50        'Image' => "#{config.image_registry}/activity:ac_#{@activity.id}",
51        'Cmd' => [''],
52        'WorkingDir' => '/activity',
53        'Tty' => true,
54        'Env' => [
55          "MAIN_WORKFLOW_ID=#{config.main_workflow_id}",
56          "MAIN_WORKFLOW_INSTANCE_ID=#{config.main_workflow_instance_id}",
57          "WORKFLOW_ID=#{config.workflow_id}",
58          "WORKFLOW_INSTANCE_ID=#{config.workflow_instance_id}",
59          "ACTIVITY_ID=#{@activity.id}",
60          "ACTIVITY_INSTANCE_ID=#{@id}",
61          "WORKDIR=#{Workflow::FileHelper.activity_instance_workdir(self)}",
62          "DATA_CONTAINER=#{config.workflow_relevant_data_container}"
63        ],
64        'HostConfig' => {
65          'Binds' => ['/var/run/docker.sock:/var/run/docker.sock'],
66          'VolumesFrom' => [config.workflow_relevant_data_container],
67        }
68      })
```

Fig. 5.3: Instantiation of an activity image in `ActivityInstance`

figuration data of the activity, e.g. name, version and parameters for a third-party container for a container-type image, or the assigned role for a manual-type image. `input.schema.json` contains the JSON schema that is used to validate the activity instance's input.

Depending on the activity type, the activity instance either starts a specified third-party container (container activity), a workflow instance container (sub-workflow activity), or issues a worklist item for manual data input (manual activity), by using the `ContainerInvocation`, `SubworkflowInvocation`, or `WorklistClient` classes respectively. The outcome of these actions is then stored in the output data file. For this prototype all other activities, i.e. the control flow activities, log the activity and activity instance IDs to that data file as a proof of their invocation.

** other code?

```
18    def start
19      case @activity_info['type']
20      when 'manual'       then start_user_input
21      when 'container'    then start_container
22      when 'subworkflow'  then start_subworkflow
23      else log_self_to_data
24      end
25
26      write_output
27    end
```

Fig. 5.4: Instantiation of an activity image in `ActivityInstance`

The invocation of another container is performed in the `ContainerInvocation` class. The container created based on the specified image and is labeled with the IDs of the overall superordinate workflow instance, the directly superordinate workflow instance, and the activity instance. Once the container's execution has finished, its standard output stream and error output stream can be accessed with the `results` method. `SubworkflowInvocation` acts similar to `ContainerInvocation`, but it uses another configuration for the container. ** lots of difference in configuration here **

5.3 System components

In the following, the implementation of the prototype's system components according to the design from 4.4.3 is described.

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.

The PostgreSQL database's support for relational data is used for storing the workflow, its elements and their relations. Its capability to store data in the JSONB format is used to store the schema-less storage of configuration information. This is valuable because the structure of those configurations is not known in advance, e.g. input validation schemas. 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 to build and push images.

As planned in 4.4.3, the service has the model classes `Activity`, `ControlFlow`, `ProcessDefinition` and `Workflow`, which act as object-relational mappers for persistence to the database as well as ensure some validity constraints. Further, there exists a controller class for each of the aforementioned models, which provides 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 contained in the model and controller classes and 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 information that is necessary to execute the corresponding workflow. An example for the serialized output can be seen in Figure 3.

Whenever 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 of the workflow's activities has to be exported. 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 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 schemas, 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 and worklist service

As visioned in 4.4.3, the organization management service and the the worklist service are structured analogously to the workflow definition service. Just like it, the services consist of three Docker containers: an application that is backed by a PostgreSQL database that is persisted to a data volume.

5.3.3 Workflow engine service

As it does not require any data persistence in the chosen setup, the workflow engine service only consists of an application container. The consumer classes `WorkflowConsumer` and `WorkflowInstanceConsumer` listen to events that concern the corresponding element types and instruct the `WorkflowEngine` to perform the according action. The `DockerHelper` class provides the connection to the swarm master that is used to manage the workflow instances. The `WorkflowInstance` class takes care of the actual instantiation of a workflow and the required preparations that come along with it.

The `WorkflowEngine` pauses, unpauses or terminates a workflow instance, by querying the swarm master for all containers which bear a label with the workflow instance's ID, and then calling the `pause`, `unpause`, or both `kill` and `delete`.

`WorkflowInstance` begins with the enactment of a workflow by creating the data container, which is used later to store the workflow data. The data container is based on the image `cogniteev/echo`, a small image that provides a single executable – `echo` – because Docker expects one executable to be present in a container [**?**]. The container's name is configured to equal

the workflow instance's ID with the prefix `data_` and the ID is also passed to the container as value for a `main_workflow_instance` label. By passing the constraint

```
"constraint:node==#{@target_node}"
```

the data container is scheduled on the node with the specified name. Each node on which workflows are executed obtains a directory `/workflow_relevant_data`, in which the working directories for the various workflow instances are kept. `WorkflowInstance` passes the instruction

```
"/workflow_relevant_data/#{@instance_id}:/workflow_relevant_data"
```

to create a such a working directory under the name of the workflow instance ID and to make it available in the data container at the path `/workflow_relevant_data`.

Then, `WorkflowInstance` resolves the appropriate image name from the workflow's ID and creates the workflow instance container – but it does not start it yet. The container's name is configured to equal the workflow instance's ID with the prefix `wfi_`. The container is labeled with the workflow instance's ID as value for both `main_workflow_instance` and `main_workflow_insta` `WorkflowInstance` passes the instructions to mount both the local Docker daemon's socket and the data volume of the previously created data container. The passed affinity

```
"affinity:container==#{@data_container.id}"
```

instructs the swarm master to create the container on the same node as the data container. Since the workflow instance is the root of the enactment's instance hierarchy, its workflow and workflow instance IDs are passed to the container as the values of the environment variables that inform the workflow instance about the superordinate workflow and its instance as well as its own and its workflow's ID.

When instructed to start the workflow instance container, `WorkflowInstance` connects the container to the `wfms_enactment` network, copies the input data into the container, starts it and waits for it to stop. Then, it copies out the output JSON file, parses and returns its content as an Ruby object to the engine. In a non-prototypical implementation, the containers would be deleted at this point – in this prototype, they are left on the node for inspection.

5.3.4 Developer gateway

The developer gateway does not require any database, as it merely forwards requests to the services via the MOM. Hence, it consists of a single Docker container that contains a RoR

application. To be reachable by the users, this container needs to expose the port that the RoR application is listening on to the host's network.

Frontend

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

5.3.5 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. To make the service accessible from outside of the system, the user gateway service's container exposes the port that the application is listening on the host's network. The user gateway is connected to the frontend network to isolate it from the internal services – to which it may only communicate through the MOM. While the backend functionality is implemented in Ruby, the frontend is served by RoR as simple HTML pages.

5.3.6 Message oriented middleware (MOM)

RabbitMQ exists as a pre-configured Docker image (`rabbitmq`) on the Docker Hub and can thus put to use swiftly. The configuration of RabbitMQ in this image takes place when the respective container is started which allows its configuration in the `docker-compose` configuration file as depicted in Figure 5.5. For the sake of simplicity, no authentication mechanism was introduced besides the simple default username/password combination. As the central point of communication, the MOM is the only container which is connected to all three overlay networks. While one would probably avoid exposing this service in a real use case, it is exposed to the host's network for its use in a prototype, as this allows to monitor the messaging activity of the services.

5.3.7 Infrastructure management service

The data this service offers – the state of the Docker swarm and its nodes – should always 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.

```
    image: rabbitmq:3-management
    restart: on-failure:3
    networks:
      - backend
      - enactment
      - frontend
    ports:
      - "8080:15672"
    environment:
      - "constraint:node==internal-machine"

```

Fig. 5.5: Configuration of the MOM service in the Docker Compose file

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.

5.3.8 Registry

A local registry is deployed in its own container in order to distribute the created images. Docker provides the `registry` image for this purpose, of which the version `registry:2.3` is used, since it is the newest stable version at the time of this writing.

The registry is configured to be restarted automatically in case that itself or the node it runs on fails. Further, it is exposed on the host's network to make it reachable for all nodes in the swarm. This is necessary as their daemons are not containers and thus cannot be connected to the overlay network. This configuration is reflected in the WfMS' Docker Compose file as follows:

```
  registry:
    image: registry:2.3
    restart: always
    ports:
      - 5000:5000
    networks:
      - backend
    environment:
      - "constraint:node==coordination-machine"

```

1: Configuration of the registry service in the Docker Compose file

5.3.9 Provisioning service

The provisioning service is implemented in a simple way – it consists of a single script that listens to the events published to the swarm master. If a named image is pushed by one of the nodes, the provisioner pulls its own node, as visible in Listing 2.

```
19 begin
20   Docker::Event.stream do |event|
21     if event.status == 'push' && !event.id.match(/sha\:/)
22       puts "pulling #{event.id}"
23       Docker::Image.create({fromImage: event.id}, local_connection)
24     end
25   end
26 rescue Exception => e
27   puts e
28   retry
29 end
```

2: Provisioning service pulling new images

5.4 Exemplary deployment

To show the feasibility of the developed solution, the prototype is deployed exemplarily. This process is supported by Docker Machine, which provides the means to create virtual machines which are provisioned with a Docker Engine and may be connected to a swarm at startup. Four machines are created to simulate an organization setup. The first machine that is started will serve as host for the image registry and the key-value store that Docker Swarm uses. It is thus named `coordination-machine`. To account for the increased need for memory of the registry, this machine receives twice the amount of memory that the other machines will be granted. Three other machines are created, which are equal in their available resources: `development-machine`, `internal-machine`, and `cloud-machine`. The latter is not actually located on a remote location, its daemon is labeled with

`edu.proto.machine_env=external`

while the other nodes receive the value `internal`, to show how such a node could be distinguished for scheduling. Each of the nodes is also labeled with its respective amount of memory, e.g. for latter three machines:

`edu.proto.ram=1024`

While the allocation of services to specific nodes would not matter in practice, it is done in a static manner for this prototype to facilitate the examination and documentation. The services are assigned to the nodes as follows: the registry is started on `coordination-machine`; the MOM, worklist, and user gateway services reside on `internal-machine`; all other services are allocated on `development-machine`. Creating the registry on the `coordination-`

`machine` is a good practice, as every Docker daemon that is part of a swarm knows the IP address of this machine if the key-value store is also hosted there. Its address can thus be resolved by accessing the daemon's configuration. Further, it may be used to distribute the WfMS' images during the installation of the system.

To recover from service failures, all containers are configured with the option `restart: on-failure:3`. That is, in case that a container terminates through an error, Docker tries to restart it up to three times.

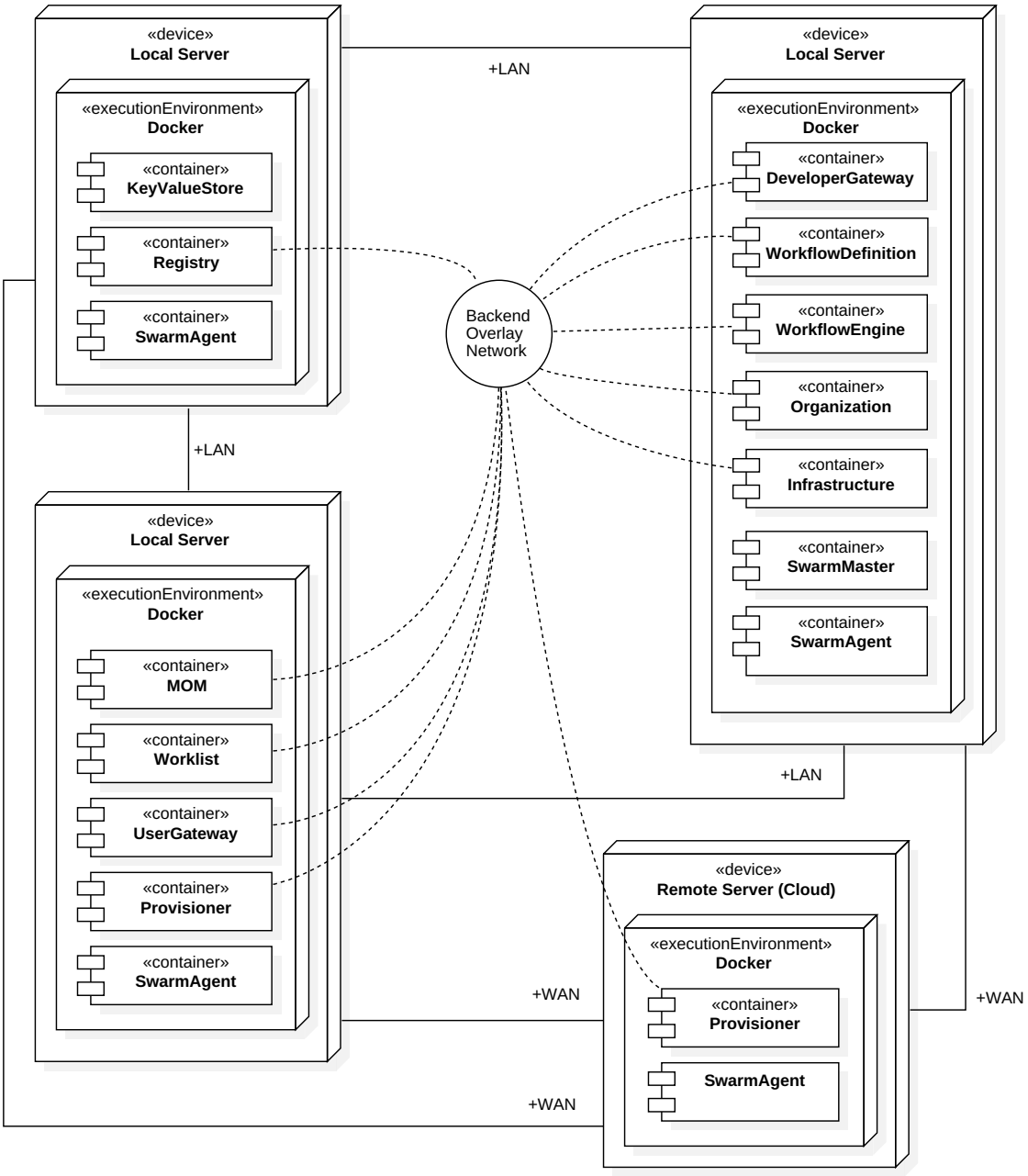
The deployment of all services is performed using Docker Compose, as this tool provides some conveniences for deployment-related tasks. First, Docker Compose enables the automatic building, starting, stopping and removal of the services' images, as described in 3.2.3. ** describe it there ** Second, the configuration of all services and the required networks that connect them may be specified in a single *YAML* file. Yet Another Markup Language (YAML) is designed with a focus on human readability [36]. The configuration file, which is depicted in Figures 6-10 may thus serve as a textual documentation of the system's architecture. For comparison, a graphical visualization of the deployment is presented in Figure 5.6.

5.5 Implementation issues and compromises

The requests that are sent from the frontends to the gateways are implemented in a synchronous fashion, i.e., the frontend sends the requests and blocks the execution until a response is received. This is also imposed on the onward communication to the other services in this prototype by the way the gateways' consumers are implemented – a blocking queue listens to the response to a specific request. In cases where multiple independent requests are made, an asynchronous implementation would enable performance gains, as these requests could be performed in parallel. However, this would raise the complexity of both frontend and gateway, as they have to account for the unknown order of incoming responses.

While a worklist item should be rescheduled in a real-world scenario if its assigned user is deleted, it is simply deleted by the `UserConsumer` in the prototype as this is a less complex way to handle this case.

Since Ruby code is interpreted rather than compiled to executables and run, it requires the runtime environment to be served as part of the images, which are thus considerably larger as images that container only contain a self-sufficient executable. Due to the previously presented benefits that come along with the layer structure, this is only relevant for image transfers for the first time that an image is up- or downloaded, though.



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. ** LAN WAN**

Fig. 5.6: Deployment Diagram of the Architecture

6 Evaluation and discussion

In Chapter 5, the prototypical implementation of a Docker-based WfMS is presented. This implementation is based on the considerations that are made in Chapter 4. In that chapter, objectives for the prototypical implementation were gathered, together with the requirements that must be met in order for these objectives to be considered fulfilled.

One of the objectives was that components of the WfMS should be alterable without a full system restart. In order to deploy a new version of a micro-service, that new version may simply be started in parallel to the old version – as long as the externally perceived behavior of the currently used functions of that service has not changed. This is made possible through the loose coupling of the micro-services via the message queue and the deployment of these services in separate containers. The two versions of the service will work on incoming requests in round robin, as they are subscribed to the same queue. If the new version of the service is deemed stable, the old version may be shut down.

Regarding the requirements that concern the failure resilience of a service, the prototype is able to let all parts of the system that do not rely on failed services continue to offer their functionality. If, for example, the organization service failed, it would still be possible to model and execute a workflow. Also, the possibility to instruct Docker to restart failed containers can help to keep the system available. In case of a micro-service failure, the unanswered requests to it remain in the queue and can be processed as soon as the service is available again. The prototype's ability to cope with failure can thus mainly be attributed to the combination of Docker with a MSA.

The management of nodes that are available for execution is mostly handled by Docker Swarm. By starting appropriately configured swarm agent containers on them, new nodes may be added to the swarm at any time. The infrastructure service notices the addition of new nodes and starts a provisioning service on them. This service in turn reacts to pushed images and instructs its respective node to pull them.

The prototype supports the user in using third-party images by providing the means to search for images on the public Docker Hub registry. Further, the invoked command can be specified for utilized third-party images. The graphical modeling environment abstracts from the fact that the container is started by an intermediate activity container.

Some of the objectives were addressed in theory only, but were not implemented in the prototype. As described in 4.2.4, nodes can be labeled and these labels can be used to enforce required properties of nodes for certain workflows or activities. ** for single entities?** The prototype applies this principle, but in a static way – not on a dynamic, per-element level.

Likewise, a solution for the priorization of activities and workflows was presented in **??** ** show it**, but it was not implemented in the prototype.

An objective that was disregarded in the implementation to keep the *what? thesis short?* is the management of permanently running services which are provided for specific workflows or activities. While it is theoretically described in **??**, there is no corresponding functionality in the prototype.

** monitoring/analysis left out

** - considering similar functionality /w subworkflow instantiation /validation / etc. - probably different concept of workflows better as suggested by [**?** , 119] - recursive structure - only one base image

7 Conclusion

In the thesis at hand, mechanisms and implementation issues that arise from the utilization of Docker for deployment and execution of WfMSs were proposed and discussed. Two general application areas for Docker were identified: on the one hand Docker may influence the architecture and design of WfMSs, on the other hand, it can provide new ways to distribute and execute workflows.

The

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

outlook: - pause + move containers: http://blog.circleci.com/checkpoint-and-restore-docker-container-with-criu/ - evaluate supported patterns? http://www.workflowpatterns.com/documentation/docu 06-22.pdf - implement resource management

Bibliography

[1] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi, "Specification and implementation of exceptions in workflow management systems," *ACM Transactions on Database Systems*, vol. 24, no. 3, pp. 405–451, 1999. [Online]. Available: http://portal.acm.org/citation.cfm?doid=328939.328996

[2] J. Becker, C. 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*, vol. Track5, 1999, pp. 10 pp.–.

[3] D. Hollingsworth, "Wfmc: Workflow reference model," Workflow Management Coalition, Specification, 1995. [Online]. Available: http://www.wfmc.org/standards/docs/tc003v11.pdf

[4] D. Wutke, D. Martin, and F. Leymann, "Model and infrastructure for decentralized workflow enactment," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 90–94. [Online]. Available: http://doi.acm.org/10.1145/1363686.1363712

[5] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst, "Workflow data patterns: Identification, representation and tool support," in *Conceptual Modeling - ER 2005*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, L. Delcambre, C. Kop, H. C. Mayr, J. Mylopoulos, and O. Pastor, Eds., vol. 3716. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 353–368. [Online]. Available: http://link.springer.com/10.1007/11568322_23

[6] P. Lawrence, Ed., *Workflow Handbook 1997*. New York, NY, USA: John Wiley & Sons, Inc., 1997.

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

[8] W. Felter, R. Ferreira, R. Rajamony, J. Rubio, W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, *An Updated Performance Comparison of Virtual Machines and Linux Containers*, 2014.

[9] 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)*, vol. 9523, pp. 813–824, 2015. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84951948191&partnerID=40&md5=a3ee2487761581896f3c54aa2ca00552

[10] O. C. Initiative, "Open containers initative." [Online]. Available: https://www.opencontainers.org

[11] I. Docker. The docker user guide. [Online]. Available: https://docs.docker.com/engine/userguide/

[12] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7158965

[13] I. Docker, "The docker user guide." [Online]. Available: https://docs.docker.com/engine/userguide/

[14] ——. Docker.com. [Online]. Available: http://docker.com

[15] ——. Docker orchestration product brief. [Online]. Available: https://www.docker.com/sites/default/files/products/PB_Orchestration_03.06.2015.pdf

[16] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *Cloud Computing, IEEE*, vol. 1, no. 3, pp. 81–84, 2014.

[17] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2723872.2723882

[18] I. Docker, "Docker documentation," 2016. [Online]. Available: https://docs.docker.com/

[19] H. Kim, "Checkpoint and restore docker container with criu," 2015. [Online]. Available: http://blog.circleci.com/checkpoint-and-restore-docker-container-with-criu/

[20] K. Merker, "How did the quake demo from dockercon work?" 2015. [Online]. Available: http://blog.kubernetes.io/2015/07/how-did-quake-demo-from-dockercon-work.html

[21] I. Miell and A. H. Sayers, "How to share docker volumes across hosts," 2015. [Online]. Available: https://jaxenter.com/how-to-share-docker-volumes-across-hosts-119602.html

[22] B. DeHamer, "Docker hub top 10," 2015. [Online]. Available: https://www.ctl.io/developers/blog/post/docker-hub-top-10/

[23] I. O. for Standardization, "De - iso 3166 - codes for the representation of names of countries and their subdivisions," 2013. [Online]. Available: https://www.iso.org/obp/ui/#iso:code:3166:DE

[24] C. Strimbei, O. Dospinescu, R.-M. Strainu, and A. Nistor, "Software architectures - present and visions," *Informatica Economica*, vol. 19, no. 4/2015, pp. 13–27, 2015. [Online]. Available: http://revistaie.ase.ro/content/76/02%20-%20Strimbei,%20Dospinescu,%20Strainu,%20Nistor.pdf

[25] J. Stubbs, W. Moreira, and R. Dooley, "Distributed systems of microservices using docker and serfnode." IEEE, 2015, pp. 34–39. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7217926

[26] S. Newman, *Building microservices: [designing fine-grained systems]*, 1st ed. Beijing: O'Reilly, 2015.

[27] G. Hohpe and B. Woolf, *Enterprise integration patterns: designing, building, and deploying messaging solutions*, ser. The Addison-Wesley signature series. Boston: Addison-Wesley, 2004.

[28] M. P. Papazoglou and W.-J. van den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB Journal*, vol. 16, no. 3, pp. 389–415, 2007. [Online]. Available: http://link.springer.com/10.1007/s00778-007-0044-3

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

[30] J. Choi, D. Nazareth, and H. Jain, "Implementing service-oriented architecture in organizations," *Journal of Management Information Systems*, vol. 26, no. 4, pp. 253–286, 2010.

[31] J. Thones, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7030212

[32] D. Tucker, "Docker networking takes a step in the right direction," 2015. [Online]. Available: https://blog.docker.com/2015/04/docker-networking-takes-a-step-in-the-right-direction-2/

[33] D. McGowan, "Proposal: Cross repository push," 2015. [Online]. Available: https://github.com/docker/distribution

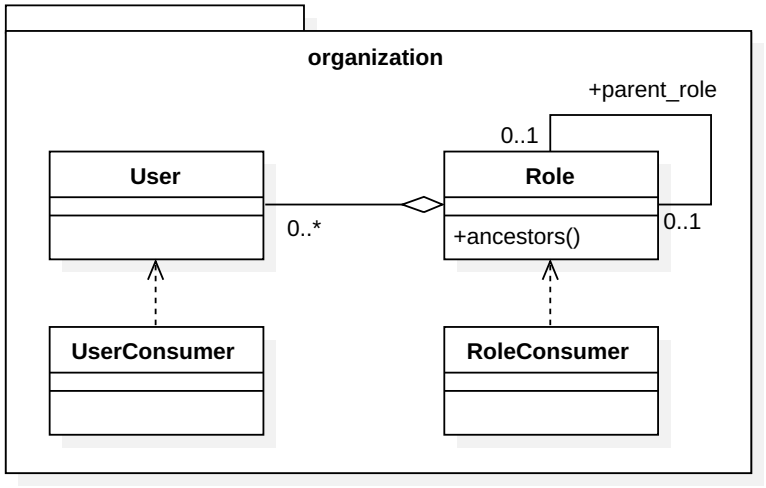
[34] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 1, 2015, pp. 778–788.

[35] M. Jazayeri, "Some trends in web application development," in *Future of Software Engineering, 2007. FOSE '07*, 2007, pp. 199–213.

[36] O. Ben-Kiki and C. Evans, "Yaml ain't markup language (yaml) version 1.2," 2009. [Online]. Available: http://www.yaml.org/spec/1.2/spec.html#id2759572

Appendix

A Architecture and design



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. A.1: UML Class Diagram for the Organization Service

B Implementation

** fix listings

B.1 Unterkapitel

```json
{
  "activities": [
    {
      "id": "81769dfd-6336-4604-a550-5264f1603269",
      "type": "start",
      "successors": [
        "a0990b04-7c47-4490-8981-8e434523121b"
      ],
      "predecessors": []
    },
    {
      "id": "5ff2892c-7325-439b-835f-d8f582005dd5",
      "type": "end",
      "successors": [],
      "predecessors": [
        "a0990b04-7c47-4490-8981-8e434523121b"
      ]
    },
    {
      "id": "a0990b04-7c47-4490-8981-8e434523121b",
      "type": "container",
      "successors": [
        "5ff2892c-7325-439b-835f-d8f582005dd5"
      ],
      "predecessors": [
        "81769dfd-6336-4604-a550-5264f1603269"
      ]
    }
  ]
}
```

3 Exported process definition in JSON format

```
# Base image
FROM ruby:2.2

# Shared files
RUN apt-get update && apt-get install -y git \
        && wget -qO- https://get.docker.com/ | sh \
         && rm -rf /var/lib/apt/lists/*

RUN mkdir /wfms
WORKDIR /wfms
ADD ["Gemfile", "Gemfile.lock", "/wfms/"]
RUN bundle install

# Activity files
RUN mkdir /activity
WORKDIR /activity
ENTRYPOINT ruby run.rb
ADD . /activity
```

4 Dockerfile for activity base image

```
# Base image
FROM ruby:2.2

# Shared files
RUN apt-get update && apt-get install -y git \
        && wget -qO- https://get.docker.com/ | sh \
         && rm -rf /var/lib/apt/lists/*

RUN mkdir /wfms
WORKDIR /wfms
ADD ["Gemfile", "Gemfile.lock", "/wfms/"]
RUN bundle install

# Workflow files
RUN mkdir /workflow
WORKDIR /workflow
ENTRYPOINT ruby run.rb
ADD . /workflow
```

5 Dockerfile for workflow base image

```yaml
version: "2"

networks:
  frontend:
    driver: overlay
  backend:
    driver: overlay
  enactment:
    driver: overlay

services:
  registry:
    image: registry:2.3
    restart: always
    ports:
      - 5000:5000
    networks:
      - backend
    environment:
      - "constraint:node==coordination-machine"

  mom:
    image: rabbitmq:3-management
    restart: on-failure:3
    networks:
      - backend
      - enactment
      - frontend
    ports:
      - "8080:15672"
    environment:
      - "constraint:node==internal-machine"

  engine:
    image: wf_engine
    build:
      context: ./engine
      args:
        - "constraint:node==development-machine"
    restart: on-failure:3
    depends_on:
      - mom
    networks:
      - backend
      - enactment
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ~/.docker/machine/certs/:/root/.docker
    environment:
      SWARM_MANAGER_CERT_PATH: /root/.docker
```

6 The whole Docker Compose file of the WfMS (1/5)

```
51
52    organization_db:
53      image: postgres
54      restart: on-failure:3
55      depends_on:
56        - organization_db_data
57      volumes_from:
58        - organization_db_data
59      networks:
60        - backend
61      environment:
62        - "POSTGRES_PASSWORD=masterarbeit"
63        - "POSTGRES_USER=organization"
64
65    organization_db_data:
66      image: cogniteev/echo
67      networks:
68        - backend
69      volumes:
70        - /var/lib/postgresql/data
71      environment:
72        - "constraint:node==development-machine"
73
74    organization:
75      image: organization
76      build:
77        context: ./organization
78        args:
79          - "constraint:node==development-machine"
80      restart: on-failure:3
81      depends_on:
82        - organization_db
83        - mom
84      networks:
85        - backend
86      environment:
87        - "constraint:node==development-machine"
88        - "POSTGRES_PASSWORD=masterarbeit"
89        - "POSTGRES_USER=organization"
90
91    worklist_db:
92      image: postgres
93      restart: on-failure:3
94      volumes_from:
95        - worklist_db_data
96      networks:
97        - enactment
98      environment:
```

7 The whole Docker Compose file of the WfMS (2/5)

```
99        - "POSTGRES_PASSWORD=masterarbeit"
100       - "POSTGRES_USER=worklist"
101
102   worklist_db_data:
103     image: cogniteev/echo
104     networks:
105       - enactment
106     volumes:
107       - /var/lib/postgresql/data
108     environment:
109       - "constraint:node==internal-machine"
110
111   worklist:
112     image: worklist
113     build:
114       context: ./worklist
115       args:
116         - "constraint:node==internal-machine"
117     restart: on-failure:3
118     depends_on:
119       - worklist_db
120       - mom
121     networks:
122       - enactment
123     environment:
124       - "constraint:node==internal-machine"
125       - "POSTGRES_PASSWORD=masterarbeit"
126       - "POSTGRES_USER=worklist"
127
128   definition_db:
129     image: postgres
130     restart: on-failure:3
131     depends_on:
132       - definition_db_data
133     volumes_from:
134       - definition_db_data
135     networks:
136       - backend
137     environment:
138       - "POSTGRES_PASSWORD=masterarbeit"
139       - "POSTGRES_USER=definition"
140
141   definition_db_data:
142     image: cogniteev/echo
143     networks:
144       - backend
145     volumes:
```

8 The whole Docker Compose file of the WfMS (3/5)

```
146       - /var/lib/postgresql/data
147     environment:
148       - "constraint:node==development-machine"
149
150   definition:
151     image: definition
152     build:
153       context: ./definition
154       args:
155         - "constraint:node==development-machine"
156     restart: on-failure:3
157     depends_on:
158       - definition_db
159       - mom
160     volumes:
161       - /var/run/docker.sock:/var/run/docker.sock
162       - ~/.docker/machine/certs/:/root/.docker
163     networks:
164       - backend
165     environment:
166       - "constraint:node==development-machine"
167       - "POSTGRES_PASSWORD=masterarbeit"
168       - "POSTGRES_USER=definition"
169
170   infrastructure:
171     image: infrastructure
172     build:
173       context: ./infrastructure
174       args:
175         - "constraint:node==development-machine"
176     restart: on-failure:3
177     depends_on:
178       - mom
179     networks:
180       - backend
181     volumes:
182       - /var/run/docker.sock:/var/run/docker.sock
183       - ~/.docker/machine/certs/:/root/.docker
184     environment:
185       - "SWARM_MANAGER_CERT_PATH=/root/.docker"
186       - "constraint:node==development-machine"
187
188   user_gateway:
189     image: user_gateway
190     build:
191       context: ./user_gateway
192       args:
193         - "constraint:node==internal-machine"
194     restart: on-failure:3
195     depends_on:
196       - mom
197     ports:
```

9 The whole Docker Compose file of the WfMS (4/5)

```
198       - "3001:3000"
199     networks:
200       - frontend
201     environment:
202       - "constraint:node==internal-machine"
203
204   developer_gateway:
205     image: developer_gateway
206     build:
207       context: ./developer_gateway
208       args:
209         - "constraint:node==development-machine"
210     restart: on-failure:3
211     depends_on:
212       - mom
213     ports:
214       - "3000:3000"
215     networks:
216       - frontend
217     environment:
218       - "constraint:node==development-machine"
219
220   provisioner:
221     image: provisioner
222     build:
223       context: ./provisioner
224       args:
225         - "constraint:node==cloud-machine"
226     restart: on-failure:3
227     volumes:
228       - /var/run/docker.sock:/var/run/docker.sock
229       - ~/.docker/machine/certs/:/root/.docker
230     environment:
231       - "SWARM_MANAGER_IP=192.168.99.101"
232       - "SWARM_MANAGER_CERT_PATH=/root/.docker"
233       - "constraint:node==cloud-machine"
```

10 The whole Docker Compose file of the WfMS (5/5)

Plagiarism declaration

I hereby declare that, to the best of my knowledge and belief, this Masterthesis titled "Prototypical Development of a Docker-based Workflow Management System" is my own work. I confirm that each significant contribution to, and quotation in this thesis from the work, or works of other people is indicated through the proper use of citations and references.

Münster, on the 03-08-2016