Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Bachelorarbeit

im Bachelor-Studiengang Computer Science

Virtualisierung von Testumgebungen zur parallelen Testausführung

von

Per Bernhardt

Betreuer:	Prof. Dr. Rudolf Berrendorf
Zweitbetreuer:	Prof. Dr. Andreas P. Priesnitz
Eingereicht am:	19. April 2015

Erklärung

Per Bernhardt
Schumannstr. 115
53113 Bonn

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

.....	.....
Ort, Datum	Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CD Continous Delivery
CI Continous Integration
CPU Central Processing Unit
DCUI Direct Console User Interface
HTTP Hypertext Transfer Protocol
IDE Integrated Development Environment
IP Internet Protocol
ISO Internationale Organisation für Normung
KVM Kernel-based Virtual Machine
OS Operating System
OVF Open Virtualization Format
PHP PHP Hypertext Preprocessor
PID Prozess-Identifikatoren
SAN Storage Area Network
SOA Serviceorientierte Architektur
SSD Solid State Drive
URL Uniform Resource Locator
VM Virtuelle Maschine
VMM Virtual Machine Monitor

1 Einleitung

Diese Bachelorarbeit entsteht in Zusammenarbeit mit der Pixelhouse GmbH. Diese betreibt seit ca. 10 Jahren die Webseite Chefkoch.de, Europas größtes Kochportal. Neben der Vermarktung der Webseite und der Betreuung der Community wird von den Mitarbeitern vor allem auch die softwaretechnische Entwicklung der Plattform und das Hosting durchgeführt (Vgl. Pixelhouse GmbH 2014).

Um besser auf zukünftige Herausforderungen reagieren zu können, hat die Pixelhouse GmbH im letzten Jahr mit der Implementierung einer neuen IT Strategie begonnen. Teil dieser Strategie ist das Einführen von Scrum als agile Softwareentwicklungsmethode. „Im Kern einer jeden agilen Methode stehen Iterationen, die Zyklen fixer Dauer sind und zum Ziel haben, Feedback zu liefern. In iterativen Vorgehensweisen werden sämtliche Aktivitäten, die wir aus klassischen Vorgehensmodellen kennen, wie Analyse, Design, Implementierung, Testen, Integrieren, Systemtest etc., durchgeführt, jedoch mehrmals.“ (S. Wintersteiger 2013, S. 18)

Zur Optimierung dieser Feedbackschleifen setzt die Pixelhouse GmbH wie auch andere Firmen auf die von Paul M. Duvall in seinem Buch Continous Integration (CI) beschriebene Methode der kontinuierlichen Integration und die dazugehörigen Werkzeuge (Vgl. Duvall et al. 2007, S. 12): So wird jede Änderung an der Software in ein Versionskontrollsystem eingespielt, welches daraufhin einen automatischen Prozess startet, der im Rahmen eines sogenannte Build-Scripts unter anderem automatische Softwaretests durchführt und die entsprechenden Ergebnisse per E-Mail an die Entwickler zurückliefert.

Beim Ausbau dieses Prozesses ist die Pixelhouse GmbH nun auf einige Probleme gestoßen, die im Rahmen dieser Bachelorarbeit allgemein diskutiert und einer Lösung zugeführt werden sollen.

1.1 Problemstellung

Das größte Problem mit dem zuvor beschriebenen Prozess ist aktuell, dass die automatischen Softwaretests nicht skalieren. So werden unter anderem Tests durchgeführt, die mit Hilfe einer programmatischen Schnittstelle normale Webbrowser fernsteuern und so die Nutzung der Webseite durch echte Nutzer simulieren. Diese Art von Test sind sehr langsam und die aktuell existierenden Tests benötigen bereits mehrere Stunden zur Durchführung. Dies bedeutet, dass man nach fertiger Implementierung einer neuen Funktion oder der Modernisierung einer bestehenden Komponente sehr lange auf das Testergebnis warten muss und so Zeit vergeht, bis sich die Änderung sicher in den Hauptentzwicklungszweig integrieren oder in Produktion nehmen lässt. Dies stört den Wunsch nach häufigen Iterationen und schnellem Feedback.

Eine mögliche Lösung für dieses Problem wird von Jez Humble und David Farley in ihrem Buch Continous Delivery (CD) beschrieben: „Assuming that your tests are all independent [..], you can run them in parallel [..]. Ultimately, the performance of your tests is only limited by the time it takes for your single slowest test case to run and the size of your hardware budget.“ (S. Humble und Farley 2010, S. 310) Beim Versuch, eine entsprechende Parallelisierung der Tests zu erreichen, scheitert die Pixelhouse GmbH bislang daran, entsprechend isolierte Testumgebungen vorzuhalten. So laufen die Tests bisher eher in der selben Umgebung, zum Beispiel auf dem gleichen Datenbankserver und hinter dem gleichen HTTP-Cache (Hypertext Transfer Protocol-Cache). Dadurch kann es vorkommen, dass ein Test die Daten oder das Caching eines anderen Tests beinflusst und ihn somit fehlschlagen lässt. Das Aufsetzen von Umgebungen ist bislang ein manueller Prozess. „It is extremely difficult to precisley reproduce manually configured environments for testing purposes.“ (S. Humble und Farley 2010, S. 49)

Als mögliche Lösung für dieses Problem empfehlen Jez Humble und David Farley den Einsatz von Virtualisierungslösungen. „The use of virtual servers to baseline host environments makes it simple to create copies of production environments, even where a production environment consists of serveral servers, and to reproduce them for testing purposes.“ (S. Humble und Farley 2010, S. 304)

1.2 Ziel

Ziel dieser Bachelorarbeit ist es zu untersuchen, ob und wie man mit Hilfe von Virtualisierungstechniken eine Lösung für das zuvor beschriebene Problem langsamer Feedbackschleifen bei der Ausführung automatischer Softwaretests schaffen kann. Mit dieser Lösung soll nicht nur die Anwendung sondern auch deren Infrastruktur in einem konkreten Zustand nachgehalten und effizient aufgesetzt werden können. Dadurch soll es möglich sein, einfach und beliebig oft Testumgebungen anbieten zu können, um die Ausführung der Tests parallelliesieren zu können und so die Gesamtlaufzeit der Tests zu reduzieren. Neben der konzeptionellen Arbeit sollen auch prototypische Umsetzungen erfolgen und diese anschließend bewertet werden.

1.3 Vorgehensweise

Diese Arbeit gibt im zweiten Kapitel „Grundlagen und Ansätze“ zunächst einen theoretischen Überblick über die für die Problemstellung relevanten Themenbereiche Softwaretests und Virtualisierung. Im darauf folgenden Kapitel „Konzeption“ soll dann aufgrund dieser theoretischen Ansätze erarbeitet werden, wie eine mögliche Lösung für die vorliegende Problemstellung aussieht. Im anschließenden Kapitel „Implementierung“ werden die Ergebnisse verschiedener prototypischer Umsetzungen - vor allem deren Probleme und Besonderheiten - beschrieben. Im vorletzten Kapitel „Evaluation“ soll eine objektive Bewertung der verschiedenen Umsetzungen anhand bestimmter Kriterien erfolgen. Ein wichtiges Kriterium ist auf jeden Fall die Reduzierung der Gesamtausführungsdauer der Tests, aber zum Beispiel auch die einfache Verwendbarkeit der Lösung. Andere Kriterien wie zum Beispiel monetäre Kosten oder die Virtualisierbarkeit anderer Umgebungen (Windows, MacOS) fallen hingegen weniger ins Gewicht. Das letzte Kapitel „Zusammenfassung und Ausblick“ fasst die Ergebnisse der Arbeit zusammen und gibt einen kurzen Ausblick auf mögliche weitere Schritte.

2 Grundlagen und Ansätze

Im Folgenden sollen nun die theoretischen Grundlagen und Ansätze erarbeitet werden, die im Umfeld der Problemstellung anzusiedeln sind.

Dazu gehört zum einen der Bereich der Softwaretests. Hierbei soll beleuchtet werden, welche Arten von Tests es gibt und inwieweit diese definierte Testumgebungen benötigen. Außerdem sollen verschiedene Strategien der Testausführung besprochen werden und diskutiert werden, wie diese durch Virtualisierungslösungen unterstützt werden können.

Zum anderen werden die Grundlagen und Ansätze zur Virtualisierung beleuchtet. Hierbei wird vor allem auf die Virtualisierung von Betriebs- bzw. Testumgebungen eingegangen.

2.1 Softwaretests

Das Testen von Software ist allgemein eine „Überprüfung des Ein-/Augabeverhaltens eines Programms anhand von Experimenten und gezielten Programmdurchläufen“ (S. Claus und Schwill 2001, S.662). Es handelt sich also um eine Qualitätssicherungsmaßnahme. Jenachdem welcher Teil eines Programms überprüft wird, unterscheidet man verschiedene Testarten. Außerdem gibt es verschiedene Strategien, Tests auszuführen.

2.1.1 Testarten

Man unterscheidet grob vier Arten von Tests: Unit-, Integrations-, System- und Aktzeptanztests (Vgl. Duvall et al. 2007, S. 129 ff).

2.1.1.1 Unit-Tests

„Unit-Tests verify the behaviour of small elements in a software system, which are most often a single class.“ (S. Duvall et al. 2007, S. 132) Eine wichtige Eigenschaft von Unit-Tests ist dabei, dass sie keine externen Abhängigkeiten wie zum Beispiel Datenbank oder Dateisystem besitzen. Tests können damit sehr früh und schnell geschrieben werden und benötigen nur wenig Zeit zur Ausführung. (Vgl. Duvall et al. 2007, S. 133) Unit-Tests benötigen deshalb aber auch keine anspruchsvolle Testumgebung oder eine Optimierung der Ausführungszeit. Für die Problemstellung dieser Arbeit sind sie also weniger interessant.

2.1.1.2 Integrations-Tests

„Integration tests collect modules together and test them as a subsystem in order to verify that they collaborate as intended to achieve some larger piece of behaviour.“ (S. Clemson 2014, S. 10) Integrations-Tests (beziehungsweise Komponenten-Tests oder Subsystem-Tests) können dabei auch Zugriff auf eine Datenbank oder das Dateissystem benötigen (Vgl. Duvall et al. 2007, S. 133). „The difference between this type of test and a system test is that integration tests (or component tests or subsystem tests) don’t always exercise a publicly preferable API.“ (S. Duvall et al. 2007, S. 136) Integrations-Tests sind aber in jedem Fall Tests, die potentiell Zugriff auf eine Testumgebung benötigen. Da ein Integrations-Tests aber nicht zwingend auf die gesamte Umgebung zugreift, sondern zum Beispiel nur auf die Datenbank aber eben nicht auf einen HTTP-Cache, wäre es für eine mögliche Implementierung einer solchen Testumgebung von Vorteil, wenn diese Umgebung auch nur teilweise gestartet werden könnte.

2.1.1.3 System-Tests

„System tests excercise a complete software and therefore require a fully installed system [..].“ (S. Duvall et al. 2007, S. 136) Da Systemtests per Definition mehr Komponenten eines Systems im Zusammenspiel testen, als dies bei Integrations- beziehungsweise Unit-Tests der Fall ist, brauchen Systemtests im Vergleich sowohl länger für das Starten der Testumgebung als auch für das Durchlaufen des zu testenden Programmcodes. (Vgl. Duvall

et al. 2007, S. 136) System-Tests sind somit sowohl hinsichtlich der von ihnen vollständig benötigten Testumgebung als auch der potentiell optimierbaren Ausführungsdauer genau die Art von Tests, die im Rahmen der Problemstellung relevant sind.

2.1.1.4 Aktzeptanz-Tests

Akzeptanz- bzw. Funktionale Tests unterscheiden sich von System-Tests insofern, als das bei ihnen die Betonung darauf liegt, das System aus Sicht des Benutzers zu testen (Vgl. Duvall et al. 2007, S. 136). Das Fernsteuern eines Browsers zum Testen einer Webanwendung ist ein typisches Beispiel für einen Akzeptanz-Test. Aber auch das manuelle Testen der Anwendung durch einen Projekt- oder Produktmanager gilt als Akzeptanztest. Genau wie der System-Tests benötigt der Aktzeptanz-Test auf jeden Fall das gesamte System und somit auch eine vollständige Testumgebung. Auch die Dauer der Testausführung ist wie bei Systemtests höher als die von Unit- und Integrations-Tests.

2.1.2 Teststrategie

Neben den unterschiedlichen Testarten gibt es auch unterschiedliche Strategien zur Testausführung.

Wie viele andere Firmen auch, hat die Pixelhouse GmbH bei der Entwicklung ihrer Softwarekomponenten lange Zeit lediglich manuelle Tests ausgeführt. „Too many projects rely solely on manual acceptance testing to verify that a piece of software conforms to its functional and nonfunctional requirements.“ (S. Humble und Farley 2010, S. 83) Dabei wurden Unit-, Integrations und Systemtests, soweit solche überhaupt vorhanden sind, lediglich sporadisch und im Rahmen der Entwicklung auf Entwickler-Laptops ausgeführt. Anschließend wurden auf einer Testumgebung durch einen Produktmanager oder Endkunden manuelle Akzeptanztests durchgeführt, die sicherstellen sollten, dass die Software den vor der Entwicklung von ihnen definierten Anforderungen entspricht. Nachteil dieser Herangehensweise ist, dass das Testen der Anwendung nicht gut skaliert. Je mehr Neuerungen durch die Entwicklung entstehen, umso mehr wird der Produktmanager zu einem Engpass bei der manuellen Abnahme dieser Neuerungen. Kommt es beim Testen der Software zu einem Fehler, ist es dem Produktmanager oder Endkunden meist auch nicht möglich, die Fehlerursache zu benennen. Vielmehr geben diese eine Anleitung an die Entwicklung zurück, wie der entsprechende Fehler reproduziert werden kann. Es folgt eine entsprechend aufwendige Fehleranalyse durch einen Entwickler. Ein Vorteil dieser Herangehensweise ist, dass weniger Aufwand für das Bereitstellen von Testumgebungen entsteht. So werden Tests entweder in der Entwicklungsumgebung durch den Entwickler ausgeführt oder eben auf einer geringen Anzahl an festen Testumgebungen, die den Produktmanagern oder Endkunden zur Verfügung steht. Es gibt außerdem auch Fehlern, die sich nur sehr schwer oder auch gar nicht über automatische Tests finden lassen, zum Beispiel optische Fehler im Userinterface einer Anwendung (Vgl. Duvall et al. 2007, S. 197). Manuelle Tests haben insofern auf jeden Fall eine Daseinsberechtigung. Insgesamt kann man aber sagen, dass die rein manuelle Ausführung von Tests zwar eine mögliche Testausführungs-Strategie ist, aber nicht die, die nach heutigem Stand der Wissenschaft empfehlenswert ist.

Idealerweise werden in heutigen Softwareprojekten von Beginn an automatische Softwaretests geschrieben. „In our ideal project, testers collaborate with developers and users to write automated tests from the start of the project.“ (S. Humble und Farley 2010, S. 83) Diese werden sogar vor der eigentlichen Implementierung geschrieben und verifizieren dann die Vollständigkeit und Korrektheit der späteren Implementierung. Als Testsausführungsstrategie empfiehlt es sich dabei, dass sämtliche Tests automatisch von einem CI-Server ausgeführt werden können, sobald sich etwas an der Anwendung ändert. (Vgl. Humble und Farley 2010, S. 83) „[..] Tests must run any time and every time something in the system changes.“ (S. Duvall et al. 2007, S. 131) Bei dieser Herangehensweise ist es

wichtig, sich zu überlegen, welche Arten von Tests man wie häufig verwendet. Wie zuvor beschrieben unterscheiden sich die unterschiedlichen Testarten in Bezug auf den Aufwand für ihre Erstellung, die Dauer ihrer Ausführung und ihre Ansprüche an eine gegebenenfalls notwendige Testumgebung. „Writing and running tests is obviously a good thing, but unless we treat them as an architectural component that requires proper categorization and structure, they can start looking like a hurdle, instead of the key, to success.“ (S. Duvall et al. 2007, S. 138) Durch die sinnvolle Gruppierung von Tests kann ein Entwicklungsteam eine sinnvolle Ausführungsreihenfolge definieren. So sollten schnell laufende Unit-Tests vor den länger laufenden Integrations- und System-Tests ausgeführt werden. Dadurch kann ein Fehler auf dieser Ebene sehr früh entdeckt und einem konkreten Stück Software zugeordnet und an die Entwicklung zurück gegeben werden. Ist auf der Unit-Ebene aber kein Fehler aufgetreten, so können die teureren und länger laufenden Integrations- und Systemtests durchgeführt werden (Vgl. Duvall et al. 2007, S. 138). „All levels are essential to ensure that the application is working and delivering the expected business value.“ (S. Humble und Farley 2010, S. 178) Auch wenn alle Ebenen wichtig sind, so wird allgemein empfohlen, mehr Unit-Tests als Integrations-, Systemtests oder Aktzeptanztests zu schreiben. Dies wird gerne über die in der Abbildung 1 zu sehende Testpyramide dargestellt.

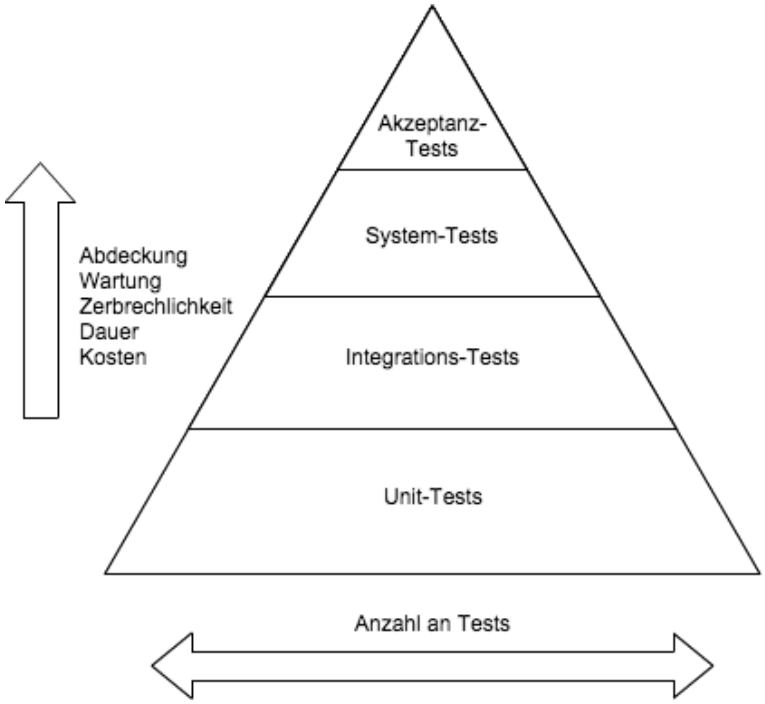

Abbildung 1: Testpyramide (Vgl. Crisp 2011)

2.2 Virtualisierung von Testumgebungen

Laut Duden leitet sich das Adjektiv „virtuell“ vom lateinischen „virtus“ für „Tüchtigkeit, Mannhaftigkeit, Tugend“ ab. Es bedeutet so viel wie „nicht echt, nicht in Wirklichkeit vorhanden, scheinbar“ (Vgl. Duden 2015).

Innerhalb der Informatik spricht man immer dann von Virtualität oder Virtualisierung, wenn einem Benutzer oder einem Softwareprozess eine Umgebung vorgespielt wird, die physisch nicht wirklich existiert.

Der für diese Arbeit interessante Bereich der Virtualisierung ist das Bereitstellen virtueller Betriebsumgebungen. Eine Betriebsumgebung meint die Umgebung, die ein Programm für seine Ausführung benötigt. Dazu gehören vor allem bestimmte Hardware-Bauteile (Prozes-

soren, Arbeitsspeicher, Festplatten, Netzwerk-Adapter), ein Betriebssystem und eventuell zusätzliche Anwendungsprogramme, mit denen das Programm interagiert. Für gewöhnlich läuft eine solche Betriebsumgebung genau auf einem Rechner (zum Beispiel Server oder Desktoprechner). Mit Hilfe der Virtualisierung ist es nun aber möglich, mehrere solcher Betriebsumgebungen auf dem gleichen Rechner laufen zu lassen. Dabei wird der Betriebsumgebung eben nur vorgespielt, auf einem eigenen Rechner zu laufen (Vgl. Damodaran et al. 2012, Abstract).

Es gibt verschiedene Ziele, die man mit der Virtualisierung von Betriebsumgebungen verfolgen kann. Zum einen lässt sich eine effizientere Nutzung der Ressourcen erreichen. „Virtualization has other benefits, such as the ability to consolidate hardware and to standardize your hardware platform even if your applications require heterogeneous environments.“ (S. Humble und Farley 2010, S. 53) Da die tatsächliche Hardware abstrahiert wird, kann diese durch das hinzufügen weiterer Betriebsumgebungen vollständig ausgenutzt werden. Eine Firma könnte also zum Beispiel für jeden Mitarbeiter eine Arbeitsumgebung auf einem zentralen Server starten. Da diese nur Ressourcen verbraucht, wenn der Mitarbeiter tatsächlich arbeitet, kommt man mit weniger Hardwareressourcen aus, als wenn man jedem Mitarbeiter einen echten eigenen Rechner zur Verfügung stellt. Der Betrieb und die Wartung von virtuellen Arbeitsumgebungen lässt sich dabei auch besser planen und einfacher durchführen. Außerdem ist in den letzten Jahren das Mieten solcher zentralen Hardwareressourcen populär geworden. Man spricht hierbei auch gerne von sogenannten Cloud-Anbietern oder Cloud-Lösungen. „The defining characteristic of cloud computing is that the computing resources you use, such as CPU [Anm. d. Verf.: Central Processing Unit], memory, storage, and so on, can expand and contract to meet your need, an you pay only for what you use.“ (S. Humble und Farley 2010, S. 312) In diesem Falle würde die Hardware erst gar nicht eingekauft und selbst betrieben oder gewartet werden. Dies würde dann in den Aufgabenbereich des Dienstleisters fallen, der (oft minutengenau abgerechnet) nur die tatsächliche Nutzung der Ressourcen in Rechnung stellt (Vgl. Zhang et al. 2010, S. 7).

Ein weiteres Ziel der Virtualisierung von Betriebsumgebungen ist die Trennung beziehungsweise Isolierung verschiedener Anwendungen (Vgl. Scheepers 2014, Abstract). Statt zum Beispiel auf einem Server einen Datenbankserver zu installieren, in dem mehrere Mitarbeiter oder auch Kunden ihre Datenbanken pflegen, kann man mit Hilfe von Virtualisierungsansätzen einfach für jeden Anwender einen eigenen Datenbankserver laufen lassen. Zwar bieten Datenbankserver natürlich auch innerhalb einer Instanz die Möglichkeit, den Zugriff auf einzelne Datenbanken zu begrenzen. Die Konfiguration solcher Zugriffsberechtigungen kann aber komplex sein und ist somit fehleranfällig. Gerade aus Sicherheitsgründen kann es somit lohnenswert sein, Anwendungen bereits auf Betriebsumgebungsebene voneinander zu trennen. Zudem gibt es Anwendungen, die selbst keine differenzierbaren Zugriffskonzepte kennen und somit so oder so für jeden Anwender separat gestartet werden müssen. Die eingangs beschriebenen Probleme bei der Testausführung für die Anwendung der Webseite Chefkoch.de lassen sich, wie später noch einmal im Detail zu sehen sein wird, auf solche mangelnde Isolation zwischen den Anwendungsteilen zurückführen.

Die Virtualisierung von Betriebssumgebungen ist aber auch mit Nachteilen verbunden. So geht durch die Virtualisierung Rechenleistung verloren. Ausgeführte Anwendungen sind also mitunter langsamer, als wenn man sie auf echter Hardware betreibt. Die virtualisierten Betriebsumgebungen sind zudem zwar grundsätzlich von einander isoliert. Da sie aber auf der selben Hardware laufen, ist es je nach Virtualisierungstechnik sehr schwer sicherzustellen, dass sie sich garnicht beeinflussen. Auch im Bereich des Datenschutzes und der Lizensierung von Softwareprodukten kommen durch Virutalisierungslösungen neue Probleme auf. So richten sich viele Lizensmodelle nach Anzahl der physischen Prozessoren und der Größe des physischen Arbeitsspeichers. Beides kann aber bei der Virtualisierung

nicht zwangsläufig genau einer Betriebsumgebung und deren Anwendungen zugeordnet werden.

Es gibt nun verschiedene Ansätze, Betriebsumgebungen zu virtualisieren. Wie in der Abbildung 2 zu sehen, unterscheidet man hinsichtlich ihrer Funktionalität und Implementierung grundsätzlich zwischen der Systemvirtualisierung mittels Hypervisor und der Betriebssystemvirtualisierung mittels OS-Containern (OS steht hierbei für Operating System). Auf diese Varianten soll nun im Näheren eingegangen werden.

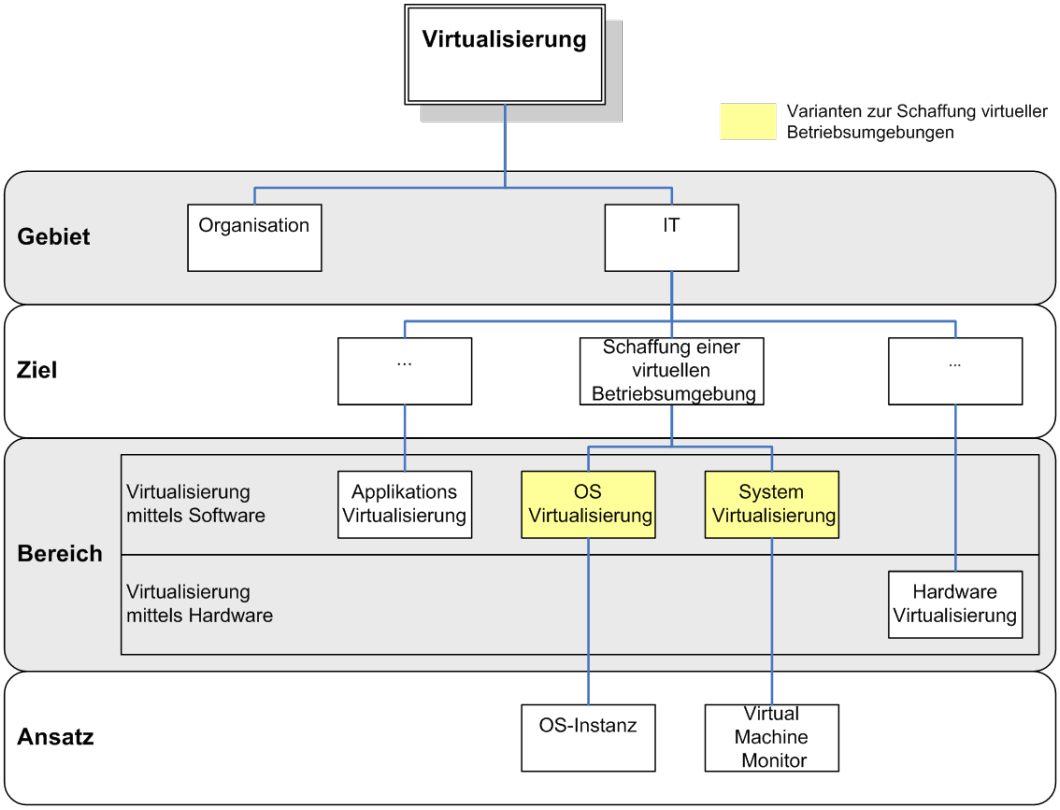

Abbildung 2: Einordnung Virtualisierungstechnologien für virtuelle Betriebsumgebungen (Hirschbach 2006)

2.2.1 Systemvirtualisierung mittels Hypervisor

Als Hypervisor oder auch Virtual Machine Monitor (VMM) wird eine Software bezeichnet, die die physisch vorhandene Hardware abstrahiert und mehreren Gastbetriebssystemen zur Verfügung stellt. Dabei spielt es den Gastsystemen jeweils vor, dass sie auf einem eigenen vollständigen Rechner mit Prozessor, Arbeitsspeicher, Festplatte und sonstigen Geräten laufen. Ein solcher virtueller Rechner wird auch Virtuelle Maschine (VM) genannt (Vgl. Popek und Goldberg 1974, S. 413).

Um die Herausforderung dieses Virtualisierungsansatzes genauer zu verstehen, ist es notwendig, sich die Prozessorarchitektur heutiger Desktop- und Serverrechner anzuschauen. Im Falle der Infrastruktur der Webseite Chefkoch.de ist dies - wie auch in einer Vielzahl anderer Umgebungen - die Familie der X86-Prozessoren (32bit) beziehungsweise deren 64bit Erweiterung, je nach Hersteller AMD64, Intel64 oder auch kurz x64 genannt.

Die folgende Beschreibung dieser Prozessorarchitektur stützt sich auf die Seiten 203-206 des Oracle VM VirtualBox User Manuals (Oracle Corporation 2014).

Prozessorarchitekturen beschreiben vor allem eine bestimmte Menge an Befehlssätzen, die der vom Prozessor ausgeführte Programmcode aufrufen kann, um bestimmte Operatio-

nen auszuführen. Solche Befehlssätze beinhalten zum Beispiel Befehle zum Kopieren von Daten zwischen Prozessor und Hauptspeicher (Transferbefehle) oder auch arithmetische Befehle zur Verrechnung von Werten. Die X86-Architektur unterstützt nun das Konzept sogenannter Ringe, das in der Abbildung 3 zu sehen ist. Ein Ring bezeichnet dabei eine Sicherheitsstufe eines Prozesses, der vom Prozessor verarbeitet wird. Je nach Sicherheitsstufe darf der Prozess dabei auf mehr oder weniger Befehlssätze innerhalb des Prozessors zugreifen. Der Ring 0 bezeichnet dabei die Sicherheitsstufe, innerhalb derer ein Prozess auf alle Befehlssätze des Prozessors zugreifen darf. Man nennt diesen Ring auch „Kernel-Mode“, da in diesem Modus typischerweise nur das Betriebssystem beziehungsweise dessen Kern, der sogenannter Kernel, ausgeführt wird. Prozesse normaler Anwendungsprogramme werden hingegen in Ring 3 ausgeführt, der nur wenige Befehlssätze ausführen darf. Man nennt diesen Ring auch „User-Mode“. Ein Prozess im Ring 3 darf zum Beispiel keine Transferbefehle ausführen. Benötigt er entsprechende Operationen, greift er auf Funktionen des Kernels des Betriebssystems zu, die dabei bestimmte Berechtigungen sicherstellen können. Ein Prozess in Ring 3 kann somit zum Beispiel nicht auf den Arbeitsspeicher anderer Prozesse zugreifen oder sich selbst in einen anderen Ausführungsmodus bringen. Würde ein Prozess in Ring 3 auf einen nicht erlaubten Befehl zugreifen, so löst der Prozessor eine Exception aus, die vom Kernel beziehungsweise vom Programm-Code in Ring 0 aufgelöst werden muss, indem er zum Beispiel einen alternativer Befehl zur Ausführung bringt oder den betroffenen Prozess zur Not beendet.

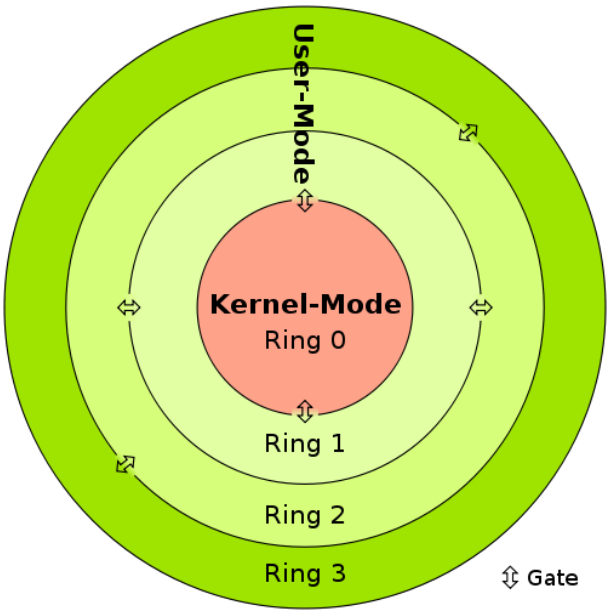

Abbildung 3: CPU Ring scheme (Sven 2014)

Um nun mit Hilfe der Virtualisierung mehrere Betriebssysteme nebeneinander auszuführen, die sich gegenseitig nicht beeinflussen können, ist es notwendig, deren Kernel nicht im Ring 0, sondern in einem höheren Ring mit weniger Berechtigung auszuführen. Da der Programm-Code des Gast-Kernels dabei natürlich dennoch Befehle rufen würde, die nur in Ring 0 ausführbar sind, kommt es zu den eben beschriebenen Exceptions. Die Lösung dieser Ausnahmesituationen ist dabei nun Aufgabe des Hypervisors. Dazu beinhaltet ein Hypervisor immer mindestens einen Prozess, der selbst in Ring 0 läuft und in der Lage ist, die Auflösung der Ausnahmesituationen zu erledigen. Ein Hypervisor kann dabei nun verschiedene Strategien anwenden. Die einfachste Strategie ist eben, auf entsprechende Exceptions zu warten und diese zum Beispiel durch Aufrufe auf das eventuell vorhandene Host-Betriebssystem oder den Ring-0-Prozess des Hypervisors zu ersetzen. Auch wenn

diese Strategie funktioniert, so ist die Behandlung tausender solcher Ausnahmesituationen sehr teuer und beinträchtigt die Leistung des Gastsystems empfindlich. Moderne Hypervisoren verfolgen deshalb eine zwar sehr viel kompliziertere aber eben auch effizientere Strategie. Sie scannen den Programm-Code des Gastsystems nach problematischen Befehlen und ersetzen diese bereits vor der Ausführung im Speicher mit Befehlen, die in der Berechtigungsstufe des Gastsystems erlaubt sind. Heutige Prozessoren kennen mitunter sogar das Konzept der Virtualisierung mittels Hypervisor und bieten spezielle Befehlssätze an, die sich besonders für diese Ersetzung eignen und sich aus eingeschränkten Berechtigungsstufen rufen lassen. Moderne Hypervisoren und Betriebssysteme kommen damit auf Ausführungszeiten, die nahezu denen auf direkter physischer Hardware entsprechen.

Robert P. Goldberg, der die Grundlagen des VMM in den 70er Jahren wissenschaftlich erarbeitet hat, unterscheidet in seiner Doktorarbeit „Architectural Principles for Virtual Computer Systems“ nun zwei Klassen des VMM (Vgl. Goldberg 1973, S. 22 ff.): Dem Typ I VMM oder auch „Bare Metal Hypervisor“ und dem Typ II VMM oder auch „Hosted Hypervisor“. In den folgenden beiden Unterkapiteln sollen diese beiden Klassen und entsprechende Implementierungen näher beleuchtet werden.

Darauf folgt abschließend ein Unterkapitel zu den sogenannten Unikernels (Vgl. Madhavapeddy et al. 2013, Abstract und Introduction), einer relativ neuen Entwicklung innerhalb der Systemvirtualisierung mittels Hypervisor, die eine bestimmte Art von VM beschreibt.

2.2.1.1 Bare Metal Hypervisors

Ein Bare Metal Hypervisor ist ein VMM, der als eigenständiges Programm direkt auf echter physischer Hardware läuft und somit kein Betriebssystem auf dem Hostsystem benötigt. Die Abbildung 4 stellt diese Situation dar.

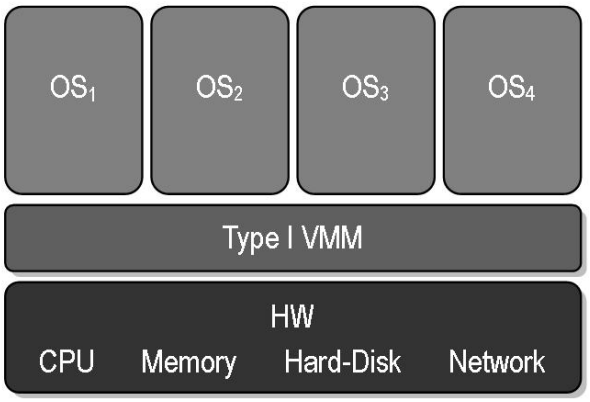

Abbildung 4: Virtual Machine Monitor Type I (Chen 2014a)

Durch diesen direkten Zugriff und das nicht notwendige Hostbetriebssystem gilt ein Bare Metal Hypervisor als ressourceneffizienter als ein Hosted Hypervisor. Größter Nachteil dieser Variante ist aber, dass sie mit mehr Installationsaufwand verbunden ist, da der Hypervisor selbst die passenden Gerätetreiber für die zugrundeliegende Hardware benötigt und man nicht auf Standardwerkzeuge wie zum Beispiel den Installationsmanagers eines Hostbetriebssystems zugreifen kann. Typische Vertreter dieser Hypervisor-Klasse sind zum Beispiel VMWare vSphere (Vgl. Trefis 2014), KVM und Citrix XenServer (Vgl. Citrix Systems, Inc. 2015b).

Bei VMWare vSphere handelt es sich um ein kommerzielles Produkt der Firma VMWare, dem Marktführer im Bereich der Virtualisierungslösungen mit einem Marktanteil von ungefähr 50% (Vgl. Trefis 2014). vSphere basiert auf einem von VMWare eigens entwickelten Betriebssystem, dem sogenannten VMkernel. Auf diesem Betriebssystem laufen das Pro-

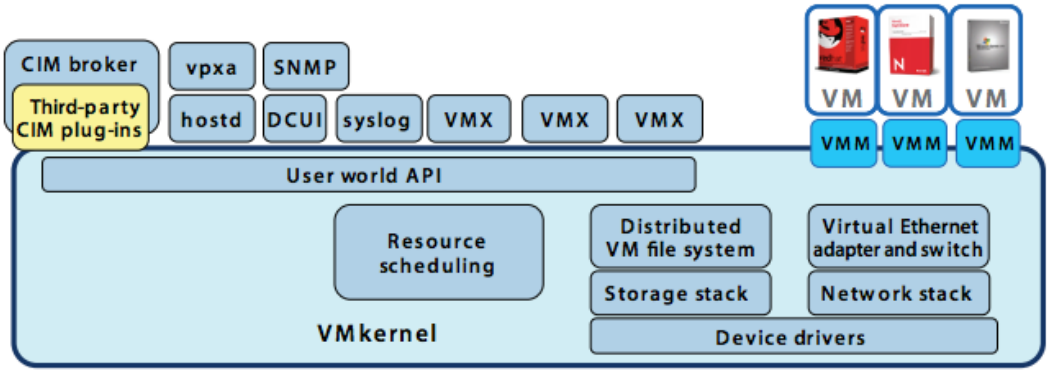

Abbildung 5: VMWare vSphere Architektur (Chaubal 2015)

gramm Direct Console User Interface (DCUI), eine Konsole zur direkten Konfiguration des Servers, und diverse weitere Schnittstellen, mit denen man die Verwaltung des Servers und der auf ihm laufenden virtuellen Maschinen auch über das Netzwerk erledigen kann. Für jede virtuelle Maschine wird ein eigener Hypervisor- und ein weiterer Hilfsprozess gestartet (VMM und VMX) (Vgl. Chaubal 2015, S. 3). Dieser Auffbau ist in der Abbildung 5 zu sehen. vSphere bietet die Möglichkeit, virtuelle Maschinen im laufenden Betrieb von einem vSphere Server auf einen anderen zu übertragen. Diese Funktion, vMotion genannt, erleichtert das Resourcenmanagement enorm, da sich eine laufende Anwendung auf andere Hardware übertragen lässt, um die vorherige Hardware zum Beispiel zu reparieren oder auch einfach nur zur Einsparung auszuschalten (Vgl. Setty 2011, S. 4).

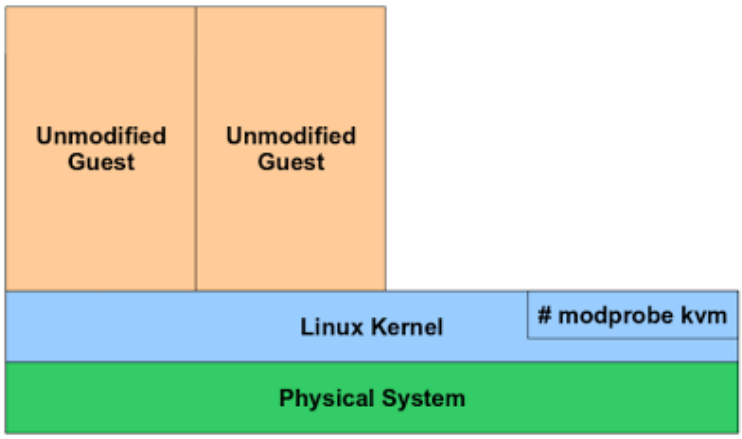

Abbildung 6: KVM Architektur (Chirammal 2014)

Beim Open-Source-Produkt KVM (Kernel-based Virtual Machine) handelt es sich um eine Lösung, bei der ein normaler Linux-Kernel über ein spezielles Kernel-Modul für die grundsätzliche Virtualisierung (kvm.io) und ein weiteres prozessorspezifisches Kernel-Modul (kvm-intel.io oder kvm-amd.io) in die Lage versetzt wird, als Hypervisor zu arbeiten und Gastsysteme zu verwalten (Vgl. Kivity et al. 2007, S. 225 - 227). Dies ist auch in der Abbildung 6 zu sehen. KVM lässt sich damit grundsätzlich auf allen gängigen Linux Distributionen installieren. Man könnte deshalb auch argumentieren, dass es sich nicht um einen reinen Bare Metal Hypervisor handelt, da er eben zusammen mit einem Betriebssystem installiert wird. Da es sich bei dem Kernel-Modul aber nicht um ein Anwendungsprogramm handelt, sondern um eine Erweiterung des eigentlichen Kernels,

setzt diese Virtualisierungslösung direkt auf der darunterliegenden Hardware und nicht auf einem dazwischenliegenden Kernel auf.

So gibt es zum Beispiel die Linux Distribution Proxmox VE (Vgl. Proxmox 2014), die man direkt auf eine leere Hardware installieren kann und die die entsprechenden Kernel-Module und weitere Hilfsprogramme (zum Beispiel zur Verwaltung der virtuellen Maschinen) bereits beinhaltet.

Auch KVM bietet die Möglichkeit, laufende virtuelle Maschinen von einem Host auf einen anderen zu übertragen (Vgl. KVM 2015).

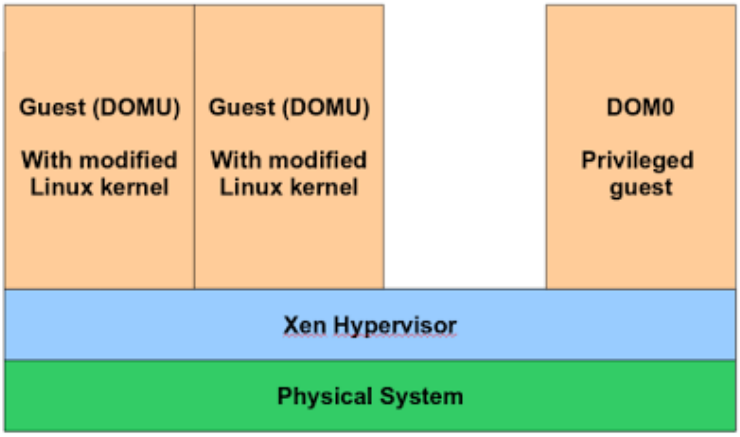

Abbildung 7: Xen Architektur (Chirammal 2014)

Auch bei XenServer von der Firma Citrix handelt es sich um einen Hypervisor (Vgl. Citrix Systems, Inc. 2015b). Auch dieser VMM setzt direkt auf der eigentlichen physischen Hardware auf. XenServer basiert ebenfalls auf einem Linux-Kernel. Es handelt sich aber nicht lediglich um Kernel-Module, die sich in eine beliebige Linux-Distribution laden lassen, sondern um einen modifizerten Linux-Kernel. XenServer lässt sich also nur auf komplett leere Hardware installieren und bietet auch nicht die vollständigen Funktionalitäten einer normalen Linux-Distribution. Das besondere an XenServer ist, dass die Prozesse zur Steuerung der virtuellen Maschinen selbst in einer virtuellen Maschine laufen. Diese auch „Privileged Guest“ genannte Maschine muss laufen, bevor weitere Gastsysteme geladen werden können. Der „Privileged Guest“ oder auch DOM0 genannte Gast hat dabei (wie der Name sagt) besondere Rechte, die ihm die Steuerung der darunterliegenden Virtualisierungsschicht im Hypervisor ermöglichen (Vgl. Scheepers 2014, S. 2). Die entsprechende Architektur ist auch in der Abbildung 7 zu erkennen.

Als weitere Gastsysteme lassen sich wie auch bei KVM beliebige unmodifzierte Gastsysteme wie Linux-Distributionen oder Windows-Systeme installieren. Besonders ist aber, dass sich auch Linux-Distributionen mit ebenfalls modifiziertem Kernel laden lassen, die direkter mit dem Hypervisor zusammen arbeiten und so eine höhere Performance bieten.

Auch XenServer bietet Live-Migrationen, also die Möglichkeit, virtuelle Maschinen von einem Hardware-Rechner auf einen anderen zu übertragen (Vgl. Citrix Systems, Inc. 2015a).

2.2.1.2 Hosted Hypervisors

Ein Hosted Hypervisor ist ein VMM, der als Anwendungsprogramm innerhalb eines Hostbetriebssystems läuft (Vgl. Goldberg 1973, S. 22 ff.). Es ist somit möglich, auch andere Programme neben dem Hypervisor und seinen Gastbetriebssystemen zu verwenden. Die Abbildung 8 macht diesen Auffbau noch einmal besser verständlich.

Größter Vorteil dieser Variante ist es also, dass das Hostsystem auch für gewöhnliche Benutzerarbeiten zur Verfügung steht und nicht ausschließlich zur Virtualisierung verwendet

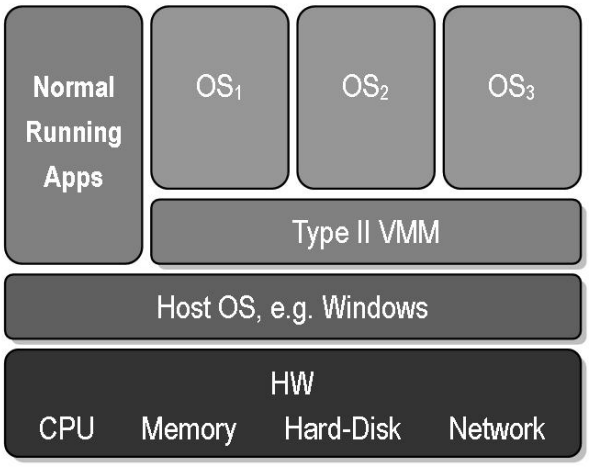

Abbildung 8: Virtual Machine Monitor Type II (Chen 2014b)

werden muss. Diese Virtualisierungslösung lässt sich sehr einfach nachträglich auf eine Vielzahl von Betriebssystemen installieren und auch wieder deinstallieren. Ein typischer Vertreter dieser Hypervisor-Klasse ist zum Beispiel VirtualBox von Oracle (Vgl. Damodaran et al. 2012, S. 24).

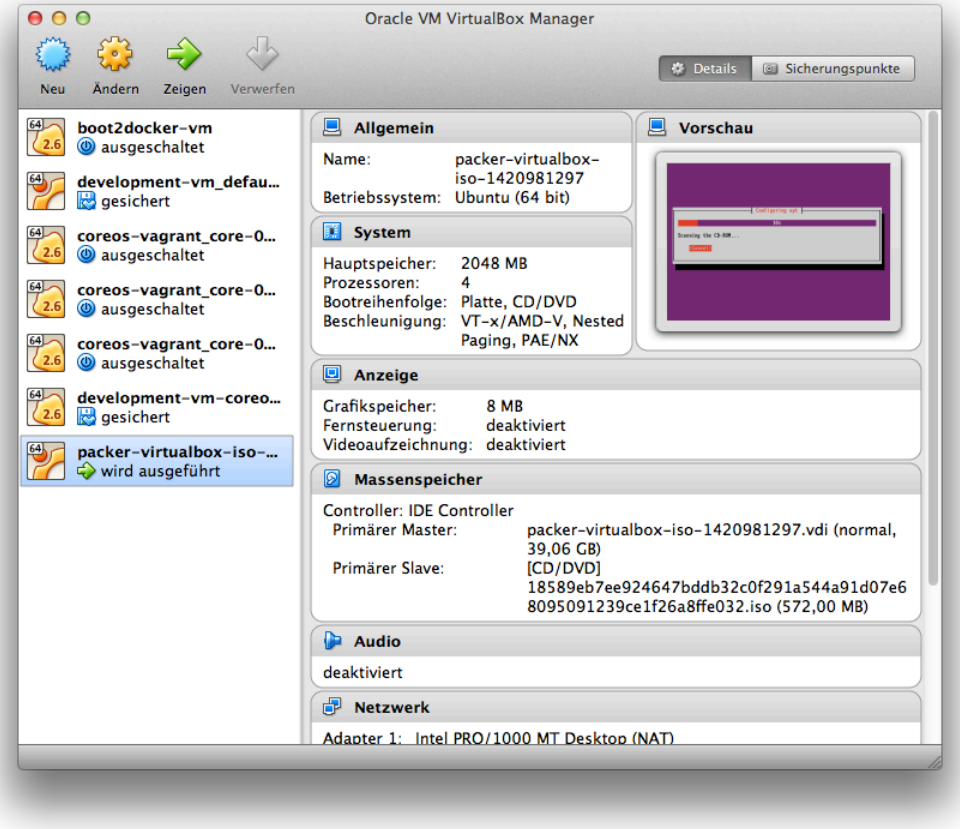

Abbildung 9: Grafische Benutzeroberfläche von VirtualBox

VirtualBox bietet nach der Installation eine grafische Benutzeroberfläche, mit der man seine virtuellen Maschinen einfach verwalten kann. Diese Oberfläche ist in der Abbildung 9 zu sehen. Zum Beispiel ist es möglich, eine neue, leere virtuelle Maschine zu definieren, ihre Hardware-Einstellungen wie zum Beispiel Menge an Prozessoren und Arbeitsspeicher zu konfigurieren und sie zu starten. Anschließend lässt sich dann zum Beispiel mit einem Installationsmedium ein Betriebssystem auf der Gast-Maschine installieren. Es lassen sich aber auch bestehende Maschinen exportieren und importieren, auf denen zum Beispiel bereits ein Betriebssystem und weitere Anwendungsprogramme installiert sind. Schließlich lassen sich virtuelle Maschinen auch einfach pausieren, stoppen und natürlich löschen (Vgl. Oracle Corporation 2014, S. 11 ff.).

Genau wie die Bare-Metal-Hypervisor, bietet auch Virtualbox die Möglichkeit, laufende Maschinen zwischen zwei Rechnern zu übertagen (Vgl. Oracle Corporation 2014, S. 111).

Neben der grafischen Benutzeroberfläche bietet VirtualBox aber auch alle Funktionen in Form von Kommandozeilen-Programmen an (Vgl. Oracle Corporation 2014, S. 113). Damit lässt sich VirtualBox auch leicht automatisiert verwenden, zum Beispiel von einem CI-Server aus, der Testumgebungen starten möchte.

2.2.1.3 Unikernels

Eine relativ neue Entwicklung innerhalb der Systemvirtualisierung mittels Hypervisor sind die sogenannten Unikernels (Vgl. Madhavapeddy et al. 2013, Abstract und Introduction). Vertreter dieser Idee kommen zum Beispiel aus dem Bereich der Serviceorientierte Architektur (SOA) oder der Microservice Architecture. In diesen Architekturen wird versucht, die Teile eines IT-Systems als einzelne Dienste zu betrachten, die (meist über Netzwerk) miteinander kommunizieren. Es ist dabei von Vorteil, wenn diese Dienste jeweils eine bestimmte Aufgabe erledigen und über eine einfache Schnittstelle angesprochen werden können. Solche Dienste können nun jeweils über eine eigene VM innerhalb der Virtualisierung abgebildet werden. Hierbei wird nun kritisiert, dass die Installation eines kompletten Gastbetriebssystems und die in ihm laufenden Anwendungsprogramme meist wesentlich mehr Resourcen und Programmcode verwenden und mehr Zugriffspunkte bieten, als für den eigentlichen Dienst benötigt wird. Als konzeptionelles Beispiel kann man sich hier einmal einen Dienst vorstellen, der als einzige Aufgabe hat, mit Hilfe einer fortlaufenden Zahl neue Kundennummern zu generieren. Es ist nun durchaus nachvollziehbar, dass ein komplettes Gastbetriebssystem inkl. aller in ihm installierten Zubehörprogramme, Dokumentationen und Treiber sehr viel mehr Resourcen verbraucht, als für die eigentliche Operation sinnvoll erscheint. Die höhere Menge an Quellcode und typische Betriebssystemschnittstellen stellen zudem auch ein erhötes Sicherheitsrisiko dar, da sie potentiell mehr Angriffsvektoren bieten.

Als Antwort entstehen deshalb in jüngster Vergangenheit neue Lösungen, bei denen man seinen eigentlichen Programmcode mit Hilfe einer Library entwickelt, die einfache Schnittstellen zur Verfügung stellt, um zum Beispiel Netzwerkkommunikation oder ähnliches zu betreiben. Der Programmcode wird dann mit Hilfe dieser Library in eine sehr viel kleinere VM kompiliert, die wieder mit Hilfe eines Hypervisors ausgeführt werden kann. Ein Beispiel für eine solche Library ist MirageOS (Vgl. Madhavapeddy et al. 2013). Der Ansatz dieser Library wird in Abbildung 10 gezeigt.

Ein Nachteil dieser Lösung ist, dass man lediglich Funktionalität wiederverwenden kann, die genau durch die benutzte Library zur Verfügung steht. So kann man auch lediglich in der Programmiersprache arbeiten, die der Compiler der Library versteht. Im Falle von MirageOS ist dies die Sprache Ocaml. Es ist somit zum Beispiel nicht möglich, einen Interpreter für PHP Hypertext Preprocessor (PHP) zu starten und PHP-Scripte auszuführen, wie das mit allen gängigen Betriebssystemen wie Linux, MacOS oder Windows der Fall ist. Unikernels lassen sich somit zumindest bisher nur für sehr spezielle Dienste und Problemestellungen einsetzen.

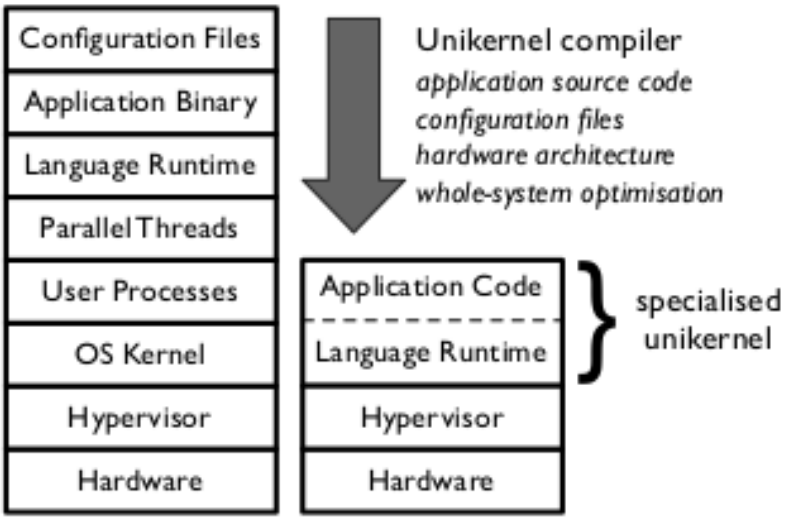

Abbildung 10: Vergleich normale VM / Unikernel (Madhavapeddy et al. 2013, Abb. 1)

2.2.2 Betriebssystemvirtualisierung mittels OS-Containern

Ein ganz anderer Ansatz als die Systemvirtualisierung mittels Hypervisor ist die sogenannte Betriebssystemvirtualisierung mittels OS-Container. Dabei wird nicht versucht, ein komplettes Gastsystem mit eigener, virtualisierter Hardware und eigenem Kernel auszuführen. Vielmehr basiert dieser Ansatz darauf, dass sich alle virtuellen Betriebssysteme die Hardware und einen gemeinsamen Kernel teilen (Vgl. Turnbull 2014, Introduction). Offensichtlich lassen sich damit nicht verschiedenste Betriebssysteme auf einer Maschine virtualisieren, sondern immer nur Betriebssysteme, die auf dem gleichen Kernel basieren wie das Hostsystem. Dafür entfällt die Notwendigkeit, Hardware zu emulieren oder zu virtualisieren. In Bezug auf das Ring-Schemas des Prozessors gilt: Der gemeinsame Kernel arbeitet im Ring 0 und somit direkt auf der physischen Hardware. Das virtualisierte System beinhaltet lediglich Bibliotheken und Anwendungsprogramme und arbeitet somit in Ring 3. Die Notwendigkeit, nicht erlaubte Aufrufe zu finden und zu behandeln entfällt damit. Eine solche isolierte Umgebung, OS-Container genannt, gilt damit als grundsätzlich effizienter als komplette virtuelle Maschinen. Der Aufbau einer OS-Virtualisierung ist noch einmal in Abbildung 11 zu sehen.

Damit die einzelnen Container dennoch isoliert laufen und weder das Hostsystem noch andere Container manipulieren können, muss der Kernel spezielle Schnittstellen anbieten. Entsprechende Technologien sind im Laufe der Zeit bei einer Reihe von Betriebssystemen entwickelt worden, zum Beispiel auch für Linux und Windows. Für andere Betriebssysteme wiederum existiert keine entsprechende Schnittstelle, zum Beispiel Mac OS.

Ein prominenter Vertreter dieser Virtualisierungs-Art ist zur Zeit das Produkt Docker (Vgl. Docker Inc. 2014). Docker ist eine Open-Source-Plattform, mit deren Hilfe eine einfache Definition und Verwaltung solcher Container für den Linux-Kernel möglich ist. Kürzlich wurde aber auch eine Kooperation zwischen Microsoft und der Docker Inc., dem Unternehmen hinter Docker, bekannt gegeben. Ziel der Kooperation ist es unter anderem, dass sich entsprechende Container auch für Windows-Betriebssysteme erstellen lassen (Vgl. Heise Zeitschriften Verlag 2014).

Docker setzt auf eine Reihe von Schnittstellen auf, die inzwischen Teil des offizielen Linux-Kernels sind. Unter anderem gehören dazu die Cgroups und die Namespaces:

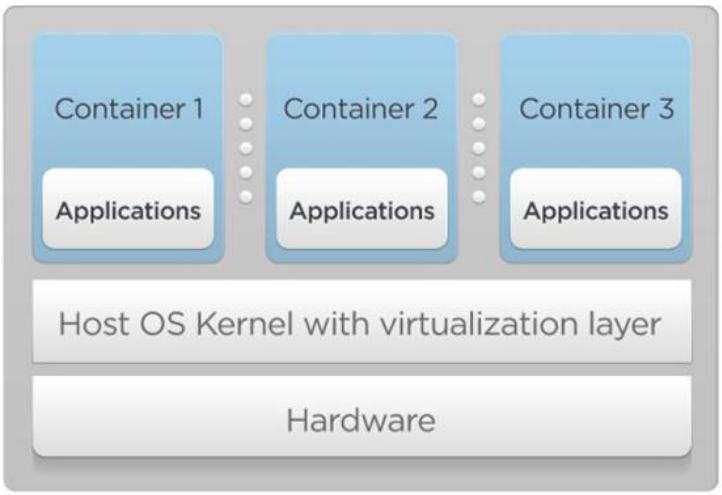

Abbildung 11: OS-Container (Francies 2014)

Cgroups (Control Groups) (Vgl. Scheepers 2014, S. 3) bieten die Möglichkeit, die Ressourcen-Nutzung bei der Ausführung von Prozessen durch den Kernel einzugrenzen. So lassen sich für Prozesse bestimmte Grenzen der Arbeitsspeicher- und Prozessor-Nutzung definieren. Außerdem lassen sich solche Prozesse auch komplett stoppen beziehungsweise pausieren und wieder starten.

Kernel Namespaces (Vgl. Scheepers 2014, S. 3) erlauben es, die Sichtbarkeit von Prozessen untereinander einzugrenzen. So können über Namespaces zum Beispiel die Prozess-Identifikatoren (PID) für jeden Container neu vergeben werden. Jeder Container kann somit einen Prozess mit der ID 1 besitzen. Aber auch andere Aspekte des Betriebssystems, wie zum Beispiel Hostname, Benutzer-IDs, Dateisystem und Netzwerk-Zugriffe lassen sich mit den Kernel Namespaces voneinander trennen.

Docker setzt zusätzlich ein weiteres nützliches Feature, sogenannte Union-Filesystems:

Union-Filesystems sind Dateisysteme, die es erlauben, Dateisysteme transparent übereinander zu legen, um so weniger Speicherplatz zu verbrauchen (Vgl. Scheepers 2014, S. 3). Das zugrundeliegende Verfahren nennt sich Copy-On-Write. Verwenden zwei Container zum Beispiel die gleiche Linux-Distrubution, so werden die entsprechenden Libraries und Anwendungsprogramme nur einmal auf der Festplatte vorgehalten. Das entsprechende Dateisystem wird dabei read-only unter ein für den Container beschreibbares Dateisystem gelegt. Die beiden Container unterscheiden sich also auf der Festplatte nur durch die tatsächlich individuell von ihnen geschriebenen Daten. Die meisten Union-Filesysteme erlauben dabei ein mehrfaches Übereinanderlegen, so dass eine ganze Hierarchie an ineinander verschachtelten Dateisystemen entstehen kann, um die Unterschiede zwischen den einzelnen Containern maximal effizient auf der Festplatte abzulegen.

In der Abbildung 12 wird zum Beispiel die gesamte Ubuntu-Distribution nur einmal auf der Festplatte abgelegt, obwohl sie von vier laufenden Containern verwendet wird. Sämtliche Dateisysteme sind read-only, bis auf die in rot angedeuteten Kästen, in denen die Container Laufzeitdaten speichern. Container 4 schreibt aber zum Beispiel keine Daten und ist somit komplett nur lesend.

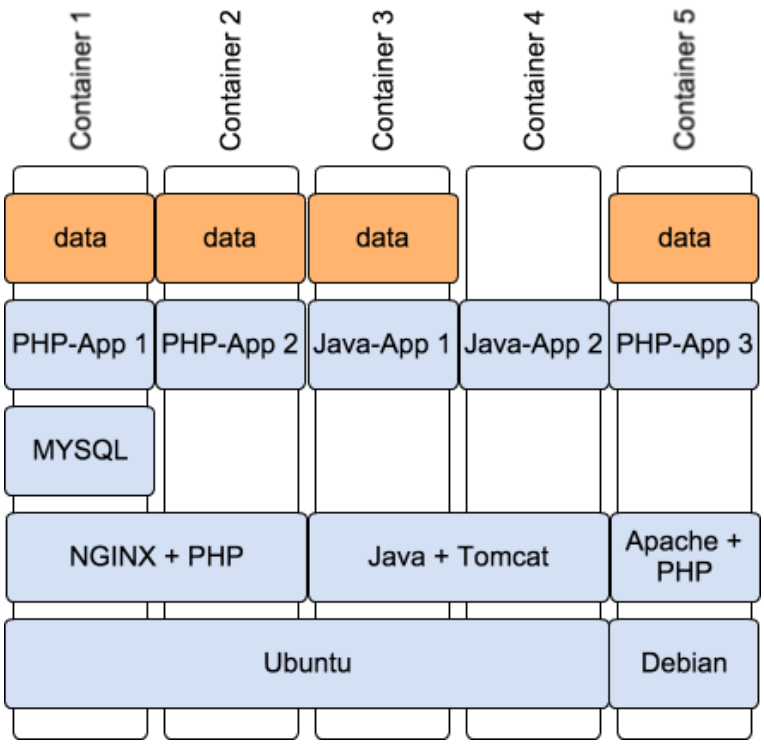

Abbildung 12: Union Filesystem

2.2.3 Vergleich der Virtualisierungsansätze

Die Tabelle 13 fasst noch einmal alle zuvor aufgeführten Virtualisierungsansätze zusammen. Deren unterschiedliche Vor- und Nachteile sollen nun vor allem auch in Hinblick auf die Problemstellung noch einmal vergleichend diskutiert werden.

Bei der Installation der verschiedenen Ansätze kann man grundsätzlich zwischen denen unterscheiden, die sich als natives Betriebssystem auf nackter Hardware installieren lassen und denen, die man als zusätzliches Anwendungsprogramm installiert. Grundsätzlich ist die Installation als Anwendungsprogramm einfacher und erfordert zum Beispiel weniger Zugriffsrechte. So lassen sich entsprechende Virtualisierungslösungen zum Beispiel auch auf dem Laptop eines Entwicklers installieren, ohne etwas an seiner grundsätzlichen Entwicklungsumgebung zu ändern. Dies würde zum Beispiel auch für die Entwickler der Pixelhouse GmbH gelten, die überwiegend auf MacOS X arbeiten. Für die eigentliche Problemstellung, nämlich die Optimierung der Testumgebungen, ist diese Möglichkeit aber nicht zwingend notwendig. So könnte rein zum Bereitstellen von Testumgebungen auch ein Bare Metal Hypervisor eingesetzt werden.

Bis auf die Produkte VMWare vSphere und Citrix XenServer handelt es sich bei allen Ansätzen um kostenlose Open-Source-Produkte. Zwar stellt das Anfallen von Lizenzkosten für die Pixelhouse GmbH kein grundsätzliches Hinderniss dar. Beide Produkte bieten in Hinblick auf die Problemstellung aber keine besondere, notwendige Funktionalität an. So bieten zwar beide Produkte zum Beispiel die Live-Migration von laufenden Maschinen an. Allerdings ist die Live-Migration von virtuellen Maschinen eben keine notwendige Funktionalität, wenn es um das Bereitstellen von Testumgebungen geht, die nach der Ausführung der Tests ohnehin wieder zerstört werden. So oder so wird eine Livemigration aber auch vom kostenlosen Produkt Oracle VirtualBox angeboten.

Grundsätzlich lassen sich mit den meisten Ansätzen verschiedene Betriebssysteme (zum Beispiel Windows und Linux) virtualisieren. Ausnahmen stellen hier lediglich MirageOS und Docker dar, da MirageOS-Unikernels kein Betriebssystem beinhalten und Docker als

OS-Container-Lösung auf Basis des Linux-Kernel lediglich Anwendungen virtualisiert und kein Betriebssystem. Für die konkrete Problemstellung ist die Virtualisierung von Windows Betriebssystemen allerdings weniger interessant. Wie die meisten Firmen, die eine Internetplattform betreiben, verwendet die Pixelhouse GmbH in ihrer Produktivumgebung lediglich Linux-Betriebssysteme. Bis auf MirageOS kommen hier also alle Lösungen in Frage.

Ein weiterer Punkt, der die Virtualisierung über MirageOS in Hinblick auf die Problemstellung unzureichend macht, ist die fehlende Möglichkeit, innerhalb einer virtuellen Maschine einen PHP-Interpreter auszuführen. Gleiches würde auch für andere typische Web-Programmiersprachen wie zum Beispiel Ruby, Java oder Javascript gelten. MirageOS bietet eben lediglich das Schreiben von OCamle-Anwendungen an.

Docker beziehungsweise OS-Container-Virtualisierungen im Allgemeinen stellen im Vergleich zu anderen Virtualisierungsansätzen eine resourcensparendere Lösung dar. So muss nicht für jede neue virtuelle Umgebung entsprechender Arbeitsspeicher und Festplattenplatz für das Betriebssystem vorgehalten werden. Innerhalb eines OS-Containers verbrauchen lediglich die tatsächlich laufenden Prozesse Arbeitsspeicher. Der Festplattenplatz wird zusätzlich noch effizienter ausgenutzt, da mit Hilfe eines Union-Filesystems sogar nur das Delta zwischen verschiedenen Containern redundant gespeichert wird. Das Ausführen mehrerer Instanzen des selben Container-Images verbaucht also keinen zusätzlichen Festplattenplatz. Hinzukommt, dass das Starten eines Containers mitunter in Bruchteilen einer Sekunde erledigt ist, wohingegen das Starten eines kompletten Betriebssystems typischerweise einige Sekunden in Anspruch nimmt. Dieser Geschwindigkeitsvorteil könnte gerade in Hinblick auf die Problemstellung interessant sein und dazu beitragen, dass sich die Gesamtdauer der Testausführung reduziert.

	Hypervisor					OS-Container
	Bare-Metal			Hosted	Unikernel	
Eigenschaft	VMWare	KVM	XenServer	VirtualBox	MirageOS	Docker
Installation						
- auf Hardware	X	X	X	-	X	-
- als Anwendung	-	-	-	X	X	X
Lizensierung						
- kostenpflichtig	X	-	X	-	-	-
- Open Source	-	X	-	X	X	X
Funktion						
- Live-Migration	X	X	X	X	-	-
- Verschiedene OS	X	X	X	X	-	-
- PHP möglich	X	X	X	X	-	X
- Dateisystem Wiederverwendung	-	-	-	-	-	X
Virtualisierung						
- umfasst Hardware	X	X	X	X	X	-
- beinhaltet OS	X	X	X	X	-	-

Abbildung 13: Zusammenfassung Virtualisierungsansätze

3 Konzeption

Nachfolgend soll nun zunächst die aktuelle Produktiv- und die aktuellen Testumgebungen der Pixelhouse GmbH beschrieben werden. Dabei werden diese in Bezug auf typische Schwächen beziehgunsweise Probleme von Systemumgebungen untersucht. Anschließend soll ein Konzept für eine neue Testumgebung beschrieben werden, das mit Hilfe der im Kapitel Grundlagen erarbeiteten Kenntnisse zur Virtualisierung die zuvor beschriebenen Probleme umgeht. Es wird außerdem eine Auswahl zweier konkreter Virtualisierungstechnologien getroffen, mit deren Hilfe später eine Implementierung des Konzepts erfolgt.

3.1 Beschreibung der aktuellen Produktivumgebung

Die Betriebsumgebung, die von der Pixelhouse GmbH in Produktion eingesetzt wird, besteht aus einer Vielzahl von Komponenten und Diensten. Diese Komponenten werden bislang jeweils auf echter Serverhardware betrieben.

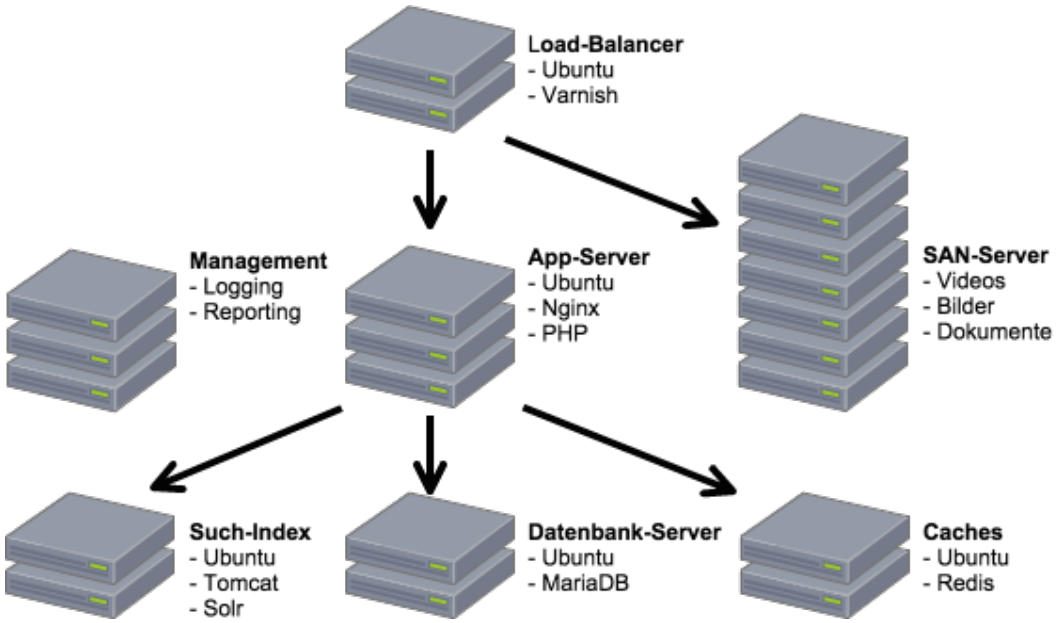

Abbildung 14: Die Produktiv-Umgebung der Webseite chefkoch.de

Der Aufbau der Produktivumgebung ist in der Abbildung 14 zu sehen. Im Zentrum der Infrastruktur stehen insgesamt drei Appserver. Bei jedem Appserver handelt es sich dabei um einen Ubuntu-Server, auf dem der Webserver Nginx (NGINX, Inc. 2015) und der Script-Interpreter PHP (The PHP Group 2015) ausgeführt wird, um HTTP-Anfragen mit Hilfe statischer oder dynamische Inhalte zu beantworten. Auf den App-Servern läuft somit auch die in PHP geschriebene Anwendung selbst. Für die langfristige Datenhaltung greifen die PHP-Scripte dabei auf zwei Datenbankserver (Master und Slave) zu, auf denen ebenfalls Ubuntu und der Mysql-Fork MariaDB (MariaDB Foundation 2015) installiert sind. Für kurzfristige und hochverfügbare Datencaches gibt es zwei Ubuntu-Server (Master und Slave) auf denen der Key-Value-Store Redis (Redis 2015) läuft. Zudem wird auf weiteren zwei Ubuntu-Servern (Master und Slave) ein Such-Index mit Hilfe der Suchmaschine Solr (The Apache Software Foundation 2015a) betrieben, die als Java-Anwendung in einem Tomcat-Application-Server (The Apache Software Foundation 2015b) läuft. Binärdateien (Videos, Bilder, Dokumente, etc.) werden auf insgesamt 7 weiteren Servern redundant abgelegt. Damit die Appserver und die SAN-Server (Storage Area Network) von außen unter einheitlichen HTTP-Adressen erreichbar sind und sich die Last gleichmäßig auf die einzelnen Server verteilt, landen sämtliche HTTP-Anfragen per DNS-Round-Robin zu-

nächst bei einem von zwei Ubuntu-Servern, auf denen Varnish (Varnish Software 2015) als Load-Balancer installiert ist. Diese nehmen die Anfragen entgegen und verteilen sie an die dahinter liegenden Server. Varnish ist zudem ein effizienter HTTP-Cache, der in der Lage ist, die von den App- oder SAN-Servern zurückgelieferten Antworten zu cachen und bei erneuter Anfrage selbst auszuliefern. Zu guter Letzt gibt es noch drei weitere Server, auf denen Software für die Administration und das Reporting läuft.

Das größte Problem der aktuellen Produktivumgebung ist, dass sie manuell konfiguriert wird. Änderungen an der Konfiguration werden typischerweise in einer Nachtschicht durchgeführt und anschließend händisch getestet. Dies hat in der Vergangenheit schon mehrfach zu Ausfällen geführt. Humble und Farley beschreiben in ihrem Buch eine sinnvolle Alternative: „The change you want to make shold first have been tested on one of your production like testing environments, and automated tests should be run to ensure that it doesn't break any of the applications that use the environment. The change should be made to version control and then applied through your automated process for deploying infrastructural changes." (S. Humble und Farley 2010, S. 287) Die aktuellen Testumgebungen, die nun im nächsten Unterkapitel beschrieben werden, ähneln der Produktion bisher aber leider nur wenig.

3.2 Beschreibung der aktuellen Testumgebungen

Die aktuellen Testumgebungen der Pixelhouse GmbH besitzen einen extrem reduzierten Aufbau. Dieser wird in Abbildung 15 gezeigt. Die Testumgebungen werden alle auf ein und derselben Hardware-Maschine installiert. Fast alle Komponenten der zuvor beschriebenen Produktivumgebung werden dabei von den einzelnen Testumgebungen gemeinsam verwendet. Beispielsweise nutzen alle Testumgebungen den gleichen Varnish für Load-Balancing und Caching. Dieser reicht die HTTP-Anfragen an den selben Nginx-Web-Server und PHP-Interpreter weiter und puffert die Antworten bei Bedarf gleichermaßen. Als Backend-Dienste werden dieselbe MariaDB-Datenbank, derselbe Redis-Cache und derselbe Solr-Such-Index verwendet. Die Binärdateien werden ebenfalls auf demselben Host von allen Testumgebungen gemeinsam verwendet. Einziger Unterschied zwischen den einzelnen Testumgebungen ist, dass aufgrund einer speziellen Konfiguration im Nginx-Webserver je nach aufgerufener URL (Uniform Resource Locator) vom PHP-Interpreter andere PHP-Scripte geladen werden und somit jeweils eine andere Version der PHP-Anwendung getestet werden kann.

Wegen der Unterschiede zur Produktivumgebung eignen sich die Testumgebungen grundsätzlich nicht zum aussagekräftigen Testen von Infrastrukturänderungen. Hinzu kommt, dass Änderungen an den gemeinsam genutzten Komponenten immer alle Testumgebungen gleichermaßen betreffen. Dies kann zum Beispiel dazu führen, dass ein Produktmanager beim Testen eines neuen Features Fehler sieht, obwohl diese nicht durch das neue Feature sondern eben durch die Änderungen an der Infrastruktur ausgelöst wurden.

Ein weiteres Problem der aktuellen Testumgebungen ist, dass sie sich die Testdaten teilen. Dies ist immer dann problematisch, wenn es sich um einen schreibenden Datenzugriff handelt. So ist es zum Beispiel nicht möglich, dass zwei Tests versuchen, sich mit der gleichen E-Mail-Adresse zu registrieren. Humble und Farley empfehlen für Testumgebung deshalb allgemein die sogenannte Test Isolation (Vgl. Humble und Farley 2010, S. 337). Dabei handelt es sich um eine Strategie, die jeden Test von anderen Tests in Bezug auf seine Daten unabhängig macht. Eine mögliche Implementierung einer solchen Strategie wäre es zum Beispiel, das Transaktionssystem der Datenbank auszunutzen, damit verschiedene Datenbank-Verbindungen sich nicht gegenseitig beeinflussen. Eine weitere Möglichkeit wäre, die Daten innerhalb der Datenbank zu partitionieren, indem man die Entitäten in der Datenbank zum Beispiel aufgrund einer Namenskonvention bestimmten Tests zuordnet. Da das Datenbankschema der Seite Chefkoch.de über einige Jahre historisch gewachsen

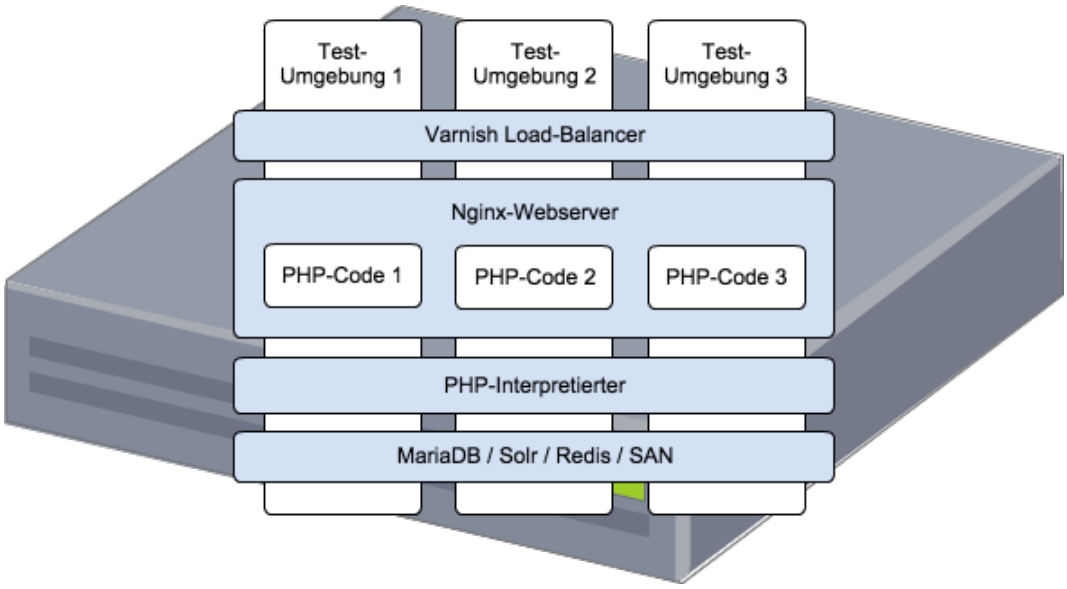

Abbildung 15: Die aktuelle Test-Umgebung

ist und an vielen Stellen ohne explizite Datenbank-Constraints (wie zum Beispiel Foreign-Keys) definiert wurde, sind alle Versuche, innerhalb der Datenbank eine Isolation der Testdaten zu erreichen bisher gescheitert.

Im Folgenden soll nun eine neue Testumgebung auf Basis von Virtualisierung konzipiert werden, in der diese Probleme nicht mehr auftreten können.

3.3 Definition einer neuen Testumgebung

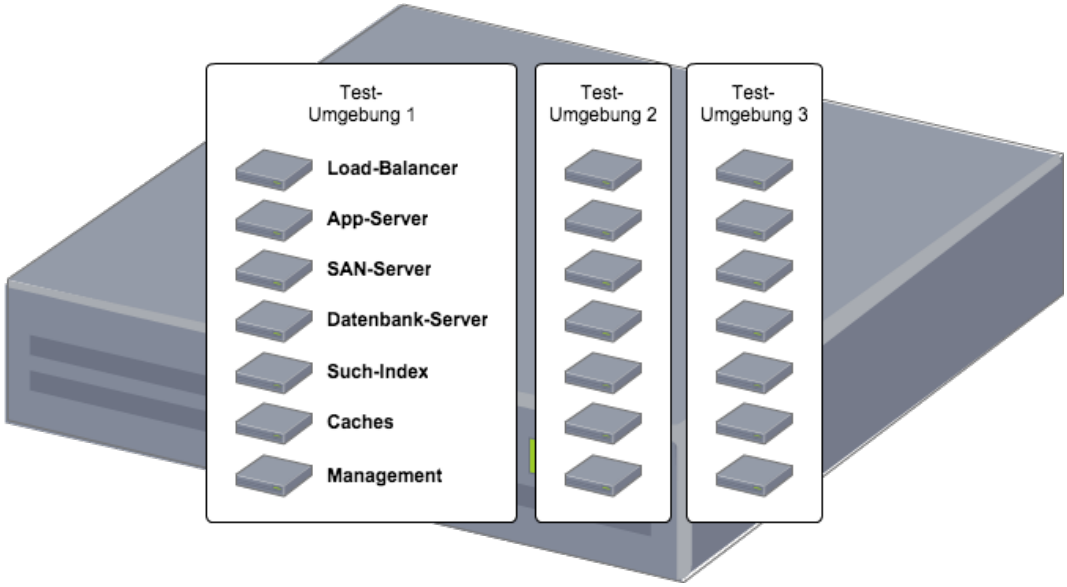

Abbildung 16: Definition der neuen Testumgebungen

Wie im Kapitel Grundlagen beschrieben, gibt es eine Reihe von Technologien, mit denen man auf einer Hardware-Maschine mehrere künstliche Maschinen simulieren kann. Es ist also zum Beispiel möglich, auf ein und der selben Hardware-Maschine mehrere Datenbankserver nebeneinander laufen zu lassen. Statt also zum Beispiel mit Hilfe von Transaktionen

oder einer Partitionierung eine Isolation innerhalb der Datenbank zu erreichen, bekommt jede Testumgebung einen komplett eigenen Datenbankserver zugewiesen. Es kann somit nicht vorkommen, dass ein Test in der einen Testumgebung die Daten eines Testes in einer anderen Testumgebung direkt verändert.

Zudem ist es grundsätzlich sogar möglich, alle Komponenten der Produktivumgebung für jede Testumgebung neu zur Verfügung zu stellen. In der Theorie kann also jede Testumgebung genau den gleichen Aufbau wie die Produktivumgebung besitzen.

Solange die konkrete Virtualisierungslösung außerdem die Möglichkeit bietet, die Zustände der einzelnen Maschinen zu sichern, ist es möglich, Konfigurationsänderungen an den Komponenten nachzuhalten und in unterschiedlichen Testumgebungen mit verschiedenen Versionsständen der Konfiguration zu arbeiten.

Der Aufbau der angestrebten Testumgebung ist noch einmal in der Abbildung 16 zu sehen.

3.4 Verwendete Virtualisierungslösungen

Im Folgenden Hauptkapitel sollen nun zwei konkrete Lösungen zur Erzeugung von Testumgebungen implementiert werden. Die konkret verwendeten Virtualisierungslösungen werden dabei von der Pixelhouse GmbH vorgegeben: Oracle VirtualBox und Docker.

Zum einen setzt die Pixelhouse GmbH bereits Oracles VirtualBox ein, um eine lokale Entwicklungsumgebung für die Anwendung der Webseite chefkoch.de auf den Laptops der Entwickler vorzuhalten. VirtualBox ist somit bereits auf fast allen Rechnern vorhanden und es existieren grundlegende Kenntnisse und Erfahrungen für dieses Produkt.

Zum anderen wird aktuell von Seiten der Server-Administratoren das Produkt Docker eingeführt, um damit Teile der Produktivumgebung der Plattform zu betreiben. Erste Produktiv-Komponenten der Webseite chefkoch.de laufen also bereits innerhalb von Linux-Containern. Die ersten Erfahrungen der Administratoren sind dabei vielversprechend. Es besteht der Wunsch, zukünftig Probleme zu minimieren, die auftreten, weil Entwicklungs-, Test- und Produktivumgebungen nicht die gleiche Technologie verwenden und sich so grundlegend voneinander unterscheiden. Es würde sich also aus Sicht der System-Administratoren anbieten, auch für die Testumgebungen Docker einzusetzen.

4 Implementierung

Bei der prototypischen Implementierung der zuvor konzipierten Testumgebung mit Hilfe des Hosted-Hypervisor Oracle VirtualBox und der OS-Containervirtualisierung Docker konnten sowohl Gemeinsamkeiten als auch Unterschiede in den beiden Ansätzen festgestellt werden.

Grundsätzlich beiden Ansätzen gemeinsam ist, dass zunächst ein ausführbares Abbild der einzelnen Komponenten erzeugt werden muss. Anschließend braucht es jeweils eine Möglichkeit, die so entstandenen Abbilder zusammen zur Ausführung zu bringen und ihnen eine Kommunikation untereinander zu ermöglichen.

Wie nun im Folgenden beschrieben wird, unterscheiden sich die beiden Ansätze dabei vor allem in der Einfachheit, die einzelnen Komponenten als gemeinesame Umgebung zu definieren und zu starten.

Die vollständige Implementierung beider Ansätze kann übrigens der beigefügten CD entnommen oder unter https://github.com/perprogramming/bachelor-arbeit/tree/master/cd online eingesehen werden.

4.1 VirtualBox

Um die zuvor konzipierte Testumgebung mit VirtualBox zu virtualisieren, ist es zunächst notwendig, fertige virtuelle Maschinen zu erzeugen, die sich bei Bedarf direkt mit VirtualBox starten lassen. Ein Werkzeug zur Erzeugung solcher Maschinenabbilder ist Packer (Siehe HashiCorp 2015a).

Es wäre außerdem von Vorteil, das Starten der einzelnen Maschinen nicht selbstständig steuern zu müssen. Ein Werkzeug, mit dem man mehrere virtuelle Maschinen zu einer Gesamtumgebung auf Basis von VirtualBox orchestrieren kann, ist Vagrant (Siehe HashiCorp 2015e).

Im Folgenden wird nun die Funktionsweise der beiden Werkzeuge Packer und Vagrant genauer beschrieben.

4.1.1 Erzeugung der einzelnen Maschinen mit Packer

„Packer ist ein Werkzeug zur Erzeugung eindeutiger Maschinenabbilder für verschiedene Plattformen auf Basis einer einfachen Konfiguration“ (Siehe HashiCorp 2015a). Packer kennt dabei vor allem drei wichtige Konzepte: Builder (Vgl. HashiCorp 2015b), Provisioner (Vgl. HashiCorp 2015d) und Post-Processor (Vgl. HashiCorp 2015c). Diese werden mit Hilfe der Konfigurations-Datei namens packer.json definiert. Die Abbildung 17 zeigt die entsprechende Konfigurations-Datei für die Komponente des Loadbalancers.

Ein Builder (Vgl. HashiCorp 2015b) beschreibt eine bestimmte Technologie, mit deren Hilfe das virtuelle Maschinen-Abbild erzeugt wird. Neben verschiedenen anderen Technologien beinhaltet Packer auch Builder zur Erzeugung von virtuellen Maschinen mit VirtualBox. Grundsätzlich gibt es dabei zwei Möglichkeiten: Zum einen kann man einen Builder vom Typ „virtualbox-iso“ verwenden. Mit diesem wird eine leere virtuelle Maschine erzeugt und anschließend mit Hilfe eines Installationsmedium im ISO-Format (Internationale Organisation für Normung) ein Betriebssystem installiert. Zum anderen kann man einen Builder vom Typ „virtualbox-ovf“ verwenden. Das Format OVF (Open Virtualization Format) ist ein generisches Format zum Import und Export von virtuellen Maschinen. Mit diesem Builder kann man also mit Hilfe von Packer ein Maschinen-Abbild auf Basis eines anderen Maschinen-Abbildes erzeugen. So kann man zum Beispiel eine fertig installierte Linux-Distribution als Ausgangspunkt zur Installation weiterer Anwendungsprogramme verwenden. Für die Testumgebung der Pixelhouse GmbH sollen die einzelnen virtuellen Maschinen idealerweise auf Basis einer fertigen Installation von Ubuntu 14.04 Server erzeugt werden, die auch im Produktivbetrieb zum Einsatz kommt.

```
{
    "builders": [
        {
            "type": "virtualbox-ovf",
            "source_path": "../ubuntu-14.04.ovf",
            "ssh_username": "vagrant",
            "ssh_password": "vagrant",
            "headless": true,
            "shutdown_command": "sudo -S shutdown -P now"
        }
    ],
    "provisioners": [
        {
            "type": "file",
            "source": "contents",
            "destination": "/tmp"
        },
        {
            "type": "shell",
            "inline": [
                "chmod a+x /tmp/contents/install.sh",
                "/tmp/contents/install.sh"
            ]
        }
    ],
    "post-processors": [
        {
            "type": "vagrant",
            "output": "vagrant.box"
        }
    ]
}
```

Abbildung 17: packer.json für den Loadbalancer

Um nun weitere Software auf der mit dem Builder erzeugten virtuellen Maschine zu installieren, kommt ein sogenannter Provisioner (Vgl. HashiCorp 2015d) zum Einsatz. Ein Provisioner ist für Packer eine Technologie, mit der es einfach möglich ist, solche Installationsvorgänge zu konfigurieren. Dabei gibt es sehr umfangreiche Ansätze wie die Werkzeuge Puppet oder Chef, die sehr mächtige Konfigurationssprachen bieten, um verschiedenste Installations- und Konfigurationsschritte in einem Betriebssystem vorzunehmen. Eine andere, eher einfache Variante wäre aber zum Beispiel der Provisioner vom Typ „Shell Script“, bei dem eben lediglich ein Shell Script innerhalb der virtuellen Maschine ausgeführt wird, um weitere Einstellungen oder Installationen vorzunehmen. Die Abbildung 18 zeigt zum Beispiel das entsprechende Script für den Loadbalancer. Ein Vorteil solcher Scripte ist, dass sie sich später auch sehr leicht für die Implementierung mit Hilfe von Docker wiederverwenden lassen.

```
#!/bin/bash

export DEBIAN_FRONTEND=noninteractive

sudo apt-get update
sudo apt-get install -y varnish

sudo mv /tmp/contents/etc/varnish/default.vcl /etc/varnish/default.vcl

sudo sed -i 's/-a :6081/-a :80/' /etc/default/varnish
```

Abbildung 18: Provisionierungs-Script für den Loadbalancer

Schließlich kennt Packer nun noch das Konzept des Post-Prozessors (Vgl. HashiCorp 2015c). Ein Post-Prozessor meint dabei eine bestimmte Art und Weise, mit der das fertige Maschinen-Abbild exportiert und für die Verwendung mit anderen Tools vorbereitet wird. So gibt es eben einen Post-Prozessor vom Typ „Vagrant“, der es ermöglicht, das fertige Maschinen-Abbild direkt mit Vagrant innerhalb einer Testumgebung zu starten. Damit sollte es möglich sein, die einzelnen fertigen virtuellen Maschinen der Testumgebung so abzulegen, dass man sie sofort als Gesamtumgebung starten kann.

4.1.2 Umgebungssteuerung mit Vagrant

„Vagrant bietet eine einfach zu konfigurierende, reproduzierbare und portierbare Betriebsumgebung auf Basis etablierter Virtualisierungs-Technologien und einen einfachen Ansatz zu deren Verwaltung [...]“ (Siehe HashiCorp 2015e).

Dazu legt man zunächst eine Konfigurationsdatei namens Vagrantfile an, in der man seine Umgebung definiert. Vagrant bietet dabei die Möglichkeit eine oder mehrere virtuelle Maschinen auf Basis von verschiedenen Virtualisierungstechnologien zu starten. So bietet es eben auch die Möglichkeit, virtuelle Maschinen mit Hilfe von VirtualBox zu starten, die mit Packer vorbereitet wurden. Es bietet weiterhin einfache Konfigurationsparameter, die zum Beispiel die Netzwerkkommunikation zwischen den virtuellen Maschinen und dem Host-Betriebssystem und auch untereinander ermöglichen. Auch einzelne Parameter wie Anzahl der virtuellen Prozessoren und die Menge des zur Verfügung gestellten Arbeitsspeicher lassen sich hier noch nachträglich definieren. Es ist auch möglich, Ordner zwischen dem Host-Betriebssystem und den virtuellen Maschinen zu teilen, um so zum Beispiel während der Entwicklung mit einer IDE (Integrated Development Environment) auf dem Host-Betriebssystem Code zu bearbeiten, der direkt in den virtuellen Maschinen ausgeführt wird. Die Abbildung 19 zeigt einen Auszug aus der fertigen Konfigurationsdatei.

Eine Besonderheit ist hier durch die Einstellung *loadbalancer.vm.provision „hosts“* gegeben. Um nämlich die Kommunikation der einzelnen Komponenten untereinander zu ermöglichen, verwenden wir das Vagrant-Plugin „vagrant-hosts“ (Vgl. Thebo 2015). Dieses wird über diese Konfigurationszeile dazu veranlasst, in die Datei */etc/hosts* jeder Komponente statische IP-Routen (Internet Protocol-Routen) auf die anderen Komponenten einzutragen. So kann zum Beispiel der Appserver davon ausgehen, dass er den Datenbankserver einfach unter dem Hostnamen *database* findet.

Mit Hilfe der fertigen Konfigurationsdatei lässt sich nun die entsprechende Umgebung mit einem einfach Kommandozeilen-Befehl starten: „vagrant up“. Vagrant lädt und startet

```
Vagrant.configure(2) do |config|
    cwd = File.dirname(__FILE__)
    ...
    config.vm.define "app" do |app|
        app.vm.box = "app"
        app.vm.box_url = "file://" + cwd + "/machines/app/vagrant.box"
        app.vm.hostname = "app"
        app.vm.network "private_network", ip: "172.28.128.6"
        app.ssh.password = "vagrant"
        app.ssh.insert_key = false
        app.vm.provider "virtualbox" do |v|
            v.memory = 256
            v.cpus = 1
        end
        app.vm.provision "hosts"
    end

    config.vm.define "loadbalancer" do |loadbalancer|
        loadbalancer.vm.box = "loadbalancer"
        loadbalancer.vm.box_url = "file://"
            + cwd + "/machines/loadbalancer/vagrant.box"
        loadbalancer.vm.hostname = "loadbalancer"
        loadbalancer.vm.network "private_network", ip: "172.28.128.7"
        loadbalancer.vm.network "forwarded_port", guest: 80, host: 8080
        loadbalancer.ssh.password = "vagrant"
        loadbalancer.ssh.insert_key = false
        loadbalancer.vm.provider "virtualbox" do |v|
            v.memory = 256
            v.cpus = 1
        end
        loadbalancer.vm.provision "hosts"
        loadbalancer.vm.provision "shell",
            inline: "sudo service varnish start",
            keep_color: true
    end
end
```

Abbildung 19: Auszug aus der Vagrantfile der fertigen Testumgebung

dann alle in der Konfigurationsdatei angegebenen Maschinen und setzt die gewünschten Konfigurationen für Netzwerkkommunikation und Hardware-Parameter um.

Anschließend lässt sich die innerhalb der Umgebung laufende Anwendung zum Beispiel mit einem Browser des Host-Betriebssystems ansteuern.

4.2 Docker

Auch bei der Verwendung von Docker zur Virtualisierung der Testumgebung mit Hilfe von OS-Containern stellt sich zunächst die Frage, wie man die einzelnen Container erzeugt.

Im Gegensatz zu VirtualBox ist dazu keine weitere Software notwendig, da Docker bereits selbst die Möglichkeit zur Erzeugung solcher Images mit sich bringt. Es ist dazu lediglich notwendig, eine einfach Definitionsdatei, eine sogenannte Dockerfile, zu schreiben (Vgl. Docker Inc. 2015c).

Eine Möglichkeit zur Orchestrierung der einzelnen Container zu einer Gesamtumgebung wird ebenfalls von der Docker Inc. angeboten und nennt sich docker-compose (Vgl. Docker Inc. 2015b).

Sowohl auf die Verwendung der Dockerfiles als auch auf die Verwendung von docker-compose wird im Folgenden näher eingegangen.

4.2.1 Images mit Dockerfiles bauen

Docker bietet eine einfache, integrierte Möglichkeit, neue Maschinen-Abbilder zu erzeugen (Vgl. Docker Inc. 2015c). Wie im Grundlagen-Kapitel beschrieben, werden diese Maschinen-Abbilder mit Hilfe eines Union-Filesystems gestartet. Einzelne Abbilder lassen sich so durch das Hinzufügen eines weiteren, überlagernden Dateisystems zu neuen Abbildern erweitern. Die Definitions-Datei eines Docker-Images, Dockerfile genannt, startet deshalb immer mit der Angabe des zugrundeliegenden Images. Anschließend lassen sich einfach Kommandozeilen-Befehle definieren, die in der laufenden Maschine ausgeführt werden. Führen diese Befehle zu Änderungen am Dateisystem, so werden diese Änderungen eben in einer neuen Dateisysteme-Ebene festgehalten, die Teil des fertigen Abbildes wird. Genauso einfach lassen sich auch Dateien in die Maschine kopieren, die auf dem Host-System liegen. Schließlich lässt sich auch ein Befehl konfigurieren, der standardmäßig ausgeführt wird, wenn man das Image später als Container startet. Die Abbildung 20 zeigt die entsprechende Datei für die Loadbalancer-Komponente. Hier ist nun schön zu sehen, dass wir das gleiche Installations-Script wie bei der Erzeugung der Maschinen-Abbilder für VirtualBox verwenden können.

```
FROM ubuntu:14.04

ADD contents /tmp/contents
RUN chmod a+x /tmp/contents/install.sh
RUN /tmp/contents/install.sh

CMD /usr/sbin/varnishd -F -f /etc/varnish/default.vcl
```

Abbildung 20: Dockerfile des Loadbalancers

Ein Image lässt sich dann einfach mit dem Befehl „docker build -t chefkoch/loadbalancer .“ erzeugen. Anschließend lässt sich das Image mit dem Befehl „docker run chefkoch/loadbalancer“ als Container ausführen. Es ist sogar möglich, das fertige Image mit dem einfachen Befehl „docker push chefkoch/loadbalancer“ in einem Web-Verzeichnis, der sogenannten docker-registry unter https://registry.hub.docker.com/, zu veröffentlichen (Vgl Docker Inc. 2015d). So ist es jedem möglich, dieses Image oder auch andere Images des Verzeichnisses als Basis für ein weiteres Image zu verwenden oder auch einfach nur als Container auszuführen. Solange die Container aber auf dem selben Hardware-Host erzeugt und gestartet werden, benötigt man diese Funktionalität nicht.

4.2.2 Steuerung der Umgebung mit docker-compose

Um nun die mit Docker erzeugten Images zu einer Gesamtumgebung zu orchestrieren, bietet sich das Tool docker-compose (Vgl. Docker Inc. 2015b) an. Dafür muss zunächst eine Konfigurationsdatei namens docker-compose.yml definiert werden, in der die einzelnen Maschinen der Umgebung und weitere Einstellungen wie zum Beispiel zur Netzwerkkommunikation zwischen den Maschinen konfiguriert werden. Die Abbildung 21 zeigt die fertige Konfigurationsdatei für die neue Testumgebung.

```
loadbalancer:
    build: ./machines/loadbalancer
    ports: [80]
    links: [app]

app:
    build: ./machines/app
    expose: [80]
    links: [database, cache, search]

database:
    build: ./machines/database
    expose: [3306]

cache:
    build: ./machines/cache
    expose: [6379]

search:
    build: ./machines/search
    expose: [8080]
```

Abbildung 21: Die docker-compose.yml der neuen Testumgebung

Auch bei diesem Ansatz müssen die einzelnen Komponenten in der Lage sein, miteinander zu kommunizieren. Tatsächlich geschieht dies ebenfalls dadurch, dass in die Datei „/etc/hosts“ der einzelnen Komponenten statische IP-Routen auf anderen Komponenten eingetragen werden. Tatsächlich ist dazu aber kein zusätzliche Plugin wie bei Vagrant notwendig, da docker-compose selbst eine entsprechende Funktionalität mitbringt, die sogenannten Container-Links (Vgl. Docker Inc. 2015a). Wie in der Abbildung 21 zu sehen, kann man an eine konkrete Komponte einfach Links zu anderen Komponenten eintragen. So kann auch hier der Appserver einfach unter dem Hostnamen *database* auf den Datenbankserver zugreifen.

Die neue Testumgebung lässt sich schließlich mit dem einfachen Befehl „docker-compose up“ starten.

5 Evaluation

Nachdem nun eine prototypsche Implementierung beider Ansätze vorgenommen wurde, soll versucht werden, eine objektive und vergleichende Evaluierung durchzuführen.

Dazu muss zunächst eine passende Methodik der Evaluation definiert werden. Anschließend kann die Evaluation dann durchgeführt werden.

5.1 Methodik der Evaluation

Grundsätzlich meint Evaluation den „Prozess der Beurteilung des Wertes eines Produktes, Prozesses oder eines Programmes, was nicht notwendigerweise systematische Verfahren oder datengestützte Beweise zur Untermauerung einer Beurteilung erfordert“ (S. Wottawa und Thierau 1990, S. 9).

Das im Folgenden verwendete systematische Verfahren soll darin bestehen, dass zunächst bestimmte messbare Kriterien definiert werden, zum Beispiel die Kosten der implementierten Lösung. Damit die einzelnen Kriterien beziehungsweise deren Messpunkte miteinander in Beziehung gebracht werden können, sind zwei Vorarbeiten notwendig:

Erstens müssen die Messpunkte der einzelnen Kriterien einer bestimmten Skala zugeordnet werden. Dazu soll hier der Einfachheit halber die ordinale Skale der natürlichen Zahlen von 1 bis 10 verwendet werden. Die 10 meint dabei immer das beste Ergebnis, die 1 das schlechteste. Zum Beispiel würde man die Kosten der teureren Implementierung mit 1 bewerten und die der günstigeren mit dem entsprechenden proportionalen Wert, also zum Beispiel 5, falls der günstigere Ansatz ungefähr die Hälfte kostet. Eine kostenlose Implementierung würde entsprechend dem Wert 10 der Skala zugeordnet werden.

Zweitens ist es notwendig, die jeweiligen unterschiedlichen Kriterien zu gewichten, da ein Kriterium in Hinblick auf die konkrete Problemstellung weniger wichtig sein kann als ein anderes. Dies soll jeweils über die Definition eines Faktors erfolgen, der sich ebenfalls im Bereich 1 bis 10 bewegt. 10 würde also bedeuten, dass ein Kriterium sehr wichtig ist. 1 hingegen bedeuet, dass ein Kriterium eine sehr geringe Rolle spielt.

Mit Hilfe dieses Verfahrens ist es möglich, am Ende einen konkrete numerische Bewertung der Lösung zu errechnen, in dem man den gewichteten Mittelwert über alle Kriterien berechnet. Das Ergebnis würde sich somit ebenfalls im Bereich von 1 bis 10 bewegen, wobei eine sehr gute Lösung eben die Bewertung 10 erhalten würde, eine sehr schlechte Lösung die Bewertung 1.

Es ist wichtig zu betonen, dass, auch wenn das Verfahren aufgrund der mathematischen Herangehensweise einen sehr rationalen und objektiven Eindruck macht, die Gewichtung der einzelnen Kriterien rein subjektiv in Bezug auf die konkrete Problemstellung und die Ausgangslage bei der Pixelhouse GmbH erfolgt. Somit ist auch die schließliche Bewertung zumindest in diesem Sinne willkürlich.

5.2 Durchführung der Evaluation

Es ist nun notwendig, den zuvor recht allgemein beschriebenen Ansatz der Evaluation mit konkreten Kriterien und deren Gewichtung zu versehen. Anschließend kann dann die Ermittlung der Messwerte und eine Diskussion des errechneten Bewertungsergebnisses erfolgen.

5.2.1 Definition und Gewichtung der Evaluationskriterien

In Bezug auf die Problemstellung und Zielsetzung der Arbeit sind für die Pixelhouse GmbH vor allem Kriterien wichtig, die die Ausführungsdauer der Tests beinflussen. Dazu werden im Folgenden drei Kriterien definiert:

Erstens spielt die Dauer für das Erzeugen der eigentlichen Maschinen eine entscheidene Rolle. So muss nach jeder Änderung am Programmcode oder bei Änderungen an der Konfiguration ein neues Maschinenabbild für VirtualBox beziehungsweise ein neues Container-Image für Docker erzeugt werden. Die Dauer für die Erzeugung lässt sich derart der Skala von 1 bis 10 zuordnen, dass die langsamere der beiden Erzeugungsvorgänge die Bewertung 1 erhält und die anderen einen entsprechend proportionalen Wert, wenn man davon ausgeht, dass eine Erzeugung die gar keine Zeit kostet mit 10 Punkte bewertet wird.

Als zweites Kriterium wird die Dauer für das Starten der ersten Umgebung herangezogen, wobei hier ebenfalls der längere Startvorgang mit 1 und ein sofortiges Starten mit 10 bewertet wird. Hier wird bewusst das Starten der ersten Umgebung gemessen, da es sein könnte, dass mit jeder weiteren Umgebgung, die auf dem gleichen Hardware-Host gestartet wird, der Startvorgang potentiell langsamer wird, da sich die einzelnen Umgebungen ja letzendlich die gleiche Hardwareresourcen teilen.

Der zuletzt angedeutete Aspekt, nämlich ein geändertes Laufzeitverhalten aufgrund von endlichen Resourcen auf einer Hardwaremaschine, soll nun ebenfalls durch ein Kriterium messbar gemacht werden. Wie bereits in der Einleitung erwähnt, ist laut Hamble und Farley bei der Parallelieserung von Tests die Gesamtausführungs aller Tests lediglich durch die Dauer des längsten Tests und die Größe des Hardware-Budgets, also durch die Menge der zur Verfügung stehenden Hardware begrenzt (Vgl. Humble und Farley 2010, S. 310). In Bezug auf die beiden unterschiedliche Ansätze kann man somit sagen, dass eine Lösung, die weniger Resourcen verbraucht, besser ist als diejenige, die mehr Resourcen verbraucht, da man mit ihr auf der gleichen zur Verfügung stehenden Hardware mehr Testumgebungen und somit mehr Tests parallelisieren kann. Zur Messung dieser Eigenschaft wird deshalb untersucht, wieviele Umgebungen sich mit dem jeweiligen Ansatz auf einer konkreten Testhardware starten lassen. Die größere Anzahl wird dann wiederrum mit dem Wert 10 bewertet, die kleinere entsprechend proportional, wobei 1 für nur eine gestartete Umgebung steht. Es wäre noch spannender, sogar das tatsächliche Laufzeitverhalten einer gestarteten Umgebung in Bezug auf die zur Verfügung stehenden Hardware-Resourcen und der Anzahl an anderen Testumgebungen zu untersuchen. So kann es zwar sein, dass beide Ansätze gleich viele Umgebungen starten können, diese aber langsamer oder schneller Anwendungsanfragen abarbeiten. Erste Versuche, entsprechende Zahlen zu ermitteln haben sich aber als sehr schwierig herausgestellt, da gerade im Grenzbereich (also zum Beispiel komplett ausgenutzter Hardwareresourcen) sehr heterogene und schwer reproduzierbare Werte entstehen. Im Rahmen dieser Evaluation wird eine solche Messung also nicht vorgenommen.

Alle drei genannten Kriterien, Erzeugungsdauer, Startdauer und Menge an Umgebungen pro konkreter Hardwareresource sind für die Problemstellung und Ausgangssituation bei der Pixelhouse GmbH von großer Bedeutung. Da aber grundsätzlich ja eine größtmögliche Parallelisierung der Testausführung angestrebt wird, ist die Menge der Umgebungen das wichtigste Kriterium. Es wird somit mit dem Faktor 10 gewichtet. Da sich auch die Erzeugungsdauer der Maschinenabbilder nur konstant auf die Ausführungsdauer aller Tests auswirkt, wird sie lediglich mit dem Faktor 4 gewichtet. Auch die Startdauer einer einzelnen Umgebung spielt eine geringere Rolle, da sie, wenn man von einer maximalen Parallelisierung ausgehen würde, die Gesamtausführungsdauer aller Tests ebenfalls nur konstant erhöht und nicht zum Beispiel linear oder sogar exponentiell. Dauert das Starten zum Beispiel 10 Minuten, würde die Gesamtausführungsdauer aller Tests sich eben um 10 Minuten erhöhen, wenn man alle Tests parallel ausführt. Da es aber fraglich ist, ob die Pixelhouse GmbH wirklich eine vollständige Parallelisierung aller Tests erreichen wird beziehungsweise bezahlen möchte, wird dieses Kriterium etwas höher, nämlich mit dem Faktor 7 gewichtet.

Neben Kriterien, die Einfluss auf die Ausführungsdauer haben, spielen für die Pixelhouse nun lediglich noch Kriterien eine Rolle, die die Verwendbarkeit des jeweiligen Ansatzes

kennzeichnen. Hier soll jeweils die bessere Lösung mit dem Wert 10 bewertet werden, die schlechtere Lösung mit 1. Die konkreten Kriterien sind dabei: Installationsaufwand, Einfachheit der Konfigurationssyntax (für Maschinen und Umgebungsdefinition) und Einfachheit der Umgebungssteuerung (Kommandozeilentools). Die Gewichtung sieht dabei so aus, dass der Installationsaufwand und die Einfacheit der Umgebungssteuer lediglich mit dem Faktor 3 gewichtet wird. Diese Aspekte spielen im Alltag der Entwicklung keine entscheidene Rolle, da die Installation nicht regelmäßig oder sogar automatisiert erfolgt und die Umgebungssteuerung im Falle der automatischen Testausführung ebenfalls programmatisch erfolgt und nicht händisch erfolgt. Die Einfachheit Konfigurationssyntax kann gerade für die Administratoren regelmäßig von Vorteil sein. Sie wird deshalb mit dem Faktor 5 gewichtet.

Die Kriterien und ihre Gewichtung werden in der Abbildung 22 noch einmal zusammengestellt.

Evaluationskriterium	Gewichtungsfaktor
Dauer der Erzeugung von Maschinen-Abbildern bzw. OS-Container-Images	4
Dauer des Startvorgangs der ersten Umgebung	7
Menge der parallelen Testumgebungen bei gegebener Hardware	10
Einfachheit der Installation der Virtualisierungslösung	3
Einfachheit der Umgebungssteuerung	3
Einfachheit der Konfigurationssyntax	5

Abbildung 22: Zusammenfassung Evaluationskriterien und Gewichtung

5.2.2 Ermittlung und Diskussion der Ergebnisse

Bei der Ermittlung der tatsächlich messbaren Kriterien ist zunächst interessant, auf welchem Testsystem die entsprechenden Zahlen ermittelt werden. Beim Testsystem handelt es sich um einen Laptop, der einen Intel Core i7 mit einer Taktung von 3 GHz und zwei Prozessorkernen enthält. Außerdem verfügt das System über 4 GB DDR3 Hauptspeicher mit einer Taktung von 1600 MHz. Außerdem verfügt das System über eine 120 Mbit Internetanbindung, was gerade beim Erzeugen der Maschinen-Abbilder beziehungsweise OS-Container-Images und der Installation von Softwarepaketen eine Rolle spielt. Bei der Festplatte handelt es sich um eine 256 GB großes Solid State Drive (SSD).

Sämtliche Messergebnisse befinden sich sowohl auf der beiliegenden CD im Ordner Evaluation als auch unter https://github.com/perprogramming/bachelorarbeit/tree/master/cd/Evaluation.

Die Erzeugung aller Maschinen-Abbilder für VirtualBox mit Hilfe des Tools Packer hat auf diesem System ca. 25 Minuten gedauert. Das Protokoll der Erzeugung kann auf der beiliegenden CD in der Datei Evaluation/build_virtualbox.log eingesehen werden. Die Erzeugung aller OS-Container-Images mit Docker hat auf dem gleichen Testsystem ca. 5 Minuten gedauert. Auch hier befindet sich das Protokoll auf der CD in der Datei Evaluation/build_docker.log. Setzt man also bei der Bewertung der Erzeugungsdauer für die längere Dauer von ca. 25 Minuten im Falle von Virtualbox und Packer den Wert 1 an, so ergibt sich daraus für die Dauer von 5 Minuten eine ungefähre Bewertung von 8.

Auch der Startvorgang der ersten Umgebung wurde für beide Ansätze auf dem zuvor genannten Testsystem durchgeführt. Die entsprechenden Protokolle befinden sich ebenfalls auf der CD im Ordner Evaluation und heißen start_virtualbox.log beziehungsweise start_docker.log. Das Starten der ersten Virtualbox-Umgebung mit Hilfe von Vagrant dauerte auf dem Testsystem ca. 4 Minuten. Das Starten der ersten Docker-Umgebung dauerte auf dem gleichen System 2 Sekunden. Bei einem Unterschied dieser Größenord-

nung kann man durchaus für die Bewertung der Startdauer für die längere Zeitspanne von 4 Minuten den Wert 1 und für den schnelleren Vorgang den Wert 10 ansetzen.

Beim Versuch, auf dem Testsystem möglichst viele Umgebungen zu starten, stellte sich vor allem der verbrauchte Arbeitsspeicher als obere Grenze heraus. So war es mit Virtualbox beziehungsweise Vagrant möglich, insgesamt 3 Umgebungen zu starten. Da in jeder Maschine der Umgebung ein vollständiges Betriebssystem läuft, war während der Tests ein zuverlässiges Starten der Maschinen nur gewährleistet, wenn man über Virtualbox jeder Maschine ein Minimum von 256 MB Arbeitsspeicher zuweist. Pro Umgebung muss Virtualbox somit ungefähr 1,2 GB Arbeitsspeicher reservieren. Das Starten der vierten Umgebung war somit nicht mehr möglich. Eine Ansicht der 3 gleichzeitig laufenden Umgebungen befindet sich auf der CD unter Evaluation/number_of_environments_virtualbox.log. Mit Docker war es auf dem oben beschriebenen Testsystem hingegen möglich insgesamt 20 Testumgebungen zu starten. Erst das Starten der 21 Umgebung endete mit der Fehlermeldung *Untar fork/exec /usr/local/bin/docker: cannot allocate memory.* Da bei der Virtualisierung mittels OS-Container kein zusätzliches Betriebssystem gestartet wird, beschränkt sich der Arbeitsspeicherverbrauch auf den der tatsächlich in den Containern laufenden Anwendungen. Der Gesamtbedarf an Arbeitsspeicher für eine vollständige Docker-Umgebung ist dabei bei lediglich ca. 200 MB. Eine Ansicht der 20 gleichzeitig laufenden Umgebungen ist auf der CD unter Evaluation/number_of_environments_docker.log. Bei der Bewertung kann der Lösung mit Docker also die beste Bewertung von 10 zugeordnet werden. Die Implementierung mit Virtualbox erhält ungefähr proportional dazu die Bewertung 2.

Bei der Bewertung der Einfachheit der Installation kann kein signifikanter Unterschied festgestellt werden. Bei beiden Ansätzen müssen mehrere Anwendungsprogramme installiert werden. Es lässt sich lediglich feststellen, dass im Falle von Docker beide Anwendungen (docker und docker-compose) vom gleichen Hersteller, nämlich der Docker Inc. entwickelt werden. Im Falle von Virtualbox sind mehrere Firmen beteiligt, nämlich Oracle für die Bereitstellung der Virtualisierung, die Firma HasiCorp für die Werkzeuge Packer und Vagrant und der Entwickler Adrien Thebo für das Vagrant-Plugin vagrant-hosts. Deshalb wird für dieses Kriterium eine Bewertung von 10 Punkten für Docker und eine Bewertung von 8 Punkten für die Virtualbox-Lösung vergeben.

Bei der Bewertung der Einfachheit der Bedienung lässt sich kein Unterschied feststellen. Bei beiden Lösungen lässt sich die Erzeugung der Maschinen-Abbilder und die Steuerung der Umgebungen leicht über Kommandozeilen-Tools ansprechen. Beide Lösungen erhalten hier die volle Punktzahl von 10 Punkten.

Bei der Bewertung der Einfachheit der Konfigurationssyntax gibt es allerdings wieder einen deutlichen Unterschied. So können bei der Lösung mit Docker sehr einfache Konfigurationsdateien geschrieben werden, die im Schnitt deutlich kürzer sind als die für die Lösung mit Virtualbox. So sind für die Definition eines Images mit Hilfe einer Dockerfile durchschnittlich 7 Zeilen notwendig. Bei der Definition der Maschinen-Abbild-Erzeugung für Packer sind hingegen jeweils ca. 32 Zeilen pro packer.json-Datei notwendig. Auch die Definition der Gesamtumgebung fällt bei docker-compose mit 21 Zeilen deutlich kürzer aus als die für Vagrant mit 74 Zeilen. Grundsätzlich handelt es sich bei der Syntax in der docker-compose.yml auch um eine Deklaration, wo hingegen die Syntax der Datei Vagrantfile um die Programmiersprache Ruby handelt. Man könnte somit mit dem Hinzufügen von Logik sogar eine kürzere Formulierung für die Vagrantfile ausarbeiten (zum Beispiel mit einer Schleife). Dies würde aber die Anforderungen für die Wartung dieser Datei entsprechend erhöhen. Insgesamt wird dieser Punkt also mit 10 Punkten für docker bewertet und mit nur 5 Punkten für den Ansatz mit Virtualbox.

In der Tabelle 23 ist das daraus resultierende Gesamtergebnis inkl. des Berechneten gewichtetet Mittelwertes zu sehen.

Insgesamt errechnet sich für die Lösung mit Hilfe von Virtualbox eine Punktzahl von 3,4375 von 10 Punkten. Der Ansatz mit Docker erhält eine Punktzahl von 9,75 von 10 möglichen Punkten. Anhand dieser Bewertung lässt sich also sagen, dass eine Virtualisierung mit Hilfe von OS-Containern unter Berücksichtigung der gewählten Kriterien und der vorgenommenen Gewichtung deutlich besser eignet als eine Lösung mit dem Hosted-Hypervisor Virtualbox.

Evaluationskriterium	Virtualbox	Docker	Gewichtung	Summe Virtualbox	Summe Docker
Dauer der Erzeugung	1	8	4	4	32
Dauer des Startvorgangs	1	10	7	7	70
Menge der parallelen Testumgebungen	2	10	10	20	100
Einfachheit der Installation	8	10	3	24	30
Einfachheit der Umgebungssteuerung	10	10	3	30	30
Einfachheit der Konfigurationssyntax	5	10	5	25	50
Summe			32	110	312
Mittelwert				3,4375	9,75

Abbildung 23: Zusammenfassung Evaluationsergebnisse

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde zunächst ein Überblick über die beiden Themenbereiche Softwaretests und Virtualsierung und deren aktuellen Stand der Forschung gegeben.

Im Bereich der Softwaretests wurden dabei die unterschiedlichen Testarten Unit-, Integrations-, System- und Akzeptanz-Tests erläutert und untersucht, dass hier vor allem die System- und Akzeptanz-Tests hohe Anforderungen an Testumgebung und die Ausführungsdauer von Tests stellen. Außerdem wurden verschiedene Strategien zur Testausführung besprochen und argumentiert, warum es sinnvoll sich für eine automatisierte Auführung der Tests zu entscheiden und welche Testarten hierbei mehr oder weniger intensiv verwendet werden sollen.

Im Bereich der Virtualisierung wurde zunächst eine grundsätzliche Kategorisierung zwischen der Systemvirtualisierung mittels Hypervisor und der Betriebssystemvirtualisierung mittels OS-Containern erarbeitet. Dabei wurden verschiedene Produkte und technische Lösungen beschrieben, ihre Vor- und Nachteile erarbeitet und abschließend vergleichend diskutiert.

Im darauf folgenden Kapitel „Konzeption“ wurden zunächst die aktuelle Produktiv- und die aktuellen Testumgebungen der Pixelhouse GmbH beschrieben und vor allem deren Nachteile erarbeitet. Anhand dieser Nachteile und mit Hilfe der im Grundlagen-Kapitel erarbeiteten Möglichkeiten der Virtualisierung wurde dann ein Konzept für eine neue Testumgebung erarbeitet. Hierbei wurde vorallem argumentiert, dass die aktuellen Probleme der Pixelhouse GmbH sich dadurch lösen lassen, dass mit Hilfe der Virtualisierung jede Testumgebung ein vollständiges Abbild der Produktivumgebung ist. Dadurch wird zum einen verhindert, dass verschiedene Tests sich beim parallellen Ausführen gegenseitig beeinträchtigen und zum anderen eröffnet es die Möglichkeit, zukünfig auch Änderungen an der Infrastruktur selbst zu testen.

Im anschließenden Kapitel Implementierung wurden zwei prototypische Umsetzungen, zum einen mit Hilfe des Hosted-Hypervisors Virtualbox von Oracle und zum anderen unter Verwendung der OS-Container-Virtualisierung Docker, vorgestellt. Hierbei wurden die verwendeten Softwarewerkzeuge beschrieben und Probleme beziehungsweise Besonderheiten der jeweiligen Implementierung aufgezeigt.

Im Kapitel „Evaluation“ wurde dann versucht, eine Evaluationsmethode zu definieren, die es der Pixelhouse GmbH ermöglicht, die beiden Lösungen in Hinblick auf die vorliegende Problemstellung und Ausgangslage zu bewerten. Dazu wurden zunächst bestimmte Evaluationskriterien definiert, beschrieben, wie sie sich ermitteln lassen und zusätzlich eine Gewichtung vorgenommen, da nicht jedes Kriterium für die Pixelhouse GmbH von gleicher Bedeutung ist. Anschließend wurde beschrieben, wie die jeweiligen Kriterien ermittelt wurden und die tatsächlichen Messwerte zu einem gewichteten Mittelwert verrechnet. Dabei ergab sich eine deutliche Tendenz für die Lösung mit Hilfe der OS-Container-Virtualisierung und dem Produkt Docker.

Dies passt positiverweise zu der Bereits von den System-Administratoren eingeschlagenen Richtung und deren Wunsch, die Produktivumgebung auf diese Virtualisierung umzustellen.

Für die Pixelhouse GmbH könnte es eine wichtige Fragestellung sein, inwieweit es möglich ist, die gleiche Betriebsumgebung auch für die Entwicklung der Anwendung auf den Laptops der Entwickler zu verwenden. Es wäre erstrebenswert, zukünftig zwischen allen drei Umgebungen, also Entwicklung, Testbetrieb und Produktion, so wenig abweichungen wie möglich zu haben.

7 Literaturverzeichnis

Chaubal 2015
Chaubal, Charu: *The Architecture of VMware ESXi*. http://www.vmware.com/files/pdf/ESXi_architecture.pdf. Version: 2015. – [Online; Stand 28. März 2015]

Chen 2014a
Chen, Qingqing: *Virtual Machine Monitor Type I*. http://commons.wikimedia.org/wiki/File:VMM-Type1.JPG. Version: 2014. – [Online; Stand 08. Dezember 2014]

Chen 2014b
Chen, Qingqing: *Virtual Machine Monitor Type II*. http://commons.wikimedia.org/wiki/File:VMM-Type2.JPG. Version: 2014. – [Online; Stand 08. Dezember 2014]

Chirammal 2014
Chirammal, Humble D.: *Xen and Kvm*. http://website-humblec.rhcloud.com/xen-and-kvm/. Version: 2014. – [Online; Stand 17. Dezember 2014]

Citrix Systems, Inc. 2015a
Citrix Systems, Inc.: *FAQ: XenMotion, Live Migration*. http://support.citrix.com/article/CTX115813. Version: 2015. – [Online; Stand 17. April 2015]

Citrix Systems, Inc. 2015b
Citrix Systems, Inc.: *Xen Server 6.5: Technical FAQ*. http://support.citrix.com/content/dam/supportWS/kA560000000Ts7qCAC/XenServer_6.5.0_Technical_FAQ.pdf. Version: 2015. – [Online; Stand 17. April 2015]

Claus und Schwill 2001
Claus, Volker ; Schwill, Andreas: *Duden Informatik*. Bibliographisches Institut, Berlin, 2001. – ISBN 3411052333

Clemson 2014
Clemson, Toby: *Testing Strategies in a Microservice Architecture*. http://martinfowler.com/articles/microservice-testing. Version: 2014. – [Online; Stand 12. April 2015]

Crisp 2011
Crisp, James: *Automated Testing and the Test Pyramid*. http://jamescrisp.org/2011/05/30/automated-testing-and-the-test-pyramid/. Version: 2011. – [Online; Stand 12. April 2015]

Damodaran et al. 2012
Damodaran, Deepak K. ; Mohan, Biju R. ; S., Vasudevan M. ; Naik, Dinesh: *Performance Evaluation of VMware and VirtualBox*. 2012

Docker Inc. 2014
Docker Inc.: *What is Docker?* https://www.docker.com/whatisdocker/. Version: 2014. – [Online; Stand 11. Dezember 2014]

Docker Inc. 2015a
Docker Inc.: *Compose yml - Docker Documentation*. https://docs.docker.com/compose/yml/#links. Version: 2015. – [Online; Stand 16. April 2015]

Docker Inc. 2015b
Docker Inc.: *Docker Compose - Docker Documentation*. https://docs.docker.com/compose/. Version: 2015. – [Online; Stand 19. April 2015]

Docker Inc. 2015c
Docker Inc.: *Dockerfile Reference*. https://docs.docker.com/reference/builder/. Version: 2015. – [Online; Stand 16. Januar 2015]

Docker Inc. 2015d
Docker Inc.: *Working with Docker Hub*. https://docs.docker.com/userguide/dockerrepos/. Version: 2015. – [Online; Stand 16. Januar 2015]

Duden 2015
Duden: *virtuell*. http://www.duden.de/rechtschreibung/virtuell. Version: 2015. – [Online; Stand 07. Januar 2015]

Duvall et al. 2007
Duvall, Paul M. ; Matyas, Steve ; Glover, Andrew: *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007. – ISBN 9780321336385

Francies 2014
Francies, Daniel: *Linux Containers: A Lightweight Replacement for Virtual Machines*. http://sdanielf.github.io/blog/2014/01/28/linux-containers-a-lightweight-replacement-for-virtual-machines/. Version: 2014. – [Online; Stand 11. Dezember 2014]

Goldberg 1973
Goldberg, Robert P.: *Architectural Principles for Virtual Computer Systems*. 1973

HashiCorp 2015a
HashiCorp: *Packer by HashiCorp*. https://packer.io/. Version: 2015. – [Online; Stand 14. Januar 2015]

HashiCorp 2015b
HashiCorp: *Templates: Builders*. https://packer.io/docs/templates/builders.html. Version: 2015. – [Online; Stand 14. Januar 2015]

HashiCorp 2015c
HashiCorp: *Templates: Post-Processors*. https://packer.io/docs/templates/post-processors.html. Version: 2015. – [Online; Stand 14. Januar 2015]

HashiCorp 2015d
HashiCorp: *Templates: Provisioners*. https://packer.io/docs/templates/provisioners.html. Version: 2015. – [Online; Stand 14. Januar 2015]

HashiCorp 2015e
HashiCorp: *Why Vagrant?* https://docs.vagrantup.com/v2/why-vagrant/index.html. Version: 2015. – [Online; Stand 14. Januar 2015]

Heise Zeitschriften Verlag 2014
Heise Zeitschriften Verlag: *Anwendungs-Container: Microsoft will Docker für Windows entwickeln*. http://www.heise.de/developer/meldung/Anwendungs-Container-Microsoft-will-Docker-fuer-Windows-entwickeln-2425205.html. Version: 2014. – [Online; Stand 27. Dezember 2014]

Hirschbach 2006
Hirschbach, Daniel: *Vergleich von Virtualisierungstechnologien*. Diplomarbeit an der Universität Trier, 2006

Humble und Farley 2010
Humble, Jez ; Farley, David: *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley, 2010. – ISBN 0321601912

Kivity et al. 2007
Kivity, Avi ; Kamay, Yaniv ; Laor, Dor ; Lublin, Uri ; Liguori, Anthony: *kvm: the Linux Virtual Machine Monitor*. 2007. – Proceedings of the Linux Symposium, Volume One, June 27th–30th, 2007

KVM 2015
KVM: *Migration*. `http://www.linux-kvm.org/page/Migration`. Version: 2015. – [Online; Stand 17. April 2015]

Madhavapeddy et al. 2013
Madhavapeddy, Anil ; Mortier, Richard ; Rotsos, Charalampos ; Scott, David ; Singh, Balraj ; Gazagnaire, Thomas ; Smith, Steven ; Hand, Steven ; Crowcroft, Jon: Unikernels: Library Operating Systems for the Cloud. In: *ASPLOS XVIII - Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013

MariaDB Foundation 2015
MariaDB Foundation: *Welcome to MariaDB!* `https://mariadb.org/`. Version: 2015. – [Online; Stand 17. April 2015]

NGINX, Inc. 2015
NGINX, Inc.: *NGINX - High Performance Load Balancer, Web Server, and Reverse Proxy*. `http://nginx.com/`. Version: 2015. – [Online; Stand 17. April 2015]

Oracle Corporation 2014
Oracle Corporation: *Oracle VM VirtualBox User Manual*. 2014 `http://dlc-cdn.sun.com/virtualbox/4.3.20/UserManual.pdf`. – [Online; Stand 14. Januar 2015]

Pixelhouse GmbH 2014
Pixelhouse GmbH: *Pixelhouse macht Chefkoch.de*. `http://pixelhouse.de`. Version: 2014. – [Online; Stand 20. Dezember 2014]

Popek und Goldberg 1974
Popek, Gerold J. ; Goldberg, Robert P.: *Formal requirements for virtualizable third generation architectures*. 1974. – Communications of the ACM, Volume 17, Nr. 7

Proxmox 2014
Proxmox: *Bare-metal ISO Installer - Proxmox VE*. `http://pve.proxmox.com/wiki/Bare-metal_ISO_Installer`. Version: 2014. – [Online; Stand 14. Dezember 2014]

Redis 2015
Redis: *Redis*. `http://redis.io/`. Version: 2015. – [Online; Stand 17. April 2015]

Scheepers 2014
Scheepers, Mathijs J.: *Virtualization and Containerization of Application Infrastructure: A Comparison*. 2014

Setty 2011
Setty, Sreekanth: *VMware vSphere vMotion - Architecture, Performance and Best Practices in VMware vSphere 5*. `http://www.vmware.com/files/pdf/vmotion-perf-vsphere5.pdf`. Version: 2011. – [Online; Stand 28. März 2015]

Sven 2014
Sven: *CPU ring scheme*. http://commons.wikimedia.org/wiki/File:CPU_ring_scheme.svg. Version: 2014. – [Online; Stand 08. Dezember 2014]

The Apache Software Foundation 2015a
The Apache Software Foundation: *Apache Solr*. http://lucene.apache.org/solr/. Version: 2015. – [Online; Stand 17. April 2015]

The Apache Software Foundation 2015b
The Apache Software Foundation: *Apache Tomcat - Welcome!* http://tomcat.apache.org/. Version: 2015. – [Online; Stand 17. April 2015]

The PHP Group 2015
The PHP Group: *PHP: Hypertext Preprocessor*. http://php.net/. Version: 2015. – [Online; Stand 17. April 2015]

Thebo 2015
Thebo, Adrien: *adrienthebo/vagrant-hosts*. https://github.com/adrienthebo/vagrant-hosts. Version: 2015. – [Online; Stand 18. April 2015]

Trefis 2014
Trefis: *Growing Competition For VMware In Virtualization Market*. http://www.nasdaq.com/article/growing-competition-for-vmware-in-virtualization-market-cm316783. Version: 2014. – [Online; Stand 08. Januar 2014]

Turnbull 2014
Turnbull, James: *The Docker Book: Containerization is the new virtualization*. 2014. – ISBN 9780988820234

Varnish Software 2015
Varnish Software: *Varnish Community | Varnish makes websites fly!* https://www.varnish-cache.org/. Version: 2015. – [Online; Stand 17. April 2015

Wintersteiger 2013
Wintersteiger, Andreas: *Scrum - Schnelleinstieg*. Entwickler.Press, 2013. – ISBN 9783868020922

Wottawa und Thierau 1990
Wottawa, H. ; Thierau, H.: *Lehrbuch Evaluation*. 1990

Zhang et al. 2010
Zhang, Qi ; Cheng, Lu ; Boutaba, Raouf: *Cloud computing: state-of-the-art and research challenges*. 2010