Topic:

# Prototypical Development of a Docker-based Workflow Management System

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

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

Submitted by: Lars Greiving
Dettenstraße 4
48147 Münster

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

Deadline: 2016-02-24

# Contents

## List of Figures

# List of Tables

## Abbreviations

| | |
|---|---|
| **API** | Application Programming Interface |
| **cgroups** | control groups |
| **CoW** | Copy-on-Write |
| **ESB** | Enterprise Service Bus |
| **GUI** | Graphical User Interface |
| **MSA** | Micro-services Architecture |
| **MOM** | Message-oriented Middleware |
| **OASIS** | Organization for the Advancement of Structured Information Standards |
| **OS** | Operating System |
| **PID** | Process Identifier |
| **REST** | Representational State Transfer |
| **SOA** | Service-oriented Architecture |
| **WFMC** | Workflow Management Coalition |
| **WfMS** | Workflow Management System |
| **WSDL** | Web Services Description Language |

# 1 Introduction and Motivation

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

## 1.1 Related Work

» Nunamaker, Chen and Purdin

# 2 Workflow Management Systems

In this chapter, the concepts of workflows and workflow management systems will be briefly introduced and related to each other. There is a plethora of term definitions and deviating understandings of workflows and the concepts related to them [5]. In large parts, the concepts presented here thus rely on specifications published by the Workflow Management Coalition (WFMC), a consortium of workflow management software vendors, researchers in the field of workflow management and Workflow Management System (WfMS) users, as they represent some form of consensus.

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

## 2.1 Concepts

### 2.1.1 Workflow

In order to achieve their business goals, organizations perform temporal and logical sequences of tasks that help to interact with business relevant entities. These sequences are known as *business processes*. If the logic that controls the processes is performed in an automated way, e.g. by an information system, one refers to the processes as *workflows* [2, 14]. The WFMC defines workflows as the computerized facilitation or automation of a business process, in whole or part [14].

*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 [14]. As the term "workflow activity" might be misunderstood as "any activity belonging to a workflow", in the following the term *automated activity* will be used instead.

### 2.1.2 Process Definition

In order to be able to execute workflows, the underlying business processes must be machine processable and thus have to be formalized from real world to an abstracted model [14]. This model is usually called *process definition* and stored in form of some high-level programming language construct [14, 25]. 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 [14]. They further may contain references to other processes, which are treated as a single activity in the process definition [14, 5].

- usually directed grpah - how stored?

### 2.1.3 Process Instance

A *process instance* is an enactment of a process definition. A process definition may be instantiated multiple times, even at the same time. [5]. If only the automated parts of such an instance are meant, the WFMC advocates for the term *workflow instance* [14].

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 [14].

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

### 2.1.4 Activity Instance

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

When an activity instance is created, it is in the *inactive* state. From this state, it may enter the *suspended* state, in which it will neither be activated nor assigned a worklist item. If the activity instance is not suspended, it is activated once its entry conditions are fulfilled. It then is in the *active* state. When the execution of the activity has finished, it finally enters the *completed* state [14].

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

### 2.1.5 Workflow Data

In a WfMS, several forms of data may occur at diverse occasions. The WFMC differentiates between three types of data: workflow relevant data, workflow application data, and workflow control data [14].

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 [14].
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 [14, 5].
Data that is internally managed by a WfMS is refered to as *workflow control data*. This data usually comprises the states of process- and activity instances and other internal statuses and is per se not interchanged in its default form [14, 5].

Russel et al differentiate seven commonly used forms of data visibility in workflow management systems [21, p. 6-15]:

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

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

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

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

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

- **Workflow Data**

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

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

They further identified six types of data interaction between the various hierarchy levels in workflows [21, p. 16-24]:

- **Activity – Activity**
  Data is passed between one activity instance and another within 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.

### 2.1.6 Workflow Participant and Worklist

There are workflows that contain activities which require user interaction. A WfMS thus provides the functionality to assign workflows and activities to workflow participants. This assignment can either be a specific one, targeting one single person, or be more general, targeting a set of users from which whe WfMS may choose during execution time. These sets are

usually based on an organizational structure that manifests itself in roles, of which an user may have one or more [14, 5].

Each user has a so called *worklist* that consist of activities to which he is assigned to and which are scheduled for execution. Depending on the actual implementation, activites may appear on multiple users' worklists until one of them signals that he/she will work on it [14, 5].

## 2.2 Typical Architecture

With a growing number of workflows in an organization, the need arises to manage their creation, distribution and execution in a structured manner. An information system is called WfMS, if it is able to define, create and manage the execution of workflows by using software that runs on one or more workflow engines, is able to interpret process definitions, can interact with involved participants, and may invoke external applications [16]. 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" [14].

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

### 2.2.1 Functional Areas

The WFMC divides the reponsibilities of a WfMS in three functional areas: *build-time* functions, *run-time process control* functions and *run-time activity interaction* functions [14, 1].

The *build-time* functionalities are concerned with the abstraction of workflows, i.e. the creation of process definitions.
The *run-time process control* functionalities of a WfMS are dealing with instanciating and controlling processes, coordinating the execution of activities within a process instance, initiating (but not performing) both participant interaction and application invocation [14].
Some activites require users to enter data or applications to perform a specific task. The *run-time activity interaction* functions of a WfMS provide the possibilities to do so. They make forms available to users, instruct other applications, and collect the respective outcome [14].

### 2.2.2 System Components

The WFMC identified four software components that most WfMSs have in common: *Process Definition Tools*, *Administration and Monitoring Tools*, *Workflow Client Applications*, and *Workflow Enactment Service* [14].

#### Process Definition Tool

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

The WFMC notes, that process definition tools do not necessarily have to be part of a WfMS, since the definition may take place in another tool as long as it is passed along in a standardized format [14].

#### Administration & Monitoring Tools

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

#### Workflow Client Applications

The core function of the workflow client applications is to let the user retrieve worklist items that were assigned to him/her. In the WFMC reference model they are thus sometimes referred to as *worklist handlers* [14].
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* [14]. The user interface may be part of the workflow client applications or exist as a separate software component.

#### Workflow Engine

Workflow engines provide the runtime control environment for the execution of workflow instances, that is, they interpret the process definition, manage the instances' status, update

worklists, determine participants, and invoke external applications. They further manage the storage and flow of workflow control data and workflow relevant data [14].

#### Workflow Enactment Service

The Workflow Enactment Service groups one or more workflow engines into one logical component that exposes a single coherent external interface to other software [14].

### 2.2.3 WFMS Implementation Structure

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

# 3 Docker

When multiple applications or application instances are intended to run on one physical machine without interfering with each other, they are usually isolated in terms of execution environments and provided with a controllable share of system resources [11]. These goals can be fulfilled by both virtual machines and software containers [20]. The difference between these two options and the basic principles of software containers are shown in 3.1.

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

## 3.1 Concepts

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

### 3.1.1 Virtualization and Software Containers

The goal of *virtualization* is to simulate the presence of multiple computers on one machine. The use of this is XXX. There are two kinds of virtualization, one that takes place on the hardware level and another that takes place on the Operating System (OS) level [20].

#### Hardware-level virtualization

In most cases when speaking about virtualization, *hardware-level virtualization* is referred to. It is usually driven by a *hypervisor* – a service that manages virtual machines and provides them with abstracted hardware devices to run on. This hypervisor either either runs in the OS of the host machine or directly on its hardware [20].
The virtual machines, i.e. the computers simulated on the host machine, require their own OS to be installed.

**OS-level virtualization – or container-based virtualization**

The other kind of virtualization, *OS-level virtualization*, is the one that Docker makes use of. It utilizes functions of the host kernel which allow the execution of several isolated userspace instances that share the same kernel, but may differ in terms of their runtime environment, e.g. file system or system libraries. These isolated userspace instances are usually called *software containers* or just *containers*. This type of virtualization is therefore also referred to as *container-based virtualization* [20].

The isolation and resource management in container-based virtualization on Linux systems are mainly achieved by two mechanisms, *control groups (cgroups)* and *namespaces*. While the former allows to group processes and manage their resource usage, the latter can be used on many system components. Namespaces may be introduced for example on network interfaces, the file system, users and user groups, Process Identifier (PID)s, and other components, in order to achieve a fine grained control over the respective isolation [20].
Besides Docker, there are several solutions that are all based on the aforementioned kernel features, e.g. LXC, LXD, lmctfy, systemd-nspawn, etc [20]. There are ongoing efforts to create a common container standard [15].

Many container solutions rely on a strategy called *Copy-on-Write (CoW)* to provide a runtime enviroment, which on the one hand lets the containers reuse system libraries and the like while on the other hand limits the container in affecting its surroundings [8, 18]. This strategy is explained in a more detailed fashion in 3.1.2 on the example of Docker.

### 3.1.2 Docker Images and Containers

CoW is a strategy which makes use of the benefits of both sharing files for read access and copying them to a local version previous to changing them. Processes that require access to a file share the same instance of that file. As soon as one process needs to alter the file, the operating system creates a copy to which only the process has access to. All other processes still use the original file [18, 8].

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

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

can be considered as an image, i.e. a set of read-only layers, with a writable layer on top of it – the *container layer*. Write operations within a container trigger a CoW operation which copies the targeted file to the container layer, where the write operation is then performed.
Besides reducing the amount of space consumed by containers, the CoW strategy also reduces the time required to start a container. This is because Docker only has to create the container layer instead of providing a copy of all the filed contained in the respective image [8].

- *lifecycle of a docker container here*

### 3.1.3 Data Volumes

Any data written to the container layer is deleted as soon as its Docker container is deleted. Also, Docker containers that store a lot of data are considerably larger than Docker containers that do not, since the write operations require space in the container layer. This is the reason why data volumes exist – they are designed to persist data. Data volumes are directories or files that are mounted directly into a Docker container and thus bypass the storage driver [9]. They are never deleted automatically and therefore must be cleaned up manually when they are not needed anymore [8].

### 3.1.4 Dockerfiles

Instead of manually creating a container, running commands on it and then commiting it to create an image, Docker can be instructed by a recipe file – the *dockerfile*. In this file, the user states an image that the new image should be based on and the commands that otherwise would be entered manually [9].
To build an image, Docker is given a Dockerfile and a directory with files required for the build, the *context*, which is usually the directory the Dockerfile is located in. This enables Docker to copy files from the context to some layer within the image, if needed [9].

### 3.1.5 Registries and Repositories

A registry stores named Docker images and distributes them on request. Each image may be available in different tagged versions in a registry [8].
Within a registry, images may be organized in collections, which are called *repositories* [9].

### 3.1.6 Docker Networks

As mentioned in 3.1.1, Docker features virtual networks in order to isolate containers in this regard, but at the same time allow containers to communicate with the host, each other and the outside world. These networks are based on virtual interfaces and are managed by the Docker daemon. Containers may be member of multiple networks at the same time [8].

By default, Docker installs three networks: a *bridge* network, a *host* network, and a *none* network. The *bridge* network, titled *docker0,* is a subnetwork that is connected to the host's networks. Docker connects containers to this network if it is not instructed otherwise. Containers that are membsers of this network can communicate with each other by using their respective IP addresses. They also may expose ports that can be mapped to the hosts network, which makes applications in them accessible from the outside.
The *host* network represents the actual hosts network. If containers are assigned to this network, they will be placed in the hosts network stack, i.e. all network interfaces defined on the host are available to the container [8].
The *none* network provides containers with their own network stack. Containers that are only members of the *none* network are completely isolated in regards to network communication, unless futher configuration is undertaken [8].

Besides the network types mentioned above, Docker features another type of network, the *overlay* network. Overlay networks are virtual networks that are based on existing network connections. They are intended to simplify the communication between containers running on multiple hosts which, in turn, run on multiple machines themselves. If a container is member of an overlay network, it is able to communicate with all other containers that are also part of this network, no matter which Docker host (or host machine) they are running on [8].
Docker's overlay network requires a key-value store to be present in order to persist information on its own state, e.g. on lower level networks that it relies on, network members, etc.

## 3.2 Docker Engine

The Docker Engine forms the core of Docker. Docker uses a client-server architecture: it features a daemon which provides the functionality and a client that controls said daemon [10]. Together, they enable the user to work with Docker containers. Both the client and the daemon may run on the same system, or be connected remotely via sockets or through a Representational State Transfer (REST) Application Programming Interface (API) [8].

## 3.3 Docker Ecosystem

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

### 3.3.1 Docker Swarm

Docker Swarm allows applications which rely on several Docker containers to be run on a cluster of machines. It provides an abstraction that lets a set of Docker Engines behave like a single Docker Engine. Further it features a mechanism that automatically assigns container to a specific host based on given rules [7].

A swarm setup typically consists of one or more *swarm managers*, multiple Docker hosts, and, in case that no remote discovery service is used, a local discovery service. By default, every new container is assigned to a swarm-specific overlay network [8].

Docker Swarm provides two kinds of mechanisms for the assignment of containers to Docker hosts, *filters* and *strategies*. Strategies tell Docker how to rank hosts for assignment by some specified criteria, e.g. resource usage or number of deployed containers.
Filters allow to specify rules, which Docker tries to apply when searching for an assignment target. Possible rules could for example be matchers for the host's name or identifier, its OS, or for custom tags, which may describe the host's role or properties like size of attached storage. It is also possible to declare the affinity of certain containers or images for being deployed on the same host [8].

### 3.3.2 Docker Machine

The goal of the Docker Machine tool is to facilitate the setup of Docker hosts. In order to fulfill this goal, Docker Machine creates one virtual machine per requested host [7, 8]. This has several reasons. First, this proceeding allows several Docker hosts to run on the same machine without having them interfere with each other. Second, it enables machines with OSs, which natively do not support Docker and Docker containers, to act as a host [8]. And third, as the virtual machine image is known, it lets the setup procedure make assumptions on its environment, which simplifies the installation and configuration of the Docker Engine.

### 3.3.3 Docker Compose

Docker Compose is a tool that enables the user to specify and run applications that consist of many containers. Similar to the way an image is described in a Dockerfile, the user lists the required containers and their respective run configuration in a YAML (?) file. Docker Compose interprets this file and sets the containers up accordingly [7].

### 3.3.4 Docker Hub

# 4 Application Design

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

## 4.1 Determination of Objectives

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

### 4.1.1 Functional Objectives

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

#### Infrastructure and Infrastructure Management

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

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

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

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

**Tab. 4.1:** Objectives and their respective requirements

### Workflow Modeling

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

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

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

### Workflow Distribution

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

### Workflow Execution

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

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

In 2.1.5, various forms of data visibility and interaction were presented. Russel et al examined the capabilities of various workflow engines with regards to these characteristics [21]. In order to obtain a WfMS with useful functionality, it should support at least those forms of data visibility and interaction that are common among existing solutions. As a rough estimation for this, each capability shall be deemed as required if a majority of solutions examined in that study supports it.

### 4.1.2 Intangible Objectives

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

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

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

## 4.2 Architecture

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

### 4.2.1 Application Structure

Micro-services, service discovery [23]

Use Docker only for execution vs docker all the things

Developers of software systems have to cope with factors which impose challenges on them, such as high complexity within their systems, an increased need for integration of internal and external functionality and evolving technologies. Several architectural approaches emerged from the attempt to overcome these challenges. Strîmbei et al consider *monolithic architecture*, *Service-oriented Architecture (SOA)* and *Micro-services* to be the most relevant [22, 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 [23, p. 35]. They communicate through shared memory and direct function calls. Monolithic applications are typically written using one programming language [22, p. 14]. In order to cope with increasing workload on a monolithic system, multiple instances of it are run behind a load balancer [23, 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 [23, 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 [22, 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 [22, p. 14].

The weaknesses of monolithic architecture stem from its cohesive nature. As its components are usually tightly coupled, changes to one component can affect other parts of the application, which complicates the introduction of new components and the refactoring of existing ones [23]. Components cannot be deployed individually, which hinders reuse of functionality across several applications more difficult and makes scaling of single bootleneck components impossible [23]. Also, if the application runs in a single process, the failure of one component may bring down the whole application [17, p. 5].

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

### Service-oriented Architecture

SOA is based on the idea that code which provides related business functions can be bundled into one component which offers said functionality to other systems *as a service*, thus avoiding duplicated implementation of the functionalities among these systems [13, p.8]. An application may then use several services in order to fulfill its own business function [19, 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 [12]. 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* [19, p. 390], [22, 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 [19, 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 [19, 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 invokation of a service and its result to be formed as expected [19, 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 [19, p. 390]. SOA thus enhances an organization's ability to respond quickly to changes [19, p. 390], [6, p. 254]. Since legacy applications can be provided with appropriate interfaces, SOA can help to integrate and extend them [19, 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 [19, 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 [19, p. 408f].

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

**Micro-services Architecture**

The concept of Micro-services Architecture (MSA) is closely related to that of SOA, as it also promotes the encapsulation of functionality in standalone services which can be used by other parts of a system. There is unambiguity whether MSA is actually a concept on its own – or rather a specialized application of SOA [23, p. 35], [22, p. 17]. Stubbs et al describe MSA as a distributed system that consists of independent services which are narrowly focused and thus considered "lightweight" [23, p. 35]. That exact principle has been described as a version of SOA before [19, 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 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

[22, 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 [23, p. 35], [17, p. 4], [24, p. 113]. The narrow focus of each service makes it less specialized to certain uses, which should theoretically enable better reuse of code [23, 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 [17, p. 5]. Due to the properties of the MSA, micro-services may be deployed, upgraded and scaled individually [24, p. 116].

Researchers also see disadvantages and problems that may go hand in hand with the use of MSA. While MSA facilitates deploying parts of an application individually, the overall amount of work required for the deployment of all services is higher and the coordination of the deployments is complexer than the deployment of a monolithic application. As services may be unavailable at times, a mechanism has to be in place that allows the discovery of services (such as the Message-oriented Middleware (MOM)) [23, p. 35]. Unless a dedicated service is introduced, there is no central access to the services' logs. Aggregation, analysis and fixing errors is thus more complicated in comparison to monolithic architecture [23, p. 35]. In order to define the different services, it is necessary to find the right size for each service, i.e. the appropriate scope of its functionality. This process is difficult and not needed to this extend in monolithic architectures or SOA.

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

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

**Choice of an architecture model**

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

While measures can be taken in monolithic applications to cope with failure of components to some degree, if the underlying machine fails or the process dies for whatever reason, the whole

application is rendered inoperative [17, p. 55]. SOA and MSA both urge to account for the possibility of a non-responding service – in the first place because of the unreliability of network communication, but that works or any other reason, too. They hence inherently support the objective of resilience better than monolithic architecture.

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

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

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

The Docker Ecosystem facilitates the setup of the infrastructure for a MSA. As stated in 4.2.1, the MOM itself contains little to no knowledge about the system using it. Thus, the MOM and all application components may simply be started in separate containers and connected using an overlay network.

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

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

### 4.2.2 Components

As noted in 4.2.1, one of the downsides of MSA is, that suitable service boundaries have to be determined. Some sources advise to first build a monolithic application and then analyze the result to single out services that can be extracted. In case of a WfMS, the identification of system components by the WFMC can be interpreted as such an analysis. Based on those components, further subordinate micro-services are then identified.

- wfmc - developer gateway - user gateway - workflow modeling service - environment management service - organization management service - worklist service - workflow engine service - workflow engine service

- extra - provisioning service - validation service

**Workflow Modeling Service**

**Environment Management Service**

**Organization Management Service**

**Worklist Service**

**Workflow Engine Service**

**Developer Interface**

**User Interface**

**Provisioning Service**

- either one or more “centralized” - or one for each

**Validation Service**

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

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

Given the decisions that ware already made, the exchange of data between the several execution time actors can happen on several ways.

| Objective | Requirements |
|---|---|
| **Support of data visibility** | • Activity Data<br>• Sub-workflow Activity Data<br>• Multiple Instance Activity Data<br>• Workflow Instance Data<br>• Workflow Data<br>• Environment Data |
| **Support of data interactions** | • Activity → activity<br>• Sub-workflow activity → sub-workflow<br>• Sub-workflow → sub-workflow activity<br>• Workflow instance → workflow instance |

**Tab. 4.2:** Objectives and their respective requirements

# 5 Prototypical Implementation

## 5.1 Toolchain

## 5.2 Realization of the Architecture

Any Compromises here?

# 6 Evaluation and Discussion

# 7 Conclusion

## Bibliography


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

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

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

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

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

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

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

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

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

[10] Inc. Docker. Docker.com.

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

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

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

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

[15] Open Container Initiative. Open containers initative.

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

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

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

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

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

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

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

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

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

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