Hochschule Karlsruhe
University of
Applied Sciences

HKA

Evaluation von Bun 1.0: Eine Verbesserung der JavaScript-Laufzeitumgebung?

Seminararbeit von
Ansgar Lichter

an der Fakultät für Informatik und Wirtschaftsinformatik

Universität:	Hochschule Karlsruhe
Studiengang:	Informatik
Professor:	Prof. Dr.-Ing. Vogelsang
Bearbeitungszeitraum:	01.10.2023 - 04.12.2023

Inhaltsverzeichnis

Abbildungsverzeichnis **iv**

Tabellenverzeichnis **v**

Quellcodeverzeichnis **vi**

Abkürzungsverzeichnis **vii**

1 Einleitung **1**

1.1 Motivation 1
1.2 Zielsetzung 2
1.3 Aufbau der Arbeit 3

2 Grundlagen **4**

2.1 Node.js 4
2.2 Bun 7
2.3 Performance als Qualitätsattribut 9

3 Performanceanalyse **12**

3.1 Vorgehensweise 12
3.2 Versuchsaufbau 13
3.3 Implementierungen 14
3.3.1 HTTP-Server 15
3.3.2 Datei-Server 16
3.3.3 Berechnung der Fibonacci-Folge 17
3.4 Ergebnisse 18
3.5 Fazit 23

4 Kompatibilität von Projekten **25**

4.1 Express 25
4.2 Nest 28
4.3 Fazit 30

5 Schlussbetrachtung **31**
5.1 Fazit 31
5.2 Ausblick 32

Literaturverzeichnis **33**

A Ergebnisse des Benchmarks **37**
A.1 HTTP-Server 37
A.2 Datei-Server 41
A.3 Berechnung der Fibonacci-Folge 50
A.4 Zusätzliche Diagramme 54

Abbildungsverzeichnis

1.1 Nutzungsstatistik von JavaScript-Laufzeitumgebungen 2

2.1 Node.js Architektur 4
2.2 Vergleich der Ökosysteme von Bun und Node.js 8
2.3 Qualitätsattribute einer Software 10

3.1 HTTP-Server - Durchschnittliche Anzahl an Anfragen pro Sekunde . 19
3.2 Datei-Server - Durchschnittliche Latenz 20
3.3 Datei-Server - Maximal verwendeter Arbeitsspeicher 21
3.4 Datei-Server - CPU-Auslastung 22
3.5 Berechnung der Fibonacci-Folge - Ausführungszeit 22

A.1 HTTP-Server - Durchschnittliche Latenz 54
A.2 Berechnung der Fibonacci-Folge - CPU-Auslastung 54
A.3 Berechnung der Fibonacci-Folge - RAM-Nutzung 55

Tabellenverzeichnis

3.1 Hardware für die Performanceanalyse 14

A.1 HTTP-Server - Ergebnisse von Bun auf macOS 37
A.2 HTTP-Server - Ergebnisse von Node.js LTS auf macOS 38
A.3 HTTP-Server - Ergebnisse von Node.js Latest auf macOS 38
A.4 HTTP-Server - Ergebnisse von Bun auf Ubuntu 23.10 39
A.5 HTTP-Server - Ergebnisse von Node.js LTS auf Ubuntu 23.10 39
A.6 HTTP-Server - Ergebnisse von Node.js Latest auf Ubuntu 23.10 . . . 40
A.7 Datei-Server - Ergebnisse von Bun auf Ubuntu 23.10 41
A.8 Datei-Server - Ergebnisse von Node.js LTS auf Ubuntu 23.10 42
A.9 Datei-Server - Ergebnisse von Node.js Latest auf Ubuntu 23.10 44
A.10 Datei-Server - Ergebnisse von Bun auf macOS 46
A.11 Datei-Server - Ergebnisse von Node.js LTS auf macOS 47
A.12 Datei-Server - Ergebnisse von Node.js Latest auf macOS 49
A.13 Berechnung der Fibonacci-Folge - Ergebnisse auf macOS 51
A.14 Berechnung der Fibonacci-Folge - Ergebnisse auf Ubuntu 23.10 52

Quellcodeverzeichnis

3.1 HTTP-Server Bun 15
3.2 HTTP-Server Node.js 15
3.3 Bombardier HTTP-Server 15
3.4 CPU- und RAM-Messung auf Ubuntu 15
3.5 CPU- und RAM-Messung auf macOS 16
3.6 Datei-Server Bun 16
3.7 Datei-Server Node.js 16
3.8 Bombardier Datei-Server 17
3.9 Berechnung der Fibonacci-Folge 17
3.10 Messung der Fibonacci-Folge auf Ubuntu 18

Abkürzungsverzeichnis

API Application Programming Interface

CRUD Create, Read, Update, Delete

JIT-Compiler Just-in-Time-Compiler

LTS Long Term Support

NPM Node Package Manager

ORM Object-Relational Mapping

Kapitel 1

Einleitung

Dieses Kapitel führt in die Thematik ein und motiviert, warum Bun evaluiert wird. Darauf folgt die Vorstellung der Ziele und des Aufbaus dieser Arbeit.

1.1 Motivation

JavaScript ist aktuell eine beliebte Programmiersprache. In einer Umfrage unter Entwicklern auf Stack Overflow, an der mehr als 89.000 Entwickler teilgenommen haben, wurde JavaScript zum elften Jahr in Folge als die am häufigsten verwendete Programmiersprache identifiziert. Mehr als 63% der befragten Entwickler haben JavaScript als ihre favorisierte Technologie angegeben. Unter den professionellen Entwicklern liegt der Anteil mit 65% noch höher. Zusätzlich ist TypeScript, eine stark typisierte Programmiersprache, die auf JavaScript aufbaut, ebenfalls beliebt [1]. Etwa 39% aller Entwickler und ungefähr 44% der professionellen Entwickler nutzen TypeScript. Somit ist TypeScript die viertbeliebteste Programmiersprache. Dies unterstreicht die hohe Relevanz des JavaScript-Ökosystems.[2]
Die Programmiersprache wird vor allem im Kontext der Web-Entwicklung verwendet [3]. Darüber hinaus nutzen etwa 3% der weltweit bekannten Server eine Laufzeitumgebung, die JavaScript ausführen kann [4]. Demnach wird die Programmiersprache nicht nur für die Entwicklung im Frontend, sondern auch im Backend eingesetzt.

Abbildung 1.1 zeigt, dass Node.js die am weitesten verbreitete Laufzeitumgebung ist. In der Umfrage zum Zustand von JavaScript gaben ca. 71% der 30.000 befragten Entwickler an, dass sie Node.js regelmäßig als Laufzeitumgebung verwenden [5]. Dennoch erscheinen immer wieder neue Laufzeitumgebungen, die versuchen, Node.js zu verdrängen. Die neuste Alternative ist Bun, das am 9. September 2023 in der Version 1.0 veröffentlicht wurde [6]. In der zuvor erwähnten Umfrage gaben

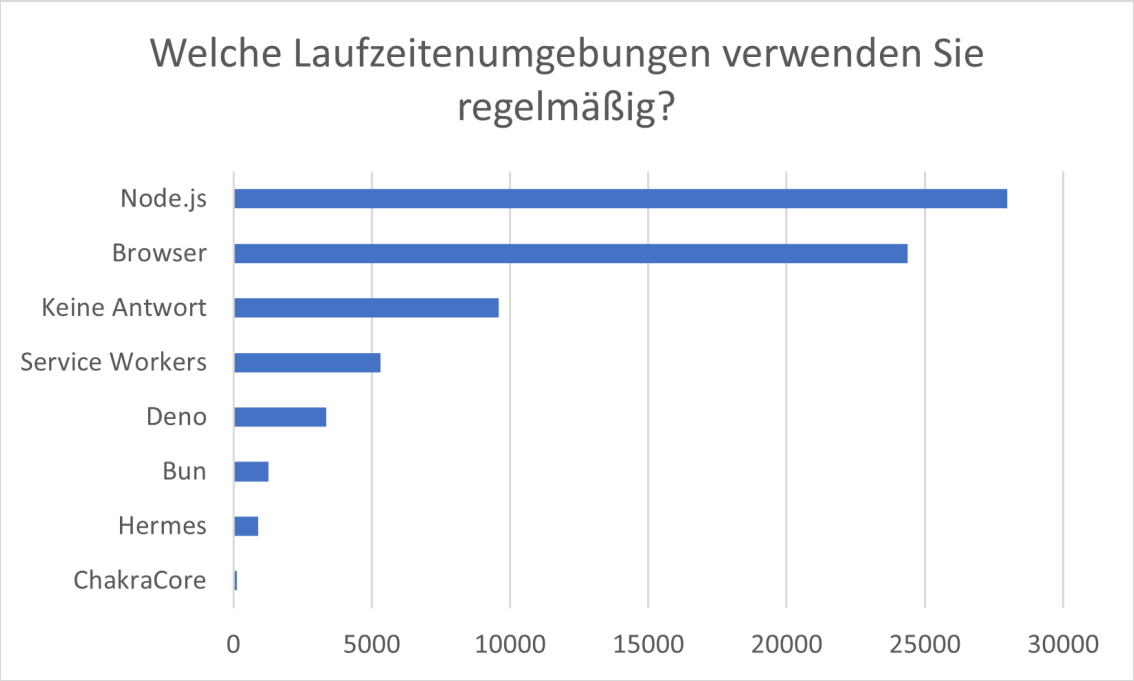

Abbildung 1.1: Nutzungsstatistik von JavaScript-Laufzeitumgebungen
Quelle: [5]

etwa 3% der Entwickler an, dass sie Bun als eine Alternative zu Node.js verwenden, obwohl sich Bun zu diesem Zeitpunkt noch im Entwicklungsstadium befunden hat [5].

Die Entwickler von Bun werben mit Features wie erheblicher Leistungssteigerung, eleganten Schnittstellen und einer angenehmen Entwicklererfahrung [7]. Die Laufzeitumgebung inkludiert einen integrierten Paketmanager, Bundler, Test-Bibliotheken und auch einen Transpiler zum Ausführen von TypeScript. Besitzt ein Projekt abhängige Pakete, können diese über den Paketmanager mit *bun install* installiert werden. Danach sorgt der Befehl *bun <Dateiname>* dafür, dass Bun die gewünschte Applikation unter Angabe des Dateinamens ausführt.[6]

1.2 Zielsetzung

Das Hauptziel dieser Arbeit besteht darin, die Version 1.0 von Bun einer eingehenden Evaluierung zu unterziehen. Konkret wird untersucht, ob die in den Ankündigungen versprochene signifikante Leistungssteigerung im Vergleich zu Node.js tatsächlich existiert und reproduzierbar ist. Darüber hinaus wird geprüft, inwiefern bestehende Projekte auf der Basis von Node.js mit Bun kompatibel sind. Die Ergebnisse dieser Arbeit können Entwicklern bei der Entscheidung helfen, ob sie auf Bun 1.0 migrieren sollten, und sie dabei unterstützen, die Leistung ihrer bestehenden Projekte zu

verbessern. Insgesamt zielt diese Untersuchung darauf ab, Klarheit über die Versprechungen von Bun 1.0 zu schaffen und Entwicklern fundierte Informationen für ihre Entscheidungsfindung zur Verfügung zu stellen. Dies spiegelt sich in den folgenden Leitfragen wider:

- Welche konkreten Leistungsverbesserungen können in Bun 1.0 im Vergleich zu Node.js festgestellt werden, und wie lassen sie sich quantifizieren?
- Inwiefern sind Node.js-Projekte kompatibel mit Bun? Wie schwierig gestaltet sich die Migration?
- Welche Herausforderungen und Vorteile ergeben sich bei der Verwendung von Bun 1.0 im Vergleich zu Node.js für Entwickler und Projekte?

1.3 Aufbau der Arbeit

Im folgenden Kapitel vermittelt die Arbeit alle notwendigen theoretischen Grundlagen, die zum Verständnis des Themas notwendig sind. Dafür werden Node.js und Bun detailliert betrachtet. Des Weiteren wird Performance als Qualitätsattribut von Software definiert, um Rückschlüsse für die Performanceanalyse zu ermöglichen.

Im dritten Kapitel werden die Performance von Bun und Node.js in ausgewählten Testszenarien verglichen. Hierzu werden zuerst die Vorgehensweise, der Versuchsaufbau und die Implementierungen vorgestellt. Im Anschluss daran folgt de die Präsentation der Ergebnisse und ein zusammenfassendes Fazit.

Das vierte Kapitel setzt den Fokus auf die Kompatibilität von Projekten auf Basis von Node.js. Die Betrachtung beschränkt sich auf die Frameworks Express und Nest. Um die Kompatibilität bewerten zu können, wird pro Framework eine Anwendung beispielhaft migriert. Die Ergebnisse werden in einem Fazit bewertet.

Das letzte Kapitel fasst die Kernaussagen der Arbeit zusammen und prüft sie vor dem Hintergrund der zuvor definierten Ziele. Abschließend wird in einem Ausblick dargelegt, welche ergänzenden Forschungsthemen verbleiben und welchen zukünftigen Nutzen die Ergebnisse mit sich bringen.

Kapitel 2

Grundlagen

Dieses Kapitel stellt die benötigten Grundlagen vor, die für das Verständnis der darauffolgenden Kapitel notwendig sind. Hierzu zählen die Vorstellung von Node.js und Bun sowie weitere Grundlagen zu Performanceanalysen.

2.1 Node.js

Node.js ist ein beliebtes Tool für eine Vielzahl an unterschiedlichen Projekten, darunter leichtgewichtige Webservices, dynamische Webanwendungen und Tools für die Kommandozeile. Es handelt sich um eine plattformunabhängige Open-Source-Laufzeitumgebung, die es ermöglicht, JavaScript außerhalb des Browsers auszuführen. Node.js verwendet die V8 JavaScript Engine von Google. Diese ist in C++ geschrieben und wird von Google Chrome verwendet. Dies ermöglicht Node.js eine hohe Performance, weshalb Unternehmen wie Netflix und Uber Node.js in ihren Softwareprojekten einsetzen.[8]

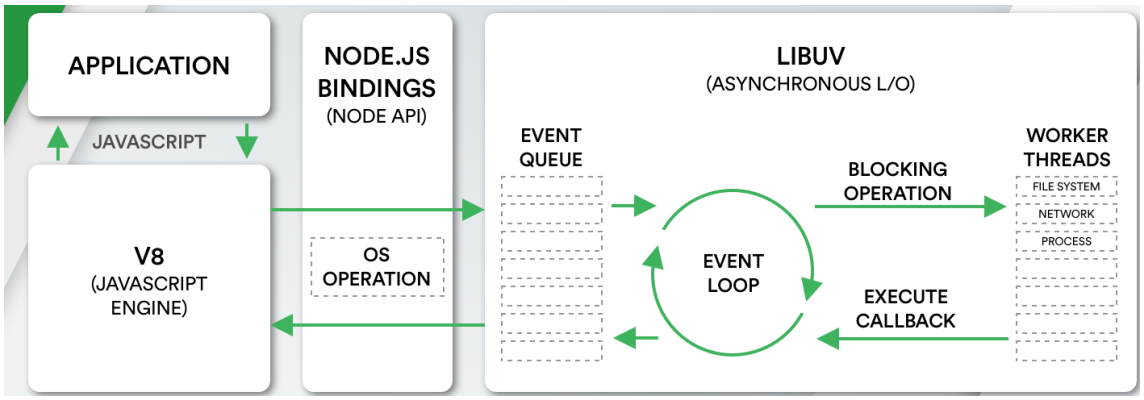

Abbildung 2.1: Node.js Architektur
Quelle: [9]

Wie in Abbildung 2.1 zu sehen, nutzt Node.js grundsätzlich nur einen Thread und erstellt nicht für jede neue Anfrage einen neuen Thread. Sobald eine Applikation gestartet wird, wird in dem einzigen Thread der Node.js-Prozess gestartet. Die V8 Engine optimiert den Maschinencode zusätzlich an häufig benötigten Stellen, wobei dies nicht sofort geschieht, da die Übersetzung in Maschinencode aufgrund der Just-in-Time-Kompilierung eine zeitsensitive Aufgabe darstellt. Darüber hinaus ist in der Engine ein Garbage Collector integriert, der nicht mehr verwendete Objekte löscht.[10]
Für weitere Aufgaben setzt Node.js auf Bibliotheken, die fertige und etablierte Lösungsansätze für häufig benötigte Aufgaben zur Verfügung stellen. Nur für Aufgaben, für die es keine etablierte Bibliothek gibt, werden eigene Implementierungen verwendet. Im Folgenden werden die wichtigsten Komponenten vorgestellt.[10]

Node.js Bindings

Node.js Bindings, auch bekannt als Node.js Add-ons, schaffen die Möglichkeit C- oder C++-Quellcode in Node.js zu integrieren. Entwickler können Erweiterungen in nativem Code erstellen und in ihren Anwendungen in JavaScript nutzen. Dies ermöglicht die Nutzung von Systemfunktionalitäten. Dies wird beispielsweise für den Zugriff auf das Dateisystem im Modul *fs* verwendet. Dieses ist in der Application Programming Interface (API) von Node.js enthalten. Darin bietet Node.js viele Lösungen für häufig benötigte Aufgaben, um die Entwicklung zu vereinfachen. Diese sind global im gesamten Anwendungscode verfügbar. Zu den globalen Objekten gehören beispielsweise *console* für die Ausgabe von Informationen in der Konsole und *Buffer* für den Umgang mit binären Daten. In der API ist neben dem Modul *fs* beispielsweise das Modul *http* enthalten, um den Umgang mit dem HTTP-Protokoll zu vereinfachen. Die Module selbst sind in JavaScript geschrieben. D. h. der Kern von Node.js liegt in C (Libuv) und C++ (V8 Engine) vor, die übrigen Komponenten sind in der Sprache der Plattform geschrieben. Allerdings ist in der Standard-API keine Unterstützung für TypeScript enthalten. Hierzu muss der TypeScript-Transpiler separat installiert werden.[10, 11, 12]

Event Loop

Node.js verwendet eine eventgesteuerte Architektur. Anstatt den Quellcode linear auszuführen, werden definierte Events ausgelöst, für die zuvor Callback-Funktionen registriert wurden. Dieses Konzept wird genutzt, um eine hohe Anzahl von asynchronen Aufgaben zu bewältigen. Um dabei den einzelnen Thread der Anwendung nicht zu blockieren, werden Lese- und Schreiboperationen an den Event Loop aus-

gelagert. Wenn auf externe Ressourcen zugegriffen werden muss, leitet der Event Loop die Anfrage weiter und die registrierte Callback-Funktion gibt die Anfrage an das Betriebssystem weiter. In der Zwischenzeit kann Node.js andere Operationen ausführen. Das Ergebnis der externen Operation wird dann über den Event Loop zurückgeliefert.[10]
Während der Laufzeit werden viele Events erzeugt und in einer Message Queue, der Event Queue, nacheinander gespeichert. Node.js nutzt First In First Out und beginnt demnach mit der Verarbeitung der ältesten Events und arbeitet sich durch die Queue, bis keine Events mehr vorhanden sind.[13]

Rechtschreibung prüfen

Libuv

Prüfen, ob tatsächlich großes L (siehe Quelle)

Der Event Loop von Node.js basiert ursprünglich auf der Bibliothek libev. Diese ist in C geschrieben und für ihre hohe Leistung und umfangreichen Features bekannt. Allerdings stützt sich libev auf native UNIX-Funktionen. Diese sind auf Windows über eine andere Schnittstelle nutzbar. Daher dient Libuv als Abstraktionsebene zwischen Node.js und den darunter liegenden Bibliotheken für den Event Loop, um die Laufzeitumgebung auf allen Plattformen nutzen zu können. Libuv verwaltet alle asynchronen I/O-Operationen, einschließlich Dateisystemzugriffe und asynchrone TCP- und UDP-Verbindungen.[10]

Node Package Manager (NPM)

Darüber hinaus bringt Node.js den NPM mit sich. Dieser Paketmanager ist entscheidend für den Erfolg von Node.js, da es im September 2022 mehr als 2,1 Millionen Pakete in diesem Ökosystem gibt. Es gibt somit ein Paket für eine Vielzahl an Anwendungsfällen. Ursprünglich wurde NPM entwickelt, um Abhängigkeiten in Projekten zu verwalten. Mittlerweile wird NPM auch als Werkzeug für JavaScript im Frontend unterstützt. Der NPM ist nicht Teil des Executables von Node.js und wird bei der Installation häufig mitgeliefert. [10, 11]

Aussage in Quelle nochmal prüfen, um besser zu formulieren

Zusammenfassend zeichnet sich Node.js durch eine eventgesteuerte Architektur und durch ein nicht blockierendes Modell für Ein- und Ausgabeoperationen aus. Dieser Aufbau macht es leichtgewichtig und effizient macht. Dies hat verschiedene Vor- und Nachteile.
Zu den Vorteilen gehören eine hohe Performance durch die Nutzung der V8 JavaScript Engine und die Plattformunabhängigkeit. Eine weitere Stärke ist die große und aktive Community an Entwicklern. Dank der Popularität gibt es viele etablierte Lösungsansätze, die den Entwicklungsprozess beschleunigen und vereinfachen. Node.js

ermöglicht die Verwendung der JavaScript-Sprache sowohl auf der Server- als auch auf der Clientseite. Dies vereinfacht die Entwicklung von Full-Stack-Anwendungen und erleichtert Entwicklern den Einstieg.[3, 11]
Allerdings existieren auch Nachteile bei der Verwendung von Node.js. Das Single-Thread-Modell kann bei rechenintensiven oder CPU-lastigen Aufgaben zu Engpässen führen, da es nur einen Hauptthread für die Ausführung von Code gibt [14]. Ein weiterer Nachteil ist, dass Node.js über eine begrenzte Standardbibliothek verfügt, sodass Entwickler häufig auf externe Module und Pakete zurückgreifen müssen, beispielsweise der Transpiler für TypeScript [11].

2.2 Bun

Bun ist ein Open-Source-Toolkit für JavaScript. Dieses kombiniert verschiedene serverseitige Komponenten, um eine leistungsstarkes Paket zur Verfügung zu stellen. Bun ist auf MacOS und Linux für die produktive Nutzung freigegeben. Die Version von Windows besitzt aktuell einen experimentellen Status und ist noch nicht für Performance optimiert. Alternativ kann auf Windows die veröffentlichte Linux-Version über das Windows Subsystem für Linux installiert werden. Ursprünglich ist Bun als ein persönliches Freizeitprojekt von Jared Sumner gestartet. Mittlerweile hat es sich zu einer wettbewerbsfähige Alternative zu bewährten Technologien in der Webentwicklung etabliert.[6, 15]

Abbildung 2.2 zeigt das Ökosystem von Bun im Vergleich zu Node.js. Im Toolkit von Bun sind folgende Komponenten enthalten:

Markus fragen, ob Verschieben der Absätze verkraftbar ist

- eine Laufzeitumgebung für JavaScript,
- ein Paketmanager wie NPM (siehe Kapitel 2.1) oder Yarn,
- ein Transpiler wie Babel,
- ein Build-Tool wie Webpack,
- Bibliotheken zum Testen wie Jest oder Vitest,
- und integrierte Unterstützung für TypeScript [6].

Bun strebt an, das Rundum-sorglos-Tool zu sein, damit alle benötigten Funktionalitäten im Kontext von JavaScript nativ verfügbar sind. Gleichzeitig sollen dadurch die Abhängigkeiten einer Software auf Basis von Bun reduziert werden. In Node.js ist nur die Laufzeitumgebung enthalten, die anderen Komponenten müssen separat installiert

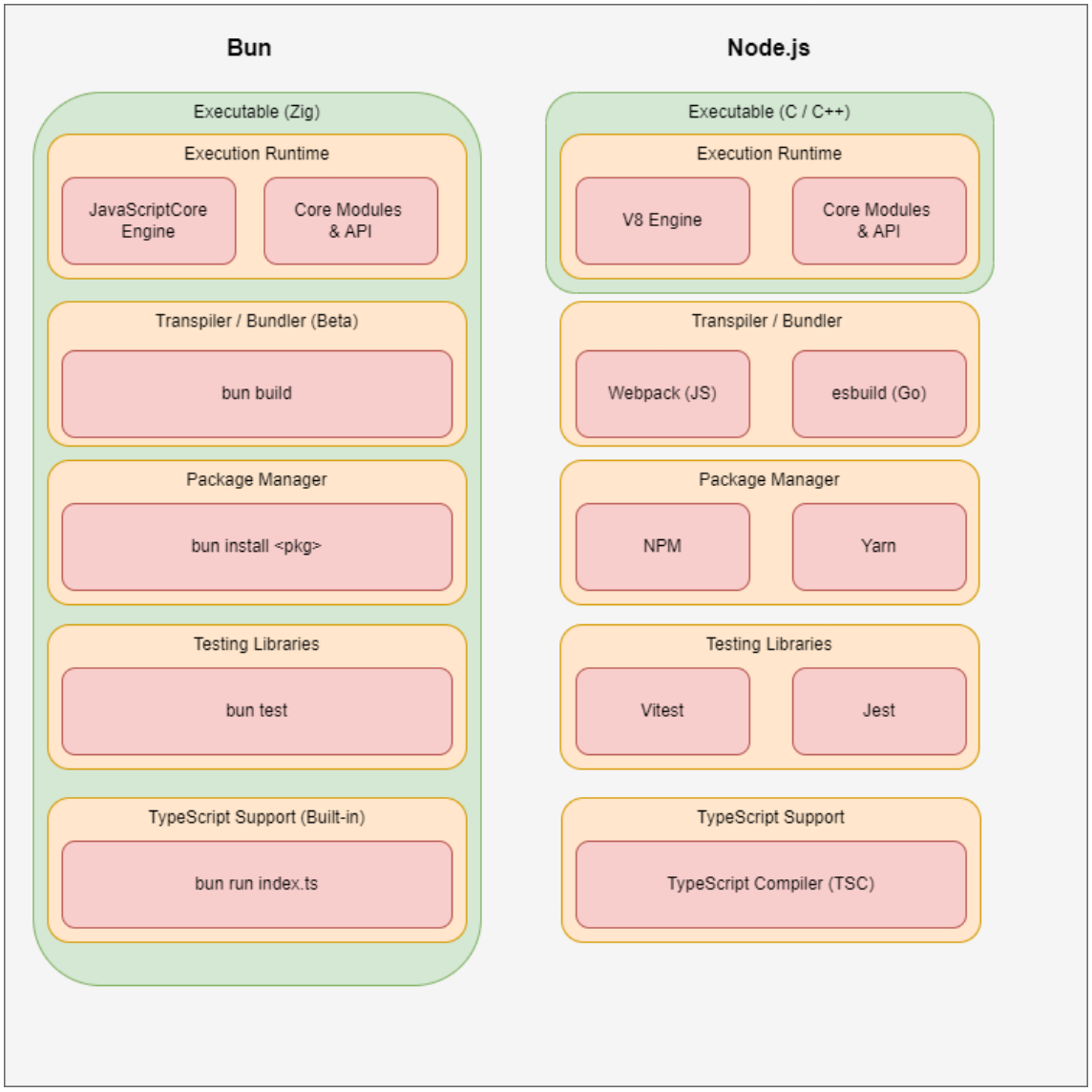

Abbildung 2.2: Vergleich der Ökosysteme von Bun und Node.js
Quelle: in Anlehnung an [10, 16]

werden. Dies bietet allerdings mehr Flexibilität bei der Auswahl der gewünschten Tools.[10, 16]

Bun ist in Zig geschrieben. Dies ist eine systemnahe Programmiersprache wie C und C++, die sich vor allem auf Einfachheit und Klarheit für ein besseres Verständnis konzentriert [17]. Die Entwickler haben sich aufgrund der sehr guten Performance und des Speichermanagements für Zig entschieden. Zig ermöglicht mit dessen manuellem Speichermanagement, Klarheit im Kontrollfluss und Klarheit beim Allokieren von Speicher weitere Verbesserungen der Effizienz. Anstatt der V8 JavaScript Engine von Google verwendet Bun die JavaScriptCore Engine. Das ist die Engine für WebKit,

die unter anderem in Apple's Safari-Browser genutzt wird. In Kombination mit Zig sorgt die JavaScriptCore Engine für eine bessere Performance und zu reduzierten Startzeiten. Dies ist auf die Architektur der JavaScriptCore Engine mit drei Just-in-Time-Compilern (JIT-Compiler) zurückzuführen. Dadurch kann die Engine den Quellcode besser optimieren. Das ist vor allem im Bereich des Serverless Computing ein Vorteil gegenüber anderen Alternativen. Die niedrigen Startzeiten helfen die Skalierbarkeit einer Software zu verbessern, indem neue Knoten schneller hinzugezogen werden können.[16, 18, 19, 20, 21]
Node.js bietet viele Module, globale Objekte und Standard-Web-APIs an (siehe Kapitel 2.1). Bun möchte eine nahtlose Integration mit Node.js anbieten. Dazu haben die Entwickler verbesserte Versionen für viele dieser Objekte implementiert. Hierzu zählen beispielsweise:

- Standard-Web-API: *fetch*, *Request*, *Response*,
- Module: http, https, path,
- Globale Objekte: btoa (Binary to ASCII), *atob* (ASCII to Binary).[16]

Allerdings existieren auch viele Module und globale Objekte, für die die Unterstützung teilweise oder komplett fehlen, zum Beispiel *tel*, *net* oder *http2* [16]. Daraus folgt, dass die Kompatibilität von bestehenden Node.js-Projekten von den verwendeten Objekten und Modulen abhängt.

2.3 Performance als Qualitätsattribut

Um eine qualitativ hochwertige Software zu entwickeln, genügt es nicht, die funktionalen Anforderungen zu erfüllen. Entwickler tragen die Verantwortung, die Anforderungen an eine Applikation, sowohl die funktionalen als auch die nichtfunktionalen Aspekte, in vollem Umfang zu erfüllen. Die Qualität einer Software besteht darin, in welchem Maße die Software die expliziten und impliziten Bedürfnisse seiner Stakeholder zufriedenstellt und so Mehrwehrt bietet. Diese Bedürfnisse werden im Qualitätsmodell nach ISO / IEC 25010:2011 dargestellt.[22]

Abbildung 2.3 zeigt die acht Charakteristika der Software-Qualität nach ISO / IEC 25010: Funktionalität, Performance, Kompatibilität, Benutzbarkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit und Portierbarkeit. Der Standard bietet ein Framework für die Bewertung der Qualität einer Software an. Er hilft so Software-Produkte zu verbessern und alle Teilbereiche zu beachten, indem der Standard als Leitfaden

Abbildung 2.3: Qualitätsattribute einer Software
Quelle: [22]

vom Identifizieren der Anforderungen bis zur Qualitätskontrolle der Software unterstützt.[23]

Performance definiert sich im IEEE Standard Glossary of Software Engineering Terminology wie folgt:

> *"The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage"* [24]

Demnach beschreibt Performance die Reaktion eines Systems auf die Durchführung einer Aktion über einen definierten Zeitraum. Um die Performance einer Software bestimmen zu können, stellt ISO / IEC 25010 drei Charakteristiken zur Verfügung (siehe Abbildung 2.3), die in den folgenden Absätzen beschrieben werden.

Zeitverhalten

Das Zeitverhalten beschreibt das Maß, in dem die Reaktions-, Verarbeitungszeiten und Durchsatzraten eines Software-Produkts bei der Ausführung definierten Anforderungen entsprechen [23]. Der Fokus liegt hier auf einer schnellen Reaktion der Software, um die definierten Vorgaben für die Performance einzuhalten. Das Zeitverhalten kann durch die Latenz und den Durchsatz genauer spezifiziert werden. Die Latenz definiert einen zeitlichen Intervall, in dem die Software eine Antwort auf die Anfrage liefern muss. Dieses Intervall wird in einem Zeitfenster durch eine minimale und maximale Zeitangabe definiert. Die Zeitangaben können absolut oder relativ in Bezug auf ein Event angegeben werden. Die Anzahl an abgeschlossenen Antworten auf eine Anfrage innerhalb eines Beobachtungsintervalls beschreibt den Durchsatz. Dadurch kann die Verarbeitungsleistung (Processing Rate) der Software abgeleitet werden. Für eine zuverlässige Angabe ist es empfohlen, mehrere Zeitfenster zu beobachten. Denn es kann sein, dass eine Software 120 Anfragen innerhalb

1 Stunde bearbeiten kann. Dennoch könnte das System versagen, wenn 40 dieser Anfragen innerhalb von 3 Minuten abgearbeitet werden müssen.[25]

Ressourcennutzung

Das Maß, in dem die Menge und Art der Ressource, die ein Produkt bei der Ausführung seiner Funktionalitäten beansprucht, entspricht der Ressourcennutzung [23]. Es geht um die effiziente Verwaltung der verfügbaren Ressourcen. Dazu zählen die CPU, der Arbeitsspeicher, die Bandbreite des Netzwerks, der Speicherplatz auf der Festplatte und viele mehr. Die wichtigsten Metriken sind die CPU-Auslastung, der Speicherbedarf sowohl im RAM als auch auf der Festplatte.[25]

Kapazität

Die Kapazität entspricht dem Maß, in dem die maximalen Grenzen eines Parameters einer Software den Anforderungen entsprechen [23]. Dadurch wird bestimmt, ob das System unter Spitzenlast funktionsfähig bleibt und dadurch skalierbar ist. Hierbei müssen die Anforderungen an die maximale Latenz eingehalten werden. Daher kann die Kapazität alternativ auch als der maximal mögliche Durchsatz unter Einhaltung der gegebenen Latenzanforderungen bezeichnet werden. Das umfasst mehrere Benutzer, die gleichzeitig auf die Software zugreifen, oder größere Transaktionen mit mehr Datenvolumen. Zu den Metriken zählen die maximale Anzahl an gleichzeitigen Benutzern und die maximale Anzahl an möglichen Transaktionen.[25]

Kapitel 3

Performanceanalyse

In diesem Kapitel wird die Performance von Bun ausführlich untersucht und mit Node.js verglichen. Hierbei liegt der Fokus darauf, die Leitfrage „Welche konkreten Leistungsverbesserungen können in Bun 1.0 im Vergleich zu Node.js festgestellt werden, und wie lassen sie sich quantifizieren?“ zu beantworten (siehe Kapitel 1). Zuerst wird die Vorgehensweise bei den Tests vorgestellt. Anschließend wird der verwendete Versuchsaufbau erläutert. Vor der Betrachtung der Ergebnisse folgt die Vorstellung der Beispielimplementierungen.

3.1 Vorgehensweise

Als Metriken für die Performanceanalyse werden die durchschnittliche Latenz, die Anzahl an HTTP-Anfragen pro Sekunde, der Anteil an erfolgreichen HTTP-Anfragen, die CPU-Auslastung und der maximal genutzte Arbeitsspeicher während der Ausführungszeit verwendet. Denn diese spiegeln die in der Theorie erarbeiteten Charakteristiken wider (siehe Kapitel 2.3). Um die Metriken zu ermitteln, werden verschiedene Testszenarien mit unterschiedlichen Implementierungen betrachtet (siehe Kapitel 3.3). Diese sind auf variierende APIs der Laufzeitumgebungen zurückzuführen. Dadurch werden die Performance von Bun und Node.js bewertet.

Zuerst erfolgt die Messung der grundlegenden Performance von HTTP-Servern in beiden Laufzeitumgebungen (siehe Kapitel 3.3.1). Dabei greifen 500 gleichzeitige Benutzer für 30 Sekunden auf den Server zu und erhalten einen kurzen Text als Antwort zurück. Dieses Szenario dient der Untersuchung der grundlegenden Netzwerkgeschwindigkeit beider Laufzeitumgebungen. Als nächstes wird ein Datei-Server verwendet, der jedem Aufrufer ein Bild zurückgibt (siehe Kapitel 3.3.2). Dieser Test wird für 50, 250, 500 und 1.000 gleichzeitige Nutzer über einen Zeitraum von 30

Sekunden durchgeführt. Die Last wird variiert, um die Server näher an ihre Grenzen zu bringen. Der letzte Testfall berechnet die Fibonacci-Folge für die Zahl 45, um die Leistung beider Laufzeitumgebungen bei rechenintensiven Aufgaben zu evaluieren (siehe Kapitel 3.3.3).
Um die Performance korrekt zu bestimmen, müssen geeignete Tools verwendet werden. In dieser Arbeit kommen folgenden Tools zum Einsatz:

- Bombardier Version 1.2.6,
- GNU Time.

Bombardier generiert die HTTP-Anfragen an die Server in den ersten beiden Testszenarien. Mit diesem Tool kann die Dauer der Lasttests sowie die Anzahl der zu versendenden Anfragen konfiguriert werden. Außerdem lässt sich festlegen, wie viele gleichzeitige Benutzer simuliert werden. Nach dem Test liefert Bombardier durchschnittliche und maximale Werte für die Anzahl der Anfragen pro Sekunde und die Latenz, einschließlich der Standardabweichung. Zusätzlich schlüsselt das Tool die erhaltenen HTTP-Statuscodes auf. Dadurch kann der Anteil an erfolgreichen und nicht erfolgreichen Anfragen bestimmt werden. Bombardier eignet sich aufgrund seinen detailreichen Aufgaben und seiner Performance. Es ist in Go geschrieben und verwendet das Paket “fasthttp“ anstelle der nativen HTTP-Implementierung von Go. Dadurch ist es ausreichend performant.[26]
Zur Erfassung der CPU-Auslastung und des maximal genutzten Arbeitsspeichers wird GNU Time verwendet. GNU Time ist auf Ubuntu bereits nativ verfügbar und daher gut geeignet [27]. Auf MacOS wird eine entsprechende Portierung dieses Tools genutzt, um vergleichbare Daten zu erheben.
Um möglichst repräsentative Ergebnisse zu erzielen, werden alle Testszenarien für jede Laufzeitumgebung jeweils 5-mal wiederholt. Die Durchschnittswerte aus den gesammelten Daten senken die Auswirkung einzelner Abweichungen.

3.2 Versuchsaufbau

Um eine konsistente und kontrollierte Umgebung für die Tests zu gewährleisten, werden diese auf spezifischer Hard- und Software durchgeführt. Das Ziel besteht darin, die Testergebnisse reproduzierbar zu gestalten, damit diese unabhängig verifiziert werden können. Aus der Reproduzierbarkeit folgt eine einheitliche Quantifizierung der Ergebnisse zwischen Bun und Node.js. Diese ist notwendig, um die Vergleichbarkeit zu gewährleisten.

Tabelle 3.1: Hardware für die Performanceanalyse
Quelle: Eigene Darstellung

Name	Desktop-PC	MacBook Pro
Prozessor	AMD Ryzen 7 2700 @ 3,6 GHz	Apple M1 Pro
Arbeitsspeicher	32 GB DDR4-3200	16 GB LPDDR5-6400
Betriebssystem	Ubuntu 23.10	MacOS 14 Sonoma

Tabelle 3.1 zeigt die verwendete Hardware und die dazugehörigen Betriebssysteme. Die Verwendung von mehreren Betriebssystemen inklusiver unterschiedlicher Hardware ermöglicht die Verifikation, ob etwaige Performance-Verbesserungen auf eine spezifische Systemumgebung zurückzuführen sind. Die native Implementierung von Bun für Windows ist experimentell und ist für die Öffentlichkeit nicht zugänglich [28]. Daher ist es nicht möglich, die Funktionsweise von Bun unter Windows zu testen.
Die folgenden Versionen der betrachteten Frameworks werden eingesetzt:

- Bun Version 1.0.6 (Neuste Version[1])
- Node.js Version 18.18.2 (Long Term Support (LTS)[1])
- Node.js Version 21.0.0 (Neuste[1])

Abkürzung überall verwenden

Die neuste Version von Bun wird für die Tests verwendet, da sie im Vergleich zur Version 1.0 bereits Fehlerkorrekturen enthält [29]. Bei der Analyse von Node.js werden zwei Versionen einbezogen. Einerseits die Version mit LTS, da Node.js diese Version für die meisten Benutzer aufgrund des langfristigen Supports empfiehlt [30]. Andererseits die neuste Version von Node,js, da diese im Vergleich zur LTS-Version mehrere Verbesserungen für die Performance enthält [31]. Dazu gehört beispielsweise die neuste Version des URL-Parsers Ada. Alleine der aktualisierte URL-Parser verspricht signifikante Performance-Verbesserungen [32].
Der Versuchsaufbau stellt sicher, dass die Performance-Tests auf aussagekräftige und vergleichbare Ergebnisse hinarbeiten und somit der Untersuchung der ersten Forschungsfrage (siehe Kapitel 1.2) dienen.

3.3 Implementierungen

Im Folgenden werden die für jedes Testszenario (siehe Kapitel 3.1) verwendeten Implementierungen vorgestellt.

[1]Stand 14.10.2023

3.3.1 HTTP-Server

Um die grundlegende Performance von Netzwerkanfragen zu bestimmen, werden zwei einfache Programme verwendet. Die Latenz spiegelt die Reaktionszeiten der betrachteten Laufzeitumgebungen wider, da sowohl Client als auch Server auf demselben Endgerät stattfinden und somit keine Verzögerungen durch das Netzwerk auftreten. In Listing 3.1 ist der Quellcode für Bun dargestellt, während Listing 3.2 den Quellcode für Node.js zeigt.

```
1  Bun.serve({
2    port: 3000,
3    fetch(request) {
4      return new Response("Hello from Bun!");
5    },
6  });
```

Listing 3.1: HTTP-Server Bun

Quelle: Eigene Darstellung

```
1  import http from "node:http";
2
3  http.createServer(function (request, response) {
4    response.write('Hello from Node.js!')
5    response.end();
6  }).listen(3000);
```

Listing 3.2: HTTP-Server Node.js

Quelle: Eigene Darstellung

Um die Performance dieser Programme zu testen, werden mit Bombardier 500 gleichzeitige Benutzer für eine Dauer von 30 Sekunden simuliert, wie im Befehl in Listing 3.3 visualisiert.

```
1  bombardier -c 500 -d 30s http://localhost:3000
```

Listing 3.3: Bombardier HTTP-Server

Quelle: Eigene Darstellung

Die Server werden mit der jeweiligen Laufzeitumgebung gestartet. Zur Ermittlung der CPU- und RAM-Auslastung wird GNU Time verwendet. Die Befehle zur Messung der CPU- und RAM-Auslastung sind in Ubuntu und MacOS in den Abbildungen 3.4 und 3.5 dargestellt. Um dieselben Messungen mit Node.js durchzuführen, muss *bun* durch *node* ersetzt werden.

```
/usr/bin/time -f "Execution Time: %e\nMaximum Resident Set Size (RSS): %M\nPercent of CPU This Job Got: %P" bun httpServer.js
```

Listing 3.4: CPU- und RAM-Messung auf Ubuntu
Quelle: Eigene Darstellung

```
gtime -f "Execution Time: %e\nMaximum Resident Set Size (RSS): %M\nPercent of CPU This Job Got: %P" bun httpServer.js
```

Listing 3.5: CPU- und RAM-Messung auf macOS
Quelle: Eigene Darstellung

Beispiel-Ausgabe von Bombardier zeigen, um zu erklären, welche Daten pro Testdurchlauf erfasst werden

3.3.2 Datei-Server

Im zweiten Szenario wird die Abfrage von Bilddateien simuliert. Denn ein Bild stellt eine große Datenmenge da und prüft so den Austausch großer Datenmengen. Dieser Prozess inkludiert die Performance von Zugriffen auf das Dateisystem. Die Implementierungen für Bun und Node.js sind in den Abbildungen 3.6 und 3.7 dargestellt.

```
const basePath = "../data";

Bun.serve({
  port: 3000,
  fetch(request) {
    const filePath = `${basePath}${new URL(request.url).pathname}`;

    try {
      return new Response(Bun.file(filePath));
    } catch (error) {
      return new Response("File not found", {
        status: 404
      });
    }
  },
});
```

Listing 3.6: Datei-Server Bun
Quelle: Eigene Darstellung

```
import { createReadStream } from "node:fs";
import http from "node:http";

const basePath = "../data";
```

```
5
6   http.createServer((request, response) => {
7     const filePath = `${basePath}${request.url}`;
8     const readStream = createReadStream(filePath);
9
10    readStream.on("open", () => {
11      response.setHeader("content-type", "image/png");
12      response.writeHead(200);
13
14      readStream.pipe(response);
15    });
16
17    readStream.on("error", () => {
18      response.writeHead(404, "Image not found");
19      response.end();
20    });
21  }).listen(3000);
```

Listing 3.7: Datei-Server Node.js
Quelle: Eigene Darstellung

Beide Server rufen dasselbe Bild aus dem Dateisystem ab, da in jeder Anfrage derselbe Pfad angegeben wird. Falls unter dem angeforderten Pfad keine Datei gefunden wird, geben beide Server eine entsprechende Fehlermeldung zurück. Dieses Testszenario wird jeweils mit 50, 250, 500 und 1.000 gleichzeitigen Benutzern für eine Dauer von 30 Sekunden getestet (siehe Kapitel 3.1). Der Befehl für 50 gleichzeitige Benutzer ist in Listing 3.8 dargestellt. Die Befehle zur Messung der CPU- und RAM-Auslastung unterscheiden sich nicht im Vergleich zum HTTP-Server (siehe Kapitel 3.3.1).

```
1   bombardier -c 500 -d 30s http://localhost:3000/example.png
```

Listing 3.8: Bombardier Datei-Server
Quelle: Eigene Darstellung

3.3.3 Berechnung der Fibonacci-Folge

Als letztes Szenario wird die Fibonacci-Folge für die Zahl 45 berechnet, um die Leistung bei der Ausführung rechenintensiver Aufgaben zu bewerten. Hierfür nutzen beide Laufzeitumgebungen die in Listing 3.9 dargestellte Implementierung.

```
1   const fibonacci = (number) => {
2     if (number <= 0) {
3       return 0;
4     } else if (number <= 1) {
5       return 1;
```

```
6     } else if (number <= 2) {
7       return 2;
8     }
9
10    return fibonacci(number-1) + fibonacci(number-2);
11  };
12
13  console.log(fibonacci(45));
```

Listing 3.9: Berechnung der Fibonacci-Folge
Quelle: Eigene Darstellung

Das Programm wird mit beiden Laufzeitumgebungen und GNU Time zur Erhebung der notwendigen Metriken ausgeführt. Listing 3.10 stellt dies beispielsweise für Node.js unter Ubuntu dar. Auf MacOS muss */usr/bin/time* durch *gtime* ersetzt werden.

```
1  /usr/bin/time -f "Execution Time: %e\nMaximum Resident Set Size (
     RSS): %M\nPercent of CPU This Job Got: %P" node fibonacci.js
```

Listing 3.10: Messung der Fibonacci-Folge auf Ubuntu
Quelle: Eigene Darstellung

3.4 Ergebnisse

Im Folgenden werden die Ergebnisse der Testszenarien vorgestellt. Im Anschluss folgt die Diskussion über die daraus resultierenden Vor- und Nachteile.

HTTP-Server Performance

Im ersten Testszenario wurde die grundlegende Leistung der HTTP-Server verglichen. Abbildung 3.1 zeigt die durchschnittliche Anzahl an Anfragen pro Sekunde, die jede Laufzeitumgebung bewältigen konnte. Bun übertraf auf beiden getesteten Geräten sowohl die LTS-Version als auch die neueste Version von Node.js. Bun konnte auf dem Desktop-PC pro Sekunde ungefähr 103.000 Anfragen bewältigen, während Node.js LTS und die neueste Version nur 30.000 bzw. 32.000 Anfragen pro Sekunde verarbeiten konnten. Diese Leistungsunterschiede spiegelten sich auch in den Latenzzeiten wider. Bun hatte eine Latenz von 6,61ms auf dem MacBook Pro und 4,84ms auf dem Desktop-PC, verglichen mit ungefähr 10 ms (MacBook Pro) und etwa 16 ms (Desktop-PC) für Node.js (siehe auch Abbildung A.1 in Kapitel A.4).
Diese signifikanten Unterschiede deuten darauf hin, dass Bun das Potential bietet, in realen Szenarien erheblich schneller zu sein als Node.js. Diese Erkenntnisse legen nahe, dass die Verwendung der JavaScriptCore Engine und der Programmiersprache Zig

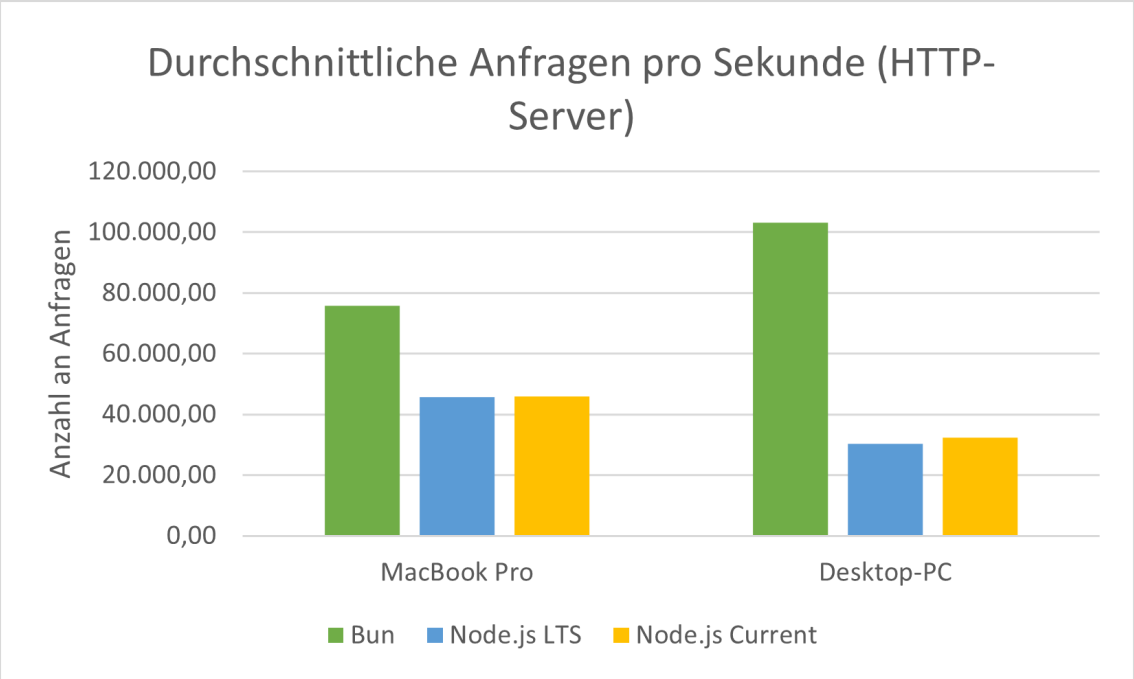

Abbildung 3.1: HTTP-Server - Durchschnittliche Anzahl an Anfragen pro Sekunde
Quelle: Eigene Darstellung

klare Vorteile bei der Bewältigung von Netzwerkanfragen bietet. Das hat erhebliche Auswirkungen auf die Effizienz und Skalierbarkeit in Produktionsumgebungen.

Datei-Server Performance

Das zweite Testszenario analysiert, inwiefern die Potentiale aus dem ersten Test in einem realen Anwendungsfall umgesetzt werden. Um die Diagramme übersichtlich zu halten, konzentriert sich die Darstellung auf den Desktop-PC. Denn die Unterschiede und Trends sind auf beiden Geräten ähnlich. Abbildung 3.2 zeigt die durchschnittlichen Latenzen beim Zugriff auf Bilddateien. Bun schnitt erneut besser ab als Node.js. Bei 50 gleichzeitigen Benutzern betrug die Latenzzeit für Bun 2 ms, während Node.js LTS und die neueste Version Latenzen von 4,2 ms bzw. 3,5 ms aufwiesen. Diese Unterschiede wurden bei steigender Anzahl gleichzeitiger Benutzer noch deutlicher. Bei 250 Nutzern beläuft sich Bun's Latenz auf ca. 10ms, der beste Wert von Node.js liegt bei ca. 21ms (Neuste Version). Bei 1.000 gleichzeitigen Benutzern benötigte Bun durchschnittlich 42 ms für die Antwort, während Node.js bei 116 ms (neueste Version) lag. Die Unterschiede von mehr als 50% führen zu erheblich schnelleren Ladezeiten von Webseiten, insbesondere bei einer hohen Anzahl von Bildern. Dadurch, dass Bun die Anfragen selbst deutlich schneller beantwortet, bewältigt Bun gleichzeitig deutlich mehr Anfragen pro Sekunde. Trotz der höheren Effizienz konnte Bun alle Anfragen erfolgreich beantworten, unabhängig von der Anzahl gleichzeitiger Benutzer. In der

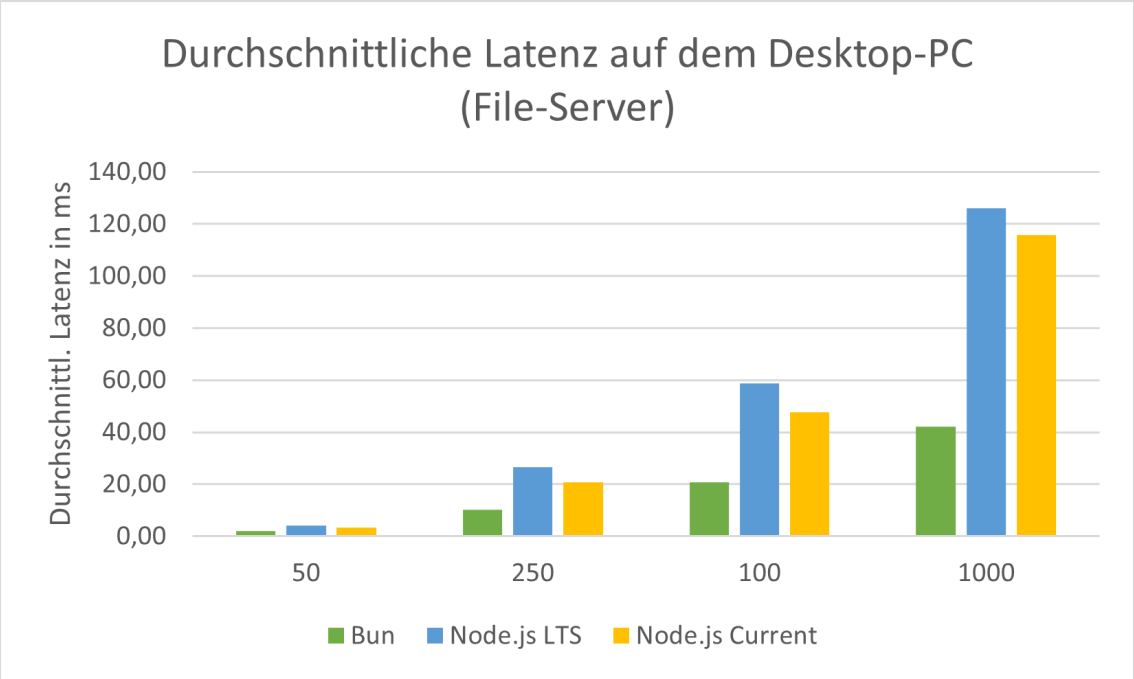

Abbildung 3.2: Datei-Server - Durchschnittliche Latenz
Quelle: Eigene Darstellung

neuesten Version von Node.js wurden ab 250 gleichzeitigen Benutzern nicht mehr alle Anfragen erfolgreich beantwortet. Bei 1.000 gleichzeitigen Nutzern sank die Erfolgsrate auf dem MacBook Pro auf 99,54%. Bun erzielte bei 50, 250 und 500 gleichzeitigen Benutzern eine hundertprozentige Erfolgsrate bei der Beantwortung aller Anfragen. Bei 1.000 simulierten Anwendern sank die Erfolgsrate auf 99,96%. Node.js LTS zeigte ebenfalls eine sehr hohe Erfolgsrate, bei der erst ab 1.000 gleichzeitigen Benutzern fehlerhafte Antworten zurückgeliefert wurden, was zu einer Erfolgsrate von 99,92% führte. Obwohl die absolute Anzahl der Anfragen mit Fehlern sehr niedrig war, ist es dennoch bemerkenswert, dass die neueste Version von Node.js hier deutlich größere Defizite aufwies. Möglicherweise sind Fehler im neusten Release enthalten, die diese Leistungsunterschiede erklären.

Die RAM- und CPU-Nutzung der Laufzeitumgebungen werden bei 1.000 gleichzeitigen Benutzern verglichen. Denn bei der höchsten Last sind die Differenzen am besten zu erkennen. Abbildung 3.3 zeigt den maximal verwendeten Arbeitsspeicher während des 30-sekündigen Tests. Bun war auch hier effizienter als Node.js auf beiden getesteten Geräten. Auf dem MacBook Pro verbrauchte Bun etwa 56 MB Arbeitsspeicher, während Node.js LTS und die neueste Version etwa 296 MB bzw. 330 MB benötigten. Auf dem Desktop-PC betrug der Arbeitsspeicherverbrauch von Bun etwa 84 MB, im Vergleich zu 152 MB (LTS) bzw. 212 MB (neueste Version)

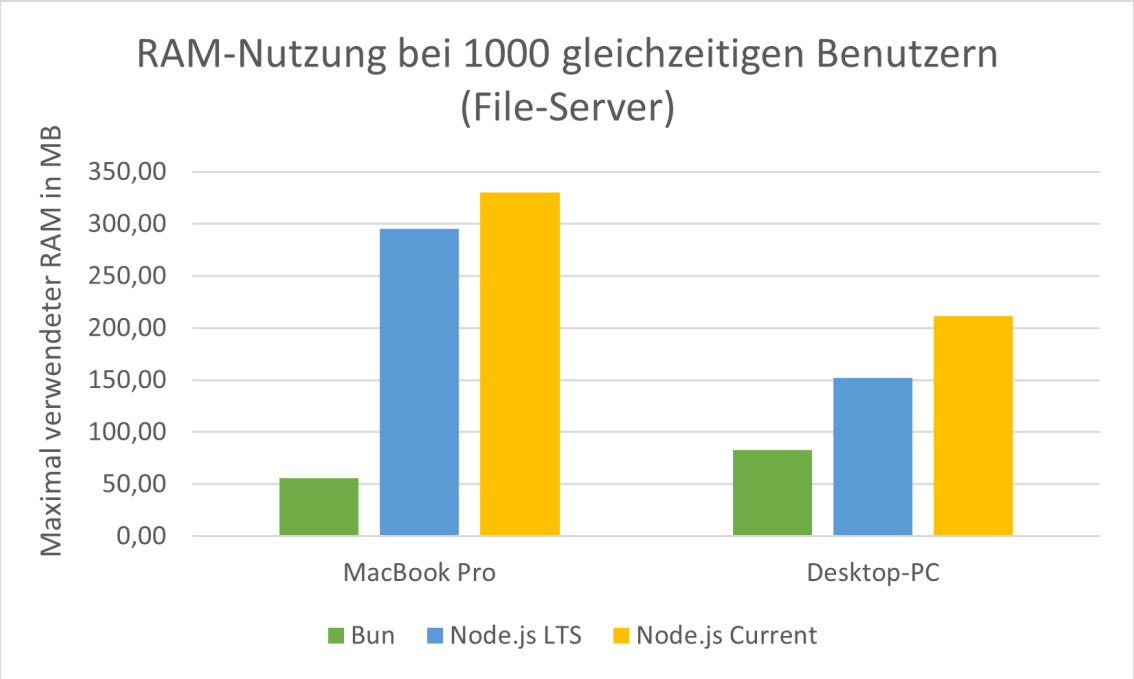

Abbildung 3.3: Datei-Server - Maximal verwendeter Arbeitsspeicher
Quelle: Eigene Darstellung

bei Node.js. Die Beobachtungen deuten auf klare Vorteile in Bezug auf Effizienz und Kosten hin. Daraus lässt sich schließen, dass Bun anspruchsvollere Aufgaben mit weniger Ressourcen bewältigen kann.

Abbildung 3.4 zeigt die CPU-Auslastung bei 1.000 gleichzeitigen Benutzern in Prozent. Bei der CPU-Auslastung bedeuten 100%, dass ein Kern vollständig ausgelastet ist. Wenn ein Computer eine CPU mit vier Kernen besitzt, beträgt die maximale CPU-Auslastung 400%. Die CPU-Auslastung bestätigt, dass Bun mit den verfügbaren Ressourcen effizienter umgeht als Node.js. Die CPU-Nutzung bei Bun betrug etwa 91% auf dem MacBook Pro und 96% auf dem Desktop-PC. Node.js LTS hingegen benötigte etwa 195% auf dem MacBook Pro und 167% auf dem Desktop-PC, während die neueste Version noch schlechter abschnitt. Bun kommt mit einem Kern der CPU aus, während Node.js zwei oder auch drei Kerne beansprucht. D. h. Node.js stellt ungefähr doppelt so starke Anforderungen an die CPU. Diese Unterschiede in der CPU-Auslastung und Arbeitsspeichernutzung bestärken die Aussage, dass Bun bei gleichzeitiger Last effizienter war und anspruchsvolle Aufgaben bewältigen konnte.

Performance bei der Berechnung der Fibonacci-Folge
Der dritte Testfall konzentriert sich auf die Leistung von Bun und Node.js bei der Ausübung rechenintensiver Aufgaben. Die Ausführungszeiten sind in Abbildung 3.5

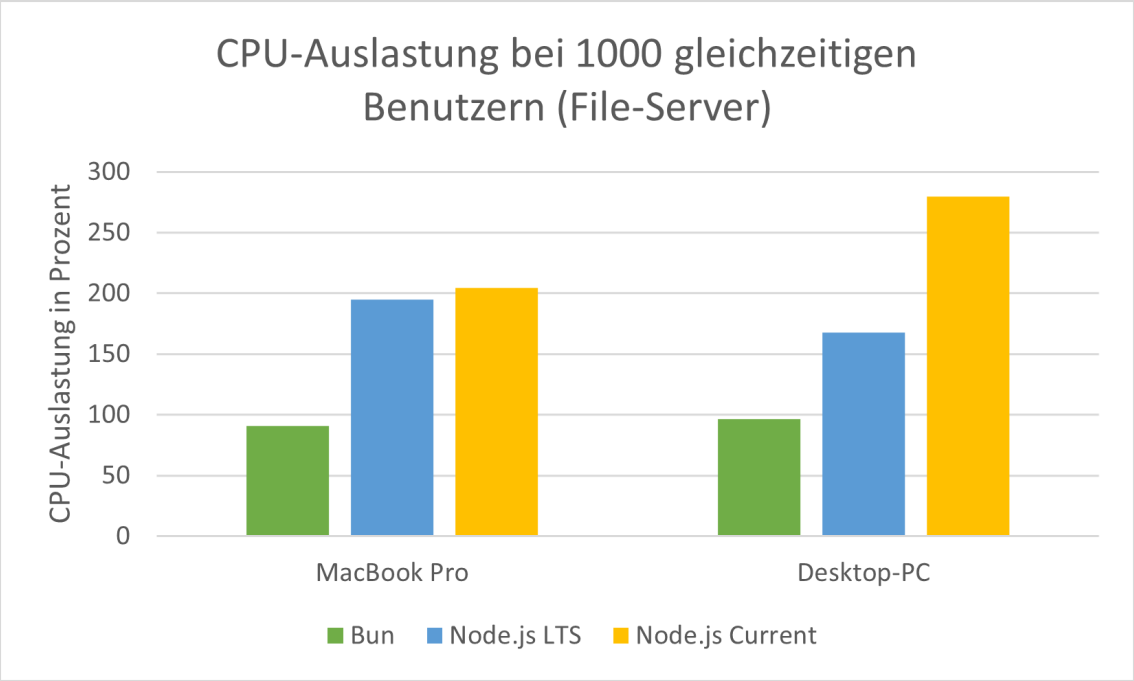

Abbildung 3.4: Datei-Server - CPU-Auslastung
Quelle: Eigene Darstellung

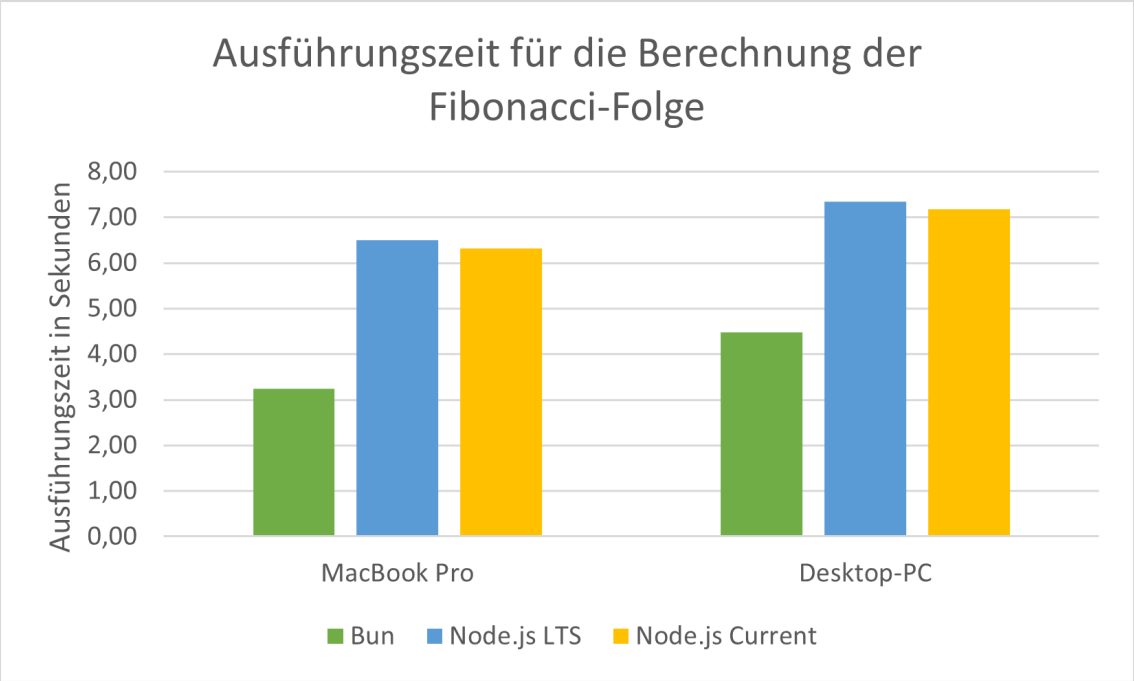

Abbildung 3.5: Berechnung der Fibonacci-Folge - Ausführungszeit
Quelle: Eigene Darstellung

dargestellt. Bun führt die Berechnung durchschnittlich in 3,24 Sekunden auf dem MacBook Pro und in 4,47 Sekunden auf dem Desktop-PC durch. Node.js zeigte kaum Unterschiede zwischen der LTS- und der neuesten Version, wobei die schnellste

Berechnung 6,32 Sekunden (MacBook Pro) und 7,18 Sekunden (Desktop-PC) dauerte. Somit ist Bun auf dem MacBook Pro ca. 50% und auf dem Desktop-PC ungefähr 40% schneller. Die CPU-Auslastung beträgt sowohl bei Bun als auch bei Node.js ca. 100% (siehe Abbildung A.2 in Kapitel A.4). Der maximal verwendete Arbeitsspeicher unterscheidet sich auf dem Desktop-PC nur um maximal 3 MB. Erwähnenswert ist, dass hier die neuste Version von Node.js 1 MB weniger Arbeitsspeicher als Bun genutzt hat. Auf dem MacBook Pro sind die Unterschiede größer. Bun benötigt maximal ca. 26 MB RAM, die neuste Version von Node.js im Vergleich dazu 37 MB und die LTS-Version 42 MB (siehe Abbildung A.3 in Kapitel A.4).
Die Betrachtung der CPU-Auslastung, der RAM-Nutzung und der Ausführungszeiten bestätigt die beobachteten Leistungsunterschiede aus den beiden vorherigen Testszenarien. Unabhängig von der betrachteten Metrik weist Bun eine deutlich höhere Leistungsfähigkeit auf beiden Endgeräten auf.

3.5 Fazit

Dieses Kapitel beantwortet die erste Leitfrage „Welche konkreten Leistungsverbesserungen können in Bun 1.0 im Vergleich zu Node.js festgestellt werden, und wie lassen sie sich quantifizieren?“ (siehe Kapitel 1). Die Ergebnisse des Benchmarks verdeutlichen, dass Bun in sämtlichen Testszenarien signifikant bessere Leistungen als Node.js erzielt hat. Dies zeigt sich in einer reduzierten Latenz, in einer geringeren Inanspruchnahme von Arbeitsspeicher und CPU-Ressourcen während der Verarbeitung von Netzwerkanfragen. Diese Resultate bestätigen sich bei der Ausführung von rechenintensiven Aufgaben. Bun hat die Fibonacci-Folge bis zu 50% schneller berechnet als Node.js. In diesem Szenario beansprucht Bun ähnlich viel Arbeitsspeicher wie Node.js, weist jedoch eine geringere CPU-Auslastung auf.
Die Ergebnisse unterstreichen, dass Bun im Entwicklungsprozess gute Entscheidungen bei der Auswahl seiner Komponenten getroffen hat. Die JavaScriptCore Engine nutzt ihre mehreren Just-in-Time-Compiler (JIT-Compiler) (siehe Kapitel 2.2) für eine effizientere Optimierung des Maschinencodes. Die Auswahl von Zig als systemnahe Programmiersprache und das intensive Testen während der Entwicklung haben (siehe Kapitel 2.2) die Effizienz von Bun im Umgang mit den verfügbaren Ressourcen gesteigert.

Das Benchmark basiert auf dem Vergleich ausgewählter Eigenschaften auf einem spezifischen Hardware-Setup. Es ist zu beachten, dass diese Ergebnisse unter Verwendung von Servern in einer Produktivumgebung potenziell abweichen können.

Die Analysen beschränken sich auf einfache Beispiele, die sich auf die Performance der Laufzeitumgebungen konzentrieren. In der Realität sind Anwendungsquellcodes oft umfangreicher und komplexer. Die vorliegenden Ergebnisse geben daher nicht zwangsläufig die Performance solcher Anwendungen wieder. Für eine umfassende Bewertung der Laufzeitleistung sollten die Unterschiede zwischen dem MacBook Pro und dem Desktop-PC genauer analysiert werden. Es ist wichtig, die Gründe für die Unterschiede im Anteil erfolgreicher Anfragen zu ermitteln. Die Untersuchung sollte auf andere Hardware-Setups ausgeweitet werden, um die Generalisierbarkeit der Ergebnisse sicherzustellen.

Kapitel 4

Kompatibilität von Projekten

Dieses Kapitel behandelt die Kompatibilität zwischen Node.js und Bun. Es beschäftigt sich mit der Beantwortung der zweiten Leitfrage „Inwiefern sind Node.js-Projekte kompatibel mit Bun? Wie schwierig gestaltet sich die Migration?“. Nur wenn der Migrationsaufwand für den Wechsel der Laufzeitumgebung möglichst gering ist, kann Bun Node.js als Marktführer verdrängen. Die Betrachtung konzentriert sich auf die Kompatibilität häufig verwendeter Backend-Frameworks, da diese auf einem Server direkt von Bun oder Node.js ausgeführt werden. Laut [5] handelt es sich dabei um Express und Nest.

4.1 Express

Express ist ein beliebtes Open-Source-Framework für Node.js, das die Entwicklung von Webanwendungen und APIs vereinfacht. Es stellt eine Vielzahl von Funktionen und Tools zur Verfügung, die Entwicklern bei der Handhabung von HTTP-Anfragen und -Antworten, Routing und weiteren Aufgaben helfen. Express nutzt ausschließlich die HTTP-Schnittstelle der Node.js-API. Diese implementiert Bun bereits im Standard (siehe Kapitel 2.2).[3]

Um die Kompatibilität von Express mit Bun zu testen, wird eine einfache Anwendung genutzt, die die Create, Read, Update, Delete-Operationen (CRUD-Operationen) für Büchern und Autoren über eine API ermöglicht . Die API verwendet Express in Verbindung mit TypeScript. Sie wird um einige Middleware-Funktionalitäten erweitert, um die Migration repräsentativer zu gestalten. Dazu gehören:

- Winston als Logger,
- Helmet zum Erhöhen der Sicherheit durch Setzen von Header in der HTTP-

Antwort,

- Morgan zur Protokollierung der ein- und ausgehenden HTTP-Anfragen,
- Mongoose als Tool für das Object-Relational Mapping (ORM) zur Speicherung von Daten in MongoDB,
- Joi zur Validierung der Bücher und Autoren in den HTTP-Anfragen,
- Compression zur Komprimierung der HTTP-Antworten,
- ein zentraler Error-Handler, um bei Fehlern verständliche Nachrichten an den Benutzer zurückzugeben.

Um die App von Node.js auf Bun zu migrieren, muss Bun's integrierter Paketmanager verwendet werden, um die passenden Abhängigkeiten zu installieren. Node.js speichert die installierten Abhängigkeiten in der *package.json*, die auch von Bun's Paketmanager genutzt wird. Da die Migration aktuell manuell/per Hand durchgeführt werden muss, ist eine vollständige Neuinstallation der Abhängigkeiten erforderlich. Dabei werden die Ordner *node_modules* und die Datei *package-lock.json* gelöscht. In diesem Ordner speichert Node.js alle lokal installierten Abhängigkeiten. Node.js nutzt die *package-lock.json*, um die installierten Abhängigkeiten lokal zu speichern. Dies sorgt dafür, dass auf anderen Geräten die Versionen installiert werden, die während der Entwicklung verwendet wurden.
Im zweiten Schritt müssen veraltete Abhängigkeiten aus dem Projekt entfernt und weitere benötigte Abhängigkeiten hinzugefügt werden. Die Typdefinitionen von Node.js und die Pakete, die es unter Node.js ermöglichen, TypeScript auszuführen (siehe Kapitel 2.1), werden unter Bun nicht mehr benötigt. Als Abhängigkeit müssen die Typdefinitionen von Bun hinzugefügt werden. Diese werden auch in der Datei *ts-config.json* inkludiert, damit global auf Bun's API ohne Fehler zugegriffen werden kann. Diese Datei definiert Einstellungen und Optionen für den TypeScript-Transpiler. Die Werte der Konfigurationsoptionen, die sowohl in der aktuellen Datei als auch in Bun's Empfehlungen vorhanden sind, werden an die empfohlenen Werte angepasst [33].
Im dritten Schritt werden die Befehle in der *package.json* angepasst, die die Ausführung des Projekts ermöglichen. Zum Ausführen des Projekts wird ab sofort *bun –watch src/index.ts* genutzt. Zuletzt müssen die Pakete mit Bun's Paketmanager installiert werden, indem der Befehl *bun install* ausgeführt wird. Danach kann die Anwendung gestartet werden.

Der erste Start der App ist nicht erfolgreich. Die Konsolenausgabe zeigt den Fehler an *SyntaxError: Import named Request not found in module node_modules/express/-index.js*. Das bedeutet, dass der Compiler den Import des Typs *Request* aus dem Express-Framework nicht auflösen kann. Um die generelle Funktionsweise der App zu überprüfen, werden die Importe aller Typen aus dem Express-Framework entfernt. Ein erneuter Start der App bestätigt die generelle Funktionsweise. Da es sich um ein Problem mit den Typdefinitionen handelt, liegt die Vermutung nahe, dass eine Einstellung des TypeScript-Compilers die Ursache ist. Eine minimale Express-App mit der empfohlenen Konfiguration des TypeScript-Compilers zeigt, dass die App auch mit den verwendeten Typen funktioniert. Der Vergleich der Konfiguration des Compilers in beiden Projekten zeigt, dass im aktuellen Projekt *emitDecoratorMetadata* aktiviert ist. Diese Option steuert, ob der Compiler Metadaten über Dekoratoren im generierten JavaScript-Quellcode generiert. Ist diese Option aktiviert, kann zur Laufzeit auf die Metadaten der Dekoratoren zugegriffen werden. Mit deaktivierter Option startet die App. Das Absenden von HTTP-Anfragen an die Applikation zeigt, dass alle API-Operationen erfolgreich funktionieren. Offiziell unterstützt Bun diese Option seit der Version 1.0.3 (siehe Kapitel 4.2) [34].

Wenn man die App nun produktiv nutzen möchte, verwendet man einen Bundler, um mit dessen Hilfe einen minifizierten Quellcode zu erstellen. Bun bietet einen integrierten Bundler an (siehe Kapitel 2.2). Dieser wird durch den Befehl *bun build src/index.ts –outdir ./dist –target bun* erzeugt. Dabei ist es wichtig, Bun als Zielplattform zur Ausführung zu spezifizieren, da alternativ auch Node.js als Zielplattform genutzt werden kann. Führt man nun das Ergebnis des Bundlers aus, erscheint die folgende Fehlermeldung: *TypeError: d is not a function. (In d(target, key, r), d is an instance of Object)*. Die Analyse zeigt, dass ein verwendeter Dekorator nicht funktioniert. Dieser sorgt dafür, dass beim Binding der Listener für die API-Funktionen der this-Kontext innerhalb des Listeners funktioniert. Dies muss manuell über eine Anpassung des Quellcodes erledigt werden. Nun startet die App mit dem minifizierten Quellcode ohne Fehler. Das Absenden von HTTP-Anfragen an die Applikation zeigt, dass auch alle API-Operationen erfolgreich funktionieren. Offiziell handelt es sich bei der Option *experimentalDecorators* um ein experimentelles Feature, das von Node.js bereits unterstützt wird und von Bun noch nicht [35].

4.2 Nest

Nest ist ein Open-Source-Framework für die Entwicklung von serverseitigen Anwendungen und APIs in Node.js [36]. Es wurde entwickelt, um die Erstellung von skalierbaren, gut strukturierten und leicht wartbaren Anwendungen zu erleichtern. Nest basiert auf TypeScript und bietet eine Abstraktionsebene über gängige HTTP-Server wie z. B. Express an. Alternativ können Entwickler über APIs andere Server integrieren und benuzten. Bun unterstützt Nest noch nicht vollständig [37]. Mit Version 1.0.3 wird der TypeScript-Dekorator *emitDecoratorMetadata*, auf dem Nest aufbaut, unterstützt [34]. Dadurch wird der Support von Nest verbessert.

Zur Prüfung der Kompatibilität wird eine App verwendet, die es ermöglicht über eine API die CRUD-Operationen für Artikel anzuwenden. Der Autor ist ein Benutzer, der sich zuvor registrieren muss. Die Anwendung verwendet die folgenden Features:

- Rollenbasierte Authentifizierung und Autorisierung,
- ORM-Integration mit TypeORM,
- Logging mit Winston,
- Validierungen mit Joi.

Als Datenbank wird eine lokale PostgreSQL-Instanz verwendet. Den Verbindungsaufbau übernimmt TypeORM. Die Schritte der Migration entsprechen der Migration von Express (siehe Kapitel 4.1). Daher werden im Folgenden nur die Unterschiede betrachtet. Die Konfiguration des Compilers entspricht den Bun's Empfehlungen. Lediglich die Optionen, die Bun als Best Practices deklariert, werden nicht übernommen. Denn diese sind in den Best Practices von Nest explizit nicht enthalten.
Beim Start der App erscheint nun die Fehlermeldung *Error: Cannot find module node_modules/bcrypt/lib/binding/napi-v3/bcrypt_lib.node*. Das Paket *bcrypt* wird verwendet, um die Passwörter zu hashen und beim Login zu validieren. Bcrypt nutzt node-gyp als Abhängigkeit [38]. Dabei handelt es sich um ein Open-Source-Build-Tool, das zum Erstellen und Kompilieren nativer Node Module eingesetzt wird [12]. In diesem Kontext wird oft *postinstall* verwendet, eine Lifecycle-Methode beim Installieren der Abhängigkeiten. Im Anschluss an die Installation des Pakets wird ein Skript ausgeführt, um beispielsweise node-gyp auszuführen. Bun blockiert diese standardmäßig, da diese Skripte potentielle Sicherheitsrisiken darstellen [39]. Vertraut man einem Paket, muss man dieses in der *package.json* als vertraute Abhängigkeit deklarieren. Fügt man nun Bcrypt hinzu und installiert alle Pakete komplett neu, ist

der Fehler gelöst.
Die Konsolenausgabe zeigt beim nächsten Start der App einen Laufzeitfehler an: *ReferenceError: Cannot access uninitialized variable.*. Dies tritt an der Stellen auf, in denen die Relation zwischen den Entitäten *Article* und *User* definiert werden. Das führt zu einer zirkulären Abhängigkeit zwischen *User* und *Article*. Dies kann, wie in [40] beschrieben, vermieden werden, indem bei der Definition der Relation der Typ *User* durch *Relation<User>* ersetzt wird. Dann löst TypeORM die Definitionen entsprechend auf, ohne dabei zirkuläre Abhängigkeiten zu erzeugen.

Die Konsole zeigt beim erneuten Start der Applikation keine Fehler mehr. Sendet man nun eine HTTP-Anfrage an die API, wird die Fehlermeldung *Die Metadaten für die Entität User können nicht gefunden werden.* zurückgegeben. TypeORM sucht aktuell alle Entitäten über ein String, der ein Pattern für den Pfad der Entitäten definiert. Laut [41] ist diese Variante veraltet. Die Entitäten müssen direkt als Referenz übergeben werden, indem sie über einen Import inkludiert werden.
Startet man die App erneut und sendet eine HTTP-Anfrage, erhält man keine Fehlermeldungen mehr. Allerdings stürzt der Server mit der Meldung *Segmentation Fault* ab, sobald sich ein neuer Benutzer registriert oder ein vorhandener Benutzer einloggt. Über die Log-Einträge kann nachvollzogen werden, dass der Server abstürzt, sobald Bcrypt zum Hashen der Passwörter aufgerufen wird. Laut [42] funktioniert Bcrypt nicht, wenn es asynchron aufgerufen wird. Ein synchroner Aufruf der entsprechenden Funktionen verläuft allerdings erfolgreich. Da ein synchroner Aufruf dieser Funktion die Ausführung anderer Aufgaben blockiert, ist diese Lösung nicht optimal. Bun bietet für diesen Zweck eine eigene API an. Nutzt man diese in der Applikation, sind auch die HTTP-Anfragen zum Registrieren oder Einloggen von Benutzern erfolgreich. Die anderen Endpunkte der API funktionieren auch. Daraus folgt, dass Nest unter Bun funktioniert. Allerdings bedeutet der Wechsel auf die API von Bun, dass das Versprechen eines Eins-zu-Eins-Ersatzes für Node.js nicht gehalten werden kann, da die Migration aktive Eingriffe in die Abhängigkeiten der Applikation erfordert.

Im letzten Test wird, wie bei Express zuvor, versucht die minifizierte Version der Anwendung zu erzeugen und lokal auszuführen. Sobald man den Bundler aufruft, zeigt dieser zahlreiche Fehlermeldungen an. Der Bundler meldet viele Abhängigkeiten, die er nicht auflösen kann. Auffällig ist, dass die als fehlend deklarierten Abhängigkeiten in den direkten Abhängigkeiten des Projekts nicht vorhanden sind. [43] zeigt, dass bereits andere Entwickler mit demselben Problem konfrontiert sind. Da dieses Problem derzeit ungelöst ist, funktioniert der Bundler für Nest nicht. Für die Lösung des

Problems wird ein Update für Bun's Bundler benötigt.

4.3 Fazit

Dieses Kapitel beantwortet die zweite Leitfrage „Inwiefern sind Node.js-Projekte kompatibel mit Bun? Wie schwierig gestaltet sich die Migration?“ (siehe Kapitel 1.2). Die Migration von Node.js-Projekten auf Bun kann erfolgreich sein, erfordert jedoch in der Regel Anpassungen in den Abhängigkeiten und Konfigurationen. Der Aufwand hängt stark von der Applikation und den verwendeten Abhängigkeiten ab. Beispielsweise erfordert die Umstellung der Nest-Applikation zusätzliche Anpassungen, insbesondere bei der asynchronen Verwendung von Bcrypt. Obwohl die Anwendung unter Bun funktioniert, ist festzustellen, dass die Migration aktive Eingriffe in die Abhängigkeiten der Anwendung erfordert. Zusätzlich funktioniert der Bundler nicht. Die Tatsache, dass die minifizierte Version der Nest-Anwendung unter Bun nicht funktionierte, wirft Fragen zur Stabilität und Zuverlässigkeit von Bun auf. Es bleibt abzuwarten, ob zukünftige Updates diese Probleme beheben können.

Es gilt anzumerken, dass die verwendeten Beispiele einfachere Applikationen waren. Produktiv genutzte Anwendungen können komplexer sein und auch mehr Abhängigkeiten verwenden. Beide Beispiele beweisen, dass häufig verwendete Komponenten und Konzepte funktionieren. Dazu zählen Authentifizierung und Autorisierung, Logging, CRUD-Operationen für Entitäten, Integration mit einer Datenbank und auch Schutzmechanismen für die Applikation selbst. Auf diesen Konzepten bauen auch produktiv genutzte Anwendungen auf. Sobald es über diese Konzepte hinausgeht, muss die Funktionsweise unter Bun separat analysiert werden.

Kapitel 5

Schlussbetrachtung

Im letzten Kapitel dieser Arbeit werden die Ergebnisse der vorangegangenen Kapitel zusammengefasst und reflektiert. Darauffolgend wird ein Ausblick auf die Zukunft von Bun und über mögliche weitergehende Forschungsthemen gegeben.

5.1 Fazit

Diese Arbeit evaluiert Bun im Vergleich zu dem etablierten Standard Node.js. Dabei liegt der Fokus auf zwei Hauptaspekten: Performance und Kompatibilität. Die Performance wurde durch mehrere Benchmarks analysiert, während die Kompatibilität durch die Migration existierender Node.js-Projekte zu Bun überprüft wurde.

Die Performance-Tests beantworten die erste der drei Leitfragen „Welche konkreten Leistungsverbesserungen können in Bun 1.0 im Vergleich zu Node.js festgestellt werden, und wie lassen sie sich quantifizieren?“ (siehe Kapitel 1.2). Die Ergebnisse zeigen, dass Bun in allen Szenarien eine signifikant bessere Leistung als Node.js aufweist (siehe Kapitel 3.4). Dies äußert sich in einer verringerten Latenz, einer geringeren Inanspruchnahme von Arbeitsspeicher und CPU-Ressourcen. Dies gilt sowohl für die Verarbeitung von Netzwerkanfragen als auch für die Ausführung rechenintensiver Aufgaben. Bun besitzt das Potenzial, in realen Szenarien schneller zu sein als Node.js. Insbesondere die effiziente Nutzung der Ressourcen besitzt Auswirkungen auf die Skalierbarkeit und Kosten in Produktionsumgebungen.

Die zweite Leitfrage „Inwiefern sind Node.js-Projekte kompatibel mit Bun? Wie schwierig gestaltet sich die Migration?“ (siehe Kapitel 1.2) beschäftigt sich mit der Kompatibilität von bestehenden Node.js-Projekten. Die Analyse (siehe Kapitel 4) zeigt, dass die Migration bestehender Projekte in der Regel Anpassungen in den

Abhängigkeiten und Konfigurationen erfordert. Der Aufwand dafür hängt von der spezifischen Anwendung und den genutzten Abhängigkeiten ab. Die Migration der untersuchten Express- und Nest-Anwendungen war erfolgreich, erforderte aber aktive Eingriffe. Darüber hinaus führt die Verwendung des Bundlers im Nest-Projekt zu Fehlern. Das wirft Fragen zur Zuverlässigkeit und Stabilität von Bun auf.

Zusammenfassend lässt sich die dritte Leitfrage „Welche Herausforderungen und Vorteile ergeben sich bei der Verwendung von Bun 1.0 im Vergleich zu Node.js für Entwickler und Projekte?“ (siehe Kapitel 1.2) wie folgt beantworten: Bun ist aufgrund seiner hohen Performance eine gute Alternative im Vergleich zu Node.js. Allerdings benötigt Bun etwas Zeit, um offene Fehler zu lösen und die Kompatibilität mit der Node.js-API und anderen Frameworks zu verbessern. Für neue Projekten ohne zusätzliche Frameworks, kann Bun aufgrund der besseren Performance bereits in Betracht gezogen werden.

5.2 Ausblick

In zukünftigen Forschungsarbeiten kann es interessant sein, die Performance und Kompatibilität von tatsächlich produktiv verwendeten Applikationen zu analysieren. Diese weisen oft komplexere Lösungen auf und verwenden bereits ein Framework oder andere Bibliotheken, das die Performance der verwendeten Laufzeitumgebung beeinflussen kann. Darüber hinaus können weitere Aspekte, wie die Sicherheit, Stabilität oder Bun's Paketmanager betrachtet werden. Zusätzlich können die Performance und Kompatibilität von Bun's Bundler und der integrierten Bibliothek zum Testen analysiert und mit etablierten Lösungen verglichen werden. Dadurch ist eine komplette Bewertung von Bun mit allen integrierten Komponenten möglich.

Schließlich bleibt abzuwarten, wie sich Bun weiterentwickelt und ob es sein Versprechen hinsichtlich Performance und Kompatibilität auch in zukünftigen Versionen erfüllt. Neue Updates können die identifizierten Probleme lösen und die Kompatibilität verbessern. Denn die Roadmap für die Entwicklung besitzt viele offene Punkte [44]. Gleichzeitig existieren zahlreiche offene Fehler, beispielsweise die Funktionsfähigkeit des Bundlers (siehe Kapitel 4.2). Daher ist es sinnvoll, die Leistung und den Migrationsprozess in regelmäßigen Abständen neu zu analysieren, um die Entwicklungsfortschritte zu bewerten.

Literaturverzeichnis

[1] MICROSOFT. *TypeScript is JavaScript with syntax for types.* o. J. URL: https://www.typescriptlang.org/.

[2] STACK OVERFLOW. *Stack Overflow Developer Survey 2023*, 2023. URL: https://survey.stackoverflow.co/2023/#overview.

[3] ETHAN BROWN. *Web development with Node and Express : leveraging the JavaScript stack*, Beijing, November 2019. URL: https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=2295093.

[4] Q-SUCCESS, Hrsg. *Usage statistics of JavaScript for websites*, 2023. URL: https://w3techs.com/technologies/details/pl-js.

[5] SACHA GREIF und ERIC BUREL. *State of JavaScript 2022*, 2022. URL: https://2022.stateofjs.com/en-US/other-tools/.

[6] JARED SUMNER und PARTOVI, ASHCON, MCDONNEL, COLIN. *Bun 1.0*, 2023. URL: https://bun.sh/blog/bun-v1.0.

[7] OVEN-SH, Hrsg. *Bun is a fast JavaScript all-in-one toolkit*, 2023. URL: https://bun.sh/.

[8] OPENJS FOUNDATION, Hrsg. *An introduction to the NPM package manager*, 2022. URL: https://nodejs.org/en/learn/getting-started/an-introduction-to-the-npm-package-manager.

[9] TEJAS KANERIYA. *What is node.js? where when & how to use it (with examples)*, 2022. URL: https://www.simform.com/blog/what-is-node-js/.

[10] SEBASTIAN SPRINGER. *Node.js : das umfassende Handbuch*. 4., aktualisierte und erweiterte Auflage. Rheinwerk Computing. Bonn: Rheinwerk, 2022. ISBN: 9783836287654. URL: http://deposit.dnb.de/cgi-bin/dokserv?id=992e511c601d4a5f84179bebaa309635&prov=M&dok_var=1&dok_ext=htm.

[11] OPENJS FOUNDATION, Hrsg. *Introduction to Node.js*, 2022. URL: https://nodejs.dev/en/learn/.

[12] OPENJS FOUNDATION. *Node.js API reference documentation*, o. J. URL: https://nodejs.org/api/documentation.html.

[13] OPENJS FOUNDATION. *Node.js*, o. J. URL: https://nodejs.org/en.

[14] NIMESH CHHETRI. *A comparative analysis of node. js (server-side javascript)*. 2016. URL: https://repository.stcloudstate.edu/csit_etds/5/.

[15] MATTHEW TYSON. *Explore bun.js: The all-in-one JavaScript runtime*, 2023. URL: https://www.infoworld.com/article/3688330/explore-bunjs-the-all-in-one-javascript-runtime.html.

[16] OVEN-SH, Hrsg. *Offizielle Dokumentation*, 2023. URL: https://bun.sh/docs.

[17] ZIG SOFTWARE FOUNDATION. *Why Zig When There is Already C++, D, and Rust?*, o. J. URL: https://ziglang.org/learn/why_zig_rust_d_cpp/#a-package-manager-and-build-system-for-existing-projects.

[18] OVEN-SH. *Why zig*, 2022. URL: https://github.com/oven-sh/bun/discussions/994.

[19] APPLE, Hrsg. *WebKit*, o. J. URL: https://webkit.org/.

[20] APPLE. *WebKit Documentation - JavaScriptCore*, o.J. URL: https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html.

[21] PAULO SILVA, DANIEL FIREMAN und THIAGO EMMANUEL PEREIRA. „Prebaking Functions to Warm the Serverless Cold Start“. In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, S. 1–13. ISBN: 9781450381536. DOI: 10.1145/3423211.3425682.

[22] *ISO/IEC 25010*, 2022. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[23] ISO / IEC. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*, URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en.

[24] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*, Piscataway, NJ, USA. DOI: 10.1109/IEEESTD.1990.101064.

[25] MARIO BARBACCI u. a. „Quality Attributes“. In: (1995).

[26] MAKSIM FEDOSEEV. *Bombardier*, 2016. URL: https://github.com/codesenberg/bombardier.

[27] FREE SOFTWARE FOUNDATION. *GNU Time*, 2018. URL: https://www.gnu.org/software/time/.

[28] BERT VERHELST. *Add note that windows binaries are not yet available*, 2023. URL: https://github.com/oven-sh/bun/pull/4780.

[29] JARED SUMNER. *Bun v1.0.6*, 2023. URL: https://bun.sh/blog/bun-v1.0.6.

[30] OPENJS FOUNDATION. *The Event Loop*, o. J. URL: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick#what-is-the-event-loop.

[31] OPENJS FOUNDATION. *Node.js 21 ChangeLog*, 2023. URL: https://github.com/nodejs/node/blob/main/doc/changelogs/CHANGELOG_V21.md#21.0.0.

[32] OPENJS FOUNDATION. *Node.js 20 ChangeLog*, 2023. URL: https://github.com/nodejs/node/blob/main/doc/changelogs/CHANGELOG_V20.md.

[33] OVEN-SH, Hrsg. *TypeScript*, 2023. URL: https://bun.sh/docs/typescript.

[34] COLIN MCDONNEL. *Bun v1.0.3*, 2023. URL: https://bun.sh/blog/bun-v1.0.3.

[35] MICROSOFT, Hrsg. *TSConfig Reference*, 2023. URL: https://www.typescriptlang.org/tsconfig#experimentalDecorators.

[36] KAMIL MYSLIWIEC. *Introduction*, 2023. URL: https://docs.nestjs.com/.

[37] JARRED SUMNER. *Support NestJS*, 2022. URL: https://github.com/oven-sh/bun/issues/1641.

[38] NICOLAS DEL GOBBO. *node.bcrypt.js*, 2018. URL: https://github.com/kelektiv/node.bcrypt.js.

[39] OVEN-SH. *Add a trusted dependency with Bun*, 2023. URL: https://bun.sh/guides/install/trusted.

[40] TYPEORM. *TypeORM - Dokumentation*, URL: https://typeorm.io/#relations-in-esm-projects.

[41] TYPEORM. *Changelog*, 2021. URL: https://github.com/typeorm/typeorm/blob/master/CHANGELOG.md#deprecations.

[42] JARED SUMNER. *bcrypt async functions segfault*, 2023. URL: https://github.com/oven-sh/bun/issues/3201.

[43] KRYSTIAN POSTEK. *Unable to build NestJS based project*, 2023. URL: https://github.com/oven-sh/bun/issues/4803.

[44] JARED SUMNER. *Bun's Roadmap*, 2022. URL: https://github.com/oven-sh/bun/issues/159.

Anhang A

Ergebnisse des Benchmarks

A.1 HTTP-Server

Tabelle A.1: HTTP-Server - Ergebnisse von Bun auf macOS
Quelle: Eigene Darstellung

Nr.	Req/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
1	75.482,69	6,62	191,22	100,00	83,00	28.928,00
2	75.978,26	6,58	162,98	100,00	83,00	28.960,00
3	76.467,36	6,54	168,01	100,00	81,00	28.832,00
4	72.942,36	6,85	123,28	100,00	83,00	28.992,00
5	77.558,43	6,45	194,28	100,00	83,00	28.800,00

Nr.	Req/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
Durchschnitt	75.685,82	6,61	167,70	100,00	82,60	28.902,40

Tabelle A.2: HTTP-Server - Ergebnisse von Node.js LTS auf macOS
Quelle: Eigene Darstellung

Nr.	Req/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
1	46.037,35	10,36	180,79	100,00	84,00	93.536,00
2	46.323,80	10,86	202,56	100,00	84,00	94.432,00
3	45.762,30	10,92	199,71	100,00	85,00	93.456,00
4	45.035,39	11,10	195,63	100,00	86,00	93.584,00
5	45.776,81	10,92	144,27	100,00	85,00	94.016,00
Durchschnitt	45.787,13	10,83	184,59	100,00	84,80	93.804,80

Tabelle A.3: HTTP-Server - Ergebnisse von Node.js Latest auf macOS
Quelle: Eigene Darstellung

Nr.	Req/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
1	45.680,37	10,94	1840,00	100,00	86,00	85.584,00
2	45.880,89	10,89	1860,00	100,00	86,00	85.520,00
3	45.939,12	10,88	1570,00	100,00	85,00	84.386,00
4	45.932,14	10,88	1650,00	100,00	86,00	83.616,00

Nr.	Req/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
5	45.974,99	10,87	1120,00	100,00	85,00	85.680,00
Durchschnitt	45.881,50	10,89	1608,00	100,00	85,60	84.957,20

Tabelle A.4: HTTP-Server - Ergebnisse von Bun auf Ubuntu 23.10
Quelle: Eigene Darstellung

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
1	103.290,25	4,84	92,06	100,00	89,00	47.188,00
2	102.852,75	4,86	120,86	100,00	97,00	47.608,00
3	102.414,80	4,88	98,60	100,00	96,00	49.392,00
4	103.809,80	4,81	104,46	100,00	98,00	47.260,00
5	103.707,81	4,82	99,36	100,00	97,00	48.484,00
Durchschnitt	103.215,08	4,84	103,07	100,00	95,40	47.986,40

Tabelle A.5: HTTP-Server - Ergebnisse von Node.js LTS auf Ubuntu 23.10
Quelle: Eigene Darstellung

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
1	30.421,51	16,44	230,80	100,00	96,00	93.856,00
2	30.448,93	16,42	208,43	100,00	96,00	93.212,00
3	29.999,89	16,67	215,64	100,00	95,00	93.144,00

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
4	30.122,16	16,59	176,54	100,00	94,00	92.928,00
5	30.488,92	16,39	199,07	100,00	96,00	93.116,00
Durchschnitt	30.296,28	16,50	206,10	100,00	95,40	93.251,20

Tabelle A.6: HTTP-Server - Ergebnisse von Node.js Latest auf Ubuntu 23.10
Quelle: Eigene Darstellung

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
1	32.358,13	15,45	7960,00	100,00	100,00	86.652,00
2	31.700,99	15,78	8340,00	100,00	90,00	86.396,00
3	32.244,05	15,51	9280,00	100,00	95,00	86.280,00
4	32.856,68	15,21	8210,00	100,00	99,00	86.284,00
5	32.269,21	15,50	8440,00	100,00	98,00	86.636,00
Durchschnitt	32.285,81	15,49	8446,00	100,00	96,40	86.449,60

A.2 Datei-Server

Tabelle A.7: Datei-Server - Ergebnisse von Bun auf Ubuntu 23.10
Quelle: Eigene Darstellung

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
			50 Benutzer			
1	24.555,25	2,03	25,73	100,00	98,00	83.484,00
2	24.688,31	2,02	40,45	100,00	100,00	82.824,00
3	25.083,61	1,99	30,04	100,00	98,00	83.876,00
4	24.978,84	2,00	25,65	100,00	101,00	83.756,00
5	24.503,56	2,04	29,26	100,00	98,00	82.056,00
Durchschnitt	24.761,91	2,02	30,23	100,00	99,00	83.199,20
			250 Benutzer			
1	24.585,85	10,17	100,97	100,00	98,00	82.372,00
2	24.988,13	10,00	96,19	100,00	97,00	83.272,00
3	24.290,11	10,29	73,98	100,00	97,00	83.132,00
4	24.399,87	10,24	105,26	100,00	98,00	82.808,00
5	24.926,90	10,03	80,57	100,00	97,00	84.088,00
Durchschnitt	24.638,17	10,15	91,39	100,00	97,40	83.134,40
			500 Benutzer			

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
1	23.899,18	20,91	116,19	100,00	96,00	82.128,00
2	24.201,74	20,65	108,40	100,00	96,00	83.196,00
3	24.153,93	20,69	115,38	100,00	100,00	82.692,00
4	23.658,70	21,12	119,64	100,00	98,00	82.672,00
5	24.399,08	20,50	117,62	100,00	99,00	82.948,00
Durchschnitt	24.155,31	20,69	116,36	100,00	98,40	83.124,80
1000 Benutzer						
1	23.963,96	41,81	1230,00	100,00	98,00	82.180,00
2	23.684,71	42,25	1160,00	100,00	95,00	81.876,00
3	23.702,16	42,25	1170,00	100,00	93,00	83.052,00
4	23.546,82	42,53	1160,00	100,00	98,00	82.880,00
5	24.036,35	41,63	1130,00	100,00	97,00	83.320,00
Durchschnitt	23.786,80	42,09	1170,00	100,00	96,20	82.661,60

Tabelle A.8: Datei-Server - Ergebnisse von Node.js LTS auf Ubuntu 23.10
Quelle: Eigene Darstellung

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
50 Benutzer						
1	11.853,46	4,22	78,54	100,00	137,00	69.480,00
2	11.924,96	4,08	71,56	100,00	135,00	71.184,00

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
3	12.253,47	4,08	71,51	100,00	135,00	74.640,00
4	11.806,59	4,23	64,23	100,00	134,00	71.224,00
5	11.819,12	4,23	73,19	100,00	133,00	76.260,00
Durchschnitt	11.931,52	4,17	71,81	100,00	134,80	72.557,60
			250 Benutzer			
1	9454,04	26,38	249,57	100,00	169,00	115.504,00
2	9395,36	26,55	221,37	100,00	169,00	121.036,00
3	9276,20	26,89	215,68	100,00	173,00	112.908,00
4	9379,20	26,59	192,39	100,00	167,00	122.620,00
5	9362,02	26,64	223,04	100,00	171,00	114.472,00
Durchschnitt	9373,36	26,61	220,41	100,00	169,80	117.308,00
			500 Benutzer			
1	7650,31	64,53	287,70	100,00	166,00	105.356,00
2	8460,46	58,41	305,46	100,00	176,00	106.928,00
3	8428,35	58,51	285,69	100,00	166,00	103.704,00
4	8292,37	59,61	313,21	100,00	171,00	106.108,00
5	8410,83	58,73	292,88	100,00	165,00	104.264,00
Durchschnitt	8398,00	58,82	299,31	100,00	169,50	105.251,00
			1000 Benutzer			
1	8491,34	118,31	1260,00	100,00	170,00	140.472,00

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
2	7656,16	130,42	1260,00	100,00	178,00	143.180,00
3	7714,78	129,42	1270,00	100,00	160,00	178.588,00
4	8174,67	123,02	1270,00	100,00	152,00	149.484,00
5	7698,82	129,62	1260,00	100,00	177,00	147.612,00
Durchschnitt	7947,15	126,16	1264,00	100,00	167,40	151.867,20

Tabelle A.9: Datei-Server - Ergebnisse von Node.js Latest auf Ubuntu 23.10
Quelle: Eigene Darstellung

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
50 Benutzer						
1	14.237,06	3,51	116,84	100,00	174,00	88.056,00
2	14.305,85	3,49	135,14	100,00	194,00	84.784,00
3	14.685,70	3,40	146,92	100,00	193,00	82.028,00
4	14.253,49	3,51	161,68	100,00	193,00	85.144,00
5	14.690,78	3,40	127,55	100,00	197,00	86.808,00
Durchschnitt	14.434,58	3,46	137,63	100,00	190,20	85.364,00
250 Benutzer						
1	12.075,16	20,70	860,00	100,00	222,00	128.460,00
2	12.006,80	20,81	816,63	100,00	218,00	124.552,00
3	12.038,16	20,76	880,00	100,00	220,00	121.192,00

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
4	11.927,89	20,95	845,54	100,00	221,00	121.700,00
5	12.158,18	20,56	860,00	100,00	221,00	121.488,00
Durchschnitt	12.041,24	20,76	852,43	100,00	220,40	123.478,40
500 Benutzer						
1	10.322,75	47,96	1710,00	100,00	268,00	172.996,00
2	10.462,03	47,33	1740,00	100,00	265,00	118.604,00
3	10.337,76	47,97	1770,00	100,00	250,00	174.576,00
4	10.478,85	47,32	1740,00	100,00	264,00	175.628,00
5	10.254,00	48,39	1740,00	100,00	262,00	175.036,00
Durchschnitt	10.383,16	47,75	1747,50	100,00	260,25	160.961,00
1000 Benutzer						
1	8535,43	116,42	4240,00	100,00	278,00	213.864,00
2	8553,51	116,28	3810,00	100,00	279,00	209.776,00
3	8573,22	115,88	4240,00	100,00	277,00	209.276,00
4	8655,02	114,96	4100,00	100,00	281,00	211.692,00
5	8630,10	115,30	4070,00	100,00	283,00	213.220,00
Durchschnitt	8589,46	115,77	4092,00	100,00	279,60	211.565,60

Tabelle A.10: Datei-Server - Ergebnisse von Bun auf macOS
Quelle: Eigene Darstellung

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
50 Benutzer						
1	25.276,77	1,98	18,82	100,00	92,00	67.328,00
2	25.302,77	1,97	14,40	100,00	94,00	65.744,00
3	25.293,36	1,98	14,06	100,00	95,00	65.840,00
4	25.292,46	1,98	13,51	100,00	93,00	65.956,00
5	25.330,78	1,97	16,10	100,00	94,00	654.230,00
Durchschnitt	25.299,23	1,98	15,38	100,00	93,60	183.819,60
250 Benutzer						
1	25.464,61	9,82	55,58	100,00	89,00	66.698,00
2	25.435,68	9,83	52,12	100,00	92,00	66.384,00
3	25.476,74	9,81	50,91	100,00	93,00	66.464,00
4	25.372,19	9,85	49,03	100,00	92,00	66.640,00
5	25.451,01	9,82	49,88	100,00	93,00	66.560,00
Durchschnitt	25.440,05	9,83	51,50	100,00	91,80	66.549,20
500 Benutzer						
1	23.707,59	21,08	195,63	100,00	90,00	64.264,00
2	23.920,81	20,90	239,50	100,00	86,00	65.040,00
3	23.909,89	20,90	204,46	100,00	90,00	65.024,00

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
4	23.825,78	20,98	206,59	100,00	89,00	64.880,00
5	23.879,56	20,93	220,08	100,00	90,00	64.880,00
Durchschnitt	23.848,73	20,96	213,25	100,00	89,00	64.817,60
1000 Benutzer						
1	24.846,18	40,22	451,41	99,96	91,00	68.192,00
2	24.776,04	40,34	466,86	99,95	90,00	67.472,00
3	24.789,88	40,38	479,30	99,95	91,00	67.280,00
4	24.746,30	40,38	479,30	99,96	90,00	6824,00
5	24.773,90	40,34	462,65	99,96	91,00	67.952,00
Durchschnitt	24.786,46	40,33	467,90	99,96	90,60	55.544,00

Tabelle A.11: Datei-Server - Ergebnisse von Node.js LTS auf macOS
Quelle: Eigene Darstellung

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
50 Benutzer						
1	21.515,00	2,32	33,96	100,00	146,00	120.544,00
2	19.785,30	2,53	36,05	100,00	142,00	125.616,00
3	19.363,94	2,58	33,90	100,00	143,00	124.096,00
4	21.722,08	2,30	29,84	100,00	143,00	109.568,00
5	21.449,79	2,33	33,51	100,00	145,00	119.792,00

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
Durchschnitt	20.767,22	2,41	33,45	100,00	143,80	119.923,20
			250 Benutzer			
1	15.917,54	15,55	70,35	100,00	193,00	195.552,00
2	16.075,37	15,51	81,90	100,00	186,00	168.992,00
3	16.074,26	15,55	65,21	100,00	189,00	191.920,00
4	16.111,43	15,51	81,90	100,00	188,00	189.536,00
5	15.924,32	15,70	70,46	100,00	191,00	176.272,00
Durchschnitt	16.020,58	15,56	73,96	100,00	189,40	184.454,40
			500 Benutzer			
1	14.317,65	34,93	248,08	100,00	189,00	229.776,00
2	14.508,20	34,46	231,62	100,00	197,00	198.496,00
3	14.466,72	34,56	214,82	100,00	185,00	208.496,00
4	14.610,55	34,21	198,59	100,00	176,00	212.880,00
5	14.647,90	34,12	238,61	100,00	190,00	209.896,00
Durchschnitt	14.510,20	34,46	226,34	100,00	187,40	211.908,80
			1000 Benutzer			
1	14.970,63	66,75	511,17	99,92	193,00	284.000,00
2	14.629,17	68,36	452,81	99,93	198,00	300.096,00
3	14.524,19	68,75	594,74	99,92	195,00	286.704,00
4	14.274,59	69,98	572,31	99,92	199,00	294.704,00

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
5	13.378,25	74,77	550,02	99,92	190,00	311.296,00
Durchschnitt	14.355,37	69,72	536,21	99,92	195,00	295.360,00

Tabelle A.12: Datei-Server - Ergebnisse von Node.js Latest auf macOS
Quelle: Eigene Darstellung

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
50 Benutzer						
1	21.553,66	2,32	81,25	100,00	146,00	117.296,00
2	21.499,51	2,32	61,80	100,00	148,00	109.888,00
3	21.808,63	2,29	61,93	100,00	146,00	112.464,00
4	21.799,37	2,29	63,66	100,00	144,00	119.520,00
5	21.519,05	2,32	60,17	100,00	148,00	115.120,00
Durchschnitt	21.636,04	2,31	65,76	100,00	146,40	114.857,60
250 Benutzer						
1	17.388,28	14,38	469,75	99,99	190,00	203.264,00
2	17.301,54	14,45	484,46	99,99	191,00	202.544,00
3	17.374,50	14,39	455,17	100,00	186,00	184.224,00
4	17.193,03	14,54	478,60	99,99	186,00	208.992,00
5	16.918,40	5,55	492,50	100,00	187,00	198.144,00

Nr.	Request/s	Durchschn. Latenz (ms)	Max. Latenz (ms)	Erfolgr. Req (%)	Max. CPU-Ausl. (%)	Max. RAM (kB)
Durchschnitt	17.235,15	12,66	476,10	99,99	188,00	199.433,60
			500 Benutzer			
1	14.790,27	33,81	1720,00	99,50	195,00	221.728,00
2	14.729,20	33,93	1790,00	99,40	194,00	219.552,00
3	14.749,95	33,91	1649,00	99,70	198,00	234.800,00
4	14.722,22	33,98	1920,00	99,40	193,00	243.920,00
5	14.628,28	34,14	1820,00	99,70	194,00	257.200,00
Durchschnitt	14.723,98	33,95	1779,80	99,54	194,80	235.440,00
			1000 Benutzer			
1	14.937,53	66,91	6000,00	99,87	210,00	283.264,00
2	14.518,01	68,83	5140,00	99,48	206,00	366.512,00
3	14.594,26	67,85	4350,00	99,46	203,00	317.392,00
4	14.719,55	68,20	4540,00	99,38	198,00	361.280,00
5	14.651,76	68,20	4540,00	99,50	206,00	323.952,00
Durchschnitt	14.684,22	68,00	4914,00	99,54	204,60	330.480,00

A.3 Berechnung der Fibonacci-Folge

Tabelle A.13: Berechnung der Fibonacci-Folge - Ergebnisse auf macOS
Quelle: Eigene Darstellung

Nr.	Ausführungszeit (s)	CPU-Auslastung	Maximale RAM-Nutzung (kbytes)
		Bun	
1	3,23	100	25.840,00
2	3,23	100	25.776,00
3	3,24	100	25.904,00
4	3,26	100	26.160,00
5	3,26	100	25.952,00
Durchschnitt	3,24	100,00	25.926,40
		Node.js LTS	
1	6,56	99	42.496,00
2	6,50	99	40.976,00
3	6,48	99	40.848,00
4	6,50	99	42.448,00
5	6,49	99	43.120,00
Durchschnitt	6,51	99,00	41.977,60
		Node.js Latest	
1	6,32	99	36.688,00
2	6,32	99	36.608,00
3	6,32	99	36.912,00

Nr.	Ausführungszeit (s)	CPU-Auslastung	Maximale RAM-Nutzung (kbytes)
4	6,32	99	36.800,00
5	6,33	99	36.672,00
Durchschnitt	6,32	99,00	36.736,00

Tabelle A.14: Berechnung der Fibonacci-Folge - Ergebnisse auf Ubuntu 23.10
Quelle: Eigene Darstellung

Nr.	Ausführungszeit (s)	CPU-Auslastung	Maximale RAM-Nutzung (kbytes)
		Bun	
1	4,48	100	44.624,00
2	4,49	100	45.644,00
3	4,46	100	45.380,00
4	4,47	100	44.744,00
5	4,46	100	43.844,00
Durchschnitt	4,47	100,00	44.847,20
		Node.js LTS	
1	7,39	98	45.696,00
2	7,29	99	46.336,00
3	7,35	100	46.336,00
4	7,29	99	46.336,00
5	7,41	100	46.080,00

Nr.	Ausführungszeit (s)	CPU-Auslastung	Maximale RAM-Nutzung (kbytes)
Durchschnitt	7,35	99,20	46.156,80
		Node.js Latest	
1	7,19	100	43.904,00
2	7,17	100	43.904,00
3	7,17	100	43.776,00
4	7,17	99	43.776,00
5	7,20	99	43.776,00
Durchschnitt	7,18	99,60	43.827,20

A.4 Zusätzliche Diagramme

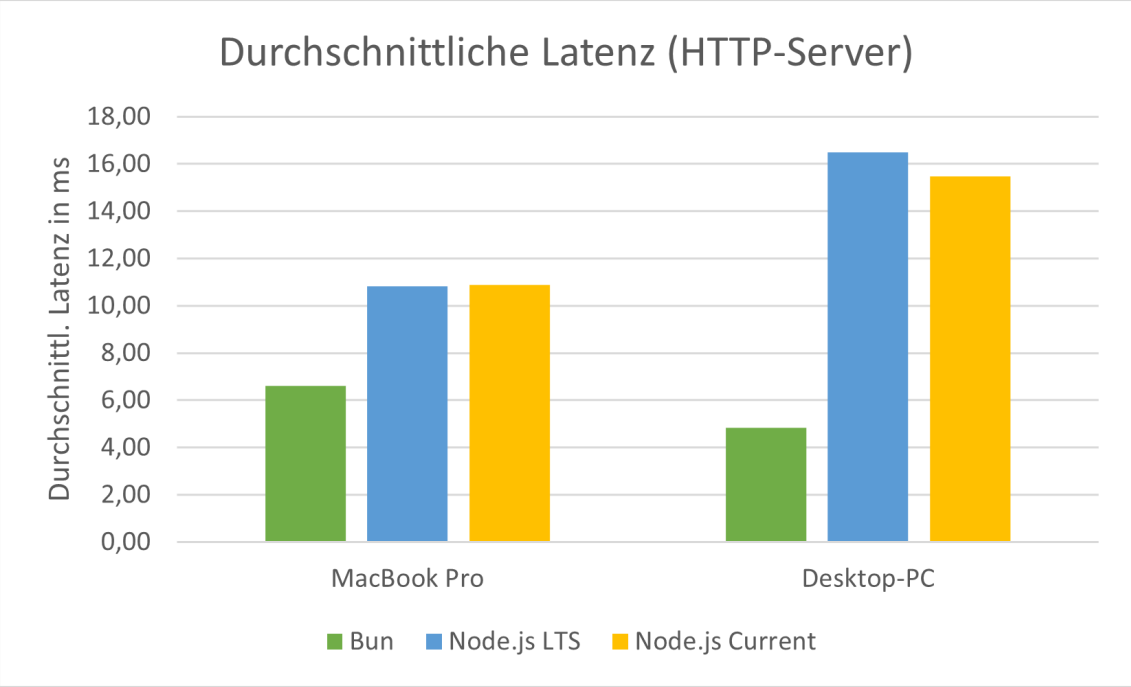

Abbildung A.1: HTTP-Server - Durchschnittliche Latenz
Quelle: Eigene Darstellung

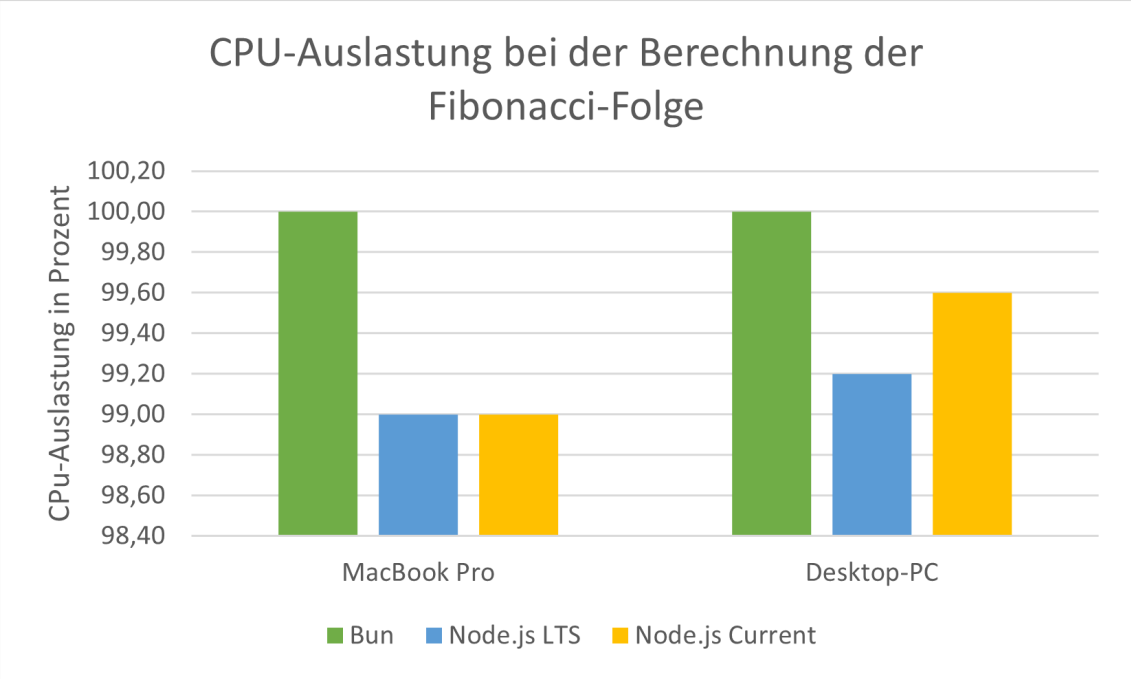

Abbildung A.2: Berechnung der Fibonacci-Folge - CPU-Auslastung
Quelle: Eigene Darstellung

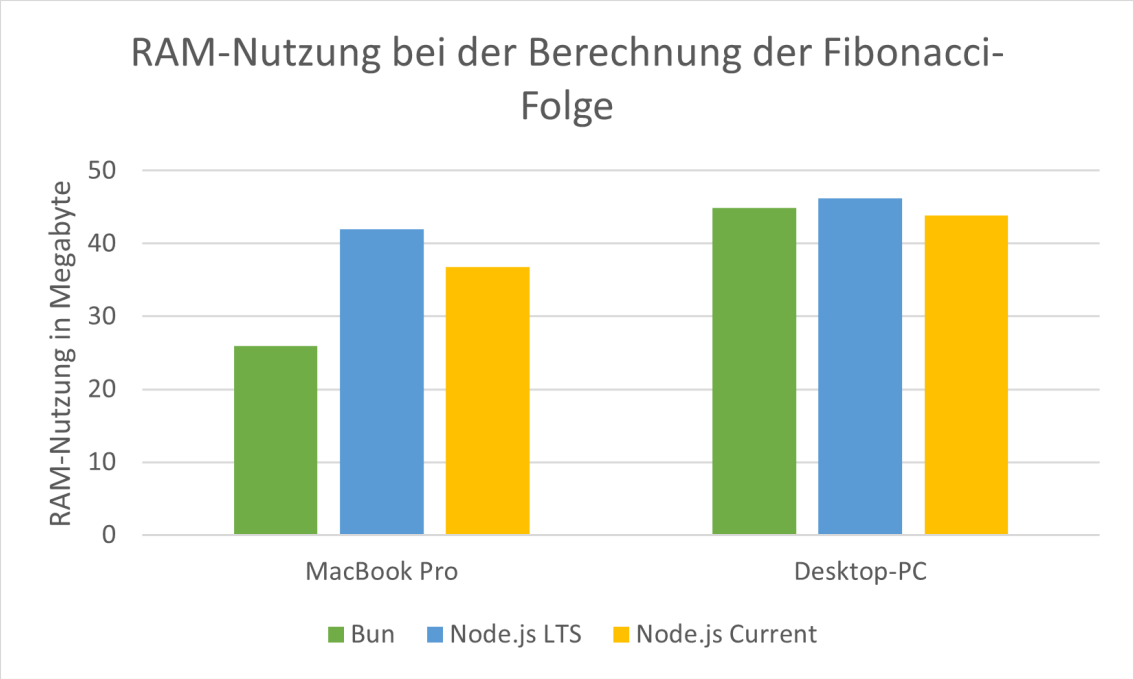

Abbildung A.3: Berechnung der Fibonacci-Folge - RAM-Nutzung
Quelle: Eigene Darstellung