h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Hochschule Darmstadt
- Fachbereich Informatik -

Die Performanzunterschiede von nativem System, KVM, Docker und Docker in KVM anhand Atlassian Jira

Wissenschaftliche Arbeit

Seminar: Literaturrecherche und Theoriearbeit

vorgelegt von

Tim Stoffel
750116
Kirchstrasse 18, 64283 Darmstadt

Referent: Prof. Dr. Lars-Olof Burchard

Eidesstattliche Erklärung

Ich, Tim Stoffel, versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 12. März 2018

(Unterschrift Tim Stoffel)

Geheimhaltung

Diese Arbeit darf weder vollständig noch auszugsweise ohne schriftliche Zustimmung des Autors vervielfältigt, veröffentlicht oder Dritten zugänglich gemacht werden. Die abgegebene Arbeit wird vom Hauptreferenten unter Verschluss gehalten. Mit ist bekannt, dass die Geheimhaltung nicht die Erstellung eines Thesis-Posters sowie die Durchführung des Kolloqiums betrifft. Die Geheimhaltungspflicht erlischt automatisch nach 5 Jahren.

Darmstadt, den 12. März 2018

(Vollständige, handschriftliche Unterschrift)

Hochschule Darmstadt

Abstract

Fachbereich Informatik

Seminar: Literaturrecherche und Theoriearbeit

Die Performanzunterschiede von nativem System, KVM, Docker und Docker in KVM anhand Atlassian Jira

von Tim Stoffel

Containerlösungen sind in der Regel schneller bereitzustellen und einfacher zu administrieren. Virtuelle Maschinen haben im Vergleich einen Leistungsverlust, sind in der Handhabung aber leichter als ein natives System. Doch gleichen die beschleunigten Prozesse durch eine Containerinfrastruktur die erwarteten Performanzverluste aus?

Das Ziel der Arbeit ist es, die Performanz der Software Atlassian Jira auf verschieden Architekturen zu messen und zu vergleichen. Dafür wird Jira direkt auf einem Linux Computer, in einer KVM VM, in einem Docker Container und in einem Docker Container in einer KVM VM installiert und betrieben. Anschließend wird die Performanz der jeweiligen Testinstallation mit Python Skripten gemessen und ausgewertet.

Die Ergebnisse sollen Aufschluss darüber geben, mit wie viel Performanzgewinn oder -verlust bei einem Wechsel der Plattform zu rechnen ist. Daraus sollen Handlungsempfehlungen abgeleitet werden, ob Containerstrukturen gegenüber bestehenden VM Landschaften zu bevorzugen sind.

In der Untersuchung wird festgestellt, dass der Container in einer virtuellen Maschine genauso schnell ist wie die virtuelle Maschine selbst. Daher steht von Seiten der Leistungsfähigkeit einem Betrieb von einer Containerinfrastruktur in virtuellen Maschinen nichts im Wege.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die moderne Informationstechnik hat die Industrie und die Arbeitsweise in den letzten Jahrzehnten stark beeinflusst. Mit der Einführung der elektronischen Datenverarbeitung konnten viele Aufgaben von Computern übernommen werden, die vorher von Menschen ausgeführt wurden. Zu Beginn der Informationstechnik wurden Großrechner in Unternehmen eingesetzt, um gleiche Aufgaben abzuarbeiten. Großrechner sind dazu in der Lage, mehrere Hundert Benutzer gleichzeitig abzuwickeln [vgl. 1].

Um die Ressourcen eines solchen Großrechners besser ausnutzen zu können und verschiedene Systeme auf einer Hardware auszuführen, entwickelte IBM 1966 das erste Betriebssystem [vgl. 2, S. 485], das Virtualisierung (siehe 2.1 Begriffsklärung) unterstützte. Diese Technologie änderte die Arbeit der IT: Seitdem war es möglich mehrere Anwendungen isoliert voneinander auf einer Hardware auszuführen.

Als 2013 Docker angekündigt wurde [vgl. 3, S. 6], wurde neben der Virtualisierung eine weitere Möglichkeit bekannt um Anwendungen isoliert zu betreiben. Containervirtualisierung gab es mit LXC schon 2008 [vgl. 4, S. 8], Docker machte diese Technologie massentauglich. Container virtualisieren Anwendungen und minimieren dabei die Abstraktionsschicht zwischen echter Hardware und Anwendung.

Die drei Technologien natives System, Virtualisierung und Containervirtualisierung haben in ihren Anwendungsbereichen Überschneidungen. So ist jetzt bei dem Betrieb einer Anwendung die Entscheidung zu treffen, welche Technologie zu verwenden ist.

Im Praxisprojekt „ALM v2.0 - automatisierte Bereitstellung eines Atlassian ALM Clusters mit Container Orchestration und DevOps Praktiken“ werden die Anwendungen des Softwareherstellers Atlassian mithilfe von Containervirtualisierung betrieben. Die Container laufen auf Systemen, die selbst virtualisiert sind. Dabei stellt sich die Frage, inwieweit sich die Performanz der Anwendungen durch die Schachtelung von Virtualisierungstechnologien verändert.

Es existieren bereits Untersuchungen, die die Leistung von Containern mit virtuellen Maschinen [siehe 4–7] oder Containern mit einer nativen Ausführung [siehe 5–7] vergleichen. Bei der Literaturrecherche wurden keine Untersuchungen über die Performanz von Containern in virtuellen Maschinen gefunden. Auch gibt es wenige belastbare Zahlen für Anwendungen von Atlassian. Auf beides soll die Arbeit eine Antwort bieten.

Die vorliegende Arbeit zeigt anhand eigener Messungen, inwieweit sich die Performanz einer Anwendung ändert, wenn sie in verschiedenen Architekturen betrieben wird. Verglichen wird die Ausführung der Software Jira auf der Hardware direkt, in einer virtuellen Maschine, in einem Container und in einem Container auf einer virtuellen Maschine. Zusätzlich sollen die Messergebnisse mit den Ergebnissen aus entsprechender Literatur verglichen werden.

Die Ergebnisse sollen Rückschlüsse darüber geben, inwiefern sich die Performanz einer Anwendung zwischen den unterschiedlichen Architekturen unterscheidet. Zusätzlich zur Performanzanalyse vergleicht die Arbeit die bisherige Virtualisierung mit der Containervirtualisierung.

Im ersten Kapitel werden Begriffe definiert und es wird die Motivation für die Arbeit dargelegt. Im Technologievergleich werden virtuelle Maschinen und Container miteinander verglichen und die Performanzuntersuchungen aus der Literatur zusammengefasst. Der Benchmark beschreibt den Messaufbau und die Ergebnisse der Messung. Abschließend wird in der Auswertung geprüft, inwieweit diese Ergebnisse Einfluss auf den Betrieb einer Containerinfrastruktur in virtuellen Maschinen haben.

Kapitel 2

Grundlagen

2.1 Begriffsklärung

In der weiteren Arbeit werden die folgenden Begriffe verwendet und hier definiert:

Virtualisierung ist eine Technologie, bei der die Ressourcen eines Computers oder Servers durch eine Softwareschicht zusammengefasst und weitergegeben werden. Dies ermöglicht eine bessere Ausnutzung der Ressourcen des Computers oder Servers [vgl. 8, S. 231]. Virtualisierung ermöglicht so die Ausführung von verschiedenen Betriebssystemen oder Anwendungen auf derselben Hardware. Der Server mit der realen Hardware wird Host genannt, die Systeme, die auf ihm laufen, Gäste.

Die Softwareschicht zwischen Host und Gast kann in zwei Kategorien eingeteilt werden:

Bei Typ 1 oder Bare Metal [vgl. 9, Folie 12] wird auf der Hardware ein Hypervisor installiert. Der Hypervisor ist ein minimales Betriebssystem. Beim Start des Hosts wird der Hypervisor hochgefahren. Dieser stellt nur Schnittstellen für den Betrieb und die Kontrolle der virtuellen Gastsysteme bereit. Dabei werden die Gastsysteme isoliert voneinander betrieben. Diese laufen in Ring 1 und sind daher nicht in der Lage, privilegierte Anweisungen auf dem Hostsystem ausführen [vgl. 10, S. 201].

Hosts mit Typ 2 oder Hosted Virtualisierung [vgl. 9, Folie 13] booten ein Betriebssystem. Dieses Betriebssystem lädt ein Kernel Modul, virtuellen Maschinenmonitor (VMM). VMM stellt dann die nötigen Schnittstellen für die Gastsysteme bereit. Die Gastsysteme laufen dabei in gering privilegierten Ringen, der VMM auf Ring 0 [vgl. 10, S. 200]. Dadurch sind die Gastsysteme wie bei Typ 1 nicht in der Lage privilegierte Anweisungen auf dem Hostsystem auszuführen.

Eine **Virtuelle Maschine**, kurz **VM**, wird in einem Hypervisor oder in einer VMM erzeugt. Die VMs auf einem Host sind abgeschottet und wissen nicht voneinander. Das Betriebssystem oder die Anwendungen auf dem System bemerken nicht, dass sie virtualisiert sind [vgl. 8, S. 231].

Kernel-based Virtual Machine, kurz **KVM**, ist eine Visualisierungstechnologie vom Typ 2 und seit 2007 Teil des Linuxkernel [vgl. 4, S. 8]. Der Kernel lädt beim Starten das benötigte Kernelmodul nach und ermöglicht so den Betrieb von nahezu jedem Gastsystem. Wenn das Gastsystem zu QEMU (Quirk-Emulator) kompatibel ist, kann KVM das Gastsystem mit sehr geringen Performanzverlusten ausführen- Wenn das System nicht kompatibel ist, emuliert QEMU die Hardware. Diese Emulation ist deutlich langsamer [vgl. 4, S. 8].

Container sind vergleichbar mit virtuellen Maschinen: Auch hier werden Computersysteme gekapselt auf einem System betrieben. Die Softwareschicht zwischen realem Computer und der Anwendung, die isoliert ausgeführt wird, wurde gegenüber einer VM deutlich reduziert.

Baun definiert Containervirtualisierung im Artikel Servervirtualisierung folgendermaßen: „Hier laufen unter ein und demselben Betriebssystemkern mehrere voneinander abgeschottete identische Systemumgebungen. Es wird kein zusätzliches Betriebssystem, sondern eine isolierte Laufzeitumgebung virtuell in einem geschlossenen Container erzeugt." [10, S. 203]

Ein Container wird immer aus einem Containerabbild gestartet, das einer Vorlage entspricht. Damit ist, eine entsprechende Vorlage vorausgesetzt, eine Anwendung in einem Container in wenigen Minuten betriebsbereit. Die Containerabbilder liegen in einem Standardformat vor. Dieses Format ist auf fast allen Plattformen und Architekturen gleich. Damit werden die Abhängigkeiten zur Ausführung eines Containers minimiert. Ein Container lässt sich, wie sein Vorbild in der Realität, einfach umziehen und fast überall starten [vgl. 3, S. 13].

Docker ist eine Entwicklerplattform für Containervirtualisierung und wurde 2013 veröffentlicht [vgl. 3, S. 6]. Der Fokus von Docker liegt auf der Bereitstellung von Applikationen. Zunächst lief Docker nur auf Unix Systemen, ist nun aber auch mit Windows kompatibel. Die einzige Abhängigkeit der Applikationen bildet die Wahl zwischen einer Linux oder Windows Basis für den Container. Ein Linux-Docker-Container läuft so nur auf einem Linuxbetriebssystem mit Docker und ein Windows-Docker-Container nur auf einem Windowsbetriebssystem mit Docker (Stand 26.02.2018).

Jira ist ein Softwareprodukt des australischen Softwarehersteller Atlassian. Diese Software ist ein Ticketsystem, das in der Software- oder Produktentwicklung eingesetzt wird. Es unterstützt agile Methoden wie Scrum oder Kanban, aber auch klassische Methoden wie das Wasserfallmodell [vgl. 11].

2.2 Motivation zur eigenen Messung

Die in Abschnitt 3.2 zusammengefassten Arbeiten zeigen die Leistungsunterschiede von Docker, KVM und einem nativen System. In der Praxis besteht selten die Möglichkeit, seine Container auf einem „echten" Server auszuführen. Server der gängigen Cloud Anbieter sind fast immer virtuell und auch innerhalb des Unternehmens :em AG, bei welchem die praktische Untersuchung für diese Arbeit stattfand, werden Server nur virtuell an Mitarbeiter weitergegeben.

Daher muss Docker hier innerhalb einer VM ausgeführt werden. Dabei liegen zwei Abstraktionsschichten übereinander; die der VM und die der Containerplattform. Inwieweit sich diese weitere Schicht negativ oder positiv auf die Leistung einer Anwendung auswirkt, soll im nächsten Kapitel untersucht werden.

Bis auf Felter untersucht keine der betrachteten Arbeiten die Performanz einer echten Anwendung. Die Ergebnisse eines Benchmarks können sich aber von einer Anwendung unterscheiden. In „An Updated Performance Comparison of Virtual Machines and Linux Containers" untersucht Felter die Leistung von Redis [vgl. 6, S. 8-9] und MySQL [vgl. 6, S. 9-11] auf Docker, KVM und einem nativen System.

Die vorliegende Arbeit soll eine solche Betrachtung mit der Anwendung Jira ergänzen. Die :em AG ist Atlassian Solution Partner und vertreibt daher Atlassian Produkte, darunter auch Jira. Jira ist mitunter das bekannteste Ticket-System für agile Entwicklungsmethoden (im Speziellen SCRUM) und hat so für die :em AG eine wichtige Bedeutung. Die Untersuchung dieser Java-Web-Applikation soll feststellen, wie hoch der erwarte Geschwindigkeitsverlust von Jira in einem Container innerhalb einer VM ist.

Kapitel 3

Technologievergleich

3.1 Virtuelle Maschine und Container im Detail

Für den Betrieb einer Anwendung stehen verschiedene Architekturen zur Auswahl: Die wichtigsten sind der Betrieb direkt auf der Hardware, virtuelle Maschinen und Container. Aus Sicherheitsgesichtspunkten ist es wichtig, Applikationen im Betrieb voneinander zu trennen [vgl. 12]. Wird eine Applikation von einem Angreifer kompromittiert, sind davon die anderen Applikationen nicht beeinflusst.

Dieses Prinzip bedeutet für den nativen Betrieb, dass für jede Applikation ein eigener Server beschafft und bereitgestellt werden muss. Die Bereitstellung einer neuen Applikation wird dadurch deutlich verlangsamt. Jeder dieser Server muss dabei so konfiguriert sein, dass auch bei Lastspitzen akzeptable Antwortzeiten gewährleistet sind.

3.1.1 Virtuelle Maschinen

Wenn Anwendungen in virtuellen Maschinen betrieben werden, teilen sich mehrerer VMs einen Server. Ein neuer Server muss nur beschafft werden, wenn die Auslastungsgrenze erreicht ist. Da die Anwendungen in den VMs häufig zu unterschiedlichen Zeiten unter Last sind, wird so eine bessere Auslastung erzeugt, als bei einem nativen Betrieb. Dadurch ergibt sich bei einem virtualisierten Betrieb eine Kostenreduktion. Im Artikel Servervirtualisierung wird davon ausgegangen, dass „(..) die Investitionen in neue Hard- und Software um bis zu 70% sinken können und im Rechenzentrum sind Kosteneinsparungen von bis zu 50% erreichbar.“ [10, S. 198]

In einer virtuellen Maschine läuft ein Betriebssystem, die Anwendung und die Bibliotheken, die für die Ausführung dieser Anwendung notwendig sind (Bins/Libs).

Zusätzlich lassen sich VMs zwischen Servern leichter umziehen, als Anwendungen auf einem Server. Auch bieten moderne Visualisierungssysteme die Möglichkeit für Snapshots, also das Einfrieren des aktuellen Zustands einer VM, um später auf diesen Zeitpunkt zurückspringen zu können [vgl. 10, S. 198]. Weiter ist die Sicherung einer VM einfacher als das Backup eines Servers, da die Beschreibung der Hardware bei der Sicherung einer VM mitgespeichert wird. Diese Beschreibung wird bei einer Wiederherstellung berücksichtigt. Einen baugleichen Server zu beschaffen, auf den die Vollsicherung eines nativen Server zurückgespielt werden kann, ist in der Praxis nicht immer einfach.

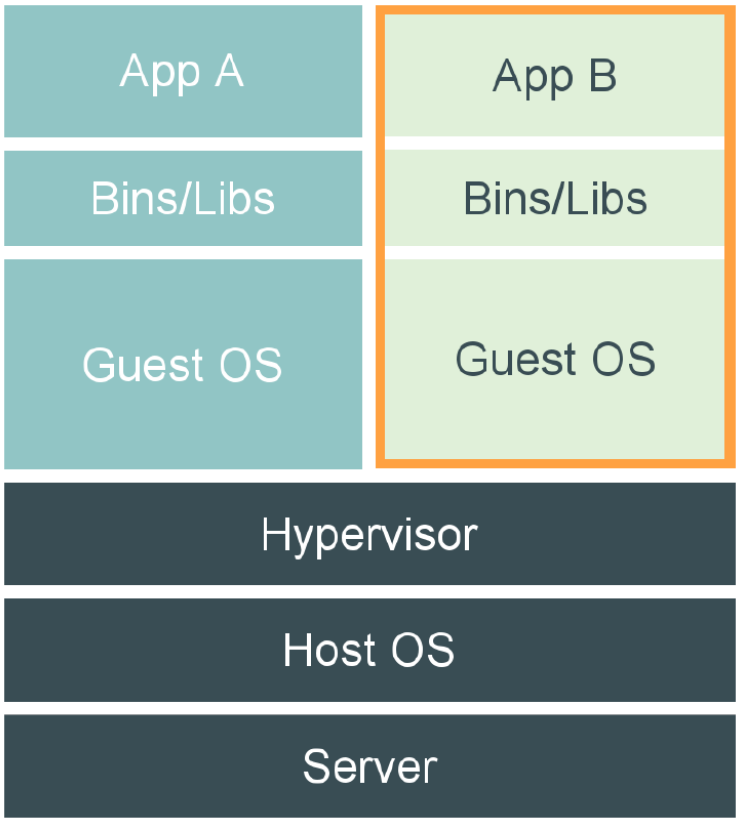

ABBILDUNG 1: Aufbau Virtualisierung mit Hypervisor [3, S. 6]

Ein weiterer Vorteil der Virtualisierung ist die Möglichkeit, virtuelle Maschinen zwischen verschieden Hosts verschieben zu können. Bei einigen Virtualisierungslösungen ist das fast ohne Ausfallzeit möglich. Die Folien von Bufolo zeigen, dass die Nichtverfügbarkeit bei unter vier Sekunden liegt [vgl. 9, Folie 19]. So können VMs auf einen anderen Server verschoben werden, ohne dass der Nutzer etwas davon merkt. Dies ermöglicht es, einen Server ohne Ausfallzeiten zu warten. Viele moderne Virtualisierungssysteme gehen einen Schritt weiter und können über diesen Mechanismus eine VM hochverfügbar machen: Fällt ein Server aus, wird die VM weiter auf einem anderen Server des Clusters betrieben [vgl. 10, S. 198].

Neben den genannten Vorteilen birgt die Nutzung von Virtualisierungstechnologie auch Nachteile: Da die Zugriffe der Gastsysteme auf die eigentliche Hardware durch einen Hypervisor oder einen virtuellen Maschinenmonitor durchgereicht werden, entstehen Verluste. Baun spricht dabei von 5% bis 10% Einbußen [vgl. 10, S. 198]. Durch den Einsatz von Multi-Kern-Systemen und die ständige Optimierung der Software sinken diese Verluste [vgl. 10, S. 198].

Manche Anwendungen benötigen spezielle Hardware die nicht virtualisierbar ist. Diese Anwendungen sind dann nicht für Virtualisierung geeignet [vgl. 10, S. 198].

Die Trennung von Applikationen durch VMs ist unsicherer als durch getrennte Hardware: Immer wieder gelingt es Angreifern Sicherheitslücken in der Abstraktionsschicht zwischen Gast und Host auszunutzen, um privilegierte Befehle auszuführen und so den Host und andere VMs zu kompromittieren [vgl. 13–15]. Auch gegen Sicherheitslücken, die die Trennung von User- und Kernelspace aushebeln, schützen

virtuelle Maschinen nicht. Aktuelle Beispiele für solche Lücken sind Spectre und Meltdown [vgl. 16, 17].

Auch führt der Ausfall eines Servers gleich zum Ausfall mehrere VMs, welches sich aber durch ein Cluster mit Hochverfügbarkeit verhindern lässt. Die Daten müssen dabei auch redundant gespeichert werden.

3.1.2 Container

Werden Anwendungen in Containern betrieben, wird als Plattform häufig Docker gewählt. In der weiteren Betrachtung wird bei der Verwendung von Containervirtualisierung von Docker als Basistechnologie ausgegangen.

Beim Betrieb von Anwendungen in Docker Containern sinken die Infrastrukturkosten gegenüber VMs, da viele Redundanzen wegfallen. Docker nennt in einem Beispiel eine Reduktion um 66% [vgl. 18].

Container werden über ihr Image gestartet und sind schneller einsatzbereit als VMs. Peichert kommt in seiner Untersuchung zu dem Schluss, dass die Startzeit eines Docker Containers etwa sechsmal schneller ist [vgl. 4, S. 21].

Ein Container speichert Daten nicht persistent, daher sind diese nach einem Containerneustart gelöscht. Um Daten zu speichern gibt es Volumes. Schlotter definiert Volumes in seiner Arbeit: „Ein Data Volume wird dafür genutzt Daten eines Containers zu speichern oder zwischen Containern zu teilen. Dazu wird ein Ordner, der auf dem Host-System bereit steht, an den Container weitergereicht und eingebunden. Falls bereits Daten beim Start des Containers in diesem Ordner vorhanden sind, werden diese in den Ordner, bzw. das Volume kopiert.“ [3, S. 15]

Das Image beschreibt den Zustand der Anwendung und ihrer Abhängigkeiten. Änderungen sind nur zur Laufzeit möglich. Dieses Image lässt sich weitergeben oder in eine Registry hochladen, um es so zu Verfügung zu stellen. Die Registry gilt als Repository, in dem alle Images vorgehalten werden.

Die Unveränderlichkeit des Images hat verschiedene Auswirkungen: Bei der Inbetriebnahme eines Docker Containers aus dem gleichen Image wird immer das gleiche Ergebnis entstehen. Damit wird ein Release reproduzierbar. Auch werden alle Änderungen, die zur Laufzeit stattfinden, genau im Container gespeichert. Diese Änderungen lassen sich anzeigen - so lassen sich ungeplante Änderungen oder Angriffe einfach nachvollziehen.

Die Definition einer Applikation über ein Image hat noch weitere Vorteile: So lassen sich mit Codescannern alle Komponenten und Bibliotheken in einem Image auf Sicherheitslücken prüfen, indem die Versionen mit CVE Listen verglichen werden. CVE lässt sich wie folgt definieren: „Common Vulnerabilities and Exposures, kurz CVE, ist ein Verzeichnis von Namen für öffentlich bekannte Sicherheitslücken (engl.: Vulnerabilities) und Sicherheitsrisiken (engl.: Exposures).“ [19, S. 4]
Wird eine Sicherheitslücke erkannt, kann die Registry die Verwendung des Images sperren. So wird eine potentiell unsichere Anwendung gar nicht in Betrieb genommen. Eine verbreitete Anwendung dafür ist „clair“. Diese kam auch in „A Study of Security Vulnerabilities on Docker Hub“ zum Einsatz [vgl. 20].

Container sind aufgrund ihres Konzeptes nicht so sicher wie VMs. Häufig ist es möglich Informationen über andere Container auf dem gleichen System zu erfahren.

Genauso wie VMs sind auch Container nicht vor Sicherheitslücken geschützt, die die Trennung von User- und Kernelspace aushebeln. Ein Beispiel sind die bereits angesprochenen Lücken Spectre und Meltdown [vgl. 16, 17].

Durch die einfache Erstellung eines Containerimages und dessen Inbetriebnahme entstehen neue Angriffsszenarien: In „A Study of Security Vulnerabilities on Docker Hub“ kommen die Autoren zu dem Ergebnis, dass über 80% der Images auf Docker-Hub, der offiziellen Registry für Docker Images, hohe Sicherheitslücken aufweisen [vgl. 20, S. 277]. Häufig werden diese Container gestartet, ohne dass diese zuvor auf Lücken geprüft worden sind.

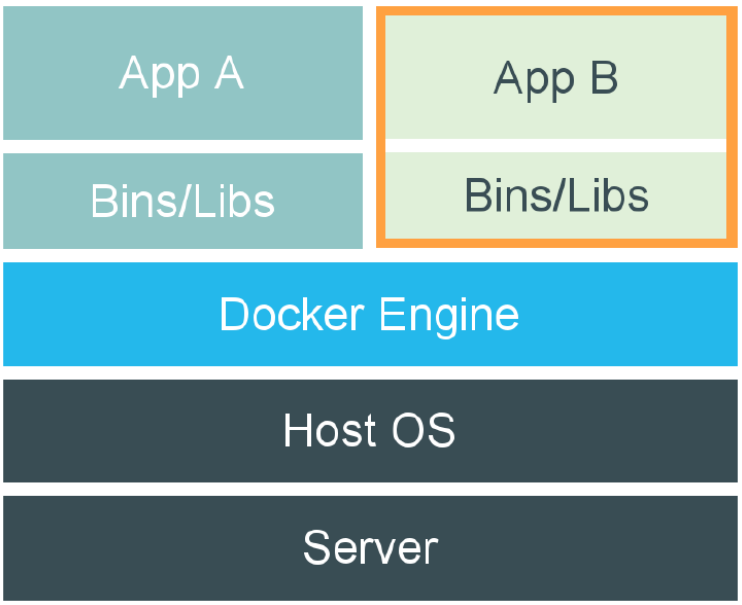

ABBILDUNG 2: Aufbau Containervirtualisierung [3, S. 7]

3.2 Ergebnisse der Literatur

Morabito und seine Kollegen vergleichen die Leistung von KVM, LXC, Docker, OSv und einer nativen Ausführung. In der Prozessor- und Arbeitsspeicherperformanz treten keine deutlichen Unterschiede auf.
Bei der Festplatten-Leistung liegen die Containervirtualisierungen Docker und LXC bei einem Test für den Datendurchsatz vor KVM und fast gleichauf mit dem nativen System. Beim zufälligen Lesen und Schreiben liegen Docker und LXC 10% bis 43% hinter dem nativen System und KVM bis zu 93%. Die Untersuchung der Netzwerkleistung zeigt, dass die Containervirtualisierung bei einem TCP Stream mit dem nativen System gleichauf ist; KVM und OSv verlieren leicht über 25%. Der Test mit einem UDP Stream zeigt, dass alle Systeme über 40% gegenüber der nativen Plattform verlieren. [vgl. 5, 5 ff]

In seiner Arbeit wertet Peichert die Leistung von Docker und KVM aus. Für seine VMs benötigt KVM doppelt so viel Speicher wie Docker bei der gleichen Anzahl an Containern. Die Startzeiten eines Containers sind fast sechs mal kürzer, als die einer VM. Bei der Berechnung von Fakultäten ist Docker etwa 5% schneller als KVM [vgl. 4, S. 21]. Weiter zitiert Peichert die Ergebnisse von Felter aus der Arbeit „An Updated Performanz Comparison of Virtual Machines and Linux Containers “ [6].

Felter vergleicht in „An Updated Performanz Comparison of Virtual Machines and Linux Containers“ die Leistung von Docker, KVM und einem nativen System: Er kommt zu dem Schluss, dass der Unterschied zwischen Docker und dem nativen System bei der CPU-Geschwindigkeit und dem Durchsatz beim zufälligen Lesen der Festplatte vernachlässigbar ist. KVM ist dabei fast immer um die Hälfte langsamer [vgl. 6, 4, 8 ff]. Keine bedeutenden Unterschiede gab es bei den Messungen zu Arbeitsspeicher und TCP-Leistung [vgl. 6, 6 f].

In „Performance Evaluation of Container-based Virtualization for High Performance Computing Environments“ untersuchen Xavier und seine Kollegen die Geschwindigkeit von LXC, OpenVZ, VServer, Xen und der nativen Ausführung. Im CPU-Test liegen alle Systeme gleichauf. In der Betrachtung der Arbeitsspeicher Leistung liegt Xen mit ca. 30% hinter den anderen Kandidaten. Bei der Schreibgeschwindigkeit liegen das native System, LCX und VServer an der Spitze, danach folgen OpenVZ und dann Xen. Die Lesegeschwindigkeit ist bei allen Systemen außer Xen sehr ähnlich. Xen liegt etwa 50% zurück. Xen liegt im Netzwerktest weiter hinten, die Unterschiede zwischen den anderen Systemen sind minimal. [vgl. 7, 4 f]

3.2.1 Zusammenfassung der Untersuchungen

Zusammenfassend kann die Aussage getroffen werden, dass sich im Bereich der CPU-Leistung ein natives System, Container und virtuelle Maschinen nicht nennenswert unterscheiden.

Bei den Untersuchungen zum Arbeitsspeicher kommen die Autoren zu dem Schluss, dass auch hier keine Unterschiede auftreten. Nur Xavier stellt fest, dass Xen fast ein Drittel [vgl. 7, S. 4] hinter den anderen Systemen liegt.

Je nach Test sind die Unterschiede bei der Festplattengeschwindigkeit zwischen Containern und dem nativen System vernachlässigbar oder bis zu 43% langsamer [vgl. 5, S. 7]. Virtuelle Maschinen sind aber klar langsamer als Container. Peichert erklärt dies in seiner Arbeit so: „(..) was in den zusätzlichen Zwischenschichten von KVM durch Hypervisor und übereinandergelegte Dateisysteme durch Gast- und Wirt-Festplatte [*Anm. d. Verf. - Wirt entspricht Host*] begründet ist. Docker überträgt die Daten per AUFS-Overlay direkt auf den Wirt und verwendet dabei das gleiche Dateisystem, wie beispielsweise ext4. Mit KVM dahingehen sind Festplattenänderungen des Gast-Dateisystems zur qcow2-Datei in das Wirt-Dateisystem hin zu übertragen.“ [4, S. 21]

In den Netzwerktest zu TCP ist die Geschwindigkeit von Containern und dem nativen System nahezu identisch, die virtuellen Maschinen sind bis zu 25% [vgl. 5, S. 8] schlechter. Bei UDP gibt es noch klare Unterschiede zwischen Virtualisierung und einem nativen System: Das native System ist ca. 40% schneller [vgl. 5, S. 8].

Kapitel 4

Benchmark

4.1 Messaufbau

4.1.1 Hardware

Ziel des Messaufbaus ist es, möglichst verlässliche Ergebnisse zu erzeugen. Dazu wurden alle Messungen auf der selben Hardware ausgeführt. Für die Ausführung der Messungen wurde ein weitere Rechner verwendet. Beide Rechner waren über einen eigenen Switch miteinander verbunden. Getrennte Hardware und ein eigener Switch sollten externe Einflüsse so weit wie möglich reduzieren.

Der Rechner, auf dem Jira läuft, hat vier CPU-Kerne, acht Gigabyte RAM und eine SSD.

4.1.2 Software

Jira, die Datenbank und alle anderen Abhängigkeiten wurden mit Ansible installiert und konfiguriert. Ansible ist ein Werkzeug, um Computer deklarativ und automatisiert zu administrieren. Dabei wird der Ansatz verfolgt, dass die Infrastruktur als Code definiert ist. Dadurch lässt sich jedes zu messende System jederzeit erneut im gleichen Zustand bereitstellen, um die Messungen wiederholen zu können.

In dieser Arbeit werden das native System, KVM, Docker und Docker in KVM verglichen. Die Basis bildete immer ein Debian in Minimalausstattung mit SSH Server. Als Datenbank für Jira wurde PostgreSQL verwendet. Die Datenbank wurde als Programm installiert, in den beiden Docker Umgebungen als Container verwendet. Die Programmversionen können der nachfolgenden Tabelle entnommen werden:

Software	Debian	QEMU-KVM	Docker	Jira	PostgreSQL
Version	9.3.0	2.8	17.12.1-ce	7.7.1	9.6

TABELLE 1: Überblick verwendete Softwareversionen

Die JVM für Jira hatte immer ein Maximum von zwei Gigabyte. Der Arbeitsspeicher war im Test nie vollständig belegt, die Maschinen lagerten daher keine Inhalte des Speichers auf die Festplatte aus. Für die KVM VM wurden alle Kerne an die VM weitergereicht und der Arbeitsspeicher auf sechs Gigabyte beschränkt. Dieser Speicher wurde nie voll ausgenutzt.

4.1.3 Skript

Die Messungen wurden über ein eigens entwickeltes Python Skript vorgenommen. Der Softwarehersteller hatte in der Vergangenheit eigene Tools bereitgestellt um die Leistung von Jira zu messen, doch leider wurden diese seit Jahren nicht mehr gepflegt und waren zu aktuellen Versionen nicht mehr kompatibel.

Das Skript befüllte das Jira dabei mit Daten. Die Daten wurden vorher mit der Python Bibliothek Faker erzeugt und als JSON abgelegt. So wurde bei jedem Testlauf die gleiche Datenbasis erzeugt und die Ergebnisse sind vergleichbar. Die Kommunikation zwischen dem Skript und Jira fand über die REST API statt. Für diese Schnittstelle wurden die Python Bibliotheken Requests und jira-python verwendet.

Der Benchmark durchlief mehrere Schritte:

Zuerst wurden fünf Projekte erzeugt. Jedes Ticket musste in Jira einem Projekt zugewiesen sein. Die Projekterzeugung war also Voraussetzung für den nächsten Schritt.

Im zweiten Schritt wurden die zuvor generierten Tickets in Jira erstellt. Dazu wurde die JSON Datei eingelesen und die einzelnen Tickets via REST API angelegt. Dabei wurde die Dauer für jedes Ticket einzeln gespeichert. Einige Beispiele für diese Vorlagen sind Anhang A zu entnehmen. Die Anzahl der Tickets konnte beim Skript Aufruf festgelegt werden.

Der nächste Schritt erzeugte nochmals Tickets. Die Anzahl war dabei zehn Prozent der ursprünglichen Ticketanzahl. An diese Tickets wurde jeweils noch ein PDF angehängt. Dabei wurde eines von 250 PDFs ausgewählt und hochgeladen. Die Größe der PDFs variiert zwischen wenigen Kilobyte bis zehn Megabyte. Auch hier wurde die Dauer für jedes Ticket einzeln gespeichert.

Anschließend wurde die Neuindizerung der Plattform angestoßen. Ein aktueller Index ist für eine performante Suche notwendig. Diese Operation belastet stark die CPU und die Festplatte.

Im letzten Schritt wurden Suchanfragen erzeugt und die Antworten entgegengenommen. Die Suchanfragen wurden auch vorher erzeugt und werden bei jedem Durchlauf des Benchmarks aus einer JSON Datei ausgelesen. Das Limit für die Anzahl der Treffer wurde auf 10.000 gesetzt. So konnte die Antwort auf eine Suchanfrage einige Sekunden dauern. Die Antwortdauer wurde einzeln gespeichert.

Die Dauer für jeden beschrieben Schritt wurde gespeichert. So kann die Durchlaufzeit für jede einzelne Plattform verglichen werden. Das Resultat kann in Anhang C betrachtet werden.

Das Skript kann keinen vollständigen Lasttest abbilden, da es nur von einem Client ausgeführt wird. Die Anfragen werden sequenziell an den Server übermittelt. Das Jira auf dem Zielsystem ist z.B. bei der Ticketerstellung nur auf einem CPU-Kern ausgelastet. Andere Schritte wie die Neuindexierung verwenden aber alle verfügbaren Ressourcen des Systems.

Das Skript kann in Anhang B eingesehen werden. Zur besseren Übersicht wurden alle Programmzeilen, die Log Einträge generieren, entfernt.

4.2 Messergebnisse

Die vier Plattformen wurden wie in 4.1.2 beschrieben nacheinander installiert und konfiguriert. Danach erfolgte immer eine Messung und die Umgebung wurde zurückgesetzt. Dieser Vorgang wurde dreimal wiederholt und die Messergebnisse gemittelt. Der Benchmark erzeugte dabei jeweils 5.000 Tickets, sowie 500 Tickets mit Anhängen und 500 Suchanfragen.

In der Summe lag Docker mit dem nativen System fast gleichauf, KVM und Docker in KVM waren etwa 16% langsamer.

	Natives System	**KVM**	**Docker**	**KVM-Docker**
Summe [min]	18,06	21,00	18,40	20,98
Abweichung zum nat. Sys.	0,00%	16,28%	1,91%	16,20%

TABELLE 2: Gesamtdauer des Benchmarks im Vergleich

Im Detail lässt sich kein eindeutiger Trend feststellen.
So war das Anlegen von Tickets in KVM ca 10% langsamer als in Docker, den Suchindex erzeugte die VM aber schneller als der Container. Bei der Suche lag die VM aber 15% hinter Docker.

KVM-Docker war in fast allen Schritten langsamer als KVM, nur bei den Suchanfragen war es ca. 7% schneller. Mögliche Grunde sind 5.1 zu entnehmen.

Die genannten Zahlen sind in Anhang C detailliert dargelegt.

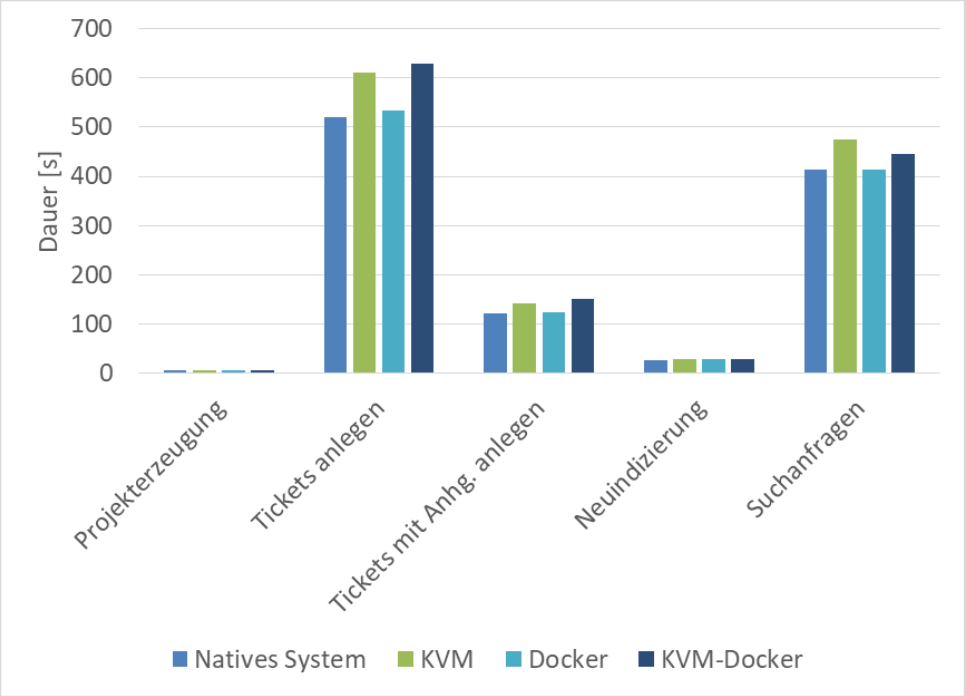

ABBILDUNG 3: Übersicht der Benchmark Schritte

Die Ergebnisse decken sich in vielen Bereichen mit den Erkenntnissen aus 3.2. Docker war fast genauso schnell wie das native System. KVM war etwas über 15% langsamer. Neu ist das Ergebnis, dass Container in einer VM fast genauso schnell sind wie die VM.

Kapitel 5

Auswertung

5.1 Interpretation der Messergebnisse

Die Messungen ergeben, dass der Container in der VM in Summe schneller ist als die VM alleine. Wenn die Ergebnisse im Detail (Anhang C) betrachtet werden, wird klar, dass Docker in KVM nur bei der Suchanfrage schneller ist. Bei der Suche in Jira wird der Index stark beansprucht. Dieser Index ist eine Datei auf der Festplatte.

Um den Grund für die fast 7% Vorsprung zu finden, wurde eine weitere Messung vorgenommen: Mit dem Linux Tool dd werden normalerweise blockweise Daten kopiert, es lassen sich damit aber auch Geschwindigkeitstest durchführen. Mit den folgenden Kommandos wurde die Leistung der Festplatte in den vier Architekturen gemessen.

```
$ dd if=/dev/zero of=/root/testfile bs=1G count=1 oflag=direct
```

LISTING 5.1: Lesen und Schreiben von einem Gigabyte

```
$ dd if=/dev/zero of=/root/testfile bs=512 count=1000 oflag=direct
```

LISTING 5.2: Lesen und Schreiben von 1000 x 512 Byte

Das erste Kommando prüft den Durchsatz von großen Daten, das Zweite gibt eine Aussage über das Ansprechverhalten des Datenträgers.

Die Befehle wurden nach einem Neustart auf einem leeren System Zehnmal ausgeführt und der Durchschnitt gebildet. Basis bildete immer ein Debian 9.3.

	Natives System	**Docker**	**KVM**	**KVM-Docker**
Übertragung 1GB	241,00 MB/s	248,75 MB/s	242,75 MB/s	261,75 MB/s
	0,00%	3,22%	0,73%	8,61%
1000 mal 512 Byte	14,40 MB/s	14,18 MB/s	5,40 MB/s	5,38 MB/s
	0%	-1,53%	-62,5%	-62,67%

TABELLE 3: Messung der Festplattengeschwindigkeit mit dd

Die Ergebnisse zeigen, dass Docker beim Lesen und Schreiben von einem Gigabyte immer schneller ist. Gerade Docker in KVM ist deutlich schneller als das native System. Die Vermutung liegt nahe, dass Docker einen effizienteren Zwischenspeicher hat als alle anderen betrachteten Systeme.

Das könnte der Grund sein, warum die Suchanfragen im Container in einer virtuellen Maschine so viel schneller waren als die VM allein. Die Index-Datei wird scheinbar besser zwischengespeichert.

Bei der Latenzmessung ist Docker etwas langsamer. Das lässt sich dadurch erklären, dass bei der Virtualisierung jedes Lesen und Schreiben durch die Abstraktionsschichten abgehandelt werden muss. Werden viele kleine Dateien gelesen und geschrieben, wächst dieser Overhead und bremst die gesamte Operation aus. Dies kann dann auch nicht durch effizientes Zwischenspeichern ausgeglichen werden. Dennoch ist Docker hier, wie auch schon in 3.2 belegt, schneller als KVM.

5.2 Auswirkungen auf das Praxisprojekt

Für den Betrieb einer Anwendung im Rahmen des Praxisprojektes ist es wichtig, dass diese isoliert betrieben wird. Damit scheidet der Betrieb der Applikationen auf einzelnen Server aus, denn dies wäre zu teuer und auch der Arbeitsaufwand der Bereitstellung einer Serverhardware ist zu hoch.

Aus diesem Grund ist nur der Betrieb einer Infrastruktur mit VMs oder Containern praxistauglich. Die Messergebnisse zeigen, dass es aus Performanzsicht fast irrelevant ist, ob die Anwendung in einer VM oder in einem Container in einer VM betrieben werden. In beiden Fällen gibt es über 15 % Verlust der Leistung. Neben der Praxistauglichkeit sind auch die Vorteile im Bereich der Bereitstellung und Orchestrierung von Containern oder VMs einem nativen System vorzuziehen.

Das Praxisprojekt verwendet ein Containercluster auf Basis von VMs für die Bereitstellung und den Betrieb der Atlassian Applikationen und bietet so verschiedene Vorteile gegenüber der Verwendung von virtuellen Maschinen: Die Bereitstellungsdauer eines Containers ist deutlich kürzer als die einer VM. Mit Containern lassen sich noch mehr Applikationen gleichzeitig auf einem Server betreiben. Auch erzeugt die Prüfung der Docker-Images auf Sicherheitslücken und die anschließende Sperrung beim Fund einer schweren Lücke eine gute Grundsicherheit.

Da die Performanzunterschiede zwischen KVM und KVM-Docker so gering sind und der Betrieb von Containern in diesem Anwendungsfall einfacher ist, wird diese Infrastruktur in Zukunft auch weiter mit Containern in virtuellen Maschinen betrieben.

In der Zukunft wäre es denkbar, Containercluster direkt auf den Servern laufen zu lassen, um die Verlustleistungen von 16% auf 2% zu verringern.

Kapitel 6

Fazit

6.1 Zusammenfassung

Sowohl für den Betrieb von Containern als auch von virtuellen Maschinen gibt es gute Gründe. Letztendlich muss in jedem Anwendungsfall entschieden werden, welche der beiden Technologien besser geeignet ist.

Die Ergebnisse der Messung decken sich in weiten Teilen mit der Literatur und sind somit belastbar. Hinzu gekommen ist die Betrachtung über die Leistung von Containern in einer virtuellen Maschine. Dass die Anwendung in einem Container in einer VM genauso schnell ist wie die Anwendung in einer virtuellen Maschine war nicht zu erwarten, da eine Abstraktionsschicht hinzukommt.

Die Virtualisierung mit Container ist fast genauso schnell wie die Ausführung auf dem nativen System und deutlich schneller als eine Lösung mit virtuellen Maschinen. Ist also die Möglichkeit gegeben, Container statt VMs zu verwenden, sind Container aus Performanzgründen vorzuziehen.

6.2 Ausblick

Die Arbeit hat die Werkzeuge geschaffen um jederzeit die Messungen zu wiederholen. Es ist denkbar, den Prozess der Anwendungsbereitstellung, der Messung und des Zurücksetzens vollständig zu automatisieren.

Dann ist es einfach möglich Messreihen abzuprüfen, also nicht nur 5000 Tickets, sondern 10, 100, 1.000, 10.000 und 100.000. Dabei soll untersucht werden, ob sich die Dauer proportional verändert und inwieweit sich die Dauer der Suche bei einer gleichbleibenden Zahl an Suchanfragen verändert.

Weiter kann genauer untersucht werden, warum Docker in KVM schneller sein kann als KVM allein.

Das Skript speichert schon die Dauer der Erzeugung eines einzelnen Tickets oder einer einzelnen Suchanfrage. Hier kann analysiert werden, inwieweit sich die Dauer im Bezug auf die Anzahl der Tickets im System ändert. Auch können Teilschritte des Benchmarks auf verschiedenen Systemen parallel auf dem Zielsystem ausgeführt werden, um so mehrere Clients zu simulieren.

Denkbar ist auch die Erweiterung des Benchmarks um andere Anwendungen. Dadurch könnte die Aussage über die Geschwindigkeitsunterschiede präzisiert werden.

Anhang A

Beispieldaten für Tickets

```
{
  "fields":{
    "project":{
      "key":"PJK5"
    },
    "summary":"The best fast play.",
    "description":"Site drop public Mrs in. Magazine long actually
       marriage.Decade accept natural spend everyone hear. Must together
       eight line choose many. Growth walk spring deal eye coach anyone.
       Throw particularly bag part. Expert computer recently radio like
       indicate avoid. ",
    "issuetype":{
      "name":"Bug"
    }
  }
},
{
  "fields":{
    "project":{
      "key":"PJK1"
    },
    "summary":"Major choose degree nothing.",
    "description":"Should into information glass someone quality method.
       And dream area bed prove.Relate drug true reach ability the single
       service. Eat former eight career like main. Rate drive team another
        determine area power.Senior make major let.Beyond interest exactly
        big senior. Might yard summer. Nor political by student behind
       task.Door research check major must. Meet since condition.",
    "issuetype":{
      "name":"Story"
    }
  }
},
{
  "fields":{
    "project":{
      "key":"PJK5"
    },
    "summary":"Memory national land.",
    "description":"Task run specific door value organization everybody.
       Boy back early control left.",
    "issuetype":{
      "name":"Story"
    }
  }
}
```

Anhang B

Benchmark Skript

```
import requests
from requests.auth import HTTPBasicAuth
from requests_toolbelt import (MultipartEncoder, MultipartEncoderMonitor)
import json
import copy
import logging
import datetime
from jira import JIRA
import csv
import sys, getopt
import os
import locale

locale.setlocale(locale.LC_ALL, '')

baseurl = "http://172.18.100.141:8080/jira/"
user = "admin"
password = "admin"
session = requests.Session()
session.auth = HTTPBasicAuth(user, password)

jira = JIRA(baseurl, basic_auth=(user, password))

def projectKeyExists(key):
    url = "{base}rest/api/2/project".format(base=baseurl)
    response = session.get(url)
    dict = json.loads(response.text)

    for item in dict:
        if item.get('key', 'None')==key :
            return True

    return False

def generateProjects(max):
    projecttemplate={   "key": "EX2",
                        "name": "Example2",
                        "projectTypeKey": "software",
                        "projectTemplateKey": "com.pyxis.greenhopper.jira:
                            gh-kanban-template",
                        "description": "Example Project description",
                        "lead": "admin",
                        "url": "http://em.ag",
                        "assigneeType": "PROJECT_LEAD"}

    url = "{base}rest/api/2/project/".format(base=baseurl)
    headers = {'Content-Type': "application/json"}

    i = 1
```

```
49      while i <= max:
50          project = copy.deepcopy(projecttemplate)
51          projektkey = "PJK{number}".format(number=i)
52
53          if not projectKeyExists(projektkey):
54              project["key"]= projektkey
55              project["name"]= "Projekt{number}".format(number=i)
56              response = session.post(url, data=json.dumps(project), headers
                    =headers)
57
58          i+= 1
59
60  def createIssue(issue):
61      url = "{base}rest/api/2/issue/".format(base=baseurl)
62      headers = {'Content-Type': "application/json"}
63      response = session.post(url, data=json.dumps(issue), headers=headers)
64
65
66  def createAttachmentIssues(issuefields, path):
67      issue = jira.create_issue(fields=issuefields)
68      jira.add_attachment(issue=issue.key, attachment=path)
69
70  def readIssueJSON(path):
71      with open(path) as json_file:
72          issues = json.load(json_file)
73          return issues
74
75  def main():
76      if len(sys.argv) == 1:
77          issuecount = 5000
78           # 10% issues with attachment
79          issueattcount = issuecount *10 /100
80      elif len(sys.argv) == 2:
81          issuecount = sys.argv[1]
82          # 10% issues with attachment
83          issueattcount = issuecount *10 /100
84      else:
85          issuecount = sys.argv[1]
86          issueattcount = sys.argv[2]
87
88      if issuecount > 100000:
89          issuecount = 100000
90
91      if issueattcount > 100000:
92          issueattcount = 100000
93
94      querymax = 500
95
96      # create CSV file for benchmark
97      filename = "benchmark_i{ic}_a{ac}.csv".format(ic = issuecount, ac =
            issueattcount)
98      ofile  = open(filename, "w", newline='')
99      benchmarkwriter = csv.writer(ofile, delimiter = ';', quotechar = '"',
            doublequote = True, skipinitialspace = False, lineterminator = '\r\
            n')
100
101     ####################
102     # generate Projects
103     ####################
104     timestart = datetime.datetime.now()
105     generateProjects(5)
106     timeend = datetime.datetime.now()
107     timedelta = timeend - timestart
```

```
108     row = ['projectcreation', locale.format('%.4f', timedelta.
            total_seconds(), True)]
109     benchmarkwriter.writerow(row)
110
111     ####################
112     # generate sample Issues
113     ####################
114     filename = "benchmark_issues{ic}.csv".format(ic = issuecount)
115     ofileissue  = open(filename, "w", newline='')
116     issuebenchmarkwriter = csv.writer(ofileissue, delimiter = ';',
            quotechar = '"', doublequote = True, skipinitialspace = False,
            lineterminator = '\r\n')
117     row = ['Issue-count', 'time']
118     issuebenchmarkwriter.writerow(row)
119     timestart = datetime.datetime.now()
120     issues = readIssueJSON("data100k.json")
121
122     counter = 1
123     for issue in issues:
124         if counter > issuecount:
125             break
126
127         timestart2 = datetime.datetime.now()
128         createIssue(issue)
129         timeend2 = datetime.datetime.now()
130         timedelta2 = timeend2 - timestart2
131         row = [counter, locale.format('%.4f', timedelta2.total_seconds(),
                True)]
132         issuebenchmarkwriter.writerow(row)
133         counter+= 1
134
135     ofileissue.close()
136     timeend = datetime.datetime.now()
137     timedelta = timeend - timestart
138     row = ['issuecreation', locale.format('%.4f', timedelta.total_seconds
            (), True)]
139     benchmarkwriter.writerow(row)
140
141     ####################
142     # generate sample issues with attachments
143     ####################
144     filename = "benchmark_attachments{ic}.csv".format(ic = issueattcount)
145     ofileattach  = open(filename, "w", newline='')
146     attachbenchmarkwriter = csv.writer(ofileattach, delimiter = ';',
            quotechar = '"', doublequote = True, skipinitialspace = False,
            lineterminator = '\r\n')
147     row = ['issue_count', 'time', 'filesize', 'delta']
148     attachbenchmarkwriter.writerow(row)
149     timestart = datetime.datetime.now()
150     issues = readIssueJSON("data_attachment_100k.json")
151
152     counter = 1
153     for issue in issues:
154         if counter > issueattcount:
155             break
156         attachmentid = (counter%250)+1
157         attachment = "samples/{id}.pdf".format(id = attachmentid)
158         timestart2 = datetime.datetime.now()
159         createAttachmentIssues(issue['fields'], attachment)
160         timeend2 = datetime.datetime.now()
161         timedelta2 = timeend2 - timestart2
162         filesize = os.path.getsize(attachment)
```

```
163         row = [counter, locale.format('%.4f', timedelta2.total_seconds(),
                True), locale.format('%.4f', filesize, True), locale.format('
                %.4f', filesize/timedelta2.total_seconds(), True)]
164         attachbenchmarkwriter.writerow(row)
165         counter+= 1
166
167     ofileattach.close()
168     timeend = datetime.datetime.now()
169     timedelta = timeend - timestart
170     row = ['issueattachmentcreation', locale.format('%.4f', timedelta.
            total_seconds(), True)]
171     benchmarkwriter.writerow(row)
172
173     ####################
174     # Reindex
175     ####################
176     timestart = datetime.datetime.now()
177     url = "{base}rest/api/2/reindex?type=FOREGROUND".format(base=baseurl)
178     response = session.post(url, timeout=10)
179     url = "{base}rest/api/2/reindex/progress".format(base=baseurl)
180     response = session.get(url, timeout=10)
181     dict = json.loads(response.text)
182
183     while dict['currentProgress'] < 100:
184         response = session.get(url, timeout=10)
185         dict = json.loads(response.text)
186
187     timeend = datetime.datetime.now()
188     timedelta = timeend - timestart
189     row = ['reindex', locale.format('%.4f', timedelta.total_seconds(),
            True)]
190     benchmarkwriter.writerow(row)
191
192     ####################
193     # Query
194     ####################
195     filename = "benchmark_query{ic}.csv".format(ic = querymax)
196     ofilequery  = open(filename, "w", newline='')
197     querychbenchmarkwriter = csv.writer(ofilequery, delimiter = ';',
            quotechar = '"', doublequote = True, skipinitialspace = False,
            lineterminator = '\r\n')
198     row = ['query_counter', 'time', 'result_count']
199     querychbenchmarkwriter.writerow(row)
200     timestart = datetime.datetime.now()
201     queries = readIssueJSON("query_strings_100k.json")
202
203     counter = 1
204     for querystring in queries:
205         if counter > querymax:
206             break
207         timestart2 = datetime.datetime.now()
208         jql = 'text ~ "{text}" order by created desc'.format(text=
                querystring)
209         result = jira.search_issues(jql, maxResults=10000)
210         timeend2 = datetime.datetime.now()
211         timedelta2 = timeend2 - timestart2
212         row = [counter, locale.format('%.4f', timedelta2.total_seconds(),
                True), result.total]
213         querychbenchmarkwriter.writerow(row)
214         counter+= 1
215
216     ofilequery.close()
217     timeend = datetime.datetime.now()
```

```
218     timedelta = timeend - timestart
219     row = ['queries', locale.format('%.4f', timedelta.total_seconds(),
            True)]
220     benchmarkwriter.writerow(row))
221
222     ofile.close()
223
224 if __name__ == '__main__':
225     main()
```

Anhang C

Benchmark Ergebnisse

Aufgabe	Natives System	KVM	Docker	KVM-Docker
Projekterzeugung [s]	4,29	5,32	4,51	5,43
Abweichung zum nat. Sys.	*0,00%*	*23,85%*	*4,93%*	*26,55%*
Tickets anlegen [s]	519,80	610,88	534,15	628,44
Abweichung zum nat. Sys.	*0,00%*	*17,52%*	*2,76%*	*20,90%*
Tickets mit Anhg. anlegen [s]	120,68	140,56	123,39	149,85
Abweichung zum nat. Sys.	*0,00%*	*16,47%*	*2,24%*	*24,17%*
Neuindizierung [s]	25,26	27,48	27,65	28,03
Abweichung zum nat. Sys.	*0,00%*	*8,80%*	*9,48%*	*10,98%*
Suchanfragen [s]	413,46	474,96	414,30	446,48
Abweichung zum nat. Sys.	*0,00%*	*14,87%*	*0,20%*	*7,99%*
Summe [min]	18,06	21,00	18,40	20,98
Abweichung zum nat. Sys.	*0,00%*	*16,28%*	*1,91%*	*16,20%*

TABELLE 4: Dauer der Schritte des Benchmark-Skripts im Vergleich

Literatur

[1] R. Lackes und M. Siepermann. *Mainframe*. Website. URL: http://wirtschaftslexikon.gabler.de/Archiv/77338/mainframe-v11.html (besucht am 26. 02. 2018).

[2] R. J. Creasy. „The Origin of the VM/370 Time-Sharing System“. In: *IBM Journal of Research and Development* 25.5 (1981), S. 483–490. ISSN: 0018-8646. DOI: 10.1147/rd.255.0483. URL: http://pages.cs.wisc.edu/~stjones/proj/vm_reading/ibmrd2505M.pdf (besucht am 20. 02. 2018).

[3] C. Schlotter, M. Schmid und J. Becker. „Die Entwicklungsplattform Docker“. In: (2015). URL: http://image.informatik.htw-aalen.de/Thierauf/Seminar/Ausarbeitungen-15SS/docker.pdf (besucht am 26. 02. 2018).

[4] A. Peichert. *Sicherer Betrieb existierender Applikationen im Unternehmensumfeld mit Open Source-Werkzeugen*. 2015. URL: http://www.andreas.peichert.com/files/peichert_master_thesis_Sicherer_Betrieb_existierender_Applikationen_im_Unternehmensumfeld_mit_Open_Source-Werkzeugen.pdf (besucht am 22. 01. 2018).

[5] R. Morabito, J. Kjallman und M. Komu. „Hypervisors vs. Lightweight Virtualization: A Performance Comparison“. In: *2015 IEEE International Conference on Cloud Engineering (IC2E)*. Piscataway, NJ: IEEE, 2015, S. 386–393. ISBN: 978-1-4799-8218-9. DOI: 10.1109/IC2E.2015.74. (Besucht am 05. 03. 2018).

[6] W. Felter u. a. *An Updated Performance Comparison of Virtual Machines and Linux Containers*. URL: https://pdfs.semanticscholar.org/0d9a/ea55a54ccc6ab64995d70bf6ae464af25f0d.pdf (besucht am 26. 02. 2018).

[7] M. G. Xavier u. a. „Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments“. In: *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2013*. Hrsg. von Peter Kilpatrick. Piscataway, NJ: IEEE, 2013, S. 233–240. ISBN: 978-1-4673-5321-2. DOI: 10.1109/PDP.2013.41. (Besucht am 06. 03. 2018).

[8] C. Baun. *Betriebssysteme kompakt*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. ISBN: 978-3-662-53142-6. DOI: 10.1007/978-3-662-53143-3. URL: https://link.springer.com/chapter/10.1007/978-3-662-53143-3_10 (besucht am 21. 02. 2018).

[9] G. Bufolo. *Virtualisierung: Betrachtung aktueller Hypervisor wie Xen, KVM und Hyper-V*. 2010. URL: http://www4.cs.fau.de/Lehre/SS10/HS_AKSS/slides/01_Vortrag_Guilherme_Bufolo.pdf (besucht am 18. 02. 2018).

[10] C. Baun, M. Kunze und T. Ludwig. „Servervirtualisierung“. In: *Informatik-Spektrum* 32.3 (2009), S. 197–205. ISSN: 0170-6012. DOI: 10.1007/s00287-008-0321-6. URL: https://link.springer.com/article/10.1007/s00287-008-0321-6 (besucht am 18. 02. 2018).

[11] *Jira*. Website. URL: https://de.atlassian.com/software/jira (besucht am 27. 02. 2018).

[12] S. Strobel. *Wie virtuelle Maschinen die Sicherheit erhöhen können*. Website. 12. 12. 2008. URL: https://www.computerwoche.de/a/wie-virtuelle-maschinen-die-sicherheit-erhoehen-koennen,1881568,2 (besucht am 27. 02. 2018).

[13] B. Kraft. *Hacker brechen aus virtueller Maschine aus*. Website. 18. 03. 2017. URL: https://heise.de/-3658416 (besucht am 27. 02. 2018).

[14] P. Siering. *VM-Ausbruch möglich: Patches für Hypervisor Xen*. Website. 06. 05. 2017. URL: https://heise.de/-3704682 (besucht am 27. 02. 2018).

[15] F. A. Scherschel. *VMware-Admins aufgepasst: Es gibt wichtige Updates für ESXi*. Website. 07. 06. 2017. URL: https://heise.de/-3736872 (besucht am 27. 02. 2018).

[16] P. Bonzini und E. Habkost. *QEMU and the Spectre and Meltdown attacks*. Website. 04. 01. 2018. URL: https://www.qemu.org/2018/01/04/spectre/ (besucht am 27. 02. 2018).

[17] J. Horn. *Reading privileged memory with a side-channel*. Website. 03. 01. 2018. URL: https://googleprojectzero.blogspot.de/2018/01/reading-privileged-memory-with-side.html (besucht am 27. 02. 2018).

[18] B. Junod. *MetLife Uses Docker Enterprise Edition to Self Fund Containerization*. Website. 27. 10. 2017. URL: https://googleprojectzero.blogspot.de/2018/01/reading-privileged-memory-with-side.html (besucht am 27. 02. 2018).

[19] M. Scherf. *Software Vulnerability Tool: Entwicklung einer Anwendung zur Analyse des CVE-Verzeichnisses - Vergleichende Analyse der Standards CVE, CPE, CVSS und CWE*. 15. 12. 2015. URL: http://public.hochschule-trier.de/~knorr/DACH2016/BA-Scherf-Software%20Vulnerability%20Tool.pdf (besucht am 03. 03. 2018).

[20] R. Shu, X. Gu und W. Enck. „A Study of Security Vulnerabilities on Docker Hub“. In: *CODASPY'17*. Hrsg. von Gail-Joon Ahn, Alexander Pretschner und Gabriel Ghinita. New York, New York: The Association for Computing Machinery, 2017, S. 269–280. ISBN: 9781450345231. DOI: 10.1145/3029806.3029832. (Besucht am 06. 03. 2018).