HTWK Hochschule für Technik, Wirtschaft und Kultur Leipzig

IM Fakultät Informatik und Medien

Das Potential von WebAssembly für die Entwicklung von Webanwendungen

BACHELORARBEIT

Zur Erlangung des akademischen Grades „Bachelor of Engineering (B.Eng.)“
im Studiengang Medientechnik der Fakultät Informatik und Medien der HTWK Leipzig

Vorgelegt von:

Samuel Schnarr

Geboren am 07. Oktober 1997 in Friedrichroda

Erstkorrektor:

Prof. Dr.-Ing Jörg Bleymehl

Zweitkorrektor:

Prof. Dr.-Ing. Robert Müller

29. Mai 2023

Abstract

WebAssembly als Kompilierungsziel bietet die Möglichkeit, nativen Code im Browser auszuführen. JavaScript ist seither nicht mehr die alleinige Programmiersprache im Web. Darüber hinaus wird WebAssembly zunehmend auch im Serverbereich eingesetzt.

Diese Arbeit gibt einen Überblick über die Grundlagen und stellt den praktischen Nutzen der Technologie in der Webentwicklung dar. Dazu wurde eine Beispielanwendung entwickelt, die WebAssembly zur Bildverarbeitung einsetzt. Im Vergleich zu einer serverseitig umgesetzten Alternativanwendung wird das Potential von WebAssembly aufgezeigt. Die Ausführungsgeschwindigkeit wird zudem mit einer nativen Anwendung verglichen.

Die durchgeführten Tests zeigen, dass der Einsatz von WebAssembly eine sinnvolle Alternative zur Bewältigung serverseitiger Aufgaben traditioneller Anwendungen darstellt. Bei der Verwendung von WebAssembly ergeben sich in den Untersuchungen signifikant schnellere Laufzeiten von bis zu 8-facher Geschwindigkeit.

WebAssembly birgt ein enormes Zukunftspotenzial, die Internetlandschaft grundlegend zu transformieren und Entwicklern eine Vielzahl neuer Möglichkeiten zu eröffnen.

Inhaltsverzeichnis

Abstract

Abkürzungsverzeichnis

1. Einleitung **1**

2. Geschichte **4**

2.1. Vorgänger-Technologien ----4

2.1.1. Plug-in Basierte Systeme ----4

2.1.2. Browserintegrierte Systeme ----6

2.2. Evolution von WebAssembly ----7

3. Anwendungsbereiche **9**

3.1. Im Browser-Kontext ----9

3.2. Außerhalb des Browser-Kontexts ----10

4. Konzeptionelle Hintergründe **12**

4.1. Grundlagen von JavaScript ----12

4.1.1. Merkmale und Prinzipien ----12

4.1.2. Funktionsweise von JavaScript im Browser ----13

4.2. Just-In-Time Kompilierung (JIT) ----14

5. Grundlagen von WebAssembly **16**

5.1. Funktionsweise ----16

5.1.1. Erstellung der WebAssembly-Module in der Entwicklungsumgebung ----16

5.1.2. Laden der WebAssembly-Module im Browser ----18

5.1.3. Ausführung der WebAssembly-Module in der Zielumgebung ----19

5.1.4. Geschwindigkeitsvergleich von WebAssembly und JavaScript im Browser ----20

5.2. Vor- und Nachteile ----20

5.3. Momentaner Entwicklungsstand ----22

6. Konzeption einer Beispielanwendung 24

6.1. Anforderungen und Voraussetzungen ----24

6.2. Entwicklungsvorgang ----25

6.2.1. Beispielanwendung in WebAssembly ----25

6.2.2. Vergleichsanwendung in Node.js ----27

6.3. Durchführung der Messreihen ----30

6.3.1. Voraussetzungen ----30

6.3.2. Testumgebungen ----31

6.3.3. Vorgehensweise ----33

7. Ergebnisanalyse nach Einflussfaktoren 34

7.1. Implementierung----34

7.2. Umgebung ----36

7.3. Hardware ----38

8. Diskussion 43

8.1. Methodenkritik ----43

8.2. Verbesserungsvorschläge ----44

9. Zusammenfassung und Ausblick 45

9.1. Erkenntnisse und Fazit ----45

9.2. Vorschläge für zukünftige Arbeiten----46

9.3. Ausblick in die Zukunft ----46

Literaturverzeichnis

Selbstständigkeitserklärung

Danksagung

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and Xml
AOT	Ahead-Of-Time Compiler
API	Application Programming Interface
CLI	Command Line Interface
CPU	Central Processing Unit
DOM	Document Object Model
GC	Garbage Collection
HTWK	Hochschule für Technik, Wirtschaft und Kultur Leipzig
IoT	Internet of Things
JIT	Just-In-Time Compiler
JPG	Bildformat entwickelt von der Joint Photographic (Experts) Group
JS	JavaScript
Mbit/s	Megabit pro Sekunde
PNG	verlustfreies Bildformat, Portable Network Graphics
SPA	Single Page Applications
UI	User Interface
UX	User Experience
WASI	WebAssembly System Interface
Wasm	WebAssembly
W3C	World Wide Web Consortium

1. Einleitung

Seit etwas mehr als fünf Jahren, genauer gesagt seit November 2017, sind die vier großen Browser (Chrome, Firefox, Safari und Edge) in der Lage, neben den bestehenden Websprachen HTML, CSS und JavaScript auch das neu entwickelte binäre Datenformat WebAssembly zu interpretieren (McConnell, 2017). WebAssembly (kurz Wasm) ermöglicht die Ausführung komplexer und hochleistungsfähiger Anwendungen im Web in nahezu nativer Geschwindigkeit, „but it does not make any Web-specific assumptions or provide Web-specific features, so it can be employed in other environments as well“ (WebAssembly Community Group, 2023).

WebAssembly folgt der Grundidee, dass eine beliebige Programmiersprache in eine Zielumgebung kompiliert und daraufhin beispielsweise in einem Webbrowser ausgeführt werden kann. Die Attraktivität liegt dabei sowohl in der Ausführungsgeschwindigkeit und Portabilität als auch im Sicherheitsaspekt, da die WebAssembly-Module in einer geschützten Sandbox-Umgebung laufen. Die Anwendungsfälle sind sehr vielfältig und reichen von rechenintensiver Bild- und Videoverarbeitung bis hin zu Gaming, maschinellem Lernen, künstlicher Intelligenz oder kryptographischen Berechnungen.

Die Einführung dieser Technologie ermöglicht es (Web-)Entwicklern, bestehende Programme, die beispielsweise in C oder C++ geschrieben wurden, nach WebAssembly zu kompilieren und diese dann im Webkontext zu verwenden, ohne dass eine Portierung nach JavaScript mehr notwendig ist. Darüber hinaus ist die Entwicklung ganzer Anwendungen in mittlerweile über 44 verschiedenen Sprachen (C, C++, C#, Rust, Python, Go, Java, PHP,...) (Akinyemi, 2023) für das Web möglich, indem diese anschließend in WebAssembly-Module kompiliert werden. Es sind keine zusätzlichen Plug-ins oder Software zur Ausführung dieser im Browser erforderlich.

Interessanterweise wird WebAssembly jedoch längst nicht mehr nur als zusätzliche Technologie neben JavaScript im Browser verwendet, sondern findet verstärkt auch außerhalb der Browser-Umgebung Anwendung. Viele namhafte Unternehmen wie z.B. Amazon, Shopify oder Disney+ verwenden WebAssembly auf dem Server als Backend-Technologie zur Bereitstellung einer Vielzahl von Diensten (Carey, 2022). Die Aussage von Docker-Mitbegründer Solomon Hykes verdeutlicht das Ausmaß und die Bedeutung dieses Wandels: „If WASM+WASI existed in 2008, we wouldn't have needed to created Docker [1]. That's how important it is. WebAssembly on the server is the future of computing“ (Hykes, 2019).

1 Eine Plattform zur Entwicklung, Bereitstellung & Ausführung von Anwendungen durch Containervirtualisierung. (Docker, 2023)

In Anbetracht dieser Entwicklungen und Veränderungen stellt sich die Frage, welche Bedeutung und welches Potential WebAssembly weiterhin im Browser haben wird. Seit der Implementierung von WebAssembly und der offiziellen Empfehlung des World Wide Web Consortiums im Jahr 2019 als Webstandard sind schon einige Jahre vergangen (W3C, 2019). Insbesondere in speziellen Anwendungsfällen findet WebAssembly inzwischen Einsatz, aber man kann bei weitem nicht sagen, dass diese vielversprechende Technologie mittlerweile zum Standard geworden wäre. Oft bleibt die Frage des praktischen Nutzens für die Entwicklung von Webanwendungen noch ungeklärt.

Diese Bachelorarbeit gibt einen Überblick über die Hintergründe und Funktionsweisen von WebAssembly und untersucht dessen Einsatzmöglichkeiten bei der Entwicklung von Webanwendungen. Dabei steht vor allem der praktische Nutzen für den Endnutzer im Vordergrund. Zwei einfache Webanwendungen zur Bildbearbeitung mit und ohne WebAssembly werden im praktischen Teil der Arbeit miteinander verglichen und auf ihre Geschwindigkeit bei unterschiedlichen Bildgrößen in verschiedenen Umgebungen getestet.

Bevor auf die theoretischen Hintergründe des Themas eingegangen wird, soll der strukturelle Aufbau der Arbeit kurz thematisiert werden. Die Kapitel können in zwei große Hauptteile untergliedert werden: zum einen in den Theorieteil, der zur Vermittlung von grundlegenden Inhalten über WebAssembly gedacht ist; und zum anderen in den Praxisteil, in dem an einem Praxisbeispiel die Verwendungsmöglichkeiten und das Potential der Webtechnologie aufgezeigt werden sollen.

Kapitel 2 widmet sich der Geschichte von WebAssembly, um die historische Entwicklung verständlicher zu machen. Dabei wird auf einige wichtige Vorläufer eingegangen und ein Vergleich mit WebAssembly gezogen.

Anschließend beschäftigt sich Kapitel 3 des theoretischen Teils mit den Anwendungsgebieten von WebAssembly. Es werden Beispiele gegeben, an welchen Stellen die Technologie bereits eingesetzt wird, beziehungsweise wo die Verwendung sinnvoll erscheint.

Kapitel 4 schafft eine Grundlage, mit welcher die Technologie WebAssembly besser eingeordnet und seine Funktionsweise tiefer verstanden werden kann. Hierfür wird besonders auf Javascript und dessen Funktionsweise im Browser eingegangen, damit die Notwendigkeit von WebAssembly für die Web-Umgebung nachvollziehbarer wird.

Das letzte Theoriekapitel geht schließlich ausführlich auf die Funktionsweise von WebAssembly ein. Es wird Schritt für Schritt erklärt, wie Programmcode aus einer Vielzahl an Programmiersprachen nach WebAssembly kompiliert, im Browser ausgelesen und in Maschinensprache übertragen werden kann. In diesem Zusammenhang werden auch die Vor- und Nachteile der Nutzung von WebAssembly beleuchtet.

Mit dem nachfolgenden 6. Kapitel wird der Praxisteil der Bachelorarbeit eingeleitet. Vorgehensweise und Methodik von Planung und Entwicklung der Testanwendung, sowie das Testverfahren werden eingehend beschrieben. Die Beispielanwendung, welche die Bildverarbeitungsbibliothek ImageMagick nach WebAssembly portiert, soll einen praktischen Anwendungsfall der Technologie aufzeigen und anhand eines Geschwindigkeitsvergleichs mit alternativen Umgebungen die Nutzbarkeit analysieren.

In Kapitel 7 werden die Ergebnisse der Untersuchung der im vorigen Abschnitt beschriebenen Anwendung gruppiert nach ihren Einflussfaktoren erläutert. Die unter Verwendung von WebAssembly entwickelte Webanwendung wird mit der Alternativanwendung mittels serverseitiger Implementierung der ImageMagick-Bibliothek verglichen. Hierzu werden Diagramme und statistische Mittel zur kritischen Auseinandersetzung herangezogen.

Die Resultate der Anwendungstests werden in Kapitel 8 detailliert diskutiert. Dabei wird die angewandte Methode nochmals kritisch betrachtet und Verbesserungsvorschläge angebracht, wie die Tests hätten optimiert werden können.

Abschließend wird im letzten Kapitel 9 aus allen gesammelten Informationen ein Fazit gezogen, welches versucht, einen guten Überblick über WebAssembly zu schaffen. Die Arbeit endet mit Vorschlägen für die weitere Forschung am Thema und einem Ausblick in die Zukunft.

2. Geschichte

Das Internet unterliegt einem stetigen Wandel. Längst ist es nicht mehr nur ein Ort der statischen Darstellung von Inhalten, sondern bietet dem User eine Vielzahl an Möglichkeiten mit den Inhalten zu interagieren (Fredriksson, 2020). Um dies zu ermöglichen, wurden über die Zeit verschiedenste Technologien entwickelt, die den Grundstein für WebAssembly gelegt haben.

2.1. Vorgänger-Technologien

In den letzten zwei Jahrzehnten hat sich ein sichtbarer Paradigmenwechsel hin zum Browser als Anwendung vollzogen (Taivalsaari et al., 2008). Schon im Jahr 2002 wurde mit der Einführung von Macromedia Flash MX der Begriff der Rich Internet Applications (RIA) geprägt (J. Allaire, 2002). Darunter wurden Anwendungen verstanden, die Funktionalitäten und Eigenschaften von nativen Desktop-Anwendungen in den Webbrowser übertragen und sich durch reichhaltige Interaktionsmöglichkeiten und clientseitige Verarbeitung über asynchrone Serververbindungen auszeichnen (Casteleyn et al., 2014). Moderne Single-Page-Applications (SPA) bauen heutzutage im Grunde genommen auf den RIAs der 2000er auf (Dawelbeit, 2014). Zur Ermöglichung dieser Implementierungen gab und gibt es einen stetigen Bedarf an neuen Technologien. Im Laufe der Zeit haben daher viele Unternehmen versucht, solche Systeme zu entwickeln. WebAssembly baut somit auf einer langen Geschichte von Vorgängern auf, die allesamt dem Ziel nacheiferten, dem Web nie dagewesene Funktionalität und Performance zu verleihen.

2.1.1. Plug-in Basierte Systeme

Mit der Entwicklung der Programmiersprache Java 1995 unter dem Leitsatz „Write Once, Run Anywhere“ konnten erstmals kleine Anwendungen durch die Verwendung von Java-Applets in den Browser gebracht werden (Fredriksson, 2020). Um diese Java-Applets nutzen zu können, musste die JRE (Java-Runtime-Environment) auf dem Client installiert sein, welche ein separates Browser Plug-in zur Ausführung der Applets enthielt (Oracle, 2022). Der in Java geschriebene Code wurde nicht innerhalb des Browsers ausgeführt, sondern innerhalb einer separaten virtuellen Maschine (JVM - Java-Virtual-Machine). Java-Applets waren zu der Zeit vor allem JavaScript in ihrer Geschwindigkeit überlegen und konnten durch die separate Ausführung auf die Hardwarebeschleunigung der Prozessoren zugreifen (Coosner, 2022). Rechenintensive und grafische Anwendungen wurden daher häufig mit Java-Applets realisiert. Der große Vorteil an dieser Technologie ist, dass durch die Plattformunabhängigkeit des Java Codes die gleiche Datei an verschiedenste Clients mit unterschiedlichen Betriebssystemen ausgeliefert werden kann, da sich die JVM um die Interpretation in den jeweilig benötigten Maschinencode küm-

mert (Burd, 2017, S. 36). Leider ist dieses System aber immer daran gebunden, dass zwingend Java auf dem Endgerät installiert sein muss. Genau dies macht den bedeutenden Unterschied zwischen Java-Applets und WebAssembly aus, da WebAssembly fest in den Browser integriert ist und somit keine extra Software benötigt. Im Laufe der Zeit, während JavaScript immer performanter und energieeffizienter wurde, verloren Java-Applets an Wichtigkeit und wurden aus verschiedensten Gründen immer weniger verwendet (Coosner, 2022). In einem White-Paper gab Oracle 2016 schließlich öffentlich bekannt, Java Applets und die dafür notwendigen Technologien ab Version 9 nicht weiter zu unterstützen, da die wichtigsten Browser schon seit 2013 damit begannen hatten, den Java-Support im Browser einzustellen (Oracle Corporation, 2016).

Anderen Plug-in-basierten Technologien erging es in dieser Zeit ähnlich. So stieg beispielsweise die Verbreitung des Flash Players rapide an, bis sie seit Beginn des Zeitalters der Smartphones wieder stetig abnahm (Google, 2023). Ursprünglich in den 90er Jahren als Desktop-Grafiksoftware von FutureWave entwickelt, gewann die Technologie nach der Neuausrichtung als Animations-Tool fürs Web schnell an Popularität. Macromedia nannte den sogenannten FutureSplash Player nach der Übernahme in Flash um, und wiederum ein paar Jahre später wurde dieser von Adobe aufgekauft (Hoffmann, 2017). Flash-Inhalte wurden mit einem separaten Programm entwickelt und dann mit Hilfe eines Plug-Ins - dem Flash Player - im Browser angezeigt. So wurden nicht nur kleine Animationen fürs Web konzipiert, sondern Spiele, aufwendige Interaktionen, Video-Player und ganze Websites. Unter anderem auch durch den Start und Erfolg von YouTube, welches ebenfalls den Flash Player zur Anzeige seiner Videos verwendete, wurde Flash immer bedeutsamer (Hoffmann, 2017). Laut Adobe war Flash im Jahr 2008 die „weltweit am weitesten verbreitete Software-Plattform, [welche] von mehr als 2 Millionen professionellen Anwendern genutzt und auf 99,0 % aller internetfähigen Desktops“ installiert war (Adobe, 2008). Obwohl Flash de facto zum Standard für Desktop-Computer wurde, konnte sich die Technologie jedoch nicht dauerhaft durchsetzen: Mit der Markteinführung des iPhones 2007 und der Smartphone-Ära begann auch der Niedergang von Flash (Hoffmann, 2017). Da auf Apple’s Mobilgeräten Flash nicht integriert wurde, waren Webentwickler immer mehr gezwungen, von der Technologie Abstand zu nehmen, um weiterhin auch mobile Nutzer aller Plattformen mit ihren Websites erreichen zu können. Steve Jobs zeigte in seinem offenen Brief „Thoughts on Flash“ im Jahr 2010 die zahlreichen Probleme von Flash auf. Er kritisierte insbesondere die unzähligen Sicherheitsprobleme der Plattform, die Inkompatibilität mit touch-gesteuerten Geräten, den geschlossenen Standard, und die Tatsache, dass mittlerweile einfachere, energieeffizientere und schlichtweg bessere Lösungen existierten (HTML5, CSS, JavaScript), die Flash obsolet machten (Jobs, 2010). Im Jahr 2017 kündigte Adobe schließlich an, den Support

für Flash bis Ende 2020 endgültig einzustellen. „Adobe strongly recommends all users immediately to uninstall Flash Player to help protect their systems“ (Adobe, 2021).

Flash oder Java-Applets waren nicht die einzigen Technologien, die per Plug-in zusätzliche Funktionalität im Browser ermöglichten. Auch Silverlight von Microsoft, direkter Konkurrent von Flash, war in der Lage, multimediale, grafische Inhalte, Animationen und vieles mehr im Browser darzustellen. Trotz allem erging es auch Silverlight nicht anders als seinen Zeitgenossen. „The only client-side technology which survived was JavaScript, due to the fact it was natively integrated with all major browsers“ (Walker, 2021).

2.1.2. Browserintegrierte Systeme

Der Vorteil der Integration von Technologien in den Browser war auch damals nichts Unbekanntes. Microsoft veröffentlichte bereits 1996 ActiveX als Bestandteil seines Webbrowsers Internet Explorer (Microsoft, 1996). Mittels ActiveX konnten ebenfalls Webanwendungen mit nativen Funktionalitäten realisiert werden, ähnlich wie dies mit Java-Applets möglich war. Der Hauptunterschied lag darin, dass eben kein extra Plug-in installiert werden musste, sondern der Browser direkt in der Lage war, den Code zu interpretieren. Hier lag aber auch das Problem des Ganzen, da diese Anwendungen nur im Internet Explorer und meist auch nur unter Windows ausführbar waren (Sheldon & White, 2021). Zusätzlich dazu liefen ActiveX-Inhalte nicht innerhalb einer geschützten Umgebung, weshalb Webseiten beispielsweise Vollzugriff auf das Dateisystem hatten. Schadsoftware konnte daher die Schwachstellen der Technologie sehr leicht ausnutzen (ebd.). Heutzutage ist ActiveX im Web quasi vollständig von der Bildfläche verschwunden und wird durch das Ende des Internet Explorers offiziell auch nicht mehr unterstützt (Microsoft Edge Team, 2015).

Auch Google versuchte 2011 mit PNaCl (ausgesprochen „pinnacle“) eine Lösung auf den Markt zu bringen, mit welcher nativer Code in C oder C++ im Browser in einer sicheren Umgebung ausgeführt werden kann (Metz, 2011). Hierbei wurde ein Modul in den Google Chrome Browser eingebaut, dass kompilierten Bitcode interpretieren und in Maschinensprache umwandeln konnte (Dodson, 2020). Ein offensichtlicher Nachteil der Technologie war wiederum die Beschränkung auf den Chrome-Browser, da es sich bei PNaCl um eine Google-eigene Technologie handelte. Somit verabschiedete sich auch Google im Jahr 2017 wieder von PNaCl, um sich stattdessen auf WebAssembly und dessen Implementierung zu fokussieren (Google, 2017).

Um dem offensichtlichen Problem entgegenzuwirken, dass die vorangegangenen Technologien beide nur innerhalb des jeweiligen Browsers funktionierten, begann kurze Zeit später ein weiterer großer Browser-Anbieter, sich dem Problem des nativen Codes im Browser zu stellen:

Mozilla entwickelte asm.js, das als Vorreiter von WebAssembly angesehen wird (Wagner, 2020). Wie schon am Namen zu erahnen, besteht asm.js aus einem bestimmten Teil von JavaScript und ist dadurch vollständig abwärtskompatibel zu allen anderen Browsern, auch wenn diese keine asm.js-spezifischen Optimierungen in ihrer JS-Engine enthalten sollten. Es fungiert als „sandboxed virtual machine for memory-unsafe languages like C or C++“ (Herman et al., 2014). Dabei wird der C oder C++ Code vorher mittels eines Transpilers wie etwa Emscripten in optimierten JavaScript-Code (asm.js) übersetzt, der zum Zeitpunkt der Ausführung nicht mehr aufwendig geparst oder JIT-kompiliert (Just-In-Time) werden muss (Resig, 2013). Der optimierte Code enthält beispielsweise statische Typdeklarationen für Variablen, weshalb eine ständige Überwachung und Analyse durch die JavaScript-Engine hinfällig wird. Durch die Ahead-of-Time Kompilierung (AOT) können somit viel Zeit und Ressourcen im Browser eingespart werden. Dennoch musste weiterhin der Umweg über JavaScript gegangen werden, welches trotz der Optimierungen ab einer gewissen Komplexität zum Flaschenhals wurde (Eich, 2015). „Thus, it is natural to view WebAssembly as the next evolutionary step of asm.js (a step many have requested and anticipated)“ (Wagner, 2015).

2.2. Evolution von WebAssembly

Mit PNaCl wurde eine Lösung gefunden, welche starke Leistung und Sicherheit darbot, aber keine Browser-Portabilität. Asm.js ermöglichte diese Browser-Portabilität kombiniert mit dem Aspekt der Sicherheit, aber war durch den Einsatz von JavaScript in seiner Leistung eingeschränkt (Sletten, 2021, S. 14). Es musste also ein Weg gefunden werden, diese beiden Ansätze miteinander zu kombinieren. Das geschah im Juni 2015 als sich Entwickler von Mozilla, Chromium, Microsoft und WebKit zusammenschlossen, um mit WebAssembly dieses Ziel zu verwirklichen (Eich, 2015). Die WebAssembly W3C Community Group wurde als browserübergreifendes Projekt gegründet, um das neue portable Datenformat mit Fokus auf Sicherheit und Performance in alle großen Browser zu bringen. Die Entwicklung einer Erstversion war circa zwei Jahre später final abgeschlossen und wurde offiziell veröffentlicht (Haas et al., 2017). Daraufhin dauerte es nicht lange, bis WebAssembly in den Browsern Firefox, Chrome, Safari und auch Edge integriert war (McConnell, 2017). Viele der Funktionen, welche sich die Arbeitsgruppe rund um das Projekt zum Ziel gesetzt hatte, wurden bereits in die erste Version von WebAssembly integriert. 2019 verkündete das W3C als internationale Organisation für Webstandards offiziell in einer Pressemitteilung: „WebAssembly becomes the fourth language for the Web which allows code to run in the browser“ (W3C, 2019).

Alle im vorherigen Unterkapitel erwähnten Technologien haben es nicht geschafft, sich über die Zeit konstant durchzusetzen. Es wurde keine Adaption und Integration in die Webbrowser erreicht, welche eine Technologie wirklich plattformunabhängig machen würde. In diesem Punkt hat WebAssembly 'gesiegt', da dessen Entwicklung nicht von einem einzelnen Unternehmen mit dem Ziel ausging, seine eigene proprietäre Lösung mit möglichst großem Marktanteil durchzusetzen, sondern als Gemeinschaftsprojekt vieler großer Unternehmen, in der Absicht, einen neuen Webstandard zu schaffen (Haas et al., 2017). WebAssembly wird beständig weiterentwickelt und findet schon lange nicht mehr nur im Web Einsatz. Das Potential der Technologie außerhalb des Browsers ist immens und wird immer weiter vorangetrieben (Sletten, 2021, S. 16). Dieses Thema soll im nächsten Kapitel näher betrachtet werden.

3. Anwendungsbereiche

Wie bereits erwähnt, wurde das Potential von WebAssembly außerhalb des Browsers schon während der Entwicklung erkannt und mitbedacht. Trotz allem stand die Implementierung im Browser anfangs eher im Fokus. Über die letzten Jahre hinweg ist die Begeisterung für den Einsatz außerhalb des Browsers immens gewachsen und WebAssembly ist heute bei weitem nicht mehr nur eine Webtechnologie. In diesem Kapitel soll deutlicher werden, in welchen Anwendungsbereichen WebAssembly praktisch genutzt werden kann. Dies wird an einigen Beispielen festgemacht, wie renommierte Unternehmen von der Technologie Gebrauch machen.

3.1. Im Browser-Kontext

Unter näherer Betrachtung fallen hauptsächlich drei größere Verwendungsmöglichkeiten für WebAssembly ins Auge, wenn die Technologie im Browser verwendet wird. Einerseits kann der gesamte Code in WebAssembly vorliegen, welcher dann die Website in all ihren Einzelheiten zum Laufen bringt. Ein Beispiel hierfür ist die Portierung von AutoCAD von Autodesk. AutoCAD ist ein CAD-Programm zur Erstellung von technischen Zeichnungen und basiert auf einer 35 Jahre alten Codebasis. Dieser C++ Code konnte nach WebAssembly kompiliert werden und ist somit als Webversion viel bequemer und leichter nutzbar (Cheung, 2018). Ein weiteres Beispiel hierfür ist das bekannte Bildbearbeitungsprogramm Photoshop, welches Adobe mittlerweile auch als Webanwendung zur Verfügung gestellt hat. Dies konnte ebenfalls nur durch den Einsatz von WebAssembly umgesetzt werden (Nattestad, 2021).

Ebenso kann WebAssembly auch nur für die Kernanwendungen verwendet werden, wobei die Benutzeroberfläche und deren Funktionalität in JavaScript gefasst ist. Eine Interaktionsebene zwischen der UI in JavaScript und der Kernanwendung im Hintergrund in WebAssembly sorgt dafür, dass Benutzereingaben in WebAssembly verarbeitet werden und anschließend wiederum über JavaScript im Frontend sichtbar werden. Als Beispiele hierfür können die Webversion von Google Earth und die UX/UI Design-Software Figma herangezogen werden. Bei Google Earth handelt es sich, genauso wie bei AutoCAD, um die Portierung eines bereits bestehenden Desktop-Programms ins Web. Für Google Earth wurde genau diese separate Benutzeroberfläche für die Web-Umgebung geschaffen, die die Bedienung im Browser erleichtert und gleichzeitig über WebAssembly weiterhin Zugriff auf die Funktionalität der eigentlichen Hauptanwendung hat (Google, 2019). Figma auf der anderen Seite wurde von Anfang an direkt als Webanwendung entwickelt. Zuerst wurde der C++ Code nach asm.js transpiliert, jedoch sobald WebAssembly 2017 aufkam, setzte Figma auf die neue Technologie und ermöglichte damit eine dreimal schnellere Ladezeit für seine Nutzer (Figma, 2017).

Der dritte Bereich, in dem WebAssembly am häufigsten im Browser Anwendung findet, ist die Wiederverwendung von vorhandenen Codebibliotheken anderer Programmiersprachen, welche nach WebAssembly kompiliert und in einer Webanwendung implementiert werden. Dies ermöglicht oft die Auslagerung von rechenintensiven Aufgaben oder die Erweiterung des Browsers um Funktionalitäten, die ohne WebAssembly nicht vorhanden wären. Beispielhaft kann hierfür die Webanwendung 'Squoosh' genannt werden (www.squoosh.app). Die Webanwendung ermöglicht die Komprimierung von Bildern, wobei verschiedene Einstelloptionen zur Verfügung stehen. In Squoosh sind die Codebibliotheken mehrerer Kompressionsalgorithmen per WebAssembly integriert, die die Komprimierung der Bilder ermöglichen. Auf diese Weise ist kein Hochladen der Bilddateien auf einen Server erforderlich, sondern die Bilddateien können direkt auf dem Endgerät verarbeitet werden (Surma, 2019). In gleicher Weise integriert auch eBay im Assistenten zum Einstellen von Artikeln bei seinem Bar Code Scanner eine Codebibliothek in C, um Barcodes verlässlich über die Handy-Kamera scannen zu können (eBay, 2019).

3.2. Außerhalb des Browser-Kontexts

Da WebAssembly im Browser innerhalb der JavaScript Engine läuft, profitiert es von allen Vorteilen der Engine, wie beispielsweise der Isolierung in der Sandbox-Umgebung oder der Verknüpfung mit dem Hostsystem. Bei der Verwendung von WebAssembly außerhalb eines Browsers sind all diese Anforderungen natürlich nicht gegeben. Aus diesem Grund wurde 2019 das WebAssembly System Interface (abgek. WASI) entwickelt. WASI stellt sicher, dass eine einheitliche Schnittstelle zur Verfügung steht, um WebAssembly-Module über den Browser hinaus in verschiedenen Umgebungen wie etwa auf einem Server auszuführen. „We aimed to create a system interface that will uphold [...] the key principles of WebAssembly — portability and security" (Clark, 2019). WebAssembly ist aufgrund seiner spezifischen Sicherheitsmerkmale, seiner Portabilität und seiner Effizienz ein idealer Kandidat für die Ausführung in einer Serverumgebung. Es benötigt weniger Speicher als Docker oder eine virtuelle Java-Umgebung und hat eine Kaltstartzeit von wenigen bis unter einer Millisekunde (Matei, 2023). Insbesondere im Serverless Computing, bei der Containerisierung oder bei Anwendungsfällen im Bereich des Internet of Things (IoT), wo Ressourcenknappheit ein großer Faktor ist, kann der Einsatz von WebAssembly unter Verwendung des WASI Standards sehr lohnenswert sein. Mittlerweile gibt es einige Runtimes, die den WASI Standard abbilden, so etwa Wasmtime, WAMR, oder Wasmer. Jedoch selbst in bestehenden JavaScript Runtimes für Serverumgebungen wie etwa Node.js oder Deno können WebAssembly-Module verwendet werden (Roadmap - WebAssembly, 2023).

Als Beispiele können hier die Platform-as-a-Service Dienste Fermyon und Cosmonic genannt werden. Die Cloud-Dienste ermöglichen die Entwicklung von Webanwendungen mit WebAssembly durch die Vereinfachung des Bereitstellungsprozesses dieser Anwendungen (Joslyn, 2022). So können Entwickler Anwendungen in beliebiger Programmiersprache schreiben, und diese auf dem Server ausführen. In diesem Bereich wird es sehr wahrscheinlich in der Zukunft die meiste Weiterentwicklung im Bezug auf WebAssembly geben, wenn mehr und mehr Unternehmen die Technologie auch für Ihr Backend einsetzen.

4. Konzeptionelle Hintergründe

Ziel dieses Kapitels ist die Vertiefung der Thematik und ihrer Zusammenhänge. Um die Funktionsweise von WebAssembly zu verstehen, ist es zunächst notwendig genauer zu betrachten, wie JavaScript im Browser ausgeführt und verwendet wird. Viele Konzepte erschließen sich dann, wenn die Eigenheiten und Merkmale von JavaScript verinnerlicht wurden.

4.1. Grundlagen von JavaScript

JavaScript (JS) wurde 1995 von Brendan Eich innerhalb von 10 Tagen für den Netscape-Browser Navigator 2.0 entwickelt, um bestehende HTML-Inhalte in Webseiten manipulieren und anpassen zu können und eine gewisse Interaktivität zu ermöglichen. Mit der Entwicklung sollte eine leicht zu erlernende und zu schreibende Sprache geschaffen werden, aber das Hauptaugenmerk lag nicht darauf, möglichst schnell zu sein. Damals hieß die neue Programmiersprache noch Mocha, wurde dann in LiveScript umbenannt und schließlich in Zusammenarbeit mit Sun Microsystems, den Entwicklern von Java, als JavaScript auf den Markt gebracht (Wirfs-Brock & Eich, 2020, S. 8). Heute hat die Sprache nicht mehr wirklich viel mit Java zu tun. Eine Grundlage für die einheitliche Weiterentwicklung wurde 1997 mit der Standardisierung durch die ECMA International geschaffen (das Akronym stand früher für 'European Computer Manufacturers Association', mittlerweile wird es als Eigenname angesehen; ebd., S. 5). Seit 2015 wird jährlich im Juni eine neue, weiterentwickelte Version von ECMAScript veröffentlicht. Die zum Zeitpunkt dieser Arbeit aktuelle Version ECMAScript 2022 wird in allen modernen Browsern nahezu vollständig unterstützt (Ecma International, 2022).

4.1.1. Merkmale und Prinzipien

Die Skriptsprache Javascript ist eine höhere, objektorientierte Programmiersprache, die zudem dynamisch ist. Das bedeutet, Variablentypen müssen nicht im vornherein definiert werden, sondern werden zum Ausführungszeitpunkt bestimmt (Mozilla, 2023). Zudem ist JS eine interpretierte Sprache und muss dementsprechend nicht im vornherein kompiliert werden, wie dies im Vergleich zu anderen objektorientierten Sprachen wie etwa Java der Fall ist (Burd, 2017, S. 28). Dies ist für Entwickler einerseits vorteilhaft, da der Code einfach und unkompliziert auf die Website gebracht werden kann. Andererseits bedeutet es aber auch, dass der Browser die zeitaufwendige Arbeit des Parsens, Kompilierens und Optimierens übernehmen muss.

Bei WebAssembly hingegen werden die Optimierungsschritte quasi zeitlich vorgelagert, indem sie zum Teil der Entwicklungskette werden. Vor der Auslieferung des Codes an die Clients wird der Quellcode in schnell dekodierbaren, binären Code umgewandelt, alle Variablen werden mit

statischen Typdeklarationen angegeben, und die Speicherzuweisung geschieht manuell. Diese und andere Optimierungsprozesse gewährleisten eine ununterbrochene Ausführung und ermöglichen eine bessere Performance als JavaScript. Darüber hinaus wird die Geschwindigkeit durch den Wegfall von Reoptimierungsschritten vorhersehbarer und konstanter (Clark, 2017).

4.1.2. Funktionsweise von JavaScript im Browser

In dieser Arbeit wird sich nur auf die Ausführung von JavaScript im Browser fokussiert, serverseitige Implementierungen wie etwa Node.js werden hier nicht im Detail mit betrachtet, funktionieren aber im Wesentlichen sehr ähnlich wi der Einsatz im Browser.

Jeder Browser besitzt neben seiner browserinternen Rendering-Engine zur Erzeugung des HTML-DOM mitsamt seiner CSS Eigenschaften eine dedizierte JavaScript-Engine. Diese Engine ist die virtuelle Maschine, in welcher der JavaScript-Code isoliert ausgeführt wird (Osaka & Chrome Developers, 2018). Wenn die Rendering-Engine eine JavaScript-Datei, beziehungsweise inline JS-Code im HTML-Template findet, wird dieser zur JS-Engine gesendet. Dort durchläuft der Code verschiedene Schritte, bevor er final ausgeführt werden kann.

Wie in der Abbildung 1 zu erkennen, muss der Code erst geparst und auf Fehler geprüft werden, bevor ein Strukturmodell des Codes erstellt werden kann. Anhand dieses Abstract Syntax Tree (AST) kann der Interpreter zeilenweise Byte-Code erstellen, welcher später für die Optimierung im Compiler notwendig ist. Moderne Browser verwenden sogenannte Just-In-Time-Compiler,

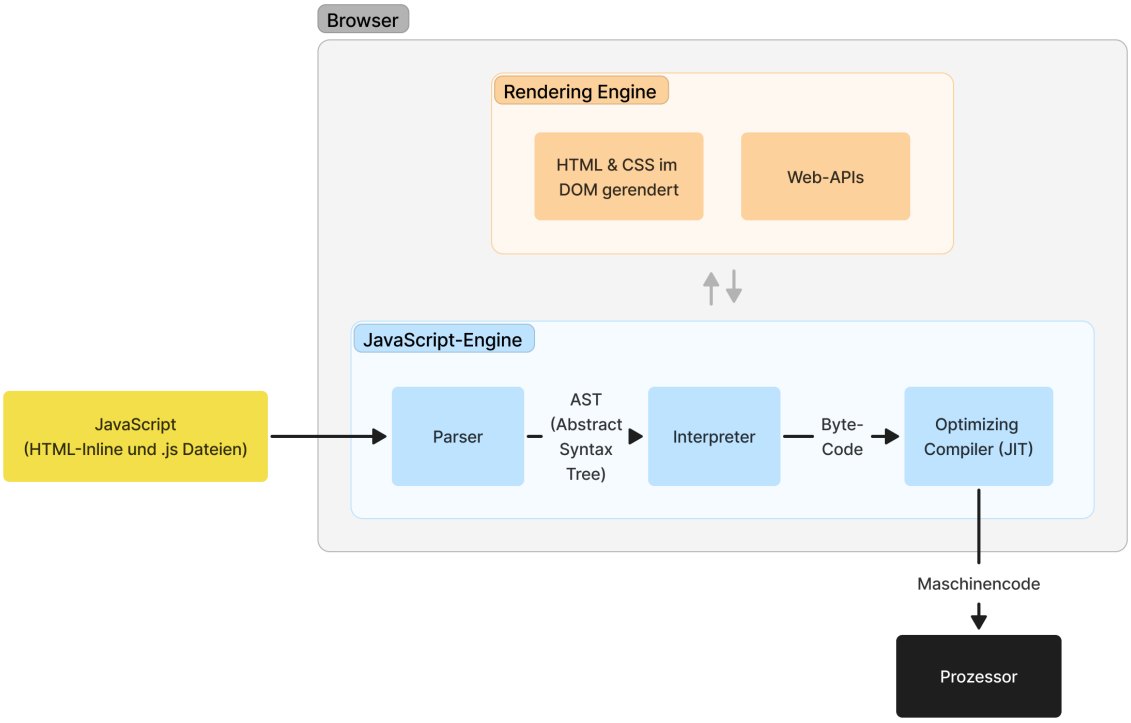

Abbildung 1: JavaScript im Browser, schematische Darstellung (eigene Grafik nach Hallie, 2020)

um die Zeit zwischen Kompilierung und Ausführung zu verkürzen und damit die Leistung zu erhöhen (abgekürzt JIT, siehe nächstes Unterkapitel). Der JIT-Compiler gibt schließlich Maschinencode aus, der vom Prozessor ausgeführt werden kann (Hallie, 2020).

Es zeichnet sich also ab, dass die Geschwindigkeit von JavaScript allein aufgrund der Art und Weise, wie der Code ausgeführt wird, für einige Anwendungen unzureichend ist, insbesondere wenn man an Echtzeitanwendungen wie Spiele mit hoher Prozessorlast denkt. Über die Jahre hinweg wurde zwar viel unternommen, um die Ausführungsgeschwindigkeit von JavaScript immer weiter zu verbessern; anfänglich verfügten die Browser noch nicht über JIT-Compiler, so dass die Leistung im Vergleich zu heutigen Implementierungen um ein Vielfaches schlechter war (Kosaka, 2018). Trotz allem muss der Code auch heute weiterhin die Schritte des Parsens, Interpretierens und Kompilierens durchlaufen, weshalb nie eine Leistung erreicht werden kann, welche mit einer nativen Ausführung vergleichbar wäre (ebd.).

4.2. Just-In-Time Kompilierung (JIT)

„In 2008, a period that people call the performance wars began. Multiple browsers added just-in-time compilers, also called JITs. As JavaScript was running, the JIT could see patterns and make the code run faster based on those patterns. [..] The introduction of these JITs led to an inflection point in the performance of JavaScript. Execution of JS was 10x faster“ (Clark, 2017). Hinter den JIT-Compilern versteckt sich eine ausgeklügelte Technologie, welche das Beste aus zwei Welten kombiniert — Compilern und Interpretern.

Compiler übersetzen Programmiercode normalerweise im Ganzen und geben nach Beendigung der Durchführung einen für die Weiterverarbeitung besser geeigneten Code vollständig aus. Nachteilig daran ist natürlich, dass der Prozess erst weiter abgearbeitet werden kann, wenn der Compiler mit seiner Arbeit fertig ist und den Code in seiner Gesamtheit zurück liefert. Von Vorteil ist, dass durch diese Art der Übersetzung Anpassungen und Verbesserungen am Code vorgenommen werden können, welche das Programm beschleunigen.

Auf der anderen Seite stehen Interpreter, welche den Code Zeile für Zeile abarbeiten und in die jeweilige Zielsprache übersetzen. Dadurch kann ein Programm gleichmäßig ausgeführt werden, Optimierungen am Code sind somit aber nicht möglich und bei sich wiederholenden Anweisungen muss jedes Mal neu übersetzt werden.

JIT-Compiler nutzen die Flexibilität der Interpreter, indem der Code Zeile für Zeile abgearbeitet, und zusätzlich dazu noch überwacht wird. Stellt der JIT-Compiler fest, dass bestimmte Codezeilen mehrfach ausgeführt werden, sendet er deren Funktionen an den Compiler, speichert den kompilierten Code und greift von nun an auf diesen zurück. Diese Codebestandteile wer-

den als ‘warm’ bezeichnet. Falls manche Codezeilen sehr oft ausgeführt werden, erkennt der JIT-Compiler diese Segmente als ‘heiß’ und verarbeitet sie im Compiler erneut, wobei er zusätzliche, lohnenswerte Optimierungen zur Geschwindigkeitsverbesserung vornimmt. Trotzdem muss bei jeder weiteren Ausführung des Codes überprüft werden, ob die optimierte Version des Codes noch weiterhin gültig ist und verwendet werden kann. Wenn sich beispielsweise ein Variablentyp innerhalb einer Wiederholungsschleife ändert, kann der gespeicherte Maschinencode nicht weiter genutzt werden und wird verworfen (dieser Schritt wird Deoptimierung genannt). Der JIT-Compiler springt zurück zum normal kompilierten Code beziehungsweise ganz an den Anfang, zum Interpreter, und beginnt von vorn (Clark, 2017).

Leider führen die Versuche, die Kompilierung von JavaScript durch JITs zu beschleunigen, auch oft dazu, dass die Ausführungsgeschwindigkeit unvorhersehbar wird, da Code unterschiedlich optimiert oder verworfen wird. Hinzu kommt auch der Aspekt der Garbage Collection (automatische Speicherbereinigung), die fest in JavaScript integriert ist. Auch diese kann unter Umständen die Ausführungszeit verlangsamen und unberechenbar machen, wenn die JS-Engine zum falschen Zeitpunkt gezwungen ist, Speicher freizugeben (edb.).

5. Grundlagen von WebAssembly

Nachdem ein grundsätzliches Verständnis über das Zusammenwirken von JavaScript-Code und Browser gelegt wurde, kann WebAssembly folgend im Detail betrachtet werden und wie diese Technologie mit der Web-Plattform als solcher interagiert.

Wie der Name schon sagt, ist WebAssembly angelehnt an das Konzept von Assemblersprachen, wobei nicht die direkte Hardwareanknüpfung im Vordergrund steht, sondern die Ausführung in einer virtuellen Maschine. Daher wird oft scherzhaft gesagt, WebAssembly ist „neither web nor assembly“ (Sletten, 2021, S. 4). Nicht wirklich eine Assemblersprache, da es nicht auf eine bestimmte Hardwarearchitektur zugeschnitten ist, und nicht wirklich Web, da WebAssembly noch viele weitere Anwendungsmöglichkeiten außerhalb des Browsers besitzt (siehe 3. Anwendungsbereiche). Durch die Ausführung in einer virtuellen Maschine bleibt die Unabhängigkeit von der Hardware der jeweiligen Zielumgebung gegeben (WebAssembly Community Group, 2023).

Als WebAssembly konzipiert wurde, war die Motivation und der Beweggrund dahinter von Anfang an, eine Ergänzung zum Web-Ökosystem zu schaffen, und nicht etwa, um JavaScript zu ersetzen (Wagner, 2020). Um die WebAssembly-Module (die .wasm Dateien) laden zu können, wird tatsächlich sogar JavaScript-Code, der sogenannte Glue-Code, benötigt. Umso wichtiger ist dementsprechend die Koexistenz beider Technologien.

5.1. Funktionsweise

5.1.1. Erstellung der WebAssembly-Module in der Entwicklungsumgebung

Bevor WebAssembly in seiner Zielumgebung, also beispielsweise im Browser, verwendet werden kann, müssen die .wasm-Dateien generiert werden. Dies geschieht nicht etwa wie bei JavaScript: Hier können .js-Dateien einfach in einem Editorprogramm angefertigt werden, welche dann, nachdem sie auf einem Server abgelegt wurden, direkt von einem Client im Browser heruntergeladen und ausgeführt werden können. Stattdessen müssen die Dateien in das .wasm Format im Vornherein vom Entwickler kompiliert werden, bevor sie auf den Server geladen werden können. Sie enthalten Bytecode im Hexadezimalformat, welcher die binären Anweisungen zur Ausführung in der virtuellen Maschine repräsentiert (Mihajlija, 2019). Da dieser Bytecode nicht per Hand geschrieben wird, gibt es Compiler, die für das Übersetzen vom Quellcode in das vom Browser lesbare Format zuständig sind. So können Programme in verschiedensten Programmiersprachen nach WebAssembly kompiliert, und im Browser verwendet werden. „WebAssembly is designed as a compile target so that code written in languages such

as C++, Rust, and others can now run on the web“ (Gallant, 2019, S. 5). WebAssembly als Kompilierziel (engl. compile target) bedeutet, dass WebAssembly selbst eben nicht dafür entworfen wurde, dass dieser Low-Level-Code direkt vom Entwickler geschrieben wird, sondern dass Tools dazu verwendet werden, Code von höheren Programmiersprachen nach WebAssembly umzuwandeln. Zwar ist es möglich, WebAssembly durch die Verwendung des repräsentativen Textformats .wat selbst von Hand zu schreiben, dieses ist aber eher zum Nachvollziehen des Codes und zur Fehlersuche gedacht (WebAssembly Community Group, 2023). Zudem wird es recht schnell sehr komplex und mühsam, anspruchsvollere Programme in dieser Art und Weise zu entwickeln.

Wenn ein Entwickler also eine Anwendung in der Programmiersprache seiner Wahl entworfen hat und ein entsprechender Compiler nach WebAssembly für diese Sprache verfügbar ist, kann beispielsweise aus einem in C geschriebenen Programm ein WebAssembly-Modul generiert werden. Compiler verwenden normalerweise einen Zwischencode (engl. Immediate Representation [IR]) als zusätzliche Abstraktionsebene, der dazu dient, die Übersetzung des Quellcodes in verschiedene Assemblersprachen für unterschiedliche Prozessorarchitekturen zu erleichtern (Clark, 2017). WebAssembly ist eine der Ausgabesprachen, die im Compiler aus diesem Zwischencode generiert werden.

Praktisch gesehen wird meist eine Toolchain, also eine Sammlung mehrerer Werkzeuge zur Erzeugung der .wasm Dateien verwendet, um den Code während des Kompilierungsvorgangs ebenfalls zu optimieren. Im Falle von C oder C++ gibt es die Emscripten Toolchain, die normalerweise für diese Sprachen verwendet wird (MDN Web Docs, 2023b). Nach der Installation des Softwareentwicklungskits Emscripten auf dem Rechner des Programmierers kann dieser mit einem einfachen Befehl in seinem Terminal den in C geschriebenen Code umwandeln. Die

```
> emcc hello_world.c
```

Toolchain erzeugt daraufhin zwei Dateien, einerseits eine .wasm Datei und andererseits eine zugehörige JavaScript-Datei mit dem Glue-Code, der für den Aufruf der WebAssembly-Datei und ihrer Funktionen verantwortlich ist. In einer HTML-Datei muss der JavaScript-Code referenziert werden, welcher das WebAssembly-Modul dann instanziiert (Emscripten, 2023). Wenn ein Browser die HTML-Datei vom Server empfängt, erkennt dieser das WebAssembly-Modul und kann somit den Programmcode aufrufen.

5.1.2. Laden der WebAssembly-Module im Browser

WebAssembly-Module exportieren und importieren Funktionen in ähnlicher Weise, wie JavaScript-Module angelegt sind. Leider ist es momentan zum Zeitpunkt der Erstellung dieser Arbeit noch nicht möglich, WebAssembly-Module in einem HTML-Dokument genauso einzubinden, wie dies mit JavaScript-Modulen durch ein einfaches Script-Tag möglich ist:

```
<script type='module'>
```

Stattdessen müssen die Module mittels JavaScript Glue-Code eingebunden werden (JS API - WebAssembly, 2023). Das folgende Codebeispiel 1 zeigt, wie der Bytecode in ein ArrayBuffer konvertiert und anschließend durch die Funktion `WebAssembly.instatiate()` sowohl kompiliert als auch instanziiert wird. Anschließend kann auf die Funktionen des WebAssembly-Moduls zugegriffen werden.

Codebeispiel 1: Instanziierung eines WebAssembly-Moduls (MDN Web Docs, 2023a)

```
fetch("module.wasm")
  .then((response) => response.arrayBuffer())
  .then((bytes) => WebAssembly.instantiate(bytes, importObject))
  .then((results) => {
    // Funktionen der WebAssembly-Module aufrufen
  });
```

Diese Art des Ladens der Module kann mittlerweile sogar noch vereinfacht werden, indem die Streaming-Funktionalität der JavaScript-API genutzt wird (siehe Codebeispiel 2). Beim Streaming der WebAssembly-Module kann der Bytecode schon während des Herunterladens kompiliert werden. Es muss also einerseits kein zusätzliches ArrayBuffer erstellt werden und andererseits muss nicht gewartet werden, bis die .wasm-Datei vollständig beim Client vorliegt. Beides führt zu deutlichen Performanceverbesserungen im Vergleich zur weiter oben beschriebenen Methode, insbesondere bei großen Dateien (Clark, 2017).

Codebeispiel 2: Streaming eines WebAssembly-Moduls im Browser (MDN Web Docs, 2023a)

```
WebAssembly.instantiateStreaming(fetch("module.wasm"), importObject)
  .then((results) => {
    // Funktionen der WebAssembly-Module aufrufen
  });
```

Bei Verwendung des Streamings von WebAssembly-Modulen sind Browser außerdem in der Lage, den kompilierten Maschinencode im Cache zwischenzuspeichern und bei einem erneutem Seitenaufruf direkt auszuführen, ohne dass eine erneute Kompilierung erforderlich ist (V8 Project & Google Inc., 2022).

5.1.3. Ausführung der WebAssembly-Module in der Zielumgebung

Der WebAssembly-Bytecode wird innerhalb der bestehenden JavaScript-Engine des Browsers ausgeführt, beziehungsweise in der virtuellen Maschine der jeweiligen Umgebung, wenn WebAssembly nicht in einem Browser, sondern beispielsweise auf einem Server genutzt wird (V8 Project & Google Inc., 2022). Dieser Ort der Ausführung wird als Laufzeitumgebung (engl. Runtime) bezeichnet. Innerhalb der Laufzeitumgebung wird der Bytecode dekodiert und anschließend in die Maschinensprache der jeweiligen Zielumgebung kompiliert, also etwa bei einem modernen Apple MacBook in die Maschinensprache eines ARM-Prozessors. Die Maschinensprachen der verschiedenen Prozessortypen unterscheiden sich zum Teil gravierend. Dieser letzte Schritt ist so grundlegend wichtig, weil dadurch der vorkompilierte, binäre Befehlssatz des WebAssembly-Moduls plattformübergreifend und portabel auf verschiedensten Geräten zum Einsatz gebracht werden kann. Für diese Funktionalität ist die JavaScript-Engine zuständig und dementsprechend kann JavaScript auch portabel auf unterschiedlichen Geräten und Plattformen verwendet werden. Der springende Punkt bei WebAssembly ist jedoch, dass der WebAssembly-Code sozusagen als Zwischencode fungiert, um Programmcode in beliebigen Programmiersprachen ins Web und somit auf alle internetfähigen Geräte zu bringen.

Im Vergleich zu JavaScript-Code muss der Code einer .wasm Datei nicht mehr zeitintensiv geparst und interpretiert werden, sondern kann direkt im Compiler der JavaScript-Engine verarbeitet und anschließend ausgeführt werden, wie in Abbildung 2 zu erkennen ist.

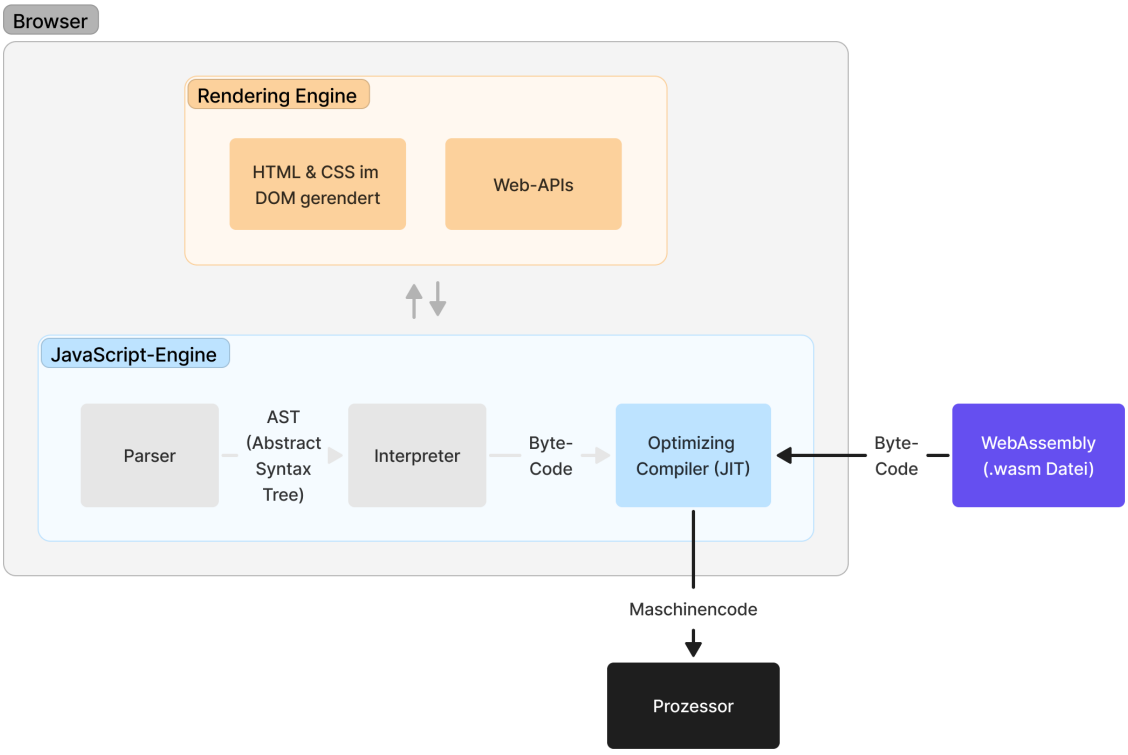

Abbildung 2: WebAssembly im Browser, schematische Darstellung (eigene Grafik nach Clark, 2017)

5.1.4. Geschwindigkeitsvergleich von WebAssembly und JavaScript im Browser

Durch das aus den vorangegangenen Unterkapiteln gewonnene Verständnis wird klar, wie WebAssembly praxisnah im Browser verwendet werden kann und dort verarbeitet wird. Der Unterschied zu JavaScript, besonders im Bezug auf die Geschwindigkeit, soll nun in diesem Abschnitt zusammengefasst werden. Abbildung 3 stellt diesen Unterschied dar.

Zunächst ist festzuhalten, dass WebAssembly im Allgemeinen kürzere Ladezeiten vom Server aufweist als JavaScript, da die Dateien kompakter geschrieben werden können, auch wenn JS-Dateien komprimiert vorliegen (Sletten, 2021, S. 2). Die darauffolgende Dekodierung nimmt weniger Zeit in Anspruch als das Parsen von JavaScript, da der Code nicht zeitaufwendig interpretiert werden muss. Auch die Kompilierungs- und Optimierungsschritte sind mit WebAssembly kürzer, weil der Code näher am Maschinencode liegt und zudem bereits Optimierungsmaßnahmen seitens des Entwicklers vorgenommen wurden. Ein weiterer Faktor, der WebAssembly im Browser potentiell schneller macht, ist der Wegfall der Reoptimierung. In JavaScript müssen wegen der fehlenden Variablentypen Annahmen bei der Optimierung getroffen werden, welche gegebenenfalls invalide werden und somit eine erneute Optimierung erfordern. In WebAssembly kann der Code somit stringenter ausgeführt werden. Im Prozess der finalen Ausführung wird mit WebAssembly normalerweise ebenfalls Zeit eingespart. In JavaScript ist das Potenzial für inkonsistenten Code wesentlich größer, da die höheren Abstraktionsebenen es dem Entwickler erschweren, Code mit konsistenter Performance zu schreiben (Clark, 2017). Die Anweisungen in WebAssembly hingegen sind maschinenfreundlicher und somit wiederum leistungssteigernd. Im Laufe der Ausführung entfällt bei WebAssembly zudem die Garbage Collection (automatische Speicherbereinigung). Diese ist bei JavaScript erforderlich, da der Speicher nicht manuell, wie bei WebAssembly, verwaltet wird.

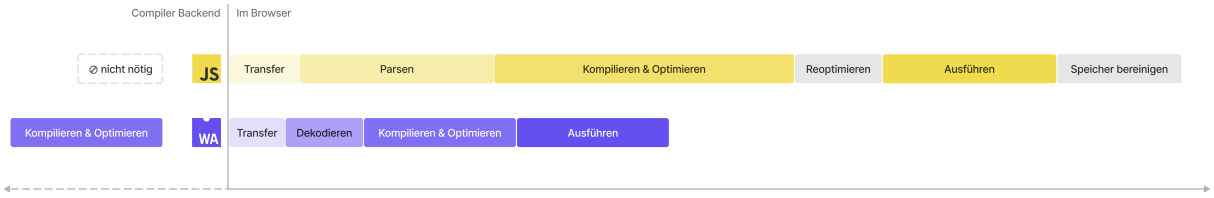

Abbildung 3: Geschwindigkeitsvergleich JavaScript vs. WebAssembly (eigene Grafik nach Clark, 2017)

5.2. Vor- und Nachteile

In den vorangegangenen Kapiteln wurden bereits viele Vorteile von WebAssembly erwähnt. WebAssembly bringt unbestritten eine Menge Vorzüge mit sich, jedoch soll das Thema nicht zu einseitig betrachtet werden. Aus diesem Grund werden im Folgenden sowohl Vor- als auch Nachteile der Technologie gegeneinander abgewogen, um ein differenzierteres Bild zu erhalten.

Eines der zweifellos größten Vorteile von WebAssembly ist die Performance. Zahlreiche Benchmark-Tests zeigen, dass Anwendungen in WebAssembly nur bis zu 20% langsamer sind, als native Anwendungen. „With WebAssembly, multimillion-line code bases can now load in a few seconds and then run at 80 percent of the speed of native programs“ (Wagner, 2017). Nicht nur die hohe Geschwindigkeit selbst macht es attraktiv, sondern auch die Stetigkeit, mit welcher diese Performance erzielt werden kann, was in JavaScript viel schwieriger zu erreichen ist (siehe Just-In-Time Kompilierung). Dadurch wird die Existenz vieler Webanwendungen überhaupt erst ermöglicht.

Im Gegensatz dazu muss in Betracht gezogen werden, dass im Zuge der Entwicklung ein Mehraufwand durch die Kompilierungsschritte entsteht. Der Code wird nicht einfach wie in JavaScript geschrieben und ausgeführt, sondern muss während der Entwicklung immer wieder neu kompiliert werden. Dies kann natürlich auch Debuggingprozesse erschweren.

Kaum zu entkräften ist das Designmerkmal der Sicherheit in WebAssembly. Dieser Hauptfaktor war von Anfang an ein wesentliches Element in der Entwicklung der Technologie. Durch die Validierung und Ausführung in einer speichersicheren Sandbox-Umgebung wird sichergestellt, dass kein unbefugter Zugriff auf Daten oder andere Sicherheitsverletzungen geschehen kann (WebAssembly Community Group, 2023). Dies gilt in allen Umgebungen, egal ob WebAssembly im Browser oder über andere Runtimes auf Servern ausgeführt wird.

Trotz alledem ist es wichtig, nicht außer Acht zu lassen, dass potentieller Schadcode in WebAssembly-Modulen schwieriger im Vornherein zu erkennen ist, da der Code als Bytecode übertragen wird. Eine Untersuchung des Codes auf Ausnutzung bekannter Schwachstellen wird somit um ein Vielfaches komplizierter. Zwar wird der Code in einer geschützten, isolierten Instanz ausgeführt, aber Sicherheitsprüfungen sind schwerer möglich.

Ein weiterer Vorteil von WebAssembly ist die Unabhängigkeit von einer Programmiersprache, was eine Vielzahl von Möglichkeiten eröffnet, Anwendungen ins Web zu bringen. Vor der WebAssembly-Ära wurden Webanwendungen grundsätzlich eher von Webentwicklern programmiert. Mittlerweile können verschiedenste andere Entwickler, welche auf diverse Sprachen spezialisiert sind, durch die Verwendung von Toolchains ebenfalls Webanwendungen auf den Markt bringen (WebAssembly Community Group, 2023). Ebenso ist es möglich, bestehende Anwendungen, die bisher nur nativ zur Verfügung standen, ins Web zu portieren.

Leider ist WebAssembly momentan noch nicht so weit entwickelt, dass es direkt auf den DOM einer Website zugreifen kann. Dies ist in vielerlei Hinsicht natürlich gegenwärtig noch ein Nachteil. Alle Interaktionen, welche zwischen visueller Anzeige und Datenverarbeitung ablau-

fen, müssen zwingend über JavaScript als Zwischenglied abgewickelt werden. Glücklicherweise ist in Planung, dies in der Zukunft zu ermöglichen (WebAssembly Proposals, 2023). Bis dahin müssen die daraus resultierenden Nachteile in Kauf genommen werden.

Final gilt es noch zu erwähnen, dass WebAssembly plattformunabhängig gestaltet ist. Der daraus resultierende Vorteil von WebAssembly gegenüber anderen Technologien liegt auf der Hand. Derselbe Code kann in Browsern eingebettet, in einer eigenständigen virtuellen Maschine ausgeführt oder in andere Umgebungen integriert werden. Dies wird durch die zusätzliche Abstraktionsebene zwischen Prozessor und Anwendung mit der virtuellen Maschine ermöglicht (WebAssembly Community Group, 2023). Besonders vorteilhaft ist, dass im Browser keine Installation von Plug-ins oder anderer Software notwendig ist, sondern dass WebAssembly-Code standardmäßig in allen modernen Browsern unterstützt und ausgeführt wird.

5.3. Momentaner Entwicklungsstand

WebAssembly wird ständig weiter entwickelt. Seit der Veröffentlichung von Version 1.0 des Core-Standards sind viele neue Funktionen hinzugekommen, besonders hinsichtlich technischer Standards. Die Browser implementieren diese neuen Funktionen jeweils schrittweise, wodurch WebAssembly immer leistungsfähiger wird. Seit April 2022 wird an der überarbeiteten Version 2.0 von WebAssembly gearbeitet, welche die neuen Funktionen unter einer Hauptversion vereint (WebAssembly Community Group, 2023). Zu den wichtigsten, in den letzten Jahren hinzugekommenen Funktionen zählen beispielsweise die Unterstützung von SIMD und Multi-Threading (WebAssembly Proposals, 2023).

SIMD (Single Instruction, Multiple Data) beschreibt die Fähigkeit moderner Prozessoren, mehrere Berechnungen innerhalb einer einzigen Anweisung auszuführen. So wird beispielsweise eine Multiplikation innerhalb eines Prozesses für viele Zahlenpaare als Vektor gleichzeitig ausgeführt (WebAssembly Proposals, 2023). Gerade in der Bildverarbeitung, bei physikalischen Simulationen oder beim maschinellen Lernen ist der Einsatz dieser Technik besonders effektiv und wichtig, da hier oft große Datenmengen gleichartig verarbeitet werden. Die Web-Version von Zoom ist beispielsweise von dieser Technologie abhängig, um die Verwendung von virtuellen Videohintergründen zu ermöglichen (Schiemann, 2020).

Das sogenannte Multi-Threading hingegen ermöglicht, dass mehrere Aufgaben eines Programms gleichzeitig ausgeführt werden können. Somit können verschiedenartige Berechnungen parallel abgearbeitet werden, was wiederum zu Geschwindigkeitsverbesserungen führt. Bibliotheken und Programme, die stark auf Multi-Threading basieren, können mittlerweile somit auch in WebAssembly verwendet werden (Eberhardt, 2020, S. 40).

Abgesehen von Verbesserungen in der Datenverarbeitung durch WebAssembly wird zum Zeitpunkt der Erstellung dieser Arbeit versucht, grundlegende Funktionalitäten einer Garbage Collection (GC) in WebAssembly zu integrieren. Derzeit wird in Chrome getestet, wie sich die GC auf die Performance der WebAssembly-Module auswirkt (Chrome Platform Status, 2023). In anderen Browsern ist dies noch nicht integriert. Ursprünglich wurde in WebAssembly keine GC integriert um den gesamten Verarbeitungsprozess zu vereinfachen. Dadurch mussten jedoch Programme in Sprachen, die auf eine GC angewiesen waren (wie etwa das Blazor-Framework mit der Sprache C#), ihre komplette eigene GC nach WebAssembly kompilieren (Eberhardt, 2020, S. 39). Dies hatte große Einbußen sowohl in der Dateigröße als auch in der Effizienz zur Folge. Durch die native Integration einer Garbage Collection in WebAssembly können in Zukunft auch Sprachen wie Scala oder Dart, die stark von einer GC abhängig sind, mit WebAssembly verwendet werden. Die Folgen sind allerdings noch weitreichender: Durch die Integration einer Garbage Collection wäre die Grundlage für WebAssembly-Code geschaffen, mit dem DOM zu interagieren oder auf JavaScript Objekte direkt zuzugreifen. Da die Verwendung der GC optional sein wird, können Sprachen wie C oder Rust weiterhin auf eine lineare, manuelle Speicherverwaltung setzen, um beste Performanceergebnisse zu erzielen (Fermyon Inc., 2022).

6. Konzeption einer Beispielanwendung

Diese Bachelorarbeit soll nicht nur einen theoretischen Einblick in die Hintergründe von WebAssembly geben, sondern auch anhand eines praktischen Beispiels den Nutzen der Technologie aufzeigen. In den folgenden Kapiteln wird somit dargestellt, wie eine einfache Anwendung aus dem Bereich der Bildverarbeitung mittels WebAssembly ins Web übertragen werden kann. Diese wird anschließend auf ihre Performance getestet und vergleichbaren Alternativanwendungen gegenübergestellt, um den realistischen Mehrwert der Verwendung von WebAssembly herauszuarbeiten.

6.1. Anforderungen und Voraussetzungen

Ein geeigneter Anwendungsfall für die Verwendung von WebAssembly ist die Bilddatenverarbeitung (Use Cases - WebAssembly, 2020). Für die oft rechenintensiven Abläufe bei der Verarbeitung einzelner Pixelwerte kann von prozessornahen Merkmalen und Spezifikationen Gebrauch gemacht werden, die mittels JavaScript nicht zur Verfügung stehen. Die Beispielanwendung dieser Arbeit macht sich deshalb die Softwarebibliothek ImageMagick zu Nutze, um einfache Bildmanipulationen direkt im Browser ausführen zu können. ImageMagick wird in der Regel als Command Line Tool, also als Werkzeug für den Einsatz in der Kommandozeile, verwendet. Zusätzlich stellt es auch eine API zur Verfügung, mit der sich die Funktionalität der Software in andere Programme integrieren lässt (ImageMagick Studio LLC, 2023).

Der Grundgedanke und die Motivation, eine Anwendung mit ImageMagick in WebAssembly zu entwickeln, besteht darin, zu zeigen, dass Bildverarbeitung heutzutage nicht mehr zwangsläufig auf einem Server durchgeführt werden muss. Es gibt einige kostenlose Tools im Web, um Bilder zu modifizieren oder beispielsweise zu konvertieren (cloudconvert.com, etc.). Viele dieser Tools verarbeiten die Bilder auf dem Server, indem Software wie etwa ImageMagick oder andere eingesetzt werden. Dies stellt nicht nur in vielen Situationen ein Datenschutzproblem dar, zusätzlich sind viele serverseitige Ressourcen notwendig, um die Verarbeitungen durchzuführen und Geschwindigkeitseinbußen durch langsame Übertragungsgeschwindigkeiten müssen in Kauf genommen werden. All dies kann umgangen werden, wenn die Bildverarbeitung im Browser des Nutzers stattfindet.

Die Webanwendung des Praxisbeispiels verwendet ein vom Nutzer hochgeladenes Bild, um es mit Hilfe eines von ImageMagick bereitgestellten Rotationsalgorithmus um 90° zu drehen. Anschließend wird das gedrehte Bild an den Nutzer als Download wieder zurückgeliefert. Dieser Schritt der Rotation erstellt eine komplett neue Version des gegebenen Bildes, indem allen Pi-

xeln eine neue Position zugeordnet wird. Zuerst wird das Bild dupliziert und skaliert, damit es anschließend um ein bestimmtes Zentrum herum gedreht werden kann. Infolgedessen werden auch die Außenmaße und Seitenverhältnisse des Bildes neu berechnet (ImageMagick Studio LLC, 2015). Natürlich ist die Drehung eines Bildes eine eher banale Bildmanipulation, aber sie zeigt in vereinfachter Form, wie durch die Einbindung einer Codebibliothek in C mit WebAssembly sonst eher schwierig zu realisierende Bildmodifikationen im Browser möglich werden.

Um die Anwendung ausgiebig testen und bewerten zu können, wurde ebenfalls eine Vergleichsanwendung in Node.js entwickelt, die die gleiche Funktionalität abbildet, jedoch serverseitig implementiert ist. Wenn ein Nutzer ein Bild auswählt, wird dieses auf den Server geladen, dort per ImageMagick transformiert und anschließend als Download wieder vom Server an den Client gesendet. Somit wird eine traditionelle Implementierungsweise eines Programms dieser Art als Vergleichsgröße herangezogen.

6.2. Entwicklungsvorgang

Da nicht der gesamte Entwicklungsprozess an dieser Stelle dokumentiert werden kann, werden einige wesentliche Punkte hervorgehoben, um die Funktionalität der entwickelten Beispielanwendungen aufzuzeigen.

6.2.1. Beispielanwendung in WebAssembly

Bei der Entwicklung einer Webanwendung, die ImageMagick per WebAssembly implementiert, ist folglich die Portierung der Codebibliothek nach WebAssembly notwendig. In Anbetracht der Komplexität der Bibliothek wurde jedoch auf die eigenständige Kompilierung nach WebAssembly verzichtet, weil hierfür sonst ein eigenes Interface für alle Funktionen geschaffen werden muss. Aus diesem Grund wurde auf ein bereits bestehendes NPM-Paket namens WASM-Magick zurückgegriffen, welches die ImageMagick-Bibliothek über Emscripten kompiliert und ein Interface per TypeScript zur Verfügung stellt (Lemstra, 2023). Auf diese Weise muss das WebAssembly-Modul nicht noch einmal manuell in JavaScript instanziiert werden, wie unter 5.1 Funktionsweise beschrieben. Stattdessen werden die einzelnen Funktionen der ImageMagick-Bibliothek per TypeScript referenziert und verwendet.

Um die Bibliothek nutzen zu können, ist eine Entwicklungsumgebung erforderlich, die TypeScript unterstützt. Deshalb wurde Vite ausgewählt, um bequem Code in TypeScript verfassen zu können und gleichzeitig von schnellen Neuladezyklen profitieren zu können (Vite, 2023). Vite bietet zusätzlich die Möglichkeit, die fertige Anwendung als Bundle nach JavaScript zu transpilieren, um eine Verwendung auf dem Testserver zu ermöglichen.

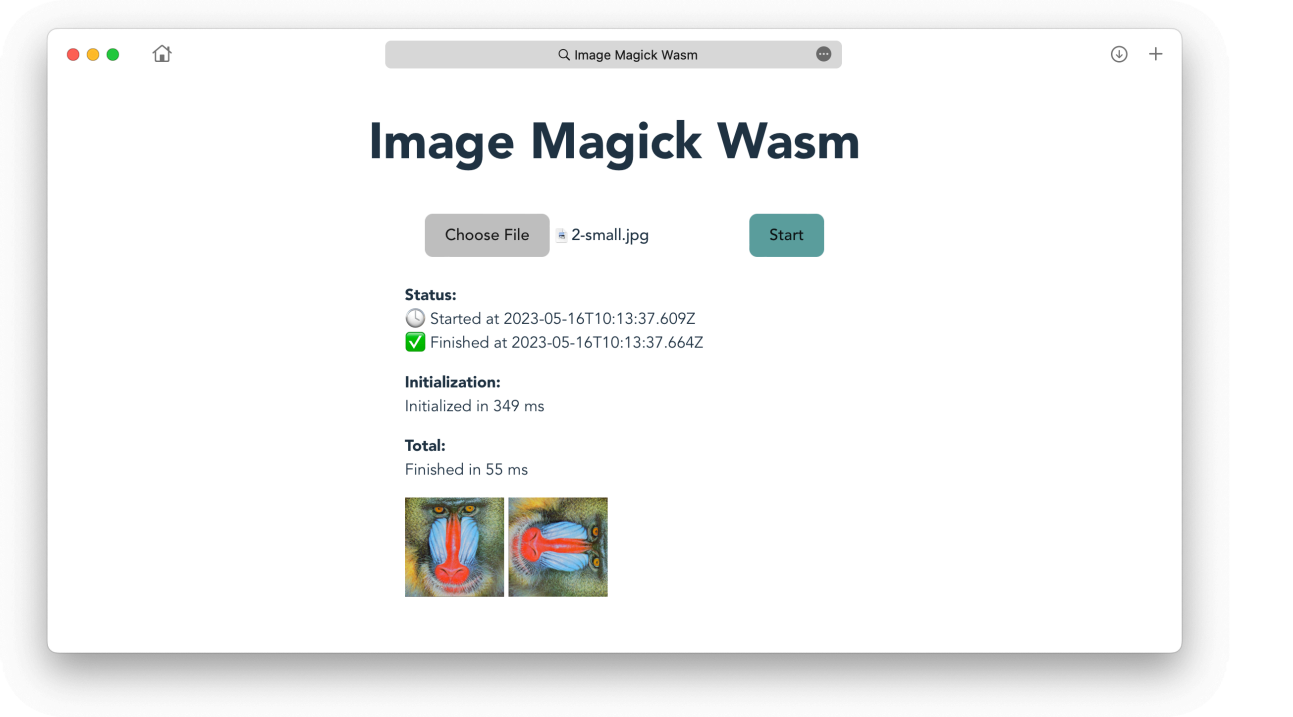

Abbildung 4: Oberfläche der Beispielanwendung unter Verwendung von WebAssembly (eigene Grafik)

Nach dem Import der WASM-Magick-Bibliothek in das Projekt ist eine Initialisierung derselben notwendig (siehe Codebeispiel 3). Dies wurde mit einem einfachen, zur Verfügung stehenden Befehl der Bibliothek umgesetzt, der alle notwendigen Schritte in die Wege leitet.

Codebeispiel 3: Initialisierung der ImageMagick Bibliothek für WebAssembly

```
1  initializeImageMagick().then(async () => {
2    // Nach der Initialisierung
3  });
```

Nachdem der Nutzer ein Bild ausgewählt hat, wird es ausgelesen und dessen Binärdaten in einem ArrayBuffer gespeichert. Um ein Bild mit ImageMagick verarbeiten zu können, ist es erforderlich, dass Bilddaten in Form eines typisierten `Uint8Array` vorliegen. Denn erst so können die einzelnen Pixelwerte ausgelesen und verarbeitet werden.

Anschließend kann das Bild an das WASM-Magick-Modul übergeben und per Rotationsbefehl gedreht werden (siehe Codebeispiel 4). Genau an dieser Stelle kommt WebAssembly ins Spiel. Das TypeScript-Interface leitet das Bild an die ImageMagick-Bibliothek weiter, das als WebAssembly-Modul im Browser läuft; die Rotationsfunktion wird ausgeführt, wobei alle Bildmanipulationen clientseitig ablaufen. Nachdem das Bild fertig verarbeitet wurde, wird es an das TypeScript-Interface zurückgeliefert und kann als neue Datei gespeichert, beziehungsweise weiterverarbeitet werden. In dieser Art und Weise ist es möglich, die gesamte Bilddatenverarbeitung lokal auf dem Gerät des Nutzers durchzuführen, ohne auf eine stabile Internetverbindung und leistungsfähige Server angewiesen zu sein.

Codebeispiel 4: Verarbeitung eines Bildes per WASM-Magick Library

```
// Binärdaten des Bildes auslesen und in Array speichern
let sourceImage = await fetch(image.src);
let arrayBuffer = await sourceImage.arrayBuffer();
let sourceBytes = new Uint8Array(arrayBuffer);

// Bild per WASM-Magick auslesen
ImageMagick.read(sourceBytes, image => {
  // Bild drehen
  image.rotate(90);
  // Bild als JPEG zurückerhalten
  image.write(data => {}, MagickFormat.Jpeg);
});
```

Damit die entwickelte Beispielanwendung sinnvoll getestet und die Geschwindigkeit der Durchläufe verlässlich gemessen werden können, wurden Teile des Codes mit einer einfachen Zeitmessung verknüpft. Diese zeichnet die Start- und Endzeit der einzelnen Durchführungen auf und verrechnet sie miteinander. Das Codebeispiel 5 stellt diese Zeitmessung vereinfacht dar, indem vor und nach einem relevanten Codeblock die Zeit gemessen wurde.

Codebeispiel 5: Zeitmessung für die einzelnen Testdurchläufe

```
// Beginn Zeitmessung
console.log('Processing started...');
let startTime = new Date();

[...]

// Ende Zeitmessung
const endTime = new Date();
let totalTime = endTime.valueOf() - startTime.valueOf();
console.log(`Processing finished in ${totalTime} ms`);

```

6.2.2. Vergleichsanwendung in Node.js

Zusätzlich zur Webanwendung mit WebAssembly wurde eine Vergleichsanwendung entwickelt, die in den Tests als Referenz herangezogen wird. Es wurde darauf geachtet, eine möglichst ähnliche Funktionsweise im Vergleich zur WebAssembly-Beispielanwendung zu gewährleisten, damit wenige zusätzlich zu beachtende Faktoren in die Ergebnisse mit einwirken. Der Hauptunterschied in der Funktionalität ist die Bildverarbeitung. Es wurde ebenfalls die Bildverarbeitungsbibliothek ImageMagick verwendet, um ein Bild genauso um 90° zu drehen. Die Oberfläche der Anwendung ist in Abbildung 5 erkennbar.

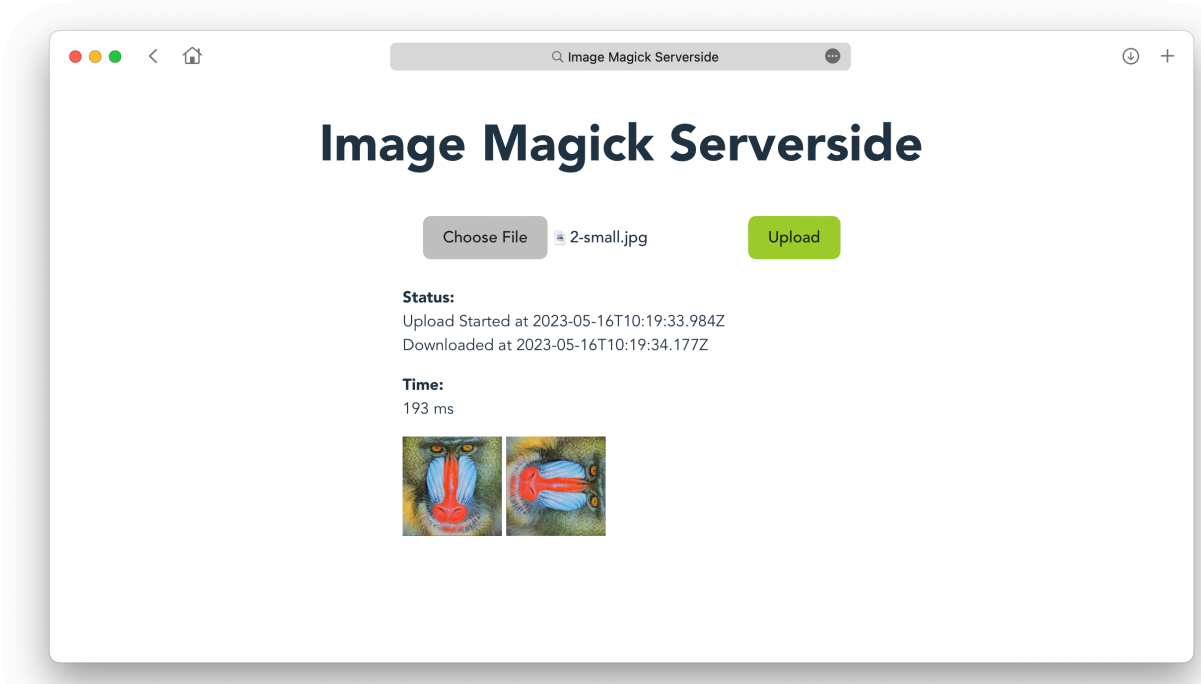

Abbildung 5: Oberfläche der Vergleichsanwendung mit serverseitiger Implementierung (eigene Grafik)

Die Anwendung wurde unter Verwendung des Express-Frameworks in Node.js umgesetzt. Wenn der Nutzer die Seite besucht, wird im Grunde genommen eine strukturell gleiche HTML-Seite wie in der ersten Beispielanwendung zurückgeliefert. Der einzige Unterschied besteht darin, dass nach der Auswahl eines Bildes ein Serverupload ausgelöst wird, wobei im Anschluss das Bild mittels ImageMagick verarbeitet und wieder über das Netzwerk zurückgesendet wird. Daher wurde die Anwendung zweigeteilt entwickelt: im Ersten der JavaScript Code clientseitig im Browser des Nutzers, der für den Bildupload und -download zuständig ist, und im Zweiten die serverseitige Implementierung der Bildverarbeitung per Node.js.

Im folgenden Codebeispiel wird sichtbar, wie das ausgewählte Bild per AJAX Request an den Server gesendet wird, wo die Umwandlung stattfindet. Gleichzeitig wird die Antwort des Servers verarbeitet und das neue Bild per Download an den Nutzer ausgeliefert.

Codebeispiel 6: Clientseitiger AJAX Request an Server zum Bildupload

```
1  // Bild zum Server Hochladen
2  fetch(window.location.origin + "/convert", {
3    method: 'POST',
4    body: formData   // Bild wird als form-data abgesendet
5  })
6    .then(res => res.blob())
7    .then(blob => {
8      // Bild clientseitig herunterladen
9  });
```

Wird ein Bild an den Node-Server gesendet, muss es anfänglich zunächst hochgeladen und als Datei abgelegt werden, um die Weiterverarbeitung zu ermöglichen (siehe Codebeispiel 7, Zeile 2). Als Uploader wurde eine Bibliothek für das Express-Framework namens Multer integriert, welche die Formulardaten entgegen nimmt und sich um die Dateiverarbeitung kümmert (ExpressJS, 2023). Nachdem ein Bild auf dem Server zur Verfügung steht, wird es direkt an die auf dem Server zur Verfügung stehende ImageMagick-Software gesendet (siehe Codebeispiel 7, Zeile 9). Dazu wird ebenfalls eine Library genutzt, die mit der serverseitigen Version von ImageMagick kommuniziert (Heckmann, 2023). Wenn keine Probleme auftreten und die Rotation des Bildes um 90° abgeschlossen ist, kann das neu-generierte Bild als Download zum Browser zurückgesendet werden.

Codebeispiel 7: Serverseitige Verarbeitung des Bildes in Node.js nach Upload

```
1  // POST Request: Bildverarbeitung und Upload
2  app.post('/convert', upload.single('file'), (req, res) => {
3
4    // Dateipfade zu den Bildern auf dem Server
5    let old_file, new_file;
6
7    try {
8      // Bild drehen
9      gm(old_file).rotate('white', 90)
10       .write(new_file, function () {
11         // Neues Bild als Download zurücksenden
12         res.download(new_file);
13       });
14   }
15   catch (error) {
16     res.send(error);
17   };
18 })
```

Die Zeitmessung der serverseitig implementierten Anwendung wurde in gleicher Weise wie in der Beispielanwendung in WebAssembly umgesetzt (siehe Codebeispiel 5), wobei durch die clientseitige Zeitmessung die Hin- und Rücklaufzeit der Daten über das Netzwerk in die Gesamtlaufzeit mit einfließen.

6.3. Durchführung der Messreihen

Die Vorgehensweise bei den Testdurchläufen wird nachfolgend beschrieben. Beide Anwendungen wurden über einen Server zur Verfügung gestellt. Um einen Vergleich zur nativen Performance von ImageMagick zu ermöglichen, wurden alle Tests zusätzlich per CLI (Command Line Interface) sowohl in der Serverumgebung als auch auf dem Macbook getestet. Somit konnten zugleich zwei Fragen beantwortet werden: Einerseits, wie sich eine einfache Anwendung in WebAssembly im Vergleich zu einer traditionellen Alternativanwendung verhält und andererseits, ob WebAssembly tatsächlich in praktischen Anwendungsfällen an native Ausführungsgeschwindigkeiten herankommt.

6.3.1. Voraussetzungen

Die Messreihen wurden unter bestimmten Bedingungen durchgeführt. In allen Testumgebungen und bei allen Implementierungen wurden bestimmte Vorkehrungen getroffen, sodass die Messung möglichst gleichmäßige, vergleichbare und repräsentative Ergebnisse liefert. Wenn nachstehend von 'Testumgebungen' gesprochen wird, sind darunter alle Infrastrukturen zu verstehen, die dem Test einen neuen Einflussfaktor geben: also verschiedene Browser (Chrome, Firefox, Safari,...) oder verschiedene Geräte (Laptop, Mobilgerät,...). Zusätzlich dazu existiert der Faktor der verschiedenen Anwendungen bzw. Testgebiete (Beispielanwendung in WebAssembly, Vergleichsanwendung in Node.js, Native Tests per CLI), der generell mit dem Begriff 'Implementierung' verdeutlicht werden soll.

Die erste Voraussetzung für aussagekräftige Testergebnisse ist die Verwendung gleicher Referenzbilder. Dementsprechend wurden sechs verschiedene Bilder unterschiedlicher Dateigröße und Dateiformate ausgewählt, um diese in allen Testumgebungen und Implementierungen miteinander vergleichen zu können (siehe Tabelle 1). Das erste Bild besteht nur aus einem Pixel, mit dem Ziel, bei Tests in der serverseitigen Implementierung durch niedrige Dateigröße den Faktor der Netzwerkgeschwindigkeit möglichst gering zu halten. Beim letzten Bild handelt es sich um ein Bild im PNG-Format mit hoher Auflösung, wodurch Edge-Case-Szenarien abgedeckt werden sollen. Alle Bilder haben besondere Farb- und Kontrasteigenschaften und unterschiedliche Seitenverhältnisse, wodurch sie sich als Testbilder für diese Messreihen gut eignen.

Bildname	Dateigröße	Abmessungen	Format
tiny	160 B	1 x 1 px	JPG
small	0.23 MB	512 x 512 px	JPG
fullhd	0.63 MB	1920 × 1080 px	JPG
medium	3.91 MB	4160 × 2080 px	JPG
large	10.2 MB	6000 × 4000 px	JPG
huge	79.1 MB	4777 × 3185 px	PNG

Tabelle 1: Verwendete Testbilder und ihre Eigenschaften

Um die Tests in variablen Umgebungen durchführen zu können, wurde sichergestellt, dass es sich generell um die gleichen Anwendungen handelt. Dementsprechend wurde ein Azure Server gemietet und der Code als Paket auf diesen Server geladen. Von dort aus wurde anschließend jeweils von den verschiedenen Geräten auf die Webanwendungen zugegriffen, wobei in der Beispielanwendung immer der gleiche WebAssembly-Code zum Client geliefert wurde, und bei der Vergleichsanwendung die Bilder auf einem einzigen Server verarbeitet werden konnten.

Zuletzt wurde gewährleistet, dass gleiche Systemvoraussetzungen in allen Szenarien bestehen. Es wurde darauf geachtet, dass bei allen Tests in allen Umgebungen der Browser als einzige aktive Anwendung geöffnet war. Dazu wurden die Testgeräte vorher neu gestartet und alle irrelevanten Hintergrundanwendungen geschlossen. Weiterhin wurden die Testergebnisse auf einem anderen Gerät notiert, um nicht durch zusätzlich geöffnete Programme auf dem Testgerät die Ergebnisse zu verfälschen. Schlussendlich wurde bei den Testdurchläufen der Vergleichsanwendung die Netzwerkgeschwindigkeit in regelmäßigen Abständen getestet, um konstante Ergebnisse zu erzielen.

6.3.2. Testumgebungen

Der erste Teil in der Durchführung der Messungen bestand darin, die Beispielanwendung in WebAssembly in verschiedenen Testumgebungen zu untersuchen. Dabei wurden einerseits unterschiedliche Browser verwendet und andererseits verschiedene Desktop- und Mobilgeräte. Die Browsertests wurden allesamt auf einem MacBook Pro 2020 durchgeführt, wobei die vier großen Browser Chrome, Safari, Firefox und Edge berücksichtigt wurden (siehe Tabelle 2). Zum Vergleich wurde außerdem ein älterer Windows-Laptop herangezogen. Bei den Tests auf Mobilgeräten wurde in gleicher Weise darauf geachtet, dass ein verhältnismäßig neueres Gerät der jeweiligen Plattform (Android, bzw. iOS) mit einem älteren Modell verglichen wurde. Des Weiteren wurde bei der Auswahl der Geräte auf eine möglichst große Leistungsvielfalt der Prozessoren Wert gelegt, um ein weites Spektrum von Anwendungsfällen der Praxis abzudecken.

Spezifikationen	MacBook Pro 2020	Asus P2520L (Windows)	iPhone 14 Pro Max	iPhone 6s	Pixel 5	Samsung A5
Prozessor	Intel i5-1038NG7	Intel i5-5200U	A16 Bionic Chip	A9 Chip	Snapdragon 765G	Exynos 7880
Prozessorkerne	4	2	6	2	8	8
RAM in GB	16	8	6	2	8	3
Erscheinungsjahr	2020	2015	2022	2015	2020	2017
Testbrowser	Firefox, Chrome, Safari, Edge	Chrome	Safari on iOS	Safari on iOS	Chrome for Android	Chrome for Android

Tabelle 2: Testumgebungen für die Beispielanwendung in WebAssembly

Im Diagramm 1 wird dieses Spektrum der Geräteleistungen sichtbar: Der Geekbench 6 CPU-Benchmark zeigt die Prozessorleistung der Laptops und Mobilgeräte auf. Da nicht davon ausgegangen werden kann, dass heutzutage jeder Endnutzer mit einem neuen, leistungsstarken Flaggschiffmodell ausgestattet ist, wurden gängige, ältere Geräte in die Tests miteinbezogen.

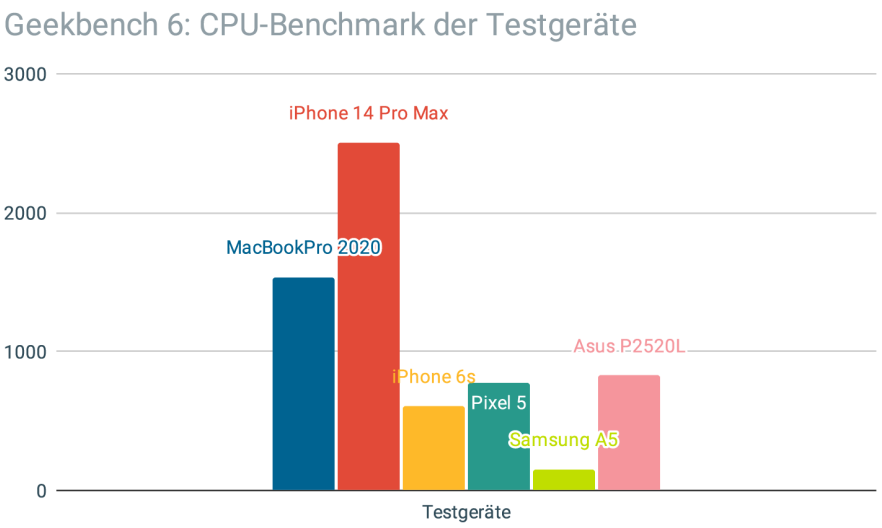

Diagramm 1: Vergleich der CPU-Benchmark Ergebnisse der Geräteprozessoren

Im Zweiten Teil der Testdurchführung wurde die Vergleichsanwendung mittels serverseitiger Implementation der ImageMagick-Bibliothek auf einer Serverinstanz und in drei verschiedenen Browsern (Chrome, Firefox, Safari) auf dem MacBook Pro 2020 (siehe Tabelle 2) geprüft und miteinander verglichen. Dazu wurde über Microsoft Azure eine Virtuelle Maschine mit Ubuntu Server gehostet, worauf wiederum Node.js zum Einsatz kam (Tabelle 3).

Als dritter und letzter Teil der Tests wurden alle Bilder ebenfalls über die CLI der ImageMagick-Bibliothek konvertiert. Dies wurde per MacBook und außerdem auf dem Azure-Server per SSH durchgeführt. Um den Einfluss des Servertyps und seine Leistung zu berücksichtigen, wurden auch Tests auf einer zweiten Serverinstanz mit größerer Rechenkapazität stichprobenartig vollzogen. Dafür wurde kurzzeitig ein Upgrade des Azure Servers vorgenommen (Tabelle 3).

Spezifikationen	Serverinstanz 1	Serverinstanz 2
Name	Azure B2s	Azure DS12_v2
Kategorie	General purpose	Memory optimized
vCPU	2	4
RAM in GiB	4	28
Max. Anzahl Datenträger	4	16
Max. Datenträger-Durchsatz	1280	12800
Temporärer Speicher (SSD): GiB	8	56
Kosten/Monat	$35.04	$276.67

Tabelle 3: Spezifikationen der beiden verwendeten Azure Server VMs (Microsoft Azure, 2022a & 2022b)

6.3.3. Vorgehensweise

Nachdem alle umfangreichen Vorkehrungen getroffen wurden, konnte mit der Durchführung der Tests begonnen werden. Pro Umgebung ist jeweils eines der sechs Bilder in zehn Durchläufen hintereinander konvertiert worden, wobei jedes Mal die Konvertierungszeit vermerkt wurde. Begonnen wurde mit der Beispielanwendung in WebAssembly, danach wurde die serverseitige Implementierung getestet und abschließend als Vergleich auf der Kommandozeile. Für jede Umgebung und jedes Bild wurden die zehn Messreihen zu einem Mittelwert zusammengefasst und die Standardabweichung errechnet, um die Qualität der Messergebnisse besser beurteilen zu können.

Bei jedem Durchlauf wurde die Anwendung durch das erneute Laden der Webseite gestartet. Bei den WebAssembly-Tests wurde zudem die Initialisierungszeit des WebAssembly-Moduls zeitlich bestimmt und erfasst. Da es sich beim letzten Bild (huge.png) um eine PNG-Datei handelt, wurde die WebAssembly-Umgebung für diese Testreihe entsprechend angepasst, sodass ein Bild im PNG-Format zurückgeliefert wurde, anstelle des JPG-Formats.

Bei Tests der serverseitigen Implementierung wurde die Netzwerkgeschwindigkeit über die Kommandozeile in regelmäßigen Abständen gemessen. Für die Browserumgebungen Firefox,

```
> networkQuality -s
```

Chrome und Safari wurden die Tests zudem in zwei verschiedenen Netzwerken mit unterschiedlicher Internetgeschwindigkeit durchgeführt, um die Einflüsse derselben auf die Endresultate bestimmen zu können.

Im dritten Schritt, der Verarbeitung per CLI, wurde der `time` Befehl verwendet, um die Dauer der Konvertierung mit ImageMagick messen zu können. So wurden in gleicher Weise alle sechs Bilder per Rotationsalgorithmus um 90° gedreht (siehe Codebeispiel 8).

Codebeispiel 8: CLI Zeitmessung der Konvertierung der Bilder mit ImageMagick

```
time convert small.jpg -rotate 90 rotated.jpg

real    0m0.037s
user    0m0.026s
SYS     0m0.008s
```

Die Messungen hängen somit von einer Vielzahl von Einflussfaktoren ab. Hierzu zählen: der Dateityp oder die Dateigröße, die Netzwerkgeschwindigkeiten und Testumgebungen (Desktop & Mobil), die Implementierungen und Serverkonfigurationen. Auf die Korrelationen dieser Faktoren und deren Einfluss auf die Ergebnisse wird nun im nächsten Kapitel Bezug genommen.

7. Ergebnisanalyse nach Einflussfaktoren

Durch die Messungen konnten zahlreiche Ergebnisse ermittelt werden. In dieser Auswertung sollen die Ergebnisse kategorisiert und nach ihren Einflussfaktoren dargestellt werden, damit diese besser einzuordnen sind.

7.1. Implementierung

Die vielleicht interessanteste oder auch wichtigste Erkenntnis der gesamten Vergleichsstudie ist, dass die Beispielanwendung in WebAssembly auf einem Desktop-Endgerät generell schneller ist als die Vergleichsanwendung mit serverseitiger Ausführung bei einer Internetgeschwindigkeit von durchschnittlich 57 Mbit/s. Dies ist insbesondere vor dem Hintergrund der durchschnittlichen Internetgeschwindigkeit in Deutschland von 84 Mbit/s interessant (Statista, 2023). In der schnellsten Desktopumgebung (in Firefox auf dem MacBook Pro) ist WebAssembly circa 8x schneller als in der serverseitigen Ausführung mit der oben beschriebenen Netzwerkgeschwindigkeit (siehe Diagramm 2). Wie bereits erwähnt, wurde der Test der serverseitigen Implementierung ebenfalls für eine schnellere Netzwerkverbindung von durchschnittlich 255 Mbit/s durchgeführt. Auch in diesem Umfeld konnte WebAssembly im Schnitt 1,7x schnellere Ergebnisse erzielen, wobei der Faktor des Geräts an dieser Stelle wiederum nicht zu vernachlässigen ist. Die WebAssembly-Anwendung auf dem alten Laptop-Modell konnte im Vergleich nur zur serverseitigen Implementierung mit langsamerer Netzwerkgeschwindigkeit schnellere Leistungen erreichen.

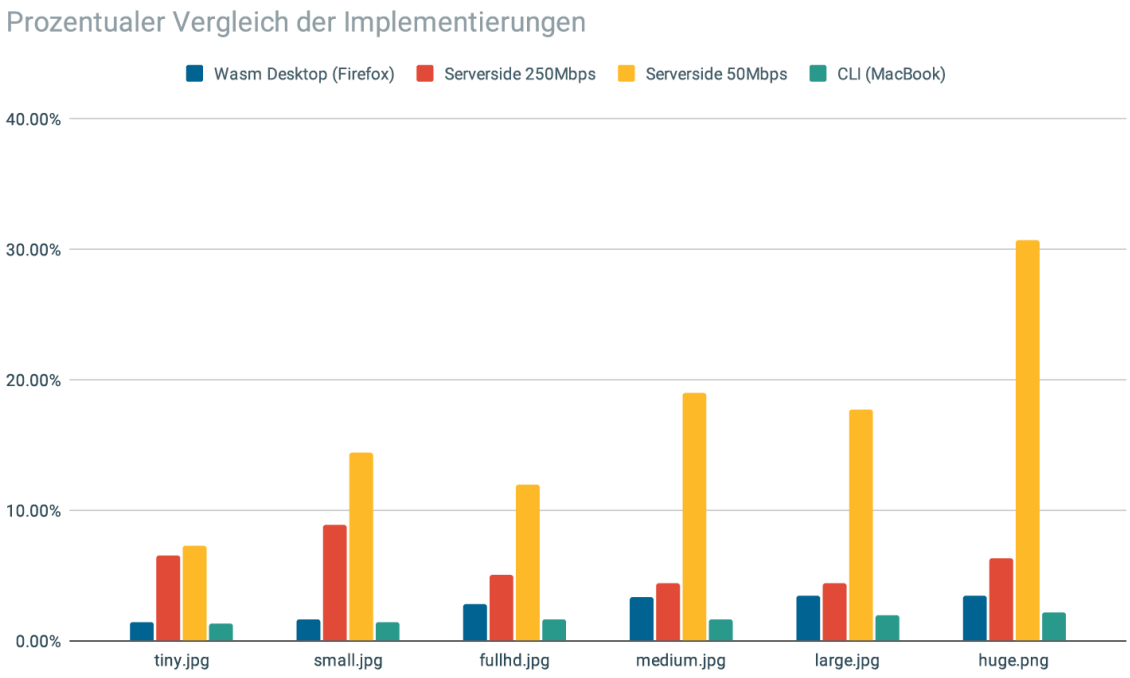

Diagramm 2: Vergleich der verschiedenen Implementierungen und Netzwerkgeschwindigkeiten

Trotz der hohen Geschwindigkeit der Beispielanwendung muss dennoch festgestellt werden, dass WebAssembly in der schnellsten Desktop-Umgebung (Firefox) trotzdem immer noch 37,7% langsamer ist als die native Ausführung der ImageMagick-Bibliothek per CLI auf gleichen Endgeräten. Somit ist die native Verwendung der Kommandozeile weiterhin die schnellste Implementierung für diesen Musterfall. Dieses Ergebnis ist in der Tat nicht verwunderlich: WebAssembly versucht nicht, schneller zu sein als nativer Code, sondern eben annähernd schnell. Es wird von Anwendungsfall zu Anwendungsfall unterschiedlich sein, wie sinnvoll die Verwendung von WebAssembly ist und ob die damit einhergehenden Performanceeinbußen verkraftbar sind.

Im Gesamtvergleich aller getesteten Umgebungen und Implementierungen zeigt sich, dass WebAssembly für diesen Anwendungsfall auf älteren Mobilgeräten weniger lohnend ist, da hier schlechtere Ergebnisse erzielt werden (siehe Diagramm 3). Die serverseitige Implementierung setzt jedoch auch eine schnelle Netzwerkgeschwindigkeit voraus, die wiederum auf Mobilgeräten nicht zwingend garantiert werden kann.

Ebenfalls wird deutlich, dass die serverseitige Vergleichsanwendung hauptsächlich bei kleinen Dateigrößen etwas schneller ist als in anderen Umgebungen wie beispielsweise auf Android-Phones oder dem alten Laptop. Dies ist auf die Übertragungszeit der Bilddaten zurückzuführen, die bei kleineren Bildern immer weniger ins Gewicht fällt.

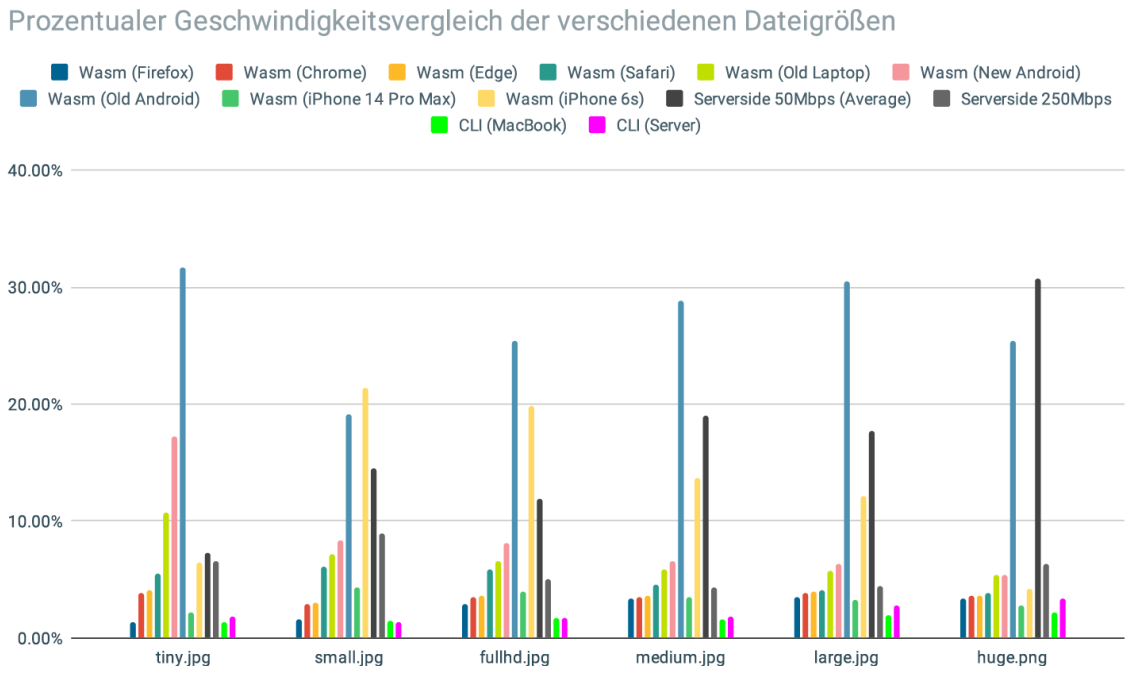

Diagramm 3: Vergleich aller Umgebungen in allen Implementierungen (WebAssembly, Serverseitig & CLI)

7.2. Umgebung

Aus Diagramm 4 wird ersichtlich, dass WebAssembly in Firefox am schnellsten ausgeführt werden kann, danach folgen die Browser Chrome und Edge, während Safari unter den Desktop-Browsern auf dem MacBook die langsamste Performance aufweist. Dies ist eine ebenfalls interessante Beobachtung, weil hieraus ersichtlich wird dass die Browserhersteller in verschiedenster Weise die WebAssembly-Spezifikationen in ihre bestehenden JavaScript-Engines integriert haben (Diagramm 4). Die Browser Chrome und Edge liefern wiederum sehr ähnliche Ergebnisse, höchstwahrscheinlich deshalb, da beide auf Chromium basieren.

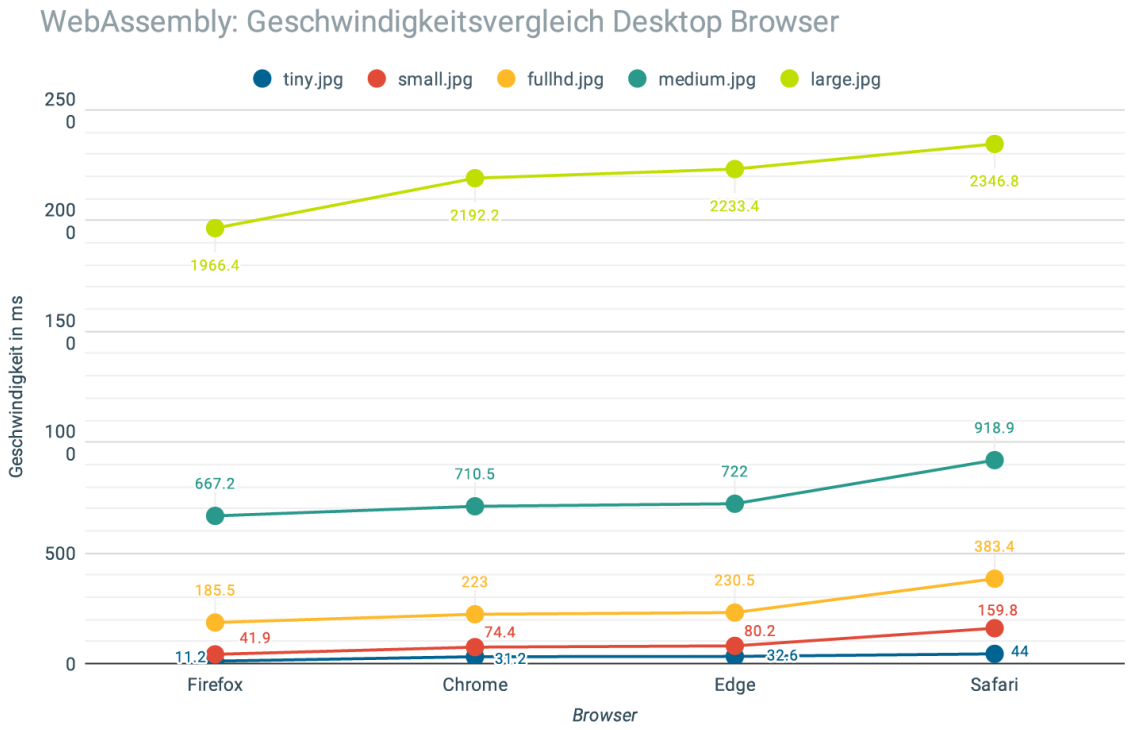

Diagramm 4: Vergleich der Desktop-Browser der Beispielanwendung in WebAssembly

Firefox schneidet insgesamt am besten in den Tests ab. Hierbei muss jedoch erwähnt werden, dass Firefox auf dem MacBook am längsten für die Initialisierung des WebAssembly-Moduls und der ImageMagick-Bibliothek benötigt als alle anderen Umgebungen, mit Ausnahme von Chrome auf dem alten Android-Gerät (siehe Diagramm 5). Hierdurch wird deutlich, wenn man dieses Ergebnis dem vorherigen gegenübergestellt, dass die hohe Geschwindigkeit von Firefox bei den Bildverarbeitungstests mit gewissen Kosten verbunden ist. Es scheint, als hat sich Mozilla dazu entschieden, ein gewisses Maß an Mehraufwand bei der Initialisierung von WebAssembly-Modulen in Form von weiteren Optimierungen, Caching oder anderen Verarbeitungstechniken in Kauf zu nehmen, um anschließend bei der Ausführung bessere Geschwindigkeiten zu ermöglichen. Im Vergleich dazu hat Chrome bei den Desktop-Browsern die schnellste Initialisierungszeit, bei den Mobilgeräten liegt das iPhone 14 vorne.

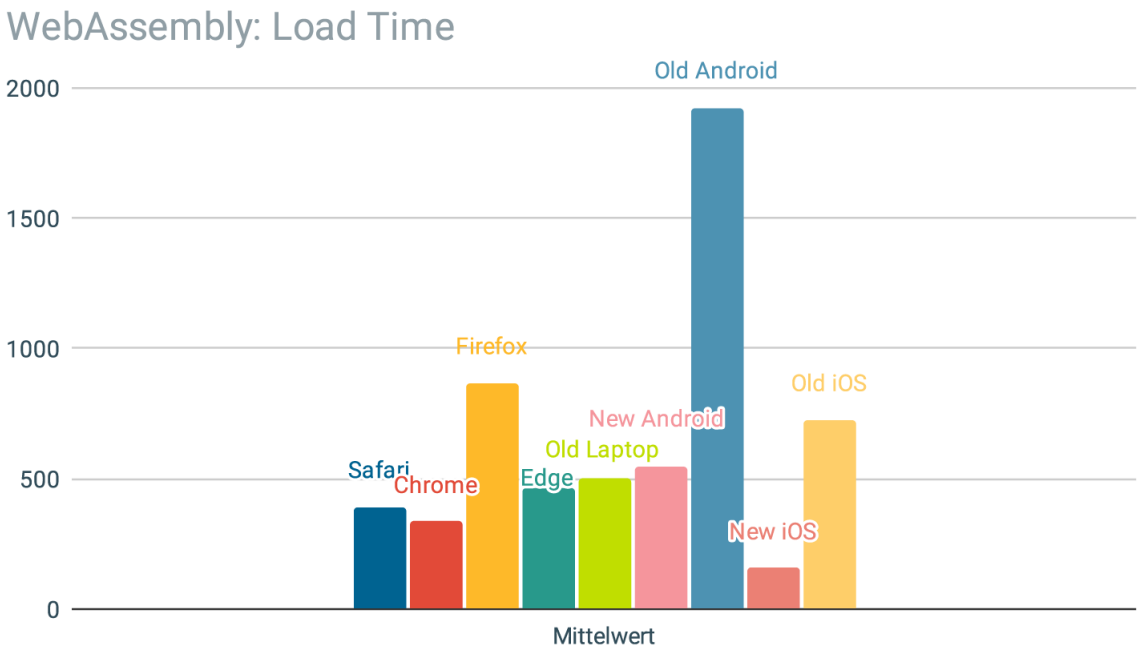

Diagramm 5: Initialisierungszeit des WebAssembly-Moduls in allen Umgebungen

Bei den Tests der serverseitigen Implementierung wurde ebenfalls überprüft, inwieweit die Umgebung einen relevanten Einflussfaktor auf die Ergebnisse darstellt. Dementsprechend wurden die Versuche in drei verschiedenen Browsern durchgeführt: Firefox, Chrome und Safari. Ziemlich schnell wurde klar, dass die Unterschiede zwischen den Browsern ziemlich marginal ausfallen (Diagramm 6). Dementsprechend wurde diese Implementierung auch in keinen weiteren Umgebungen getestet, um den Umfang der Messreihe nicht unnötig in die Höhe zu treiben. Die kleinen Differenzen zwischen den drei Browsern sind vermutlich auf schwankende Netzwerkgeschwindigkeiten zurückzuführen.

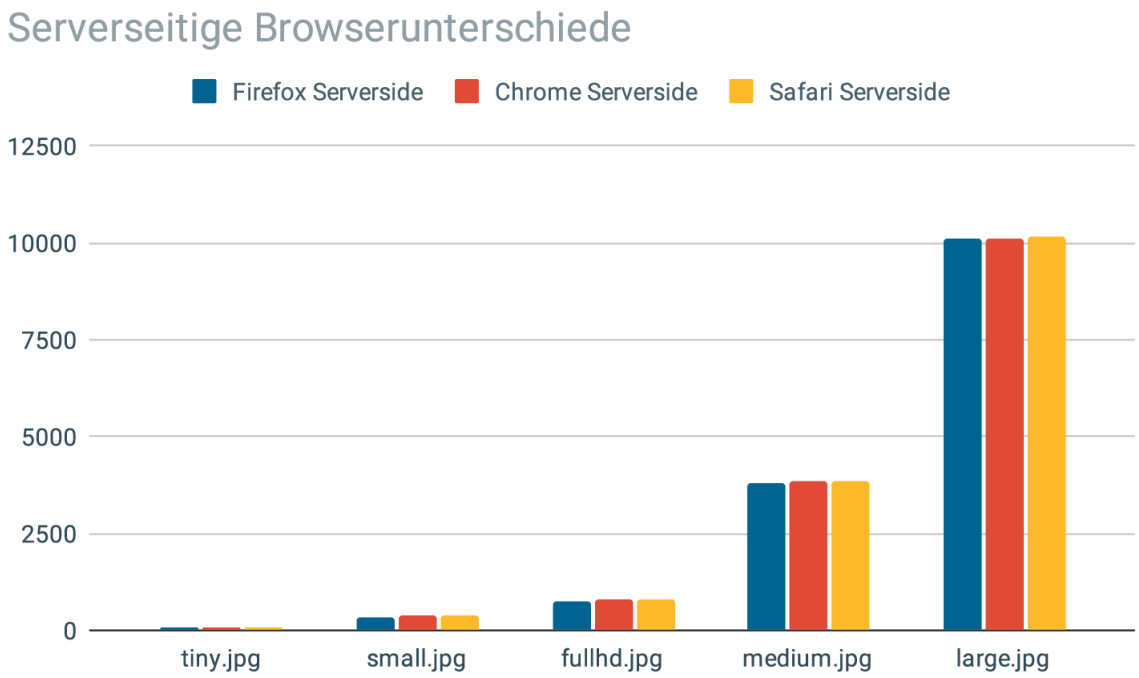

Diagramm 6: Geschwindigkeitsvergleich bei serverseitiger Implementierung in drei Browsern

Als letztes kann auf die Umgebungsunterschiede in den CLI Tests eingegangen werden. Im Allgemeinen ist die Ausführung per Kommandozeile auf dem Macbook im Durchschnitt schneller als auf den getesteten Servern (Diagramm 7). Die Unterschiede sind jedoch nicht gravierend, wobei natürlich darauf geachtet werden muss, dass die Server nicht die gleichen Systemvoraussetzungen haben wie das MacBook. Die ausgewählten Server sind in ihrem Aufbau und ihrer Leistungsverfügbarkeit nur schwer mit einem dedizierten Gerät zu vergleichen. Obwohl die verwendeten Server im CoreMark CPU-Benchmark Test bessere Werte als das MacBook erzielten, kann dies nicht auf jede Situation gleich übertragen werden, wie in den durchgeführten Tests zu erkennen ist (Microsoft Azure, 2020).

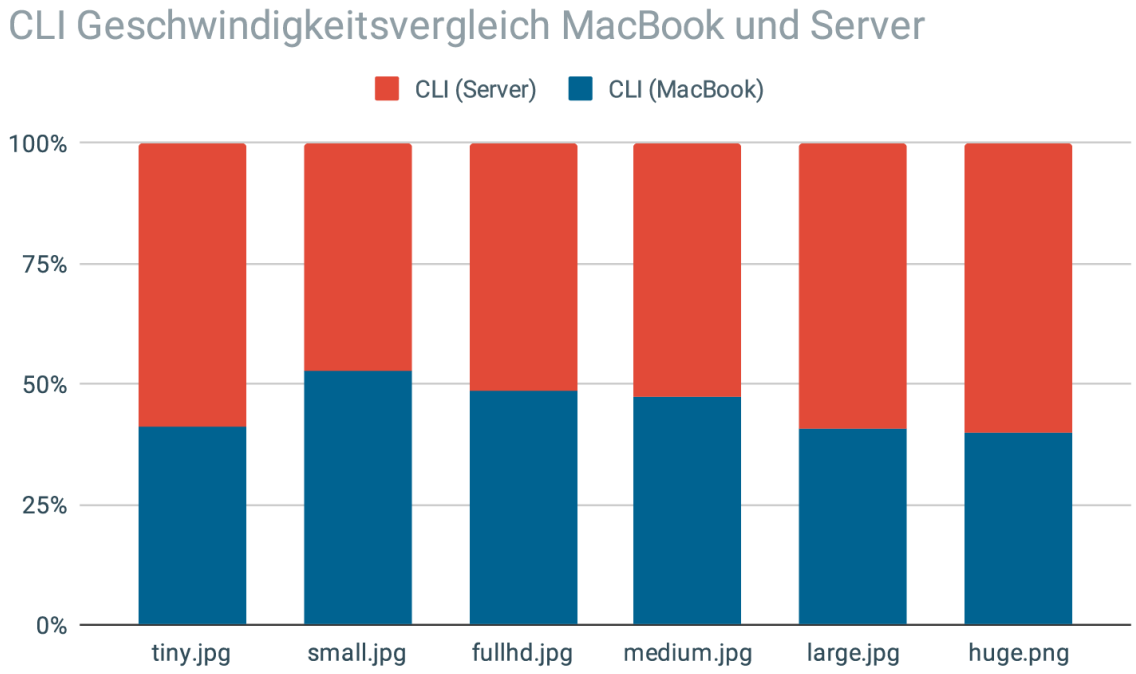

Diagramm 7: Geschwindigkeitsvergleich bei der Verarbeitung per Kommandozeile

7.3. Hardware

Als dritter Einflussfaktor auf die Ergebnisse soll die verwendete Hardware besprochen werden. Unter Hardware werden die verwendeten Geräte, die verschiedenen Netzwerkgeschwindigkeiten und die beiden andersartigen Serverkonfigurationen verstanden.

Generell kann gesagt werden, dass innerhalb der verschiedenen Umgebungen in WebAssembly die Performanceunterschiede stetig abnehmen, je größer die hochgeladene Datei ist. Es findet eine Annäherung statt, wie in Diagramm 8 deutlich zu erkennen ist. Besonders die Testergebnisse in den Desktop-Browsern gleichen sich immer weiter an. Interessant zu beobachten ist auch, dass in den Tests auf dem älteren iPhone Modell bei steigender Bildgröße verhältnismäßig immer bessere Resultate erzielt werden, wobei auf dem älteren Android-Gerät der gegenteilige Effekt auftritt.

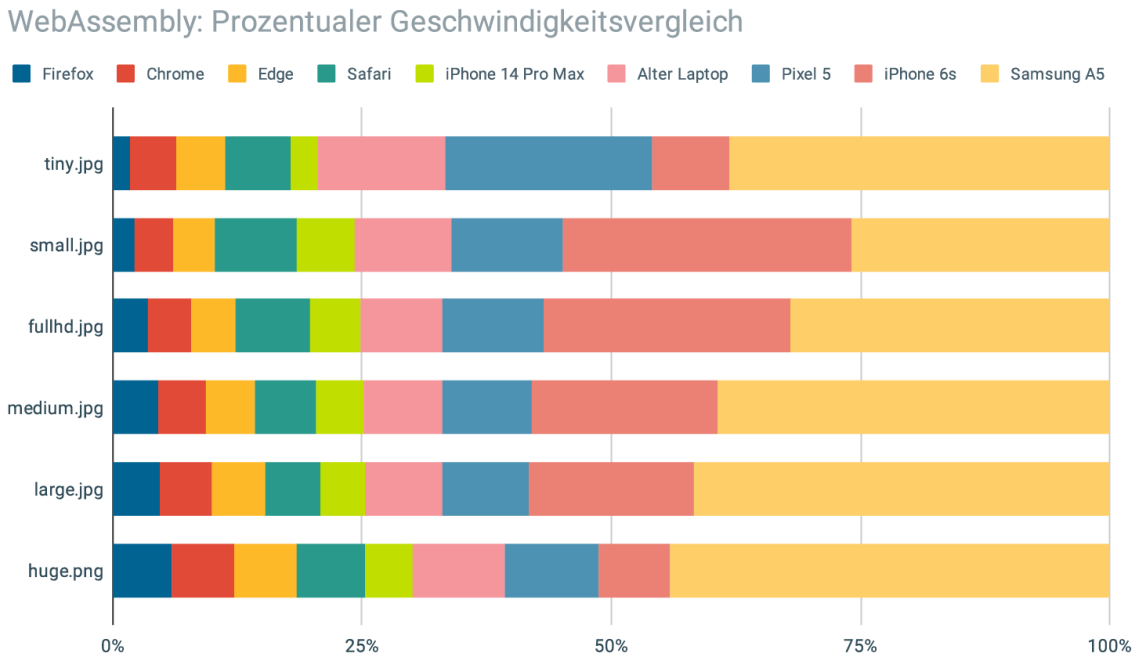

Diagramm 8: Geschwindigkeitsvergleich über alle Geräte hinweg in WebAssembly

Sowohl im Diagramm 8 als auch im Diagramm 9 wird deutlich, wie variabel die Geschwindigkeiten auf verschiedenen Geräten ausfallen. Der Faktor des Geräts und der damit verbundenen Hardware spielt also eine große Rolle bei der Verwendung von WebAssembly. So kann festgestellt werden, dass ältere und mobile und dementsprechend meist weniger leistungsfähige Geräte im Test deutlich langsamer ausfallen als andere Geräte. Auch an dieser Stelle ist zu erwähnen, dass die als 'neu' oder 'alt' deklarierten Geräte sehr unterschiedlich ausgestattet und dementsprechend nicht unbedingt direkt miteinander vergleichbar sind. Der Fokus der Tests lag aber darauf, eine möglichst große Bandbreite an Geräten abzudecken um dahingehend aussagekräftige Werte zu erhalten.

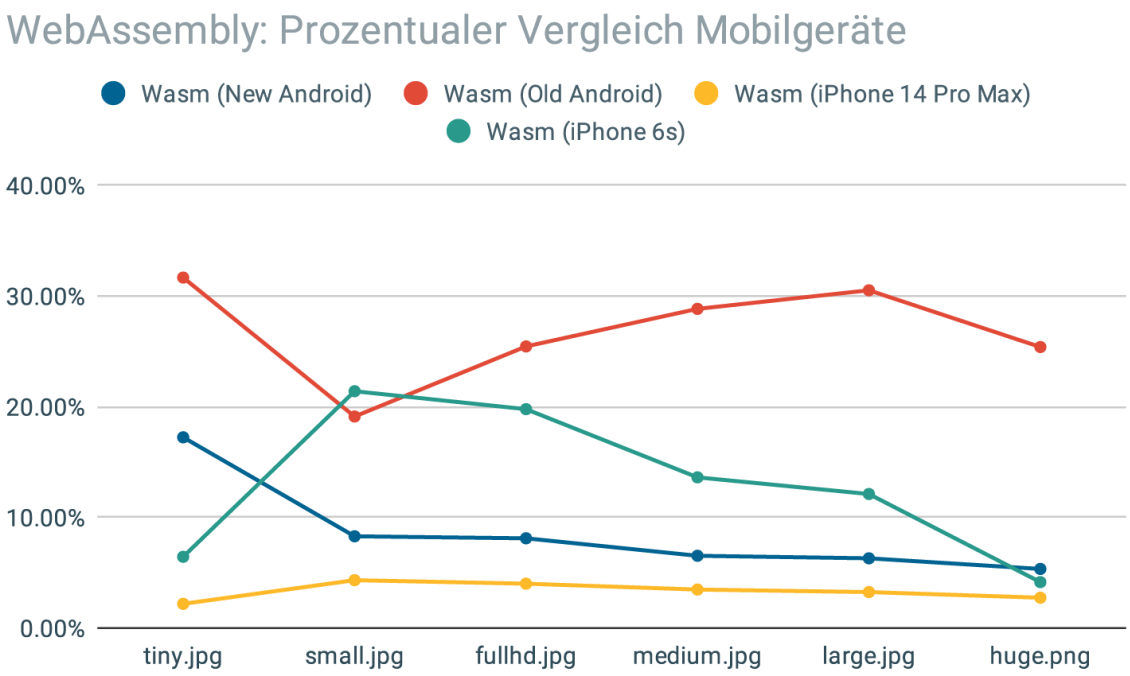

Diagramm 9: Vergleich der Mobilgeräte für die Beispielanwendung in WebAssembly

Dass mobile Geräte in der Lage sind, WebAssembly nicht nur zufriedenstellend, sondern sogar mit herausragender Leistung ausführen zu können, zeigt Diagramm 10. In manchen Fällen schlägt das iPhone Top-Modell sogar die Desktop Performance der Beispielanwendung auf dem MacBook, besonders wenn größere Dateien verarbeitet werden müssen. Generell ist das getestete, aktuelle iPhone Modell in allen Tests immer mit an der Spitze. Somit kann geschlussfolgert werden, dass die Technologie WebAssembly, je nach Anwendungsfall, in gleicher Weise auch auf Mobilgeräten mit guten Ergebnissen zum Einsatz kommen kann, da mittlerweile die Rechenleistung mobiler Geräte immer weiter steigt. Dies gilt in diesem Fall natürlich besonders für das Spitzenmodell von Apple, nichtsdestotrotz ließen sich aber auch mit anderen relativ modernen Smartphones akzeptable Geschwindigkeiten erreichen. Es bleibt zu hoffen, dass Mobilgeräte immer weiterentwickelt werden, um in gleicher Weise als Zielumgebung für WebAssembly genutzt werden zu können.

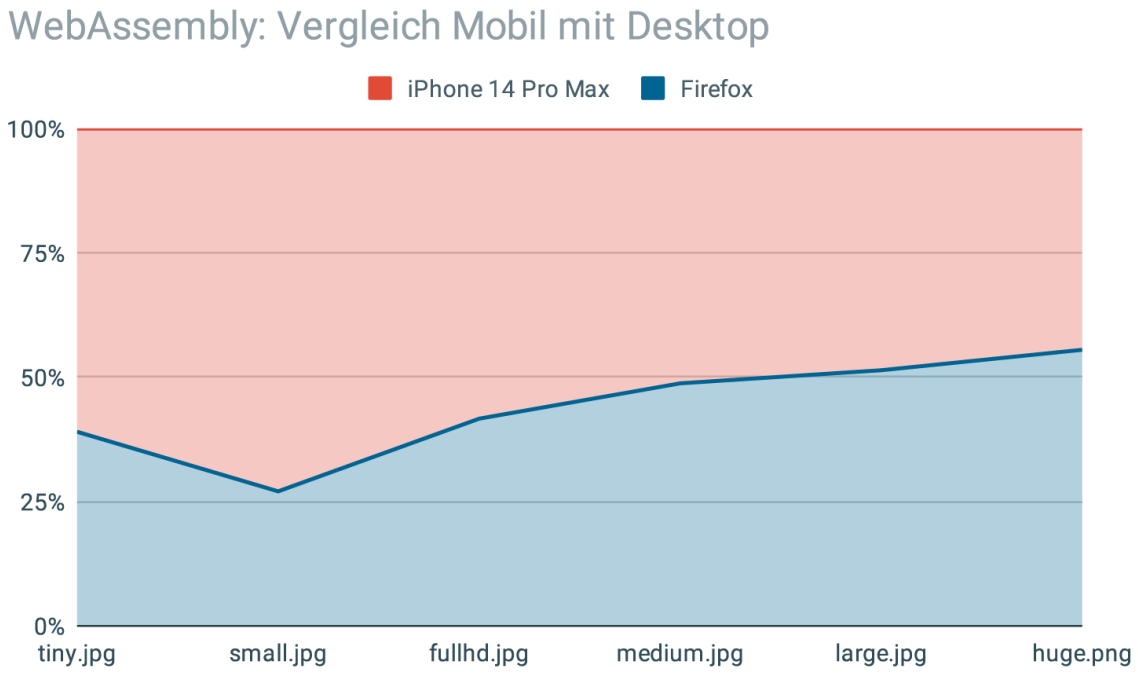

Diagramm 10: Mobilgerät schlägt Desktop-Performance für große Dateien in WebAssembly

Für die Tests der serverseitigen Vergleichsanwendung spielt die Hardware und das Netzwerkumfeld einen entscheidenden Faktor. Aus der Praxis wird ersichtlich, dass die Anwendung bei steigender Dateigröße zunehmend von einer schnellen Netzwerkverbindung profitiert, da Upload- und Download-Zeit nicht mehr so stark ins Gewicht fallen. Im Diagramm 11 ist dieser Sachverhalt deutlich sichtbar. Bei einer Internetverbindung von circa 57 Mbit/s beeinflusst die Dateigröße die Ergebnisse verstärkt, bei einer sehr schnellen Netzwerkgeschwindigkeit von durchschnittlich 255 Mbit/s hat der Datentransfer dagegen keine großen Auswirkungen. Da solche Internetgeschwindigkeiten in Deutschland momentan jedoch nicht flächendeckend garantiert werden können, kann WebAssembly, wie bereits erwähnt, große Vorteile bieten.

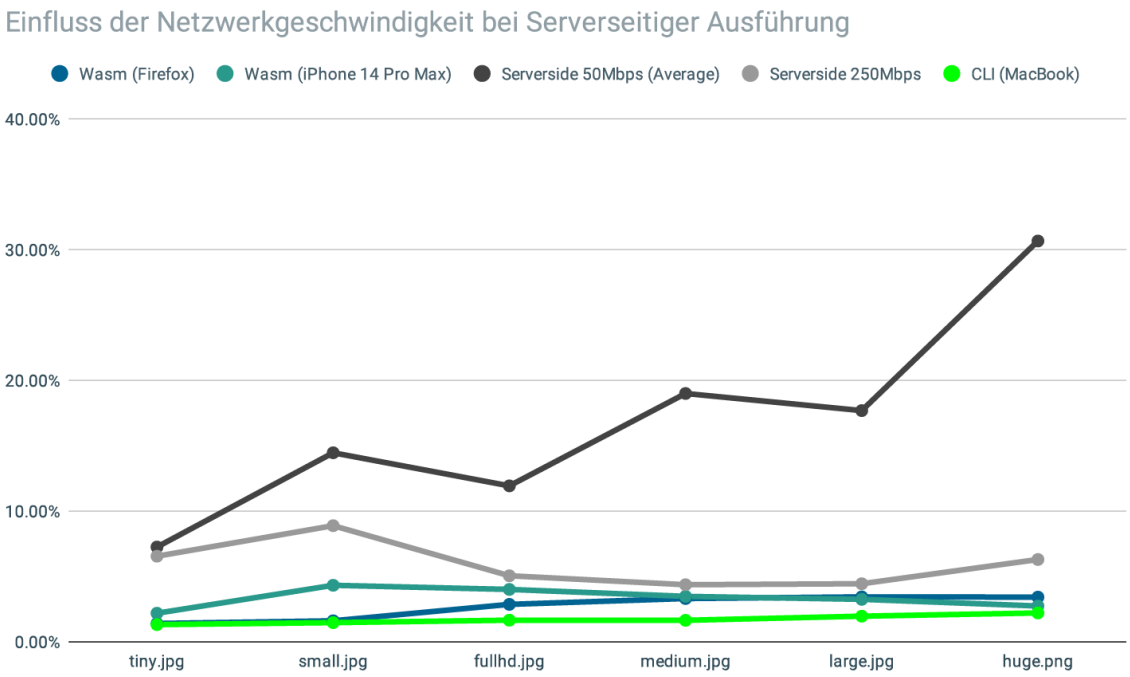

Diagramm 11: Vergleich verschiedener Implementierungen und Internetgeschwindigkeiten

Weiterhin konnte in Bezug auf die Netzwerkgeschwindigkeit festgestellt werden, dass Browser-Unterschiede bei der serverseitigen Implementierung vorliegen. Alle drei Browser wurden einmal im langsameren Heimnetz und anschließend im schnelleren Hochschulnetzwerk getestet. Im Forschungsnetz der HTWK erreichte der Firefox-Browser interessanterweise viel höhere Uploadgeschwindigkeiten als dies in Chrome oder Safari der Fall war (siehe Diagramm 12). Daraus resultieren folglich die Performanceunterschiede der Testanwendung zwi-

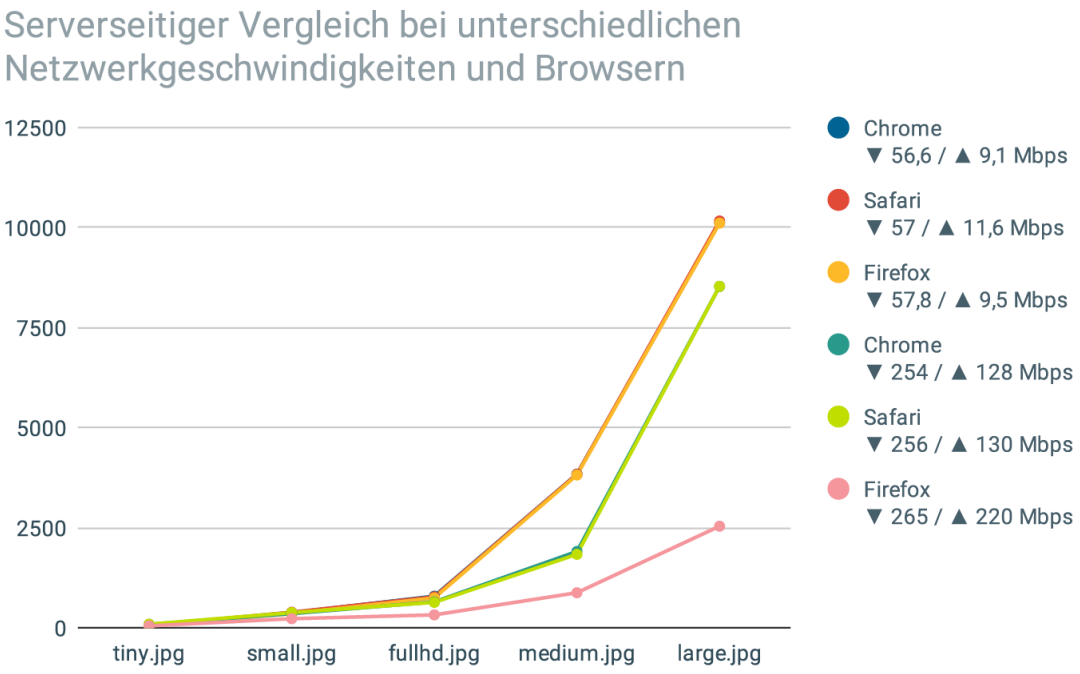

Diagramm 12: Vergleich der Internetgeschwindigkeit in verschiedenen Browsern

schen den Browsern im schnellen Netzwerk. Eine direkte Ursache für diese Differenzen konnte leider nicht gefunden werden, auf Nachfrage beim IT-Servicezentrum der HTWK wurden jedoch unterschiedliche Proxy-Konfigurationen als mögliche Ursache genannt.

Die CLI-Tests auf dem Server liefern insofern wichtige Informationen, da durch den Vergleich der beiden unterschiedlichen Serverkonfigurationen der tatsächliche Einfluss der verfügbaren Hardware untersucht werden konnte. Auf dem leistungsfähigeren Server wurden die Tests aus Kostengründen nur für drei der sechs Bilder durchgeführt. Hierbei konnte festgestellt werden, dass eine Änderung der Serverkonfiguration eine geringfügige Wirkung auf die Verarbeitungsgeschwindigkeit hat. Dies ist ein weiterer Grund, weshalb die leistungsstärkere Serverinstanz nur für wenige Messreihen kurzzeitig verwendet wurde. Zum Vergleich wurden im Diagramm 13 ebenfalls die Werte der CLI-Messungen auf dem MacBook dargestellt. Die geringen Unterschiede im Zuge des Serverupgrades sind vermutlich auf die Eigenschaften und Funktionsweisen der ImageMagick-Bibliothek zurückzuführen.

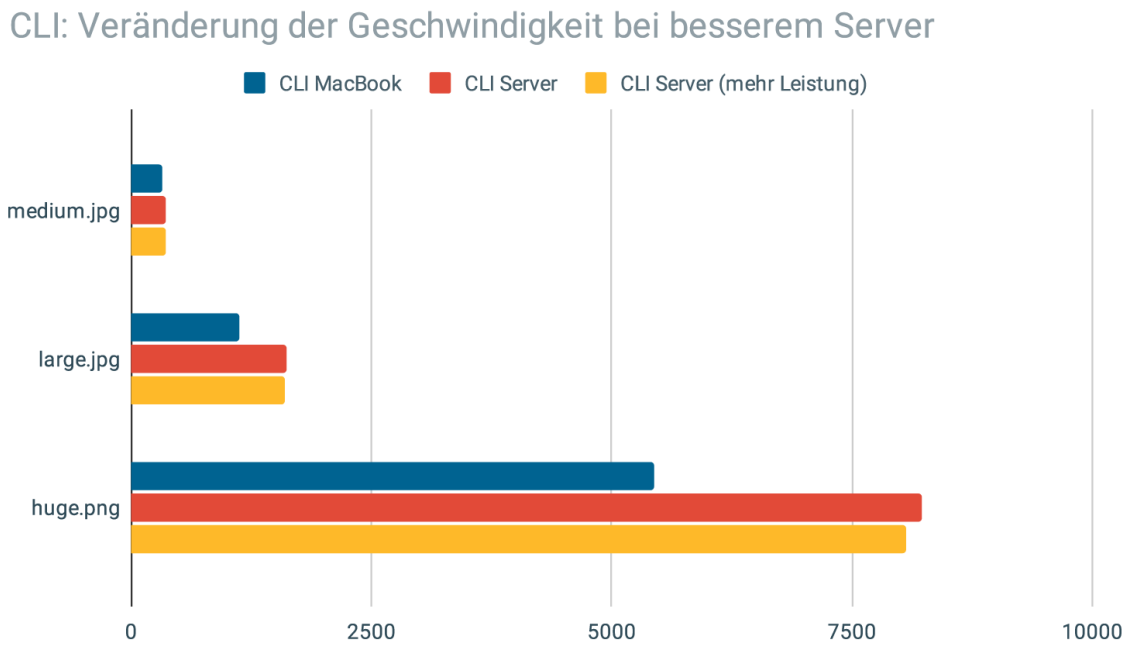

Diagramm 13: Unterschied der Serverkonfiguration bei Tests über die Kommandozeile

8. Diskussion

Abschließend sollen die gewonnenen Erkenntnisse kritisch betrachtet und die eigene Vorgehensweise rückblickend evaluiert werden. Hierdurch werden Verbesserungsvorschläge formuliert, wie die Qualität der Fallstudie im Rahmen dieser Arbeit weiterentwickelt werden kann.

8.1. Methodenkritik

Obwohl in der Durchführung der Messungen stark darauf geachtet wurde, die Testergebnisse möglichst unverfälscht von äußeren Faktoren aufzuzeichnen, konnte nicht hundertprozentig sichergestellt werden, dass die gemessenen Ergebnisse frei von anderweitigen Beeinflussungen waren. Allem voran muss erwähnt werden, dass die Testergebnisse nicht in perfekten Laborbedingungen entstanden sind. Es wurde zwar sichergestellt, dass auf den Geräten keine anderen Programme zur gleichen Zeit geöffnet waren, dennoch ist eine etwaige, temporäre Prozessorbeanspruchung durch Hintergrundprozesse nicht auszuschließen.

Gleiches gilt für die Tests der serverseitigen Vergleichsanwendung, da es bei dieser Art von Tests, bei denen eine Netzwerkverbindung über das Internet aufgebaut wird, viele Einflüsse gibt, welche die Ergebnisse beeinflussen, beispielsweise wenn die Netzwerkanfragen mit anderen Anfragen konkurrieren. Messungen innerhalb einer Laborumgebung waren im Aufwandsrahmen dieser Arbeit nicht möglich. Stattdessen zeigen die Tests eine gewisse Praxisnähe zum Anwendungsfall in natürlichen Umgebungen, was wiederum als Vorteil gewertet werden kann.

Um möglichst genaue Ergebnisse zu erzielen, wäre es darüber hinaus von Vorteil gewesen, in allen Implementierungen die gleiche ImageMagick Hauptversion zu verwenden. Bei der Anwendung in WebAssembly wurde die neuere Version 7.1.x statt der Version 6.9.x verwendet, was erst nach der Durchführung der Tests auffiel (siehe Tabelle 4). Im Zuge der Weiterentwicklung der ImageMagick-Bibliothek wurde besonders auf die Unterstützung neuer Dateiformate sowie auf HDR-Bildverarbeitung Wert gelegt (ImageMagick Studio LLC, 2021). Somit liegen zwar einerseits Unterschiede in den Softwarebibliotheken vor, andererseits beziehen sich diese jedoch besonders auf spezifische Anwendungsfälle. Da in der Testanwendung nur die sehr elementare Funktion der Rotation eines Bildes zur Anwendung kam, hatten die Versionsunterschiede mit großer Wahrscheinlichkeit keine spürbar großen Einflüsse auf die Messergebnisse.

Implementierung	ImageMagick Version
WebAssembly Binary	ImageMagick 7.1.00-52 Q8 x86_64
Node.js Anwendung	ImageMagick 6.9.10-23 Q16 x86_64
CLI auf Server	ImageMagick 6.9.10-23 Q16 x86_64
CLI auf MacBook	ImageMagick 6.9.12-72 Q16 x86_64

Tabelle 4: ImageMagick Versionen der einzelnen Implementierungen

Des Weiteren sollte mit aufgeführt werden, dass die Vergleiche zwischen Mobil- und Desktopbrowsern an manchen Stellen weniger aussagekräftig sind, da jeweils verschiedene Browserversionen zum Einsatz kamen. Daraus resultieren gegebenenfalls leicht unterschiedliche Implementierungen von WebAssembly und somit wiederum potentiell abweichende Ergebnisse in den Tests.

8.2. Verbesserungsvorschläge

Die Ergebnisse des letzten Kapitels und deren graphischer Darstellung in den Diagrammen zeigen deutlich die Bedeutung von WebAssembly. Für zukünftige Testverfahren dieser Art lassen sich dennoch folgende Verbesserungsvorschläge anbringen.

Da die serverseitigen Tests nur in drei Browsern auf dem MacBook durchgeführt wurden, sind die Ergebnisse in dieser Umgebung tatsächlich nicht so umfassend. Eine Verbesserung kann erreicht werden, indem die serverseitigen Tests in allen Umgebungen und auf allen Geräten durchgeführt werden. Nur dadurch hätte wirklich sichergestellt werden können, dass der Faktor der Umgebung bei der serverseitigen Vergleichsanwendung einen eher geringeren Einfluss auf die Ergebnisse hat.

In gleicher Weise wurde die leistungsfähigere Serverinstanz nur für die CLI-Tests verwendet. Ein breiter angelegter Test, der auch die serverseitige Anwendung mit einbezieht, hätte hier gegebenenfalls Geschwindigkeitsverbesserungen aufzeigen können, falls der effizientere Server in der Lage gewesen wäre, die Netzwerkanfragen schneller zu verarbeiten.

Zur Testanwendung selbst ist zu sagen, dass die alleinige Verwendung des Rotationsbefehls der ImageMagick-Bibliothek als ausschlaggebender Benchmark-Faktor vielleicht etwas zu einfach gewählt ist. Im Bereich der Bildverarbeitung bietet ImageMagick eine Vielzahl von komplexeren Manipulations- und Konvertierungsfunktionen, die ebenfalls hätten getestet werden können, um so die Effektivität von WebAssembly in diesem Gebiet zu untersuchen.

9. Zusammenfassung und Ausblick

Das enorme Potenzial von WebAssembly ist unbestritten. Sowohl im Frontend zur Erstellung von clientseitigen Webanwendungen als auch zunehmend im Backend-Bereich gewinnt die Technologie mehr und mehr an Bedeutung. Die im Rahmen dieser Bachelorarbeit durchgeführten Tests konnten erfolgreich demonstrieren, dass der Einsatz von WebAssembly bei Aufgaben traditioneller, serverseitiger Anwendungen sinnvoll sein kann.

9.1. Erkenntnisse und Fazit

WebAssembly hat es geschafft, als vierte Websprache offiziell akzeptiert, standardisiert und flächendeckend in Browsern implementiert zu werden (W3C, 2019). Dadurch ermöglicht es die Entwicklung von hochleistungsfähigen, modernen Webanwendungen wie nie zuvor und kombiniert die Produktivität von plattformübergreifender Entwicklung, die Leistung nativer Anwendungen und die Offenheit des Webs in einer Codebasis. Das bis dato im Browser vorherrschende JavaScript konnte dies alleine nicht bieten.

Im Praxisteil dieser Arbeit konnte herausgearbeitet werden, dass im Anwendungsfall der Bilddatenverarbeitung die Verwendung von WebAssembly sinnvoll ist, da hier bessere Ergebnisse erzielt werden können, als bei einer konventionellen Umsetzung der Beispielanwendung. So konnten in den Untersuchungen bis zu 8x schnellere Laufzeiten beim Einsatz von WebAssembly nachgewiesen werden. Im Test sind außerdem Browserunterschiede in der Ausführungsgeschwindigkeit von WebAssembly aufgefallen, wobei Firefox die beste und Safari die schlechteste Performance zeigte. Weiterhin wurde ersichtlich, dass die Technologie genauso gut auf moderneren Mobilgeräten performant nutzbar ist und somit beispielsweise auch für den Bereich der PWAs relevant ist.

Es wurde festgestellt, dass die Leistung der entwickelten WebAssembly-Anwendung in den Vergleichstests circa 37% langsamer ausfällt als die native Ausführung der Beispielanwendung. Dies mag zunächst ernüchternd klingen, da WebAssembly besonders damit beworben wird, annähernd native Geschwindigkeiten im Browser zu ermöglichen. Bei der Bewertung der Ergebnisse sollte jedoch beachtet werden, dass die der Arbeit zugrundeliegende Beispielanwendung ein gewisses Optimierungspotential hat und in dieser Weise vielleicht nicht allen Anwendungsfällen entspricht. Zudem können die getesteten Geschwindigkeiten der WebAssembly Anwendung unter normalen Umständen nicht mit einer serverseitig implementierten Anwendung erreicht werden, was allein deshalb schon als Entscheidungskriterium für die Verwendung von WebAssembly angesehen werden kann. Ob der Einsatz der Technologie für ein bestimmtes Projekt letztlich sinnvoll ist, sollte generell im Vorfeld gründlich geprüft werden.

9.2. Vorschläge für zukünftige Arbeiten

Um das Thema WebAssembly weiterführend im Detail zu studieren, ist es ratsam, zusätzliche Untersuchungen anzustellen und Messungen durchzuführen. Da sich diese Studie nur mit der Bildverarbeitungsbibliothek ImageMagick beschäftigte, wäre es interessant, einen eigenen Bildverarbeitungscode in verschiedenen Sprachen zu entwickeln, um umfassendere Ergebnisse zu erhalten. Auch auf die Unterschiede zwischen verschiedenen Umsetzungsmöglichkeiten clientseitiger Webanwendungen wurde in dieser Arbeit nicht eingegangen. Die Entwicklung einer Vergleichsanwendung in JavaScript könnte somit weitere wichtige Ergebnisse liefern.

Abseits von der Integration von WebAssembly in Webanwendungen kann dieses natürlich auch im Serverbereich Einsatz finden. Die Möglichkeiten für Untersuchungen in diesem Gebiet sind groß, besonders weil zu diesem Thema bisher noch wenig Studien durchgeführt wurden.

9.3. Ausblick in die Zukunft

WebAssembly eröffnet einerseits viele Möglichkeiten zur Entwicklung von Webanwendungen, was mit einer hohen Flexibilität einhergeht. Andererseits bringt es auch ein riesiges Zukunftspotential mit sich, da es durch die Entwicklung von WASI nicht mehr nur auf die Ausführung im Browser begrenzt ist. Mittlerweile können deshalb auch Backend-Anwendungen in WebAssembly realisiert werden, was den Serverless-Markt revolutioniert. Genau an dieser Stelle wird es in Zukunft höchstwahrscheinlich die größten Weiterentwicklungen geben.

Die offizielle Präsentation der neu verfügbaren Garbage Collection in WebAssembly auf der diesjährigen Entwicklerkonferenz von Google zeigte ebenfalls eindrücklich, wie sich die Technologie immer weiter entwickelt und verbessert (Google I/O, Mai 2023). Da nun endlich auch Sprachen wie Java, Kotlin oder Dart nach WebAssembly kompiliert werden können, eröffnet sich ein völlig neuer Markt für den Einsatz der Technologie: mobile Cross-Platform Apps können nun noch schnellere und effizientere Webanwendungen bereitstellen.

Schlussendlich ist zu sagen, dass trotz des zweifellos hohen Potenzials von WebAssembly bisher vor allem große Unternehmen in speziellen Anwendungsfällen davon Gebrauch machen. Obwohl ein allgemeines Interesse an der Technologie besteht, wird es von vielen Entwicklern bisher noch nicht in vollem Umfang genutzt. Es bleibt zu hoffen, dass wenn mehr Unternehmen eine Infrastruktur für die Entwicklung von WebAssembly-Anwendungen zur Verfügung stellen, auch die Akzeptanz und Verwendung von WebAssembly in allen Bereichen zunehmen wird, insbesondere wenn es um Serverless, IoT, oder Edge Computing geht. Ob und inwieweit die Technologie dann eine allgegenwärtige Verbreitung erreichen wird, bleibt abzuwarten.

Literaturverzeichnis

Buchquellen

Battagline, R. (2021). The Art of WebAssembly: Build Secure, Portable, High-Performance Applications. National Geographic Books.

Burd, B. (2017). Java For Dummies. John Wiley & Sons.

Eberhardt, C. (2020). What Is WebAssembly?

Gallant, G. (2019). WebAssembly in Action. Manning.

Sletten, B. (2021). WebAssembly: the Definitive Guide: Safe, Fast, and Portable Code. O'Reilly Media.

Fachliteratur

Casteleyn, S., Garrigós, I. & Mazón, J. (2014). Ten Years of Rich Internet Applications. ACM Transactions on The Web, 8(3), 1–46. https://doi.org/10.1145/2626369

Fredriksson, S. (2020). WebAssembly vs. its predecessors - A comparison of technologies [Bachelorarbeit]. Linnaeus University. http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-97654

Haas, A. F., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M. W., Gohman, D., Wagner, L., Zakai, A. & Bastien, J. (2017). Bringing the web up to speed with WebAssembly. Programming Language Design and Implementation. https://doi.org/10.1145/3062341.3062363

Taivalsaari, A., Mikkonen, T., Ingalls, D. H. H. & Palacz, K. (2008). Web Browser as an Application Platform. Proceedings of the . . . EUROMICRO Conference. https://doi.org/10.1109/seaa.2008.17

Wirfs-Brock, A. & Eich, B. (2020). JavaScript: the first 20 years. Proceedings of the ACM on programming languages, 4(HOPL), 1–189. https://doi.org/10.1145/3386327

Online-Quellen

Adobe. (2008, 8. Juli). Adobe - Flash Player Statistics. Abgerufen am 20. März 2023 https://web.archive.org/web/20080730040225/adobe.com/products/player_census/flashplayer/

Adobe. (2021, Januar). Adobe Flash Player End of Life. Abgerufen am 15. März 2023 https://www.adobe.com/products/flashplayer/end-of-life.html

Akinyemi, S. (2023). Awesome WebAssembly Languages. GitHub. https://github.com/appcypher/awesome-wasm-langs/blob/master/README.md

Allaire, J. (2002, März). Macromedia Flash MX, Abgerufen am 14. März 2023 http://static1.1.sqsp-cdn.com/static/f/115864/862648/1181454829823/Macromedia+rich+client+Allaire+2002.pdf

Carey, S. (2022, 28. Februar). The rise of WebAssembly. InfoWorld. https://www.infoworld.com/article/3651503/the-rise-of-webassembly.html

Cheung, K. (2018, 26. September). AutoCAD & WebAssembly: Moving a 30 Year Code Base to the Web. InfoQ. https://www.infoq.com/presentations/autocad-webassembly/

Chrome Platform Status. (2023, 7. April). Chrome Platform Status. Abgerufen am 2. Mai 2023 https://chromestatus.com/feature/6062715726462976

Clark, L. (2017, 13. Juni). What makes WebAssembly fast? - Mozilla Hacks - the Web developer blog. Mozilla Hacks – the Web developer blog. https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/

Clark, L. (2019, 9. August). Standardizing WASI: A system interface to run WebAssembly outside the web - Mozilla Hacks - the Web developer blog. Mozilla Hacks – the Web developer blog. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

Coosner, L. (2022, 8. Juni). The Life and Death of Java Applets. Incus Data Programming Courses. https://incusdata.com/blog/life-and-death-of-java-applets

Dawelbeit, O. (2014, 23. März). Single Page Apps - A new concept or a revived old one? IT with Passion. Abgerufen am 14. März 2023 https://omerio.com/2014/03/23/single-page-apps-a-bleeding-edge-new-concept-or-a-revived-old-one/

Docker. (2023, 20. März). Docker overview. Docker Documentation. https://docs.docker.com/get-started/overview/

Dodson, R. (2020, 2. November). NaCl and PNaCl - Chrome Developers. Chrome Developers. Abgerufen am 20. März 2023 https://developer.chrome.com/docs/native-client/nacl-and-pnacl/

eBay. (2019, 31. Juli). WebAssembly at eBay: A Real-World Use Case. https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case

Eberhardt, C. (2019, 17. Mai). Build your own WebAssembly Compiler. Scott Logic Blog. Abgerufen am 22. März 2023 https://blog.scottlogic.com/2019/05/17/webassembly-compiler.html

Ecma International. (2022, 13. Juni). ECMAScript® 2022 Language Specification. TC39. Abgerufen am 22. März 2023 https://262.ecma-international.org

Eich, B. (2015, 17. Juni). From ASM.JS to WebAssembly – Brendan Eich. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/

Emscripten. (2023). Emscripten Tutorial. Emscripten.org. Abgerufen am 31. März 2023 https://emscripten.org/docs/getting_started/Tutorial.html

ExpressJS. (2023). npm: multer. npm. Abgerufen am 16. Mai 2023 https://www.npmjs.com/package/multer

Fermyon Inc. (2006, 11. Januar). WebAssembly Language Support Matrix. Fermyon • Experience the next wave of cloud computing. https://www.fermyon.com/wasm-languages/webassembly-language-support

Figma. (2017, 8. Juni). Figma is powered by WebAssembly. https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/

Google. (2017, 30. Mai). Goodbye PNaCl, Hello WebAssembly! Chromium Blog. https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html

Google. (2019, 20. Juni). WebAssembly brings Google Earth to more browsers. Chromium Blog. https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html

Google. (2023). Flash Player, Google Trends 2004 - present. Google Trends. Abgerufen am 15. März 2023. https://trends.google.de/trends/explore?date=all&q=Flash%20player

Google I/O. (2023, 10. Mai). WebAssembly: a new development paradigm for the web. Abgerufen am 24. Mai 2023 https://io.google/2023/program/1d176349-7cf8-4b51-b816-a90fc9d7d479/

Hallie, L. (2020, 19. Januar) JavaScript Visualized: JavaScript Engine. DEV Community. Abgerufen am 27. März 2023 https://dev.to/lydiahallie/javascript-visualized-the-javascript-engine-4cdf

Heckmann, A. (2023). gm: GraphicsMagick and ImageMagick for node. npm. Abgerufen am 16. Mai 2023 https://www.npmjs.com/package/gm

Herman, D., Wagner, L. & Sakai, A. (2014, 18. August). asm.js. Abgerufen am 20. März 2023 http://asmjs.org/spec/latest/

Hoffmann, J. (2017, 7. August). Flash And Its History On The Web - The History of the Web. The History of the Web. https://thehistoryoftheweb.com/the-story-of-flash/

Hykes, S. (2019, 27. März). If WASM existed in 2008, we wouldn't have needed to created Docker... Twitter. Abgerufen am 11. März 2023 https://twitter.com/solomonstre/status/1111004913222324225

ImageMagick Studio LLC. (2015, Oktober). ImageMagick - Rotate. ImageMagick. Abgerufen am 10. Mai 2023 https://imagemagick.org/script/command-line-options.php#rotate

ImageMagick Studio LLC. (2021). ImageMagick. ImageMagick. Abgerufen am 16. Mai 2023 https://imagemagick.org/script/porting.php

ImageMagick Studio LLC. (2023, 27. Februar). ImageMagick. ImageMagick. Abgerufen am 10. Mai 2023 https://imagemagick.org/

Jobs, S. (2010, April). Thoughts on Flash. Apple. https://web.archive.org/web/20100501010616/apple.com/hotnews/thoughts-on-flash/

Joslyn, H. (2022, 10. November). The Latest Milestones on WebAssembly's Road to Maturity. The New Stack. https://thenewstack.io/the-latest-milestones-on-webassemblys-road-to-maturity/

JS API - WebAssembly. (2023). WebAssembly.org. Abgerufen am 26. April 2023 https://webassembly.org/getting-started/js-api/

Klabnik, S. (2018, 25. August). Is WebAssembly the return of Java Applets & Flash? Abgerufen am 31. März 2023 https://steveklabnik.com/writing/is-webassembly-the-return-of-java-applets-flash

Kosaka, M. (2018, 5. September). Inside look at modern web browser (part 1) - Chrome Developers. Chrome Developers. https://developer.chrome.com/blog/inside-browser-part1/

Lemstra, D. (2023). imagemagick/magick-wasm. npm. Abgerufen am 15. Mai 2023 https://www.npmjs.com/package/@imagemagick/magick-wasm

Matei. (2023, 22. März). Spin 1.0 — The Developer Tool for Serverless WebAssembly. Fermyon • Experience the next wave of cloud computing. https://www.fermyon.com/blog/introducing-spin-v1

McConnell, J. (2017, 13. November). WebAssembly support now shipping in all major browsers. The Mozilla Blog. Abgerufen am 8. März 2023 https://blog.mozilla.org/en/mozilla/webassembly-in-browsers/

MDN Web Docs. (2023a, Februar 23). Loading and running WebAssembly code - WebAssembly | MDN. Abgerufen am 26. April 2023 https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running

MDN Web Docs. (2023b, März 23). WebAssembly Concepts - WebAssembly | MDN. Abgerufen am 31. März 2023 https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

Metz, C. (2011, 16. September). Google's Native Client goes live in Chrome. The Register. Abgerufen am 20. März 2023 https://www.theregister.com/2011/09/16/native_client_debuts_in_chrome/

Microsoft. (1996, 12. März). Microsoft Announces ActiveX Technologies. Microsoft News. https://news.microsoft.com/1996/03/12/microsoft-announces-activex-technologies/

Microsoft Azure. (2020, 24. September). Compute-Benchmarkergebnisse für virtuelle Azure Linux-Computer - Azure Virtual Machines. Microsoft Learn. Abgerufen am 23. Mai 2023 https://learn.microsoft.com/de-de/azure/virtual-machines/linux/compute-benchmark-scores

Microsoft Azure. (2022a, September 6). B-series burstable - Azure Virtual Machines. Microsoft Learn. Abgerufen am 22. Mai 2023 https://learn.microsoft.com/en-gb/azure/virtual-machines/sizes-b-series-burstable

Microsoft Azure. (2022b, Dezember 21). Memory optimized Dv2 and Dsv2-series VMs - Azure Virtual Machines. Microsoft Learn. Abgerufen am 22. Mai 2023 https://learn.microsoft.com/en-gb/azure/virtual-machines/dv2-dsv2-series-memory

Microsoft Edge Team. (2015, 6. Mai). A break from the past, part 2: Saying good-bye to ActiveX Windows Blogs. Abgerufen am 20. März 2023 https://blogs.windows.com/msedgedev/2015/05/06/a-break-from-the-past-part-2-saying-goodbye-to-activex-vbscript-attachevent/

Mihajlija, M. (2019, 8. November). WebAssembly: How and why - LogRocket Blog. LogRocket Blog. https://blog.logrocket.com/webassembly-how-and-why-559b7f96cd71/

Moore, K. (2023, 14. Februar). Roadmap - WebAssembly. WebAssembly. Abgerufen am 21. März 2023 https://webassembly.org/roadmap/

Mozilla. (2023, 21. Februar). JavaScript language overview - JavaScript | MDN. MDN. Abgerufen am 24. März 2023 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_overview

Nattestad, T. (2021, 26. Oktober). Photoshop's journey to the web. web.dev. https://web.dev/ps-on-the-web/

Oracle. (2022). Java Applets. Java. https://www.java.com/de/download/help/enable_browser.html

Oracle Corporation. (2016, Januar). Migrating from Java Applets to plugin-free Java technologies. Oracle. Abgerufen am 20. März 2023 https://www.oracle.com/technetwork/java/javase/migratingfromapplets-2872444.pdf

Osaka, M. & Chrome Developers. (2018, 20. September). Inside look at modern web browser (part 3) - Chrome Developers. Chrome Developers. Abgerufen am 27. März 2023 https://developer.chrome.com/blog/inside-browser-part3/

Resig, J. (2013, 3. April). Asm.js: The JavaScript Compile Target. Abgerufen am 20. März 2023 https://johnresig.com/blog/asmjs-javascript-compile-target/

Roadmap - WebAssembly. (2023, 14. Februar). Abgerufen am 2. Mai 2023 https://webassembly.org/roadmap/

Schiemann, D. (2020, 18. August). Zoom on Web: WebAssembly SIMD, WebTransport, and WebCodecs. InfoQ. https://www.infoq.com/news/2020/08/zoom-web-chrome-apis/

Sheldon, R. & White, C. (2021, 21. Dezember). ActiveX controls. Enterprise Desktop. Abgerufen am 20. März 2023 https://www.techtarget.com/searchenterprisedesktop/definition/ActiveX-control

Statista. (2023, 16. Mai). Durchschnittliche Internetgeschwindigkeit in Deutschland bis April 2023. Abgerufen am 22. Mai 2023 https://de.statista.com/statistik/daten/studie/416534/umfrage/durchschnittliche-internetgeschwindigkeit-in-deutschland/

Surma. (2019, 14. Februar). Replacing a hot path in your app's JavaScript with WebAssembly - Chrome Developers. Chrome Developers. https://developer.chrome.com/blog/hotpath-with-wasm/

Use Cases - WebAssembly. (2020, August). https://webassembly.org/docs/use-cases/

V8 Project & Google Inc. (2022, 22. Dezember). WebAssembly compilation pipeline ·V8. v8.dev JavaScript Engine. Abgerufen am 26. April 2023 https://v8.dev/docs/wasm-compilation-pipeline

Vite. (2023). Vite. Next Generation Frontend Tooling. Abgerufen am 15. Mai 2023 https://vitejs.dev/

W3C. (2019, 5. Dezember). World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation. w3.org. Abgerufen am 13. März 2023, von https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en

Wagner, L. (2015, 17. Juni). WebAssembly | Luke Wagner's Blog. https://blog.mozilla.org/luke/2015/06/17/webassembly/

Wagner, L. (2017, 21. November). WebAssembly Will Finally Let You Run High-Performance Applications in Your Browser. IEEE Spectrum. Abgerufen am 15. März 2023

https://spectrum.ieee.org/webassembly-will-finally-let-you-run-highperformance-applications-in-your-browser

Wagner, L. (2020, 16. Juni). FAQ - WebAssembly. Abgerufen am 20. März 2023 https://webassembly.org/docs/faq/

Walker, S. (2021, 30. November). WebAssembly: The 4th Official Language of the Web. Abgerufen am 15. März 2023 https://www.cognizantsoftvision.com/blog/webassembly-the-4th-official-language-of-the-web/

WebAssembly Community Group. (2023, 6. März). Introduction — WebAssembly 2.0. https://webassembly.github.io/spec/core/intro/introduction.html

WebAssembly Proposals. (2023). GitHub - WebAssembly/proposals: Tracking WebAssembly proposals. Abgerufen am 3. Mai 2023 https://github.com/WebAssembly/proposals

Selbstständigkeitserklärung

Hiermit erkläre ich an Eides statt, dass ich die vorgelegte Bachelorarbeit selbständig angefertigt und nur die im Quellenverzeichnis enthaltenen veröffentlichten oder unveröffentlichten Schriften sowie persönlichen Mitteilungen verwendet habe.

Sämtliche Textstellen, die wörtlich oder sinngemäß aus der Literatur übernommen wurden, und Angaben, die auf mündlichen Auskünften beruhen, sowie von anderen Personen bereitgestellte Materialien oder erbrachte Dienstleistungen sind als solche gekennzeichnet. Die vorgelegte Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Leipzig, den 29. Mai 2023 — Samuel Schorr

Ort, Datum — Unterschrift

Danksagung

Sehr geehrter Professor Bleymehl, mit großer Dankbarkeit möchte ich Ihnen meine aufrichtige Anerkennung für Ihre unermüdliche Betreuung und fachliche Führung während meiner Bachelorarbeit ausdrücken. Ihre immense Kenntnis, Ihr inspirierender Geist und Ihre ermutigenden Worte waren von unschätzbarem Wert für meinen akademischen Fortschritt. Sie haben mich herausgefordert, meine Grenzen zu erweitern, und ich bin Ihnen zutiefst dankbar für Ihre Geduld, Ihre Bereitschaft zur Zusammenarbeit und Ihre unerschütterliche Unterstützung.

Meiner lieben Frau Katharina möchte ich einen ganz besonderen Dank aussprechen. Deine bedingungslose Liebe, ermutigende Worte und unermüdliche Unterstützung waren mein stärkster Antrieb während dieser intensiven akademischen Phase. Du hast mir immer den Rücken gestärkt, warst für mich da und hast mich dazu ermutigt, die Bachelorarbeit zu Ende zu bringen. Deine Opferbereitschaft und deine Geduld sind für mich unermesslich wertvoll. Ohne dich an meiner Seite wäre diese Arbeit nicht möglich gewesen.

Meinen Freunden Theresia, Conrad, Jonas und Michel möchte ich ebenfalls Danke sagen. Ihr wart eine essentielle Unterstützung bei meiner Arbeit. Durch eure sorgfältige Korrektur und wertvolle Rückmeldungen habt ihr dazu beigetragen, dass meine Arbeit ihre höchste Qualität erreicht hat. Eure Hingabe, eure Zeit und eure Kompetenz haben mir geholfen, die Fehler und Problemstellen der Arbeit zu beheben.

Meinen Eltern und meiner Familie Lydia, Aaron, Antonia, Jonathan und Phoebe, sowie meinen weiteren Freunden Vanessa, Annalisa, Carsten und Judith möchte ich von ganzem Herzen danken. Ihr habt mich durch eure ermutigenden Worte, euer aufrichtiges Interesse und eure moralische Unterstützung motiviert und gestärkt. Euer Zuspruch hat mir den Mut gegeben, mich der Herausforderung zu stellen.

Der größte Dank gebührt jedoch Jesus ganz allein. Er hat mich durch diese Zeit des Studiums bis zur Bachelorarbeit hindurch getragen und das Unmögliche möglich gemacht. Ihm verdanke ich die Kraft jedes einzelnen Tages und die Konzentration, die Arbeit in ihrer Gesamtheit zu verfassen. Ohne Jesus, believe it or not, hätte ich diese Arbeit nicht schreiben können. Alle Ehre meinem besten Freund und Retter, König Jesus. #solideogloria