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

Eidesstattliche Erklärung

Ich versichere, dass ich diese Masterthesis selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

(Ort, Datum)	(Ansgar Lichter)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

PoC Proof of Concept

Kapitel 1

Einleitung

1.1 Motivation

Die kontinuierliche Evolution von Softwaretechnologien und -umgebungen hat einen erheblichen Einfluss auf die Entwicklung und Leistung von Anwendungen. Kürzlich wurde die Version 1.0 der JavaScript Laufzeitumgebung Bun veröffentlicht, die vielversprechende Verbesserungen in Bezug auf die Leistung im Vergleich zu Node.js ankündigt [1].
JavaScript ist eine Programmiersprache, die vor allem im Kontext der Web-Entwicklung verwendet wird [2]. Aktuell erfreut sich JavaScript großer Beliebtheit. In einer Umfrage an Entwickler von Stack Overflow wurden mehr als 89.000 Entwickler befragt. JavaScript ist zum 11. Jahr in Folge die am häufigsten verwendete Programmiersprache. Mehr als 63% der befragten Entwickler haben JavaScript als beliebteste Technologie gewählt. Bei den professionellen Entwicklern ist der Anteil mit mehr als 65% sogar noch höher. Außerdem ist TypeScript, eine stark typisierte Programmiersprache, die auf JavaScript aufbaut, unter den Teilnehmer auch beliebt. Ca. 39% aller Entwickler und ca. 44% der professionellen Entwickler verwenden auch TypeScript. Damit ist TypeScript die 4. beliebteste Programmiersprache. [3] Daraus folgt, dass das Ökosystem von JavaScript eine hohe Praxisrelevanz besitzt.

JavaScript wird nicht nur für die Entwicklung im Frontend, sondern auch für die Entwicklung im Backend verwendet, ungefähr 3% der weltweit bekannten Server verwenden eine Laufzeitumgebung, die JavaScript ausführen kann. [5] Um JavaScript auf einem Server ausführen zu können, wird eine Laufzeitumgebung benötigt. Wie Abbildung 1.1 zeigt, ist Node.js die am weitesten verbreitete Laufzeitumgebung. In der gezeigten Umfrage zum Zustand von JavaScript beantworteten ca. 71% von 30.000 befragten Entwickler, dass sie Node.js als Laufzeitumgebung regelmäßig verwenden.

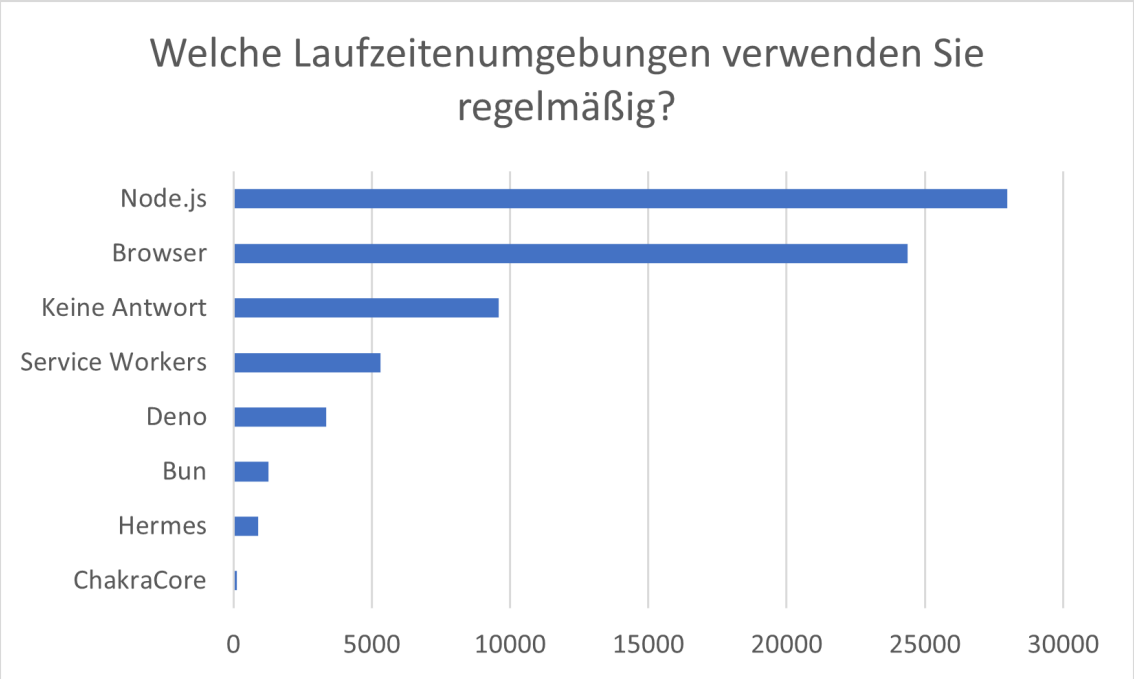

Abbildung 1.1: Nutzungsstatistik von JavaScript-Laufzeitumgebungen [4]

Nur ca. 9% der befragten Entwickler verwenden Deno und ca. 3% Bun als eine Alternative zu Node.js. [4]

1.2 Zielsetzung

Aktuell ist Node.js die Laufzeitumgebung, die am weitesten verbreitet ist. Dennoch gibt erscheinen immer wieder neue Laufzeitumgebungen für JavaScript, die versuchen Node.js zu verdrängen. Eine mögliche Alternative ist Bun, das am 9. September 2023 in der Version 1.0 veröffentlicht worden ist. Die Entwickler von Bun werben mit Features wie erheblicher Performancesteigerung, eleganten Schnittstellen und einer angenehmen Entwicklererfahrung. [1]

Das Hauptziel dieser Arbeit besteht darin, die Version 1.0 der JavaScript Laufzeitumgebung 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 Projekte auf der Basis von Node.js kompatibel mit Bun? Wie schwierig gestaltet sich die Migration?
- Welche Herausforderungen und potenziellen Vorteile ergeben sich bei der Verwendung von Bun 1.0 im Vergleich zu Node.js für Entwickler und Projekte?

1.3 Aufbau der Arbeit

Kapitel schreiben

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 weiterer Grundlagen zu Performanceanalysen.

2.1 Node.js

Node.js ist ein beliebtes Tool für eine große Varianz an Projekten, darunter leichtgewichtige Webservices, dynamische Webanwendungen und Tools für die Kommandozeile. Es handelt sich um eine Open Source, platformunabhängige 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. [6]

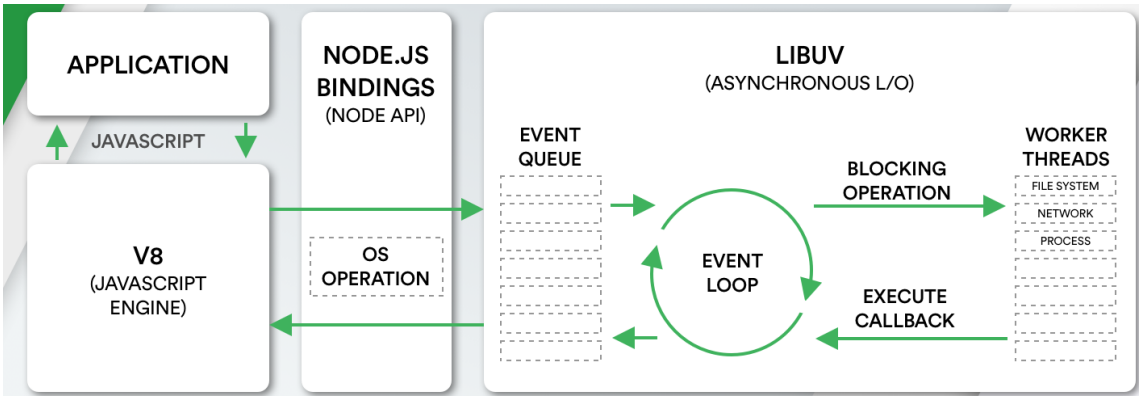

Abbildung 2.1: Node.js Architektur [7]

Abbildung 2.1 zeigt die Architektur von Node.js. Grundsätzlich nutzt Node.js 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 zeitsensitive Aufgabe darstellt. Darüber hinaus ist in der Engine ein Garbage Collector integriert, der nicht mehr verwendete Objekte löscht. [8]
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. [8]

Node.js Bindings

Node.js Bindings, auch bekannt als "Node.js Addons", 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 im Modul "fs"verwendet. Das Modul "fsïst in der Standard-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 beispielsweise das Module "fs"für den Zugriff auf das Dateisystem oder "http", um den Umgang mit dem HTTP-Protokoll zu vereinfachen. [**OpenJSFoundation.**] 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. [8] Allerdings ist in der Standard-API keine Unterstützung für TypeScript enthalten. Hierzu muss der TypeScript Compiler separat installiert werden. [6]

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 ausgelagert. 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. [8]
Während der Laufzeit werden viele Events erzeugt und in einer Message Queue, der Event Queue, nacheinander gespeichert. Node.js nutzt FIFO und beginnt demnach mit der Verarbeitung der ältesten Events und arbeitet sich durch die Queue, bis keine Events mehr vorhanden sind. [9]

Libuv
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, die unter Windows auf andere Weise verfügbar sind. 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. [8]

Darüber hinaus bringt Node.js den Node Package Manager (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 nahezu alle Anwendungsfälle. Ursprünglich wurde NPM entwickelt, um Abhängigkeiten in Projekten zu verwalten, wird aber mittlerweile auch als Werkzeug für JavaScript im Frontend unterstützt. [6] Der NPM ist nicht Teil des Executables von Node.js, wird bei der Installation häufig mitgeliefert [8].

Zusammenfassend zeichnet sich Node.js durch eine eventgesteuerte Architektur und durch ein nicht blockierendes Modell für Ein- und Ausgabeoperationen aus, was 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. [6] 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. [2] Das effiziente nicht blockierende I/O-Modell

von Node.js ermöglicht es, mehrere gleichzeitige Anfragen effizient zu verarbeiten und eignet sich daher gut für anwendungsspezifische Aufgaben, die viele gleichzeitige Verbindungen erfordern.

Quelle einfügen

Skalierbarkeit?

Allerdings existieren auch Nachteile bei der Verwendung von Node.js. Das Single-Threaded-Modell kann bei rechenintensiven oder CPU-lastigen Aufgaben zu Engpässen führen, da es nur einen Hauptthread für die Ausführung von Code gibt [10]. Bei komplexen Anwendungen kann die Verwaltung von Callbacks und Promises zur Bewältigung von asynchronem Code kompliziert werden.

Quelle einfügen

Ein weiterer Nachteil ist, dass Node.js im Vergleich zu einigen anderen Laufzeitumgebungen über eine begrenzte Standardbibliothek verfügt, sodass Entwickler häufig auf externe Module und Pakete zurückgreifen müssen. Darüber hinaus kann die Qualität einiger NPM-Pakete variieren, und schlecht gewartete Module können zu Kompatibilitätsproblemen und Sicherheitsrisiken führen.

Quelle einfügen

Insgesamt ist Node.js eine leistungsfähige und vielseitige Laufzeitumgebung, die sich gut für bestimmte Anwendungsfälle eignet, insbesondere wenn es darum geht, skalierbare und asynchrone Anwendungen zu entwickeln.

Quelle einfügen

2.2 Bun

Bun ist ein Open Source Toolkit für JavaScript. Dieses kombiniert verschiedene wichtige serverseitige Komponenten, um eine leistungsstarkes Paket zur Verfügung zu stellen. Bun ist auf macOS und Linux für die produktive Nutzung freigegeben. Unter Windows steht aktuell nur eine experimentelle Version zur Verfügung, deren Performance noch nicht optimiert worden ist. Als Alternative kann über das Windows Subsystem für Linux die veröffentlichte Linux-Version genutzt werden. [1] Ursprünglich ist Bun als ein persönliches Freizeitprojekt von Jared Sumner gestartet. Mittlerweile hat es sich als wettbewerbsfähige Alternative zu etablierten Technologien in der Webentwicklung etabliert. [11]

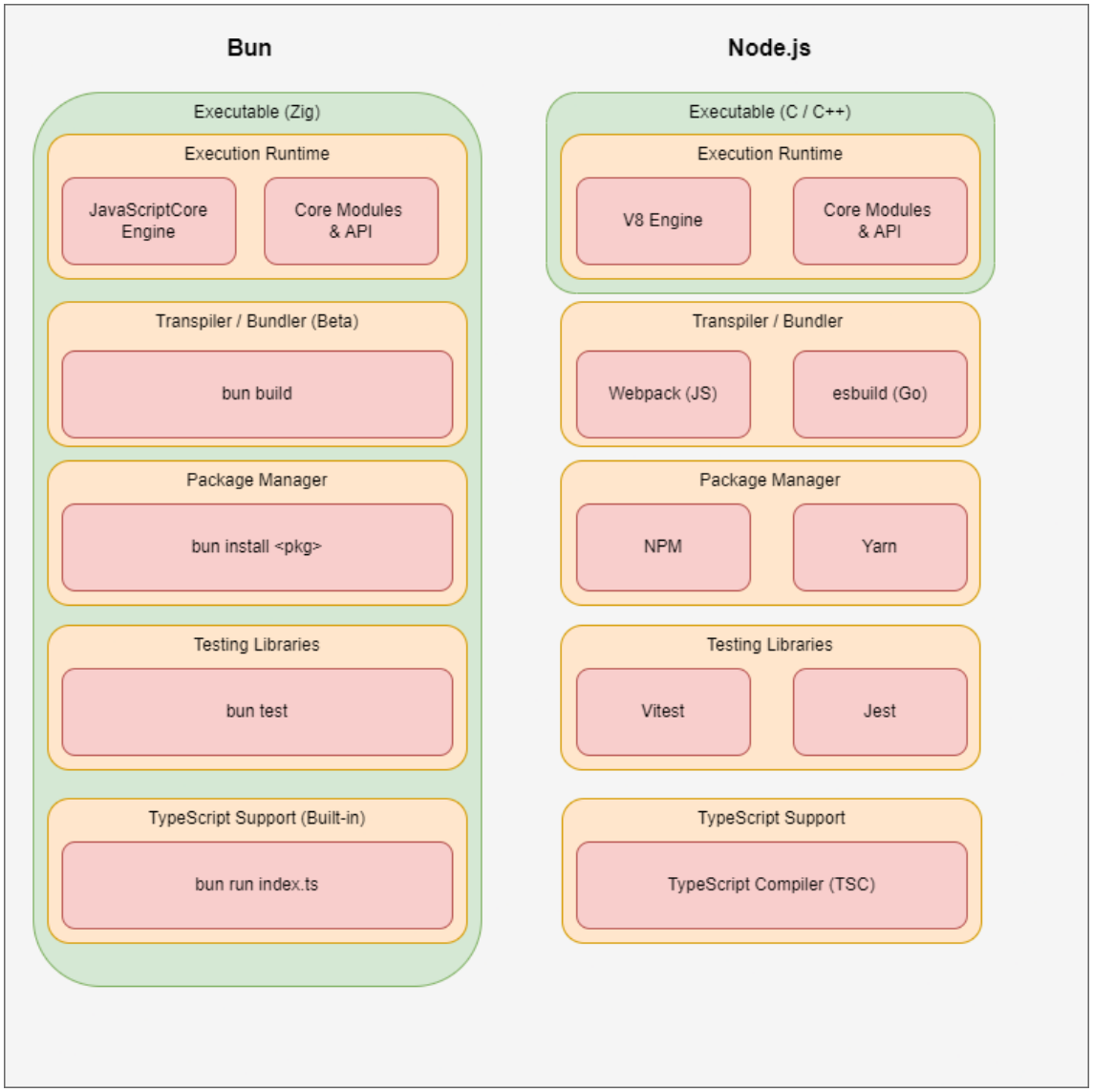

Abbildung 2.2: Vergleich der Ökosysteme von Bun und Node.js

Quellen für Abbildung oder wird das aus dem Text ersichtlich?

Abbildung 2.2 zeigt das Ökosystem von Bun im Vergleich zu Node.js. Im Toolkit von Bun sind folgende Komponenten enthalten: 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. [1] 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. [12] In Node.js ist nur die Laufzeitumgebung enthalten, die anderen Komponenten müssen separat installiert werden. [8] Dies bietet allerdings mehr Flexibilität bei der Auswahl der gewünschten Tools.
Bun ist in Zig geschrieben [12]. Zig ist eine systemnahe Programmiersprache wie C

und C++, die sich vor allem auf Einfachheit und Klarheit für ein besseres Verständnis konzentriert [13]. Die Entwickler haben sich aufgrund sehr guten Performance und des Speichermanagements für Zig entschieden. Anstatt der V8 Engine von Google verwendet Bun die JavaScriptCore Engine [12]. Diese ist die Engine für WebKit, die unter anderem in Apple's Safari-Browser genutzt wird [14]. In Kombination mit Zig sorgt die JavaScriptCore Engine für die bessere Performance von Bun. Dies ist auf die Architektur der JavaScriptCore Engine mit 3 Just-In-Time-Kompilern und Interpreter. Dadurch kann die Engine den Quellcode noch besser optimieren. So verbessert sich die Ausführungszeit des Quellcodes mit Bun. [15] Darüber hinaus ermöglicht Zig mit dessen manuellem Speichermanagement und Klarheit im Kontrollfluss und Allokieren von Speicher weitere Verbesserungen der Effizienz. [12] Zusätzlich führen die beiden Komponenten noch zu drastisch reduzierten Startzeiten [12]. Das ist vor allem im Bereich des Serverless Computing ein enormer Vorteil gegenüber anderen Alternativen. Denn dies hilft die Skalierbarkeit einer Software zu verbessern, indem neue Knoten schneller hinzugezogen werden können. [16]
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, atob. [12]

Allerdings existieren auch viele Module und globale Objekte, für die die Unterstützung teilweise oder komplett fehlt, zum Beispiel assert oder http2 [12]. Daraus folgt, dass die Kompatibilität von bestehenden Projekten auf Basis von Node.js von den verwendeten Objekten und Modulen abhängt.

2.3 Performance als Qualitätsattribut

Um eine qualitativ hochwertige Software zu entwickeln, genügt es nicht, lediglich 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 dargestellt. [17]

Abbildung 2.3: Node.js Architektur [17]

Eigene Abbildung auf der Basis der Leseprobe von ISO selbst

Abbildung 2.3 zeigt die 8 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 Unternehmen ihre Software-Produkte zu verbessern und alle Teilbereiche zu beachten, indem der Standard als Leitfaden vom Identifizieren der Anforderungen bis zur Qualitätskontrolle der Software unterstützt. [18]

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"* [.]

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 3 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 [18]. 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. Sobald die Reaktionszeit die maximale Zeitangabe überschreitet, erfüllt die Software die Latenzanforderung nicht. 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. [19] Bessere Quelle suchen

Ressourcennutzung

Das Maß, in dem die Menge und Art der Ressource, die ein Produkt bei der Ausführung seiner Funktionalitäten beansprucht, den Anforderungen entsprechen, entspricht der Ressourcennutzung [18]. 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. [19] Bessere Quelle suchen

Kapazität

Die Kapazität entspricht dem Maß, in dem die maximalen Grenzen eines Parameters einer Software den Anforderungen entsprechen [18]. Dadurch wird bestimmt, ob das System unter Spitzenlast funktionsfähig bleibt und dadurch gut skaliert. 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. [19] Das umfasst mehrere Bessere Quelle suchen
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. Quelle

Kapitel 3

Performanceanalyse

In diesem Kapitel wird die Performance von Bun detailliert betrachtet 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. Zuerst wird die Vorgehensweise bei den Tests vorgestellt. Anschließend wird der verwendete Versuchsaufbau und die Beispielimplementierungen präsentiert. Darauffolgend werden die Ergebnisse der Tests analysiert.

3.1 Vorgehensweise

Als Metriken werden die durchschnittliche Latenz, die Anzahl an HTTP-Anfragen pro Sekunde, der Anteil an erfolgreichen HTTP-Anfragen, die CPU-Auslastung un der maximal genutzte Arbeitsspeicher während der Ausführungszeit (siehe Kapitel 2.3). Um diese Metriken zu ermitteln, werden verschiedene Szenarien inklusive unterschiedlicher Implementierungen verwendet (siehe Kapitel 3.3). Die unterschiedlichen Implementierungen sind auf variierende APIs der Laufzeitumgebungen zurückzuführen. Diese müssen verwendet werden, damit die Leistung der Laufzeitumgebungen und nicht die Performance des Quellcodes geprüft wird.

Verifizieren, dass die Metriken dort inkludiert sind

Abkürzung einführen, falls in Theorie nicht geschehen

Zuerst wird die grundlegende Performance von HTTP-Server beider Laufzeitumgebungen gemessen (siehe Kapitel 3.3.1). Es greifen 500 gleichzeitige Benutzer auf den Server für 30 Sekunden lang zu und erhalten einen String als Antwort zurück. Dieses Szenario veranschaulicht die grundlegende Netzwerkgeschwindigkeit beider Laufzeitumgebungen. Als zweites Szenario wird ein Datei-Server verwendet, der jedem Aufrufer ein Bild zurückgibt (siehe Kapitel 3.3.2). Diese Aufgabe wird für 50, 250, 500 und 1000 gleichzeitige Nutzer für eine Dauer von 30 Sekunden gemessen.

Die Last wird variiert, um die Server näher an ihre Grenzen zu bringen. Der letzte Testfall berechnet die Fibonacci-Folge für die Zahl 45, damit die Leistung beider Laufzeitumgebungen bei rechenintensiven Aufgaben evaluiert wird (siehe Kapitel 3.3.3).
Um die Performance korrekt zu bestimmen, müssen die richtigen Tools verwendet werden. In dieser Arbeit werden die Folgenden genutzt:

- Bombardier,
- GNU Time.

Bombardier generiert die HTTP-Anfragen an die Server im ersten und zweiten Testszenario. Im Tool kann die Dauer der Lasttests oder auch die Anzahl an zu versendeten Anfragen konfiguriert werden. Zusätzlich kann per Parameter bestimmt werden, wie viele gleichzeitige Benutzer simuliert werden. Nach dem Test gibt das Tool für die Anzahl an Anfragen pro Sekunde und für die Latenz die durchschnittlichen und maximalen Werte sowie die Standardabweichung aus. Zusätzlich kann der Anteil an erfolgreichen Anfragen bestimmt werden. Denn Bombardier gibt auch die Anzahl an Anfragen pro HTTP-Statuscode aus. Das Tool eignet sich aufgrund seinen detailreichen Aufgaben und seiner Performance. Es ist in Go geschrieben und verwendet das Paket “fasthttp“ statt der nativen HTTP-Implementierung von Go und ist dadurch ausreichend performant. Damit die CPU-Auslastung und der maximal verwendete Arbeitsspeicher in den Ergebnissen berücksichtigt werden kann, wird GNU Time verwendet. GNU Time ist in Ubuntu bereits nativ verfügbar und eignet sich dadurch. Auf MacOS wird eine entsprechende Portierung dieses Tools verwendet, um vergleichbare Daten zu erheben.
Um ein möglichst repräsentatives Ergebnis zu erhalten, wird jedes Testszenario für jede Laufzeitumgebung jeweils 5-mal ausgeführt. Aus den gesammelten Daten wird pro Eigenschaft der Durchschnitt gebildet. Dadurch fallen einzelne Ausreißer weniger ins Gewicht.
Die vorgestellte Testkonfiguration ermöglicht die Quantifizierung der Performance-Metriken, um aus diesen Metriken fundierte Aussagen über die Performance beider Laufzeitumgebungen ableiten zu können und die 1. Forschungsfrage (siehe Kapitel 1.2) zu beantworten.

3.2 Versuchsaufbau

Um eine konsistente und kontrollierte Umgebung für die Tests zu schaffen, werden diese auf spezifischer Hardware und Software durchgeführt. Das Ziel besteht darin,

die Testergebnisse so reproduzierbar wie möglich zu gestalten. Des Weiteren wird dadurch eine Vergleichbarkeit zwischen Bun und Node.js gewährleistet, was die Quantifizierung der verwendeten Metriken ermöglicht.

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: Hardware für die Performanceanalyse

Die Tests werden auf verschiedenen Geräten mit unterschiedlichen Betriebssystemen durchgeführt, wie in Tabelle 3.1 dargestellt. Dies dient der Verifikation, ob etwaige Performance-Verbesserungen auf eine spezifische Systemumgebung zurückzuführen sind. Die native Implementierung von Bun für Windows ist experimentell und nicht vollständig für die Performance optimiert (siehe Kapitel 2.2). Die experimentelle Lösung ist nicht für die Öffentlichkeit zugänglich [20]. Daher kann die Funktionsweise von Bun unter Windows nicht getestet werden.
Um die tatsächlichen Tests auszuführen, werden die folgenden Versionen der betrachteten Frameworks verwendet:

- Bun Version 1.0.6 (Neuste Version)
- Node.js Version 18.18.2 (LTS)
- Node.js Version 21.0.0 (Neuste)

Vermwerk, zu welchem Datum es sich um die aktuellsten Versionen handelt

Die neuste Version von Bun wird für die Tests verwendet, da sie im Vergleich zur Version 1.0 bereits Fehlerkorrekturen enthält [1]. Bei der Analyse von Node.js werden zwei Versionen einbezogen. Zum einen die Version mit Long Term Support (LTS), da Node.js diese Version für die meisten Benutzer aufgrund des langfristigen Supports empfiehlt [9]. Zum anderen die neuste Version von Node,js, da in Version 20 beispielsweise die neuste Version des URL-Parsers Ada eingeführt wurde, die signifikante Performance-Verbesserungen mit sich bringt [21]. Zusätzlich enthält Version 21 weitere kleine Verbesserungen hinsichtlich der Performance [22].

3.3 Implementierungen

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

3.3.1 HTTP-Server

Um die grundlegende Performance von Netzwerkanfragen zu bestimmen, werden die zwei einfache Programme verwendet. Abbildung 3.1 zeigt den Quellcode für Bun, Abbildung 3.2 für Node.js.

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

Abbildung 3.1: HTTP-Server Bun

```
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);
```

Abbildung 3.2: HTTP-Server Node.js

Um die Performance der ersten beiden Programme zu testen, werden mit Bombardier 500 gleichzeitige Benutzer für eine Dauer von 30 Sekunden simuliert. Der dafür notwendige Befehl wird in Abbildung 3.3 visualisiert.

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

Abbildung 3.3: Bombardier HTTP-Server

Die Server wurden mit der entsprechenden Laufzeitumgebung gestartet. Um die Auslastung der CPU und des RAMs zu bestimmen, wird GNU Time benutzt. Der dafür erforderliche Befehl wird in Abbildung 3.4 für Ubuntu und in Abbildung 3.5 für MacOS dargestellt. In MacOS wurde “gtime“ genutzt, das eine Portierung von “time“ unter Linux ist.

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

Abbildung 3.4: Bombardier HTTP-Server

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

Abbildung 3.5: Bombardier HTTP-Server

3.3.2 File-Server

Eine häufige Aufgabe von Web-Servern ist es, Bilder für das Frontend zur Verfügung zu stellen . Daher wird dieses Szenario als Nächstes gemessen, um auch die Performance von Zugriffe auf das Dateisystem in den Vergleich einfließen zu lassen.
Abbildung 3.6 zeigt die Implementation für Bun, Abbildung 3.7 für Node.js.

Quelle?

```
1  const basePath = "../data";
2
3  Bun.serve({
4    port: 3000,
5    fetch(request) {
6      const filePath = `${basePath}${new URL(request.url).pathname
     }`;
7
8      try {
9        return new Response(Bun.file(filePath));
10     } catch (error) {
11       return new Response("File not found", {
12         status: 404
13       });
14     }
15   },
16 });
```

Abbildung 3.6: File-Server Bun.js

```
1  import { createReadStream } from "node:fs";
2  import http from "node:http";
3
4  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);
```

Abbildung 3.7: File-Server Node.js

Mit Hilfe von Bombardier rufen beide Server dasselbe Bild aus dem Dateisystem ab. Falls dieses nicht gefunden wird, geben beide eine entsprechende Fehlermeldung zurück. Dieses Testszenario wird jeweils mit 50, 250, 500 und 1000 gleichzeitigen Benutzern für eine Dauer von 30 Sekunden getestet (siehe Kapitel 3.1). Abbildung 3.8 zeigt das resultierende Kommando für 50 gleichzeitige Nutzer. Die Befehle zum Messen 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
```

Abbildung 3.8: Bombardier File-Server

3.3.3 Fibonacci

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 Abbildung 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));
```

Abbildung 3.9: Berechnung der Fibonacci-Folge

Das Programm wird mit beiden Laufzeitumgebungen und GNU Time zur Erhebung der notwendigen Metriken ausgeführt. Abbildung 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
```

Abbildung 3.10: Fibonacci Node.js

3.4 Ergebnisse

Im Folgenden werden die Ergebnisse der Testszenarien vorgestellt. Im Anschluss folgt die Diskussion über mögliche Konsequenzen.

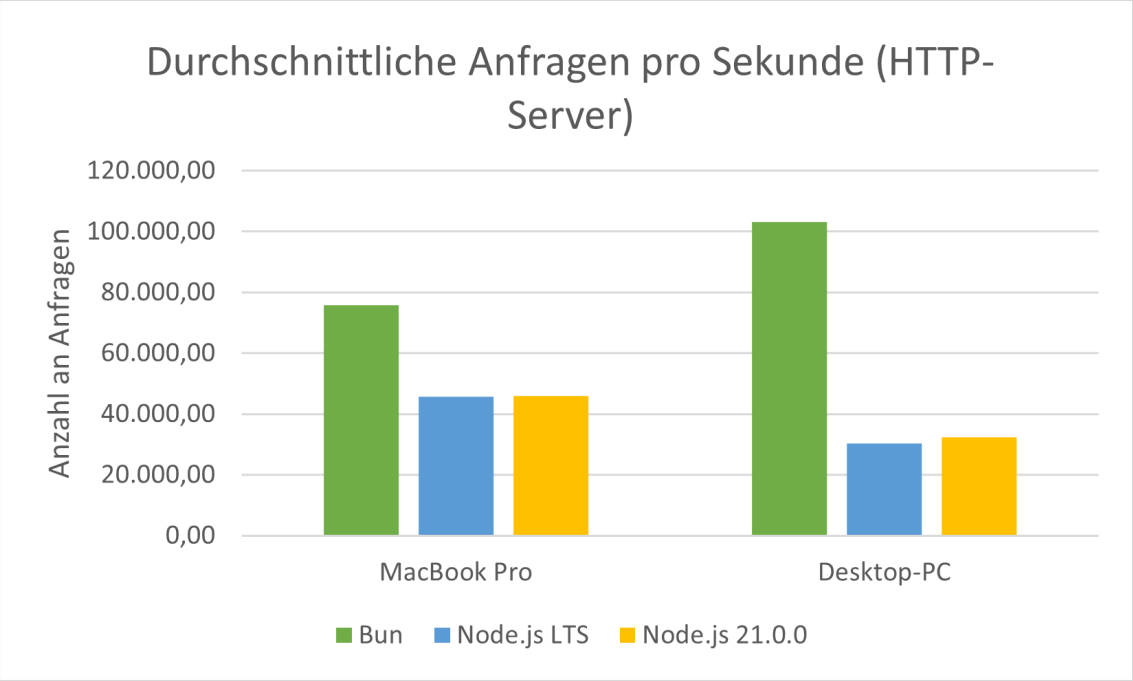

Abbildung 3.1: HTTP-Server - Durchschnittliche Anzahl an Anfragen pro Sekunde

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 bei 500 Benutzern bewältigen konnte. Bun hat auf beiden Endgeräten deutlich besser abgeschnitten als die Node.js, unabhängig von dessen Version. Bun konnte auf dem Desktop-PC pro Sekunde ungefähr 103.000 Anfragen bewältigen, Node.js nur 30.000 (LTS) und 32.000 (Latest). Derselbe Sachverhalt macht sich auch in den Latenzzeiten bemerkbar. Bun hatte eine Latenz von 6,61ms auf dem MacBook Pro und 4,84ms auf dem Desktop-PC . Im Vergleich musste ein Benutzer bei Node.js ungefähr 10ms auf dem MacBook Pro und ca. 16ms auf dem Desktop-PC auf eine Antwort warten.

Diagramm im Anhang hinzufügen

Dieser signifikante Unterschied zeigt, dass Bun Potential bietet auch in realen Szenarien deutlich schneller zu sein als Node.js. Der erste Test zeigt, dass die Verwendung der JavaScriptCore Engine und Zig als Programmiersprache Vorteile zu bieten scheint. In der Produktionsumgebung können so mehrere gleichzeitige Nutzer ohne Performanceverluste bedient werden.

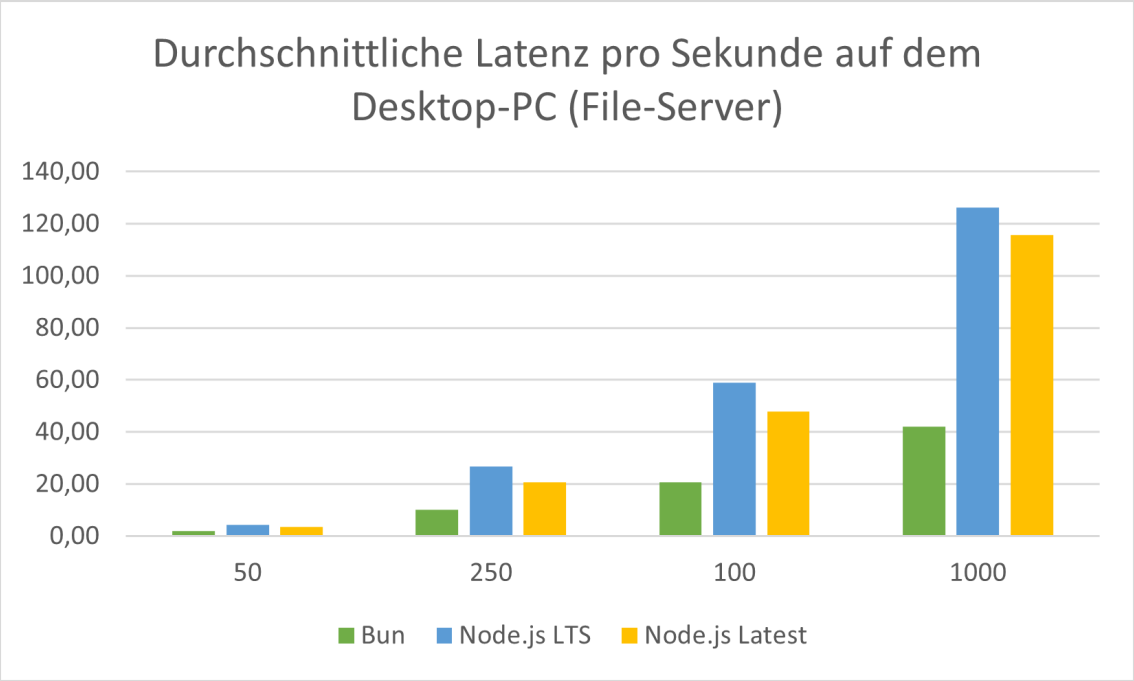

Abbildung 3.2: File-Server - Durchschnittliche Latenz

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 Latenzzeiten beim Zugriff auf ein Bilddatei, die an den Aufrufer zurückgegeben wird. Bun schneidet deutlich besser ab als Node.js. Dies ist über alle 4 Vergleichsszenarien zu beobachten. Bei 50 gleichzeitigen Benutzern hat Bun eine Latenzzeit von 2ms, währned Node.js eine Latenz von 4,2ms (LTS) bzw. 3,5ms (Latest) aufweist. Der Unterschied zwischen Bun und Node.js wächst mit der Anzahl an gleichzeitigen Benutzern an. Bei 250 Nutzern beläuft sich Buns Latenz auf ca. 10ms, der beste Wert von Node.js liegt bei ca. 21ms (Latest). Bei 1000 gleichzeitigen Nutzern benötigt Node.js im Durchschnitt für ca. 116ms (Latest) für die Antwort ein den Benutzer. Bun schafft es dagegen in ca. 42ms. Die Unterschiede betragen teilweise mehr als 50%. Dies führt zu drastischen Unterschieden bei der Ladezeit von Webseiten, wenn Bilder teilweise mehr als 50% schneller geladen werden können. Dadurch, dass Bun die Anfragen selbst deutlich schneller beantwortet, bewältigt Bun gleichzeitig deutlich mehr Anfragen pro Sekunde. Dies ist unabhängig von der Anzahl an gleichzeitigen Benutzern. Auf dem Desktop-PC schaffen es alle Laufzeitumgebungen alle gesendeten Anfragen erfolgreich zu beantworten. Auf dem MacBook Pro beantwortet Bun bei 50, 250 und 500 gleichzeitigen Anwendern alle Anfragen erfolgreich. Bei 1000 Nutzern sinkt der Wert auf 99,96%. Die LTS-Version von

Node.js liefert auch erst bei 1000 gleichzeitigen Benutzern fehlerhafte Antworten zurück. Der Anteil erfolgreicher Anfragen beläuft sich auf 99,92%. Erst bei der neusten Version von Node.js treten Unterschiede zu Bun und Node.js LTS auf. Diese neuste Version beantwortet ab 250 gleichzeitigen Nutzern nicht mehr alle Anfragen erfolgreich. Bei 1000 Nutzern sinkt der Wert auf 99,54%. Die absolute Anzahl an Anfragen, die mit einem Fehler beantwortet wurden, ist sehr niedrig. Dennoch ist es sehr auffällig, dass die neuste Version von Node.js hier deutlich größere Defizite aufweist. Möglicherweise ist hier auch noch ein Fehler im neusten Release enthalten.

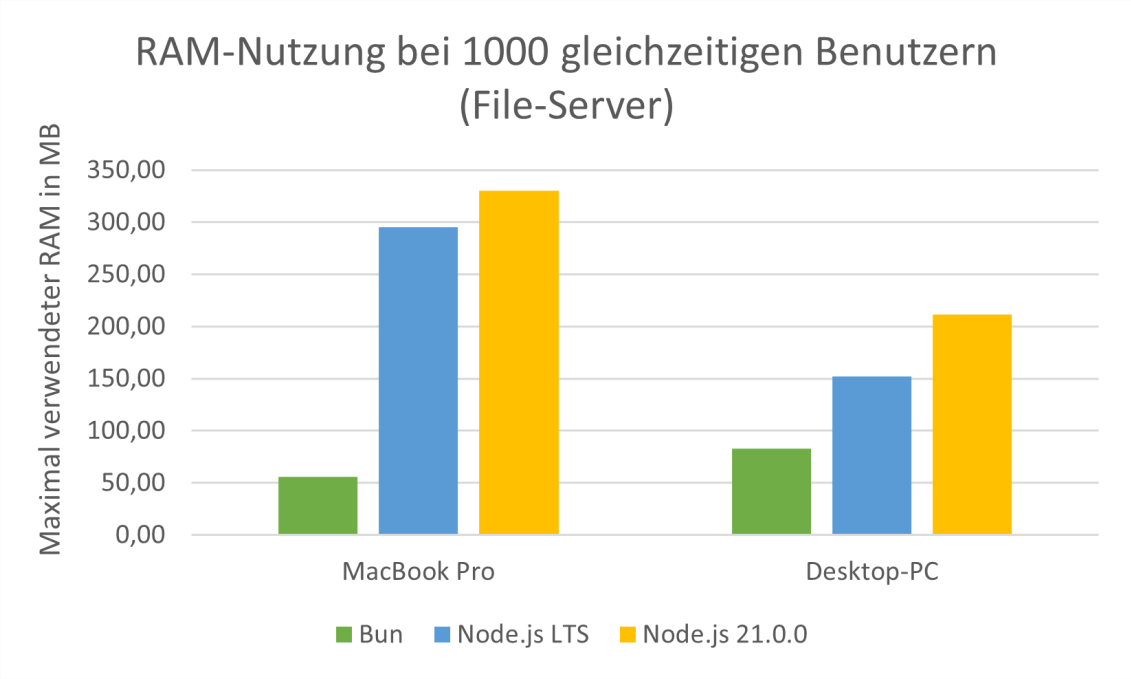

Abbildung 3.3: File-Server - Maximal verwendeter Arbeitsspeicher

Bei der höchsten Last werden die RAM- und CPU-Nutzung der Laufzeitumgebungen verglichen. Denn bei der höchsten Last sind die Differenzen am besten zu erkennen sein. Abbildung 3.3 zeigt den maximal verwendeten Arbeitsspeicher während des Testzeitraums von 30 Sekunden. Hier manifestiert sich der Eindruck der zuvor betrachteten Eigenschaften. Bun performt deutlich besser als Node.js, unabhängig von der verwendeten Version. Auf dem MacBook Pro benötigt Bun ca. 56 MB Arbeitsspeicher, Node.js dagegen ca. 296 MB (LTS) bzw. ca. 330 MB (Latest). Die Differenz in der RAM-Nutzung ist auf dem Desktop-PC nicht so groß wie auf dem MacBook Pro. Bun verwendete ca. 84 MB. Node.js benötigt ca. 152 MB (LTS) bzw. ca. 212 MB (Latest). Die maximale Nutzung des Arbeitsspeicher ist ein kritischer Indikator für die Effizienz bei der Speichernutzung. D. h. Bun ist hier siginfikant effizienter als Node.js. Die hohe Effizienz könnte ein Grund für die bessere Perfor-

mance sein. Der Sachverhalt zeigt, dass Zig und die JavaScriptCore Engine in Bun helfen die Speichereffizienz zu steigern. Die Beobachtungen deuten auf Vorteile in der Skalierbarkeit und Kosteneffizienz hin.

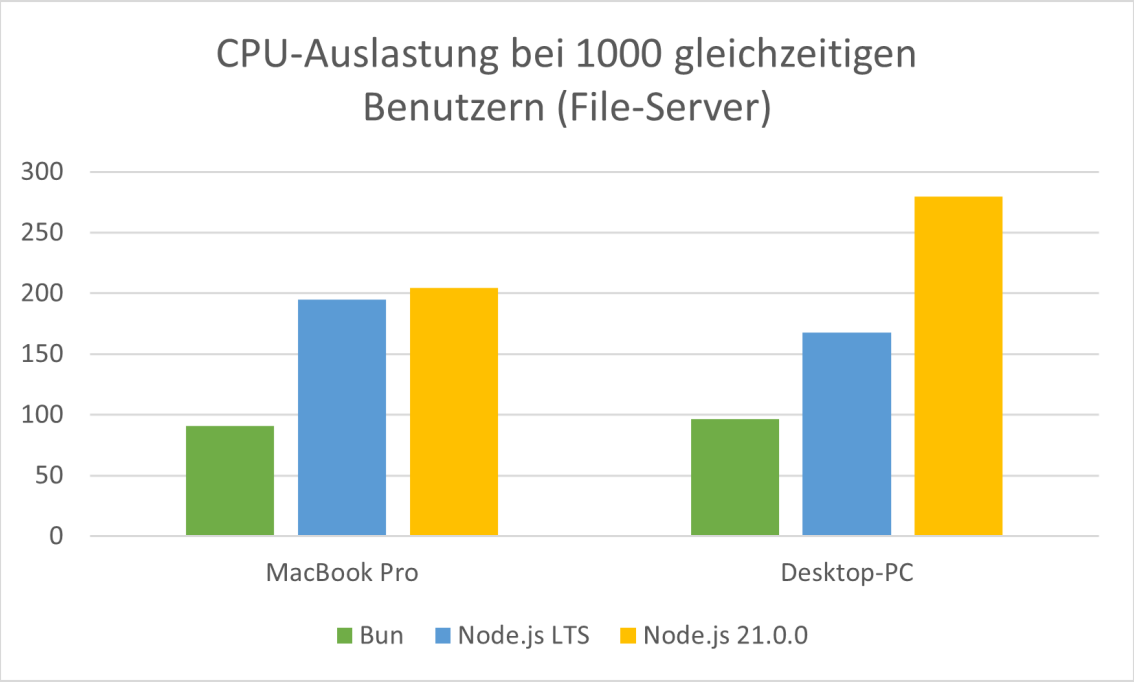

Abbildung 3.4: File-Server - CPU-Auslastung

Abbildung 3.4 zeigt die CPU-Auslastung bei 1000 gleichzeitigen Benutzern in Prozent. Node.js ist auch im Bereich der CPU-Nutzung weniger effizienter als Bun. Bun beansprucht ungefähr 91% der CPU auf dem MacBook Pro und 96% der CPU auf dem Desktop-PC. Node.js LTS benötigt ca. 195% auf dem MacBoook Pro und 167% auf dem Desktop-PC. Die aktuellste Version von Node.js schneidet nochmals schlechter ab. Auf dem MacBook Pro ist der Unterschied zu Node.js LTS mit einer höheren CPU-Auslastung von 10% relativ gering. Auf dem Desktop-PC nutzt die neuste Version von Node.js 280% der CPU im Vergleich zu 167% bei der LTS-Version. Dies zeigt wie die Nutzung des Arbeitsspeichers, dass Bun mit den vorhandenen Ressourcen effizienter umgeht. Daraus folgt, dass Bun bei begrenzten Ressourcen intensivere Aufgaben erledigen kann.

Mehr als 100% CPU Nutzung erklären

Der dritte Testfall vergleicht die Leistung von Bun und Node.js bei der Ausübung rechenintensiver Aufgaben. Abbildung 3.5 veranschaulicht die benötigte Zeit für die Berechnung. Beim Berechnen der Fibonacci-Folge für die Zahl 45 benötigt Bun im Durchschnitt nur 3,24 Sekunden auf dem MacBook Pro und 4,47 Sekunden auf dem Desktop-PC. Die Versionen von Node.js unterscheiden sich kaum voneinander.

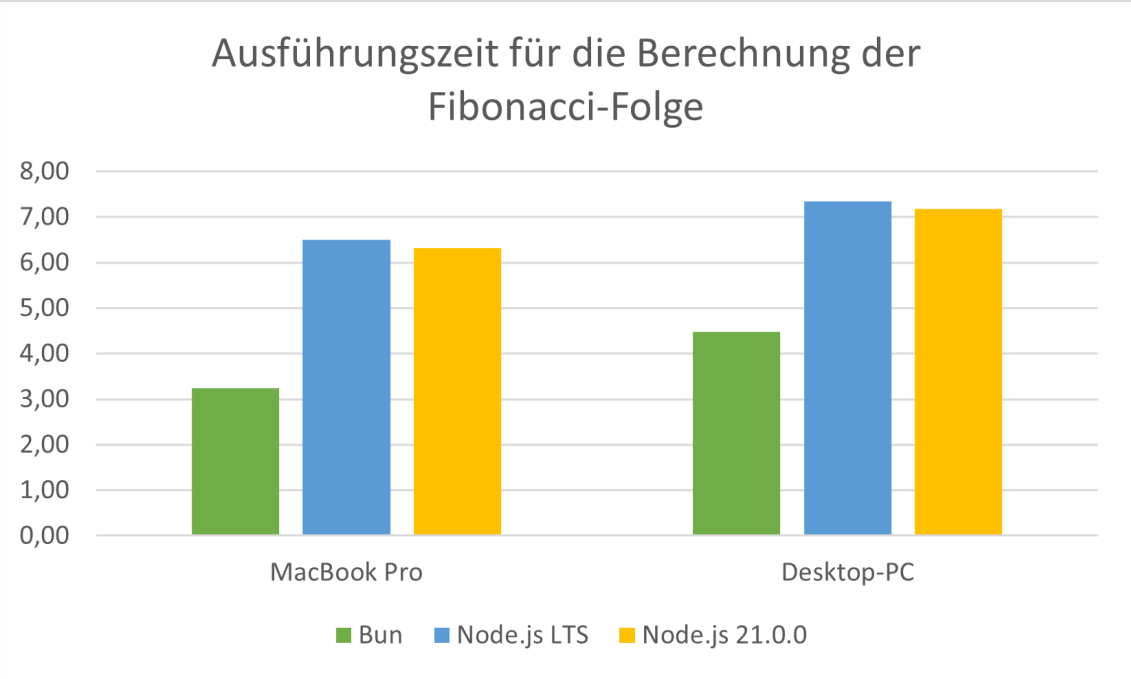

Abbildung 3.5: Ausführungszeit für die Berechnung der Fibonacci-Folge

Die neuste Version ist auf beiden Endgeräten ca. 0,2 Sekunden schneller als die LTS-Version. Dennoch beläuft sich die schnellste Berechnung mit Node.js auf 6,32 Sekunden auf dem MacBook Pro und auf 7,18 Sekunden auf dem Desktop-PC. 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%. 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 deutlich größer. Bun benötigt maixmal ca. 26 MB RAM, Node.js im Vergleich dazu 42 MB (LTS) und 37 MB (Latest).
Die Betrachtung der CPU-Auslastung, der RAM-Nutzung und der Ausführungszeiten im Kontext der Fibonacci-Folge bestätigt das Verhalten aus den zwei vorangegangenen Testszenarien. Bun besitzt eine deutlich höhere Leistung auf den genannten Endgeräten unabhängig von der betrachteten Metrik.

3.5 Fazit

Dieses Kapitel widmet sich auf die Beantwortung der ersten 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 deutlich reduzierten Latenz, in einer geringeren Inanspruchnahme von Arbeitsspeicher und CPU-Ressourcen während der Verarbeitung von Netzwerkanfragen. Diese Resultate bestätigen sich bei der Ausführung rechenintensiver 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.
Diese Ergebnisse unterstreichen, dass Bun bei der Entwicklung gute Entscheidungen bei der Auswahl seiner Komponenten getroffen hat. Die JavaScriptCore Engine mit mehreren JIT-Kompilern (siehe Kapitel 2.2) ermöglicht eine effizientere Optimierung es den Maschinencode. Die Auswahl von Zig als systemnahe Programmiersprache und das intensive Testen während der Entwicklung haben die Effizienz von Bun im Umgang mit den verfügbaren Ressourcen gesteigert. Allerdings ist die Performance nicht der alleinige Faktor bei der Wahl einer Laufzeitumgebung. Stabilität, Sicherheit, Zuverlässigkeit und auch Kompatibilität spielen eine ebenso wichtige Rolle.

Das Benchmark basiert auf den Vergleich ausgewählter Eigenschaften auf einem spezifischen Hardware-Setup. Es ist zu beachten, dass diese Ergebnisse unter Verwendung von Servern aus der Produktionsumgebung potenziell abweichen können. Des Weiteren sollten weitere Eigenschaften analysiert werden, um die Überlegenheit von Bun zu verifizieren. Selbst bei einer Reproduktion des Benchmarks auf identischer Hardware kann es aufgrund unterschiedlicher Systemkonfigurationen und laufender Hintergrundprozesse zu Abweichungen kommen.
Zudem beschränken sich die bisherigen Analysen 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 komplexeren 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 signifikanten Unterschiede im Anteil erfolgreicher Anfragen zu ermitteln. Zudem sollte untersucht werden, warum die RAM-Nutzung bei der Berechnung der Fibonacci-Folge unter Ubuntu kaum variiert. Die Untersuchung sollte auf andere Hardware-Setups ausgeweitet werden, um die Generalisierbarkeit der Ergebnisse sicherzustellen. Schließlich könnten umfangreichere Anwendungsbeispiele und realistischere Szenarien in die Analyse einbezogen werden, um die tatsächliche

Performance in komplexen Umgebungen besser abzubilden.

Kapitel 4

Kompatibilität von Projekten

Dieses Kapitel betrachtet die Kompatibilität zwischen Node.js und Bun. Es ist von hoher Bedeutung, dass Bun möglichst viele Frameworks und Pakete aus dem Ökosystem von Node.js unterstüzt. Nur wenn der Migrationsaufwand für den Wechsel der Laufzeitumgebung möglichst gering und einfach ist, kann Bun Node.js als Platzhirsch 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 der Umfrage "State of JavaScript 2022" handelt es sich dabei um Express und NestJS [4].

4.1 Express

TODO

4.2 NestJS

TODO

4.3 Fazit

TODO

Kapitel 5

Schlussbetrachtung

5.1 Fazit

TODO

5.2 Ausblick

TODO

Literatur

[1] JARED SUMNER. *Bun v1.0.6*, 2023. URL: \url{https://bun.sh/blog/bun-v1.0.6}.

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

[3] STACK OVERFLOW. *Stack Overflow Developer Survey 2023*, 2023. URL: \url{https://survey.stackoverflow.co/2023/#overview}.

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

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

[6] OPENJS FOUNDATION, Hrsg. *An introduction to the NPM package manager*, 2022. URL: \url{https://nodejs.dev/en/learn/an-introduction-to-the-npm-package-manager/}.

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

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

[9] OPENJS FOUNDATION. *Node.js*, o. J. URL: \url{https://nodejs.org/en}.

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

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

[12] BUN, Hrsg. *Offizielle Dokumentation*, URL: \url{https://bun.sh/docs}.

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

[14] APPLE, Hrsg. *WebKit*, o. J. URL: \url{https://webkit.org/}.

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

[16] 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: \url{10.1145/3423211.3425682}.

[17] *ISO/IEC 25010*, 2022. URL: \url{https://iso25000.com/index.php/en/iso-25000-standards/iso-25010}.

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

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

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

[21] OPENJS FOUNDATION. *Node.js 20 ChangeLog*, 2023.

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

Anhang A

Anhang Kapitel 1