Berufsakademie Sachsen
Staatliche Studienakademie Leipzig

Testen von Benutzeroberflächen in virtualisierten Umgebungen

Praxisarbeit Studiengang Informatik

Eine Projektarbeit von:
Marcel Schmied

Datum: 10. Oktober 2023
Aktualisiert: 2. Januar 2024

Inhaltsverzeichnis

1. Einleitung

Die Softwareentwicklung in der heutigen Zeit ist geprägt von einem stetigen Streben nach Effizienz und Leistungssteigerung. In diesem Entwicklungsprozess haben sich virtuelle Maschinen (VMs) heutzutage als eines der unverzichtbaren Werkzeuge etabliert, um Anwendungen in virtualisierten Umgebungen auszuführen und zu testen. Doch seit mehr als einer Dekade gibt es eine Alternative, die sich mehr und mehr durchsetzt und die Landschaft der virtualisierung grundlegend verändern könnte: Container. [4]

Diese schlanke und leichtgewichtige virtuelle Umgebung ermöglicht es, Anwendungen effizient zu verpacken und auszuführen, ohne den Overhead, der oft mit virtuellen Maschinenverbunden ist. [10] Doch es stellt sich die Frage: Bietet die Container Umgebung wirklich eine performante Alternative zur bewährten virtuellen Maschine? Hat diese Entscheidung Auswirkungen auf die Effizienz und Leistungsfähigkeit von Testumgebungen?

Das Ziel dieser Arbeit ist es, einen ausführlichen Vergleich zwischen virtuellen Maschinen und Container Umgebungen durchzuführen, insbesondere im Zusammenhang mit Webtesting. Wir werden die Testdurchführung, Laufzeiten und Performance-Metriken sorgfältig statistisch analysieren, um Erkenntnisse zu gewinnen, welche der beiden Virtualisierungstechnologien die performantere für das Testen von Webapplikationen ist. Dabei setzen wir nicht nur auf theoretische Überlegungen, sondern auch auf praktische Experimente, um die Realitätsnähe unserer Untersuchung zu gewährleisten.

Im folgenden Abschnitt werden die Grundlagen zum Thema Virtualisierung und automatisiertem Testen erläutert, um einen Einblick in die Thematik zu gewinnen. Anschließend werden wir die erhobenen Daten detailliert statistisch analysieren, um eine ausführliche Schlussfolgerung ziehen zu können, auf die Frage: Container Umgebung oder VM, was ist die beste Umgebung zum Webtesten von Benutzeroberflächen.

2. Grundlagen

2.1 Virtualisierung

2.1.1 Grundlegende Konzepte

Bei der Virtualisierung handelt es sich um einen zentralen Bestandteil des Cloud-Computings, welches auf der Bereitstellung von Informationstechnologie (IT)-Ressourcen in virtualisierter Form beruht. Diese Technologie ermöglicht die Abstraktion physischer IT-Ressourcen wie Hardware, Software, Speicher und Netzwerkkomponenten. Dabei entstehen virtuelle oder logische Komponenten, die in ihrer Verwendung den physischen Gegenstücken gleichen.

Ein wesentlicher Vorteil dieser Technologie liegt in der Schaffung einer Abstraktionsebene zwischen den physischen Ressourcen und ihren virtuellen Repräsentationen. Diese Abstraktionsschicht ermöglicht eine flexible und bedarfsgerechte Bereitstellung der Ressourcen an unterschiedliche Nutzer, was wiederum zu einer verbesserten Auslastung der IT-Infrastruktur führt.

Es ist wichtig, die Virtualisierung von ähnlichen Konzepten wie Simulation und Emulation abzugrenzen. Während Simulatoren und Emulatoren ein softwarebasiertes Modell eines Computersystems implementieren, um Inkompatibilitäten zu überbrücken, zielt die Virtualisierung darauf ab, IT-Ressourcen unabhängig von ihrer physischen Grundlage bereitzustellen. Dabei ist es das Ziel, möglichst wenig zu simulieren oder zu emulieren, um Performance-Verluste zu minimieren.

Der Begriff "Virtualisierung"wurde in den 1960er Jahren geprägt und bezog sich zunächst auf das Konzept der Hardwarevirtualisierung, welches die Erstellung virtueller Maschinen ermöglichte. Heutzutage sind verschiedene Formen der Virtualisierung in modernen IT-Landschaften anzutreffen, darunter Hardware-Virtualisierung, Software-Virtualisierung, Speicher-Virtualisierung, Daten-Virtualisierung und Netzwerk-Virtualisierung. Diese bieten vielfältige Möglichkeiten zur effizienten Nutzung von IT-Ressourcen in virtualisierten Umgebungen. [9]

2.1.2 Hardwarevirtualisierung

In der IT bezieht sich der Begriff der Virtualisierung im Allgemeinen meist auf die Hardwarevirtualisierung. Dabei werden physikalische Computer zu virtuellen Umgebungen, wie virtuellen Maschinen, abstrahiert. Diese virtuellen Umgebungen entstehen aus Softwarekomponenten, die beispielsweise einen Prozessor nachbilden, den Zugriff auf virtuellen Speicher oder eine virtuelle Festplatte ermöglichen. Auf einem realen Rechner können mehrere dieser virtuellen Umgebungen gleichzeitig laufen, wobei sie sich die physikalischen Ressourcen teilen. Ein Betriebssystem, das in einer virtuellen Umgebung ausgeführt wird, ist sich dessen in der Regel nicht

bewusst und muss auch nicht angepasst werden – es glaubt also, die Hardware des Computers exklusiv nutzen zu können. [19]

2.1.3 Vor- und Nachteile

Vorteile:

Virtualisierung ermöglicht eine verbesserte Ressourcennutzung durch die Zusammenführung von physischen Servern zu virtuellen Umgebungen auf einem leistungsstarken Server. Dadurch und durch die Möglichkeit, auf einem leistungsfähigen Host-System sowohl Windows, Linux oder andere Umgebungen einzurichten, ohne zusätzliche Investitionen in zusätzliche Hardware tätigen zu müssen, wird der Energieverbrauch reduziert und die Notwendigkeit für umfangreiche Wartung verringert. Des Weiteren führt sie zu kleineren Rechenzentren, da mehrere virtuelle Umgebungen auf einem einzigen physischen Host betrieben werden können.

Des Weiteren führt die Virtualisierung zu einer erhöhten Sicherheit und geringeren Ausfallzeiten. Da virtuelle Maschinen in isolierten Umgebungen laufen, sind sie weitgehend unabhängig voneinander. Das bedeutet, dass ein Ausfall oder eine Sicherheitsverletzung in einer virtuellen Umgebung die anderen nicht beeinträchtigt. Zudem gibt es die Möglichkeit, mit einem einzigen Systembackup eine ganze Reihe virtueller Systeme zu sichern, eine erhebliche Vereinfachung der Datensicherung und erhöht somit die Betriebssicherheit. [12]

Nachteile:

Trotz der zahlreichen Vorteile gibt es auch einige Nachteile zu beachten. Eines der zentralen Probleme entsteht in Situationen, in denen spezialisierte Software spezielle Hardwarelösungen erfordert. Ein Beispiel hierfür wäre die Steuerung von Sensoren und Automaten in der Produktion, wo zahlreiche Schnittstellen benötigt werden. In solchen Fällen kann die Virtualisierung an ihre Grenzen stoßen und spezielle Hardwareanforderungen nicht erfüllen. [12] Des Weiteren ist es wichtig zu bedenken, dass trotz der umfassenden Virtualisierung Desktop-Rechner nach wie vor notwendig sind. Sie sind unverzichtbar für Arbeitsplätze oder für die Einrichtung und Verwaltung virtueller Maschinen. Zudem können grafik- und speicherintensive Programme in virtualisierten Umgebungen oft nicht mit der gleichen Effizienz ausgeführt werden wie auf physischen Maschinen. Diese Herausforderungen müssen bei der Implementierung von Virtualisierungstechnologien berücksichtigt werden. Trotzdem überwiegen die Vorteile und bieten erhebliche Chancen zur Effizienzsteigerung und Kosteneinsparung in der IT-Infrastruktur. [11]

2.1.4 Vergleich - VMs und Container-Umgebungen

Virtuelle Maschinen und Container Umgebungen sind beides Möglichkeiten, um virtualisierte Umgebungen zu erstellen. [3] Bei VMs unterscheidet man grundsätzlich zwischen den zwei verschiedenen Hypervisoren, Typ 1 Hypervisor und Typ 2 Hypervisor, die die VM als Virtualisierung ermöglichen. Der Hypervisor, auch bekannt als Virtual Machine Monitor (VMM), ist hauptsächlich dafür zuständig die virtuelle Maschine, die auf ihm läuft zu überwachen. [14] Der Typ 1 Hypervisor, auch bekannt als bare metal Hypervisor, läuft direkt auf der Hardware (Infrastruktur), wie in Abbildung 1 sichtbar. Im Vergleich dazu benötigt der Typ 2 Hypervisor, auch hosted Hypervisor genannt, ein normales Betriebssystem, auf dem er installiert werden kann. [3] Hierzu müsste man in Abbildung 1 das Host Operating System wie bei der Containerized Application noch zwischen der Infrastruktur und dem Hypervisor hinzufügen.

Die erzeugten Virtualisierten Umgebungen haben sowohl in einer VM als auch in einem Container keinen Zugriff auf das Host Betriebssystem unter ihnen. Sie wissen nicht mal von dessen Existenz. Auf einem Host Betriebssystem können je nach dem, wie leistungsfähig die Hardware ist, beliebig viele VMs erstellt werden. Sie alle werden über den Hypervisor verwaltet und bekommen eigene Ressourcen wie CPU, RAM und Speicher zugewiesen. [4] Das ermöglicht unabhängig von dem Host Betriebssystem andere Betriebssysteme wie Linux, MacOS oder Windows auf einem Personal Computer (PC), beziehungsweiße auf der darauf befindlichen VM zu installieren. Der bekannteste Anbieter für VMs mit einem bare metal Hypervisor ist VMware und für einen hosted Hypervisor ist es VirtualBox. [3]

Ein Container virtualisiert nicht wie eine VM den gesamten PC bis zur Hardware, sondern Container virtualisieren nur die Softwareebene oberhalb des Betriebssystems. Dennoch besitzen sie alle benötigten Abhängigkeiten, die benötigt werden, um eine Softwareanwendung in einer virtualisierten Umgebung auszuführen. [3] Damit sind sie deutlich weniger speicherintensiv als VMs. Des Weiteren verwenden sie keinen Hypervisor was einen schnelleren Ablauf bezüglich Ressourcenzuweisung und Verfügbarkeit der Anwendungen ermöglicht. [4] Auf einem PC können auch mehrere Container installiert werden. Jedoch anders als bei VMs teilen sie sich den Betriebssystemkern, wenn zwei oder mehr Container den Gleichen benötigen, wie in Abbildung 1 sichtbar. Ausgeführt sind es jedoch komplett isolierte Prozesse für jede Anwendung. Ein Container benötigt sehr wenig Platz, da er alle benötigten Daten zur Ausführung der gewünschten Softwareanwendung wie den Code, Laufzeit, Systemtools- und Bibliotheken sowie Einstellungen als Abhängigkeiten verpackt. Somit ist er universell auf jeder Computerumgebung einsetzbar und zusätzlich sehr schnell. Wenn der Container ausgeführt wird, wird ein Container-Image, ein Abbild des Containers erzeugt, welches die Softwareanwendung isoliert ausführt. Von einem Container können parallel mehrere Images erzeugt werden. [10]

Die am meisten verwendete Container Umgebung sind Docker Container. Sie besteht aus der Docker Engine, welche als Laufzeitumgebung für Container dient. Mit ihrer Hilfe kann man Container erstellen, verwalten und ausführen.

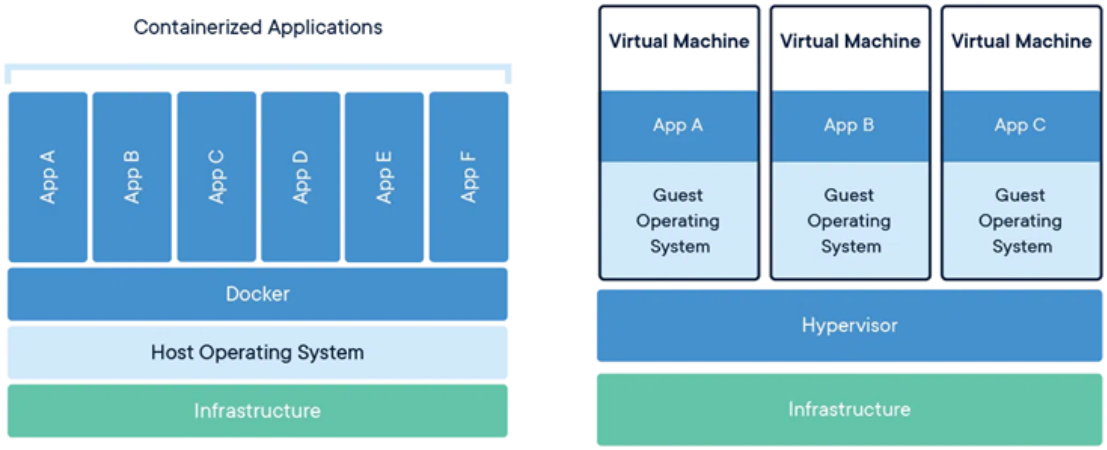

Abbildung 1: Vergleich VM und Docker

2.2 Testautomatisierung

2.2.1 Definition

Testautomatisierung bezeichnet den Prozess, durch den wiederholbare Testfälle automatisiert ausgeführt werden, anstatt sie manuell einzeln durchzuführen. Sie spielt eine entscheidende Rolle in der Qualitätssicherung von Softwareprojekten. Ein manueller Testfall beinhaltet die Beschreibung von Systemvoraussetzungen, Testumgebung und den zu prüfenden Aspekten. Nach Fehlerbehebung können unter identischen Bedingungen erneute Tests durchgeführt werden, was als "Retest"bezeichnet wird. [20] Automatisierte Tests bieten viele Vorteile: Sie erhöhen die Testabdeckung, ermöglichen zuverlässige Wiederholungen der Testfälle und sparen erheblich Zeit und Kosten. Sie sind besonders effizient für sich wiederholende, zeitaufwändige Testaufgaben und ermöglichen es, dass sich die Entwickler oder Tester auf andere Aufgaben, wie zusätzliche Testabdeckung konzentrieren können. [21]

Eine bedeutende Anwendung der Testautomatisierung ist der Regressionstest, der nach Implementierung von Änderungen unerwünschte Auswirkungen auf andere Funktionen überprüft. Dies ermöglicht eine bessere Messbarkeit der Softwarequalität und die frühzeitige Erkennung möglicher Nebenwirkungen. Zusätzlich bietet die Testautomatisierung Metriken zur Bewertung des Testfortschritts und der Softwarequalität. [16]

Des Weiteren beschleunigt die Testautomatisierung den Entwicklungsprozess erheblich. Durch die automatische Ausführung von Produktion, Installation und Test können zeitraubende manuelle Schritte vermieden und Ergebnisse rasch verfügbar

gemacht werden. [16] Zusammenfassend lässt sich festhalten, dass die Testautomatisierung essenziell in der Softwareentwicklung ist, um die Effizienz zu steigern und eine gute Qualität sicherzustellen.

2.2.2 Herausforderungen

Die meisten IT-Unternehmen, die eine eigene Development Abteilung haben, setzten heutzutage auf automatisierte tägliche Builds, testen jedoch immer noch manuell. Diese manuellen Testzyklen verzögern durch ihre Dauer von bis zu mehreren Wochen erheblich die Entwicklung, da die Entwickler auf die Ergebnisse der Tests angewiesen sind. Wenn Unternehmen es nicht schaffen, regelmäßig zu überprüfen, wie sich die neusten Veränderungen im Code auf entscheidende Interaktion des Nutzers mit der grafischen Benutzerschnittstelle (GUI) auswirken, hat dies negative Auswirkungen auf die Nutzererfahrung. GUI-Tests zu automatisieren, ist jedoch komplex und bringt diverse Herausforderungen mit sich. Ein häufiges Problem ist die falsche Kalkulation des Aufwands, der benötigt wird, um ein neues Testframework an die eigene Anwendung anzupassen, sowie eine nachhaltige Testautomatisierung über einen längeren Zeitraum zu betreiben. So muss das zur Testautomatisierung benutzte Framework kontinuierlich mit der eigenen Anwendung synchronisiert und aktualisiert werden. Ein weiteres Problem ist das Bereitstellen der richtigen Testdaten, damit eine realistische Testumgebung erzeugt werden kann. Wenn man eine solche Testumgebung erschaffen hat, müssen die erhaltenen Fehlerprotokolle noch richtig ausgewertet werden. Hierbei stellt das Herausfiltern von False Positives, sogenannte Fehlalarme, eine wichtige Rolle. Sie müssen schnell erkannt und eliminiert werden, damit man den Testergebnissen noch trauen kann und nicht irgendwann echte Fehlermeldungen für einen falschen Alarm hält. [8]

3. Testen von GUI

3.1 Was macht eine GUI aus

Eine grafische Benutzerfläche, im englischen Graphical User Interface (GUI) ist die Verbindung zwischen dem Computer und dessen Benutzer. Sie ermöglicht mithilfe von grafischen Elementen wie Bildern, Symbolen, Menüs und Schaltflächen das Interagieren zwischen Menschen und Maschine. So kann man durch Eingabefelder, Schaltflächen und Textfelder, Programme und Webseiten möglichst einfach bedienen. Das Ziel hierbei sollte sein die GUI möglichst einfach, intuitiv und benutzerorientiert zu gestalten, damit auch Benutzer mit wenig bis keiner technischen Erfahrung sich gut zurechtfinden und ein positives Nutzererlebnis haben. Weitere Ziele sollten eine visuelle Ansprechbarkeit, gute Struktur sowie ein zweckorientiertes Layout sein. [5] Orientieren kann man sich hierbei an den 10 Usability Heuristics von Jakob Nielsen, in denen die grundlegenden allgemeinen Gestaltungsprinzipien für Nutzerinteraktionen beschrieben werden. [1]

Sowohl Desktop-Anwendungen als auch Web-Anwendungen besitzen ein GUI, um mit den Benutzern zu interagieren. Während Desktop-Anwendungen auf einen einzelnen PC beschränkt sind und extra installiert werden müssen, können Web-Anwendungen überall von mehreren Geräten gleichzeitig genutzt werden, solange eine Internetverbindung vorhanden ist. Web-Anwendungen werden über einen Browser wie Firefox oder Chrome aufgerufen und sind bei uns in Form des GUIs, ohne etwas installiert zu haben, sichtbar, da die eigentliche Anwendung auf einem Remote-Server liegt und nur über ein Hypertext Transfer Protokoll (HTTP) oder Hypertext Transfer Protokoll Secure (HTTPS) zu dem Anwender übertragen wird. [7]

3.2 Web-Anwendung testen

Da Web-Anwendungen nicht auf einem Gerät installiert werden, sondern universell von allen internetfähigen Geräten aufgerufen werden können, stellt das Testen eine besondere Herausforderung dar. So kann eine Website sowohl von einem Desktop PC als auch von mobilen Endgeräten wie Mobiltelefonen aufgerufen werden, die meistens komplett verschiedene Darstellungen von Webseiten besitzen, wie zum Beispiel das Format. So muss eine Benutzerinteraktion mit der GUI, die auf dem Desktop PC funktioniert, nicht auch gleich auf dem mobilen Endgerät funktionieren. Hier ist das Testen der GUI besonders wichtig und herausfordernd, da man eine Anwendung auf mehreren Betriebssystemen wie Windows, Linux, MacOS, Android und IOS testen muss, um eine gute Benutzererfahrung garantieren zu können. Um nicht für jedes Betriebssystem ein neues technisches Gerät zum Testen anschaffen und warten zu müssen, lassen sich mithilfe von virtualisierten Umgebungen auf einem

einzigen Gerät alle Testumgebungen realisieren, die zusätzlich noch individuell konfigurierbar und isoliert voneinander sind, so dass alle Tests möglichst reibungslos und gleichzeitig unverfälscht ausführbar sind. Getestet werden dabei alle GUI-Elemente, mit denen der Benutzer interagieren kann, wie Radio Buttons, Check, Text und List Boxen getestet, ob sie die gewünschten Funktionen in jeglicher Umgebung korrekt ausführen.

3.3 Vergleich Browser und Headless Browser

Ein Headless Browser funktioniert genauso wie ein normaler Browser, mit dem Unterschied, dass er keine GUI besitzt. Ein Beispiel dafür ist der Google Chrome Browser, mit seinem dazugehörigen kopflosen Headless Modus. Er bietet eine Möglichkeit, um Tests von Webanwendungen in großem Maßstab durchzuführen, da durch die fehlende GUI kein menschliches Eingreifen nötig ist, zum Beispiel bei der Navigation von einer Website zur nächsten. Des Weiteren ist er schneller als ein normaler Browser, da er keine GUI besitzt. [15] Durch viele praktische Features für sowohl Tester als auch Webentwickler nimmt Headless Chrome immer mehr an Popularität zu. Einige Beispiele dafür sind dynamisches Rendering, um die Webseiten zu Rendern, sowie Laufzeiteninhalte darzustellen, Webseiten-Ranking, um Schwachstellen der eigenen Website zu erkennen und optimieren zu können, Laufzeitenberechnung, zur Bewertung der JavaScript Laufzeitdauer, sowie der damit verbunden Optimierungsmöglichkeit der Geschwindigkeit der Webseite und verbesserte Tests, um die eigene Webseite besser testen zu können. [13] Headless Browser zu testen ist für viele Testtools schwierig, da sie auf geometrische und grafische Verfahren zur Erkennung der einzelnen Komponenten setzten, was bei einem Headless Browser nicht möglich ist, da keine GUI vorhanden ist.

3.4 Einrichtung der Testumgebung

Als Testumgebung für das Testtool wurde das Betriebssystem Linux, genauer Ubuntu, auf jeweils einer VM und Container Umgebung eingerichtet. Die VM wurde mit Virtual Box erzeugt und hat Zugriff auf 3043MB RAM sowie zwei Prozessor Kerne mit bis zu 100% Auslastung. Des Weiteren verwendet sie einen Typ-2 Hypervisor. Als Container Umgebung für das Testtool dient ein Docker Container. Dieser besitzt identische Hardwarebeschränkungen wie die VM, die jedoch nicht in dem Interface von Docker eingerichtet werden konnten, wie es bei Virtual Box beim Erzeugen einer neuen VM einfach möglich ist. Hierzu muss der Hypervisor selber eingeschränkt werden, was mithilfe des Windows Subsystem für Linux (WSL) 2 in der .wslconfig Datei gemacht wurde. Damit sind zwei virtualisierte Umgebungen, mit identischer, zu Verfügung stehender Hardware, erstellt. Auf beiden kann ein Testtool installiert

werden, um die benötigten Messwerte zur Analyse messen zu können. Gesucht wird nur der Vergleich der benötigten Dauer zur Ausführung des Testfalls. Das bedeutet, das zu testende Programm spielt keine entscheidende Rolle, da es in beiden Fällen identisch ist und die reine Testdauer gesucht wird.

3.5 Betrachtetes Testtool: QF-Test

3.5.1 Auswahlkriterien für Testtool

Das wichtigste Auswahlkriterium für das Testtool ist die Unterstützung für Webtests, da eine Webanwendung getestet wird. Das bekannteste Testframework hierfür ist Selenium, welches mit dem Testtool QF-Test verglichen wird. Dabei sind die Auswahlkriterien: Setup und Programmierung, Zielgruppe, Aufnahme und Wiedergabe, Objekterkennung und Test Wiederverwendbarkeit. Um ein Selenium Setup (Integrated Development Environment (IDE) und Server) aufzusetzen benötigt man gute Entwicklerkenntnisse, da Selenium über kein einfach zu verstehendes Design verfügt. QF-Test dagegen ist auch für Tester ohne gute Programmierkenntnisse einfach zu verstehen und zu benutzen. Das ist wichtig, weil viele der Tester meistens keine bis wenig Erfahrung damit haben eine Entwicklungs-Umgebung aufzusetzen. Die Zielgruppe besteht bei beiden aus Entwicklern die Programmierkenntnisse haben, jedoch bei QF-Test auch noch aus Testern ohne Programmierkenntnisse, da QF-Test über eine GUI verfügt in der die Tests einfach und intuitiv modular zusammengebaut werden können. Die Aufnahme und Wiedergabe ist in Selenium nur eingeschränkt über ein Firefox Plugin möglich. In QF-Test jedoch ist es eine der Grundfunktionen, jegliche Aktionen und Checks aufzunehmen und anschließend auch noch direkt bearbeiten zu können. In Selenium müssen jegliche GUI-Objekte im Code der Website oder durch durchgehende einheitliche Programmiermuster erkannt und verwaltet werden. Wohingegen QF-Test einen eigenen mehrstufigen und anpassbaren Erkennungsalgorithmus, der tolerant gegen Elementänderungen in der Hierarchie ist, besitzt. Des Weiteren können GUI-Komponenten anhand von festen IDs der GUI-Objekte erkannt werden. Um Tests nochmals zu verwenden ist es in Selenium nötig, grundlegende Kenntnisse in objektorientierter Programmierung zu besitzen, da es keine grafische Oberfläche wie in QF-Test gibt, in der Tests modular darstellbar sind. Das sind einige der vielen Kriterien, weshalb als Testtool QF-Test verwendet wurde, da es eine sehr gute und einfach zu verstehende grafische Oberfläche bietet, in der Tests aufgenommen, bearbeitet und automatisiert werden können.[2]

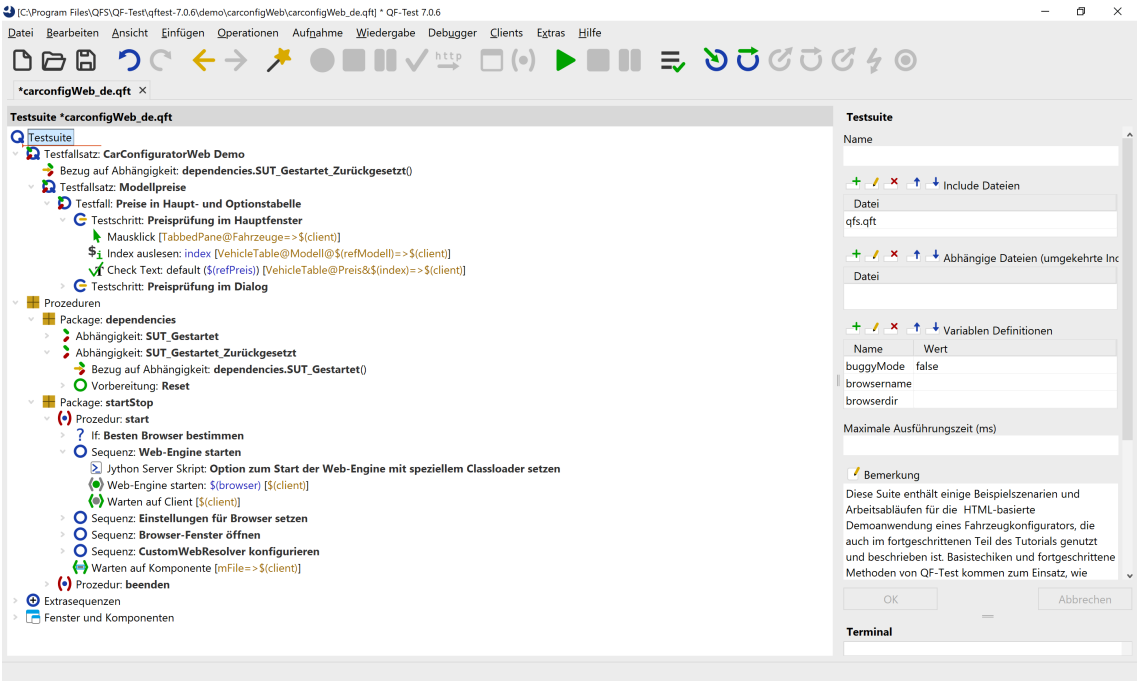

Abbildung 2: Vereinfachte Demotestsuite zum Webtesten in QF-Test

3.5.2 Aufbau eines Tests

Ein Test in QF-Test wird in einer sogenannten Testsuite abgespeichert. In dieser befinden sich beliebig viele Testfallsätze, die zur Strukturierung der Testsuite dienen. In den einzelnen Testfallsätzen wiederum befinden sich die einzelnen Testfälle. Der Testfall ist der wichtigste Knotentyp. In ihm befindet sich der eigentliche Test, bestehend aus Vorbereitungs- und Aufräumtätigkeiten und den einzelnen Testschritten, die das zu testende System (SUT) testen. Die Vorbereitungs- und Aufräumtätigkeiten sorgen dafür, dass eine homogene Testumgebung bei jedem Testdurchlauf vorhanden ist und somit Komplikationen mit vorrangehenden oder nachfolgenden Testfällen oder Testfallsätzen verhindert werden. Die einzelnen Testschritte können diverse Knoten wie Eventknoten (zum Beispiel: Mausevents, Tastaturevents oder Texteingaben), Checkknoten (zum Beispiel: Check Text, Check Element oder Check Geometrie), Prozeduraufrufe oder auch Ablaufsteuerungen (zum Beispiel: Schleifen, If Abfragen oder Server- und SUT-Skripte). Einzelne Testschritte oder die darin enthaltenen Events können in Sequenzen zusammengefasst werden. Diese Sequenzen können wiederum als Prozedur abgespeichert und durch einen Prozeduraufruf an der jeweiligen Stelle aufgerufen werden. Das führt zu einer besseren Übersichtlichkeit und vereinfacht sich wiederholende Abschnitte im Test, wie das Starten des SUTs. In Abbildung 2 ist eine, der von QF-Test zur Verfügung gestellten Demo Testsuiten zum Thema Webtesten vereinfacht zusehen, um den Aufbau einer Testsuite in QF-Test verständlicher zu machen. In der Testsuite befindet sich ein übergeordneter Testfallsatz mit dem Namen CarConfiguratorWeb Demo. Ihm untergeordnet ist ein weiterer Testfallsatz sowie eine Abhängigkeit auf eine Prozedur, die dafür sorgt, dass bei dem

Starten der Testsuite das zu testende SUT, in diesem Fall der CarConfigurator (eine von QF-Test mitgelieferte Demoapplikation), wenn noch nicht gestartet, jetzt gestartet und auf den Ausgangszustand zurückgesetzt wird. Weiter unten in der Baumstruktur befinden sich die einzelnen Testschritte, hier zum Beispiel Preisprüfung im Hauptfenster mit den einzelnen Eventknoten wie dem Mausklick. Die Werte in den eckigen Klammern hinter dem Namen des Eventknoten geben konkrete Werte in Verbindung zu dem zu testenden SUTs an, sodass der Mausklick auch auf die richtige Komponente ausgeführt wird, in diesem Fall die Komponente Fahrzeuge. Unter Prozeduren sind alle in der Testsuite benutzten Prozeduren einmal abgespeichert. Zum Beispiel die Prozedur dependencies.SUT_Gestartet_Zurückgesetzt() die unter dependencies abgespeichert ist und wiederum auf andere Prozeduren aus dem startStop Packet zugreift, um das SUT je nach Ausgangslage zu starten, oder zurückzusetzen.[18]

3.5.3 Erstellung der Tests

Für die Ermittlung der benötigten Testlaufzeiten, wurde der Test auf zwei Testsuiten aufgeteilt. In der ersten Testsuite befindet sich ein einzelner Testfall in einem Testfallsatz. In diesem wird zuerst der sogenannte Testrunlistener in Form einer Prozedur aufgerufen. Der Testrunlistener ist dafür zuständig alle ausgeführten Knoten, Knotenevents wie Mausevents und die gesamte Testlaufzeit zu messen und in Form eines Textdokumentes im selben Ordner wie die Testsuite abzuspeichern. Dieser Ablauf ist in Form eines Jython Server Skripts implementiert.

Nach dem der Testrunlistener registriert, also aktiviert wurde, wird nun die in Abbildung 2 gezeigte Demotestsuite für den CarConfigoratorWeb 5-mal aufgerufen und ausgeführt und währenddessen alle benötigten Werte gemessen. Nach Vollendung der fünf Durchgänge werden die Werte abgespeichert und der Testrunlistener wieder deaktiviert. Der genaue Aufbau der Testsuite ist in Abbildung 3 sichtbar.

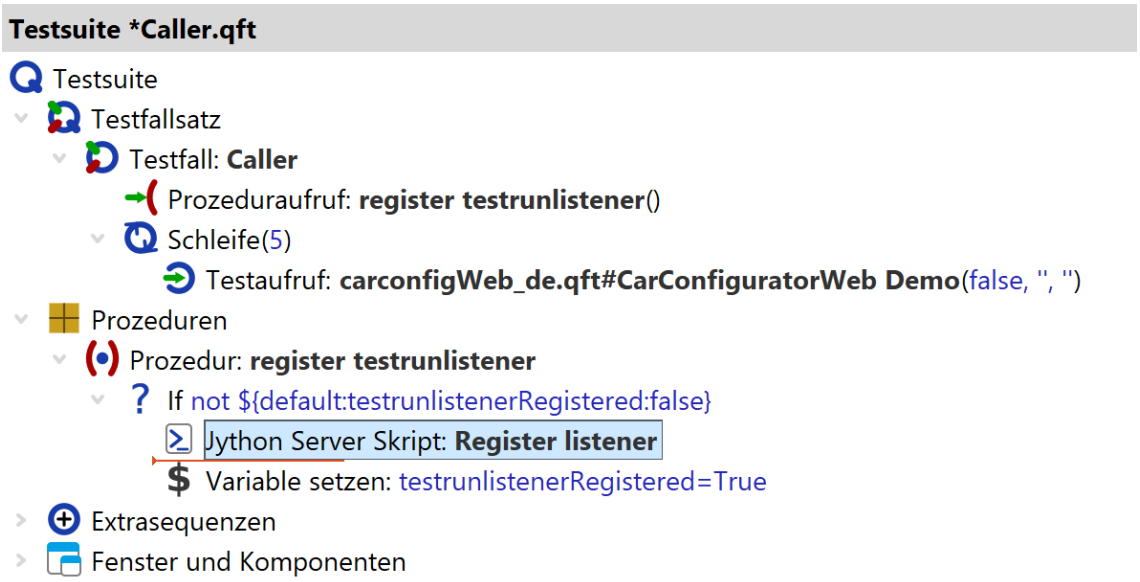

Abbildung 3: Caller Testsuite

4. Performance Metriken

Um die Leistungsfähigkeit der aufgesetzten virtualen Maschine und dem Docker Container vergleichen zu können, müssen die Laufzeiten bei der Ausführung der Tests gemessen werden um anschließend genügend Daten zu besitzen, damit aussagekräftige Performance Metriken erstellt werden können.

4.1 Erfassung der Daten

4.1.1 Testlaufdauer

Um die gesamte Testlaufdauer der auszuführenden Tests zu ermitteln, liefert QF-Test bereits ein Feature mit, welches die Testlaufdauer vom Starten des Tests bis hin zum erfolgreichen Beenden der Testsuite misst. Ablesen lässt sich die Zeit nach Aktivierung des Features im Ergebnisprotokoll, wo vor jedem Knoten, je nach Dauer in Abhängigkeit der gesamten Testlaufdauer, ein farblich markanter Balken sichtbar ist, wie in Abbildung 4 sichtbar. Für genaue Werte lassen sich die einzelnen Knoten oder Knotenevents auswählen und unter Echtzeit ist die Dauer des jeweiligen Knotens, je nach Dauer in Sekunden, Minuten oder Stunden sichtbar. [18]

Abbildung 4: Ergebnisprotokoll Testlaufzeit

4.1.2 Geschwindigkeit einzelner Events

Die gesamte Testlaufdauer sind bei fünf Wiederholungen des CarConfigWeb Demo Tests mit jeweils Chrome und Headless Crome in der VM, sowie im Container nur 20 Werte. Da jedoch zu jedem einzelnen Knoten und jedem einzelnem Eventknoten, wie Maus- oder Tastenevents, die Dauer gemessen werden soll, steigt die Anzahl der Daten in die Tausende, so dass sie sich nicht mehr manuell ablesen und aufschreiben lassen. Hierzu wurde in den Testrunlistener, der bereits unter Unterabschnitt 3.5.3 erwähnt und erklärt wurde, ein Skript eingebaut, welches die Werte automatisch misst und direkt in eine Textdatei einträgt. Das ausführliche Jython Server Skripts ist im Anhang zu finden.

Somit erhalten wir am Ende vier Datensätze, aufgeteilt in einen für die VM mit Chrome und einen für die VM mit Headless Chrome, einen für die Container Umgebung mit Chrome und einen für die Container Umgebung mit Headless Chrome, mit jeweils der gesamten Testlaufdauer sowie allen anderen Laufzeiten der einzelnen Knoten und Events.

4.2 Strategien der Statistischen Ergebnisanalyse

4.2.1 Performance Analyse

Um die gewonnen Datensätze auszuwerten, gibt es verschiedene Analysemethoden der Statistik. Einige davon sind Deskriptive Statistik, Induktive Statistik oder auch Inferenzstatistik. Die Deskriptive Statistik verwendet statistische Methoden wie das Darstellen von Daten in Graphiken und Tabellen zum Beschreiben und Auswerten von Daten. Hierzu werden Kennzahlen wie der Mittelwert, die Streuung oder die Standardabweichung berechnet. Wichtig hierbei ist, dass die Aussagen immer nur zu dem vorliegenden Datensatz getätigt werden und nicht darüber hinaus. Wenn man darüber hinaus Aussagen zu einer Grundgesamtheit treffen möchte, befindet man sich in der Induktiven Statistik. [6] Die Inferenzstatistik steht wie die Induktive Statistik im Gegensatz zur Deskriptive Statistik. Sie versucht mithilfe von Stichproben Aussagen über die Grundgesamtheit zu treffen. Es wir nur ein kleiner Datensatz verwendet, aus denen Hypothesentests gebildet werden. Hierbei gibt es einfache Testverfahren, wie den Binominal Test, aber auch Regressions- und Korrelationsanalysen. [17]

4.2.2 Zuverlässigkeit bei Wiederholung der Testdurchläufe

Um vertrauenswürdige Daten zu erhalten, muss man die Stichprobengröße möglichst groß wählen. Das führt zu zuverlässigeren Ergebnissen. [17] Hierzu beinhaltet der Test jegliche Interaktionsmöglichkeiten mit der GUI, um alle Fehlerquellen abzudecken. So kann ein einfacher Mausklick auf beiden virtualisierten Systemen die gleiche Zeit dauern, jedoch eine komplexere Aktion, wie das Verschieben eines Fensters komplett unterschiedlich sein.

Zufällige Messfehler sind nicht auszuschließen. So kann eine erhöhte Raumtemperatur zu schlechteren CPU-Leistungen führen, oder andere Programme im Hintergrund können die Messung der Laufzeiten beeinflussen. Um diesen Fehler zu minimieren wurde der Test auf jedem virtualisierten System mehrmals durchgeführt und im Anschluss der Mittelwert gebildet. Da die zu testende Anwendung nur eine Demo Testanwendung ist, wird davon ausgegangen, dass alle Testdurchläufe durchlaufen werden, ohne das Fehler auftreten. Im realen Testalltag einer jeden Entwicklung laufen die Tests meistens täglich. Und hierbei treten regelmäßig Fehler bei

Veränderungen an dem Softwarecode auf. Hier ist es besonders wichtig, dass die Testergebnisse zuverlässig und vertrauenswürdig sind und die Tests nicht falsche Testergebnisse liefern. [21]

4.3 Anwendung auf das Beispielszenario

Zum Auswerten der Datensätze wurden mehrere Strategien der Statistischen Ergebnisanalyse benutzt, angefangen mit der Deskriptiven Statistik. Hierzu wurde aus den fünf Testdurchgängen jeweils getrennt für jeden Datensatz der Durchschnitt für jeden Knoten und jedes Event gebildet, wie in Tabelle 1 sichtbar.

Knoten	**VM Interactive**	**Docker Interactive**	**VM Headless**	**Docker Headless**
CheckStringStep	193,5061224	125,4	130,6897959	53,65306122
MouseEventStep	354,9719008	143,685124	199,3347107	101,2917355
TextInputStep	533,5333333	183	271,8	157,6
ClientWaiter	117,4634146	131,1219512	111,804878	99,14634146
ComponentWaiter	155,3141361	67,90575916	100,0418848	38,67539267
FetchIndexStep	267,5	178,6333333	188,1333333	90
FetchTextStep	130,84	75,77	109,63	62,02
SelectionEventStep	242,8380952	103,1619048	143,0095238	68,36190476
DocumentWaiter	3910	1789	1533	568
InstallCWRStep	1480	1330	1005	557
BrowserClientStarter	4041	2590	2077,5	688,5
TestCall	136015,8	61840	82283,6	38531,8
RootStep	681625	309825	412658	192930

Tabelle 1: Wertetabelle Testlaufzeiten in Sekunden (ms)

In Tabelle 1 sind diverse Knoten aufgelistet, von denen allen die Testlaufzeiten genommen wurden. Unterteilen lassen sie sich in Eventknoten, Checkknoten und Knoten zum Ausführen des Tests. Zu den Eventknoten gehören MouseEventStep, bei dem eine beliebige Mausinteraktion mit der GUI ausgeführt wird, TextInputStep, bei dem eine Tastatureingabe in ein Textfeld erfolgt und SelectionEventStep, bei dem etwas in einem Auswahlmenü ausgewählt wird. Zu den Checkknoten gehört CheckStringStep, bei dem eine Komponente der GUI mit einem String auf ihre Korrektheit des Inhaltes überprüft wird. ClientWaiter, ComponentWaiter und DocumentWaiter halten den Test an bis der Client, die Komponente oder das Dokument reagiert. BrowserClientStarter startet den Client im Browser und InstallCWRStep wird von QF-Test, zur Aufnahme und Wiedererkennung von Komponenten benötigt. Der Knoten RootStep steht für den Start der Testsuite und somit für die gesamte Testlaufzeit und in dem Knoten TestCall wird die Demo Testsuite zum Testen des CarConfiguratorWeb aufgerufen.

Nach dem der Datensatz mithilfe Deskriptiver Statistik durch Tabellen (siehe Tabelle 1) und Grafiken (siehe Abbildung 5, Abbildung 6 und Abbildung 7) ausgewertet wurde, wird in die Induktive Statistik übergegangen, um anhand der Messwerte generelle Aussagen zu dem Vergleich von virtuellen Maschinen und Containern zu treffen.

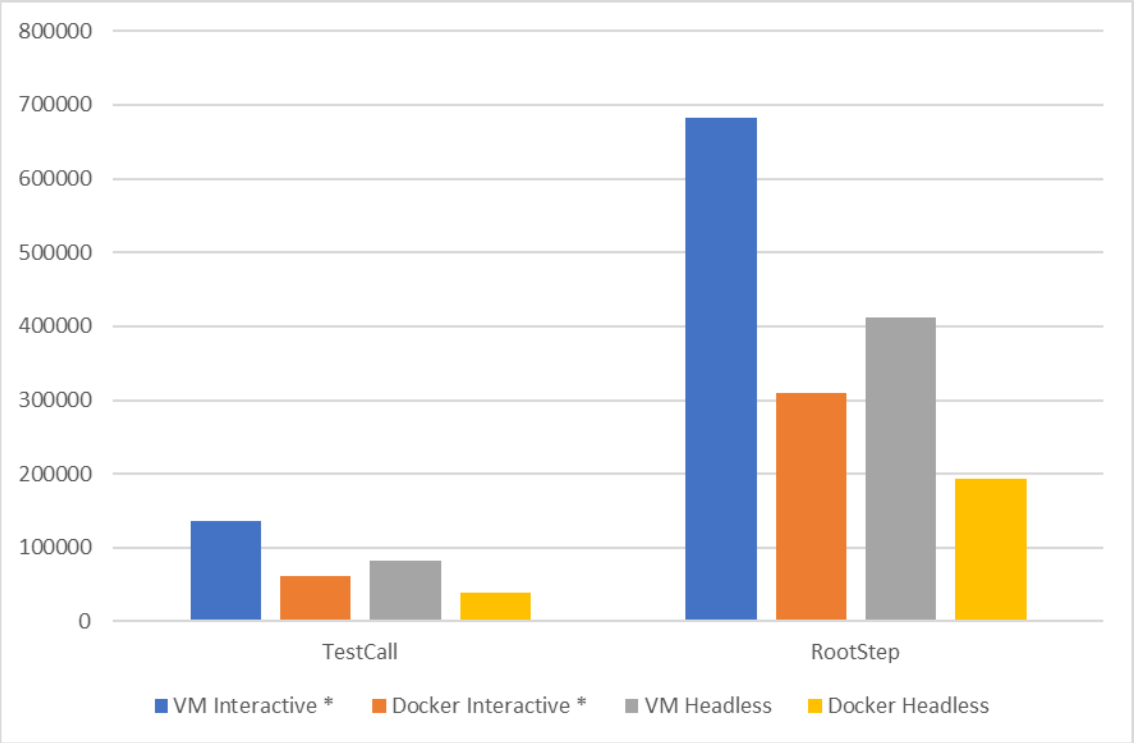

Abbildung 5: Gesamte Testlaufzeit

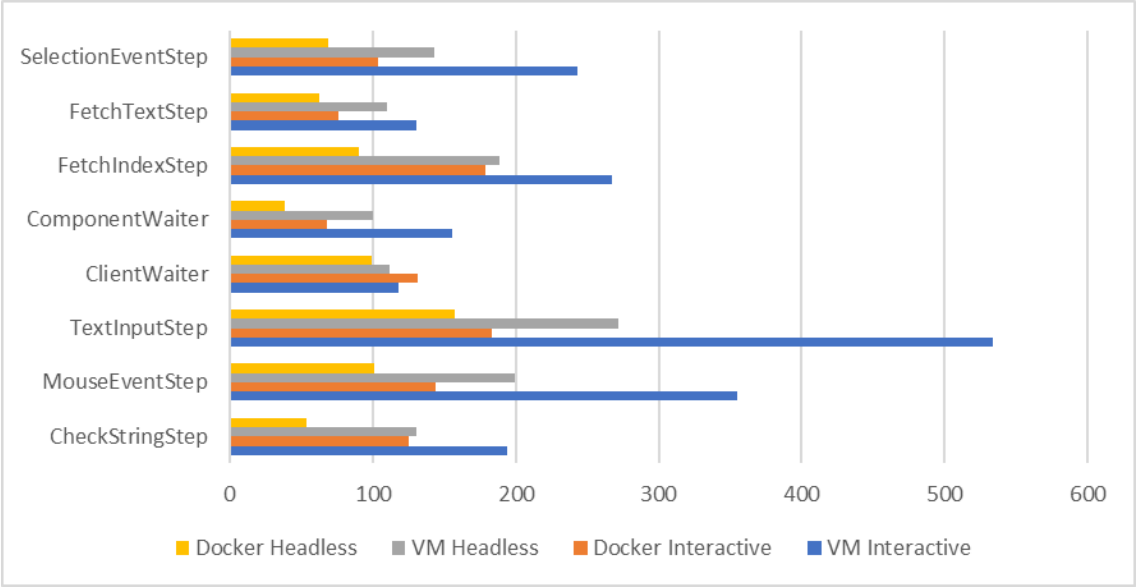

Abbildung 6: Knoten mit kurzen Laufzeiten

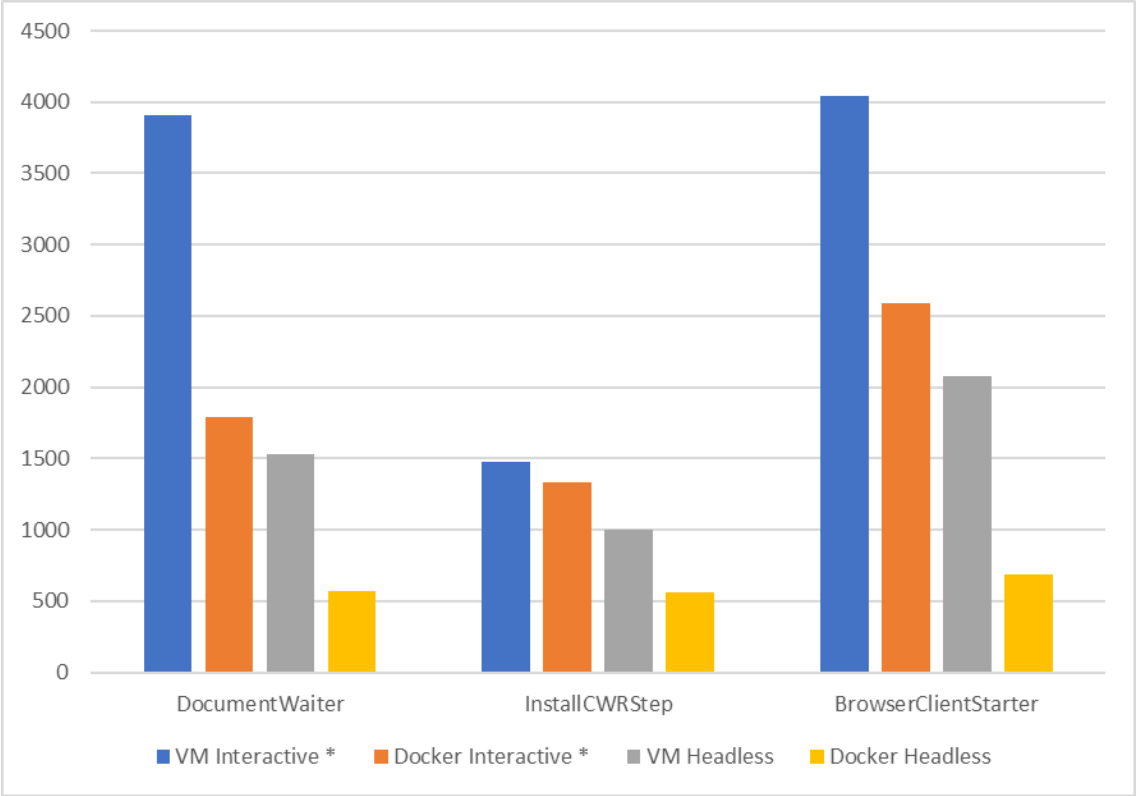

Abbildung 7: Knoten mit langen Laufzeiten

5. Auswertung und Ausblick

5.1 Auswertung

In Abbildung 5 ist gut sichtbar zu erkennen, dass sowohl bei der VM als auch bei dem Docker Container der Headless Chrome Modus ca. 66% schneller ist als das Interaktive Chrome mit GUI. Vergleichen wir VM und Container miteinander, so ist die Container Umgebung in Bezug auf die gesamte Laufzeit des Testes, sowohl interaktiv als auch Headless mehr als doppelt so schnell. Für eine möglichst schnelle Testumgebung bei gleichen zur Verfügung stehenden Ressourcen sollte man für Webtests eine Container Umgebung mit einem Headless Browser verwenden. Somit spart man im Vergleich zu einer virtuellen Maschine in Kombination mit einem normalen interaktiven Browser bis zu zwei Drittel der Zeit, die für einen Testdurchlauf benötigt wird. Und dennoch ist das heutzutage immer noch die am meisten genutzte Kombination zum Webtesten in virtualisierten Umgebungen.

Wenn man sich die einzelnen Knoten anschaut, erkannt man, dass in einigen Knoten, wie zum Beispiel ClientWaiter oder InsallCWRStep, die Unterschiede relativ gering sind oder sogar die VM minimal schneller ist. Bei anderen Knoten jedoch ist der Docker Container nochmal schneller im Vergleich zur VM, als beim Vergleich der Gesamttestlaufzeit. Beispiele dafür sind MouseEventStep, SelectionEventStep und TextInputStep, wo der Container ganze 300% schneller ist, siehe Abbildung 6 und Abbildung 7. Das alles sind Eventknoten, Knoten, die direkt mit der GUI interagieren. Diese schafft der Docker Container ohne eine GUI, die sehr viel Leistung kostet, deutlich schneller zu testen als die VM. Das wird belegt dadurch, dass wenn man den Browser in der VM Headless, also ohne GUI ausführt und man jetzt nicht mehr geometrische Erkennungsmerkmale beim Erkennen der Komponente, sondern nur noch Indices auf Codeebene benutzt, so wird auch die Testlaufzeit bei der VM doppelt so schnell, während der Zeitgewinn bei der Container Umgebung bei diesen Knoten nur noch ca. 25% beträgt.

Doch auch bei den einzelnen Knoten lässt sich zusammenfassend sagen, dass die Container Umgebung deutlich schneller ist, als die virtuelle Maschine und somit hoffentlich trotz ihrer etwas weniger intuitiven und einsteigerfreundlichen Aufsetzung deutlich mehr Benutzer in der Softwareentwicklung findet.

5.2 Aussicht

In der Zukunft, zur Vertiefung und Ausarbeitung der Arbeit, sollte die virtuelle Maschine und die Docker Umgebung nicht auf einem einfachen Desktop Computer, sondern auf einem Cloudserver eingerichtet werden, wo sie beide auch abseits des Hostsystems komplett isoliert sind, um jegliche zufälligen Fehler zu vermeiden. In Verbindung damit ist eine Einbindung der generierten Datensätze in Grafana sinnvoll, um sie automatisch auszuwerten und grafisch darstellen zu können. Das bietet die Möglichkeit, noch größere Mengen an Daten auszuwerten und zu analysieren, sowie bei einer Weiterentwicklung der Testsoftware, der genutzten VM oder der Container Umgebung regelmäßig die Tests laufen zu lassen, um eine Performance-Änderung direkt feststellen zu können.

Literaturverzeichnis

[1] 10 usability heuristics for user interface design. https://www.nngroup.com/articles/ten-usability-heuristics/. [Online, zugegriffen am 01.10.2023].

[2] Comparing selenium vs. qf-test. https://www.qfs.de/produkt/qf-test/web-testen/testautomatisierung-selenium-vs-qf-test.html. [Online, zugegriffen am 30.09.2023].

[3] Container vs. virtuelle computer. https://www.qfs.de/produkt/qf-test/web-testen/testautomatisierung-selenium-vs-qf-test.html. [Online, zugegriffen am 01.10.2023].

[4] Containers vs. virtual machines (vms): What's the difference? https://www.ibm.com/blog/containers-vs-vms/. [Online, zugegriffen am 01.10.2023].

[5] Containers vs. virtual machines (vms): What's the difference? https://www.ionos.de/digitalguide/websites/web-entwicklung/was-ist-ein-gui/. [Online, zugegriffen am 01.10.2023].

[6] Definition deskriptive statistik. https://de.statista.com/statistik/lexikon/definition/49/deskriptive_statistik/. [Online, zugegriffen am 06.10.2023].

[7] Desktop- vs web-app – die wichtigsten vor- und nachteile. https://www.idesis.de/desktop-vs-web-anwendung/#:~:text=Dabei%20wird%20die%20Webanwendung%20NICHT,lokal%20auf%20dem%20Rechner%20verf%C3%BCgbar. [Online, zugegriffen am 03.10.2023].

[8] Die 5 größten herausforderungen bei der testautomatisierung. https://www.it-daily.net/it-management/business-software/die-5-groessten-herausforderungen-bei-der-testautomatisierung. [Online, zugegriffen am 28.09.2023].

[9] Konzepte der virtualisierung im Überblick. https://www.ionos.de/digitalguide/server/konfiguration/virtualisierung/. [Online, zugegriffen am 23.09.2023].

[10] Use containers to build, share and run your applications. https://www.docker.com/resources/what-container/. [Online, zugegriffen am 01.10.2023].

[11] Virtualisierung in unternehmen: Vorteile, nachteile & lösungen. https://www.heise.de/download/specials/

Virtualisierung-in-Unternehmen-Vorteile-Nachteile-Loesungen-6338923. [Online, zugegriffen am 24.09.2023].

[12] Vor- und nachteile von virtualisierung. https://basic-tutorials.de/news/vor-und-nachteile-von-virtualisierung/. [Online, zugegriffen am 24.09.2023].

[13] What is the google chrome headless mode? https://www.debugbar.com/what-is-the-google-chrome-headless-mode/. [Online, zugegriffen am 03.10.2023].

[14] Pratik Sharma Bhautik Patel Ankita Desai, Rachana Oza. *Hypervisor: A survey on concepts and taxonomy. International Journal of Innovative Technology and Exploring Engineering.* [2013].

[15] A BALABASH, M.; OSKIN. The inside of headless chrome. [https://elib.psu.by/bitstream/123456789/31117/1/154-155.pdf], 2020.

[16] Manfred BAUMGARTNER. *Basiswissen Testautomatisierung: Aus-und Weiterbildung zum ISTQB® Advanced Level Specialist–Certified Test Automation Engineer.* dpunkt. verlag, [2021].

[17] Ludwig FAHRMEIR. *Statistik: Der weg zur datenanalyse.* Springer-Verlag, [2016].

[18] Quality First Software GmbH. Qf-test - das handbuch. [https://archive.qfs.de/qftest/manual _de.pdf], Oktober 2023.

[19] Christoph MEINEL. *Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht.* Universitätsverlag Potsdam, [2011].

[20] Alberto VIVENZIO. *Testautomation mit SAP®: SAP Banking erfolgreich einführen.* Vieweg+Teubner Verlag Wiesbaden, [2010].

[21] Frank WITTE. *Testmanagement und Softwaretest: Theoretische Grundlagen und praktische Umsetzung.* Springer-Verlag, [2016].

Tabellenverzeichnis

Abbildungsverzeichnis

A. Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Projektarbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.

Ort, Datum	Vorname Name

B. Testrunlistener Jython Server Skript

```
from de.qfs.apps.qftest.extensions.qftest import
   AbstractTestRunListener
from java.io import File
from java.lang import System
from time import strftime
from java.util import Date, Calendar, TimeZone
from java.text import SimpleDateFormat

class Tracer (AbstractTestRunListener):

    def init(self, level=0):
        #force Jython to use right classes
        from java.text import SimpleDateFormat
        from java.lang import System
        from java.io import File
        now = System.currentTimeMillis()
        dateFormat = SimpleDateFormat("yyyy_MM_dd_HH_mm")
        starttime = dateFormat.format(Date(now))
        directory = rc.lookup("qftest", "suite.dir")

        #create teststart time directory if set
        testStartTime = rc.lookup("default", "testStartTime:XXX")
        if testStartTime:
            directory = directory + "/" + testStartTime

        try:
            baseFile = File(directory)
            baseFile.mkdirs()
        except:
            pass

        self.logFile = directory + "/" + starttime + ".log"
        print "New logfile:", self.logFile
        self.error = "OK"
        self.dateformat = dateFormat

    def runStarted(self, event):
        pass

```

```
    def runStopped(self, event):
        pass

    def nodeEntered(self, event):
        #rc.callProcedure("startStop.monitor_cpu_ram", "nodeType"=
    event.getNode().getType())
        pass

    def nodeExited(self, event):
        comment = event.getNode().getComment()
        nodeType = event.getNode().getType()
        if nodeType not in ["Dependency",
                "TestSet", "RepeatSequence", "TestStep",
                "TestCase","TryStep", "CatchSequence",
                "DependencyReference", "ElseSequence", "IfSequence"
    , "ElseifSequence",
                "SetupSequence", "ClientScriptStep",
                "SetGlobalStep", "CommentStep", "BreakStep",
                "ServerScriptStep", "ReturnStep", "TableDataBinder"
    ,
                "DataDriver", "BasicSequence","Procedure","
    ProcedureCall"]:

            #print "comment", comment
            #if comment.find("@transaction") != -1:
            #cpu_mem=rc.callProcedure("startStop.monitor_cpu_ram")
            theFile = open(self.logFile, 'a')
            try:
                #force Jython to use right classes
                from java.lang import System

                now = System.currentTimeMillis()
                formatedNow = self.dateformat.format(Date(now))
                duration = int(rc.context.getCurrentLog().
    getDuration())
                logText =  "{nodeType},{duration}".format(nodeType=
    nodeType, duration = str(duration))
                print logText
                theFile.write(logText)
                theFile.write("\n")
            finally:
                if theFile != None:
                    theFile.close()
                self.error = "OK"

def problemOccurred(self, event):
        if event.getState() > 1:
            self.error = "KO"
        pass
```

```
82
83 global tracer
84 try:
85     rc.removeTestRunListener(tracer)
86 except:
87     pass
88
89 tracer = Tracer()
90 rc.addTestRunListener(tracer)
```