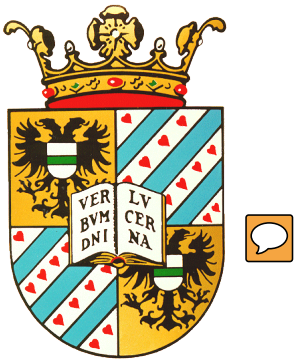
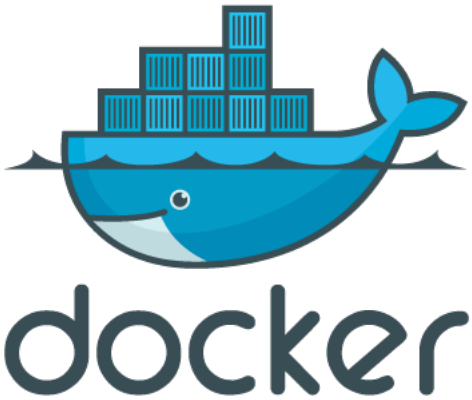


University of Groningen

Software Patterns
Team 2

Pattern-based Recovery & Evaluation: Docker



Authors:
Putra, Guntur
Fakambi, Aurélie
Schaefers, Joris
Menninga, Wouter

Monday 4th January, 2016

Version 0.2

Authors

Name	E-Mail
Putra, Guntur	G.D.Putra@student.rug.nl
Fakambi, Aurélie	A.Fakambi@student.rug.nl
Schaefers, Joris	J.Schaefers@student.rug.nl
Menninga, Wouter	W.G.Menninga@student.rug.nl

Revision History

Version	Author	Date	Description
0.1	Menninga	12-12-15	Added stakeholders and concerns.
	Schaefers	11-12-15	Added stakeholder concern table
	Schaefers	11-12-15	Added key drivers
	Putra	13-12-15	First draft of introduction chapter.
	Fakambi	11-12-15	System context
0.2	Menninga	29-12-15	Improved stakeholders section
	Menninga	02-01-16	Added Client-Server pattern
	Putra	03-01-16	Revised the introduction.
	Menninga	03-01-16	More additions client-server pattern + added security to stakeholders
	Putra	03-01-16	Added layers pattern documentation.
	Menninga	04-01-16	Added Plugin pattern.
	Menninga	04-01-16	Some improvements to System Context.
	Schaefers	04-01-16	Changed the names of the stakeholders concerns to include both the main- and sub quality
	Schaefers	04-01-16	Added image showing the relation between the stakeholders and the concerns
	Schaefers	04-01-16	Changed the key drivers, added references, changed the explanations

Contents

List of Figures

List of Tables

1 Introduction

This document presents architecture recovery of Docker[1] by identifying software patterns and performing an evaluation of the architecture based on the identified patterns. This document is part of the Software Pattern assignment at the University of Groningen.

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux [12].

This project utilizes the IDAPO[2] process to recover the architecture [13]. The PBAR[3] approach is used to perform the evaluation [8].

The rest of the document is explained as follows. Chapter 2 gives brief explanation with regard to Docker. Chapter 3 elaborates on the stakeholders involved in the Docker project and its corresponding key-drivers. Patterns discovered in the Docker project are documented in chapter 5. Evaluation is presented in chapter 6. Chapter 7 gives several recommendation for the Docker project. Lastly, a conclusion is drawn in chapter 8.

[1] `https://www.docker.com/`
[2] Identifying Architectural Patterns in Open Source Software
[3] Pattern-Based Architecture Reviews

2 System Context

In this chapter the context of Docker is presented.

2.1 System Context

Docker is a open-source-software designed with the purpose of making the deployment of distributed application easier. 'Docker provides an integrated technology suite that enables development and IT operations teams to build, ship, and run distributed applications anywhere'[5].

Docker helps to automate the deployment of applications inside so-called 'containers' by making use of operating-system-level virtualization.

Operating-system-level virtualization is a virtualization method where the kernel of an operating system allows the isolation of multiple instances of the user space of the operating system. Leveraging this type of virtualization over full virtualization (i.e. hardware emulation) offers a significant performance increase, because there is no need for emulation of the hardware.[14]

Figure 2.1 illustrates the difference between virtualization using hardware emulation and operating-system-level virtualization. As opposed to a virtual machine, a container does not have a guest operating system. Additionally, it does not need a hypervisor for emulating virtual hardware and trapping privileged instructions. This saves a lot of performance overhead.

One tool that makes use of operating-system-level virtualization is Docker. Docker can be used to package applications and dependencies in a 'container'. Running this container using Docker will run it within a separate user space using operating-system-level virtualization.
Docker containers are instances of running Docker images. The process of creating a Docker image is relatively uncomplicated. It involves the creation of a so-called 'Dockerfile', which is a type of Makefile, containing instructions that are to be executed to install the application and dependencies (using the `RUN` instruction), and copy relevant data into the container (using `ADD`). Every Dockerfile extends another image (using the `FROM` instruction), which is either a previously created image or a base image (often containing the files of a specific operating system).

Docker images use a layered file system. Every command which is executed during the build-process creates a new layer, where all modifications to the file system are stored. Docker supports exporting images to tar-archives, which can be imported between similar versions of Docker.

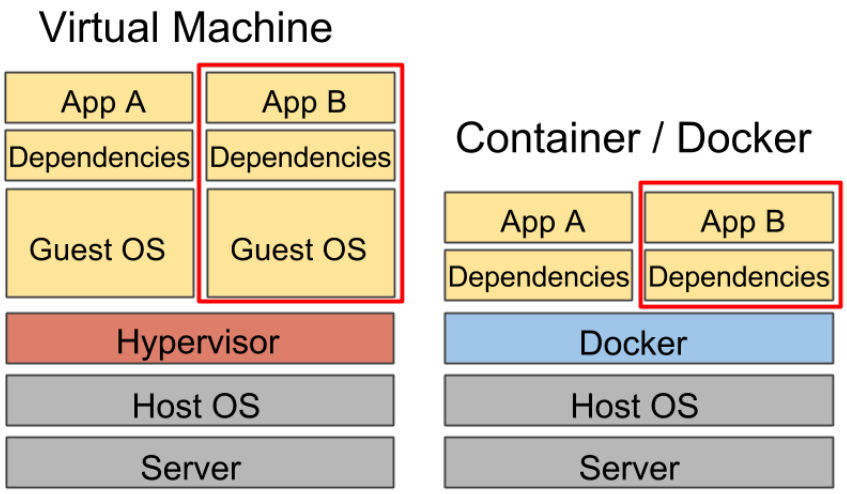


Figure 2.1: Virtual machines compared to containers[11]

The first open-source version of Docker was released in March 2013. Docker is written in the Go Programming language

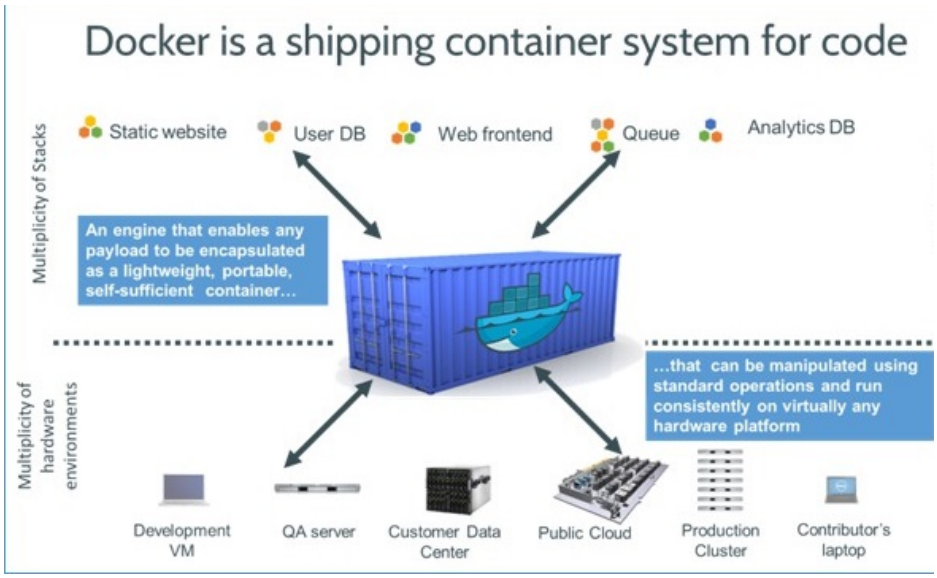


Figure 2.2: Docker

2.2 Community

Docker started as an internal project within dotCloud, a platform-as-a-service company with four main developers.
The following organizations are the main contributors to Docker: the Docker team, Red Hat, IBM, Google, Cisco Systems and Amadeus IT Group.

There are several ways to get in contact with the community of Docker:

- IRC – Here, the most knowledgeable Docker users reside. At the `#docker` and `#docker-dev` groups on `irc.freenode.net`.
- Google Groups – The developers are active in the `docker-dev` Google group.
- Stack Overflow – Questions about Docker itself can be asked on the Stack Overflow.

3 Stakeholders and Concerns

This section defines all stakeholders of the Docker system and describes the concerns of these stakeholders. A stakeholder is a person, group of persons, or organization that are involved in our system. This section uses the quality attributes as described in ISO-25010[10].

3.1 Stakeholders

Docker has the following stakeholders:

- Software developers
- Software Maintainers
- Cloud providers
- The open container initiative
- Docker developers
- Plugin developers

Software Developers

The developers are those that use Docker to create their software systems. It is used by developers to deploy software and create architectures consisting of multiple containers. Developers using Docker for their system, expect it to be reliable and want it to have a good usability.

Concerns

Usability	Since the software developers are using Docker, they want it to have good usability. This means it is easy to learn how to use Docker and it is not hard to work with. In fact, this is one of the main benefits of Docker, since it makes it easier to containerize applications, which could be done before, but was very difficult to do in practice.
Functional Suitability*: (Functional correctness)*	Software developers care about the functional correctness, since bugs in the Docker software makes the development of their software more difficult.

Software Maintainers

Software maintainers are responsible for deploying the software and keeping the software product running. Docker is often used for software deployment (especially to the cloud). The software maintainers expect Docker to be reliable and portable.

Concerns

Reliability	Software maintainers are responsible for keeping the software working while it is deployed. They will only use Docker if it is reliable. It has to be tolerant against failing containers.
Portability	The software maintainers want to be able to run Docker and its containers on a variety of different environments.
Performance efficiency	The software maintainers want Docker to have a good performance. They want the resources of their servers to be used as efficiently as possible and therefore want Docker to have no overhead, like Virtual Machines often do have.

Cloud providers

There are numerous cloud providers offering services which are based on Docker[1]. These cloud providers offer Container-based cloud computing, sometimes referred to as CaaS (Containers-as-a-Service).

Concerns

***Portability**: (Installability)*	The cloud providers want to integrate Docker into their cloud computing architecture.
***Compatibility**: (Co-Existence)*	Docker has to share the environment with the existing architecture of the cloud providers.
Reliability	The customers using the cloud providers' services expect a good reliability. If cloud providers are to implement Docker in their architectures, they need Docker to have good reliability.

The Open Container Initiative

The Open Container Initiative[2] was formed with the purpose of creating an open industry standard for container formats and runtime in June 2015.

They are interested in creating a formal, open, industry specification around container formats and runtime. This specification should be independent of particular clients/orchestration stacks, commercial vendors or projects and should be portable across a wide variety of operating systems, hardware etc.

Docker has donated its container format and runtime, knows as 'runC' to this initiative and is one of the members of the initiative, together with a lot of other members (including competing technologies, such as 'rkt' from CoreOS)[3].

Concerns

Portability	The main goal of the initiative is to standardize the container format and runtime used also by the Docker project. This allows users of Docker to use the Docker containers with other container runtimes.
***Compatibility**: (Interoperability)*	Docker has to be able to work with the containers from the initiative.

Docker developers

The Docker developers are the developers contributing to the Docker code base.

Concerns

Maintainability	These developers contribute new features and bugfixes to the existing code base. Therefore, they want the project to have good modifiability, such that additions and improvements can be realized without to much effort.
Security	The Docker developers care about the security of the product they are working on. Their changes will not be accepted by the community if they can introduce security flaws.

Plugin developers

Docker allows extending its capabilities with plugins[4]. These plugins are created by the plugin developers.

Concerns

***Portability**: (Adaptability)*	The developers of the plugins want to extend the functionality of Docker in an effective and efficient way.

[1] `https://www.docker.com/partners#/service`
[2] `https://www.opencontainers.org/`
[3] A list is available at `https://www.opencontainers.org/about/members`
[4] `https://blog.docker.com/2015/06/extending-docker-with-plugins/`

Overview

The stakeholders and their related concerns is visualized in the figure below.

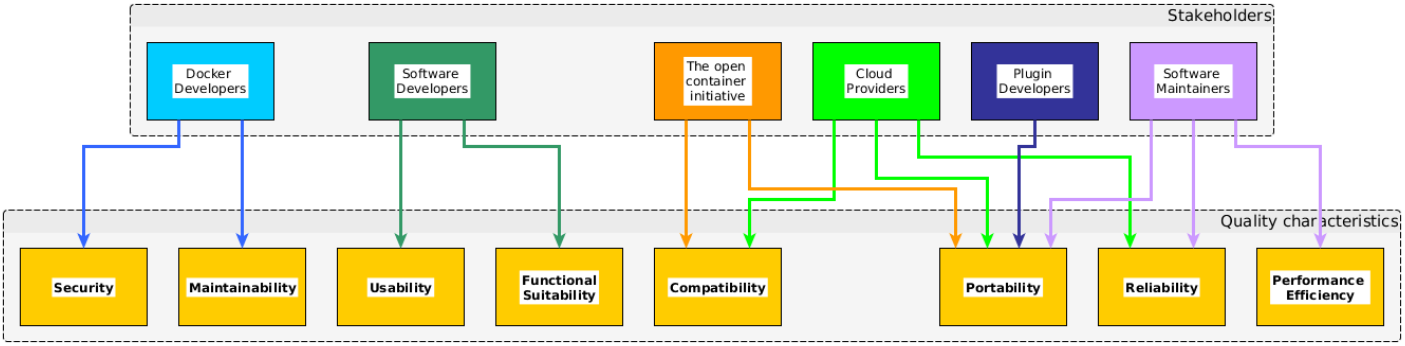


Figure 3.1: Stakeholders and the quality attributes they are concerned about

3.2 Key Drivers

Docker has the following key drivers

- KD- 1 – Portability
- KD- 2 – Reliability
- KD- 3 – Compatibility

3.2.1 Portability

Looking at figure 3.1, which visualizes the quality concerns of the stakholders, it is apparent that portability is the quality that is most desired among the stakeholders. One of the main features, if not the main feature of docker, is to provide "agility and control for Development and IT Operations teams to build, ship, and run any app, anywhere" [2].
Docker want's the containers running the applications to be "hardware-agnostic" and "platform-agnostic" [7], meaning that docker containers do not know anything about the hardware, nor the platform it is run on, thus making it very portable. This allows developers to focus on the actual application, without having to worry about the underlying hardware or platform. The applications the developers create could then be run everywhere where docker is supported.
To provide this quality attribute, Docker is also focused on increasing the performance. By making the docker containers small, yet having a high performance, the containers become usable by a wide variety of systems and are able to replace the need to use any virtual machines that would otherwise be used to achieve this level of portability. [7]
To increase the portability even more, the docker hub makes it very easy for docker containers to be shared and used on other machines.

3.2.2 Reliability

The main function of a docker container is to run a certain application. However, a system usually needs to run a set of different applications and should not be the case that running these applications in docker compromises the system's reliability.
Dependencies often lead to problems [7], decreasing the reliability of software. The Docker files are text files containing a small set of instructions that will run an application. This means that the instructions needed to run an application are conveniently located in a single place, the docker files. This allows viewing and the dependencies of all the different applications and making them easily maintainable.
Docker increases the reliability of the applications it runs, by increasing the testability of the containers and applications.
To make sure docker itself stays reliable, docker provides a test framework and test functions that thoroughly

test docker.

3.2.3 Compatibility

The main goal of the docker project is to “Build,Ship, and Run Any App, Anywhere” [2]. To accomplish this, the docker containers need to be able to run on many different kinds of systems and platforms.
Docker containers behave like a virtual machine and can easily be configured to communicate with each other. Docker also allows running the same application on multiple platforms. This allows the underlying operating system of an application to be changed, without compromising the system as a whole.

4 Software architecture

This section discusses the architecture of Docker.

4.1 Development View

4.1.1 Components

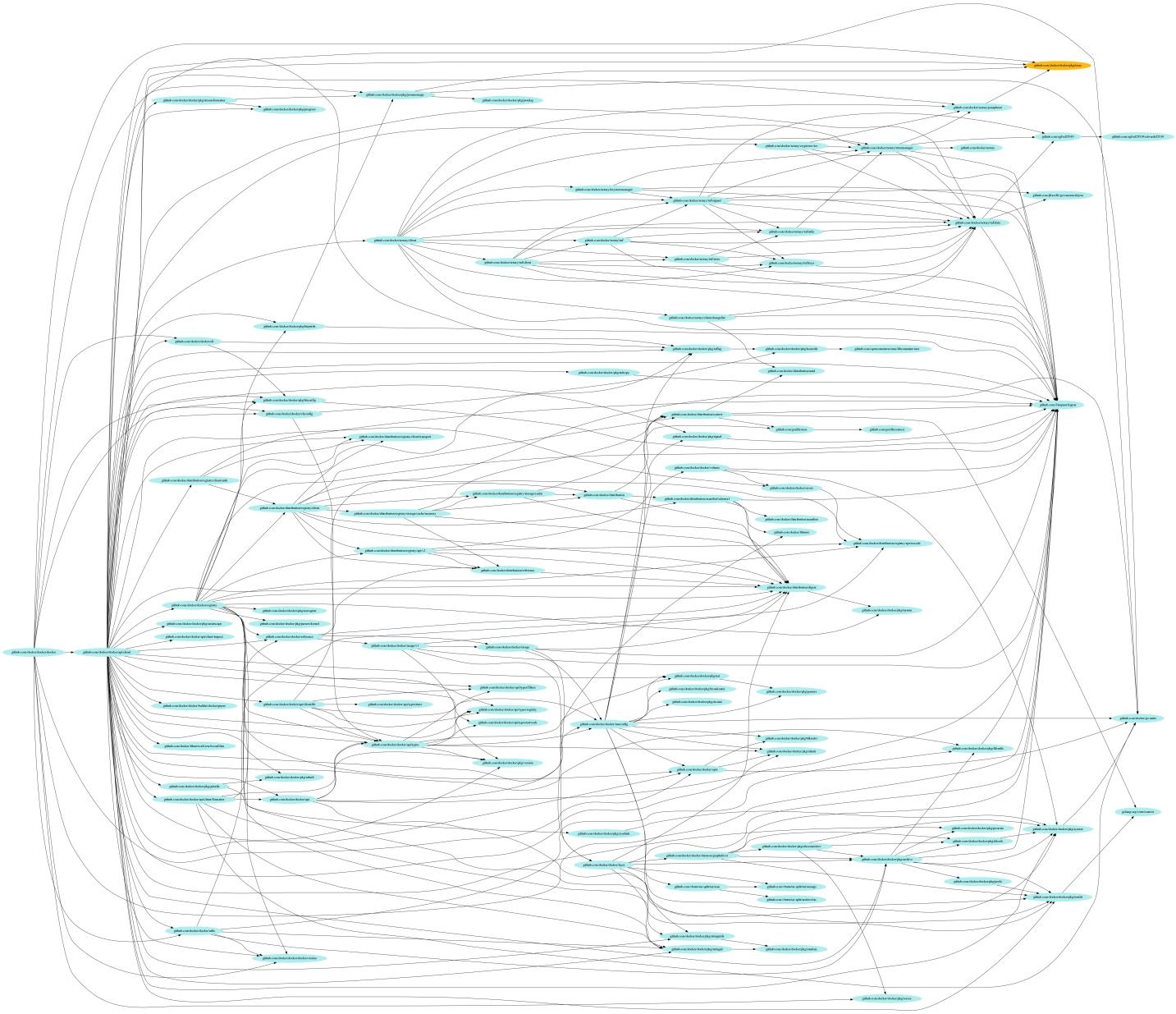


Figure 4.1: Dependency graph docker

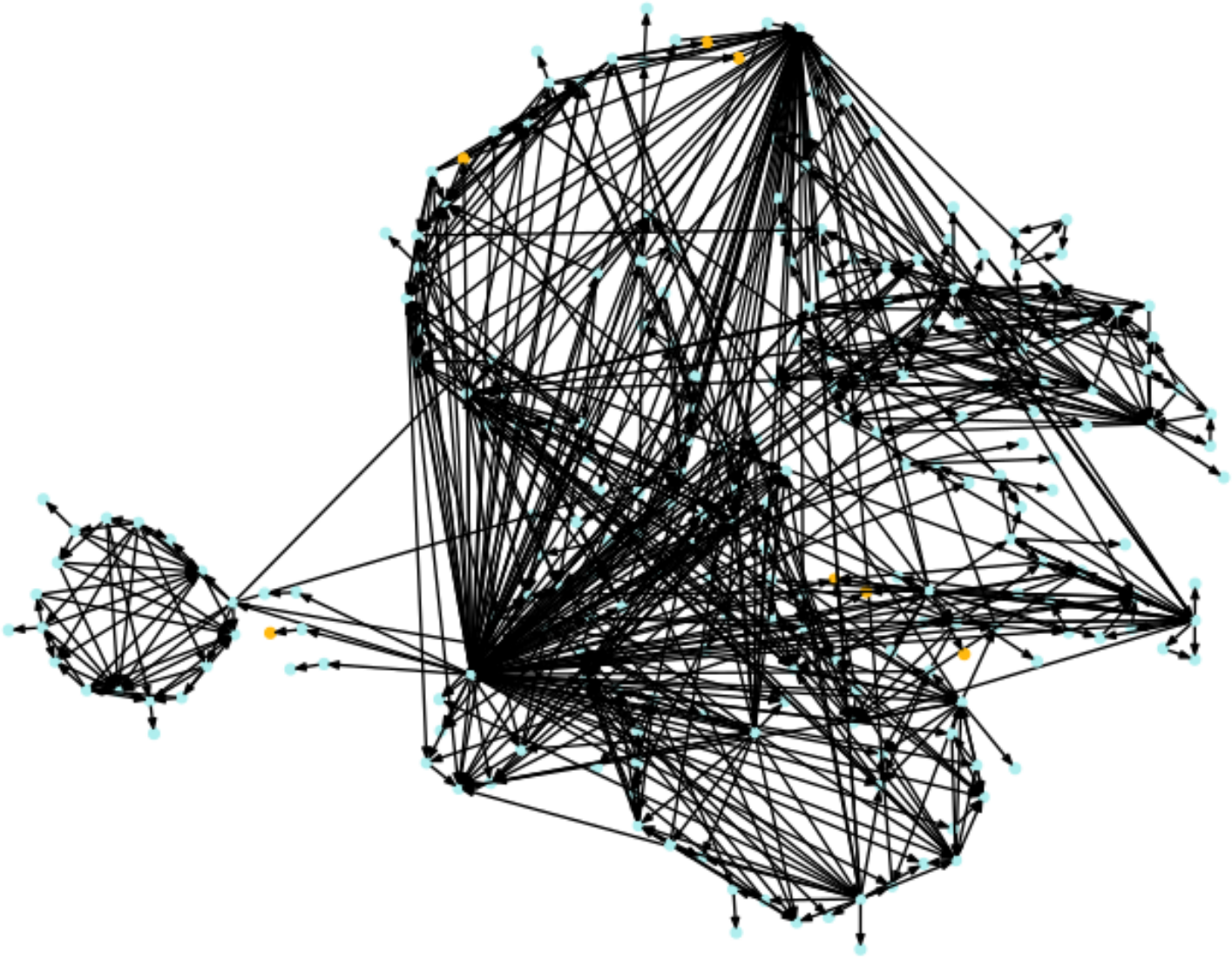


Figure 4.2: Dependency graph daemon circular

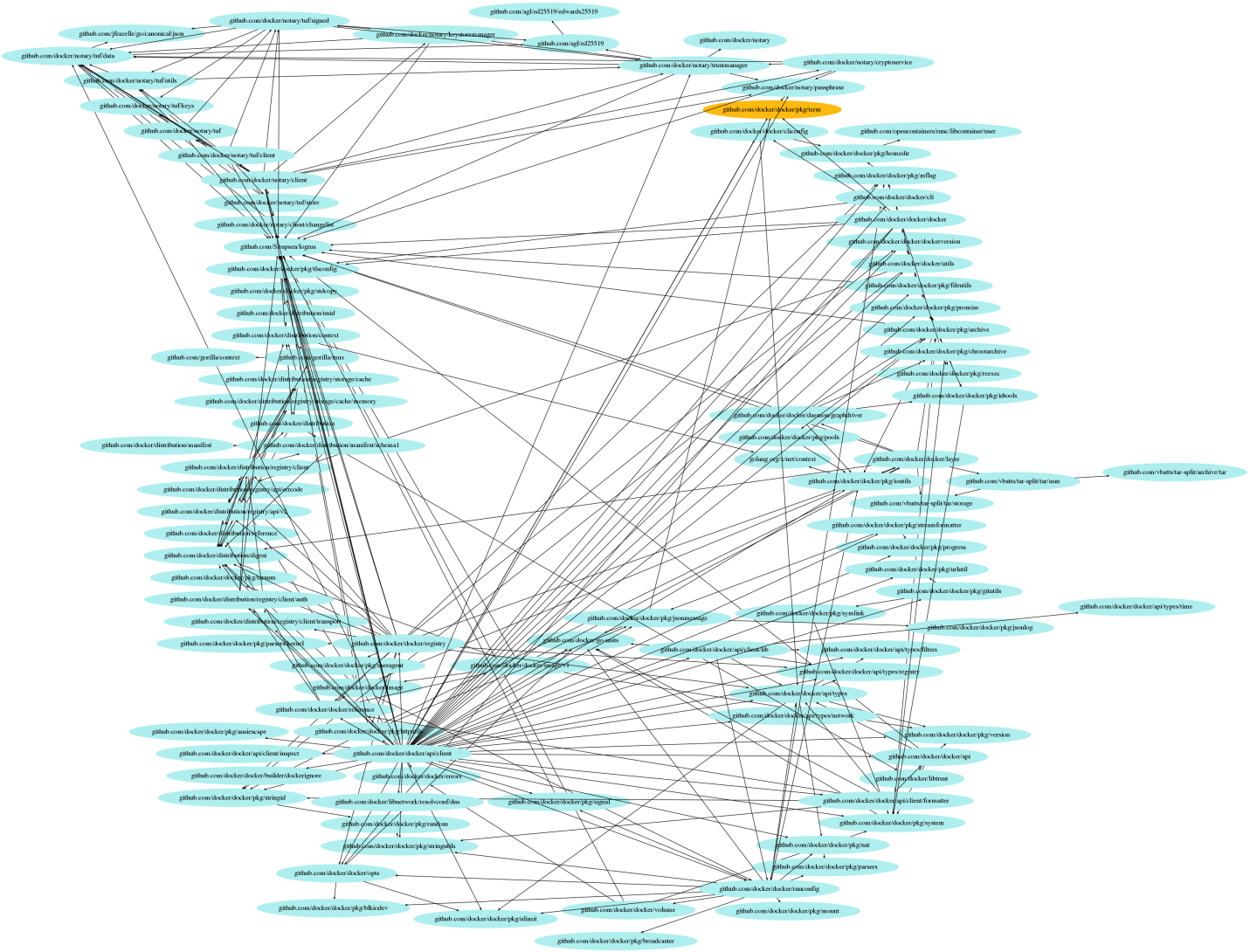


Figure 4.3: Dependency graph docker circular

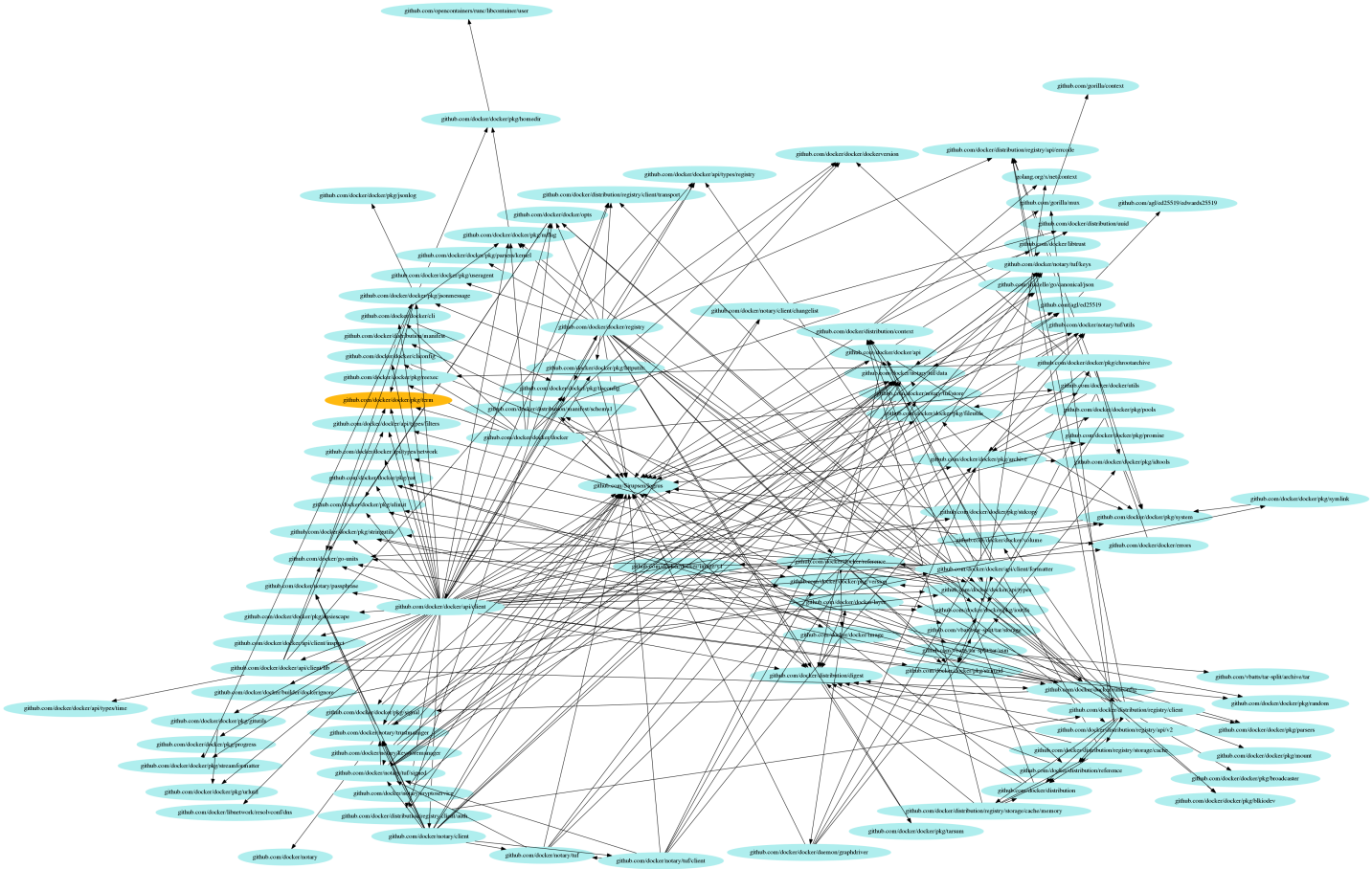


Figure 4.4: Dependency graph docker

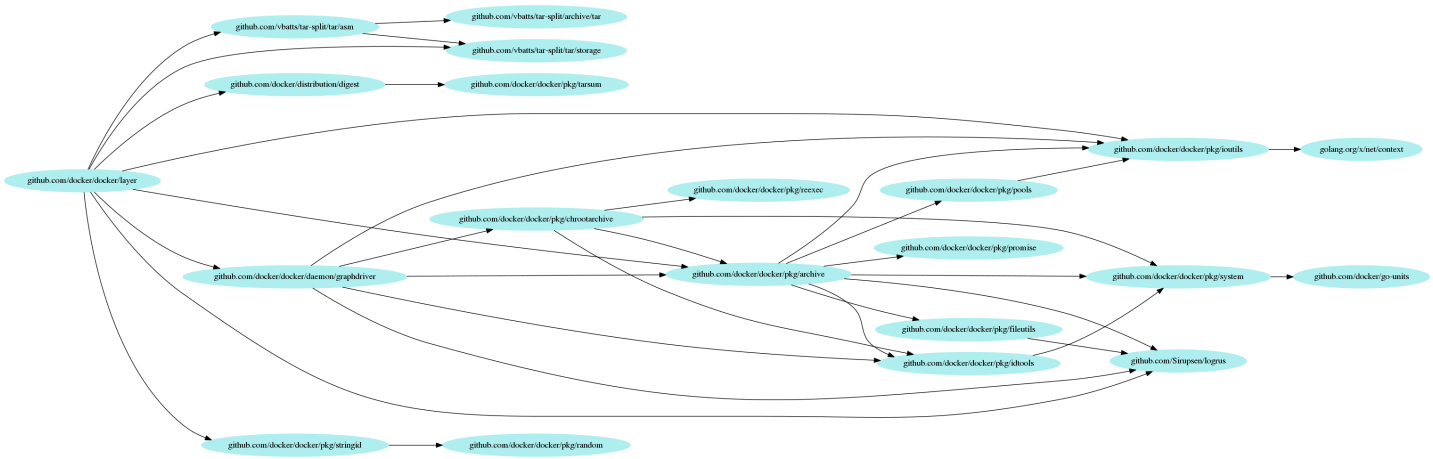


Figure 4.5: Dependency graph layers

4.2 Process View

5 Pattern Documentation

5.1 Core

5.1.1 Layers

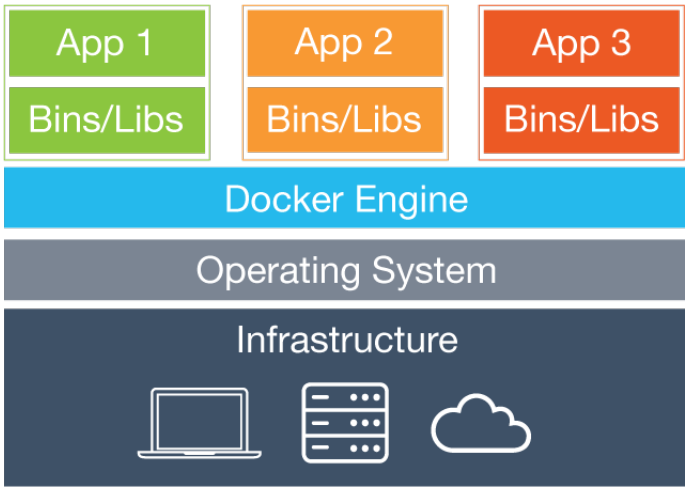


Figure 5.1: A layered overview of the Docker. Source [5]

Traceability
The Layers pattern can be deducted from the online documentation[4].

Source
Architectural patterns revisited – a pattern language, P. 29 [1]

Issue
For a good encapsulation, each instance of docker is modeled using layers that interact with each other.

Assumptions/Constraints
The interaction or communication between layers is carried out through socket, RESTful, or TCP/IP connection.

Solution
Docker utilizes layers pattern. The system is modeled using layers and each component resides on a specific layer. An example can be seen in Figure 5.1.

Rationale
By separating each component in separated layers, the system will be more modular and have better security.

Implications
There must be a clear closure of each layer. Communications protocol must also be defined.

Related Patterns

- Client-server

5.1.2 Client-Server

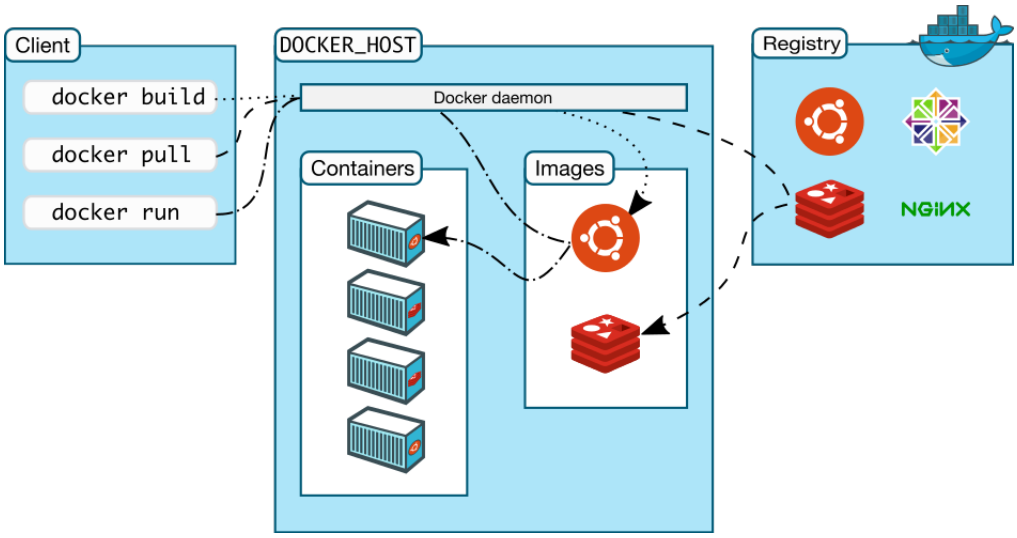


Figure 5.2: An overview of the Docker architecture, showing the client and daemon. Source: [4].

Traceability
The Client-Server pattern can be deducted from the online documentation[4].

Source
Architectural patterns revisited – a pattern language, P. 29 [1]

Issue
For good interoperability, it should be possible for Docker containers to be started remotely. A single interface should be able to control containers on numerous hosts.

Assumptions/Constraints

Solution
Docker uses a Client-Server architecture. The client, a binary supplying a command-line interface, act as the primary interface for the user. The user enters commands into this client, which are then send to a server: the docker daemon.

Rationale
The daemon is a background process, which supplies the requested services to the client. The daemon exposes a REST interface.

For Docker, the client can be configured to connect to other daemon processes than the one running on the local machine. It can be configured to connect to remote Docker daemons as well, allowing the user to issue commands to daemons running remotely.

Implications
The use of the Client-Server pattern results in two different executable binaries: a daemon and a client.

The use of a separated client handling the user interaction increases the modularity. Furthermore, it increases the interoperability, since the client can send commands to daemons running on remote machines and the local machine.

All the services offered by the daemon have to be made available to the client using a REST interface.

Related Patterns

5.1.3 Shared repository

Can we consider the docker registry a shared repository?

5.1.4 Plugin

Traceability
The existence of Docker Plugins becomes apparent from it's documentation at [3].

Additionally, the directories `docker/pkg/plugins/` [1] and `docker/daemon/graphdriver/plugin.go` [2] (among others) in the project's repository contain the code for discovering plugins and the interfaces the plugins should implement.

Source
Patterns of Enterprise Application Architecture, P. 499 [6]

Issue
The users of Docker want to have customization, by extending Docker with third party custom-built tools. This customization means that third parties should be able to write plugins that extend Docker's core functionality.[9]

The implementation of such plugins is only available at runtime.

Assumptions/Constraints

Solution
Docker uses the Plugin pattern to link the implementation of the interfaces of several extendable components with third-party implementation at runtime.

Rationale
Still have to figure out how exactly it is implemented.

Implications
The use of the Plugins pattern means that the adaptability increases.

Related Patterns

5.2 Modules

5.2.1 Interceptor

5.2.2 Plugin

[1] `https://github.com/docker/docker/tree/master/pkg/plugins`
[2] `https://github.com/docker/docker/blob/master/daemon/graphdriver/plugin.go`

6 Evaluation

6.1 Patterns

6.1.1 Pipes and Filters

6.1.2 Shared Repository

6.2 Subsystems

6.2.1 Core Subsystem

6.2.2 Modules Subsystem

7 Recommendations

8 Conclusion

8.1 Pattern-Based Architecture Reviews

8.2 IDAPO

8.3 Final Words

A Time Tracking

Week 1

Person	Task	Hours
Putra	Coaching session, researching about Docker, working on first draft of introduction	3.5
Fakambi	Coaching session, Research about Docker, Work on the system context	3.5
Schaefers	Coaching session, setting up initial layout and new git repository, defining key drivers, research.	6.5
Menninga	Coaching session, stakeholders and concerns	4.0

Week 2

Person	Task	Hours
Putra	Catching up after winter break, reading Docker docs, revising intro, and writing layers pattern documentation	6
Fakambi	Coaching session, Meeting for discussing the patterns	3
Schaefers	Creating scripts to find patterns, building and running docker, creating various dependency graphs, changing the key driver section, adding stakeholders and concern image	8.5
Menninga	Searching patterns, improving stakeholders, client-server and plugin patterns. Improvements to System context.	7.5

Bibliography

[1] Paris Avgeriou and Uwe Zdun. Architectural patterns revisited – a pattern language. 2005.

[2] Docker. Docker - build,ship, and run any app, anywhere. `https://www.docker.com/`, 2015. [Online; accessed 13-December-2015].

[3] Docker. Docker - understand docker plugins. `https://docs.docker.com/engine/extend/plugins/`, 2015. [Online; accessed 3rd January 2015].

[4] Docker. Docker - understanding the architecture. `https://docs.docker.com/engine/introduction/understanding-docker/`, 2015. [Online; accessed 30th December 2015].

[5] Docker. What is docker? `https://www.docker.com/what-docker`, 2016. [Online; accessed 3-January-2016].

[6] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[7] GitHub. Docker - the open-source application container engine. `https://github.com/docker/docker`, 2015. [Online; accessed 13-April-2015].

[8] NeilB. Harrison and Paris Avgeriou. Using pattern-based architecture reviews to detect quality attribute issues - an exploratory study. In James Noble, Ralph Johnson, Uwe Zdun, and Eugene Wallingford, editors, *Transactions on Pattern Languages of Programming III*, volume 7840 of *Lecture Notes in Computer Science*, pages 168–194. Springer Berlin Heidelberg, 2013.

[9] Adam Herzog. Extending docker with plugins. `https://blog.docker.com/2015/06/extending-docker-with-plugins/`, 2015. [Online; accessed 3rd January 2015].

[10] ISO. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. `http://iso25000.com/index.php/en/iso-25000-standards/iso-25010`, 2011.

[11] W.G. Menninga. Benchmarking Tool for the Cloud. Bachelor Thesis, University of Groningen, 2015.

[12] Maureen OĠara. Ben golub, who sold gluster to red hat, now running dotcloud. `http://maureenogara.sys-con.com/node/2747331`, 2013. [Online; accessed 13-December-2015].

[13] Klaas-Jan Stol, Paris Avgeriou, and Muhammad Ali Babar. Design and evaluation of a process for identifying architecture patterns in open source software. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 147–163. Springer Berlin Heidelberg, 2011.

[14] John Paul Walters, Vipin Chaudhary, Minsuk Cha, Salvatore Guercio Jr, and Steve Gallo. A comparison of virtualization technologies for hpc. In *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*, pages 861–868. IEEE, 2008.