



Umwelt-Campus
Birkenfeld

H O C H
S C H U L E
T R I E R

Automatisierung von Messverfahren zur Bestimmung der Ressourceneffizienz von Software mittels Continuous Integration

Bachelor Thesis

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

im Studiengang

Medieninformatik
am Umwelt-Campus Birkenfeld

vorgelegt von

Maik Kreutzer

am 23.03.2022

Erstprüfer: Prof. Dr. Stefan Naumann
Zweitprüfer: Prof. Dr. Rolf Krieger

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen.

Datum: _____ Unterschrift: _____

Danksagungen

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich im Laufe der Erstellung dieser Bachelorthesis unterstützt haben.

Herrn Prof. Dr. Naumann möchte ich besonders für die Betreuung meiner Arbeit danken. Für die hilfreichen Anregungen und Vorschläge, sowie auch für konstruktive Kritik während der Anfertigung möchte ich mich recht herzlich bedanken.

Bei Herrn Prof. Dr. Krieger möchte ich mich ebenfalls für die Übernahme des Zweitprüfers bedanken.

Ebenfalls einen besonderen Dank möchte ich an Achim Guldner aussprechen, welcher mich auf das Thema aufmerksam gemacht hat, sich regelmäßig nach Problemen bei mir erkundigt hat und mir dabei auch immer gut helfen konnte.

Für die angenehme Kooperation mit dem gesamten sam.ware-Team von der Fresenius Digital Technology bedanke ich mich bei allen Mitarbeitern, insbesondere beim Teamleiter Hendrik Schneider, welcher sich im Zuge meiner Thesis für ein Interview bereitgestellt hat.

Für die persönliche Motivation und Unterstützung im Alltag, möchte ich mich bei meiner Familie, meiner Freundin und Freunden bedanken, die mir die Erstellung der Thesis deutlich erleichtert haben.

Gender-Disclaimer

„Zur besseren Lesbarkeit wird in der vorliegenden Arbeit auf die gleichzeitige Verwendung männlicher und weiblicher Sprachformen verzichtet. Es wird das generische Maskulinum verwendet, wobei beide Geschlechter gleichermaßen gemeint sind.“[mon19]

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einleitung | 1 |
| 1.1 | Motivation | 1 |
| 1.2 | Ziel der Thesis | 2 |
| 1.3 | Struktur der Thesis | 3 |
| 2 | Grundlagen | 4 |
| 2.1 | Green IT | 4 |
| 2.1.1 | Bedeutung und Ziele | 4 |
| 2.1.2 | Nachhaltige Software | 4 |
| 2.1.3 | Bewertung von Software | 6 |
| 2.2 | Qualitätskontrolle | 6 |
| 2.2.1 | Wichtigkeit von Softwarequalität | 6 |
| 2.2.2 | Qualitätsmerkmale | 7 |
| 2.2.3 | Vorgehensmodelle | 7 |
| 2.2.4 | Softwaretests | 8 |
| 2.2.5 | Verschiedene Arten von Softwaretests | 8 |
| 2.3 | Continuous Integration | 9 |
| 2.3.1 | Bedeutung und Ablauf | 9 |
| 2.3.2 | Analyse von CI-Tools | 10 |
| 2.3.3 | Konfiguration von Jenkins | 12 |
| 2.3.4 | Zwischenfazit | 13 |
| 3 | Qualitätskontrolle im Kontext von nachhaltiger Software | 14 |
| 3.1 | Qualitätsmodelle | 14 |
| 3.1.1 | Modelle | 14 |
| 3.1.2 | Ergebnisse | 14 |
| 3.2 | Vorgehen in der Softwareentwicklung | 16 |
| 3.2.1 | Tests und Messungen | 16 |
| 3.2.2 | Automatisierung | 16 |
| 3.3 | Beispiel aus der Praxis für Qualitätskontrolle | 17 |
| 3.3.1 | Interview | 17 |
| 3.4 | Probleme und Ausblick | 18 |
| 4 | Messkonzept mit Continuous Integration | 19 |
| 4.1 | Einführung | 19 |
| 4.2 | Testanwendungen | 19 |
| 4.2.1 | Efficient | 19 |
| 4.2.2 | Obelisk | 21 |
| 4.3 | Versuchsaufbau | 21 |
| 4.3.1 | Bisheriger Aufbau des ISS | 21 |
| 4.3.2 | Unterscheidung von Baseline, Leerlauf und Nutzungsszenario | 23 |
| 4.3.3 | Nutzungsszenarien mit Selenium | 24 |
| 4.3.4 | Hardware-Messungen | 25 |
| 4.3.5 | Continuous Integration Server | 25 |
| 4.3.6 | Visualisierung der Messergebnisse durch Autoplot | 26 |
| 4.3.7 | Allgemeiner Ablauf | 28 |
| 4.4 | Erster Vergleich des bisherigen und neuen Versuchsaufbaus | 29 |

| | | |
|----------|--|-----------|
| 4.5 | Versuchsvorbereitung | 30 |
| 4.5.1 | Automatisierungs-Skripte | 30 |
| 4.5.2 | Definition der Nutzungsszenarien | 31 |
| 4.5.3 | Definition zweier Anwendungszustände | 32 |
| 4.6 | Versuchsdurchführung | 34 |
| 4.6.1 | Ablauf des Messverfahrens | 34 |
| 5 | Evaluation und Fazit | 37 |
| 5.1 | Ergebnis der Messungen | 37 |
| 5.1.1 | Berechnung von Leerlauf und Nutzungsszenario | 37 |
| 5.1.2 | Baseline-Ergebnisse | 37 |
| 5.1.3 | Leerlauf-Ergebnisse | 40 |
| 5.1.4 | Nutzungsszenario-Ergebnisse | 45 |
| 5.2 | Bewertung der Messergebnisse | 53 |
| 5.3 | Bewertung des Messkonzepts | 54 |
| 5.4 | Fazit und Ausblick | 55 |
| | Literaturverzeichnis | 57 |
| | Abbildungsverzeichnis | 62 |
| | Tabellenverzeichnis | 64 |
| | Abkürzungsverzeichnis | 65 |
| | Anlagen | 66 |

1 Einleitung

1.1 Motivation

Der Klimawandel beschreibt die langfristigen Veränderungen von Faktoren wie der Temperatur, des Niederschlages und der Meeresströmungen, welche in den letzten Jahrzehnten durch zahlreiche Studien nachgewiesen wurden [Car22]. Ursache für diesen enormen Anstieg ist der Treibhauseffekt. Die in der Erdatmosphäre emittierten Treibhausgase sind in den letzten Jahrzehnten, in denen sich der Klimawandel bemerkbar macht, durch die rasant steigende Weltbevölkerung sowie die voranschreitende Industrialisierung, drastisch angestiegen. [www16] Das Gas mit der größten Auswirkung auf die globale Erderwärmung ist Kohlendioxid (CO₂).

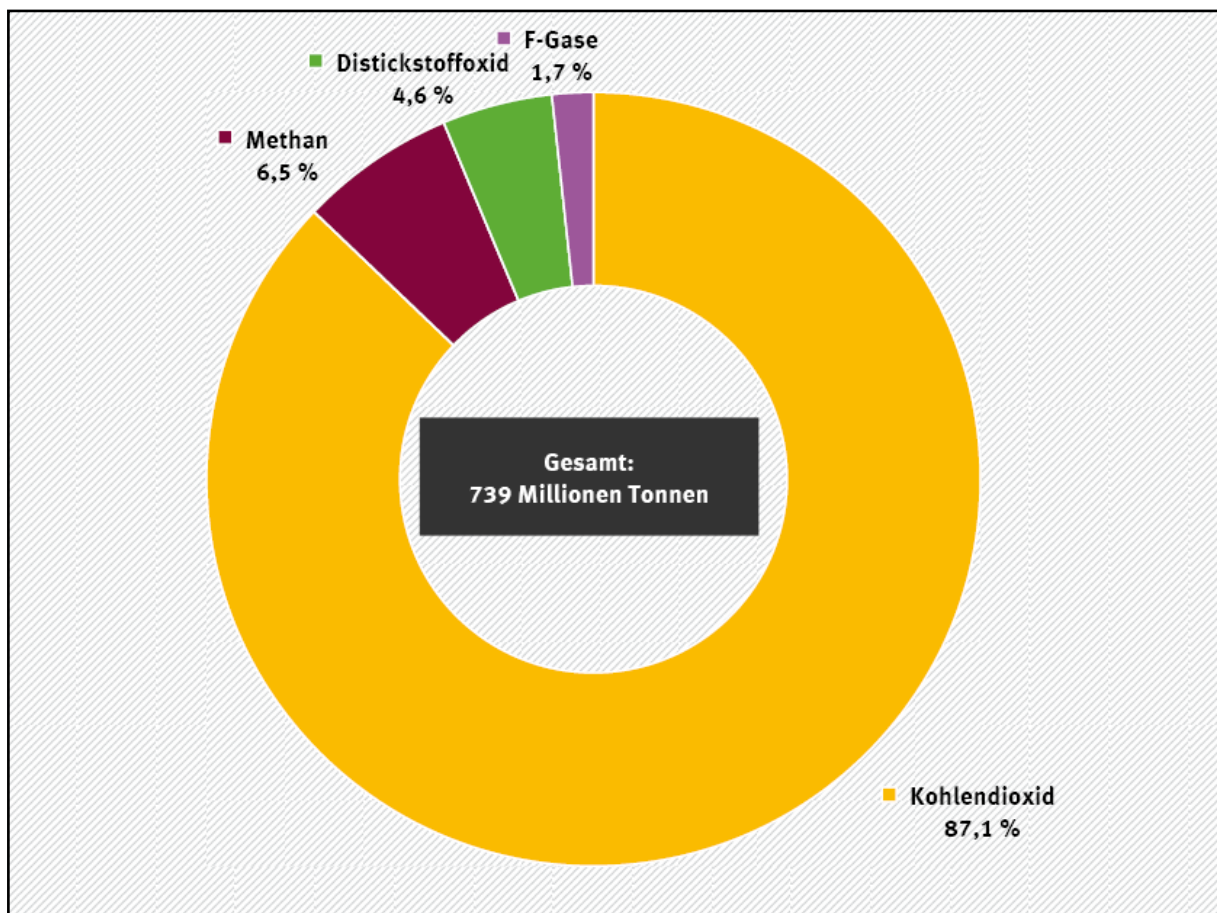


Abbildung 1: Anteile der Treibhausgase an den Emissionen aus dem Jahr 2020 [Umw22]

Abbildung 1 veranschaulicht die Anteile der Treibhausgase an den Emissionen und zeigt deutlich die Bedeutung von CO₂. Im Jahr 2020 betrugen die CO₂-Emissionen 87,1 Prozent von insgesamt 739 Millionen Tonnen [Umw22]. Die Differenz zum Methangas, welches mit einem Anteil von 6,5 Prozent das zweitgrößte Treibhausgas ist, beträgt rund 80 Prozent, wodurch verdeutlicht wird, wie wichtig es ist, sich genauer mit dem Gas auseinanderzusetzen. CO₂ entsteht dabei aus der Energiegewinnung durch Verbrennung von Stein- und Braunkohle, sowie Erdöl und -gas. Der durch hohe Treibhausgas-Emissionen resultierende Klimawandel entsteht also hauptsächlich durch den hohen Energiebedarf der Menschen. Dazu fehlen saubere Energiequellen, um diesen Bedarf zu decken [www16].

Die Informationstechnologie ist längst ein fester Bestandteil unseres Lebens sowie auch unserer Umwelt. Durch die ständige Weiterentwicklung von Produkten der Informations- und Kommunikationstechnik (IKT) sowie allgemein durch die Digitalisierung, werden die CO₂-Emissionen durch den damit einhergehenden Energieverbrauch beeinflusst. Auf digitaler Ebene ist es demnach relevant, Software auf ihren Energie- und Ressourcenverbrauch zu untersuchen. Eine Studie der Bitkom Akademie aus dem Jahr 2020 liefert das Ergebnis, dass digitale Technologie fast die Hälfte dazu beitragen kann, dass Deutschland seine Klimaziele bis zum Jahr 2030 erfüllt [Aka22]. Eine Voraussetzung dafür ist, dass die Energieeffizienz fortlaufend untersucht und voran gebracht wird [AE15].

Die Forschungsgruppe *Green Software Engineering*, welche zum Institut für Softwaresysteme (ISS) des Umwelt-Campus Birkenfeld gehört, beschäftigt sich mit der Energie- und Ressourceneffizienz von Software. Eines der Ziele dabei ist die Zertifizierung von nachhaltigen Softwareprodukten. Der *Blaue Engel* ist das Umweltzeichen der Bundesregierung und wurde im Jahr 2020 um Kriterien für ressourcen- und energieeffiziente Softwareprodukte erweitert [Inf22]. Aktuell gibt es den Blauen Engel nur für Desktop-Software, womit selbstverständlich nicht alle Arten von Softwareprodukten abgedeckt sind. Die Green Software Engineering Arbeitsgruppe plant bis Ende des Jahres 2023 auch Client-Server-Systeme und mobile Anwendungen mit dem Blauen Engel für ressourcen- und energieeffiziente Softwareprodukte zertifizieren zu können [Sci22a].

Diese Ausarbeitung entsteht in Kooperation mit dem ISS. Demnach werden durch das ISS bzw. durch die Green Software Engineering Arbeitsgruppe bekannte Modelle als Basis verwendet.

1.2 Ziel der Thesis

Der letzte Abschnitt zeigt deutlich, warum es wichtig ist, sich mit dem Einfluss von Software auf den Klimawandel auseinanderzusetzen. Eine Möglichkeit dafür ist die Untersuchung der Energie- und Ressourceneffizienz von verschiedenen Softwareprodukten. In Abschnitt 4.3.1 wird der Aufbau des aktuellen Messverfahrens von Software des ISS ausführlich erklärt. Wird eine Software gemessen, müssen dabei einige Faktoren beachtet werden, was im Endeffekt viel Zeit kosten kann. Zusätzlich reicht es nicht aus, ein Softwareprodukt nur einmal auf Energie- und Ressourceneffizienz zu messen. Da sich eine Software stetig weiterentwickelt, ist es ebenso möglich und wahrscheinlich, dass aus einer nachhaltigen Software über mehrere Monate oder Jahre eine Software wird, die die Kriterien der Nachhaltigkeit nicht mehr erfüllt. Um repetitive Softwaremessungen bzw. dazugehörige Aufgaben zu vermeiden, soll ein neues Messverfahren entwickelt werden.

Das Konzept basiert dabei auf dem Automatisierungswerkzeug Continuous Integration, welches im Abschnitt 2.3 erklärt wird. Die Softwaremessungen sollen vollautomatisiert laufen, sodass der Entwickler das zu messende Softwareprojekt lediglich an einer Stelle anlegen muss. Die Messungen sollen täglich in der Nacht durchgeführt werden, sodass am frühen Morgen bei Arbeitsbeginn bereits die Messergebnisse von gestern vorliegen und die Entwickler wissen, inwiefern sich die Energie- und Ressourceneffizienz bei welchen Änderungen am Quellcode angepasst hat. Die Messwerte selbst sollen dabei visualisiert werden. Darunter zählt die Betrachtung der Energie- und Ressourceneffizienz der jeweiligen Software.

Ziel dieser Thesis ist die Konzeption und Entwicklung eines solch wie beschriebenen automatisierten Messverfahrens, um das ISS bei der Messung und Bewertung von Software auf ihre Ressourceneffizienz zu unterstützen.

1.3 Struktur der Thesis

In dieser Thesis werden die folgenden Themen vorgestellt:

- Grundlagen der Green IT, Qualitätskontrolle und Continuous Integration
- Integration von Aspekten der Green IT in den Softwareentwicklungsprozess
- Versuchsaufbau und -durchführung eines automatisierten Messverfahrens zur Bestimmung der Ressourceneffizienz von Software
- Bewertung der Ressourceneffizienz der Messwerte
- Bewertung des Messverfahrens und Ausblick mit Handlungsempfehlungen für die Softwareentwicklung

Die Grundlagen, bestehend aus den beiden größeren Kapiteln Green IT und Qualitätskontrolle, werden nach einer kurzen Einführung in das Thema genauer erklärt. Die dabei beschriebenen Konzepte und Aspekte werden anschließend im dritten Kapitel aufgegriffen und miteinander verbunden. Anschließend wird der Aufbau und die Durchführung des neuen Messkonzepts vorgestellt und ebenso werden dabei die Unterschiede zum bisherigen Messaufbau aufgelistet. Kapitel 5 stellt zum einen die visualisierten Messergebnisse vor und trifft Aussagen zur Ressourceneffizienz, zum anderen wird das neue Messverfahren bewertet. Dazu gibt es einen Ausblick, wie die Ergebnisse dieser Thesis weiterverwendet werden können sowie mehrere Handlungsempfehlungen, wie sich die Softwareentwicklung in Zukunft anpassen sollte.

2 Grundlagen

2.1 Green IT

2.1.1 Bedeutung und Ziele

„Unter *Green IT* sind umweltverträgliche Produkte und Dienstleistungen der Informations- und Kommunikationstechnik (IKT) sowie der Nutzung von IKT zur Umweltschonung zu verstehen. Dies umfasst die Berücksichtigung des gesamten Lebenswegs von IKT-Produkten sowie deren Auswirkungen auf das Klima und andere Umweltwirkungen, wie zum Beispiel die Inanspruchnahme kritischer Rohstoffe.“ [Bun22a]

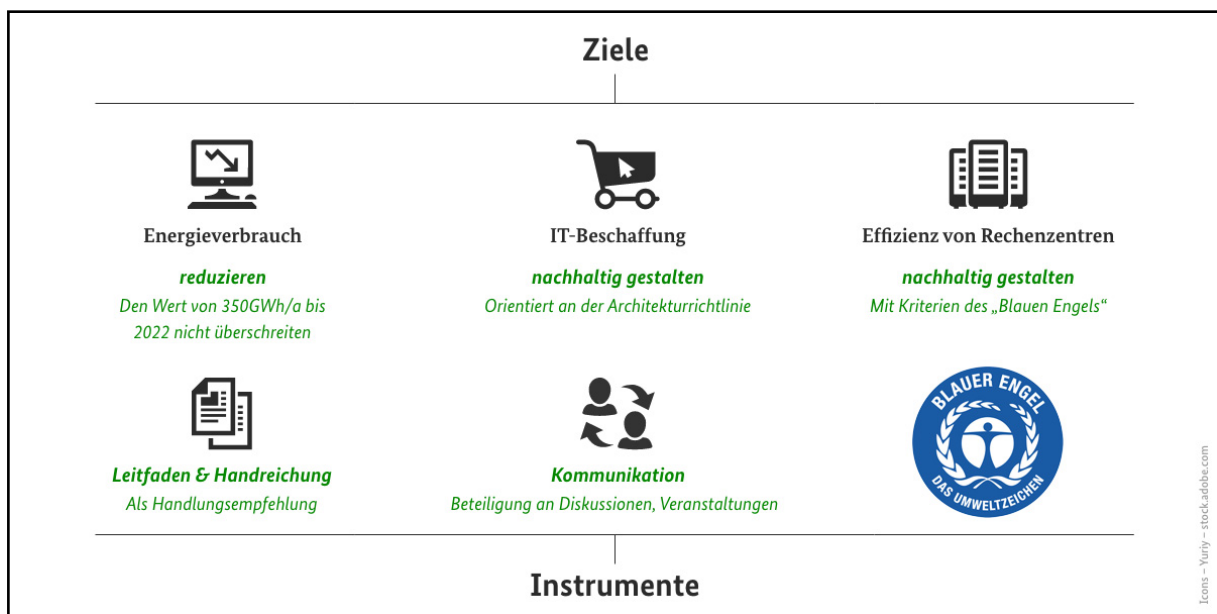


Abbildung 2: Ziele und Instrumente der Green IT [Bun22b]

Das Bundesministerium für Umwelt, Naturschutz, nukleare Sicherheit und Verbraucherschutz stellt in der obigen Abbildung die wichtigsten Ziele und Instrumente der Green IT vor [Bun22b]. Der Fokus liegt dabei vor allem auf der Reduktion des Energieverbrauchs von IKT-Produkten sowie der nachhaltigen Gestaltung nach Kriterien des Blauen Engels, dem Umweltzeichen der Bundesregierung. Im nächsten Abschnitt werden diese Ziele im Bezug zu nachhaltiger Software betrachtet.

2.1.2 Nachhaltige Software

Nachhaltige Software ist ein Teilbereich der Green IT. Darunter zu verstehen ist Software, deren direkte und indirekte negative Auswirkungen auf Menschen, Gesellschaft und Umwelt über ihren gesamten Lebenszyklus hinweg minimal sind und die bestenfalls einen zusätzlichen positiven Beitrag zur nachhaltigen Entwicklung leistet [DNH10]. Als Entwickler sollte man dementsprechend darauf achten, die Komplexität der zu entwickelnden Anwendung möglichst klein zu halten. Die Komplexität von Software ist an vielen Stellen zu erkennen, beispielsweise in der Algorithmik, der Struktur, oder dem notwendigen Software-Stack [Inf22].

Auf Software-Ebene wird die Ressourceneffizienz stark durch die Wahl des richtigen Frameworks und des oftmals damit einhergehenden Paketoverheads beeinflusst [Inf22]. Verwendet ein Entwickler für einfache Probleme in der Implementierung jedes Mal ein zusätzliches Paket, beispielsweise von NPM, steigt nicht nur die Wahrscheinlichkeit höherer Komplexität, sondern auch das Sicherheitsrisiko der eigenen Anwendung. In vielen Fällen ist es natürlich empfehlenswert, auf fertige Pakete zurückzugreifen. Jedoch sollte der Entwickler, mit Ziel einer nachhaltigen Software, erst überlegen, ob das Problem nicht durch z.B. einfachere Algorithmen zu lösen ist.

Auch für die Hardware-Ebene ist nachhaltige Software relevant, da bei der Ausführung einer Software ständig Hardware-Ressourcen verbraucht werden. Eine hohe Hardware-Auslastung steht in direktem Zusammenhang mit hohem Energieverbrauch.

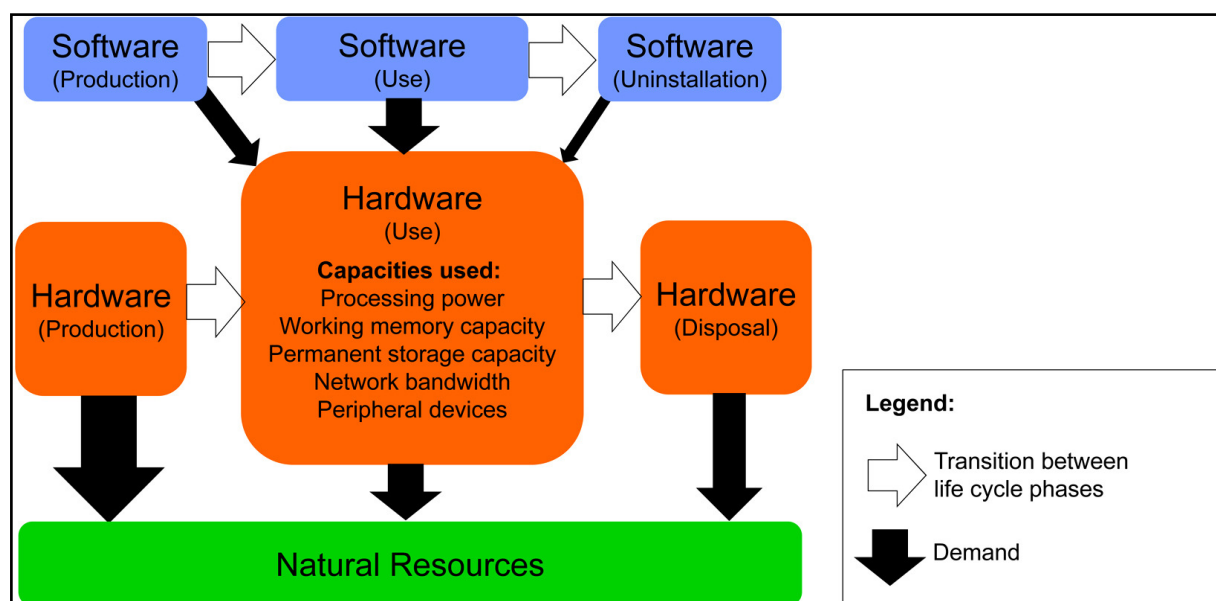


Abbildung 3: Lebenszyklen von Hardware und Software [Ker+18]

Abbildung 3 veranschaulicht den Zusammenhang der Lebenszyklen von Hardware und Software. Die horizontale Dimension zeigt die Lebenszyklen von Hardware und Software, auf vertikaler Ebene wird die Beanspruchung von Ressourcen dargestellt. Durch die Nutzung von Software wird entsprechende Hardware benötigt, die wiederum natürliche Ressourcen beansprucht. Durch steigende Software-Anforderungen muss also immer wieder neue Hardware angeschafft werden, die entsprechende Möglichkeiten bietet. [Ker+18]

Zusätzlich muss der Netzdatenverkehr beachtet werden, welcher durch eine Software mit Internetnutzung verursacht wird. Das Internet funktioniert über große Server, welche pausenlos in Betrieb sind und dafür Strom brauchen. Da jede Datei durch verschiedene Server geleitet, oder gespeichert werden muss sowie Suchanfragen verwaltet werden müssen, steigt der Energieverbrauch an [Sch19].

2.1.3 Bewertung von Software

Kriterienkatalog

Der letzte Abschnitt hat gezeigt, was genau nachhaltige Software ist und welche Kriterien auf Seiten der Hardware und Software dafür eine Rolle spielen. Um eine Software nun konkret auf ihre Ressourceneffizienz analysieren zu können, braucht es einen Kriterienkatalog, der eine Software aus technischer Sicht bewertet. Zu Beginn des Jahres 2020 hat Deutschland mit dem Blauen Engel für ressourcen- und energieeffiziente Softwareprodukte ein einheitliches Umweltzeichen für nachhaltige Software veröffentlicht [Inf22]. Eine Software wird als besonders nachhaltig eingestuft, wenn sie die drei Kategorien des Kriterienkatalogs erfüllt. Dies betrifft die Aspekte Ressourcen- und Energieeffizienz, die potenzielle Lebensdauer der Hardware und die Benutzerunabhängigkeit [Blu22].

Standardnutzungsszenarien

Ein Standardnutzungsszenario dient der Beschreibung von Arbeitsabläufen und Nutzungskontext der Benutzer [msg20]. Dabei soll eine realitätsnahe Nutzung einer Software simuliert werden [Grö+18]. Im Kontext der Green IT können einheitliche Szenarien dazu verwendet werden, um den Energieverbrauch bzw. die Hardware-Inanspruchnahme von verschiedenen Softwareprodukten, oder sogar der gleichen Software in verschiedenen Zuständen, miteinander zu vergleichen [Grö+18]. Ein solches Nutzungsszenario kann dabei ganz verschieden aussehen. Benutzer können die festgelegten Arbeitsabläufe manuell steuern, oder im besseren Fall wird ein Automatisierungstool zur Ausführung des Standardnutzungsszenarios verwendet.

2.2 Qualitätskontrolle

Kapitel 2.1 hat gezeigt, was nachhaltige Software ist und anhand welcher Kriterien Entwickler deren Softwareprodukte bewerten können. Ziel dieser Thesis ist die Entwicklung und Konzeption eines automatisierten Messverfahrens, mit dem die Ressourceneffizienz von Software-Applikationen gemessen und evaluiert werden kann. Dieses Kapitel stellt dafür alle Grundlagen der Qualitätskontrolle vor, die im späteren Verlauf der Thesis vorausgesetzt werden. Insbesondere in Kapitel 3 wird der Zusammenhang zwischen Qualitätskontrolle und nachhaltiger Software deutlich.

2.2.1 Wichtigkeit von Softwarequalität

In der heutigen Zeit werden die Nutzer von IKT-Produkten täglich immer wieder mit neuen und moderneren Anwendungen überschüttet. Um die Qualität der entstehenden Softwareprodukte zu gewährleisten, ist es daher von größter Wichtigkeit, einen sauberen und qualitativ hochwertigen Quellcode zu schreiben. Durch unsauberen bzw. schlecht strukturierten Code erhöht sich die Entwicklungsgeschwindigkeit und das Potenzial von Fehlern sowie den damit verbundenen Folgen steigt an. [Beh11]

Anwendungen werden teilweise von Millionen bis hin zu Milliarden von Menschen genutzt. Sollte eine Software dabei aufgrund von diversen Problemen abstürzen oder sogar für mehrere Stunden komplett ausfallen, könnte dies fatale Folgen haben. Solche Probleme können durch schlechten Quellcode auftreten, oder auch durch jede Art von technischer Anpassung, sei es eine Änderung der Datenbank, der Serverkonfiguration oder Sonstiges. Im Jahr 2021 führten bei Facebook Konfigurationsänderungen an den Backbone-Routern, die den Netzwerkverkehr zwischen den Rechenzentren koordinieren, zu einer Unterbrechung der Kommunikation [wat22].

Finanzexperten schätzten den Umsatzausfall von Facebook basierend auf dem Gewinn des zweiten Quartals des Jahres dabei auf rund 99,75 Millionen Dollar. [Sen21].

Um solche Fehler frühzeitig zu erkennen, sollten die entwickelten Softwareprodukte daher stabil sein und in der Entwicklungsphase bestenfalls getestet werden.

2.2.2 Qualitätsmerkmale

Um die Qualität und Bewertung von Softwareprodukten anhand von verschiedenen Kriterien nachzuvollziehen, braucht es ein einheitliches Konzept. Die internationale Norm ISO/IEC 25010 *Software product Quality Requirements and Evaluation (SQuaRE)* wurde von dem Normungsgremium *Software and systems engineering* erstellt und gilt seit dem Jahr 2005 unter anderem als Nachfolger der Norm ISO/IEC 9126, welche ein Qualitätsmodell für die Evaluation von Softwareprodukten aufgestellt hat. [ISO22a] Die untergeordnete Norm ISO/IEC 25010 beinhaltet in der neuen Normreihe das Qualitätsmodell und Leitlinien, welches in Abbildung 4 dargestellt wird. Nach der Norm gibt es acht verschiedene Qualitätsmerkmale von Software, welche wiederum in weitere Unterpunkte unterteilt sind. Dazu zählen unter anderem die Effizienz, Benutzbarkeit, oder auch die Wartbarkeit.

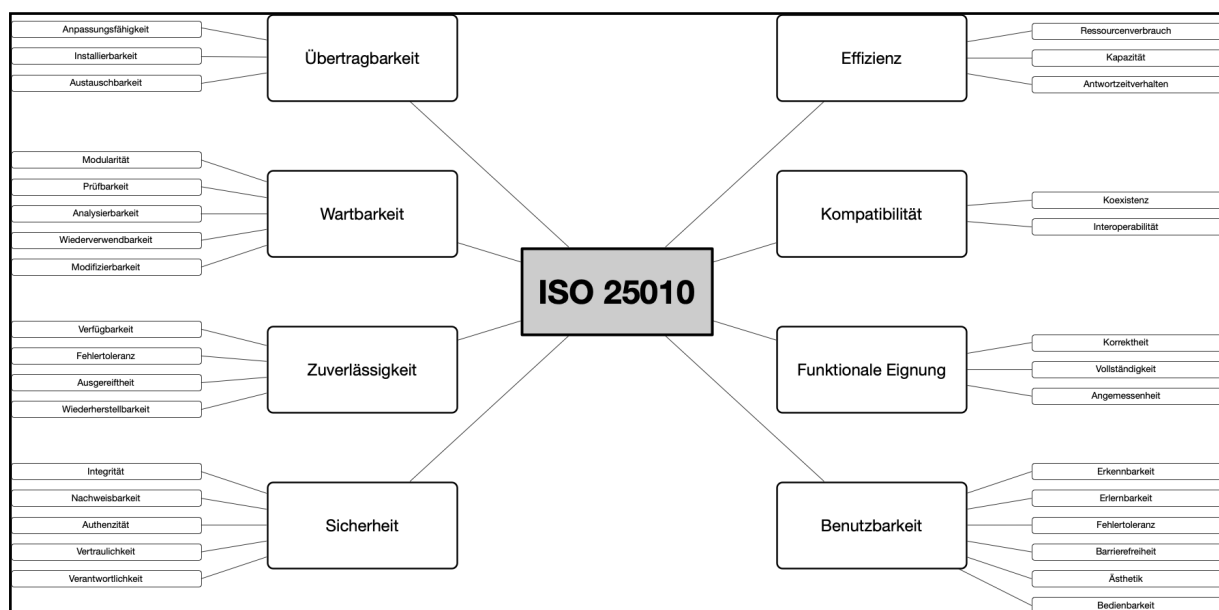


Abbildung 4: ISO/IEC 25010 als Qualitätsmodell [Plö20]

2.2.3 Vorgehensmodelle

Ein wichtiger Bestandteil, um die Einhaltung der in der ISO/IEC 25010 Norm verankerten Qualitätsmerkmale besser zu kontrollieren bzw. gewährleisten zu können, sind Vorgehensmodelle. In der Softwareentwicklung gibt es viele verschiedene Ansätze, in welchen Zyklen und in welchen Schritten eine Software entwickelt wird. Deshalb werden während des Entwicklungsprozesses Vorgehensmodelle als Orientierung verwendet, die für den gesamten Software-Lebenszyklus genau bestimmen, welche Aktivitäten in welcher Reihenfolge ausgeführt werden [Shi19].

Bekannte Vorgehensmodelle sind beispielsweise das Wasserfallmodell, das V-Modell, oder auch Scrum, welches einen agilen Ansatz verfolgt. Am Beispiel von Scrum wird in diesem Abschnitt das Vorgehen einer zu entwickelnden Software vorgestellt.

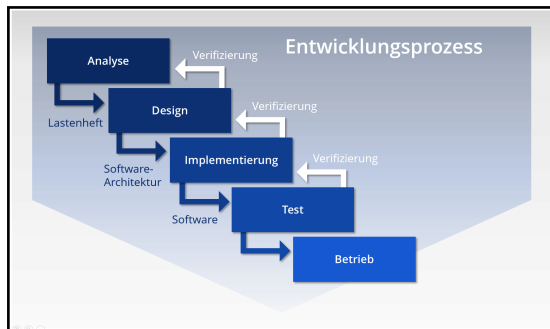


Abbildung 5: Wasserfallmodell [ION22]

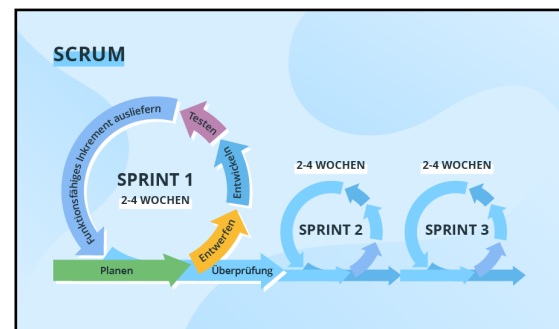


Abbildung 6: Scrum [Shi19]

Abbildung 6 zeigt einen Ablauf der Softwareentwicklung mit Scrum, wie er für ein agiles Projekt aussehen könnte. Neue Versionen eines Softwareprodukts werden in kurzen Iterationen ca. alle zwei bis vier Wochen ausgerollt. Ein einzelner Sprint umfasst dabei jeweils Planung, Entwurf, Entwicklung, Testen, Auslieferung und Überprüfung der Software [Shi19]. Der Vorteil von Scrum gegenüber anderen Vorgehensmodellen wie dem Wasserfallmodell, welches in Abbildung 5 dargestellt wird, sind die kürzeren Iterationen. Der gesamte Softwareentwicklungsprozess wird also verhältnismäßig häufiger wiederholt und die Software wird somit auch häufiger auf ihre funktionalen bzw. nicht-funktionalen Anforderungen getestet.

2.2.4 Softwaretests

Softwaretests sind, in welcher Form sie auch durchgeführt werden, ein wesentlicher Bestandteil der Qualitätskontrolle. Durch das Testen einer Software sollen die im vorletzten Abschnitt vorgestellten Qualitätsmerkmale sichergestellt werden. Ziel dabei ist ausschließlich die Fehlererkennung, sodass mögliche Fehler im nächsten Schritt behoben werden können. Softwaretests können dabei statisch oder dynamisch ausgeführt werden und je nach Vorgehensmodell, nach dem entwickelt wird, häufiger oder eher weniger häufig durchgeführt werden.

2.2.5 Verschiedene Arten von Softwaretests

Softwaretests lassen sich in zwei verschiedene Typen unterscheiden, zum einen gibt es funktionale und zum anderen nicht-funktionale Tests. Unter Betrachtung verschiedener Aspekte eines Softwareprodukts, werden auch unterschiedliche Testtypen benötigt. [Lee20]

Funktionale Tests werden entwickelt, um geschäftskritische Funktionen zu überprüfen. Dabei soll sichergestellt werden, dass eine Software wie erwartet und ohne Fehler läuft [Lee20]. Nicht-funktionale Tests betrachten, wie die Software sich unter bestimmten Umständen verhält. Dabei geht es um Erwartungen und Leistung unter Simulation von gewissen Zuständen [Tec20]. Abbildung 7 zeigt eine grobe Übersicht über die bekanntesten Testarten, unterschieden durch Funktionalität bzw. Nicht-Funktionalität.

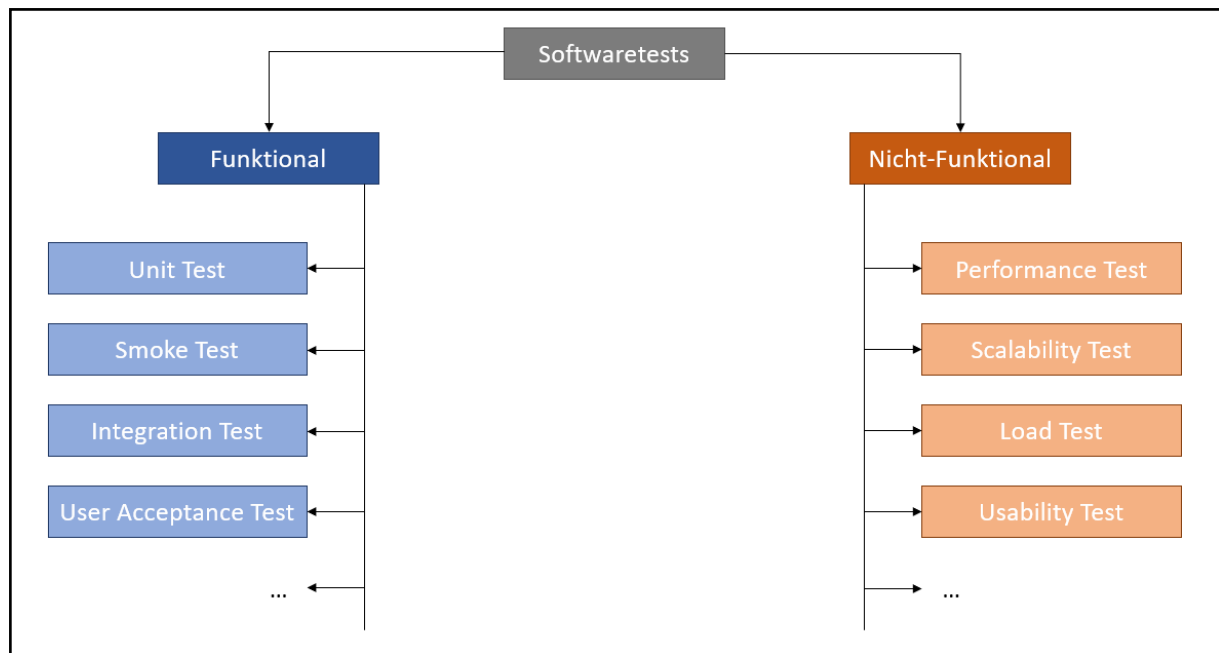


Abbildung 7: Arten von Softwaretests (eigene Darstellung, basierend auf [Hac22], [Tec20])

Softwaretests lassen sich manuell von einem Nutzer ausführen oder mit Hilfe eines zusätzlichen Tools auch einfach automatisieren. Als Grundlage für das weitere Vorgehen der Thesis, wird im Folgenden jeweils ein Test für Funktionalität sowie Nicht-Funktionalität vorgestellt.

Unit Test

Zur Überprüfung korrekter Funktionen einer Software kann z.B. ein *Unit Test* entwickelt werden. Diese sind sehr einfach gestaltet und erfolgen nah an der Quelle der Anwendung. Getestet werden bei diesem Verfahren einzelne Methoden und Funktionen der von der Software verwendeten Klassen, Komponenten oder Module. [Atl22]

Performance Test

Zum Testen des Verhaltens eines Systems unter erheblicher Belastung, können *Performance Tests (Leistungstests)*, ein Beispiel für nicht-funktionale Softwaretests, implementiert werden. Das Ziel dieses Verfahrens ist die Ermittlung der Zuverlässigkeit, Robustheit sowie Verfügbarkeit. Dies kann in Form von Beobachtung der Antwortzeiten bei der Ausführung einer hohen Anzahl von Anfragen geschehen. [Atl22]

2.3 Continuous Integration

2.3.1 Bedeutung und Ablauf

Continuous Integration (CI) ist ein Werkzeug, welches in der Integrationsphase der Softwareentwicklung verwendet wird [Wag22]. CI wird ganz speziell zur automatisierten Integration von Codeänderungen genutzt, die mehrere Mitarbeiter an einem einzigen Softwareprojekt vornehmen. Den Entwicklern wird ermöglicht, Codeänderungen häufig an einem zentralen Platz zusammenzuführen, an dem z.B. Bauprozesse und Tests der Software ausgeführt werden [Atl21a]. Diese zentrale Ablage wird Repository genannt und zur Verwaltung wird i.d.R. ein Versionsverwaltungssystem verwendet. Der de facto Standard ist dabei das Tool *git* [Atl21b].

Ein Bauprozess führt normalerweise ein Bash-Skript aus, welches die Befehle zur erfolgreichen Ausführung und Komprimierung enthält. Für eine Node.js Anwendung beispielsweise müssen zur erfolgreichen Ausführung mit Hilfe des *Node Package Manager (NPM)* die Abhängigkeiten der Software installiert werden. Der Bauprozess an sich kann je nach Softwareprojekt ganz verschieden aussehen. Der Entwickler könnte z.B. eine JavaScript-Datei implementieren, die das gesamte Projekt zur Software-Sicherheit sowie auch zur Einsparung von Speicherplatz komprimiert. Die Dateien, welche als Ergebnis entstehen, können dann nach Belieben auf dem Server zu einem anderen Verzeichnis geschoben werden. Zusätzlich dazu kann der Quellcode des Repositorys mit CI auch auf funktionale bzw. nicht-funktionale Anforderungen getestet werden. Die verschiedenen Tests werden ebenfalls über ein Bash-Skript gestartet.

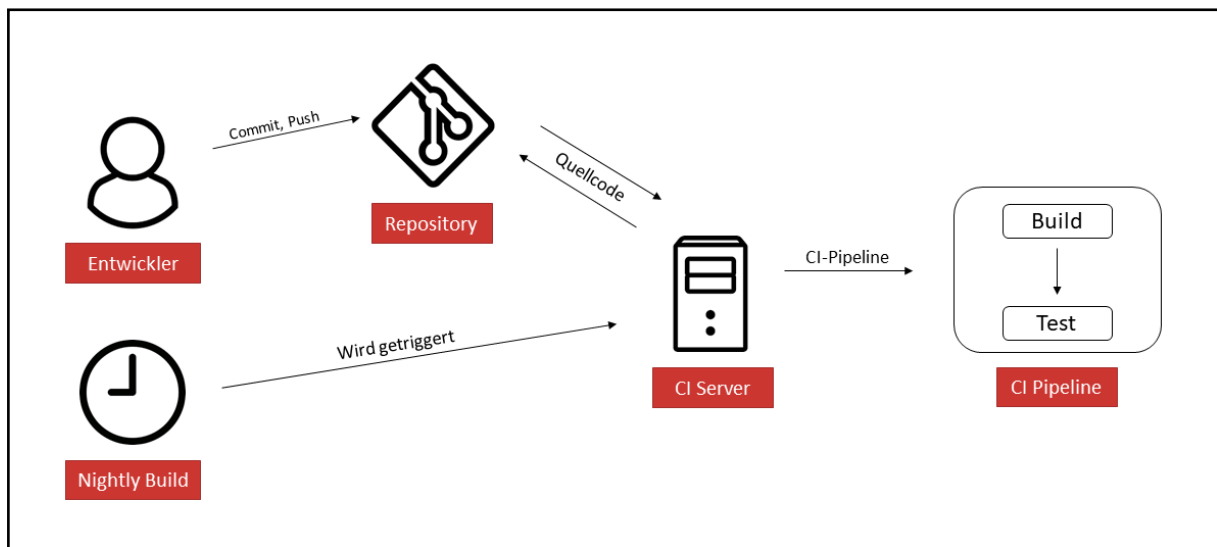


Abbildung 8: Ablauf von Continuous Integration (eigene Darstellung, basierend auf [Add22])

Abbildung 8 zeigt ein Beispiel für den Ablauf von Continuous Integration. Die erste Möglichkeit, den Prozess zu starten, erfolgt durch den Entwickler, der seine Codeänderungen mit den Operationen *Commit* und *Push* an das Repository sendet. Dies löst durch eine in dem CI-Tool konfigurierte Option automatisch die CI-Pipeline aus. Daraufhin wird der Quellcode aus dem Repository gebaut und bei Erfolg anschließend getestet. Ist der Build nicht erfolgreich, wird automatisch eine Email an den Verantwortlichen gesandt. Die andere Möglichkeit, wie sich der Prozess starten lässt, nennt sich *Nightly Build*. Die Konfiguration erfolgt ebenfalls über das CI-Tool, wobei man hier einen festen zeitlichen Start des Prozesses festlegen kann. I.d.R. wird eine Zeit wie z.B. 03:00 Uhr Nachts verwendet, da zu dieser Zeit kein Entwickler an einem Projekt arbeiten sollte.

2.3.2 Analyse von CI-Tools

Ohne ein spezielles Tool werden die Abläufe bei der kontinuierlichen Integration weniger einfach und insgesamt komplexer. Speziell ausgelegte Software kann dabei Unterstützung beim Aufbau eines Repositorys, beim Testing und Building, sowie der Versionskontrolle bzw. der kontinuierlichen Integration selbst bieten [ION21]. Für diese Thesis wurden zur Analyse des CI-Tools spezielle Kriterien aufgestellt. Wichtig ist bei der Auswahl vor allem, dass das Tool frei verwendbar bzw. auch kostenlos ist und sich leicht installieren, wie auch bedienen lässt. Zusätzlich wünschenswert ist die Plattform-Unabhängigkeit sowie das Gewicht.

| Jenkins | GitLab | Travis CI |
|-----------------------|------------------------|-------------------------|
| Frei verwendbar | Frei verwendbar | Frei verwendbar für OSS |
| Einfache Installation | Einfache Installation | Einfache Installation |
| Benutzerfreundlich | Benutzerfreundlich | Benutzerfreundlich |
| Plattformübergreifend | Plattformübergreifend | Mac, Linux, iOS |
| Zahlreiche Plugins | Issue Tracking | Kein Hosting Server |
| Leichtgewichtig | Git Versionsverwaltung | Integration mit GitHub |

Tabelle 1: Vergleich von CI-Tools

Die drei oben verglichenen Tools erfüllen alle die aufgestellten Hauptkriterien. Dennoch hat jedes Tool seine individuellen Stärken, die in der obigen Tabelle ebenfalls genannt sind. GitLab lässt sich neben dem CI-Prozess auch gut als Weboberfläche für die Git Versionsverwaltung verwenden. Dazu kommt das Issue Tracking, d.h. dass GitLab sogar als Plattform für Projektmanagement verwendet werden kann. Zum Ausführen der Jobs wird ein Runner benötigt, womit später auch mehrere Jobs gleichzeitig laufen können. Diese Runner müssen entsprechend konfiguriert werden, was Zeit kostet und bei Einsteigern zu Problemen führen kann. Als Allrounder-Tool wäre GitLab also sehr gut geeignet. Für diese Thesis eignet es sich durch das größere Gewicht aber eher weniger.

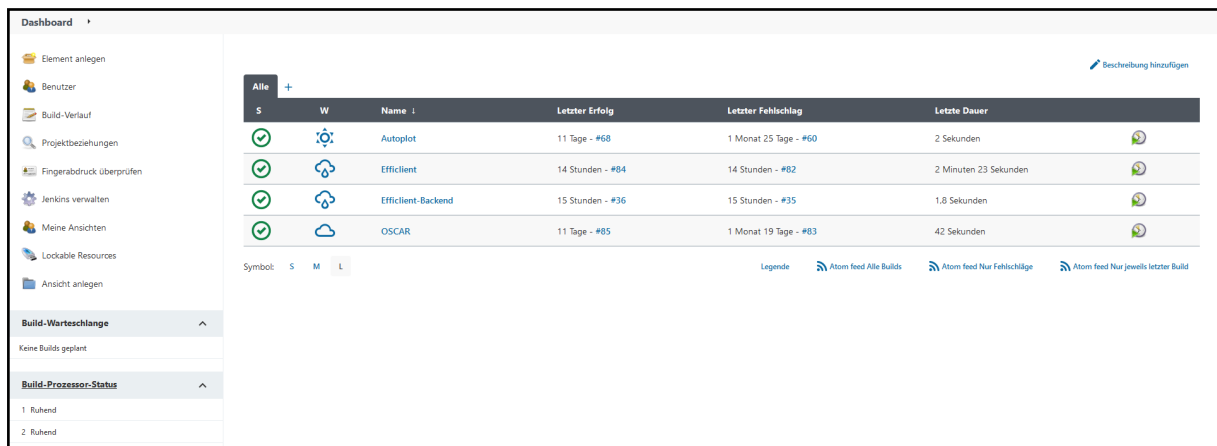
Travis CI ist eher mit Jenkins und weniger mit GitLab zu vergleichen, d.h. Funktionen wie das Issue Tracking oder die Git Versionsverwaltung fehlen komplett. Dafür gibt es mit Travis aber eine sehr gute von Haus aus mitgebrachte Integration zu GitHub. Ebenfalls ausgezeichnet wird das Tool dadurch, dass kein Hosting Server notwendig ist. Die Konfiguration ist hier also am schnellsten und einfachsten. Zu bemängeln ist das Kostenprinzip des Projektes. Travis CI ist nur für Open Source Software (OSS) kostenlos und frei verwendbar, ansonsten gibt es eine Preisliste. Zudem wird Windows als Betriebssystem nicht unterstützt, wobei für diese Thesis sowieso ein Linux System verwendet wird.

Jenkins glänzt durch seine Einfachheit und Leichtgewichtigkeit. Innerhalb von Minuten lässt sich die Software installieren sowie auch konfigurieren und der Entwickler kann so schnellstmöglich seine CI-Pipeline aufbauen. Eine zusätzliche Stärke ist die Verfügbarkeit unzählbarer Plugins, durch welche sich Jenkins mit vielen weiteren Softwareprodukten verknüpfen lässt. So gibt es z.B. ein Plugin zur Verbindung mit GitHub oder auch GitLab.

Für diese Thesis fällt GitLab als CI-Tool wie erwähnt raus, da die meisten und nützlichsten Funktionen schlicht nicht benötigt werden. Da im Messlabor des Institut für Softwaresysteme am Umwelt-Campus in Birkenfeld sowieso bereits ein Server steht, bietet es sich an, diesen auch zu nutzen und als CI-Server für Jenkins zu verwenden.

2.3.3 Konfiguration von Jenkins

Jenkins bietet eine Übersicht über alle verknüpften Projekte, welche in Abbildung 9 zu sehen sind.

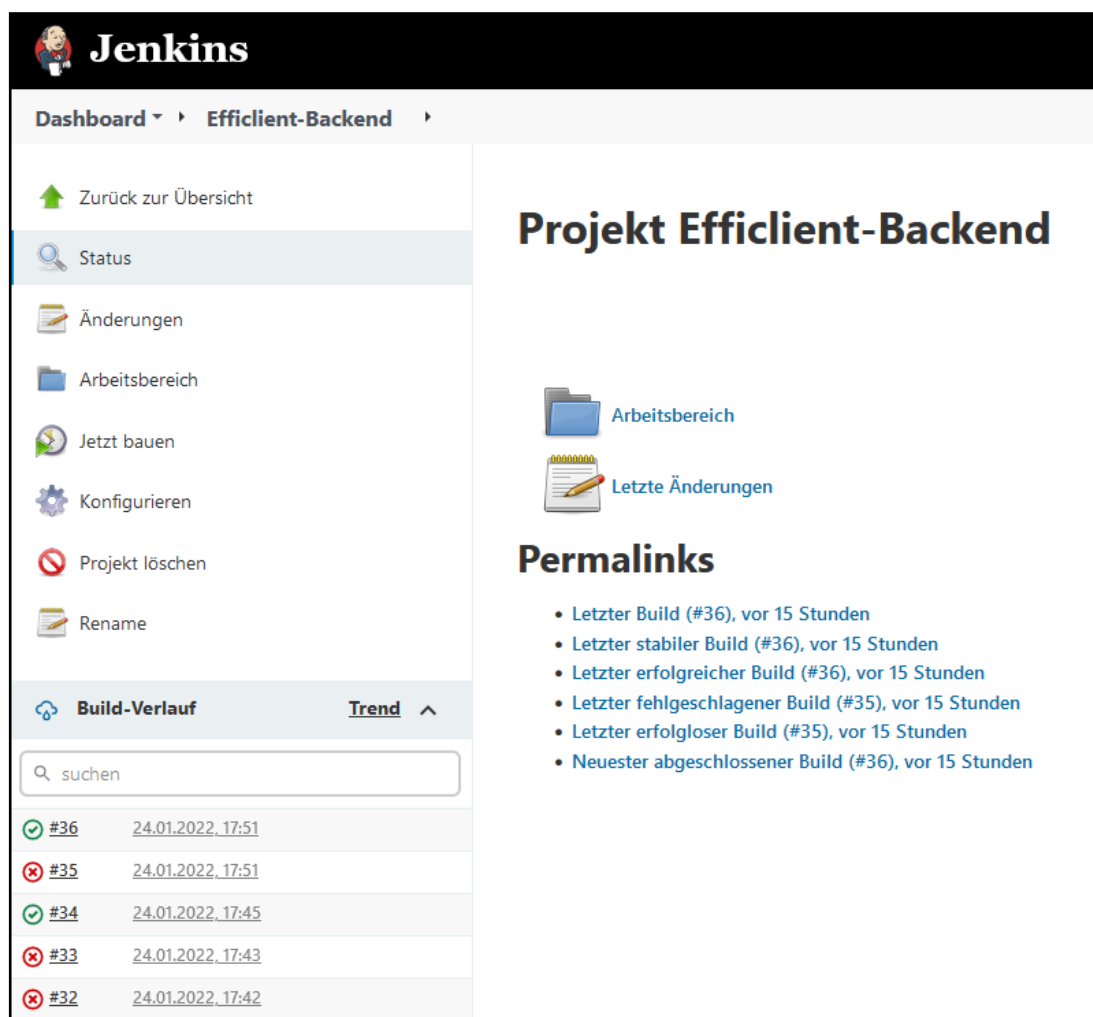


The screenshot shows the Jenkins Dashboard with a sidebar on the left containing navigation links like 'Element anlegen', 'Benutzer', 'Build-Verlauf', etc. The main area displays a table of projects with columns for status, name, last success, last failure, and last duration.

| S | W | Name | Letzter Erfolg | Letzter Fehlschlag | Letzte Dauer |
|---|----|-------------------|------------------|-----------------------|-----------------------|
| ✓ | ⚙️ | Autopilot | 11 Tage - #68 | 1 Monat 25 Tage - #60 | 2 Sekunden |
| ✓ | 🔗 | Efficient | 14 Stunden - #84 | 14 Stunden - #82 | 2 Minuten 23 Sekunden |
| ✓ | 🔗 | Efficient-Backend | 15 Stunden - #36 | 15 Stunden - #35 | 1.8 Sekunden |
| ✓ | ☁️ | OSCAR | 11 Tage - #85 | 1 Monat 19 Tage - #83 | 42 Sekunden |

Below the table, there are filters for 'Symbol' (S, M, L) and a legend for Atom feed links.

Abbildung 9: Übersicht über die Jenkins Oberfläche



The screenshot shows the Jenkins project page for 'Efficient-Backend'. It includes a sidebar with navigation options like 'Zurück zur Übersicht', 'Status', 'Änderungen', 'Arbeitsbereich', 'Jetzt bauen', 'Konfigurieren', 'Projekt löschen', and 'Rename'. The main content area displays the project name, a list of links for 'Arbeitsbereich' and 'Letzte Änderungen', and a section for 'Permalinks' listing various build links.

Projekt Efficient-Backend

Arbeitsbereich

Letzte Änderungen

Permalinks

- [Letzter Build \(#36\), vor 15 Stunden](#)
- [Letzter stabiler Build \(#36\), vor 15 Stunden](#)
- [Letzter erfolgreicher Build \(#36\), vor 15 Stunden](#)
- [Letzter fehlgeschlagener Build \(#35\), vor 15 Stunden](#)
- [Letzter erfolgloser Build \(#35\), vor 15 Stunden](#)
- [Neuester abgeschlossener Build \(#36\), vor 15 Stunden](#)

Build-Verlauf **Trend**

suchen

| | |
|-------|-----------------------------------|
| ✓ #36 | 24.01.2022, 17:51 |
| ✗ #35 | 24.01.2022, 17:51 |
| ✓ #34 | 24.01.2022, 17:45 |
| ✗ #33 | 24.01.2022, 17:43 |
| ✗ #32 | 24.01.2022, 17:42 |

Abbildung 10: Übersicht über ein einzelnes Jenkins Projekt

Wählt man ein bestimmtes Projekt aus der Liste aus, gelangt man zur Einzelansicht. Abbildung 10 zeigt dabei, wie ein Projekt in Jenkins aussehen könnte. Der Benutzer erhält eine Übersicht über den Build-Verlauf. Es ist auf den ersten Blick einsehbar, ob die letzten Builds erfolgreich waren, oder ob die CI-Pipeline fehlerhaft war. Interessant ist für diese Thesis insbesondere der Abschnitt *Konfigurieren*. Dort wird das Jenkins Projekt mit einem Repository verknüpft. Die Testprojekte dieser Thesis liegen auf einem GitLab-Repository. Durch ein Jenkins Plugin lassen sich die Projekte mit GitLab verknüpfen, wodurch es möglich ist, den Build-Status für jeden Commit unter der GitLab-Weboberfläche anzuzeigen. Zudem enthält die Konfiguration den Auslöser für die CI-Pipeline. Im Falle der obigen Abbildung wird ein zeitlicher Auslöser verwendet und das Projekt wird jede Nacht um 03:00 Uhr gebaut. Die CI-Pipeline an sich ist ebenfalls in der Konfiguration notiert.

```
1 #!/bin/bash
2 npm install
3 npm run build
```

Listing 1: Einfache CI-Pipeline

Das obige Bash-Skript ist ein Beispiel für eine sehr einfache CI-Pipeline einer Node.js Anwendung. Zuerst werden die Abhängigkeiten installiert und anschließend wird das Projekt gebaut. Die Funktionsweise hinter dem Bauprozess ist dabei im Quellcode versteckt.

2.3.4 Zwischenfazit

Die letzten Abschnitte haben die Bedeutung und den Ablauf von Continuous Integration geklärt. Zudem wurden die drei CI-Automatisierungstools Jenkins, GitLab und Travis CI miteinander verglichen, wobei die Konfiguration des gewählten Tools Jenkins genauer beleuchtet wurde. Alle der vorgestellten Tools können einfach installiert und bedient werden und eignen sich zur Automatisierung des Softwareprozesses. Von Vorteil ist der Aspekt, dass bei jedem Build sehr einfach automatisierte Softwaretests laufen können und den Entwicklern somit regelmäßig einen Überblick über Funktionen oder Leistungen der Applikation geben. Der Begriff Continuous Integration existiert dabei schon sehr lange und die Verwendung nimmt stetig zu [Qen22]. Laut einer Studie der *International Data Corporation* aus dem Jahr 2020 setzen fast vier von fünf der befragten Unternehmen auf die kontinuierliche Integration mit samt seinen Vorteilen [IDC22].

3 Qualitätskontrolle im Kontext von nachhaltiger Software

Der Abschnitt 2.1 dieser Thesis hat die Wichtigkeit von Green IT aufgezeigt und die Ziele, insbesondere im Zusammenhang mit nachhaltiger Software, aufgelistet. Abschnitt 2.2 erklärt, warum es nützlich ist, neben der Implementierung auch auf die Qualität der Software zu achten, am Beispiel von verschiedenen Softwaretests. Der Abschnitt 2.3 zeigt, wie die Integrationsphase eines Softwareprodukts mit Hilfe eines CI-Tools leicht automatisiert werden kann. Dieses Kapitel soll zeigen, wie sich die Aspekte Green IT, Qualitätskontrolle und Continuous Integration miteinander kombinieren lassen.

3.1 Qualitätsmodelle

Das Qualitätsmodell nach der Norm ISO/IEC 25010 dient zur Evaluation der Qualität von Software in Bezug auf funktionale und nicht-funktionale Anforderungen. Verschiedene Qualitätsmodelle sind nützlich, da sie dabei helfen, Anforderungen an Software zu formulieren, zu messen und zu beurteilen [Kla09]. Im Laufe des letzten Jahrzehnts wurden mehrere Qualitätsmodelle mit Fokus auf Nachhaltigkeit erstellt. Je nach Modell gibt es ganz verschiedene Ansätze wie nachhaltige Software im Kontext des ISO/IEC 25010 Standards bewertet werden könnte [GGP21].

3.1.1 Modelle

Gerstlacher et al. vergleichen in einem Artikel aus dem Jahr 2021 sieben verschiedene Qualitätsmodelle, welche alle auch den Aspekt nachhaltiger Software berücksichtigen [GGP21].

- Modell 1: Quality Model Green and Sustainable Software von Kern et al. [Ker+13] bzw. Naumann et al. [Nau+15]
- Modell 2: 25010+S von Calero et al. [Cal+15]
- Modell 3: Generic Sustainable Software StarModel (GS3M) von Amri und Bellamine Ben Saoud [AB14b], [AB14a]
- Modell 4: Green internal/external quality model von Beghouira et al. [BBB17]
- Modell 5: Integrating Environmental Sustainability in Software Product Quality von Koçak et al. [KAB14], [KAB15]
- Modell 6: IT-oriented Software Quality Models von Gordieiev und Kharchenko [GK18]
- Modell 7: Software SustainabilityModel von Condori-Fernandez und Lago [CL18b], [CL18a], [CL19]

3.1.2 Ergebnisse

Einen direkten Bezug zum ISO/IEC 25010 Standard stellen die Modelle 2, 4 und 5 her. Der ISO Standard wurde dabei um eine ökologische Nachhaltigkeitsdimension (*Greenability/Green Efficiency/Environmental*) erweitert [GGP21]. Der Beitrag von Modell 7 liegt in der Bewertung von Kriterien der Produktqualität (PQ) nach deren Relevanz für Green IT.

Insbesondere die Aspekte Performance Efficiency, Functional Suitability, Reliability und Usability haben die Autoren als relevant erachtet [GGP21]. Modelle 1 und 3 stellen einen indirekten Bezug zum ISO-Standard her, wobei beide Modelle Nachhaltigkeit als eigenes Softwarequalitätskonzept ansehen. Mit dem dreidimensionalen *Quality Model for Green and Sustainable Software* von Kern et al. werden bei der Entwicklung Effekte erster, zweiter und dritter Ordnung berücksichtigt [Ker+13]. Das Modell adressiert soziale und ökologische Nachhaltigkeit und betrachtet die Kriterien Efficiency, Reusability, Modifiability und Usability des ISO-Standards als relevant [GGP21]. Zusammenfassend werden die Kriterien Usability, Performance Efficiency, Functional Suitability, Resource utilization, Reliability und Portability im Kontext von nachhaltiger Software in allen Modellen am häufigsten genannt [GGP21].

| PQ-Kriterium | Subkriterium | Häufigkeit |
|------------------------|----------------------|------------|
| Performance efficiency | | 4/7 |
| | Resource utilization | 3/7 |
| Usability | | 5/7 |
| | Accessibility | 3/7 |
| Reliability | | 3/7 |
| Functional suitability | | 3/7 |
| Portability | | 3/7 |
| | Adaptability | 2/7 |
| Maintainability | | 2/7 |
| | Reusability | 2/7 |
| | Modifiability | 2/7 |

Tabelle 2: Grüne und nachhaltige Produktqualitätskriterien [GGP21]

Tabelle 2 zeigt Kriterien bzw. Subkriterien der ISO/IEC 25010 Norm und von wie vielen der sieben Modelle sie als relevant betrachtet werden.

„Das Thema Performance Efficiency hat erwartungsgemäß eine hohe Bedeutung, der inhaltliche Zusammenhang zum Thema ökologische Nachhaltigkeit und damit Green Software erschließt sich dabei unmittelbar. Die hohe Bedeutung von Usability und Functional Suitability ist vor dem Hintergrund der sozial nachhaltigen und langfristigen Nutzung von Software durch unterschiedlichste Benutzergruppen sehr gut verständlich und daher aus unserer Sicht auch wichtig. Die identifizierten Qualitätsattribute Reliability, Portability und Maintainability repräsentieren jedenfalls die ökonomische und technische Nachhaltigkeit auf die Produktqualität eines Softwareproduktes und sollten daher auch immer im Auge behalten werden.“[GGP21]

3.2 Vorgehen in der Softwareentwicklung

Im letzten Abschnitt wurden Qualitätsmodelle mit Bezug auf Nachhaltigkeit betrachtet, die einen Überblick darüber geben, welche Kriterien in der Entwicklung möglichst erfüllt werden sollten. Zu klären ist nun, wie ein allgemeiner Ablauf der Entwicklung eines Softwareproduktes, in Hinblick auf Einhaltung der Qualitätsmerkmale für nachhaltige Software, ablaufen könnte.

3.2.1 Tests und Messungen

Anhand des Kriterienkatalogs für den Blauen Engel für ressourcen- und energieeffiziente Softwareprodukte lässt sich nun die Ressourceneffizienz einer Software messen [Inf22]. Die Einführung eines neuen nachhaltigen Konzepts der Softwareentwicklung setzt neben dem Blick auf Nachhaltigkeit allerdings nach wie vor die Einhaltung von sauberem Quellcode voraus. Für viele Softwareprodukte ist es sinnvoll dafür Unit Tests zu entwickeln. Für die meisten Programmiersprachen werden Frameworks zur Erstellung dieser Tests zur Verfügung gestellt, womit sich die Funktionalität einfach überprüfen lässt. Sinnvoll ist es ebenfalls Performance Tests während der Testphase durchzuführen. So ist den Entwicklern eines Softwareproduktes ersichtlich, wie die Anwendung sich unter erheblicher Belastung verhält [Atl22]. Zudem ist das Thema Performance ein wichtiges Kriterium von nachhaltiger Softwarequalität. Zusätzlich zu den Softwaretests ist es für die Gewährleistung nachhaltiger Software von großer Wichtigkeit, den Ressourcen- und Energieverbrauch zu messen. Für beide Aspekte gibt es jeweils Tools zur Unterstützung. Alle in diesem Abschnitt genannten Techniken zum Testen bzw. Messen von Software lassen sich manuell oder automatisiert durchführen und in jedes Vorgehensmodell, wie z.B. Scrum, miteinbinden.

3.2.2 Automatisierung

Die Automatisierung von Prozessen ist in der Softwareentwicklung heutzutage ein wichtiger Bestandteil des Prozesses. Werden repetitive, d.h. sich wiederholende Aufgaben automatisiert, werden Zeit und Kosten gespart. Zudem werden unnötige Fehler beim Bearbeiten von Aufgaben vermieden. Wird ein Softwaretest beispielsweise komplett manuell durchgeführt, muss der Tester genau aufpassen und darf keinen Fehler machen, da die Ergebnisse sonst verfälscht werden. Auch bei automatisierten Tests können sich Fehler einschleichen, diese sind im Nachhinein aber i.d.R. leichter erkennbar.

Das Test- bzw. Messverfahren zur Bestimmung von qualitativ hochwertiger bzw. nachhaltiger Software kann direkt in den automatisierten Continuous Integration Prozess integriert werden. Dafür müssen die Tests bzw. Messungen allerdings automatisiert laufen, da sie am Beispiel von dem CI-Tool Jenkins über ein Bash-Skript gestartet werden. Abbildung 11 zeigt eine erweiterte Version des Ablauf von Continuous Integration, welcher in Abbildung 8 dargestellt ist.

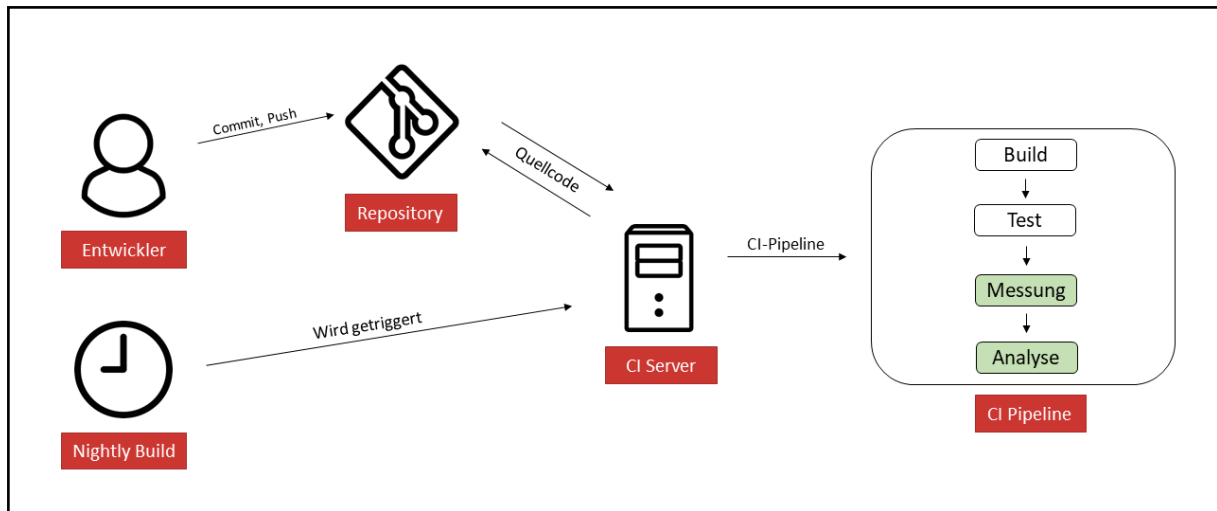


Abbildung 11: Erweiterter Ablauf von Continuous Integration (eigene Darstellung, basierend auf [Add22])

Der Aufbau ist im Gesamten gleich, verändert hat sich nur die Konfiguration der CI-Pipeline. Hier kommen nun die Schritte *Messung* und *Analyse* hinzu. Zusätzlich zu dem üblichen Bauprozess sowie verschiedenen Softwaretests, wird die Ressourceneffizienz der Anwendung beim Durchführen der CI-Pipeline gemessen. Die Ergebniswerte dieser Messung werden im letzten Schritt der Pipeline analysiert, sodass alle Mitarbeiter des Projektes ein aussagekräftiges Ergebnis bekommen, ob die Software nachhaltig ist oder nicht. Abschnitt 4.2 dieser Thesis zeigt an mehreren Praxisbeispielen, wie ein Messverfahren für ressourceneffiziente Software konkret aussehen kann.

3.3 Beispiel aus der Praxis für Qualitätskontrolle

In Abschnitt 4.2 werden zwei Testanwendungen vorgestellt, um das in der Thesis entwickelte Messverfahren zu testen bzw. zu bewerten. Die beiden Anwendungen sind Softwareprodukte der Fresenius Digital Technology. Das Unternehmen ist Teil der Fresenius SE und für dutzende Applikationen im produktiven Betrieb verantwortlich.

3.3.1 Interview

Die letzten Abschnitte haben gezeigt, wie wichtig Qualitätskontrolle im Kontext von nachhaltiger Software ist. Des Weiteren wurde ein Ablauf präsentiert, wie die Überprüfung von Software auf ihre Nachhaltigkeit recht einfach in den Ablauf von automatisierter Qualitätskontrolle integriert werden kann. Um einen Einblick in die Praxis zu bekommen, wie Qualitätskontrolle konkret aussehen kann und ob auch auf nachhaltige Aspekte geachtet wird, wurde ein Interview mit dem Leiter des sam.ware-Teams Hendrik Schneider der Fresenius Digital Technology geführt. Das Interview ist in schriftlicher Form in Anlage 1 genauer nachzulesen. Die Befragung hat ergeben, dass sehr viel Wert auf genaue Dokumentation und Softwarequalität gelegt wird. Dies ist verständlich, wenn man den Aspekt beachtet, dass die Fresenius Medical Care medizinische Produkte herstellt. Durch die Verwendung des klassischen Wasserfallmodells wird während des Softwareentwicklungszyklus erst relativ spät getestet. Die Softwaretests selbst erfolgen dabei ausschließlich in manueller Form. Der Entwicklungsprozess wird im sam.ware-Team durch einen Continuous Integration Server automatisiert.

Als CI-Tool wird hierbei GitLab verwendet, welches in dieser Thesis unter dem Abschnitt 2.3.2 mit den beiden anderen Tools Jenkins und Travis CI verglichen wurde. Als Trigger für die CI-Pipeline in GitLab wird kein Nightly Build verwendet, sondern die Softwareprojekte werden bei jedem Commit bzw. Push ins Repository gebaut. Im Bezug auf nachhaltige Software achten die Mitarbeiter des Teams während des Softwareentwicklungsprozesses nicht aktiv, sondern nur indirekt durch Einhalten des *Dont't repeat yourself (DRY)* Ansatzes. DRY besagt, dass Duplikate im Quellcode eliminiert werden sollten, da dies eine Verschwendung von Ressourcen ist und *Copy Paste Programmierung* zudem die Softwarequalität senkt [Dev22]. Zudem ergab das Interview, dass eine allgemeine Guideline zur Softwareentwicklung existiert, allerdings seien keine Aspekte der Green IT inkludiert. Dennoch wird bei der Realisierung von Anforderungen auf Einsparung von Ressourcen geachtet, seien es Speicherplatz, Systemauslastungen oder Infrastrukturen. Der Teamleiter empfindet den in Abbildung 11 beschriebenen erweiterten Ablauf von Continuous Integration im Bezug auf nachhaltige Software interessant und steht einem Test des Konzepts nicht im Wege.

3.4 Probleme und Ausblick

Insbesondere das Interview mit Hendrik Schneider hat gezeigt, dass Qualitätskontrolle in großen Unternehmen i.d.R. bereits gut abgedeckt ist. Allerdings ist immer noch viel Luft nach oben, wenn man den Aspekt der Nachhaltigkeit betrachtet. Ein gewisses Augenmerk auf nachhaltigen und ressourcensparenden Quellcode liegt bei diesem Praxisbeispiel im Falle des sam.ware-Teams zwar vor, jedoch fehlt die aktive Überprüfung und Bewertung komplett. Das Interview hat ergeben, dass grundsätzlich Interesse da ist, es aber an Experten mangelt bzw. dass ein Konzept zur Bewertung von Software im Bezug auf Nachhaltigkeit bisher komplett gefehlt hat. Um zu überprüfen, wie gut Messungen in automatisierter Form nicht nur theoretisch, sondern auch praktisch funktionieren, soll das in Abschnitt 3.2 vorgestellte Vorgehen in der Softwareentwicklung als Basis für ein neues automatisiertes Messkonzept verwendet werden.

4 Messkonzept mit Continuous Integration

4.1 Einführung

Dieses Kapitel erläutert ausführlich das in dieser Ausarbeitung zu entwickelnde Konzept unter Betrachtung von Continuous Integration zur Messung und Evaluation von Software. Als Basis für das Messverfahren dient das bisherige Konzept der Green Software Engineering Forschungsgruppe des ISS, welches in Abschnitt 4.3.1 genauer erklärt wird. Als Testanwendungen für dieses Messkonzept wurden zwei Webapplikationen der Fresenius Digital Technology aus gesucht, welche beide im nächsten Abschnitt vorgestellt werden. Die beiden Softwareprodukte eignen sich gut, um bestimmte Teile des Quellcodes auf Nachhaltigkeit zu untersuchen, da für die Ausführung der Anwendungen jeweils eine enorme Datenmenge benötigt wird. Für jede Software werden jeweils zwei Zustände definiert, die zu Testzwecken einmal einen niedrigen und einmal einen hohen Ressourcenverbrauch verursachen sollen. Diese beiden Zustände stehen für zwei Commits, welche im Versionsverwaltungssystem getätigt werden. Die Messergebnisse sollen entsprechend verglichen und ausgewertet werden.

4.2 Testanwendungen

4.2.1 Efficient

Der *Efficient* als Webanwendung zur Visualisierung von aktuellen Produktionsdaten ist eine von der Fresenius Digital Technology entwickelte Software. Die Fresenius Medical Care stellt Produkte für Menschen mit chronischem Nierenversagen her, die auf eine Dialysetherapie angewiesen sind [Fre22]. Darunter fallen unter anderem Dialysefilter, welche bei Fresenius einen eigenen Produktionsbetrieb darstellen. Insgesamt werden diese Filter an neun verschiedenen Produktionslinien hergestellt. Der Efficient visualisiert diese Produktionsdaten aus den Filterlinien, welche einmal pro Minute aktualisiert werden. Der Benutzer kann in der Anwendung dabei keine Daten schreiben, sondern ausschließlich lesen. Abbildung 14 zeigt eine Übersicht über alle neun Filterlinien. Interessant sind dabei die Prozentwerte für die Schicht- und Jahreseffizienz, sowie die Zieleffizienz. Die Abbildungen 12 und 13 geben zudem eine Veranschaulichung der Schichteffizienz pro einzelner Linie, sowie der entsprechende Jahresübersicht, die die Effizienz jeden Tages des aktuellen Jahres visualisiert.

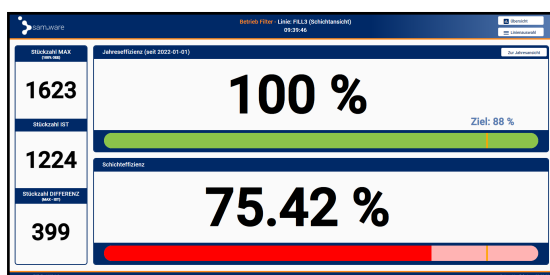


Abbildung 12: Efficient-Schichtansicht

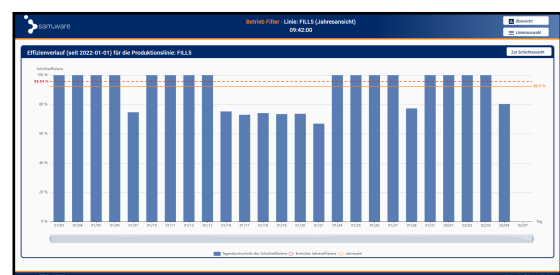


Abbildung 13: Efficient-Jahresansicht



Abbildung 14: Efficient-Übersicht

Auf technischer Seite besteht die Webanwendung aus zwei Teilen und gilt als Client-Server-System. Beide Teilprojekte sind mit der Programmiersprache TypeScript implementiert. Das Frontend basiert dabei auf dem JavaScript-Framework *Angular* [LLC21], das Backend ist mit dem NodeJs-Framework *Express* [Exp22] implementiert. Das Backend stellt dabei die Daten, welche visualisiert werden sollen, über eine RESTful-API bereit, damit das Frontend auf diese Daten per API-Anfrage zugreifen kann. Für jede der drei Ansichten, gibt es eine eigene REST-Route, die entsprechend verschiedene Daten zurückliefert. Tabelle 3 zeigt die Routen in einer Übersicht, samt der Daten, welche dabei geliefert werden.

| REST-API-Route | Liefert |
|--------------------|--|
| /api/linien | Effizienzdaten zu allen Linien |
| /api/linien/:linie | Effizienzdaten zu parametrisierter Linie |
| /api/jahr/:linie | Jahresansicht zur parametrisierter Linie |

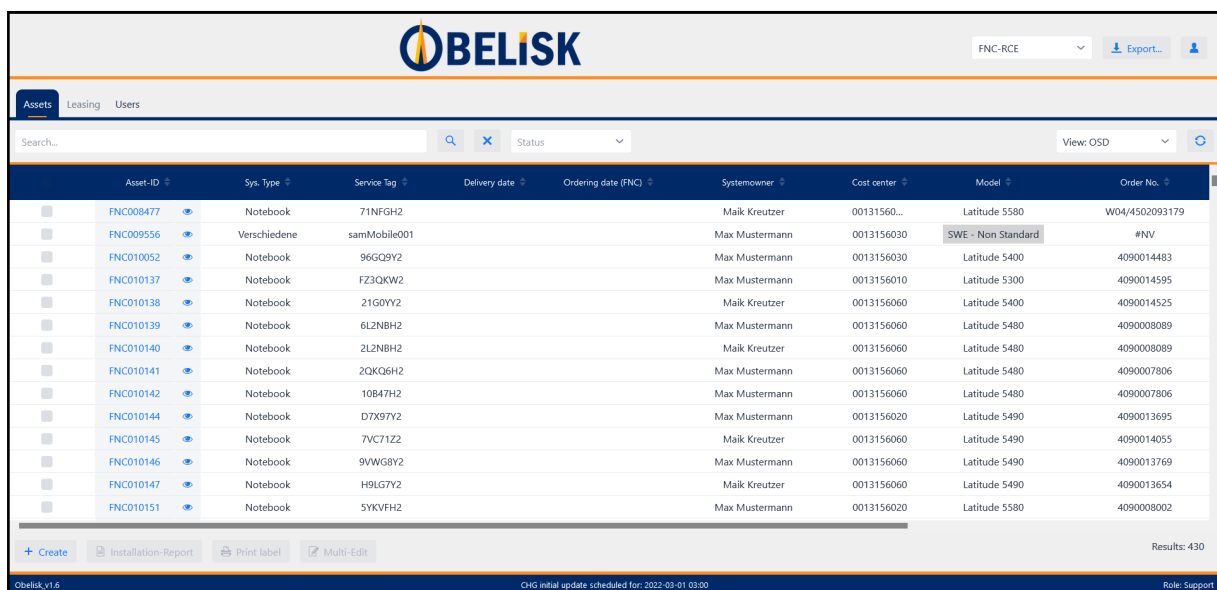
Tabelle 3: Liste der Efficient REST-API-Routen

Da die Produktionsdaten aus neun verschiedenen sowie unabhängigen Linien kommen, gibt es auch für jede einzelne Linie eine eigene Datenbank, welche alle auf MySQL basieren. Zusätzlich dazu gibt es noch eine zehnte PostgreSQL-Datenbank, die weitere erforderliche Daten zu allen Filterlinien enthält. Die in der Jahresansicht visualisierten Daten sollen nur einmal täglich um 06:00 Uhr morgens geladen werden. Die Liniendaten für die Schichtansicht bzw. Übersicht werden alle 60 Sekunden aktualisiert. Das Backend der Anwendung legt die Daten der API dabei serverseitig in einen Zwischenspeicher, sodass pro Minute bzw. für die Jahresansicht pro Tag nur eine Anfrage des Clients an den Server zugelassen wird. So werden unnötige Datenbankabfragen vermieden und Ressourcen gespart. Obwohl der Benutzer keine Daten schreiben kann, wird der Ressourcenverbrauch durch die API-Anfragen dennoch erhöht und ohne Zwischenspeicher auf der Serverseite entsteht eine enorm hohe Anzahl von Datenbankabfragen.

Für diese Thesis ist der Vergleich interessant, bei dem das Backend der Webanwendung einmal mit und einmal ohne serverseitiges Caching läuft. In Abschnitt 4.5.3 wird dieser Gedanke aufgenommen.

4.2.2 Obelisk

Der Assetmanager *Obelisk* ist eine firmeninterne Webanwendung der Fresenius Digital Technology, die zur Verwaltung von Assets dient. Abbildung 15 zeigt dabei die Übersicht über alle Assets. Benutzer können je nach Berechtigung Assets erstellen, ansehen oder bearbeiten sowie auch Etiketten mit Infos zu den jeweiligen Assets drucken. Zudem lassen sich Leasing-Geräte einfach verwalten. Mit Hilfe von entsprechenden Status, ist immer ersichtlich, in welcher Phase sich das Leasing-Gerät gerade befindet. Läuft die Frist des Geräts bald ab, wird beispielsweise eine automatische Email an den technischen Verantwortlichen gesendet. Jegliche Informationen, seien es Benutzer, Assets, oder Geräte, lassen sich per CSV-Datei importieren sowie auch exportieren. Die Webanwendung basiert auf dem freien Java-Framework *Vaadin* [Vaa22], welches als Bibliothek für sowohl Frontend als auch Backend dient. Über die PostgreSQL-Datenbank werden enorme Datensätze geladen und die CSV-Exports können je nach Szenario ebenfalls sehr groß werden. Zusätzlich durch den großen Overhead von Vaadin ist das Softwareprodukt für diese Thesis interessant zu messen.



| Asset-ID | Sys. Type | Service Tag | Delivery date | Ordering date (FNC) | Systemowner | Cost center | Model | Order No. |
|-----------|--------------|--------------|---------------|---------------------|----------------|-------------|--------------------|----------------|
| FNC008477 | Notebook | 71NFGH2 | | | Maik Kreutzer | 00131560... | Latitude 5580 | W04/4502093179 |
| FNC009556 | Verschiedene | samMobile001 | | | Max Mustermann | 0013156030 | SWE - Non Standard | #NV |
| FNC010052 | Notebook | 96GQ9Y2 | | | Max Mustermann | 0013156030 | Latitude 5400 | 4090014483 |
| FNC010137 | Notebook | FZ3QKW2 | | | Max Mustermann | 0013156010 | Latitude 5300 | 4090014595 |
| FNC010138 | Notebook | 21G0YV2 | | | Maik Kreutzer | 0013156060 | Latitude 5400 | 4090014525 |
| FNC010139 | Notebook | 6L2NBH2 | | | Max Mustermann | 0013156060 | Latitude 5480 | 4090008089 |
| FNC010140 | Notebook | 2L2NBH2 | | | Maik Kreutzer | 0013156060 | Latitude 5480 | 4090008089 |
| FNC010141 | Notebook | 2QKQGH2 | | | Max Mustermann | 0013156060 | Latitude 5480 | 4090007806 |
| FNC010142 | Notebook | 10B47H2 | | | Max Mustermann | 0013156060 | Latitude 5480 | 4090007806 |
| FNC010144 | Notebook | D7X9Y2 | | | Max Mustermann | 0013156020 | Latitude 5490 | 4090013695 |
| FNC010145 | Notebook | 7VC71Z2 | | | Maik Kreutzer | 0013156060 | Latitude 5490 | 4090014055 |
| FNC010146 | Notebook | 9VVG8Y2 | | | Max Mustermann | 0013156060 | Latitude 5490 | 4090013769 |
| FNC010147 | Notebook | H9LG7Y2 | | | Maik Kreutzer | 0013156060 | Latitude 5490 | 4090013654 |
| FNC010151 | Notebook | 5YKVFH2 | | | Max Mustermann | 0013156020 | Latitude 5580 | 4090008002 |

Abbildung 15: Obelisk-Assetansicht

4.3 Versuchsaufbau

4.3.1 Bisheriger Aufbau des ISS

Das Institut für Softwaresysteme der Hochschule Trier am Standort Birkenfeld untersucht schon seit einigen Jahren den Energie- und Ressourcenverbrauch für Software. Als Basis zur Evaluation der Qualität dient das in Abschnitt 3.1.1 kurz vorgestellte Qualitätsmodell für grüne und nachhaltige Software ([Ker+13] und [Nau+15]). Die Forschungsgruppe Green Software Engineering des ISS entwickelt seit 2011 stetig an einer Mess- und Analysemethode zur Einschätzung des Energie- und Ressourcenaufwands anhand eines Nutzungsszenarios weiter [Inf22].

Die Norm ISO/IEC 14756 gilt als Anhaltspunkt für die Messung und Bewertung der Leistung sowie der Software-Effizienz von Datenverarbeitung-Systemen [ISO22b]. Basierend auf diesem Standard zeigt Abbildung 16 den bisherigen Messaufbau der Green Software Engineering Arbeitsgruppe.

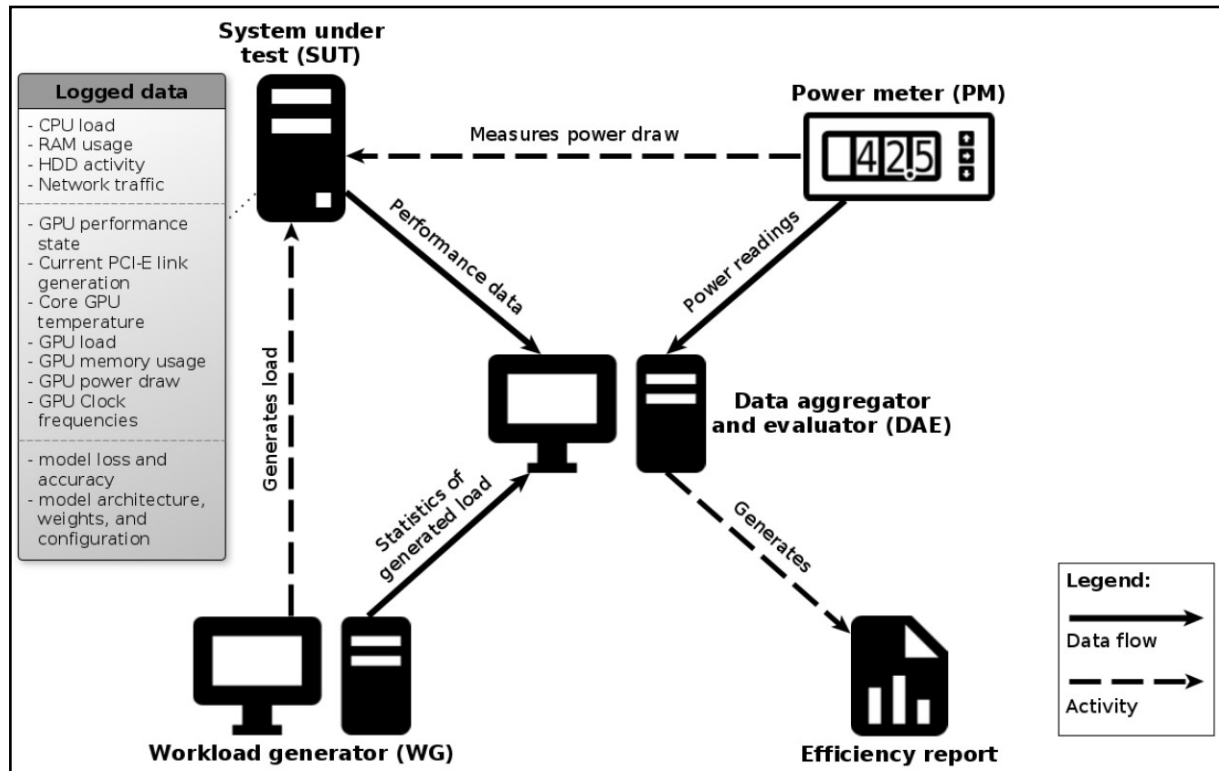


Abbildung 16: Messaufbau zur Ermittlung des Energie- und Ressourcenverbrauchs [GKN21]

Der Messaufbau beinhaltet die Komponenten *System Under Test (SUT)*, *Leistungsmessgerät (PM)*, *Workload Generator (WG)* sowie *Datensammlung und -analyse (DAE)*. Das Nutzungsszenario der zu messenden Software wird auf dem SUT durchgeführt. Je nach Bedarf kann ein einfacher Server, PC, Mobilgerät oder auch ein IoT-Gerät als SUT verwendet werden. Die Hardwarenutzung wird während der Ausführung des Szenarios auf dem SUT mittels der Software *Collectl* aufgezeichnet. Darunter fallen die CPU-, Disk- und RAM-Nutzung sowie der Netzdatenverkehr. Ebenfalls während dem Szenario wird die Leistungsaufnahme des SUT durch das PM gemessen. Gesteuert wird das Nutzungsszenario durch einen WG, welcher auf dem SUT eine Last erzeugt. Dies kann unter anderem durch die Ausführung eines Skripts, oder den wiederholten Aufruf einer API, Website oder Datenbank erfolgen. Ebenso ist es denkbar, dass der WG ein Werkzeug ist, welches auf dem SUT selbst ausgeführt wird. Das DAE sammelt die generierten Daten, welche während des Szenarios gemessen bzw. erstellt wurden. Darunter fallen letztendlich die Hardwarenutzung, elektrische Leistung sowie die vom WG generierte Log-Datei. Zu jeder Aktion des Szenarios wird dabei jeweils ein Start- und End-Zeitstempel aufgenommen. [Inf22]

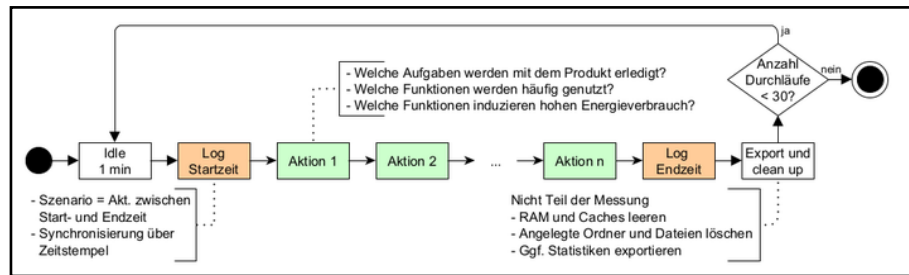


Abbildung 17: Beispielhafter Ablauf des Nutzungsszenarios [Inf22]

Ein beispielhafter Ablauf eines Nutzungsszenarios ist in Abbildung 17 dargestellt. Um mögliche Ausreißer zu erkennen, wird das Szenario insgesamt 30 mal wiederholt. Bei jeder Messung, sei es die Ausführung von verschiedenen Funktionen der Software oder seien es Lasttests, die die Anwendung unter erheblicher Belastung überprüfen sollen, ist es notwendig, ergänzend auch ein Baseline-Szenario zu messen. Dabei läuft die Software selbst nicht, allerdings das Betriebssystem sowie alle Voraussetzungen zur Nutzung der zu messenden Software. Der Einfluss des WG in das SUT wird somit ebenfalls in der Baseline miteinbezogen und kann später bei der Analyse herausgerechnet werden. Das SUT wird nach jeder Software auf einen Zustand vor der Installation der Anwendung zurückgesetzt, um zu gewährleisten, dass sich die Messungen nicht gegenseitig beeinflussen. [Inf22]

Zur Analyse der Messdaten wurde von der Green Software Engineering Arbeitsgruppe die Auswertungssoftware *OSCAR* (*Open Source Consumption Analysis and Reporting*) entwickelt. Dabei können sowohl Datensätze des während des Nutzungsszenarios gemessenen Energieverbrauchs als auch die entsprechende Hardware-Auslastung ausgewertet werden [Sci22b].

4.3.2 Unterscheidung von Baseline, Leerlauf und Nutzungsszenario

Um die Messergebnisse einer Software korrekt auszuwerten, werden zusätzlich zu den Nutzungsszenarien noch andere Messwerte benötigt. Für jede Software muss zum Vergleich analog zum bisherigen Messaufbau des ISS eine Baseline-Messung vorliegen. Dort sind ausschließlich das Betriebssystem im Hintergrund aktiv sowie alle Voraussetzungen zur Nutzung der zu messenden Software. Die Software selbst wird dabei nicht ausgeführt, sondern nur die Programme oder Dienste, welche zur Ausführung notwendig sind. Darunter fallen z.B. Programmiersprachen wie Java oder Jenkins als Automatisierungssoftware, um den Continuous Integration Prozess zu ermöglichen. Zusätzlich zählt ein Browser wie Firefox dazu, welcher zur Ausführung einer Webapplikation erforderlich ist. Eine ausführliche Auflistung wird in Tabelle 4 dargestellt.

| Dienste und Programme der Baseline |
|------------------------------------|
| Betriebssystem |
| Java |
| Node |
| Npm |
| Jenkins |
| R |
| Python |
| Firefox |

Tabelle 4: Software-Stack der Baseline

Die Messung für den Leerlauf ist eine Erweiterung der Baseline, im Gegensatz zu dieser wird hier aber auch die Software an sich gestartet. Zusätzlich werden weitere Dienste gestartet, ohne die ein realistisches Szenario nicht möglich wäre. Darunter zählt unter anderem das Starten einer Datenbank, welche nicht direkt zu der Anwendung gehört, ohne Datenbankwerte ist allerdings auch kein realistisches Szenario möglich.

Für die Messung des Nutzungsszenarios werden zusätzlich zur Anwendung und den Diensten, welche für den Leerlauf gestartet werden, auch Nutzeraktionen durchgeführt. Diese Aktionen, welche in Abschnitt 4.5.2 definiert werden, sollen Last auf das SUT generieren.

Das bisherige Messkonzept des ISS bildet also den Grundstein für die drei Messtypen. Die Ausführung erfolgt dabei ähnlich, aber dennoch verschieden. Aus Gründen der Einfachheit werden hier keine 30 Durchläufe verwendet. Mit einem automatisierten Messkonzept werden die Messungen täglich durchgeführt, sodass insgesamt deutlich mehr Messwerte zustande kommen als beim bisherigen Konzept des ISS. In Abschnitt 4.3.6 wird ein für diese Thesis entwickeltes Auswertungstool vorgestellt, welches die Messergebnisse grafisch darstellt. Die Nutzungsszenarien werden analog zum bisherigen Messverfahren getrennt von Baseline und Leerlauf ausgeführt bzw. gemessen. Dies hat in diesem Fall den Grund, dass die beiden Messergebnisse mit Hilfe des Auswertungstools auf diese Weise direkt miteinander verglichen werden können.

4.3.3 Nutzungsszenarien mit Selenium

Basierend auf dem bisherigen Aufbau werden die Nutzungsszenarien bei dem neuen Konzept mit dem gleichen Ablauf durchgeführt. Als Automatisierungswerkzeug dient hier Selenium, ein Open Source Werkzeug zur Automatisierung von Browser-Anwendungen, das seit vielen Jahren zu den bekanntesten und beliebtesten UI Testautomatisierungslösungen für Webanwendungen zählt [Com09]. Unterstützt werden dabei die Browser Treiber von Mozilla, Microsoft, Google, Opera sowie Apple [Sel22]. Die automatisierten Tests können dabei dank der für viele unterschiedlichen Programmiersprachen bereitgestellten *Selenium Language Bindings* in der favorisierten Software-Entwicklungsumgebung geschrieben werden [Com09]. Die Programmiersprache Python bietet für Selenium eine eigene Bibliothek, welche über das Paketverwaltungsprogramm *pip* installiert werden kann.

Dadurch werden dem Entwickler der Tests einige vorhandene Funktionen bereitgestellt, mit denen sich eine Webanwendung automatisiert öffnen und nach Belieben steuern lässt. Der Browser kann beispielsweise neu geladen werden, es können verschiedene HTML- oder CSS-Elemente angesteuert werden, womit sich letztendlich auch Nutzeraktionen nachbilden lassen. Für das Messverfahren dieser Thesis wurde als Grundlage zur Umsetzung der Nutzungsszenarien die Python-Bibliothek Selenium verwendet. Damit das Nutzungsszenario auch automatisiert in der CI-Pipeline unter Jenkins gestartet werden kann, muss Selenium mit einem *kopflosen* (*headless*) Browser gestartet werden. Ein kopfloser Browser verhält sich analog zu einem gängigen Browser, allerdings gibt es keine Benutzeroberfläche, was für das automatisierte Testen sehr hilfreich ist [Sma22]. Für die Baseline muss ein zusätzliches Selenium-Skript entwickelt werden. Dabei werden keine HTML- oder CSS-Elemente angesprochen, sondern es wird nur ein Browser-Fenster geöffnet. Dies ist ein notwendiger Schritt, um die Messungen der Baselines und Nutzungsszenarien im Teil der Auswertung besser zu vergleichen. Auf diese Weise lassen sich die Werte, welche für das Öffnen des Browser mit Hilfe von Selenium verantwortlich sind, einfach von den anderen Werten abziehen. Die Automatisierungssoftware öffnet dabei bei jedem Durchlauf einen frischen Browser und schließt diesen nach Ablauf des Szenarios bzw. für die Baseline nach 60 Sekunden wieder.

4.3.4 Hardware-Messungen

Für das entwickelte Messverfahren ist die Hardware-Auslastung besonders interessant. Bei Ausführung der Nutzungsszenarien wird darauf geachtet, die Funktionen und Module auszuführen, die besonders für einen Anstieg der Hardware-Ressourcen verantwortlich sind. Um diese Auslastung zu messen, wird unter Linux das Tool Collectl [Col18] verwendet. Das Tool zeichnet Systeminformationen zur CPU, der Platte, dem Speicher und dem Netzwerk sowie weitere Informationen jeweils mit Zeitstempel auf [Col18]. Um die Ergebnisse nach der Messung auszuwerten, gibt es bei dem Collectl-Befehl eine Option, die Daten im CSV-Format zu exportieren.

4.3.5 Continuous Integration Server

Tabelle 5 zeigt die Hardware-Spezifikationen zu den jeweiligen Komponenten für den Continuous Integration Server.

| Komponente | Spezifikation |
|------------|---|
| Maschine | FUJITSU ESPRIMO P958 |
| CPU | Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz |
| RAM | 2 x Micron 8GiB DIMM DDR4 Synchron 2667 MHz |
| Mainboard | FUJITSU D3602-A1 |
| SSD | 500GB Samsung SSD 860 |
| GPU | UHD Graphics 630 |

Tabelle 5: Hardware-Spezifikation des CI-Servers

Tabelle 6 zeigt eine Übersicht über alle auf den CI-Server installierten Softwareprodukte. Diese werden alle zur Ausführung bzw. zur Auswertung der zu messenden Testanwendungen benötigt.

| Software | Version |
|------------------|--------------------|
| Betriebssystem | Ubuntu 20.04.4 LTS |
| Desktop-Umgebung | Gnome 3.38 |
| Kernel | 5.13.0-28-generic |
| Nginx | 1.18.0 |
| Jenkins | 2.319.3 |
| Collectl | 4.3.1 |
| Java | openjdk 1.8.0-312 |
| Node | v12.22.10 |
| Npm | 6.14.16 |
| Python | 3.8.10 |
| R | 3.6.3 |
| MySQL | 8.0.28 |
| PostgreSQL | 12.9 |
| Firefox | 97.0 |

Tabelle 6: Software-Stack des CI-Servers

4.3.6 Visualisierung der Messergebnisse durch Autoplot

Für die Visualisierung der Ergebnisse aus den Hardware-Messungen wurde für diese Thesis ein Tool entwickelt, welches im Folgenden *Autoplot* genannt wird. Die Software ist in der plattformunabhängigen Programmiersprache R entwickelt. Autoplot bekommt die aus Collectl exportierte CSV-Datei übergeben und visualisiert die Messergebnisse.

| | Collectl-Feld / Berechnung | Bedeutung |
|---------|----------------------------|------------------------------------|
| CPU | Totl | CPU-Auslastung (Prozent) |
| Disk | ReadKBTot / WriteKBTot | Gelesene / geschriebene Daten (KB) |
| Memory | Used | Verwendeter Arbeitsspeicher (MB) |
| Network | RxKBTot / TxKBTot | Empfangen / übertragen (KB) |

Tabelle 7: Relevante Collectl-Felder

Für die Auswertung relevante Collectl-Felder werden in Tabelle 7 vorgestellt. Insgesamt werden die vier Kategorien CPU, Disk, Memory und Network analysiert. Wird das Autoplot-Skript ausgeführt, wird aus der aktuell beiliegenden CSV-Datei jeweils eine Bilddatei pro Kategorie erstellt. Die erzeugten Bilder enthalten je nach Kategorie entsprechend nur einen Graphen bzw. mehrere, wenn mehr als ein Collectl-Feld für die Kategorie analysiert wird. Die Messwerte enthalten jeweils einen Zeitstempel, welcher sich auf der X-Achse aller Graphen befindet. Dargestellt werden die Stempel dabei in Minuten-Schritten. Der Mittelwert der Datenverteilung wird durch eine rote horizontale Linie dargestellt. Eine beispielhafte Grafik zur Visualisierung des Arbeitsspeicher-Verbrauchs zeigt Abbildung 18.

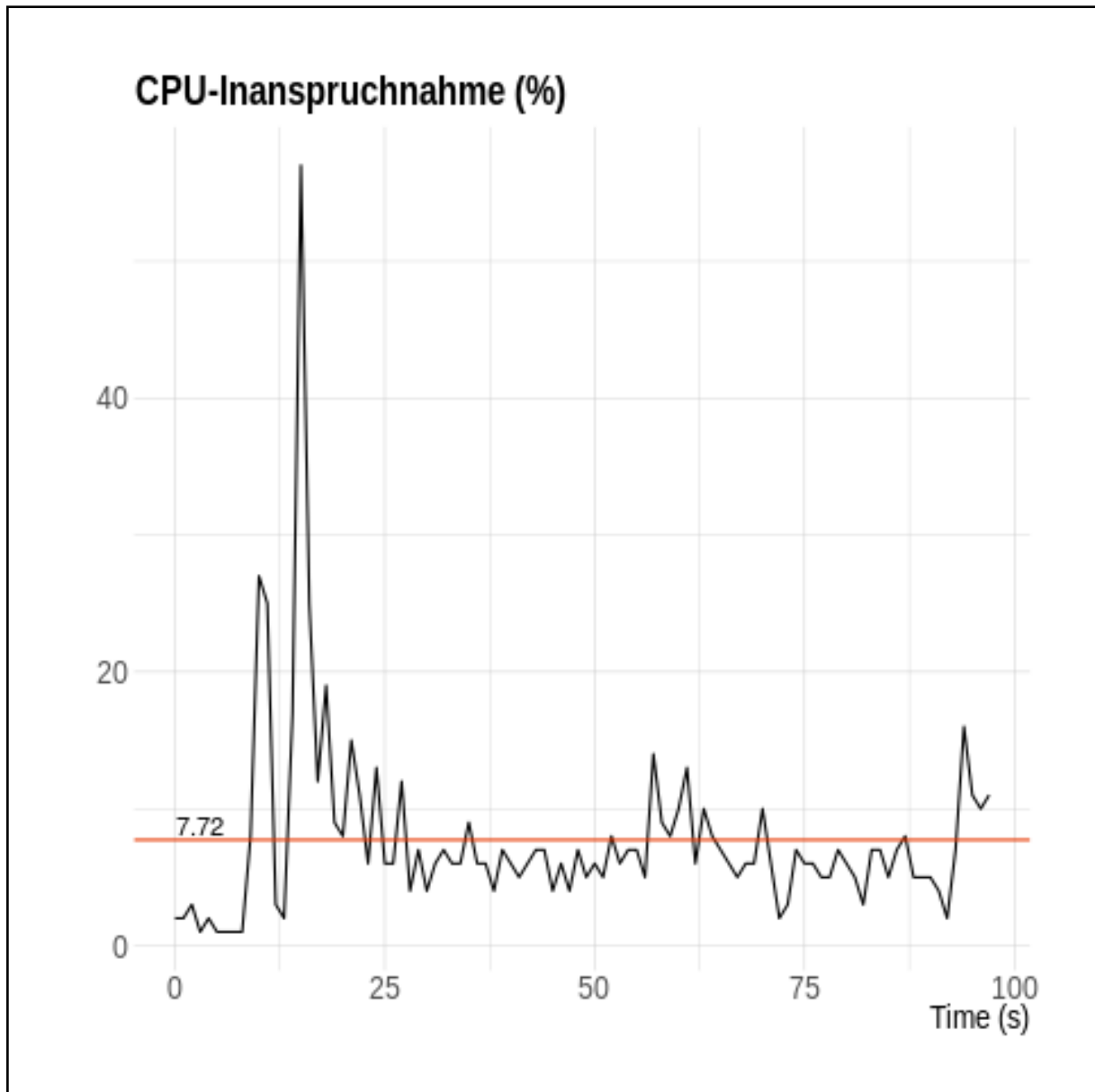


Abbildung 18: Beispielhafte Visualisierung der CPU-Inanspruchnahme

4.4 Erster Vergleich des bisherigen und neuen Versuchaufbaus

Für das im Rahmen dieser Thesis entwickelte automatisierte Messverfahren zur Bestimmung der Ressourceneffizienz von Software wurde der in Abschnitt 4.3.1 vorgestellte Messaufbau bzw. das -verfahren des ISS als Anhaltspunkt verwendet. Vieles wurde dabei übernommen, unter anderem die Messung der Hardware-Auslastung. Unter Linux wird mit Collectl dabei auf das gleiche Tool gesetzt und es werden auch die gleichen Werte gemessen bzw. evaluiert. Ebenso wurde bei dem automatisierten Konzept der *Workload generator* übernommen, welcher dabei Last auf das SUT ausübt. Als *System Under Test* wird hier der Continuous-Integration-Server verwendet. Die Nutzungsszenarien sehen bei beiden Messkonzepten ähnlich aus. Zwar wird für das automatisierte Verfahren mit Selenium eine andere Software zur Ausführung verwendet, im Ablauf verhalten sich die Szenarien aber gleich. Die Messung der elektrischen Leistung fällt bei dem automatisierten Konzept aus Komplexitätsgründen komplett raus. In Abschnitt 5.4 wird dieser Aspekt noch einmal in ausführlicher Form aufgegriffen. Das DAE unterscheidet sich ebenfalls, da in diesem Fall nur die Hardware-Auslastung gemessen wird. Die Ergebnisse werden im CSV-Format an das Auswertungstool Autoplot weitergegeben und dort analysiert. Autoplot ist im Gegensatz zu OSCAR keine Webanwendung und ist insgesamt deutlich weniger komplex. Es können keine Daten hochgeladen werden. Stattdessen werden die gemessenen Daten während der CI-Pipeline in das Verzeichnis von Autoplot verschoben, sodass bei Ausführung des Skriptes direkt darauf zugegriffen werden kann. Erstellt werden dabei schließlich vier Bilddateien zu der Hardware-Auslastung der CPU, Festplatte, dem Speicher sowie dem Netzdatenverkehr.

Dass das bisherige Messkonzept des ISS gut funktioniert, zeigen die zahlreichen Projekte der letzten Jahre, im Rahmen dessen eine Vielzahl an Softwareprodukten gemessen wurde. Der Messaufbau ist zudem recht einfach gehalten, was sehr positiv zu betrachten ist. Die Messungen können von Personen mit wenigen bis keinen Vorkenntnissen recht schnell durchgeführt werden. Zum anderen ist man mit diesem einfachen Konzept im Bereich der Automatisierung sehr eingeschränkt. Die Personen, welche die Messungen durchführen, müssen das Tool, welches das Nutzungsszenario ausführt, manuell starten sowie alle Daten zu elektrischer Leistung bzw. Hardware-Auslastung manuell speichern und zur Auswertung bei OSCAR hochladen. Von Vorteil wäre es natürlich diese für eine Person recht einfachen Aufgaben automatisiert durchzuführen.

Mit dem vorgestellten Messkonzept, basierend auf Continuous Integration, muss der Nutzer nichts weiteres tun, als ein neues Softwareprodukt in der Automatisierungssoftware Jenkins einzupflegen. Die Messungen bzw. Szenarien werden dabei automatisch gestartet und analysiert. Ebenfalls muss niemand vor Ort sein, da sich der CI-Server, welcher bei diesem Konzept das SUT darstellt, vor bzw. nach Ablauf der Messung selbst ein- bzw. ausschaltet. Voraussetzung dafür ist natürlich, dass der CI-Server korrekt konfiguriert ist. Gegebenenfalls müssen zur Ausführung bestimmter Softwareprodukte weitere Programme oder Dienste installiert werden, beispielsweise ein Datenbankmanagementsystem. Die Green Software Engineering Gruppe des ISS plant bis Ende des Jahres 2023 auch Client-Server-Systeme und mobile Anwendungen mit dem Blauen Engel für Softwareprodukte zertifizieren zu können [Sci22a]. Mit der in Abschnitt 4.2.1 vorgestellten Anwendung Effcient wird im Rahmen dieser Thesis bereits ein Client-Server-System auf ihre Ressourcen-Inanspruchnahme evaluiert. Eine mobile Anwendung ist mit diesem Konzept jedoch genauso realisierbar. Der Nachteil von der Automatisierungssoftware Selenium ist allerdings, dass nur Web- und mobile Anwendungen getestet bzw. ausgeführt werden können.

Sollen andere Programme gemessen werden, muss bei dem automatisierten Verfahren auf ein anderes Tool zur Ausführung der Nutzungsszenarien gesetzt werden. Zudem werden zur Entwicklung von Szenarien mit Selenium bzw. Python Programmierkenntnisse vorausgesetzt [Com09]. Die Auswertung des bisherigen Messkonzepts bietet in Bezug auf nachhaltige Software aktuell zudem einen höheren Informationsgehalt, da die Messung der elektrischen Leistung bei dem Konzept unter Continuous Integration komplett fehlt. Ein weiterer Nachteil von Automatisierung ist die steigende Komplexität. Zum einen muss der CI-Server erst konfiguriert werden, zum anderen kann es bei der Einführung von neuen Softwareprodukten in der Jenkins-Plattform immer wieder zu Problemen der Konfiguration kommen. Die CI-Pipeline verschiedener Softwareprodukte kann relativ gleich sein, kann sich aber durch unterschiedliche Softwarearchitektur auch stark unterscheiden.

Insgesamt lässt sich sagen, dass beide Konzepte ihre Vor- und Nachteile haben. Das Konzept automatisierter Messungen beinhaltet einen höheren Erstaufwand der Installation bzw. Konfiguration, erspart den Entwicklern allerdings auch sehr viel Zeit bei einfachen Aufgaben. Das bisherige ISS-Messkonzept ist deutlich weniger komplex gestaltet, dafür wiederholen sich aber immer wieder einfache Aufgaben.

4.5 Versuchsvorbereitung

4.5.1 Automatisierungs-Skripte

Bevor die Continuous Integration Pipeline ausgeführt werden kann, müssen die entsprechenden Testanwendungen erst vorbereitet werden. Hierfür müssen einige Skripte erstellt werden, welche in Tabelle 8 aufgelistet werden.

| Name | Funktion |
|--------------------------------|---|
| runApplication.sh | Starten der Anwendung |
| stopApplication.sh | Stoppen der Anwendung |
| runStandardNutzungsszenario.sh | Ausführung des Standard-Nutzungsszenarios |
| runIdleNutzungsszenario.sh | Ausführung des Leerlauf-Nutzungsszenarios |
| runBaselineNutzungsszenario.sh | Ausführung des Baseline-Nutzungsszenarios |
| standardNutzungsszenario.py | Standard-Nutzungsszenario mit Selenium |
| idleNutzungsszenario.py | Leerlauf-Nutzungsszenario mit Selenium |
| baselineNutzungsszenario.py | Baseline-Nutzungsszenario mit Selenium |
| runMeasurement.sh | Ausführung des Standard-Messverfahrens |
| runIdleMeasurement.sh | Ausführung des Leerlauf-Messverfahrens |
| runBaselineMeasurement.sh | Ausführung des Baseline-Messverfahrens |
| runSqlUpdateScript.sh | Aufruf des SQL Skriptes (Efficient) |
| updateShiftcounter.sql | Ausführung der SQL-Queries (Efficient) |

Tabelle 8: Übersicht über die Automatisierungsskripte

Für jede Software muss jeweils ein Nutzungsszenario mit Selenium unter Python geschrieben werden. Zusätzlich muss ein Bash-Skript erstellt werden, welches das Selenium-Skript über die Kommandozeile ausführt. Analog dazu wird jeweils ein Selenium- und Bash-Skript für die Ausführung der Baseline bzw. des Leerlaufs benötigt. Ebenso sind Bash-Skripte notwendig, um die gesamte Anwendung zu starten bzw. zu stoppen. Diese Skripte werden jeweils in einem weiteren Bash-Skript ausgeführt, welches das Messverfahren im Gesamten letztendlich durchführt. Dort wird zuerst die Anwendung gestartet, anschließend wird der Collectl-Befehl gestartet, um die Hardware-Messungen zu starten, gefolgt von dem Starten des Nutzungsszenarios und anschließendem Stoppen der Anwendung, sobald das Szenario endet. Collectl speichert die gemessenen Daten im CSV-Format. Um den produktiven Betrieb der Efficient Software speziell nachzubilden, wird nur für diese Anwendung ein zusätzliches Skript benötigt, welches für alle Datenbanken sekundliche Updates sendet. Darunter fallen die Datenbanken, welche in der Produktion als Speicherort für fertige Teile pro Schicht verwendet werden. Ein SQL-Skript führt deswegen jede Sekunde für alle neun benötigten Datenbanken gleichzeitig ein Update-Befehl aus, der jeweils eine Spalte der Datenbanktabelle ändert.

4.5.2 Definition der Nutzungsszenarien

Basierend auf dem allgemeinen Ablauf eines Nutzungsszenarios, welcher in Abbildung 17 dargestellt ist, werden auch die Szenarien zu den beiden Testanwendungen definiert.

Efficient

Das Nutzungsszenario für den Efficient sieht wie folgt aus:

- 10 Sekunden Leerlauf
- Aufruf der URL im Browser (Startseite zeigt einen Dialog, bei dem die Produktionslinie ausgewählt werden kann)
- Klick auf den Button für Linie 1 (Navigation zur entsprechenden Schichtansicht)
- Klick auf den Jahresansicht-Button (Navigation zur Jahresübersicht für Linie 1)
- Klick auf den Übersicht-Button (Navigation zur Hauptansicht, die Details zu allen neun Linien anzeigt)
- Durchlaufen einer Schleife: Für jede Linie werden die folgenden Schritte durchgeführt
 - Klick auf den Linienauswahl-Button
 - Klick auf den Jahresansicht-Button
 - Klick auf den Übersicht-Button
- Schließen des Browsers

Zwischen jedem einzelnen Schritt des gesamten Nutzungsszenarios wird jeweils eine Pause von drei Sekunden eingelegt. Dies soll ein realistisches Szenario abbilden, bei welchem sich der Nutzer durch die gesamte Anwendung klickt und dabei jeweils kurz die Werte betrachtet.

Obelisk

Das Nutzungsszenario für den Obelisk sieht wie folgt aus:

- 10 Sekunden Leerlauf
- Aufruf der URL im Browser (Startseite zeigt die Login-Ansicht)
- Eingabe von Benutzernamen sowie Passwort für die beiden Felder
- Klick auf den Login-Button (Navigation zur Asset-Ansicht)
- Klick auf den Export-Button (öffnet einen Dialog)
- Klick auf den Download-Button im Dialog (lädt eine Excel-Datei herunter)
- Seitenaktualisierung (Navigation zur Asset-Ansicht)
- Filterung nach Namen im Suchfeld
- Klick auf das erste Asset der gefilterten Liste (Navigation zur Editier-Ansicht)
- Klick auf den Edit-Button (Aktivierung der Editier-Funktion)
- Überschreibung des Wertes für das *Service-Tag*-Feld mit neuer Eingabe
- Klick auf den Save-Button (Änderungen werden gespeichert und der Dialog wird geschlossen)
- Seitenaktualisierung
- Schließen des Browsers

Zwischen jedem einzelnen Schritt wird auch hier eine Pause von drei Sekunden vorgenommen. Eine Ausnahme ist der Klick auf den Login-Button zu Beginn, nach welchem fünf Sekunden gewartet wird, da der Prozess etwas dauert.

4.5.3 Definition zweier Anwendungszustände

Ziel des automatisierten Messverfahrens ist die frühzeitige Erkennung von Schwankungen der Ressourceneffizienz einer Software. Dadurch, dass die Anwendung jede Nacht automatisch gemessen und visualisiert wird, lassen sich die verschiedenen Git-Commits bzw. Versionen einfach miteinander vergleichen. Um Unterschiede der Ressourceneffizienz von verschiedenen Zuständen der Softwareprodukte testweise zu erkennen, müssen diese Zustände zu Beginn erst definiert werden.

| | Funktion | Zustand 1 | Zustand 2 |
|-----------|-----------------|-------------------|-------------------------|
| Efficient | Backend-Caching | Aktiviert | Deaktiviert |
| Obelisk | CSV-Export | Einfache Schleife | Verschachtelte Schleife |

Tabelle 9: Definition verschiedener Anwendungszustände

Tabelle 9 definiert jeweils zwei Zustände pro Anwendung, deren Messwerte in Kapitel 5 miteinander verglichen werden. Zustand 2 steht dabei für eine Verschlechterung der Ressourceneffizienz, die ein Entwickler womöglich unabsichtlich eingebaut hat. Im Folgenden werden die Zustände jeweils genauer beschrieben und erläutert.

Das Standardnutzungsszenario für den Efficient beschreibt hauptsächlich Aktionen, welche innerhalb der Anwendung zu verschiedenen Ansichten navigieren. In der Regel verbleiben die Anwender die meiste Zeit auf der Schichtansicht einer einzelnen Produktionslinie. Wird ein Überblick über alle Linien benötigt, muss dafür zur Übersicht navigiert werden. Die Jahresansicht zeigt pro Linie zusätzlich noch weitere Informationen an. Deshalb wird im Nutzungsszenario viel navigiert, um ein ansatzweise realistisches Szenario abzubilden, welches zudem eine Last generiert. In Abschnitt 4.2.1 wird ausführlicher erklärt, warum das Backend der Anwendung serverseitig zwischengespeichert wird. Wird dieses Caching deaktiviert, werden jedes Mal, wenn zu einer Ansicht navigiert wird, die entsprechenden REST-API-Routen angefragt, welche wiederum Backend-Funktionen aufrufen, die letztendlich für die Datenbank-Anfragen zuständig sind. Wird also, wie im beschriebenen Nutzungsszenario für den Efficient viel navigiert, ist ein Vergleich von aktiviertem und deaktiviertem Zwischenspeicher sehr interessant. Abschnitt 5.1.4 vergleicht und bewertet die Messergebnisse zweier Nutzungsszenarien, die jeweils einen der beiden definierten Zustände repräsentieren.

Eine Aktion des Nutzungsszenarios für den Obelisk ist für einen CSV-Export verantwortlich. Um den Inhalt der CSV-Datei genauer zu verstehen, bedarf es einer anwendungsspezifischen Erklärung. Grundlegend lassen sich die in Obelisk dargestellten Assets in drei Typen unterscheiden. Die einen sind Teil der Anwendung Obelisk, die anderen Assets sind Teil des *Active Directory (AD)*. Der dritte Typ ist sowohl im Obelisk, als auch im AD gepflegt. Das Active Directory ist dabei ein von Microsoft bereitgestellter Verzeichnisdienst [Lub19]. Über einen Cronjob wird jede Nacht um eine festgelegte Zeit automatisch der Export der CSV-Datei *AD-Checkup* ausgeführt, welcher in dem beschriebenen Nutzungsszenario nachgestellt werden soll. Die Datei beinhaltet die beiden ersten genannten Typen von Assets mit entsprechenden Details, die die Excel-Tabelle füllen. Die Assets, welche sowohl in Obelisk als auch in AD sind, sollen vom Export ausgenommen werden.

```

1      List<Asset> assetsNotInAD = allAssets.stream().filter(asset -> !
      asset.hasAD()).collect(Collectors.toList());
2      List<ADInfo> adInfos = ADInfoService.getAllADInfos(kunde.getName());
3      for (Asset asset : assetsNotInAD) {
4          addCells("No"); // not in AD
5          if ((asset.isInObelisk(asset) == true)) {
6              addCells("Yes"); // in Obelisk
7          } else {
8              addCells("No"); // not in Obelisk
9          }
10     }
11 }
12 for (ADInfo info : adInfos) {
13     if ((asset.isInObelisk(asset) == true)) {
14         addCells("Yes"); // in AD
15         addCells("No"); // not in Obelisk
16     }
17 }

```

Listing 2: Obelisk AD-Checkup Export

Der in Listing 2 abgebildete Codeausschnitt ist ein Teil der CSV-Export-Funktion, die speziell auf den AD-Checkup angepasst ist. In einer Schleife werden alle Assets betrachtet, welche in Obelisk, aber nicht in AD gepflegt sind. Dazu wird die Variable *assetsNotInAD* erstellt, welche eine Liste von nur im Obelisk enthaltenden Assets bekommt. Die Variable *adInfos* bekommt eine Liste von allen Assets, welche im AD sind, die aber gleichzeitig auch in Obelisk gepflegt sein können. Anhand des Codeausschnitts wird ein Programmierfehler aufgezeigt, der passieren könnte. Die Funktion *isInObelisk(asset)* gibt ein *Boolean* zurück, d.h. entweder den Wert *Wahr*, oder *Falsch*. Für die zweite Schleife muss es eine Abfrage geben, die Assets aus Obelisk vollkommen ausschließt. Für die erste Schleife ist die Funktion allerdings redundant. Innerhalb der Funktion werden alle in der Datenbank bzw. in Obelisk gepflegten Assets durchlaufen. Da die Funktion jedoch selbst innerhalb von zwei Schleifen aufgerufen wird, führt dies zu einer enormen Last, was dazu führt, dass der CSV-Export bis zu 40 Sekunden, statt nur einer Sekunde benötigt. Bei Ausnahmefällen kann die Anwendung dabei sogar komplett abstürzen. Deshalb ist es für die zweite Schleife wichtig, welche nur die Assets aus AD betrachten soll, die Prüfung außerhalb der Schleife zu tätigen, sodass alle Assets nur einmal durchlaufen werden, statt bis zu 3000 Mal. Als zwei Zustände für den Vergleich der jeweiligen Ressourceneffizienz wird also einmal eine einfache und einmal eine verschachtelte Schleife verwendet. Analog zum Effizient werden auch diese beiden Zustände in Abschnitt 5.1.4 miteinander verglichen.

4.6 Versuchsdurchführung

4.6.1 Ablauf des Messverfahrens

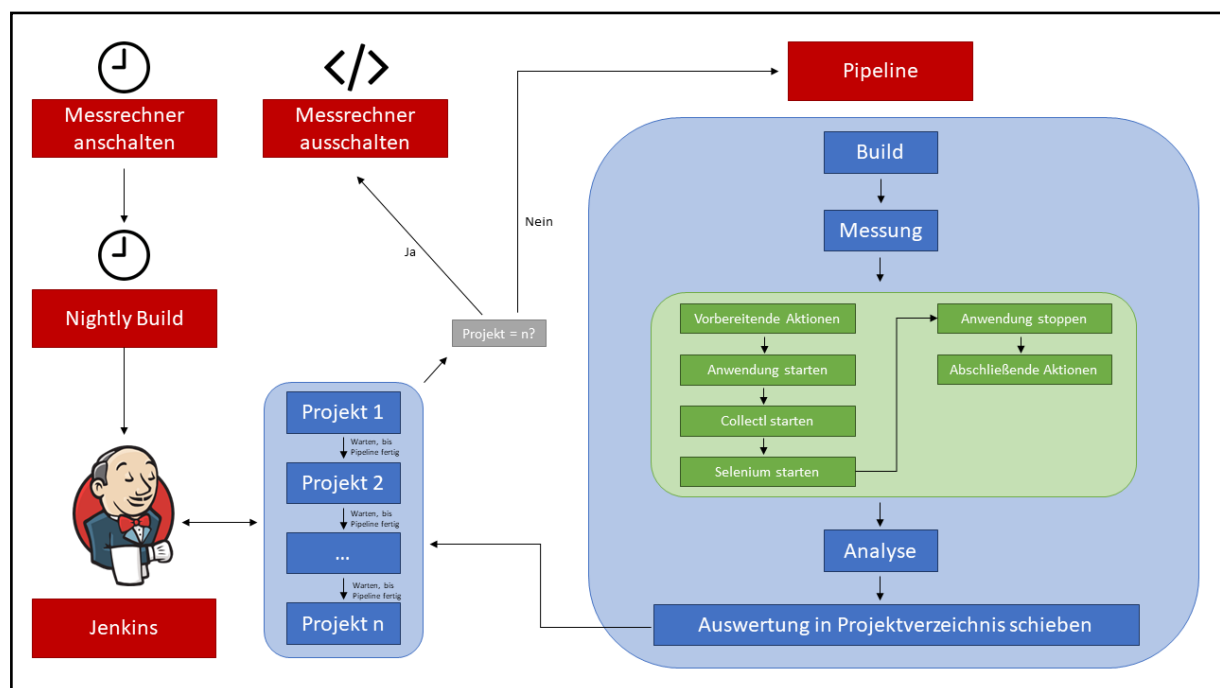


Abbildung 20: Ablauf des automatisierten Messverfahrens (eigene Darstellung)

Wie in Abbildung 20 gezeigt, startet der Ablauf des Messverfahrens mit dem Anschalten des Messrechners, welcher auch als CI-Server verwendet wird. Die Einschaltung erfolgt, wie unter 4.3.7 bereits beschrieben, über Wake on LAN. Dies erfolgt durch ein Skript automatisch um 02:55 Uhr Nachts.

Fünf Minuten danach, um 03:00 Uhr, startet ebenfalls zeitgesteuert der Nightly Build in der CI-Automatisierungssoftware Jenkins. Die Plattform ist mit den vier Projekten *Autoplot*, *Efficient-Frontend*, sowie dem *Efficient-Backend* und zuletzt der *Obelisk* Anwendung verknüpft. Zusätzlich dazu kommen noch die beiden Jenkins-Projekte für die Efficient- sowie die Obelisk-Baseline bzw. zwei Projekte für den Leerlauf. Davon sollen nur die beiden Anwendungen Efficient-Frontend und Obelisk gemessen werden bzw. die Baselines und Leerläufe für beide Projekte. Das Efficient-Backend wird implizit gemessen, da das Frontend auf dessen bereitgestellte Daten zugreift. Autoplot wird überhaupt nicht gemessen. Allerdings wird der aktuelle Code benötigt, da für den Schritt *Analyse* auf das Autoplot-Skript zugegriffen wird. Die beiden Anwendungen müssen also, obwohl sie nicht getestet bzw. gemessen werden sollen, trotzdem in Jenkins gebaut werden, damit der Quellcode auf das verknüpfte GitLab-Repository zugreift und zu jeder Zeit aktuell ist. Autoplot wird als erstes Projekt gestartet und wird um 03:00 Uhr von Jenkins getriggert. Anschließend wird die CI-Pipeline durchgeführt und das Projekt wird lediglich gebaut. In der Projektkonfiguration ist die Efficient-Baseline als Nachfolge-Anwendung vermerkt. Sobald der Autoplot-Build also fertig ist, startet automatisch der Build-Prozess für das Efficient-Backend. Ist dieser Prozess auch abgeschlossen, startet die Efficient-Baseline, der Efficient-Leerlauf, das Efficient-Frontend, die Obelisk-Baseline sowie der Obelisk-Leerlauf und zuletzt startet der Bauprozess des Obelisk. Alle zu messenden Projekte werden jeweils einmal durchlaufen und dauern jeweils 60 bis 100 Sekunden, wobei die Zeit von äußeren Faktoren wie der Einwirkung von Last abhängig ist. So sollten alle Jenkins Builds bis spätestens um 03:30 Uhr abgeschlossen sein, sodass auch die Entwickler, welche bereits früh am Arbeitsplatz sind, die Auswertung der Softwareprodukte vom gestrigen Stand einsehen können. Um diese Zeit wird der Messrechner mittels eines Cronjobs wieder heruntergefahren.

Der Ablauf der CI-Pipeline an sich wird im rechten Teil von Abbildung 20 dargestellt. Zuerst wird die Software gebaut, dieser Prozess sieht von Projekt zu Projekt verschieden aus. Gemeinsam haben die Applikationen, dass zu Beginn erst die Abhängigkeiten installiert werden müssen. Für die hier verwendeten Programmiersprachen bzw. Frameworks TypeScript (unter dem JavaScript-Framework Angular) sowie Java, gibt es jeweils vorgefertigte Befehle, welche das Projekt bauen. Der Quellcode wird dabei komprimiert, d.h. er wird minimiert und geschützt, sodass der Code von außen nicht einsehbar ist.

Anschließend beginnt der Messprozess, dessen Funktionalität in einem Bash-Skript der entsprechenden Anwendung gespeichert ist. Zuerst werden hierbei vorbereitende Aktionen vorgenommen. Spezifisch für den Efficient bzw. den Obelisk fällt das Starten der Datenbank-Services (MySQL bzw. PostgreSQL) darunter. Anschließend wird die Anwendung gestartet und kurz gewartet, um sicherzustellen, dass die URL der Anwendung auch bereit ist, um vom Browser aufgerufen zu werden. Im nächsten Schritt werden zusammen das Collectl- sowie das Selenium-Skript gestartet, um das Nutzungsszenario auszuführen und gleichzeitig zu messen. Ist das Nutzungsszenario abgeschlossen, wird automatisch auch die Collectl-Messung gestoppt und im nächsten Schritt werden die Kommandos zum Stoppen der Anwendung bzw. abschließende Aktionen wie das Stoppen der DB-Services ausgeführt.

Sind alle Dienste gestoppt, wird das Archiv, in welchem der aktuelle Collectl-Output enthalten ist, entpackt und in den Arbeitsbereich des Analyse-Tools Autoplot verschoben. Zusätzlich im Autoplot-Verzeichnis wird eine Text-Datei, die den Applikationsnamen enthält, mit dem Namen der aktuellen Anwendung der Pipeline überschrieben.

Dies ist notwendig, um nach der Analyse einen Ordner für jedes einzelne Projekt zu erhalten, welcher die jeweilige Auswertung enthält. Sobald die von Collectl erstellte CSV-Datei in das Autoplot-Verzeichnis verschoben bzw. die Text-Datei überschrieben wurde, wird das Analyse-Skript gestartet, welches für die vier Bereiche CPU, Disk, Memory und Network jeweils eine PNG-Datei erstellt. Diese Bilddateien werden in einem untergeordneten Ordner für den aktuellen Tag in das Verzeichnis für das entsprechende Projekt abgespeichert. Zur Sicherheit werden die in der entsprechenden CSV-Datei enthaltenen Rohdaten von Collectl ebenfalls in diesem Verzeichnis abgespeichert. Der gesamte Ordner wird im letzten Schritt in den aktuellen Arbeitsbereich von dem Jenkins-Projekt verschoben, sodass die Auswertung jeweils bei den einzelnen Softwareprodukten liegt.

5 Evaluation und Fazit

5.1 Ergebnis der Messungen

Das letzte Kapitel hat ein automatisiertes Messkonzept vorgestellt, welches eine Software auf ihre Ressourceneffizienz prüft. Als Testanwendungen wurden die beiden Webapplikationen Efficient und Obelisk präsentiert. Für jede Anwendung wurden jeweils zwei Zustände definiert, anhand derer eine Veränderung der Ressourceneffizienz der entsprechenden Software simuliert wurde. Zudem wurden zu den Nutzungsszenarien auch Messungen zu der Baseline und dem Leerlauf getätigt. Die dabei entstandenen Messergebnisse wurden in der Continuous Integration Pipeline durch das Tool Autoplot visualisiert und werden in diesem Abschnitt genauer dargestellt und beleuchtet.

5.1.1 Berechnung von Leerlauf und Nutzungsszenario

In den folgenden Abschnitten werden die Messergebnisse für die Baseline, den Leerlauf und die Nutzungsszenarien vorgestellt sowie verglichen. Um für sowohl Leerlauf als auch Nutzungsszenario einen besseren Überblick zu erhalten, wie sich die Softwareprodukte ohne grundlegende Faktoren der Baseline verhalten, wird in diesem Abschnitt auf die Berechnung eingegangen. Die beiden aufgelisteten Formeln zeigen, wie sich der Mittelwert M eines jeweiligen Messwertes berechnet.

$$M_{\text{Leerlauf-Baseline}} = M_{\text{Leerlauf}} - M_{\text{Baseline}} \quad (1)$$

$$M_{\text{Nutzungsszenario-Baseline}} = M_{\text{Nutzungsszenario}} - M_{\text{Baseline}} \quad (2)$$

Die erste Formel zeigt die Berechnung, um den Mittelwert $M_{\text{Leerlauf-Baseline}}$ für einen Messwert der Leerlauf-Messung ohne Einfluss der Baseline sowie ohne Nutzeraktionen zu erhalten. Dabei muss der entsprechende Mittelwert der Baseline, beispielsweise der CPU-Inanspruchnahme, von dem Wert für die Leerlauf-Messung abgezogen werden. Durch diese Berechnung ist ersichtlich, wie hoch die isolierte Ressourcen-Inanspruchnahme der Software ohne Nutzeraktionen ist. Zur Berechnung des Mittelwerts $M_{\text{Nutzungsszenario-Baseline}}$ für einen Messwert des Nutzungsszenarios ohne den Einfluss der Baseline wird Formel 2 verwendet. Hierbei werden für die Mittelwerte des Nutzungsszenarios analog die Werte der Baseline-Messung abgezogen. Dadurch lässt sich herauslesen, welche Ressourcen-Inanspruchnahme konkret durch das Starten der Software samt Nutzeraktionen erfolgt.

5.1.2 Baseline-Ergebnisse

Die Baseline für den Efficient und Obelisk ist ähnlich aufgebaut. Im Hintergrund ist das Betriebssystem aktiv sowie andere Dienste und Programme zum Ausführen der Softwareprodukte. Tabelle 4 aus Abschnitt 4.3.2 bietet diesbezüglich einen genaueren Überblick. Aufgrund des nahezu gleichen Aufbaus ist es demnach nicht verwunderlich, dass die beiden Baseline-Messungen der Applikationen nahezu gleich sind. Zu minimalen Schwankungen kann es aufgrund äußerer Einflüsse trotzdem kommen.

Tabelle 10 stellt die Mittelwerte aller zu untersuchenden Werte für beide Anwendungen vor.

| | Bezeichnung | Efficient | Obelisk |
|------------|----------------------------------|-----------|---------|
| Prozessor | CPU-Inanspruchnahme (Prozent) | 1,54 | 1,57 |
| Festplatte | Gelesene Daten (KB) | 0 | 0,06 |
| Festplatte | Geschriebene Daten (KB) | 491,89 | 633,61 |
| Speicher | Verwendeter Arbeitsspeicher (MB) | 3953,57 | 4211,29 |
| Netzwerk | Netzwerk Dateneingang (KB) | 30,12 | 30,15 |
| Netzwerk | Netzwerk Datenausgang (KB) | 1,38 | 1,4 |

Tabelle 10: Baseline Ergebnisse der Anwendungen Efficient und Obelisk

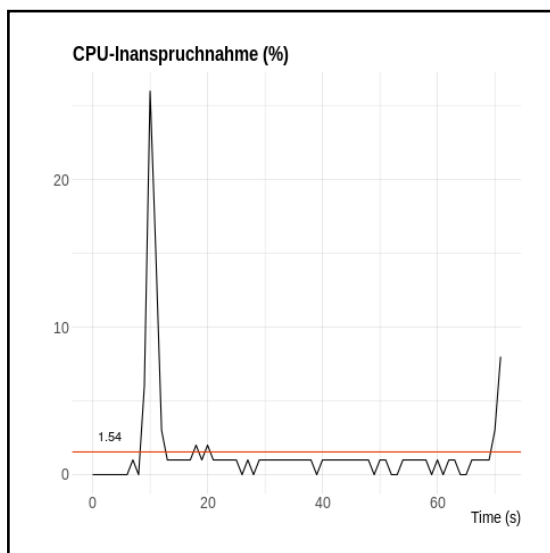


Abbildung 21: CPU-Inanspruchnahme (Efficient - Baseline)

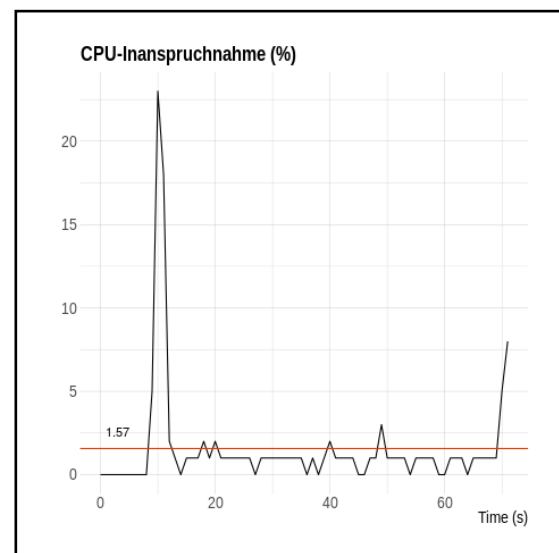


Abbildung 22: CPU-Inanspruchnahme (Obelisk - Baseline)

Wie in den beiden Abbildungen 21 und 22 zu erkennen, wird die CPU zu Beginn hoch beansprucht. Dies liegt jeweils am Öffnen des Browsers. Unter Verwendung des Automatisierungstools Selenium wird bei Ausführung des Skriptes zu Beginn ein frischer Browser geöffnet. Am Beispiel der Baseline-Messungen wird der Browser nach 60 Sekunden wieder geschlossen. Wie zu sehen ist, gehen die Werte an diesem Zeitpunkt wieder leicht hoch. Ansonsten wird die CPU mit durchschnittlichen 1,54 bzw. 1,57 Prozent während der Baseline nur wenig beansprucht.

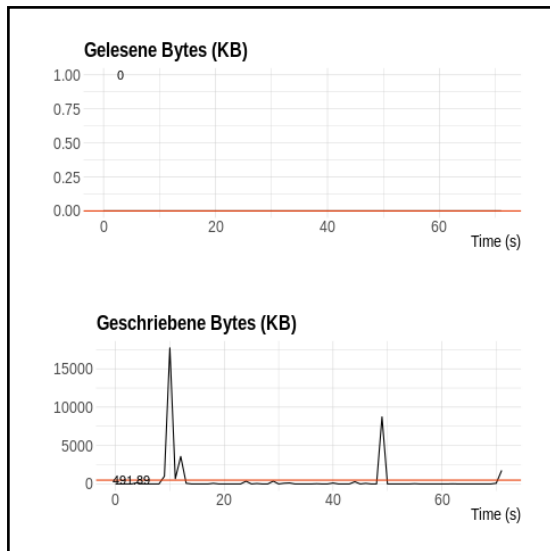


Abbildung 23: Gelesene/geschriebene Bytes der Festplatte (Efficient - Baseline)

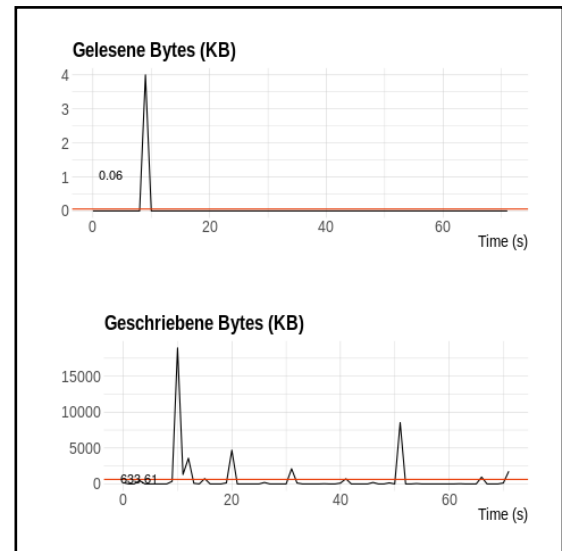


Abbildung 24: Gelesene/geschriebene Bytes der Festplatte (Obelisk - Baseline)

Der Graph zur Visualisierung der gelesenen Daten der Festplatte in KB bleibt bei beiden Applikationen weitestgehend bei 0. Die geschriebenen Daten gehen teilweise relativ hoch, ca. bis 17500 KB. Insgesamt verlaufen sich die Werte aber eher gegen 490 bzw. 630 KB.

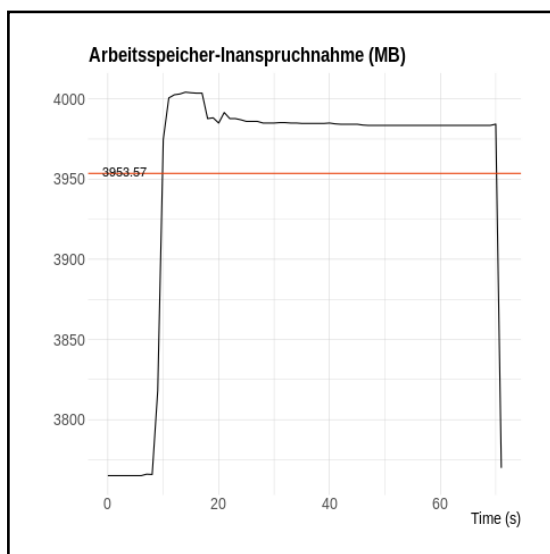


Abbildung 25: RAM-Inanspruchnahme (Efficient - Baseline)

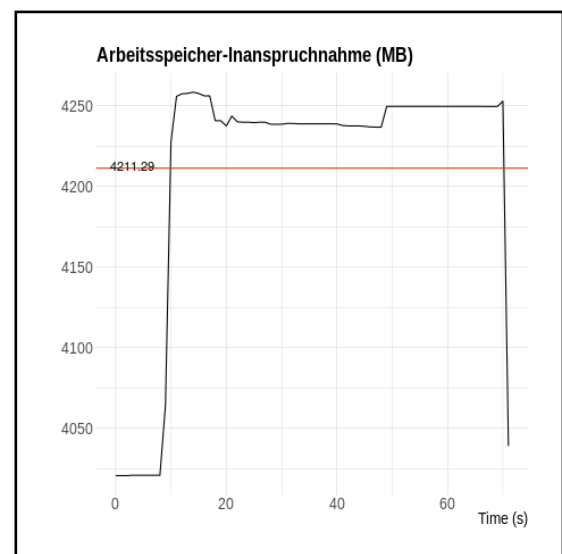


Abbildung 26: RAM-Inanspruchnahme (Obelisk - Baseline)

Der genutzte Arbeitsspeicher beträgt beim Starten des Efficient ca. 3750 KB, der Obelisk startet mit einer Inanspruchnahme von ungefähr 4025 KB. Sobald der Browser nach zehn Sekunden geöffnet wird, ist bei beiden Messungen eine hohe Steigung von grob 200 KB zu sehen. Dieser Wert bleibt im Gesamten relativ gleich auf einer Linie, mit Ausnahmen von wenigen Schwankungen. Erst gegen Ende, beim Schließen des Browsers, wird wieder etwas an Arbeitsspeicher gewonnen.

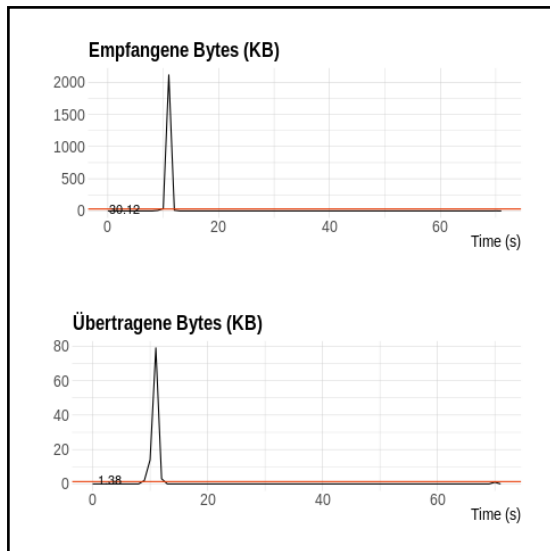


Abbildung 27: Empfänger/übertragener
Netzwerkdatenverkehr
(Efficient - Baseline)

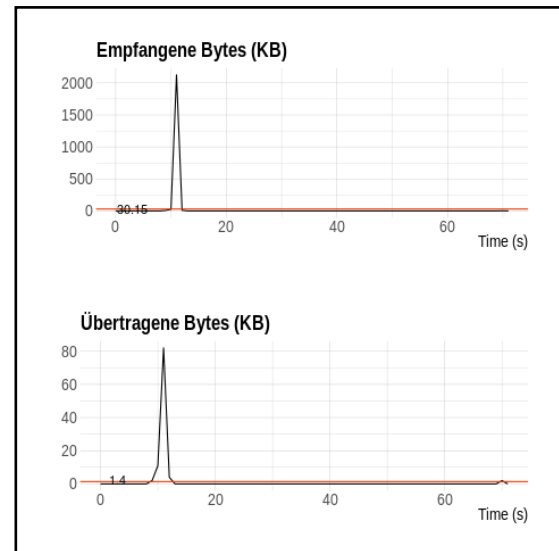


Abbildung 28: Empfänger/übertragener
Netzwerkdatenverkehr
(Obelisk - Baseline)

Die beiden Abbildungen 27 und 28 zeigen den Netzdatenverkehr der beiden Softwareprodukte. Empfangen werden beim Starten des Firefox-Browsers Daten bis knapp unter 2500 Kilobyte, übertragen hingegen nur bis zu 80 KB. Mit Ausnahme der anfänglichen Steigung der empfangenen Daten sind die Werte ansonsten ca. bei nur 30 KB. Analog dazu verhalten sich auch die Graphen für die übertragenen Daten bei durchschnittlichen 1,38 bzw. 1,4 KB.

5.1.3 Leerlauf-Ergebnisse

Die Messungen zum Leerlauf sind eine Erweiterung zu der jeweiligen Baseline. Zusätzlich zu den Grundvoraussetzungen für die Ausführung der Software wird hier auch die Anwendung selbst gestartet, sowie die benötigten Datenbank-Services, ohne die ein realistisches Szenario nicht möglich wäre. Dabei werden im Gegensatz zum Standardnutzungsszenario keine Nutzerinteraktionen simuliert. Für ausschließlich den Efficient zählt jedoch das in 4.5.1 vorgestellte SQL-Skript dazu, welches verwendet wird, um die Felder aus bestimmten Datenbanktabellen sekundlich zu verändern. Damit wird keine Aktion gemessen, sondern zählt das Skript eher als Grundlast, die mitwirkt, sobald die Software gestartet wird. In diesem Abschnitt werden sowohl die Leerlauf-Ergebnisse für den Efficient als auch für den Obelisk vorgestellt und jeweils mit den zugehörigen Baseline-Messungen verglichen.

Efficient

| | Bezeichnung | Baseline | Leerlauf | Differenz |
|------------|----------------------------------|----------|----------|-----------|
| Prozessor | CPU-Inanspruchnahme (Prozent) | 1,66 | 3,33 | 1,67 |
| Festplatte | Gelesene Daten (KB) | 0,02 | 56,22 | 56,2 |
| Festplatte | Geschriebene Daten (KB) | 411,39 | 1225,53 | 814,14 |
| Speicher | Verwendeter Arbeitsspeicher (MB) | 3998,15 | 4272,28 | 274,13 |
| Netzwerk | Netzwerk Dateneingang (KB) | 26,32 | 57,67 | 31,35 |
| Netzwerk | Netzwerk Datenausgang (KB) | 1,61 | 28,89 | 27,28 |

Tabelle 11: Mittelwerte der Baseline, Leerlauf und Differenz (Efficient)

Tabelle 11 gibt eine Übersicht der Mittelwerte für die Baseline- und Leerlauf-Messung sowie deren Differenz $M_{\text{Leerlauf}-\text{Baseline}}$, die nach der in Abschnitt 5.1.1 definierten ersten Formel berechnet wurde.

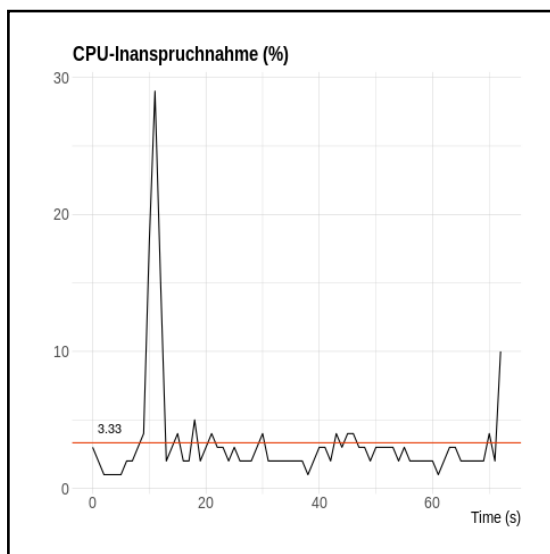


Abbildung 29: CPU-Inanspruchnahme (Efficient - Leerlauf)

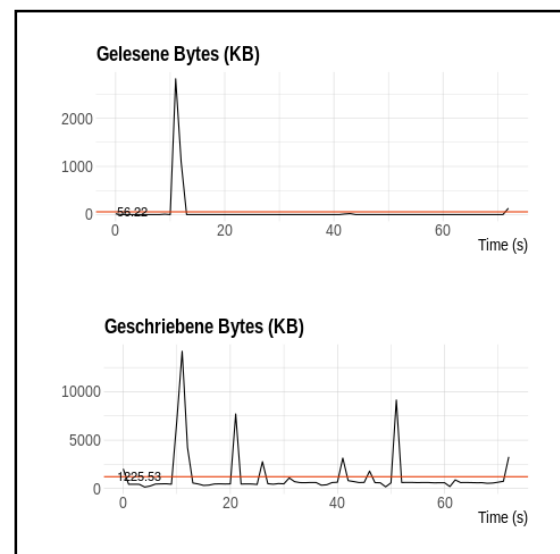


Abbildung 30: Gelesene/geschriebene Bytes der Festplatte (Efficient - Leerlauf)

An der in Abbildung 29 dargestellten CPU-Inanspruchnahme lässt sich bereits ein leichter Anstieg von 1,67 Prozent im Gegensatz zur Baseline erkennen. Dies könnte unter anderem an dem beim Leerlauf ausgeführten SQL-Skript liegen, welches nur für den Efficient entwickelt wurde. Ebenfalls lässt sich dies an den insgesamt höheren Schwankungen erkennen, die sich durch die komplette Messung ziehen.

Die gelesenen Kilobytes steigen nach etwa zehn Sekunden auf ca. 3000 KB an, ansonsten verlaufen die Werte eher niedrig. Mit durchschnittlichen 56,22 KB liegt der Mittelwert deutlich über dem der Baseline-Messung.

Auch der Graph zur Visualisierung der geschriebenen Bytes auf der Festplatte trennt sich optisch nicht nur auf den ersten Blick von dem der Baseline, wie in Abbildung 30 zu sehen ist. Statt den durchschnittlichen rund 411 KB sind es hier ca. 1225 KB, wobei der Graph deutlich mehr an Bewegung zeigt.

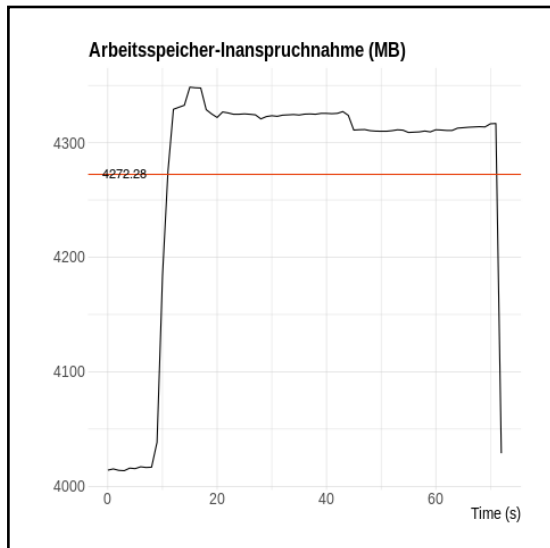


Abbildung 31: RAM-Inanspruchnahme
(Efficient - Leerlauf)

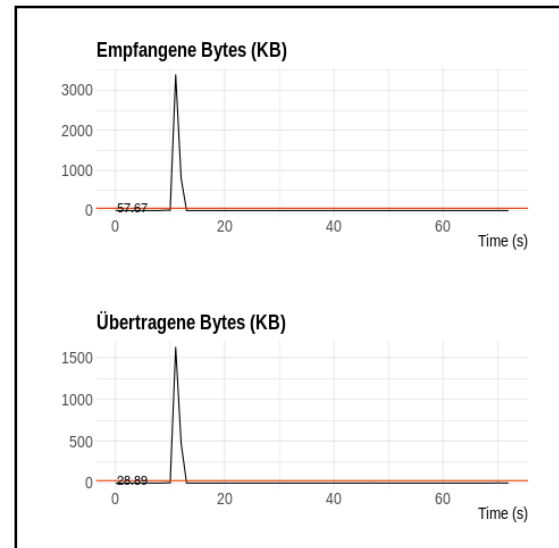


Abbildung 32: Empfangener/übertragener
Netzwerkdatenverkehr
(Efficient - Leerlauf)

Abbildung 31 zeigt, dass der Verlauf für den verwendeten Arbeitsspeicher gleich zu dem der Baseline bleibt. Im Gegensatz zu den Baseline-Messwerten sind dabei nur die Werte an sich höher, insgesamt um rund 274 KB.

Der Netzdatenverkehr verläuft ebenfalls gleich zur Baseline, wie in Abbildung 32 zu sehen ist. Dabei fällt der Mittelwert durch einen größeren Ausschlag zu Beginn bei Öffnen des Firefox-Browsers auch insgesamt deutlich höher aus.

Obelisk

| | Bezeichnung | Baseline | Leerlauf | Differenz |
|------------|----------------------------------|----------|----------|-----------|
| Prozessor | CPU-Inanspruchnahme (Prozent) | 2,03 | 3,29 | 1,26 |
| Festplatte | Gelesene Daten (KB) | 6,54 | 6,68 | 0,14 |
| Festplatte | Geschriebene Daten (KB) | 479,83 | 663,67 | 183,84 |
| Speicher | Verwendeter Arbeitsspeicher (MB) | 4206,93 | 5974,44 | 1767,51 |
| Netzwerk | Netzwerk Dateneingang (KB) | 27,02 | 28,81 | 1,79 |
| Netzwerk | Netzwerk Datenausgang (KB) | 12,42 | 1,45 | -10,97 |

Tabelle 12: Mittelwerte der Baseline, Leerlauf und Differenz (Obelisk)

Analog zum Efficient bietet Tabelle 12 hier eine Übersicht für den Obelisk über die Mittelwerte für die Baseline- und Leerlauf-Messung sowie deren Differenz $M_{\text{Leerlauf}-\text{Baseline}}$, die nach der in Abschnitt 5.1.1 definierten ersten Formel berechnet wurde.

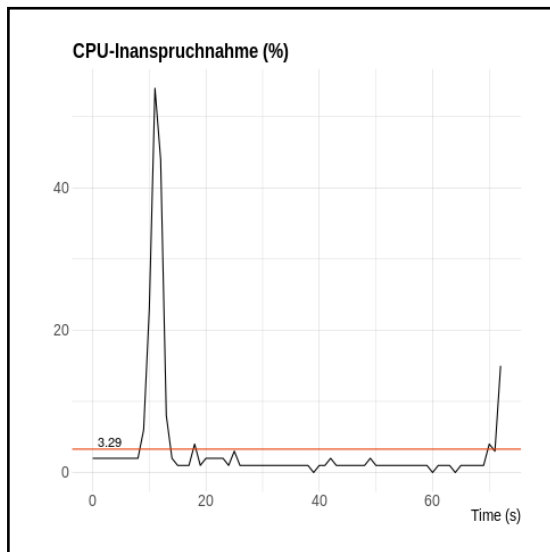


Abbildung 33: CPU-Inanspruchnahme
(Obelisk - Leerlauf)

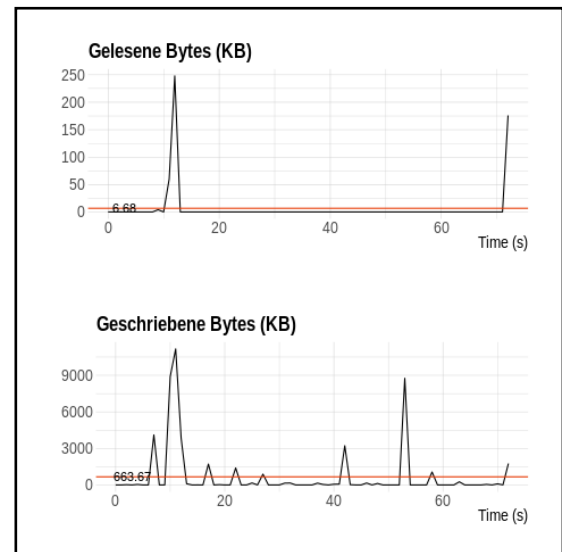


Abbildung 34: Gelesene/geschriebene
Bytes der Festplatte
(Obelisk - Leerlauf)

Wie in Abbildung 33 zu sehen ist, unterscheidet sich der Graph im Ablauf nicht von der Baseline-Messung. Durch den Ausschlag zu Beginn bei Öffnen des Browsers, der insgesamt deutlich höher ist als bei der Baseline, geht auch der gesamte Mittelwert höher und zwar von durchschnittlichen 2,03 auf 3,29 Prozent.

Der Graph für die gelesenen Kilobytes der Festplatte in Abbildung 34 verhält sich gleich zur Baseline und insgesamt kommt es nur zu einem Unterschied von 0,14 KB. Die geschriebenen Daten schwanken im Graph deutlich mehr und höher, wodurch es letztendlich zu einer Steigung von rund 183 KB kommt.

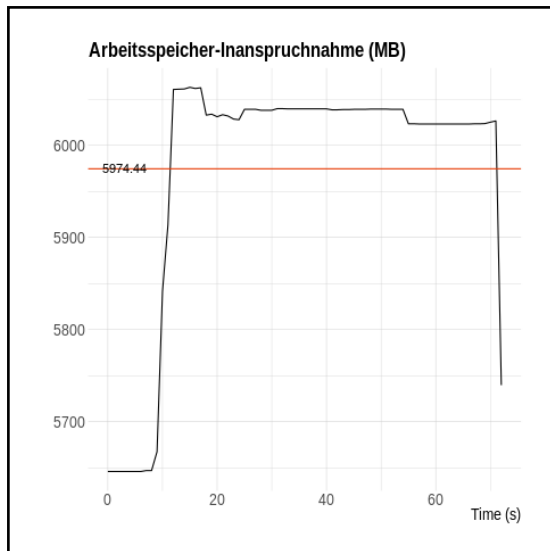


Abbildung 35: RAM-Inanspruchnahme
(Obelisk - Leerlauf)

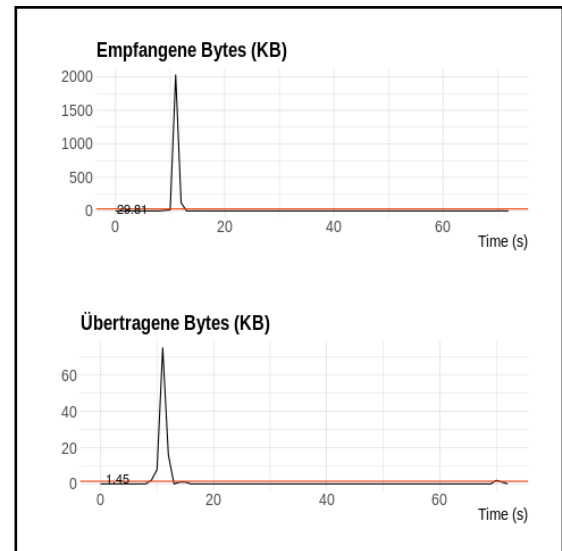


Abbildung 36: Empfangener/übertragener
Netzwerkdatenverkehr
(Obelisk - Leerlauf)

Interessant ist ebenfalls die Betrachtung des verwendeten Arbeitsspeichers für die Leerlauf-Messung. Während die Baseline-Messwerte ca. mit einer RAM-Inanspruchnahme von 4025 KB starten, liegt der Startwert für den Leerlauf bereits bei 5650 KB. Der Graph verläuft dabei, wie in Abbildung 35 zu sehen ist, gleich zur Baseline. Durch das Starten der Software bzw. der Datenbank werden also rund 1767 KB mehr an Arbeitsspeicher in Anspruch genommen.

Abbildung 36 zeigt, dass auch die Graphen für den Netzdatenverkehr gleich zu den Baseline-Messungen verlaufen. Die in das Netzwerk eingegangenen Daten sind in etwa 1,79 KB höher als bei der Baseline. Auffällig ist für den Netzwerkdatenanfang, dass der Baseline-Mittelwert rund 11 KB höher ist als der Leerlauf-Mittelwert.

5.1.4 Nutzungsszenario-Ergebnisse

Für die Messung der Nutzungsszenarien kommen zum Leerlauf zusätzlich die in 4.5.2 definierten Aktionen hinzu, welche ein realistisches Szenario eines Benutzers nachbilden. Hierfür werden nicht mehr die beiden Testanwendungen miteinander verglichen, sondern die beiden in Abschnitt 4.5.3 definierten Zustände für jede Software.

Efficient

Tabelle 13 zeigt die Messergebnisse beider Zustände im Vergleich.

| | Bezeichnung | Zustand 1 | Zustand 2 |
|------------|----------------------------------|-----------|-----------|
| Prozessor | CPU-Inanspruchnahme (Prozent) | 5,92 | 7,72 |
| Festplatte | Gelesene Daten (KB) | 1900,98 | 1901,02 |
| Festplatte | Geschriebene Daten (KB) | 1373,84 | 1314,42 |
| Speicher | Verwendeter Arbeitsspeicher (MB) | 4740,44 | 4748,67 |
| Netzwerk | Netzwerk Dateneingangs (KB) | 44,79 | 45,33 |
| Netzwerk | Netzwerk Datenausgang (KB) | 23,24 | 23,81 |

Tabelle 13: Nutzungsszenario Ergebnisse (Efficient)

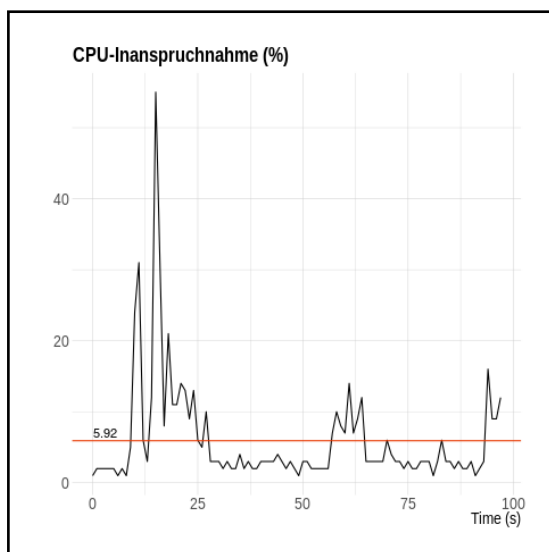


Abbildung 37: CPU-Inanspruchnahme (Efficient - Zustand 1)

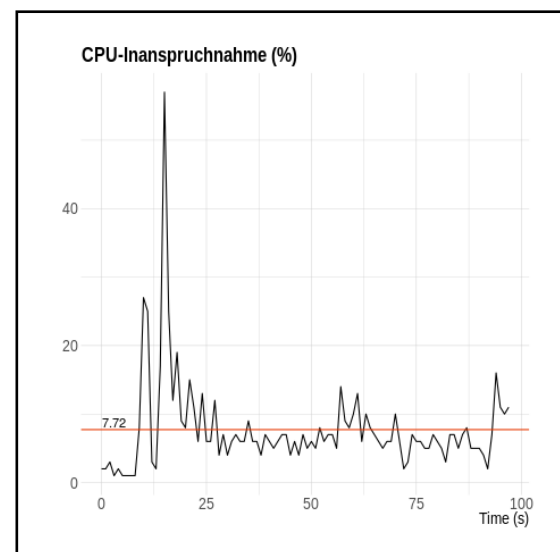


Abbildung 38: CPU-Inanspruchnahme (Efficient - Zustand 2)

Bei beiden Zuständen schlägt die CPU-Inanspruchnahme zu Beginn des Szenarios deutlich höher aus, als bei Baseline oder Leerlauf. Die beiden Abbildungen 37 und 38 zeigen nach zehn Sekunden der Messung, analog zum Leerlauf-Ergebnis, dass die CPU durch das Öffnen des Browsers ca. 30 Prozent in Anspruch genommen wird. Kurz darauf gehen die Werte beider Zustände hoch und erreichen dabei etwa 55 bis knapp unter 60 Prozent. Danach fallen die Messwerte direkt wieder.

Im Gegensatz zur Leerlauf-Messung befindet sich die Applikation nicht wie nach initialem Start in der Linienauswahl, sondern es wird zu Beginn zur Übersicht über alle neun Produktionslinien navigiert. Dadurch wird diese enorme Steigung verursacht. Im weiteren Verlauf schwankt die CPU-Inanspruchnahme des zweiten Zustands deutlich mehr. In Relation betrachtet ist der Mittelwert für die Inanspruchnahme des Prozessors des ersten Zustands ca. 23 Prozent niedriger als für den zweiten Zustand.

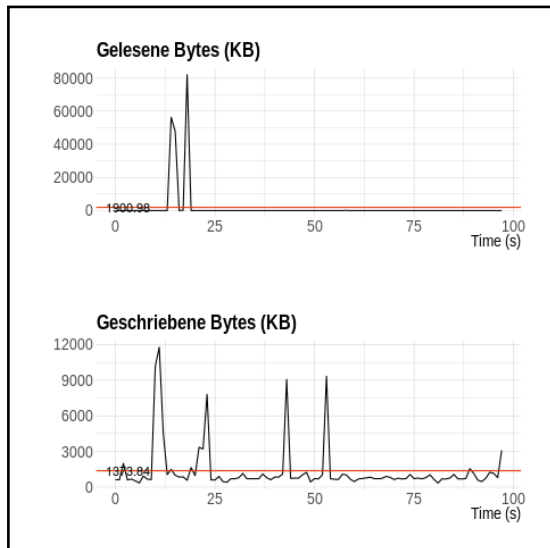


Abbildung 39: Gelesene/geschriebene Bytes der Festplatte (Efficient - Zustand 1)

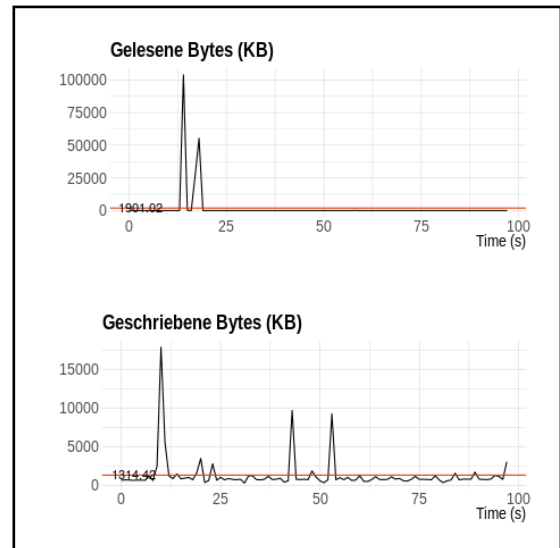


Abbildung 40: Gelesene/geschriebene Bytes der Festplatte (Efficient - Zustand 2)

Die Festplatte liest deutlich mehr Daten als zuvor während dem Leerlauf. Anstatt rund 56 KB sind es nun ca. 1900 KB, was einen enormen Anstieg zeigt. Die beiden Zustände unterscheiden sich dabei nur im sehr geringen Kilobyte-Bereich. Auch die geschriebenen Daten steigen im Vergleich zur Leerlauf-Messung etwas an. Wie auf den Abbildungen 39 und 40 zu erkennen ist, lässt sich für beide Zustände ein Unterschied von ungefähr 60 KB feststellen. Im Gegensatz zur CPU werden die Festplattenressourcen des ersten Zustands hier mit 4,5 Prozent höher beansprucht, als für den zweiten Zustand.

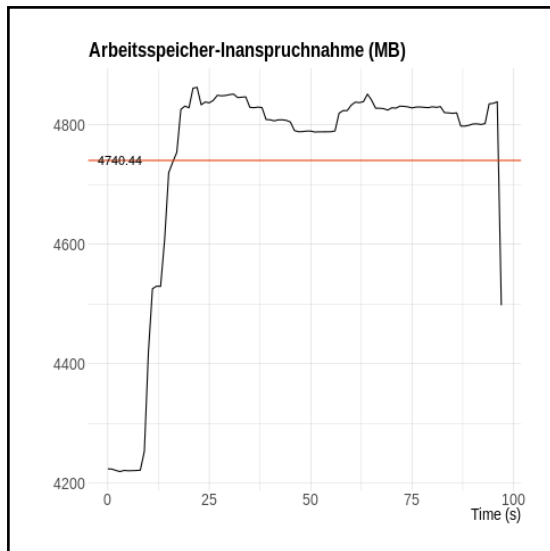


Abbildung 41: RAM-Inanspruchnahme
(Efficient - Zustand 1)

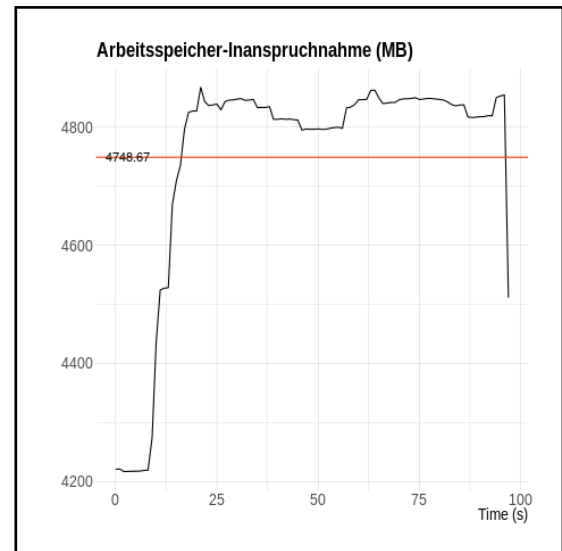


Abbildung 42: RAM-Inanspruchnahme
(Efficient - Zustand 2)

Der während des Szenarios verwendete Arbeitsspeicher steigt im Vergleich zum Leerlauf um ca. 470 bzw. 480 MB an. Der Mittelwert des zweiten Zustands, welcher die Anwendung ohne Backend-Caching darstellt, ist ca. um 8 MB größer als der des ersten Zustands. Der Unterschied ist in Relation betrachtet deutlich weniger als ein Prozent und ist somit eher weniger zu beachten.

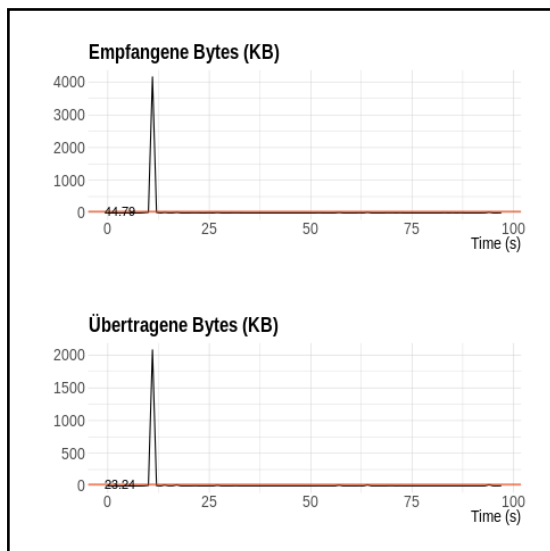


Abbildung 43: Empfangener/übertragener
Netzwerkdatenverkehr
(Efficient - Zustand 1)

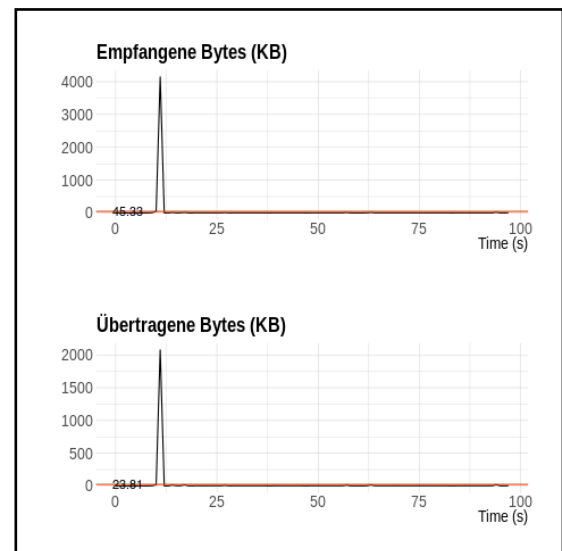


Abbildung 44: Empfangener/übertragener
Netzwerkdatenverkehr
(Efficient - Zustand 2)

Die über das Netzwerk empfangenen bzw. übertragenen Daten in KB unterscheiden sich nur wenig von der Leerlauf-Messung und auch nicht sonderlich innerhalb der beiden Zustände, wie in den beiden Abbildungen 43 und 44 zu erkennen ist. Vergleicht man die empfangenen Kilobytes der Leerlauf-Messung in Relation zu den gerundeten 45 KB aus beiden Zuständen, fällt auf, dass der Mittelwert des Leerlaufs hierbei um ca. 28 Prozent höher ausfällt.

Für die übertragenen Daten der Leerlauf-Messung kommt ebenso ein Unterschied von ungefähr 21 bzw. 24 Prozent zu Stande, bei welchem der Leerlauf-Mittelwert jeweils höher als Zustand 1 bzw. Zustand 2 ist.

Tabelle 14 zeigt anhand der in 5.1.1 vorgestellten Formel die Mittelwerte $M_{\text{Nutzungsszenario-Baseline}}$ beider Zustände für das Nutzungsszenario ohne Einfluss der in der Baseline festgelegten Dienste und Programme.

| | Bezeichnung | Zustand 1 | Zustand 2 |
|------------|----------------------------------|-----------|-----------|
| Prozessor | CPU-Inanspruchnahme (Prozent) | 4,26 | 6,06 |
| Festplatte | Gelesene Daten (KB) | 1900,96 | 1901 |
| Festplatte | Geschriebene Daten (KB) | 962,45 | 903,03 |
| Speicher | Verwendeter Arbeitsspeicher (MB) | 742,29 | 750,52 |
| Netzwerk | Netzwerk Dateneingang (KB) | 18,47 | 19,01 |
| Netzwerk | Netzwerk Datenausgang (KB) | 21,63 | 22,2 |

Tabelle 14: Nutzungsszenario Ergebnisse ohne Einfluss der Baseline (Efficient)

Die Übersicht zeigt, dass die CPU-Inanspruchnahme den größten und wichtigsten Unterschied auch ohne Einfluss der Baseline ausmacht. Die Lücke zwischen den beiden Zuständen wird sogar noch etwas größer. Die CPU-Inanspruchnahme des ersten Zustands, welcher die Software unter Verwendung einer serverseitigen Zwischenspeicherung simuliert, ist um fast 30 Prozent geringer, als die des zweiten Zustands. Ebenso steigt auch die Differenz der geschriebenen Daten etwas an, wobei der erste Zustand hier etwa 6,5 Prozent mehr an Kilobytes auf die Festplatte schreibt, als der zweite Zustand. Die Unterschiede der anderen Messwerte sind eher weniger interessant.

Obelisk

Tabelle 15 gibt eine Übersicht über die verschiedenen Mittelwerte beider definierten Zustände im direkten Vergleich.

| | Bezeichnung | Zustand 1 | Zustand 2 |
|------------|----------------------------------|-----------|-----------|
| Prozessor | CPU-Inanspruchnahme (Prozent) | 12,33 | 15,89 |
| Festplatte | Gelesene Daten (KB) | 692,97 | 396,96 |
| Festplatte | Geschriebene Daten (KB) | 879,38 | 644,66 |
| Speicher | Verwendeter Arbeitsspeicher (MB) | 6890,21 | 6894,95 |
| Netzwerk | Netzwerk Dateneingang (KB) | 37,55 | 27,56 |
| Netzwerk | Netzwerk Datenausgang (KB) | 1,97 | 1,39 |

Tabelle 15: Nutzungsszenario Ergebnisse (Obelisk)

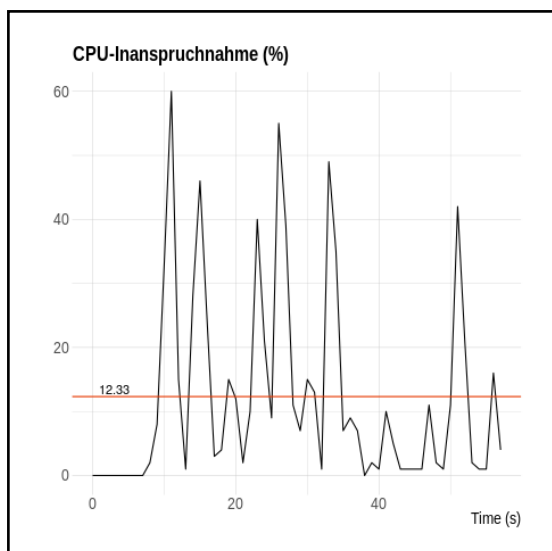


Abbildung 45: CPU-Inanspruchnahme
(Obelisk - Zustand 1)

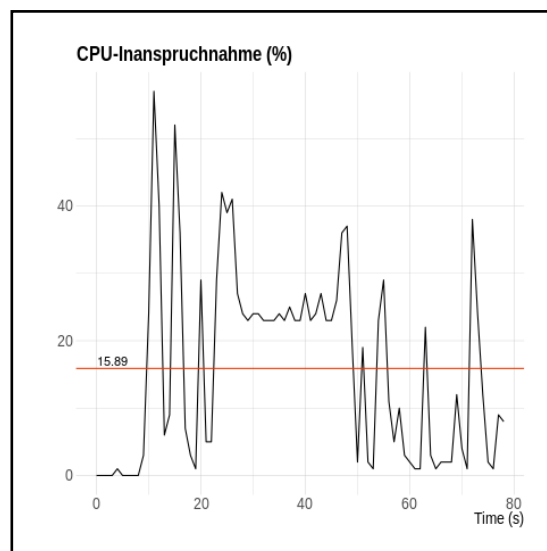


Abbildung 46: CPU-Inanspruchnahme
(Obelisk - Zustand 2)

Die beiden Abbildungen 45 und 46 veranschaulichen die CPU-Inanspruchnahme der zwei definierten Zustände. Von einer CPU-Auslastung von ca. 3,3 Prozent, steigen die Messwerte von dem Leerlauf zum Nutzungsszenario auf 12,33 bzw. 15,89 Prozent. Die beiden dargestellten Graphen verlaufen dabei bis zu einem gewissen Punkt ähnlich. Nach 10 Sekunden Leerlauf zu Beginn der Messung wird jeweils der Browser geöffnet und die entsprechende URL der Webapplikation aufgerufen. Dabei ist bei beiden Abbildungen ein deutlicher Ausschlag zu erkennen, der bis ungefähr 60 Prozent geht. Danach werden verschiedene Aktionen durchgeführt, bei denen die Ausschläge der beiden Zustände zwar leicht unterschiedlich ausfallen, aber dennoch ähnlich. Ab Sekunde 20 bis 25 wird der für diese Messung maßgebliche CSV-Export durchgeführt. Dieser ist für den ersten Zustand so implementiert, dass die generierte CSV-Datei innerhalb von mehreren Millisekunden bereit zum Herunterladen ist. Der zweite Zustand setzt dabei auf eine verschachtelte Implementierung, sodass während des Exports tausende Schleifen durchlaufen werden, was zu einer Verzögerung von bis zu 30 Sekunden führen kann. Deswegen ist es nicht verwunderlich, dass beide Graphen ab diesem Zeitpunkt komplett verschieden aussehen. Während der erste Zustand mit den weiteren Aktionen des Szenarios fortführt, hängt der zweite Zustand von etwa Sekunde 25 bis 50 mit nur einer Aktion fest und bleibt dabei die meiste Zeit ungefähr auf einer Linie mit mehreren leichten Schwankungen. Auch die letzten automatisierten Nutzeraktionen der beiden Messungen haben nur noch wenig Ähnlichkeit, da der zweite Zustand insgesamt deutlich höher anschlägt und die Schwankungen auch größer sind. Der erste Zustand nimmt in Relation betrachtet 22,4 Prozent der CPU weniger in Anspruch als der zweite Zustand. An dieser Messung wird besonders deutlich gemacht, dass ein kleiner Unterschied in der Implementierung schnell einen großen Unterschied in der Ressourceneffizienz bewirken kann. Nicht nur, dass die zeitliche Ausführung steigt, sondern auch das gesamte Verhalten des Prozessors nach Ausführung der besagten Aktion verändert sich.

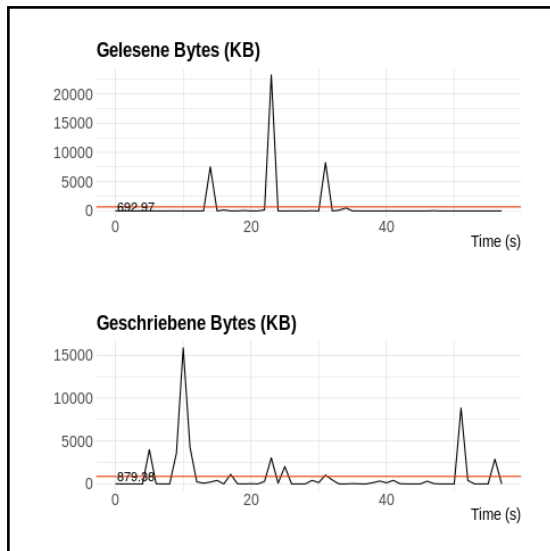


Abbildung 47: Gelesene/geschriebene Bytes der Festplatte (Obelisk - Zustand 1)

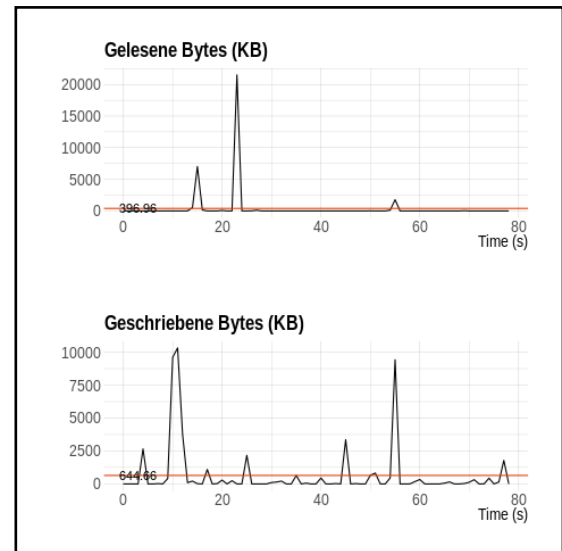


Abbildung 48: Gelesene/geschriebene Bytes der Festplatte (Obelisk - Zustand 2)

Auf Seiten der Festplatte lässt sich ein großer Unterschied erkennen, sowohl zum Leerlauf, als auch untereinander. Im Gegensatz zum Leerlauf werden beim zweiten Zustand anstatt rund 6,7 KB etwa 397 KB an Daten gelesen. Der erste Zustand liest sogar knapp 693 Kilobytes an Daten. Die beiden Graphen, welche in Abbildung 47 bzw. 48 dargestellt sind, sehen dabei im Grunde fast gleich aus. Bei näherer Betrachtung fällt jedoch auf, dass der Graph des ersten Zustands öfter ausschlägt als für den zweiten Zustand, wodurch der Unterschied zwischen der Menge der gelesenen Daten erklärt ist. Der Mittelwert für den zweiten Zustand ist dabei um 42,7 Prozent geringer als für den ersten. Die geschriebenen Kilobytes sind ebenfalls bei beiden Messungen angestiegen und auch hier ist Zustand 1 mit rund 880 KB erneut am höchsten. Während die Daten für den zweiten Zustand im Verhältnis nur um 2,9 Prozent höher sind als bei dem Leerlauf, ist der Mittelwert trotzdem um 26,7 Prozent kleiner als für den ersten Zustand.

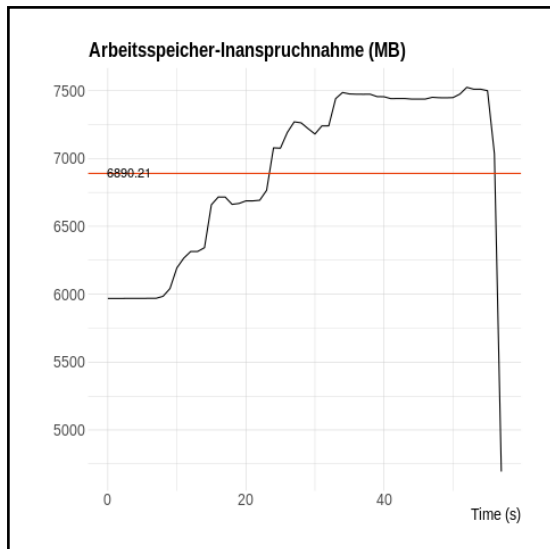


Abbildung 49: RAM-Inanspruchnahme
(Obelisk - Zustand 1)

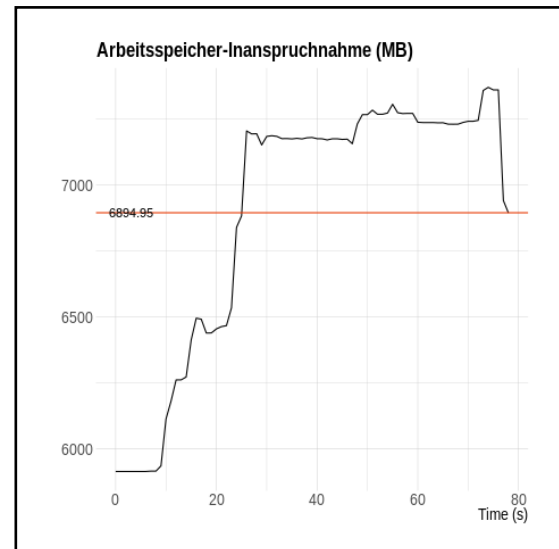


Abbildung 50: RAM-Inanspruchnahme
(Obelisk - Zustand 2)

Betrachtet man den verwendeten Arbeitsspeicher während des Szenarios, fällt auf, dass die Anwendung bei beiden Zuständen zu Beginn bei etwas unter einer Inanspruchnahme von 6000 MB des Arbeitsspeichers liegt. Der Mittelwert der beiden Zustände liegt dabei relativ ähnlich bei ungefähr 6890 Megabyte und ist damit rund 15 Prozent höher als für den Leerlauf. Nach Öffnen des Browser bzw. der URL der Applikation, wird der Login durchgeführt. Dabei sehen die Graphen ähnlich aus und steigen auch ungefähr gleich hoch an. Der anschließende Verlauf wirkt dabei grundverschieden. Analog zum Messergebnis für die CPU-Inanspruchnahme lässt sich hier die 20 Sekunden Marke nennen. Hier wird der CSV-Export durchgeführt, der für den ersten Zustand schnell ausgeführt wird. Deswegen steigt der entsprechende Graph stetig bis zu einem gewissen Punkt an, der bei ca. 7500 MB liegt. Die Aktionen, welche gegen Ende des Szenarios erfolgen, haben also keinen großen Einfluss mehr auf den Arbeitsspeicher. Für die zweite Messung sieht der Verlauf ab Sekunde 20 allerdings anders aus. Der Graph steigt deutlich steiler an und geht bis kurz unter eine Inanspruchnahme von 7500 MB. Von Sekunde 25 bis 50 bleiben die Messwerte auf einer Linie und bei Abschluss bzw. Ausführung der nächsten Aktionen steigt der Graph erst auf die 7500 MB bzw. teilweise sogar noch etwas höher. Bei Betrachtung der beiden Abbildungen 49 und 50 fällt sofort auf, dass das Ende verschieden aussieht. So sind beim zweiten Zustand mehr Schwankungen zu erkennen. Außerdem fällt der Graph bei Schließen des Browsers nicht so weit herunter wie für den ersten Zustand. Trotz der vielen Unterschiede des Messverlaufs nimmt Zustand 2 nur weniger als 0,1 Prozent mehr in Anspruch als der erste Zustand. Dies liegt daran, dass der erste Zustand eine deutlich längere Ausführungszeit hat und dass der Graph zwar mehr schwankt bzw. teilweise höher ausschlägt, allerdings erreicht der erste Zustand auch wesentlich schneller die 7500 MB Marke.

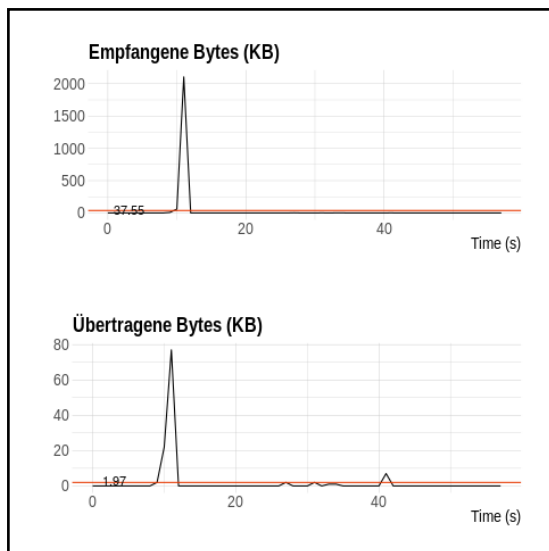


Abbildung 51: Empfangener/übertragener
Netzwerkdatenverkehr
(Obelisk - Zustand 1)

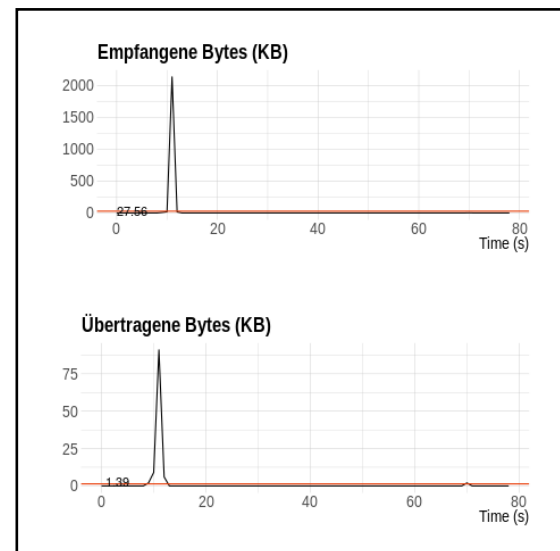


Abbildung 52: Empfangener/übertragener
Netzwerkdatenverkehr
(Obelisk - Zustand 2)

Die in Abbildung 51 und 52 dargestellten Messwerte zum Netzdatenverkehr unterscheiden sich nur unbedeutend von den Leerlauf-Messungen. Der Verlauf der beiden Messungen für empfangene Kilobytes wirkt identisch. Bei genauere Betrachtung fällt allerdings auf, dass die beiden Ausschläge zwar gleich hoch gehen, jedoch benötigt der erste Zustand etwas mehr Zeit für die Ausführung dieser Aktion. Deswegen ist der Mittelwert des zweiten Zustands im Verhältnis um 26,6 Prozent geringer. Die übertragenen Daten über das Netzwerk unterscheiden sich dabei noch geringer.

Tabelle 16 gibt eine Übersicht darüber, wie die Messergebnisse ohne Einfluss der in der Baseline festgelegten Dienste und Programme aussehen. Von den Messergebnissen werden also nach Verwendung der in 5.1.1 vorgestellten Formel die Werte der Baseline-Messergebnisse abgezogen und somit wird jeweils der Mittelwert $M_{Nutzungsszenario-Baseline}$ berechnet.

| | Bezeichnung | Zustand 1 | Zustand 2 |
|------------|----------------------------------|-----------|-----------|
| Prozessor | CPU-Inanspruchnahme (Prozent) | 10,3 | 13,86 |
| Festplatte | Gelesene Daten (KB) | 686,43 | 390,42 |
| Festplatte | Geschriebene Daten (KB) | 399,55 | 164,83 |
| Speicher | Verwendeter Arbeitsspeicher (MB) | 2683,28 | 2688,02 |
| Netzwerk | Netzwerk Dateneingang (KB) | 10,53 | 0,54 |
| Netzwerk | Netzwerk Datenausgang (KB) | -10,45 | -11,03 |

Tabelle 16: Nutzungsszenario Ergebnisse ohne Einfluss der Baseline (Obelisk)

Interessant für die Ressourceneffizienz ist es, dass die Lücke der beiden Zustände für die CPU-Inanspruchnahme ohne Einfluss der Baseline größer wird. Zustand 1 ist somit nicht mehr 22,4 Prozent, sondern sogar 25,7 Prozent kleiner als der zweite Zustand. Ebenso fällt die Lücke für die eingegangenen Netzwerkdaten deutlich höher aus, wenn auch nur im niedrigen Kilobyte-Bereich. Die anderen Messwerte unterscheiden sich im Verhältnis ohne den Einfluss der Baseline nicht besonders zu den in diesem Abschnitt vorgestellten Ergebnissen. Auffällig ist zudem noch, dass die übertragenen Daten über das Netzwerk bei beiden Zuständen insgesamt kleiner sind als die Werte für die Baseline, weswegen diese in dieser Übersicht negative Mittelwerte annehmen.

5.2 Bewertung der Messergebnisse

Die letzten Abschnitte haben alle Messergebnisse zu sowohl Baseline, Leerlauf als auch zu den beiden Zuständen für die Nutzungsszenarien vorgestellt. Dieser Abschnitt soll die visualisierten und beschriebenen Ergebnisse bewerten und dabei soll insbesondere ein Augenmerk auf den Vergleich beider Zustände der Softwareprodukte gelegt werden.

Insgesamt lässt sich sagen, dass die Menge der Netzwerkdaten für den Ein- und Ausgang bei allen Messungen relativ niedrig und gleich geblieben ist. Ebenso unterscheiden sich die gelesenen sowie geschriebenen Daten auf der Festplatte nicht bedeutsam, dort ist eher ein großer Unterschied zur Baseline bzw. dem Leerlauf zu erkennen. Interessanter wird es bei der Betrachtung der anderen beiden Messwerte. Die CPU-Inanspruchnahme unterscheidet sich für den Efficient bzw. Obelisk mit 23 bzw. 27 Prozent zwischen den jeweiligen Zuständen. Hier wird deutlich, dass sich die CPU-Inanspruchnahme durch Anpassungen am Quellcode enorm ändern kann bzw. dass der Entwickler eine solche Erhöhung einfach vermeiden kann. Am Beispiel der Messungen des Obelisk wird deutlich, dass trotz unbedeutender Unterschiede der Arbeitsspeicher-Inanspruchnahme zweier Zustände dennoch ein komplett verschiedener Verlauf zu sehen sein kann. Auffällig ist bei allen Messungen der beiden Zustände des Obelisk, dass die Zeit unterschiedlich ist. Einmal werden rund 60 und einmal 80 Sekunden benötigt, um das Nutzungsszenario durchzuführen. Dies liegt daran, dass bei den Messungen für den zweiten Zustand ein verlangsamter CSV-Export ausgeführt wird. Diese Aktion hat wie bereits erwähnt den gesamten Verlauf der Arbeitsspeicher-Inanspruchnahme verändert, obwohl der Mittelwert der beiden Messungen dabei fast gleich ist.

Durch die Messungen der Applikationen stehen nun etliche Informationen zu der Hardware-Auslastung zur Verfügung. Ein konkreter Energieverbrauch kann jedoch nicht genannt werden, da die Beziehung zwischen Energieverbrauch und Hardware-Inanspruchnahme noch nicht klar ist. Ein zusätzliches Problem ist, dass der Entwickler anhand der visualisierten Ergebnisse nicht hundertprozentig sagen kann, zu welchem Zeitpunkt welche Aktion durchgeführt wird. Die steigende CPU- oder Arbeitsspeicher-Inanspruchnahme könnte beispielsweise durch den Quellcode oder auch die Datenbank verursacht werden. Wünschenswert wäre an dieser Stelle eine genaue Auskunft, welche Komponente für den Anstieg oder die Senkung von Messwerten verantwortlich ist.

5.3 Bewertung des Messkonzepts

Das in Abschnitt 4.3.1 vorgestellte bisherige Messkonzept des ISS wurde um ein automatisiertes Messkonzept unter Verwendung von Continuous Integration erweitert. Ziel dieses neuen Konzepts ist die Unterstützung des ISS bei der Messung und Bewertung von Software auf deren Ressourceneffizienz, indem Softwareprodukte in den CI-Server eingepflegt werden und anschließend jede Nacht automatisiert gemessen werden. Abschnitt 4.4 hat bereits Vor- und Nachteile beider Konzepte aufgezeigt. Nun soll das Messkonzept erneut bewertet werden, allerdings nachdem das gesamte Verfahren anhand der beiden Testapplikationen getestet wurde.

Es hat sich herausgestellt, dass ein CI-Server sehr leicht und schnell eingerichtet ist, es aber dennoch komplex bei der Einpflegung und Konfiguration von Projekten in der Jenkins-Plattform werden kann. Dies liegt daran, dass sich die beiden Testanwendungen in diesem Fall im Aufbau und ihrer Architektur sehr unterscheiden. Zudem wird das Gestalten von Nutzungsszenarien mit Selenium sehr schnell kompliziert. Grund dafür ist vor allem ein komplexer Aufbau innerhalb eines Frameworks für das Frontend. Mit Vaadin, welches die Basis für die Obelisk-Applikation darstellt, ist es teilweise nicht einfach, verschiedene Elemente über CSS-Selektoren anzusprechen. Dafür sind sehr fortgeschrittene Kenntnisse notwendig, weswegen man sich auch erst in den Aufbau des Document Object Model (DOM) bzw. in die Selenium-Bibliothek einarbeiten muss. In Abschnitt 5.4 wird das Gesamtfazit dieser Ausarbeitung erläutert sowie ein Ausblick gegeben. An dieser Stelle werden die Probleme der Automatisierungssoftware Selenium abschließend noch einmal aufgegriffen.

Der letzte Abschnitt hat bereits gezeigt, dass es bisher an genauen Ergebnissen zum Zusammenhang zwischen Energieverbrauch und Hardware-Inanspruchnahme fehlt. Deswegen lassen sich für diese Thesis anhand des neuen Messkonzepts auch keine konkreten Äußerungen zum Energieverbrauch tätigen. Aushilfe dafür könnte eine Ergänzung des Konzepts schaffen, die neben der Messung der Hardware-Auslastung auch ebenso den Energieverbrauch misst. Dieser Punkt wird ebenso im Fazit abschließend noch einmal aufgegriffen.

Die meisten der genannten Probleme lassen sich allerdings recht einfach beheben. Für die Konfiguration der Jenkins-Projekte lassen sich Vorlagen erstellen und für die Nutzung von Selenium muss eine kurze oder längere Einarbeitungszeit erfolgen. Steht das Nutzungsszenario, profitiert man als Entwickler deutlich von den Vorteilen des automatisierten Konzepts, da täglich neue Messwerte bzw. visualisierte Ergebnisse vorliegen, ohne dass man sich darum kümmern muss. Zudem ist die Weiterentwicklung von Szenarien oder Sonstigem nicht sehr kompliziert, da man sich an bisherigen Abläufen gut orientieren kann. Ein weiterer Vorteil ist die Einsparung von Energie beim Messkonzept an sich, da der CI-Server zeitgesteuert angeschaltet wird und am Ende der letzten Messung wieder heruntergefahren wird. Auf diese Weise muss der Server nicht dauerhaft angeschaltet sein, sondern wird nur hochgefahren, wenn er genutzt wird, wodurch keine unnötige Energie verschwendet wird.

5.4 Fazit und Ausblick

In dieser Thesis wurde auf Basis des Messkonzepts des Institut für Softwaresysteme am Umwelt-Campus in Birkenfeld ein erweitertes Konzept entwickelt, welches mittels Continuous Integration automatisiert durchgeführt werden kann. Ziel des Messkonzepts ist die Unterstützung des ISS bei der Messung und Bewertung von Software auf Ressourceneffizienz. Der automatisierte Weg soll das ISS insbesondere bei den aktuellen Plänen unterstützen, Client-Server-Systeme bzw. mobile Anwendungen mit dem Blauen Engel zertifizieren zu können.

In Kapitel 5 wurden zwei Softwareprodukte zum Test des Messkonzepts vorgestellt bzw. betrachtet. Dabei wurden Messwerte zum Prozessor, der Festplatte, dem Arbeitsspeicher sowie dem Netzdatenverkehr untersucht. Ziel der Messungen war es, einen signifikanten Unterschied in der Hardware-Inanspruchnahme zwischen zwei Versionen der gleichen Software zu finden. Dafür wurden in Abschnitt 4.5.3 für beide Testapplikationen jeweils zwei Zustände definiert, anhand derer ein Unterschied festzustellen sein sollte. Jeder Zustand steht dafür sinnbildlich für einen Commit im Versionsverwaltungssystem. Das heißt, dass an einem Tag eine neue Version in das verwendete Repository gepusht wird und die nächste Version beispielsweise bereits am darauffolgenden Tag. So werden zwei verschiedene Softwareversionen betrachtet, die entweder nur wenige und unwichtige Änderungen aufweisen oder aber werden bedeutende Algorithmen angepasst, die einen Einfluss auf die Ressourceneffizienz haben.

Der letzte Abschnitt hat gezeigt, dass durch einfache Änderungen am Quellcode bzw. an der Algorithmik bereits große Unterschiede der Ressourcen-Inanspruchnahme zu erkennen sind. Ein Problem ist dabei, dass der Zusammenhang zwischen Hardware-Inanspruchnahme und Energieverbrauch immer noch nicht ganz klar ist, sodass es beim entwickelten Messkonzept an einer konkreten Zahl mangelt, die etwas über den Energieverbrauch aussagt. Beim bisherigen Messaufbau des ISS wird zur Energiemessung das Tool GridVis für Windows verwendet. Sinnvoll wäre es also, dass das in dieser Ausarbeitung entwickelte Messverfahren um eine Energiemessung mit GridVis erweitert wird. Dieser wichtige Aspekt wurde hier aus Komplexitätsgründen ausgelassen, da man nicht ohne weiteres mit einem automatisierten Ansatz an die Ergebnisse der Energiemessung herankommt. Dabei wäre es sinnvoll, sich näher mit dem *Simple Network Monitoring Protocol (SNMP)* auseinanderzusetzen. Das Protokoll dient zur Übertragung von Verwaltungsinformationen in Netzwerken und ermöglicht dabei die Sammlung von Informationen zu vernetzten Geräten über mehrere Hardware- und Softwaretypen hinweg [Pae22].

Das Experteninterview mit Hendrik Schneider von der Fresenius Digital Technology (FDT), welches in Abschnitt 3.3.1 aufgegriffen wurde, hat gezeigt, dass die Qualitätskontrolle im bisherigen Sinne am Beispiel des großen Unternehmens FDT gut abgedeckt ist. Abschnitt 2.1 hat allerdings auch die Wichtigkeit der Green IT bzw. nachhaltiger Software aufgezeigt. Laut Interviewpartner Hendrik Schneider wird im Entwicklungsteam aktiv nicht auf nachhaltige Software geachtet, sondern nur indirekt durch sauberen Quellcode. Zudem wird bei der Realisierung von Anforderungen Wert auf Einsparung von Ressourcen gelegt. Das in dieser Thesis entwickelte Messkonzept unterstützt die Entwickler bei der aktiven Untersuchung auf Ressourceneffizienz. Für Unternehmen wie die FDT, die bereits Erfahrungen im Bereich Continuous Integration gemacht haben bzw. aktiv auf CI-Prozesse setzen, sollte es nicht allzu kostspielig sein, das automatisierte Messkonzept einzuführen.

Ein erster Vergleich des bisherigen Messkonzepts des ISS und des in dieser Thesis entwickelten Verfahrens wurde in Abschnitt 4.4 bereits gezogen und erläutert. Dieser Vergleich hat gezeigt, dass das bisherige Messkonzept gut funktioniert, da in den letzten Jahren schon eine Vielzahl an Softwareprodukten gemessen und bewertet wurden, wobei dabei immer aussagekräftige Ergebnisse zum Vorschein kamen. Zudem können die Messungen im Gegensatz zum automatisierten Messkonzept von Mitarbeitern mit wenigen Vorkenntnissen durchgeführt werden. Deshalb sollte man nicht mit manuellen Messungen aufhören, eher sollte man diese bei Möglichkeit um automatisierte Softwaremessungen ergänzen, da Automatisierung heutzutage eine große Rolle spielt. Durch die automatische Ausführung von repetitiven Aufgaben werden sowohl Zeit als auch Kosten gespart. Zudem können unnötige Fehler verhindert werden, die ein Mitarbeiter bei manueller Bearbeitung machen könnte. Der Untersuchungsprozess auf nachhaltige Software sollte sich in Zukunft demnach in jedem Fall in eine automatisierte Richtung bewegen und dennoch das Augenmerk auf manuelle Messungen nicht verlieren.

Das in dieser Thesis entwickelte Messkonzept kann umfangreich weiterentwickelt werden. Die größte Beachtung sollte der Ergänzung von Energiemessungen gegeben werden. Ein weiterer wichtiger Aspekt, welcher bislang noch unklar ist, ist das Verhältnis von Hardware-Inanspruchnahme und Energieverbrauch. Um die in der Evaluation aufgestellten Ergebnisse besser auswerten und analysieren zu können, wäre es wichtig zu wissen, was eine bestimmte Auslastung von beispielsweise der CPU für konkrete Auswirkungen auf den Energieverbrauch hat. Zusätzlich dazu könnten anstatt Selenium weitere Tools und Bibliotheken zur Durchführung der Nutzungsszenarien verwendet werden. Zur Einfachheit wurde für diese Ausarbeitung auf die Ausführung von 30 Durchläufen innerhalb eines Szenarios verzichtet, stattdessen wird das Szenario jeweils nur einmal durchgeführt. Um genauere Messwerte zu bekommen, wäre auch dies ein Thema, welches in Zukunft angegangen werden könnte. Eine genaue Auskunft zu jeder Messung ist eine weitere Funktion, die für das automatisierte Konzept wünschenswert wäre. Dabei ist interessant, welche Komponente für einen Anstieg oder eine Senkung der Messwerte verantwortlich ist.

Literaturverzeichnis

- [Com09] ComOrg. *3rd International Symposium on Empirical Software Engineering and Measurement, 2009: ESEM 2009 ; 15 - 16 Oct. 2009, Lake Buena Vista, Florida, USA ; [in conjunction with] MetriSec, [the International Workshop on Security Measurements and Metrics ; part of the Empirical Software Engineering International Week (ESEIW)]*. Piscataway, NJ: IEEE, 2009. ISBN: 9781424448425. URL: <https://www.computer.org/csdl/proceedings/esem/2009/4842/00/index.html>.
- [Kla09] Michael Klaes. *3rd International Symposium on Empirical Software Engineering and Measurement, 2009: ESEM 2009 ; 15 - 16 Oct. 2009, Lake Buena Vista, Florida, USA ; [in conjunction with] MetriSec, [the International Workshop on Security Measurements and Metrics ; part of the Empirical Software Engineering International Week (ESEIW)]*. Piscataway, NJ: IEEE, 2009. ISBN: 9781424448425. URL: <https://www.computer.org/csdl/proceedings/esem/2009/4842/00/index.html>.
- [DNH10] Markus Dick, Stefan Naumann und Alexandra Held. “Green Web Engineering. A Set of Principles to Support the Development and Operation of “Green” Websites and their Utilization during a Website’s Life Cycle”. In: *Filipe, Joaquim* (2010), S. 7–10.
- [Beh11] Mario Behrendt. *Jenkins: Kurz & gut*. 1. Aufl. Köln: O’Reilly, 2011. URL: <https://learning.oreilly.com/library/view/-/9783868993905/?ar>.
- [Ker+13] Eva Kern u. a. “Green software and green software engineering—definitions, measurements, and quality aspects”. In: *First International Conference on Information and Communication Technologies for Sustainability (ICT4S2013), 2013b ETH Zurich*. 2013, S. 87–91.
- [AB14a] R Amri und N Bellamine Ben-Saoud. *2014 Fourth International Conference on Advances in Computing and Communications (ICACC): 27 - 29 Aug. 2014, Kochi, Kerala, India*. Piscataway, NJ: IEEE, 2014. ISBN: 9781479943623. URL: <http://ieeexplore.ieee.org/servlet/opac?punumber=6901615>.
- [AB14b] R Amri und N Bellamine Ben-Saoud. “Using sustainable software model to identify constraints and opportunities toward sustainability in software engineering”. In: *Proc. 23rd IBIMA Conference*. 2014.
- [KAB14] Sedef Akinli Koçak, Gülfem Isiklar Alptekin und Ayse Bener. “Evaluation of Software Product Quality Attributes and Environmental Attributes using ANP Decision Framework.” In: *RE4SuSy@ RE*. 2014, S. 37–44.
- [AE15] Anders S. G. Andrae und Tomas Edler. “On Global Electricity Usage of Communication Technology: Trends to 2030”. In: 6 (2015), S. 117–157. URL: <https://www.mdpi.com/2078-1547/6/1/117>.
- [Cal+15] Coral Calero u. a. “Green Software and Software Quality”. In: *Green in Software Engineering*. Hrsg. von Coral Calero und Mario Piattini. Cham: Springer International Publishing, 2015, S. 231–260. ISBN: 978-3-319-08581-4. DOI: 10.1007/978-3-319-08581-4_10. URL: https://doi.org/10.1007/978-3-319-08581-4_10.

- [KAB15] Sedef Akinli Koçak, Gülfem Isiklar Alptekin und Ayse Basar Bener. “Integrating Environmental Sustainability in Software Product Quality.” In: *RE4SuSy@ RE*. 2015, S. 17–24.
- [Nau+15] Stefan Naumann u. a. “Sustainable Software Engineering: Process and Quality Models, Life Cycle, and Social Aspects”. In: *ICT Innovations for Sustainability*. Springer, Cham, 2015, S. 191–205. DOI: 10.1007/978-3-319-09228-7₁₁. URL: https://link.springer.com/chapter/10.1007/978-3-319-09228-7_11.
- [www16] www.umweltschutz.de. *Umweltschutz - Daten, Fakten, Hintergründe & Zusammenhänge*. 2016. URL: <http://www.umweltschutz.de/> (besucht am 22.02.2022).
- [BBB17] Mohamed Amine Beghoura, Abdelhak Boubetra und Abdallah Boukerram. “Green software requirements and measurement: random decision forests-based software energy consumption profiling”. In: *Requirements Engineering* 22.1 (2017), S. 27–40. ISSN: 0947-3602. DOI: 10.1007/s00766-015-0234-2.
- [Col18] Collectl. *collectl*. 2018. URL: <http://collectl.sourceforge.net/> (besucht am 07.02.2022).
- [CL18a] O.N. Condori Fernandez und P. Lago. *A Sustainability-quality Model*. English. Version 1.0. VU Technical Report, Nov. 2018.
- [CL18b] Nelly Condori-Fernandez und Patricia Lago. “Characterizing the contribution of quality requirements to software sustainability”. In: *Journal of systems and software* 137 (2018), S. 289–305.
- [GK18] Oleksandr Gordieiev und Vyacheslav Kharchenko. “IT-oriented software quality models and evolution of the prevailing characteristics”. In: *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. IEEE. 2018, S. 375–380.
- [Grö+18] Jens Gröger u. a. “Entwicklung und Anwendung von Bewertungsgrundlagen für ressourceneffiziente Software unter Berücksichtigung bestehender Methodik-Abschlussbericht”. In: *UBA TEXTE* 105 (2018).
- [Ker+18] Eva Kern u. a. “Sustainable software products—Towards assessment criteria for resource and energy efficiency”. In: *Future Generation Computer Systems* 86 (2018), S. 199–210. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2018.02.044>. URL: <https://www.sciencedirect.com/science/article/pii/S0167739X17314188>.
- [CL19] Nelly Condori-Fernandez und Patricia Lago. “Towards a software sustainability-quality model: Insights from a multi-case study”. In: *2019 13th International Conference on Research Challenges in Information Science (RCIS)*. IEEE. 2019, S. 1–11.
- [Lub19] Stefan Lubert. “Was ist ein Active Directory?” In: *IP-Insider* (2019). URL: <https://www.ip-insider.de/was-ist-ein-active-directory-a-626455/> (besucht am 23.02.2022).

- [mon19] mona-gleissner.com. *Geschlechtergerechte Sprache: Gendern leicht gemacht*. 2019. URL: <https://mona-gleissner.com/2019/06/28/geschlechtergerechte-sprache-gendern-leicht-gemacht/> (besucht am 01.02.2022).
- [Sch19] Andreas Schneider. “So viel Energie verbraucht das Internet”. In: *Quarks* (2019). URL: <https://www.quarks.de/technik/energie/so-viel-energie-verbraucht-das-internet/> (besucht am 01.02.2022).
- [Shi19] Boris Shiklo. “8 Vorgehensmodelle der Softwareentwicklung: mit Grafiken erklärt”. In: *ScienceSoft* (2019). URL: <https://www.scnsoft.de/blog/vorgehensmodelle-der-softwareentwicklung> (besucht am 02.02.2022).
- [Lee20] Glenn Lee. “Arten von Softwaretests: Unterschiede und Beispiele”. In: *LoadView by Dotcom-Monitor* (2020). URL: <https://www.loadview-testing.com/de/blog/arten-von-softwaretests-unterschiede-und-beispiele/> (besucht am 01.02.2022).
- [msg20] msg. *Nutzungsszenario — User-Experience-Methoden-Katalog*. 2020. URL: <https://www.user-experience-methods.com/specification/usage-scenario.html> (besucht am 07.03.2022).
- [Plö20] Michael Plöd. “Identifikation von Qualitätsanforderungen mit Quality Storming — Collaborative Modeling für die skillübergreifende Erfassung und Priorisierung von Qualitätsanforderungen”. In: *INNOQ* (2020). URL: <https://www.innoq.com/de/articles/2020/03/quality-storming-workshop/> (besucht am 09.02.2022).
- [Tec20] Technosoft.de. *Funktionale Tests*. 2020. URL: <https://www.technosoft.de/qa-testing/funktionale-tests> (besucht am 01.02.2022).
- [Atl21a] Atlassian. *Was ist Continuous Integration?* — Atlassian. 2021. URL: <https://www.atlassian.com/de/continuous-delivery/continuous-integration> (besucht am 10.12.2021).
- [Atl21b] Atlassian. *Was ist Git: Mit diesem Leitfaden wirst du zum Git-Profi*. 2021. URL: <https://www.atlassian.com/de/git/tutorials/what-is-git> (besucht am 11.12.2021).
- [GGP21] Julia Gerstlacher, Iris Groher und Reinhold Plösch. “Green und Sustainable Software im Kontext von Software Qualitätsmodellen”. In: *HMD Praxis der Wirtschaftsinformatik* (2021), S. 1–16.
- [GKN21] Achim Guldner, Sandro Kreten und Stefan Naumann. “Exploration and systematic assessment of the resource efficiency of Machine Learning”. In: *INFORMATIK 2021* (2021).
- [ION21] IONOS Digitalguide. *Die besten Continuous-Integration-Tools im Überblick*. 2021. URL: <https://www.ionos.de/digitalguide/websites/web-entwicklung/continuous-integration-tools/> (besucht am 11.12.2021).
- [LLC21] Google LLC. *Angular*. 2021. URL: <https://angular.io/> (besucht am 02.03.2022).

- [Sen21] Lea Senn. “Facebook-Störung kostet Zuckerberg 6 Milliarden – der Ausfall in Zahlen”. In: *watson* (2021). URL: <https://www.watson.ch/digital/daten/308705719-facebook-stoerung-kostet-zuckerberg-6-milliarden-der-ausfall-in-zahlen> (besucht am 24.01.2022).
- [Add22] AddTeq. *jenkins_pipeline.png (PNG-Grafik, 920 × 295 Pixel)*. 2022. (Besucht am 05.03.2022).
- [Aka22] Bitkom Akademie. *Bitkom Akademie — Wir qualifizieren die Digitalwirtschaft*. 2022. URL: <https://www.bitkom-akademie.de/> (besucht am 22.02.2022).
- [Atl22] Atlassian. *Die unterschiedlichen Arten von Softwaretests — Atlassian*. 2022. URL: <https://www.atlassian.com/de/continuous-delivery/software-testing/types-of-software-testing> (besucht am 01.02.2022).
- [Blu22] Blue Angel. *Blue Angel, software products, resources and energy efficient, transparent interfaces — Blauer Engel*. 2022. URL: <https://www.blauer-engel.de/en/productworld/resources-and-energy-efficient-software-products> (besucht am 01.02.2022).
- [Bun22a] Bundesumweltministeriums. *Green IT*. 2022. URL: <https://www.bmu.de/themen/nachhaltigkeit-digitalisierung/konsum-und-produkte/produktbereiche/green-it> (besucht am 01.02.2022).
- [Bun22b] Bundesumweltministeriums. *Green-IT-Initiative des Bundes*. 2022. URL: <https://www.bmu.de/themen/nachhaltigkeit-digitalisierung/digitalisierung/green-it-initiative> (besucht am 01.02.2022).
- [Car22] Care.de. *Klimawandel: Was ist Klimawandel & wie entsteht er?* 2022. URL: <https://www.care.de/schwerpunkte/klimakrise/klimawandel/> (besucht am 22.02.2022).
- [Dev22] DevIQ. *Don't Repeat Yourself — DevIQ*. 2022. URL: <https://deviq.com/principles/dont-repeat-yourself> (besucht am 05.03.2022).
- [Exp22] Express. *Express - Node.js-Framework von Webanwendungen*. 2022. URL: <https://expressjs.com/de/> (besucht am 02.03.2022).
- [Fre22] Fresenius. *Dialyse - Fresenius SE & Co. KGaA*. 2022. URL: <https://www.fresenius.de/dialyse> (besucht am 07.02.2022).
- [Hac22] Hackr.io. *Types of Software Testing: Different Testing Types with Details*. 2022. URL: <https://hackr.io/blog/types-of-software-testing> (besucht am 05.03.2022).
- [IDC22] IDC: The premier global market intelligence company. *IDC: The premier global market intelligence firm*. 2022. URL: <https://www.idc.com/> (besucht am 07.03.2022).
- [Inf22] Informatik Aktuell. *Software und Nachhaltigkeit – Wie passt das zusammen?* 2022. URL: <https://www.informatik-aktuell.de/management-und-recht/digitalisierung/software-und-nachhaltigkeit-wie-passt-das-zusammen.html> (besucht am 01.02.2022).

- [ION22] IONOS Digitalguide. *Das Wasserfallmodell*. 2022. URL: <https://www.ionos.de/digitalguide/websites/web-entwicklung/wasserfallmodell/> (besucht am 02.02.2022).
- [ISO22a] ISO. *ISO 25000 STANDARDS*. 2022. URL: <https://iso25000.com/index.php/en/iso-25000-standards> (besucht am 09.03.2022).
- [ISO22b] ISO. *ISO/IEC 14756:1999*. 2022. URL: <https://www.iso.org/standard/25492.html> (besucht am 16.02.2022).
- [Pae22] Paessler. *Was ist SNMP? - Definition und Details*. 2022. URL: <https://www.paessler.com/de/it-explained/snmp> (besucht am 15.03.2022).
- [Qen22] Qentelli. *An Introduction to Continuous Integration — Qentelli*. 2022. URL: <https://www.qentelli.com/thought-leadership/insights/continuous-integration> (besucht am 21.03.2022).
- [Sci22a] Sciences, Hochschule Trier - Trier University of Applied. *ReFoPlan 2020*. 2022. URL: <https://www.umwelt-campus.de/green-software-engineering/refoplan-20> (besucht am 16.02.2022).
- [Sci22b] Sciences, Hochschule Trier - Trier University of Applied. *UFOPLAN-SSD 2015*. 2022. URL: <https://www.umwelt-campus.de/forschung/projekte/green-software-engineering/projekte/ufoplan-ssd> (besucht am 16.02.2022).
- [Sel22] Selenium. *Ecosystem*. 2022. URL: <https://www.selenium.dev/ecosystem/> (besucht am 14.02.2022).
- [Umw22] Umweltbundesamt. *Kohlendioxid-Emissionen*. 2022. URL: <https://www.umweltbundesamt.de/daten/klima/treibhausgas-emissionen-in-deutschland/kohlendioxid-emissionen#kohlendioxid-emissionen-im-vergleich-zu-anderen-treibhausgasen> (besucht am 22.02.2022).
- [Vaa22] Vaadin. *Vaadin - An open platform for building web apps in Java*. 2022. URL: <https://vaadin.com/> (besucht am 02.03.2022).
- [Wag22] Sarah Wagner. *Continuous Integration: Was es ist, warum es wichtig ist und Tools für den Einstieg*. 2022. URL: <https://www.inwt-statistics.de/blog-artikel-lesen/continuous-integration-was-es-ist-warum-es-wichtig-ist-und-tools-fuer-den-einstieg.html> (besucht am 24.01.2022).
- [wat22] watson.ch. *Facebook nimmt Stellung: Das ist die Ursache des Totalausfalls*. 2022. URL: <https://www.watson.ch/digital/international/751763846-facebook-nimmt-stellung-das-ist-die-ursache-des-totalausfalls> (besucht am 24.01.2022).
- [Sma22] SmartBear.com. *Why Should I Run My Selenium Tests in Headless?* 12022. URL: <https://smartbear.com/blog/selenium-tests-headless/> (besucht am 15.02.2022).

Abbildungsverzeichnis

| | | |
|----|---|----|
| 1 | Anteile der Treibhausgase an den Emissionen aus dem Jahr 2020 [Umw22] | 1 |
| 2 | Ziele und Instrumente der Green IT [Bun22b] | 4 |
| 3 | Lebenszyklen von Hardware und Software [Ker+18] | 5 |
| 4 | ISO/IEC 25010 als Qualitätsmodell [Plö20] | 7 |
| 5 | Wasserfallmodell [ION22] | 8 |
| 6 | Scrum [Shi19] | 8 |
| 7 | Arten von Softwaretests (eigene Darstellung, basierend auf [Hac22], [Tec20]) | 9 |
| 8 | Ablauf von Continuous Integration (eigene Darstellung, basierend auf [Add22]) | 10 |
| 9 | Übersicht über die Jenkins Oberfläche | 12 |
| 10 | Übersicht über ein einzelnes Jenkins Projekt | 12 |
| 11 | Erweiterter Ablauf von Continuous Integration (eigene Darstellung, basierend auf [Add22]) | 17 |
| 12 | Efficient-Schichtansicht | 19 |
| 13 | Efficient-Jahresansicht | 19 |
| 14 | Efficient-Übersicht | 20 |
| 15 | Obelisk-Assetansicht | 21 |
| 16 | Messaufbau zur Ermittlung des Energie- und Ressourcenverbrauchs [GKN21] | 22 |
| 17 | Beispielhafter Ablauf des Nutzungsszenarios [Inf22] | 23 |
| 18 | Beispielhafte Visualisierung der CPU-Inanspruchnahme | 27 |
| 19 | Allgemeiner Ablauf des Messkonzepts (eigene Darstellung, basierend auf [GKN21]) | 28 |
| 20 | Ablauf des automatisierten Messverfahrens (eigene Darstellung) | 34 |
| 21 | CPU-Inanspruchnahme (Efficient - Baseline) | 38 |
| 22 | CPU-Inanspruchnahme (Obelisk - Baseline) | 38 |
| 23 | Gelesene/geschriebene Bytes der Festplatte (Efficient - Baseline) | 39 |
| 24 | Gelesene/geschriebene Bytes der Festplatte (Obelisk - Baseline) | 39 |
| 25 | RAM-Inanspruchnahme (Efficient - Baseline) | 39 |
| 26 | RAM-Inanspruchnahme (Obelisk - Baseline) | 39 |
| 27 | Empfänger/übertragener Netzwerkdatenverkehr (Efficient - Baseline) | 40 |
| 28 | Empfänger/übertragener Netzwerkdatenverkehr (Obelisk - Baseline) | 40 |
| 29 | CPU-Inanspruchnahme (Efficient - Leerlauf) | 41 |
| 30 | Gelesene/geschriebene Bytes der Festplatte (Efficient - Leerlauf) | 41 |
| 31 | RAM-Inanspruchnahme (Efficient - Leerlauf) | 42 |
| 32 | Empfänger/übertragener Netzwerkdatenverkehr (Efficient - Leerlauf) | 42 |
| 33 | CPU-Inanspruchnahme (Obelisk - Leerlauf) | 43 |
| 34 | Gelesene/geschriebene Bytes der Festplatte (Obelisk - Leerlauf) | 43 |
| 35 | RAM-Inanspruchnahme (Obelisk - Leerlauf) | 44 |
| 36 | Empfänger/übertragener Netzwerkdatenverkehr (Obelisk - Leerlauf) | 44 |
| 37 | CPU-Inanspruchnahme (Efficient - Zustand 1) | 45 |
| 38 | CPU-Inanspruchnahme (Efficient - Zustand 2) | 45 |
| 39 | Gelesene/geschriebene Bytes der Festplatte (Efficient - Zustand 1) | 46 |
| 40 | Gelesene/geschriebene Bytes der Festplatte (Efficient - Zustand 2) | 46 |
| 41 | RAM-Inanspruchnahme (Efficient - Zustand 1) | 47 |
| 42 | RAM-Inanspruchnahme (Efficient - Zustand 2) | 47 |
| 43 | Empfänger/übertragener Netzwerkdatenverkehr (Efficient - Zustand 1) | 47 |
| 44 | Empfänger/übertragener Netzwerkdatenverkehr (Efficient - Zustand 2) | 47 |
| 45 | CPU-Inanspruchnahme (Obelisk - Zustand 1) | 49 |

| | | |
|----|---|----|
| 46 | CPU-Inanspruchnahme (Obelisk - Zustand 2) | 49 |
| 47 | Gelesene/geschriebene Bytes der Festplatte (Obelisk - Zustand 1) | 50 |
| 48 | Gelesene/geschriebene Bytes der Festplatte (Obelisk - Zustand 2) | 50 |
| 49 | RAM-Inanspruchnahme (Obelisk - Zustand 1) | 51 |
| 50 | RAM-Inanspruchnahme (Obelisk - Zustand 2) | 51 |
| 51 | Empfangener/übertragener Netzwerkdatenverkehr (Obelisk - Zustand 1) | 52 |
| 52 | Empfangener/übertragener Netzwerkdatenverkehr (Obelisk - Zustand 2) | 52 |

Tabellenverzeichnis

| | | |
|----|--|----|
| 1 | Vergleich von CI-Tools | 11 |
| 2 | Grüne und nachhaltige Produktqualitätskriterien [GGP21] | 15 |
| 3 | Liste der Efficient REST-API-Routen | 20 |
| 4 | Software-Stack der Baseline | 24 |
| 5 | Hardware-Spezifikation des CI-Servers | 25 |
| 6 | Software-Stack des CI-Servers | 26 |
| 7 | Relevante Collectl-Felder | 26 |
| 8 | Übersicht über die Automatisierungsskripte | 30 |
| 9 | Definition verschiedener Anwendungszustände | 32 |
| 10 | Baseline Ergebnisse der Anwendungen Efficient und Obelisk | 38 |
| 11 | Mittelwerte der Baseline, Leerlauf und Differenz (Efficient) | 41 |
| 12 | Mittelwerte der Baseline, Leerlauf und Differenz (Obelisk) | 42 |
| 13 | Nutzungsszenario Ergebnisse (Efficient) | 45 |
| 14 | Nutzungsszenario Ergebnisse ohne Einfluss der Baseline (Efficient) | 48 |
| 15 | Nutzungsszenario Ergebnisse (Obelisk) | 48 |
| 16 | Nutzungsszenario Ergebnisse ohne Einfluss der Baseline (Obelisk) | 52 |

Abkürzungsverzeichnis

| | |
|-------------|---|
| AD | Active Directory |
| CI | Continuous Integration |
| DAE | Datensammlung und -analyse |
| DRY | Don't repeat yourself |
| DOM | Document Object Model |
| IKT | Informations- und Kommunikationstechnik |
| ISS | Institut für Softwaresysteme |
| NPM | Node Package Manager |
| OSS | Open Source Software |
| PM | Power Meter (Leistungsmessgerät) |
| PQ | Produktqualität |
| SNMP | Simple Network Monitoring Protocol |
| SUT | System Under Test |
| WG | Workload Generator |
| WOL | Wake on LAN |

Anlagen

Anlage 1: Experteninterview mit Hendrik Schneider

Datum: 03.03.2022

Name Interviewpartner: Hendrik Schneider

Funktion/ Berufsbezeichnung: Senior Manager Manufacturing Execution System (MES)

1. Für welche Bereiche sind Sie und ihr Team innerhalb des Unternehmens verantwortlich?

Meine Gruppe ist bei der Fresenius Digital Technology verantwortlich für die Eigenentwicklung sam.ware. Dies ist ein Manufacturing Execution System am Standort FME St. Wendel, welches in allen Produktionsbereichen in unterschiedlicher Ausbaustufe zum Einsatz kommt. Das beinhaltet den Betrieb, Support und die Weiterentwicklung der Applikationen.

2. Welche Art von Software entwickeln Sie hauptsächlich?

Unser Fokus sind Java Web- und Client-Applikationen, sowie Webservices und Schnittstellen zu ERP-, Maschinen- und Historian-Systemen.

3. Wie sieht die Qualitätskontrolle bei Ihnen aus?

Unsere Applikationen werden im GxP-relevanten Umfeld eingesetzt und werden daher mit dem hierzu definierten Vorgehen geprüft und validiert. Aktuell setzen wir dort auf das klassische Wasserfallmodell. Somit werden die Applikationen von Seiten der Entwicklung per OQ, also Operational Qualifikation, und Seiten Kunde PQ, d.h. Performance Qualification, geprüft, validiert und abgenommen. Des Weiteren wird unsere Qualität des Quellcode per Design specification und der Applikation SonarQube validiert und geprüft. Zusätzlich dazu wird zur Qualitätsprüfung noch die Installation qualification benötigt, die einen validierten bzw. geprüften Rollout unterstützt und den Betrieb der Applikationen.

4. Welche Art von Softwaretests nutzen Sie und aus welchem Grund?

Aktuell setzen wir auf einen hohen Anteil an manuellen, also Blackbox-Tests, da diese aus Validierungsdokumentationsgründen realisiert und dokumentiert werden müssen. Bei automatischen Softwaretests müsste das Vorgehen zuerst validiert werden. Anschließend kann dieses Vorgehen auch später verwendet werden.

5. Verwenden Sie einen Continuous Integration Server, der den Entwicklungsprozess automatisiert und falls ja, welches Tool nutzen Sie dafür?

Wir haben unsere Landschaft im Bereich CI/CD mit GitLab abgebildet. Dieses Tool wird noch mit Ansible Playbook erweitert.

6. Setzen Sie auf Nightly Builds, oder gibt es ein anderes Verfahren?

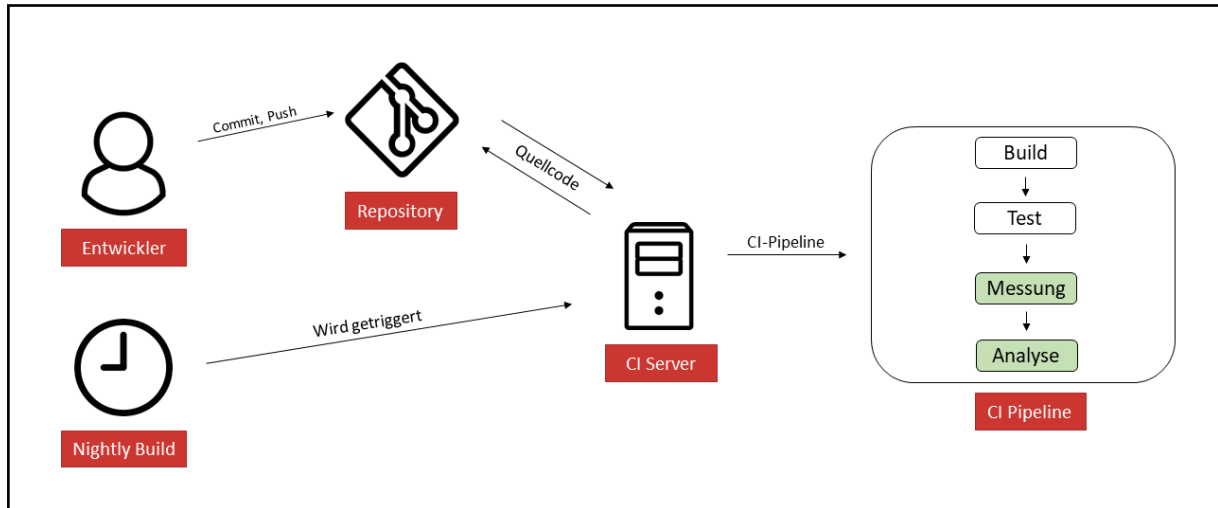
Wir haben keine Nightly Builds im Einsatz. Stattdessen werden die Teilprojekte nach Commits in GitLab automatisch gebaut.

7. Achten Sie während des Softwareentwicklungsprozesses auf Nachhaltigkeit in Bezug auf Ihre Software?

Aktiv wird dies nicht getestet bzw. berücksichtigt. Natürlich verfolgen wir den Don't repeat yourself- Ansatz, damit Quellcode modular implementiert wird und auch keine Redundanzen existieren. Eine allgemeine Entwicklungs-Guideline existiert bei unserer Gruppe, welche aktuell aber keinen Bereich der Green IT inkludiert. Dies bedeutet nicht, dass wir nicht auf die Ver-

schwendung von Ressourcen achten. Speicherplatz, Systemauslastungen und Infrastrukturen sind immer ein relevanter Punkt bei der Realisierung von Anforderungen.

8. Wie offen wären Sie für dieses abgebildete automatisierte Messkonzept, welches während des Softwareentwicklungsprozesses Unterstützung im Bezug auf nachhaltige Software bietet?



Ich finde den Ansatz eines automatisierten Konzepts sehr spannend und innovativ. Gerne würden wir hier auch neue Perspektiven berücksichtigen und in unserem Tagesgeschäft mit einfließen lassen.