Hochschule Bremerhaven

Fachbereich II
Management und Informationssysteme
Informatik B.Sc.

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science

Konzipierung und Implementierung einer plattformunabhängigen Open-Source-Entwicklungsumgebung für die Lehre am Beispiel einer Infrastruktur für verteilte Datenbanksysteme

---

Vorgelegt von: Lennart Steffen MatNr. 00000
Vorgelegt am: 29. Oktober 2025
Erstgutachter: Prof. Dr.-Ing. Oliver Radfelder
Zweitgutachterin: Prof. Dr. Nadija Petram

Inhaltsverzeichnis

1 Einleitung

Mit der zunehmenden Digitalisierung und der weltweiten Vernetzung durch das Internet sowie der steigenden Nutzung von Social-Media-Plattformen wächst das Volumen an täglich produzierten Daten exponentiell an [1]. So haben die über zwei Milliarden täglich aktiven Nutzer:innen auf Facebook [2] im Jahr 2023 mehrere Petabyte an Daten am Tag produziert [1].

Diese großen und oft wenig strukturierten Datenmengen werden auch *Big Data* genannt. Das zeitnahe Analysieren und Verarbeiten von *Big Data* ist mit einem einzelnen Rechner nicht realisierbar und erfordert die Verwendung von verteilten Systemen mit vielen Maschinen in einem Rechencluster [3, S. 5]. Durch neue Konzepte und Werkzeuge können große Datenmengen im gesamten Cluster verteilt werden und zur Verarbeitung die CPU-Ressourcen aller beteiligten Maschinen genutzt werden. Facebook nutzt dieses Verfahren mindestens seit 2010 [4].

Um in der Hochschullehre solche Systeme kennenzulernen, ohne ein großes Rechencluster und den Zugriff auf dieses zu verwalten, kann durch Virtualisierung und Containerisierung ein Cluster von verteilten Datenbanksystemen lokal auf einem Gerät nachgebaut werden.
Mit dieser Methode lässt sich eine komplette Entwicklungsumgebung für Studierende zur Verfügung stellen, die diese auf eigenen Geräten installieren können.

Zu Beginn dieser Arbeit wird für diesen Anwendungszweck in dem Modul Datenbanken II an der Hochschule Bremerhaven die *Big Data Lite Virtual Machine* von *Oracle* [5] verwendet. Dabei gibt es einige Einschränkungen und Schwierigkeiten diese Umgebung in der Lehre zu Nutzen.
Zum einen verbietet die Lizenz [6] dieser Softwaresammlung das Bereitstellen auf Hochschulservern. So müssen Studierende sich alle mit einem *Oracle*-Account anmelden und die VM direkt von *Oracle* herunterladen. Da die Lizenz die Entwicklung für Nicht-*Oracle*-Produktionssysteme verbietet, kann die kennengelernte Software später nicht frei für andere Projekte verwendet werden.
Die Komplexität der Software, von der nur ein kleiner Teil auch wirklich für die Veranstaltung genutzt wird, erschwert es, die verwendeten Technologien zu verstehen. Auch besteht die *Big Data Lite Virtual Machine* teilweise aus proprietärer Software, obwohl die Kerntechnologien, die in der Lehre verwendet werden sollen, Open Source sind.
Das steht in starkem Gegensatz zu der Philosophie von Transparenz, Open-Source-Technologien und digitaler Souveränität, die sonst an der Hochschule Bremerhaven gelebt wird.

Ziel dieser Bachelorarbeit ist es, eine alternative, containerisierte, quelloffene und plattformunabhängige Entwicklungsumgebung für verteilte Datenbanksysteme zu konzipieren und zu implementieren.
Die konkrete Implementierung der Umgebung beinhaltet die Technologien *Hadoop*, *Hive*, *HBase* und *Kafka* von *Apache* und soll in Verbindung mit der *Bash* und mindestens den Editoren *Vim* oder *VSCodium* genutzt werden können.

Das erarbeitete Konzept könnte aber auch für andere Technologien oder außerhalb der Lehre verwendet werden.

Dabei soll primär die folgende Frage beantwortet werden:
Wie können Virtualisierungs- und Containerisierungstechniken genutzt werden, um eine plattformunabhängige, quelloffene Entwicklungsumgebung zu schaffen?
Hierzu wird analysiert, inwiefern eine virtuelle Maschine als Docker-Host dienen kann, um auf heterogener Hardware eine einheitliche Basis für containerisierte Software bereitzustellen.
Darüber hinaus wird untersucht, welche Kriterien eine gute Entwicklungsumgebung für die Lehre erfüllen muss. Basierend auf Literatur zu Open-Source-Software, Unix-Philosophie und didaktischen Anforderungen werden nicht-funktionale Eigenschaften wie Transparenz, Wartbarkeit, Nachhaltigkeit und Zuverlässigkeit definiert und auf eine konkrete Entwicklungsumgebung angewendet.

Im Verlauf dieser Arbeit werden zunächst die technischen Grundlagen von Virtualisierung und Containerisierung (Kapitel 1.1 und 1.2) erläutert.
In Kapitel 2 werden die grundlegenden Konzepte der Entwicklungsumgebung, wie eine orthogonale Werkzeugsammlung und ein *Remote-Workspace* beschrieben. Zusammen mit den erarbeiteten Grundlagen aus Kapitel 1 wird eine Architektur von virtueller Maschine als Host für Docker Container festgelegt. Anschließend werden die nicht-funktionalen Anforderungen an Software in der Lehre aufgestellt und damit das Konzept der Entwicklungsumgebung ergänzt.
Kapitel 3 behandelt die Umsetzung der konkreten Entwicklungsumgebung für verteilte Datenbanksysteme. Dafür werden aus den funktionalen Anforderungen die benötigten Technologien identifiziert und beschrieben.
Im 4. Kapitel wird die konkrete Umsetzung auf die Einhaltung der definierten Konzepte und Anforderungen überprüft. Dafür wird der bisherige Wartungsprozess beschrieben und die zu erwartenden Aktualisierungszyklen identifiziert sowie beispielhaft eine neue Technologie in die Entwicklungsumgebung aufgenommen. Zusätzlich werden Entscheidungen bei dem Entwurf des Build Prozesses der Docker Images genauer beleuchtet.
Abschließend werden die Erkenntnisse zusammengefasst und ein Ausblick auf mögliche Erweiterungen gegeben.

1.1 Virtuelle Maschinen in der Lehre

Virtuelle Maschinen (VM) werden mindestens seit den frühen 80er Jahren in der Informatiklehre verwendet. Anwendungszwecke waren zum Beispiel Kompatibilität von Programmen mit sich rasch entwickelnder Hardware zu gewährleisten, aber auch um Praktikumssystem für die Informatiklehre bereitzustellen [7, K. 3].
Damit auf einem Host-System ein oder mehrere Gast-Systeme laufen können, wird die Hardware virtuell bereitgestellt [8, S. 19]. Bei vollständiger Virtualisierung mit modernen Lösungen (wie QEMU/KVM) ist das Gastbetriebssystem gegenüber der zugrundeliegenden Plattform

agnostisch und muss nicht an die zugrundeliegende Virtualisierungssoftware speziell angepasst werden [8, S. 27].

Virtualisierung erzeugt virtuelle Hardware, auf der ein beliebiges Gast-Betriebssystem installiert werden kann [8, S. 19]. Das Gast-Betriebssystem ist der Virtualisierung gegenüber agnostisch und muss nicht an die Virtualisierung oder die spezielle Virtualisierungssoftware angepasst werden [8, S. 27].

Bei der Software Virtualisierung wird die komplette Hardware mithilfe von Software simuliert. CPU Anweisungen des Gastsystems, die sonst in wenigen CPU Zyklen auf der Hardware ausgeführt werden, müssen in mehrere Anweisungen auf dem Hostsystem übersetzt werden [8, S. 8]. Diese Emulation von Hardware ist nicht besonders performant und kann in der Leistung und Effizienz nicht mit nativen Systemen mithalten. Für eine performatere Software Lösungen kann, als Alternative zur vollständigen Virtualisierung, Paravirtualisierung eingesetzt werden, bei der sich das Gast-System der Virtualisierung bewusst ist und direkt mit dem Host-System kommuninizieren kann [8, S. 27].
Hardware-gestützter Virtualisierung ermöglicht es virtuellen Maschinen direkt auf der Host Hardware laufen. Die CPU Anweisungen müssen hier nicht übersetzt werden, sondern werden an die reale CPU durchgereicht.
So kann zwar keine fremde Hardware emuliert werden aber jedes Betriebssystem, das mit der zugrunde ligender Hardware kompatibel ist, kann installiert werden.
Das Einteilen der CPU Ressourcen und das Verwalten von virtueller Hardware verursacht zwar einen Mehraufwand aber Hardware-gestützte virtuelle Maschinen können annähernd mit der Performanz von nativen Systemen mithalten [9, S. 55].

Mit Hardware-gestützter Virtualisierung ist es, trotz des Mehraufwand, oft effizienter mehrere virtuelle Maschinen auf einem Host laufen zu lassen, um diesen besser auszulasten. Diese Technik wird heute bei einem Großteil von IT-Unternehmen verwendet um virtuelle Server flexibel und effizient auf physische Hardware zu verteilen. [8, S. 6]

Eine virtuelle Maschine mit einem bestimmten Gast-Betriebssystem auf unterschiedlicher Hardware und Host-Betriebssystemen zu installieren ist für die Lehre besonders relevant.
Studierende kommen mit unterschiedlichen Geräten und Betriebssystemen an die Hochschule. Das Voraussetzen bestimmter Software, die nicht auf allen Plattformen verfügbar ist, könnte dazu führen, dass Studierende sich neue Geräte anschaffen müssen.
Eine virtuelle Maschine mit einem Linux Betriebssystem kann als gemeinsame Basis für die genutzte Software dienen, da Virtualisierung auf den verbreiteten Geräten und Betriebssystemen unterstützt wird.

Virtuelle Maschinen sind in der Lehre bereits eine weit verbreitete Technik um eine kuratierte Sammlung von Werkzeugen für die Softwareentwicklung zu konzipieren, zu verwalten und zu verteilen [10].
Für die Lehre von vernetzten Systemen oder *Big Data* Anwendungen in einem Rechencluster muss aber nicht nur eine Maschine simuliert werden, sondern ein komplexes Netzwerk von

mehreren Maschinen.
Dieses Netzwerk könnte auch mit mehreren virtuellen Maschinen aufgebaut werden, allerdings ist die Beschreibung der Topologie von mehreren virtuellen Maschinen größtenteils eng an die Virtualisierungssoftware gebunden [11]. Dadurch wird es schwierig bis unmöglich die Umgebung auf unterschiedlichen Betriebssystemen zu verwenden.
Außerdem wäre der Speicherbedarf für mehrere virtuelle Maschinen sehr hoch, da nicht nur die installierten Programme und relevante Daten gespeichert werden, sondern für jede VM das komplette Betriebssystem mit allen Bibliotheken. Das ist besonders bei Laptops von Studierenden ein Problem, da auf diesen, Programme und Daten für viele Module gespeichert werden müssen und sie oft mit relativ wenig Speicher ausgestattet sind.
Für die Optimierung des Speicherbedarfs von virtuellen Maschinen können Techniken wie *copy-on-write* und *Linked Clones* angewendet werden. Dabei wird eine Basis Maschine geklont und für jede neue VM werden nur die Änderungen von dieser Basis gespeichert [8, S. 263-265]. Diese Methoden sind aber, genau wie die Vernetzung der Maschinen, an die jeweilige Virtualisierungssoftware gekoppelt und dadurch auf unterschiedlichen Betriebssystemen schwierig umzusetzen.

Virtuelle Maschinen können also in der Lehre dazu eingesetzt werden identische Umgebungen auf heterogener Hardware und unterschiedlichen Betriebssystemen aufzusetzen. Darüber hinaus sind sie aber keine ausreichende Lösung um ein komplexes Netzwerk von Rechnern, lokal auf Laptops von Studierenden zu simulieren.

1.2 Containerisierung

Containerisierung ist eine weitere Methode um Prozesse oder ganze Softwaresysteme voneinander zu isolieren. Während bei der Hardware Virtualisierung die Hardware des Hostsystems genutzt wird und ein neues Betriebssystem darauf installiert wird, wird bei der Containerisierung der Kernel des Hostsystem genutzt und nur ein neues Dateisystem darauf aufgesetzt. Dadurch kann mit Containern kein fremdes Betriebssystem gestartet werden und Linux Container können nur auf einem Linux Host laufen.
Durch die Nutzung des Host Kernels sind Container sehr leichtgewichtig, brauchen kaum zusätzliche Ressourcen und können schnell gestartet und heruntergefahren werden. Containerisierung wird auch *Containervirtualisierung* oder *leichtgewichtige Virtualisierung* genannt.

Mit *jails* für FreeBSD und *VServer* für Linux kamen Anfang der 2000er Jahre die ersten Technologien für die Containerisierung in die Unix Welt. *Cgroups*, eine Funktion die Prozesse zu Gruppen zusammenfasst und die Basis für die heutigen Linux Container bildet, wurde 2007 in den Linux Kernel aufgenommen. Mit der Docker Engine wurde 2013 eine Sammlung von Userland Werkzeugen zur einfachen Verwaltung und Benutzung von Linux-Containern veröffentlicht. Zwei Jahre später war die Docker Engine so weit verbreitet, dass die 2015 gegründete Open Container Initiative (OCI) sich die Docker Schnittstellen als Basis für den

Open-Container-Standard nahm. Die OCI Runtime Spezifikation wurde 2015 veröffentlicht und seit Docker Version 1.11, veröffentlicht im April 2016, basiert die Docker Engine darauf. [12]

Die Art und Weise wie Informatiker:innen Software entwickeln, hat sich durch die Containerisierung maßgeblich verändert. Zum einen kann Software in einem Container getestet werden, der die Produktivumgebung nachbildet oder sogar damit identisch ist. Dadurch kann ein klassisches Problem in der Softwareentwicklung vermieden werden: *It works on my machine*.
Alle beteiligten Entwickler:innen können identische Container zur Entwicklung und zum Testen nutzen. Durch das schnelle Starten von Containern sind dabei kurze Entwicklungszyklen möglich.

Ein Container wird aus einem Image erzeugt, in dem das Dateisystem, mit dem der Container gestartet wird, enthalten ist. Das Image kann über eine öffentliche oder private Datenbanken, eine *Docker Registry*, verteilt oder als *Tar*-Datei exportiert werden.
Da Container nur die Veränderungen von diesem Image und Metadaten über Zustand des Containers speichern müssen, ist auch der Festplattenspeicherbedarf viel niedriger als von einer VM, die das komplette Betriebssystem speichern muss.

Für gewöhnlich wird in einem Container nur ein einziger Prozess ausgeführt. Zum Beispiel kann für eine Web-Anwendung ein Webserver, zusammen mit allen benötigten Dateien, in einem Container installiert werden. Im Produktionsbetrieb der Web-Anwendung wird innerhalb des Containers einzig der Prozess des Webservers ausgeführt. Zusätzliche Software, wie eine Datenbank, wird meist in einem eigenen Container realisiert.
Die Kommunikation mit anderen Containern erfolgt über ein von der *Docker Engine* eingerichtetes Netzwerk.
Dieses Verhältnis von einem Prozess zu einem Container ist aber keineswegs notwendig. In Kapitel 2.2 wird beschrieben wie ein Container als Workspace genutzt werden kann, in dem diverse Prozesse für den Entwicklungsvorgang laufen.
Die Container für den Workspace und die Test- und Produktivumgebung können dabei alle auf demselben Image basieren.

Da Docker Container keine native Virtualisierung bieten und nur auf 64-Bit-Linux Systemen laufen, [13, S. 11] ist Docker alleine keine Möglichkeit um Software für heterogene Hardware von Studierenden bereitzustellen.
Hier bietet sich die Kombination aus virtueller Maschine und Containerisierung als Lösung an: Auf den Endgeräten wird, mit der jeweiligen Virtualisierungssoftware, die für das Host-Betriebssystem verfügbar ist, eine virtuelle Maschine mit Ubuntu Linux erstellt.
Diese VM ist die gemeinsame Basis auf der eine Infrastruktur mit Docker Containern aufgesetzt werden kann.
Der einzige relevante Unterschied zwischen den Systemen ist die CPU-Architektur, die mit der Hardware-gestützten Virtualisierung nicht emuliert werden kann. Die verschiedenen CPU Architekturen müssen beim Erstellen der Docker Images berücksichtigt werden. Dieser Prozess wird in Kapitel 4.3 beschrieben.

2 Entwicklungsumgebung für die Lehre

Eine Entwicklungsumgebung besteht aus Betriebssystem, Software zur Erstellung, Übersetzung und Ausführung des Quellcodes sowie allen weiteren Werkzeugen, Maschinen oder Software Systemen die zur Entwicklung und zum Testen von Software erforderlich sind. Darunter fallen z. B. Texteditor, Versionsverwaltung, Compiler, Debugger, Datenbanken, Server für Integrations- und Systemtests und Werkzeuge um mit der Anwendung zu kommunizieren.

2.1 Integrierte oder orthogonale Entwicklungsumgebung

Eine integrierte Entwicklungsumgebung (IDE) wird häufig als Synonym für den Begriff Entwicklungsumgebung verstanden.
IDEs vereinen eine Vielzahl von Werkzeugen in einer einzigen Anwendung und legen dabei oft eine spezifische Arbeitsweise sowie eine bestimmte Projektstruktur zugrunde. Durch diese Standardisierung lassen sich einzelne Entwicklungsschritte vereinfachen oder sogar vollständig automatisieren.

Während die Automatisierung und Abstraktion interner Prozesse die kognitive Belastung von Entwickler:innen reduzieren und somit die Produktivität steigern kann, erschwert dies Studierenden den Zugang zu einem umfassenden Verständnis des Entwicklungsprozesses.
Zudem zeigt sich, dass Studierende oft mehr Zeit in das Erlernen der IDE selbst investieren müssen, wodurch weniger Zeit für das eigentliche Erlernen des Programmierens bleibt [14, S. 996].
Ein weiteres Problem stellt der Übergang von kleinen Projekten in einer IDE hin zu komplexen Infrastrukturprojekten dar. Der Wechsel von einer IDE, in der eine bereitgestellte *Build-and-Run*-Funktion genutzt wird, zu einem an das Projekt angepasstem Build- und Deployment-Prozess ist für viele Studierende eine große Hürde.
Die automatische Syntaxüberprüfung in einer IDE kann beim Schreiben von Code zwar dabei helfen Fehler schneller zu finden, dennoch kann der Debug-Prozess durch die Verwendung verlangsamt werden. So zeigen Beobachtungen [14, S. 996], dass sich Studierende häufig nur auf die von der IDE angezeigten Fehler konzentrieren und Logikfehler leichter übersehen.

An der Hochschule Bremerhaven wird daher besonders Wert auf das Verstehen und Beherrschen einzelner Werkzeuge und deren effektive Verknüpfung gelegt.
Inspiriert von der Unix-Philosophie, die besagt, dass Programme jeweils nur eine Aufgabe erfüllen und deren Ein- und Ausgabe mit anderen Werkzeugen kombiniert werden sollte [15, K. 1], werden Werkzeuge durch Skripte oder definiert Schnittstellen miteinander verbunden. Auf diese Weise arbeiten beispielsweise der Code-Editor und der Compiler unabhängig voneinander und greifen beide auf die Quellcodedateien im Dateisystem zu. Die Werkzeuge lassen sich somit einzeln austauschen und flexibel kombinieren.

Eine solche Sammlung an Werkzeugen ermöglicht eine hohe Anpassungsfähigkeit an verschiedene Technologien und Projektanforderungen und kann als orthogonal beschrieben werden. In orthogonalen Softwaresystemen sind die einzelnen Komponenten voneinander unabhängig, beeinflussen sich nicht gegenseitig und haben getrennte Aufgaben, die sich nicht überschneiden sollten [15, K. 4].
Viele dieser Werkzeuge können projekt- und technologieübergreifend eingesetzt werden. So lassen sich Aufgaben wie die Bearbeitung von Quellcode oder die Versionsverwaltung unabhängig von anderen Schritten im Entwicklungsprozess und in unterschiedlichen Projekten mit denselben Werkzeugen durchführen. Dies gilt sowohl für einfache *Bash* Skripte als auch für komplexe Java-Webanwendungen.

Ziel dieser Arbeit ist es, eine flexible Entwicklungsumgebung auf Basis dieser orthogonalen Werkzeuge für die Lehre im Bereich verteilter Datenbanksysteme zu konzipieren.

2.2 Containerisierter Workspace

Besonders für Studienanfänger:innen, die noch nie mit den verwendeten Technologien gearbeitet haben, kann das Einrichten einer Entwicklungsumgebung erheblichen Aufwand und Schwierigkeiten bedeuten.
Auch ist zu erwarten, dass die von Studierenden eingerichteten Entwicklungsumgebungen sich durch unterschiedliche Vorkenntnisse und unterschiedlicher Hardware und Betriebssystemen teilweise stark voneinander unterscheiden. Für die Lehre stellt das ein Problem dar, weil Dozent:innen mehr Zeit darauf verwenden müssen die einzelnen Umgebungen zu verstehen, um effektive Hilfestellung zu geben.
Beides spricht dafür eine fertige Entwicklungsumgebung vorzugeben und für alle Studierende bereitzustellen.

Die Bereitstellung einer Entwicklungsumgebung, die auf orthogonalen Unix Werkzeugen basiert, lässt sich mit einem Linux Container realisieren. Der Container beinhaltet alle benötigten Werkzeuge, Bibliotheken und Beispielprojekte und kann für jede:n Entwickler:in instanziiert werden.
Die Entwickler:innen können von ihrem Endgerät über SSH eine Shell in dem Container öffnen und darin auf der Kommandozeile Projekte anlegen und darin arbeiten.

Als Alternative kann *VSCodium*, der Open-Source-Build von den *VSCode*- Quelldateien [16], mit der Erweiterung *Open Remote - SSH* [17], auf den Container zugreifen und ihn als Remote Workspace verwenden [18].

Wie in Abbildung 2.1 zu sehen ist, kann *VSCodium* in diesem Workspace Quellcode bearbeiten und Programme ausführen, testen und debuggen. Dabei kann der Workspace entweder auf dem lokalen Gerät, einem Server oder der Cloud gestartet werden.

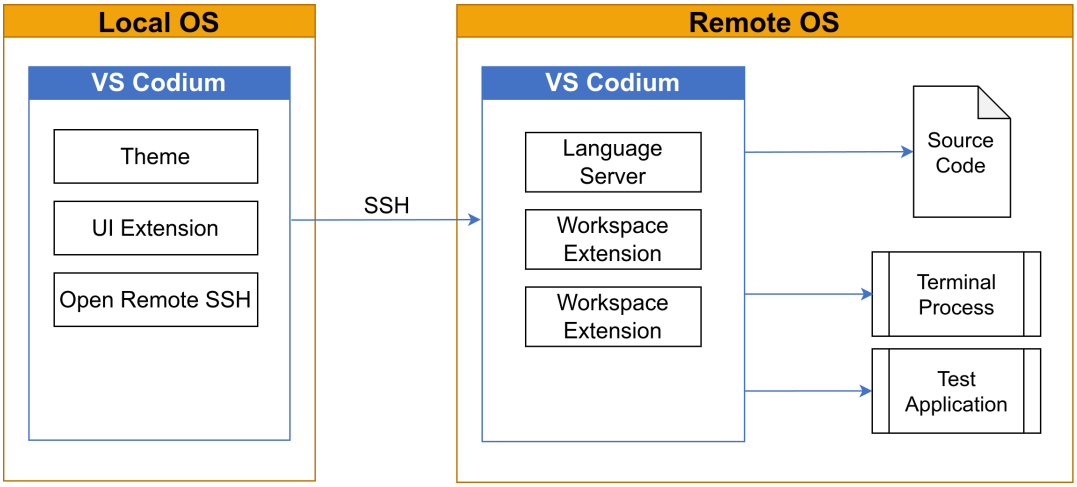

Abbildung 2.1: Remote Development mit VS Codium

Prozesse wie das Kompilieren und Ausführen des Projektes, sowie Editorerweiterungen beanspruchen dabei die Ressourcen des Containers und nicht des Endgeräts auf dem die *VSCodium* Oberfläche läuft.

Wenn der Container auf einem geteilten Server läuft, können dadurch auf dem Endgerät der Entwickler:innen Ressourcen gespart werden und so die Verwendung von leistungsschwächerer Hardware unterstützt werden [19, S. 20].

Github Codespaces sind Microsofts Version dieser Idee. Hier wird in der Cloud ein Docker Container, der den Workspace mit der Dateistruktur, Bibliotheken, Compilern und weiteren nützlichen Tools beinhaltet, gestartet. *Visual Studio Code* als Desktop Programm oder als Web-App kann sich dann mit diesem Workspace verbinden [20].
Auch die *JetBrains* IDEs haben ein ähnliches Feature, welches aber nur in der Ultimate Edition verfügbar ist [21].

Das *Language Server Protokoll* bietet die Möglichkeit einige gefällige, hilfreiche Features von IDEs in diverse Texteditoren zu integrieren. *Language Server*, die das Protokoll implementieren, abstrahieren Aktionen, wie das Springen zu Definitionen oder Referenzen, Fehlerdiagnosen oder Code Refactoring von der jeweiligen Sprache und stellen diese Informationen und Aktionen für Editoren oder andere Werkzeuge bereit. So kann ein Editor mit unterschiedlichen *Language Servern* für unterschiedliche Programmiersprachen kommunizieren und dabei die gleichen oder ähnliche Aktionen ausführen.
Die Echtzeit-Analyse des Quellcodes und die in den Editor integrierte Anzeige von Syntax Fehlern sowie intelligente Code Vervollständigung sind Features, die oft ein Grund für die Verwendung einer IDE sind. *Language Server* stellen diese Features unabhängig vom Rest der Entwicklungsumgebung über das Protokoll zur Verfügung. Ein einfacher Editor kann so für viele Sprachen auf dieselbe Weise verwendet werden. Damit müssen Studierende nicht für jede

Sprache eine neue IDE kennenlernen, sondern können sich auf die Eigenschaften der Sprache konzentrieren.

Die Trennung von Editor und Code Analyse ermöglicht es außerdem diese Features auch beim Remote-Entwickeln zu Verwenden. Bei der Verwendung eines Containers als Remote Workspace wird der *Language Server* in dem Workspace ausgeführt und verwendet die Ressourcen des Containers.
In einem Hochschulprojekt konnte festgestellt werden, dass bei einem typischen Java Entwicklungsvorgang von Studierenden der *Language Server* deutlich mehr Ressourcen benötigt als alle anderen Komponenten und Schritte im Entwicklungsprozess [19, S. 20].

Der Workspace Container beschränkt sich dabei auf Programme ohne grafische Oberfläche. GUI-Programme wie zum Beispiel ein Code-Editor sollten direkt auf den jeweiligen Endgeräten installiert werden und mit Protokollen wie *SSH* oder *HTTP* mit den Werkzeugen und Servern in dem Container verbunden werden.

2.3 Virtuelle Maschine als Docker-Host

Im Modul Vernetzte Systeme an der Hochschule Bremerhaven wird das Erzeugen und Verwalten von Docker Containern auf dem lokalen Gerät gelehrt. Dazu erstellen die Studierenden eine virtuelle Maschine in der eine Docker Umgebung aufgebaut werden kann, meistens ein Ubuntu Linux. Auf Windows Geräten wird die VM über das *Windows-Subsystem für Linux (WSL)* realisiert, auf Macs und Linux Geräten werden die VMs mit *Lima* erstellt. Mit der VM haben alle eine gemeinsame Basis auf der die Docker Engine installiert werden kann. Lediglich die CPU-Architektur unterscheidet sich.

Geöffnete Ports innerhalb der VM werden sowohl von dem WSL [22], als auch von Lima [23] automatisch an den Host weitergereicht.

2.4 Nicht-funktionale Anforderungen

Die Konzeption einer Entwicklungsumgebung für den Einsatz in der Informatiklehre erfordert eine differenzierte Betrachtung der Qualitätsmerkmale, da sich die Anforderungen teilweise von denen produktiver Softwaresysteme unterscheiden. Während in industriellen Umgebungen vor allem die äußeren Softwarequalitäten, wie Zuverlässigkeit, Performanz und Gebrauchstauglichkeit im Fokus stehen, rücken in der Lehre die inneren Softwarequalitäten mehr in den Vordergrund.
Dies liegt darin begründet, dass Studierende nicht nur die Nutzung der Werkzeuge erlernen, sondern vor allem ein tiefgreifendes Verständnis der zugrundeliegenden Konzepte und Prozesse entwickeln sollen.
Die ISO/IEC 25010 definiert zwar umfassende Kriterien für Softwarequalität [24, S. 71-73],

doch viele dieser Merkmale sind primär auf produktive Systeme ausgelegt und müssen für den Lehrkontext neu gewichtet werden. Vor allem die Merkmale Transparenz, Wartbarkeit und Nachhaltigkeit sollen hier hervorgehoben werden.

2.4.1 Transparenz

Transparenz beschreibt die Eigenschaft eines Systems, keine versteckten Mechanismen oder unerwartetes Verhalten aufzuweisen, und ist damit ein zentrales Kriterium für den Lehrbetrieb. Eric S. Raymond betont in „The Art of Unix Programming“ [15, K. 6], dass *Transparenz* es den Nutzer:innen ermöglicht, die Funktionsweise eines Systems vollständig zu durchdringen.
Im Kontext der Informatiklehre bedeutet dies, dass alle Prozesse und Werkzeuge nachvollziehbar und sichtbar sein müssen, um Studierenden ein umfassendes Verständnis der zugrundeliegenden Prinzipien zu vermitteln.
Ein *transparentes* System verzichtet auf unnötige Abstraktionen oder automatische Fehlerkorrekturen, die zwar die Nutzung vereinfachen, aber das Lernen erschweren würden. Stattdessen sollten Fehlermeldungen und Logs explizit dargestellt werden, um Studierenden die Möglichkeit zu geben, Probleme selbstständig zu analysieren und zu lösen.

Die *Einfachheit* der Architektur spielt dabei eine entscheidende Rolle. Ein System gilt als einfach, wenn seine Komponenten weitgehend unabhängig voneinander funktionieren oder durch klare Schnittstellen verbunden sind. Einfachheit kann hier im Sinne von geringer Komplexität, also Verwobenheit verstanden werden.

Neben *Transparenz* als passive Eigenschaft beschreibt Eric S. Raymond *Zugänglichkeit* als aktive Eigenschaft um den Einstieg in ein Softwaresystem zu erleichtern.
Eine gut strukturierte Dokumentation sowie beschreibende Namen für Variablen, Funktionen und Module erhöhen die *Zugänglichkeit* und tragen dazu bei, den Einstieg in die Entwicklungsumgebung zu erleichtern [15].

Um ein tieferes Verständnis für die verwendete Software und die zugrunde liegenden Konzepte zu erlangen, ist es sinnvoll eine Entwicklungsumgebung für die Lehre transparent und zugänglich zu gestalten.

2.4.2 Wartbarkeit

Wartung ist von vom IEEE als Prozess der Veränderung eines Softwaresystems nach der Inbetriebnahme zur Fehlerbehebung, Leistungsverbesserung oder Anpassung an eine veränderte Umgebung definiert [25]. Wartbarkeit kann als die Leichtigkeit dieses Prozesses verstanden werden. Es ist Anzunehmen, dass für die Lehre konzipierte Softwaresysteme über mehrere Jahre hinweg eingesetzt werden. Angesichts der begrenzten Ressourcen für Wartung und Weiterentwicklung erweist sich eine hohe Wartbarkeit als unverzichtbar, um den langfristigen Betrieb zu

gewährleisten.
Ein modularer Aufbau mit klar definierten Schnittstellen ermöglicht es, einzelne Komponenten isoliert zu verstehen, zu erweitern oder auszutauschen. Dies reduziert nicht nur den Wartungsaufwand, sondern erleichtert auch die langfristige Anpassung an neue Lehrinhalte oder technologische Entwicklungen.
Externe Abhängigkeiten führen hingegen zu einem erhöhten Wartungsaufwand, da sie regelmäßige Aktualisierungen oder den Austausch veralteter Komponenten erfordern. Um den langfristigen Pflegeaufwand zu minimieren, sollten sie daher auf ein notwendiges Minimum reduziert werden.
Wartbarkeit steht dabei in engem Zusammenhang mit der Transparenz des Systems, da ein gut strukturiertes und dokumentiertes Softwaresystem leichter zu pflegen ist.

2.4.3 Nachhaltigkeit

Ein weiteres zentrales Qualitätsmerkmal ist die *Nachhaltigkeit*, die in der Informatik in drei Dimensionen betrachtet werden kann: ökologisch, sozial und ökonomisch.

Ökologische Nachhaltigkeit zielt darauf ab, ressourcenschonende Lösungen zu schaffen. Zum einen können dadurch Energiekosten eingespart werden. Zum anderen ermöglicht es aber auch die Verwendung von leistungsschwacher Hardware und kann somit die Lebensdauer älterer Geräte verlängern. Insbesondere bei Endgeräten wie Laptops fällt die Lebensdauer deutlich mehr ins Gewicht als die direkten Energiekosten bei der Verwendung, da hier der größte Teil der CO2e-Emissionen in der Herstellungsphase anfällt [26].
Da Studierende oft schon vor dem Studium über unterschiedliche Hardware verfügen, ist neben ressourcenschonender Software auch die Kompatibilität der eingesetzten Software mit verschiedener Hardware relevant. Ein hoher Ressourcenbedarf oder mangelnde Kompatibilität mit den gängigsten Geräten könnten die Teilhabe erschweren und würde zu einem erhöhten Bedarf an neuer Hardware führen.
In der Lehre verwendete Software sollte daher ressourcenschonend und kompatibel mit den Betriebssystemen Linux, macOS und Windows sowie den CPU-Architekturen x86_64 und ARM64 sein.

Soziale Nachhaltigkeit zeigt sich in der Informatik unter anderem durch offenen und diskriminierungsfreien Zugang zu Bildung und digitalen Ressourcen ab.
Durch den Einsatz freier und offener Technologien wird nicht nur die Souveränität der Nutzer:innen und Entwickler:innen gefördert, sondern auch eine gerechtere Verteilung von Wissen ermöglicht.
Dies ist besonders relevant, da wirtschaftliche Abhängigkeiten im Bereich der Softwareentwicklung Machtungleichgewichte zwischen industrialisierten und weniger entwickelten Regionen reproduzieren können. Open-Source-Software bietet hier einen Gegenentwurf, indem sie Entwickler:innen in weniger privilegierten Regionen Zugang zum intellektuellen Kapital der Industriestaaten ermöglicht und somit zur Demokratisierung von technologischem Wissen

beiträgt [27, S. 22-23].
Im Kontext der Informatiklehre kann durch Open-Source-Software der freie Zugang zu Technologien und Wissen gewährleistet und Abhängigkeiten von Technologiekonzernen vermieden werden.

Ökonomische Nachhaltigkeit wird durch die Verwendung kosteneffizienter Hard- und Software und offener Standards erreicht. Dadurch können Lizenzkosten gespart werden, eine langfristige Nutzung von Software ermöglicht und Lock-in-Effekte vermieden werden.

Die Verwendung von Open-Source-Software trägt maßgeblich zum Umsetzen dieser Nachhaltigkeitskonzepte bei und wird in der Umsetzung der Entwicklungsumgebung konsequent vorausgesetzt.

2.4.4 Zuverlässigkeit

Zuverlässigkeit beschreibt die Fähigkeit eines Systems die zugewiesenen Aufgaben unter bestimmten Bedingungen in einer definierten Zeitspanne zu erfüllen. Zuverlässige Systeme sind robust gegenüber Fehleingaben und haben eine hohe Verfügbarkeit. Das ist besonders für sicherheitskritische Produktivsysteme relevant [24, S. 72].
Im Rahmen der Informatiklehre ist zwar die Verfügbarkeit und Funktionalität eines Softwaresystems von Relevanz, jedoch unterscheiden sich die Anforderungen an die Fehlerbehandlung grundlegend von denen in Produktivumgebungen. Während in produktiven Systemen die Minimierung von Fehlern und die Effizienz der Aufgabenbewältigung im Vordergrund stehen, liegt der Schwerpunkt in Lehrsystemen darauf, Fehler explizit aufzuzeigen, um den Lernprozess zu fördern. Durch das Lesen von Fehlermeldungen und Logs kann ein System besser verstanden werden. Ein System, das falsche Eingaben automatisch korrigiert, wäre in der Lehre kontraproduktiv.

3 Umsetzung einer Entwicklungsumgebung für verteilte Datenbanksysteme

Als Beispiel für diese allgemeine Idee einer didaktischen Entwicklungsumgebung wird im Folgenden eine konkrete Umsetzung beschrieben.
In dem Modul Datenbanken II an der Hochschule Bremerhaven beschäftigen sich Studierende unter anderem mit dem Aufbau und der Verwendung von verteilten Datenbanksystemen zur Verarbeitung von Big Data. Hierzu gehören die Technologien *Apache Hadoop*, *Apache Hive*, *Apache HBase* und *Apache Kafka*.
Dabei soll es weniger um die Installation und Administration der Technologien als um das Verwenden und das Verstehen der Konzepte hinter verteilten Systemen gehen.

Die Erfahrungen aus dem Lehrmodul zeigen, dass die Installation und Inbetriebnahme der Technologien bei vielen Studierenden Probleme verursachen und teilweise gar nicht durchgeführt werden konnten. Da diese Technologien nur einen Teil des Moduls ausmachen, ist es kaum möglich in der vorgesehenen Zeit eine passende Entwicklungsumgebung für die Bearbeitung der Aufgaben einzurichten.
Auch mit der *Big Data Lite VM* von *Oracle* konnte nicht immer eine funktionierende Umgebung bereitgestellt werden. Der Zugang zu der *Big Data Lite VM* ist nur durch die Anmeldung mit einem *Oracle* Konto möglich und die Lizenz verbietet das Verteilen der Umgebung über andere Plattformen wie z. B. einem Hochschulserver. Außerdem darf die virtuelle Maschine außerhalb der Lehre nur auf *Oracle* Systemen verwendet werden [6].
Diese Probleme und Einschränkungen sind die Motivation um eine alternative, konsequent Open-Source-basierte Entwicklungsumgebung für diese Technologien zu schaffen.

3.1 Funktionale Anforderungen

Aus den bisherigen Lehrmaterialien und der Beschreibung des Lehrmoduls ergeben sich folgende Funktionen, die die Entwicklungsumgebung mindestens unterstützen muss.
Als Basis für mehrere verteilte Datenbanksysteme wird ein persistentes *Hadoop Distributed Filesystem (HDFS)* vorausgesetzt. Darüber hinaus sind ein persistenter *Hive* Server sowie ein persistenter *HBase* Server erforderlich, die jeweils auf das *HDFS* zugreifen können. Zudem muss die Umgebung einen *Kafka Broker* unterstützen.
Diese Technologien werden im folgenden Kapitel 3.2 genauer beschrieben.

Zusätzlich muss eine Shell mit entsprechenden Werkzeugen existieren über die Nutzer:innen mit den installierten Servertechnologien interagieren können.
Ergänzend dazu müssen Werkzeuge für die Entwicklung von Java Programmen verfügbar sein. Damit diese Programme mit den installierten Servertechnologien kommunizieren können, müssen die entsprechenden Bibliotheken verfügbar sein. Die Nutzung dieser Bibliotheken soll mit Java Beispielen demonstriert werden.

Darüber hinaus gelten die folgenden Rahmenbedingungen. Die Entwicklungsumgebung soll lokal auf den Geräten der Studierenden installiert werden. Dadurch muss keine zusätzliche Hardware bereitgestellt und administriert, und kein Autorisierungsverfahren implementiert und verwaltet werden.
Dabei sollte die Installation trotzdem möglichst ähnlich zu einem echten Clustern sein. Obwohl Hadoop einen Standalone-Modus unterstützt, in dem alle Komponenten in einem einzigen Java-Prozess laufen, wird dieser bewusst nicht verwendet, da er vorrangig für Debugging-Zwecke empfohlen wird und die Konfiguration eines echten Clusters nicht ausreichend abbildet.

3.2 Verwendete Technologie

Im Folgenden werden die Kernkomponenten der Entwicklungsumgebung beschrieben: *Hadoop*, *Hive*, *HBase* und *Kafka*.

Apache Hadoop ist ein Framework für das Verarbeiten von großen Datenmengen in einem verteilten Netzwerk. Es besteht aus dem *Hadoop Distributed Filesystem (HDFS)*, dem Resourcemanager *YARN* und *Mapreduce*, einem Framework für das Verarbeiten von Daten in dem *HDFS*.

Das *HDFS* ist ein verteiltes Dateisystem und kann Daten in einem Cluster von bis zu mehreren tausend Maschinen speichern. Die einzelnen Maschinen, die an dem Cluster beteiligt sind, werden auch Knoten genannt. Das *HDFS* ist für einen hohen Durchsatz entworfen und bildet die Grundlage für viele Technologien zur Verarbeitung von großen Datenmengen.
In einem *Hadoop* Cluster gibt es mehrere Server die traditionell auf unterschiedlichen Maschinen installiert werden.
Pro Cluster gibt es einen *Namenode*, der das Dateisystem und den Zugriff auf dieses verwaltet. Alle Anfragen an das Dateisystem gehen zuerst an den diesen Server.
Das Speichern der Daten im HDFS wird von den *Datanodes* verwaltet. Pro Knoten, auf dem Daten gespeichert werden sollen, gibt es einen *Datanode*. Beim Schreiben von Daten werden diese in Blöcke geteilt und auf die Knoten im Cluster verteilt und repliziert, sodass der Ausfall eines Knotens zu keinem Datenverlust führt. Diese Verteilung wird vom *Namenode* durchgeführt und ist sich der Topologie des Clusters bewusst. So kann sichergestellt werden das mindestens eine Kopie eines Blockes in einem zweiten Serverrack gesichert ist, um die Ausfallsicherheit zu erhöhen und eine bessere Ausnutzung der zur Verfügung stehenden Netzwerkbandbreite zu gewährleisten [28].
Beim Lesen der Daten wählt der *Namenode* Knoten aus, die möglichst dicht am lesenden Client liegen um den Netzwerkverkehr zu minimieren. Die Daten selber fließen dabei nicht über den *Namenode*, sondern es wird eine direkte Verbindung zwischen dem Client und den jeweiligen *Datenodes* hergestellt [28].
Da sowohl der *Namenode* als auch die *Datanodes* einen hohen Arbeitsspeicherbedarf haben, ist in produktiv eingesetzten Clustern, der *Namenode* meist auf einer eigenen Maschine installiert.

Diese braucht keinen Zugriff auf einen großen Datenspeicher, da der *Namenode* nur Metadaten über das Dateisystem und die Änderungen daran speichert.

In produktiv eingesetzten Clustern gibt es für das HDFS noch einen *Secondary Namenode*, der den *Namenode* wie folgend beschrieben unterstützt.
Der *Namenode* schreibt alle Änderungen am Dateisystem in eine Logdatei. Der aktuelle Zustand des Dateisystems ergibt sich aus dem Startzustand, gespeichert in der Datei *fsImage*, und den Logdateien, die seit dem Starten geschrieben werden. Diese Logdateien werden erst bei einem Neustart des *Namenodes* in das *fsImage* integriert und können mit zunehmender Zahl das Dateisystem verlangsamen. Der *Secondary Namenode* kann diese Logdateien, in einem laufenden Cluster, periodisch in das *fsImage* integrieren, um dieser Verlangsamung vorzubeugen [29].
Diese Funktion ist für große Cluster, die über längere Zeiträume hochgefahren sind, unabdingbar. Das vorliegende Hadoop Cluster wird aber für Verwendung in der Lehre auf Endgeräten von Studierenden vorgesehen. Es kann angenommen werden, dass das Cluster im Normalfall nur wenige Stunden am Stück in Betrieb sein wird.
Im Test wurden nach 50 Durchläufen der Beispielprogramme nur eine Logdatei mit einer Größe von 2 MB erstellt. Bei dieser Größenordnung ist keine relevante Verlangsamung zu erwarten. Auch der im User Guide [29] angegebene Wert von einer Million Transaktionen um den *Secondary Namenode* auszulösen, lässt darauf schließen, dass dieser Prozess in dem vorliegenden Cluster nicht notwendig ist. Aus Gründen der Einfachheit wird deshalb auf einen *Secondary Namenode* verzichtet.

Neben dem Speichern der Daten bietet das Hadoop Framework auch Werkzeuge, um die gespeicherten Daten zu Analysieren und zu Verarbeiten.
In Anwendungen mit geringen Datenmengen werden traditionell die Daten dorthin bewegt, wo die Rechenkapazität liegt, z. B. von einem Datenspeicher im Netzwerk in den lokalen Speicher. Bei der Verwendung von *Big Data* ist es oft nicht möglich, oder zumindest nicht sinnvoll, die großen Datenmengen auf einen einzigen Rechner zu kopieren. Stattdessen ist jeder Datenknoten mit genug Rechenleistung ausgestattet, um die Daten direkt auf diesen Knoten auszuwerten. So kann Netzwerkverkehr vermieden werden und die Rechenkapazität von vielen Maschinen im Cluster effizient genutzt werden.
Um die Daten dort zu Verarbeiten, wo sie liegen, braucht es Programme, die an die jeweiligen Knoten gesendet werden, um sie dort auszuführen. *Hadoop MapReduce* ist ein Framework für solche Programme. Es ist eine Open-Source-Implementation des *MapReduce* Modells, das ursprünglich von Google entwickelt wurde und bildet die Basis für die meisten Jobs, die in einem Hadoop Cluster ausgeführt werden.
Das *MapReduce* Modell besteht aus drei Phasen:
In der *Map*-Phase werden alle Eingabedaten auf einem Knoten in eine Form von Key-Value-Paaren umgewandelt.
Die *Shuffle*-Phase sammelt alle Paare mit demselben Schlüssel auf einem Knoten und gibt sie an den *Reduce*-Job weiter.
In der *Reduce*-Phase werden diese Paare zu einem Ergebnis pro Schlüssel zusammengefasst und an den Client gesendet [30].

Die genaue Implementierung der *Map* und *Reduce* Funktionen ist dabei den Anwender:innen überlassen und bildet die Logik der Aufgabe ab.

Ein einfaches und oft genutztes Beispiel ist das Zählen von Wörtern in einem großen Text.
In der *Map*-Phase wird in jedem Knoten der dort liegende Teil des Textes durchlaufen und jedes Wort in ein Paar mit dem Wort als Schlüssel und der Anzahl des Wortes als Wert abgelegt. Das Framework sammelt in der *Shuffle*-Phase alle Paare mit dem gleichen Wort auf einem Knoten und übergibt sie an die *Reduce* Funktion. Diese muss bei diesem einfachen Beispiel nur die Werte aller Paare, die sie bekommen hat zusammenzählen. Das Ergebnis wird dann von allen *Reduce* Jobs an den Client gemeldet.

Für die Verteilung dieser Arbeit auf die Knoten ist *YARN (Yet Another Resource Negotiator)* zuständig. *YARN* läuft einmal pro Cluster als *Resourcemanager* und pro Datenknoten als *Nodemanager*. Die *MapReduce* Jobs werden vom Client an den *Resourcemanager* gesendet und von dort an die passenden *Nodemanager* gesendet, wo die Programme, mit den lokalen Daten als Eingabe, ausgeführt werden.
YARN beinhaltet auch den *Historyserver*, welcher Informationen zu erledigten Jobs speichert. Dadurch können *Resource-* und *Nodemanager* erledigte Jobs aus ihrem Speicher entfernen und die Speicheranforderungen für diese Komponenten klein halten. Der *Historyserver* muss dagegen mehr Speicher, aber weniger Rechenleistung haben. Durch diese Trennung können die Funktionen auf sehr angepasster Hardware installiert werden.

Apache Hive ist ein Data-Warehouse-System das auf *HDFS* aufgesetzt werden kann um Daten in einer *SQL*-ähnlichen Sprache (*HiveQL*) abzufragen oder zu bearbeiten. Dabei arbeitet *Hive* mit Daten, die im *HDFS* gespeichert sind und speichert selber nur Metadaten darüber wie die Daten im Dateisystem zu interpretieren sind [31].
Die *HiveQL*-Abfragen werden dabei in *MapReduce* Jobs umgewandelt und anschließend zur Bearbeitung an den *Resourcemanager* gesendet. Dieser Prozess lässt sich in den Logausgaben von *Hive* und der Weboberfläche des *Resourcemanagers* nachvollziehen. *Hive* selber benötigt kaum Ressourcen, da das Speichern der großen Datenmengen vom *HDFS* und das Ausführen der Abfragen von *MapReduce* auf den *Nodemanagern* übernommen wird.
Neben dem *Hive* Server, der die *HiveQL* Abfragen entgegennimmt und umwandelt, besteht *Hive* aus einem *Metastore* und der dazugehörigen Datenbank *PostgreSQL* um Metadaten über Tabellen zu speichern. Diese drei Komponenten können auf drei unterschiedlichen Maschinen installiert werden und kommunizieren über ein Netzwerk [32].

Apache HBase ist ein verteiltes, nicht-relationales Datenspeichersystem, das darauf spezialisiert ist sehr große, nicht strukturierte Datenmengen zu speichern und abzurufen. Die Daten sind dabei spaltenorientiert abgelegt [33].
Auch *HBase* verwendet *HDFS* als zugrundeliegendes Dateisystem. Dabei setzt *HBase* aber nicht auf *MapReduce* und den bestehenden *Resourcemanager*, sondern installiert auf jedem Knoten, auf dem Daten gespeichert werden, einen *Regionserver*, der das Speichern und Lesen der Daten übernimmt [34, K. 70].
Pro Cluster gibt es einen aktiven Hauptserver, der die knotenübergreifenden Aufgaben wie

Region-Failover und das Aufteilen von Daten auf die Regionen übernimmt. Der Server übernimmt damit eine ähnliche Rolle wie der *Namenode* im HDFS und wird auch oft auf derselben Maschine installiert [34, K. 69]. Die meiste Kommunikation mit den Clients und den Austausch der Daten übernehmen aber die *Regionserver*.
Zur Koordination innerhalb des Clusters wird *Zookeeper* eingesetzt [34, K. 206].

Apache Kafka ist eine Event-Streaming-Plattform, in der Echtzeitdaten von verschiedenen Quellen als Ereignisse aufgenommen, in Echtzeit verarbeitet und an verschiedenen Technologien weitergeleitet werden. Ein *Kafka* Cluster besteht aus *Brokern*, die Ereignisse dauerhaft speichern und *Kafka Connect* Servern die Daten in und aus dem Cluster führen. Als Clients können von *Kafka* mitgelieferte Kommandozeilenwerkzeuge oder Bibliotheken für Java und viele weitere Programmiersprachen verwendet werden.

3.3 Aufbau der Entwicklungsumgebung

Als Basis für die Entwicklungsumgebung wird eine virtuelle Maschine mit Ubuntu 24.04, ohne grafische Oberfläche vorgesehen. Prinzipiell kann aber jede, Debian-basierte, Linux Distribution genutzt werden. Die Mindestanforderungen für diese virtuelle Maschine sind 4 GB Arbeitsspeicher und 15 GB freier Speicherplatz für das Git Repository und das Herunterladen der Docker Images.
Für die Konfiguration dieser VM können z. B. die Virtualisierungstools *Lima* auf Linux und MacOS und das *Windows-Subsystem für Linux (WSL)* auf Windows verwendet werden. Beides sind bedienungsfreundliche Werkzeuge zum einfachen Erstellen und Verwalten von virtuellen Maschinen und werden bereits in der Informatiklehre an der Hochschule Bremerhaven verwendet.

Als einzige zusätzliche Software wird die Docker Engine installiert. Die im Kapitel 3.2 beschriebenen Serverkomponenten sind als Docker Images bereitgestellt. Daraus werden zum Hochfahren der Entwicklungsumgebung beliebig viele Container erzeugt, um die Maschinen in einem Cluster zu simulieren.

Wie in Abbildung 3.1 zu sehen ist, lassen sich diese Container zu den Modulen *HDFS*, *Hive*, *HBase* und *Kafka* zusammenfassen. Das Betrachten der Umgebung auf dieser Modulebene erleichtert das Bedienen und Verstehen der Entwicklungsumgebung.
Ein Modul besteht dabei aus einem oder mehreren Servercontainern, den Images, aus denen sie erzeugt werden und den *Dockerfiles* aus denen diese Images gebaut werden. Darüber hinaus gehören dazu die Konfigurationsdateien, die in alle Container eines Moduls eingebunden werden sowie Bibliotheken und Beispielprojekten, die in den Workspace eingehängt werden. Für die Steuerung ist außerdem pro Modul ein *Bash* Skript vorgesehen.
Im Kapitel 3.6 wird die Steuerung der Module näher betrachtet.

Die Abhängigkeiten der Technologien sind auch auf der Modulebene implementiert. Die Module *Hive* und *HBase* greifen auf das *HDFS* zu und können daher nur starten, wenn das Modul *HDFS*

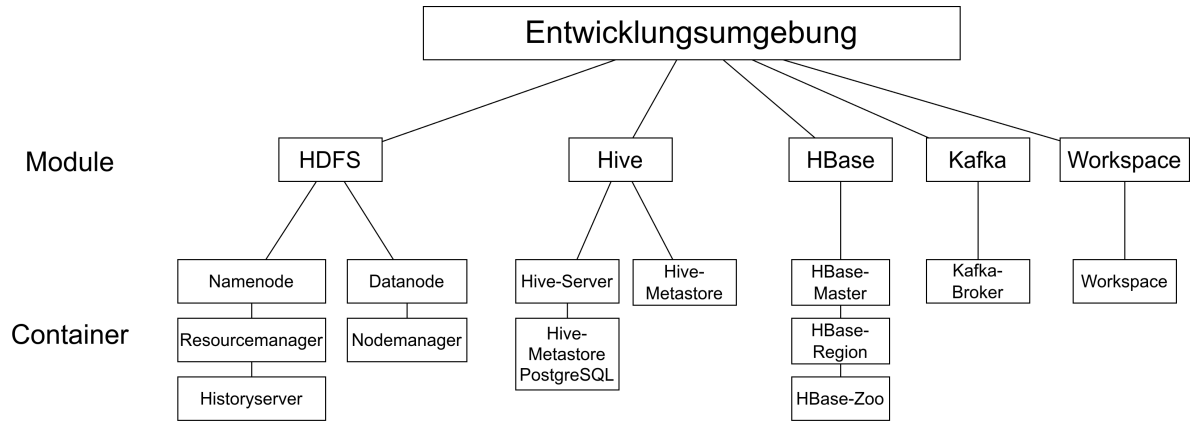

Abbildung 3.1: Einteilung der Container in Module

bereits läuft. Beim Starten von Modulen werden daher alle benötigten Module geprüft und sichergestellt, dass diese vorher gestartet werden.

Ausgangspunkt der Entwicklungsumgebung ist ein Git-Repository mit *Dockerfiles*, Skripten, Konfigurationsdateien, Beispielprojekten und Dokumentationen für einen zugänglichen Einstieg in die verwendete Software.
Aus diesem Repository können alle Docker Images erzeugt und die einzelnen Module und der Workspace gestartet werden.
In der enthaltenen *Readme*-Datei sind die Schritte zum Herunterladen oder Erstellen der Docker-Images und die Befehle zum Bedienen der Entwicklungsumgebung beschrieben.

3.4 Workspace-Container

Neben den im Hintergrund laufenden Server-Container der einzelnen Module gibt es einen Workspace-Container, von dem aus die Umgebung verwendet wird.
In dem Container läuft ein SSH-Server, über den sich die Benutzer:innen in dem Workspace einloggen und arbeiten können. Das SSH-Protokoll dient als offene Schnittstelle, über die der Workspace von diversen Programmen oder von anderen Maschinen verwendet werden kann. Das ist besonders relevant, um mit Programmen, die auf dem Gastsystem der virtuellen Maschine laufen, auf den Workspace zuzugreifen.

Für das Entwickeln von Java Programmen sind hier die benötigten Java Versionen, sowie die Java Bibliotheken von allen Modulen installiert. Die Kommandozeilenwerkzeuge `hbase shell`, `beeline`, die Shell für *Hive* und `hdfs dfs`, für den Zugriff auf das *HDFS*, können aus dem Workspace-Container aufgerufen werden und verbinden sich mit den jeweiligen Server-Containern.

Das Homeverzeichnis im Workspace Container ist ein eingehängtes, benanntes Docker-Volume, um erstellte Projekte dauerhaft zu speichern. In dem persistenten Homeverzeichnis kann auch neue Software installiert werden, ohne dass der Container neu gebaut werden muss. Zum Beispiel kann ein zusätzlicher Kommandozeileneditor, in das Verzeichnis *$HOME/bin*, installiert werden.

In dem Workspace-Container ist zudem ein Verzeichnis mit Java-Beispielprogrammen für die verschiedenen Module eingehängt. Dieses Verzeichnis dient als Startpunkt für die Verwendung des Containers zur Entwicklung von Java Programmen. Die Beispielprogramme bestehen aus minimalen Quelldateien und jeweils einem Bash-Skript zum Kompilieren und Ausführen der Beispiele.

Der Workspace ist auch als Modul entworfen, um dieselben Skripte und Mechaniken wie die anderen Module verwenden zu können. Die oben erwähnte Mechanik zum Erkennen von Abhängigkeiten und Starten von benötigten Modulen kann genutzt werden, um den Workspace bei jedem Modul mitzustarten.

Abbildung 3.2 zeigt alle Container die an der Entwicklungsumgebung beteiligt sind.
Der Workspace Container befindet sich im selben Docker-Netzwerk wie die Container mit den Server Komponenten und kann über die entsprechenden Hostnamen und Ports auf diese zugreifen.

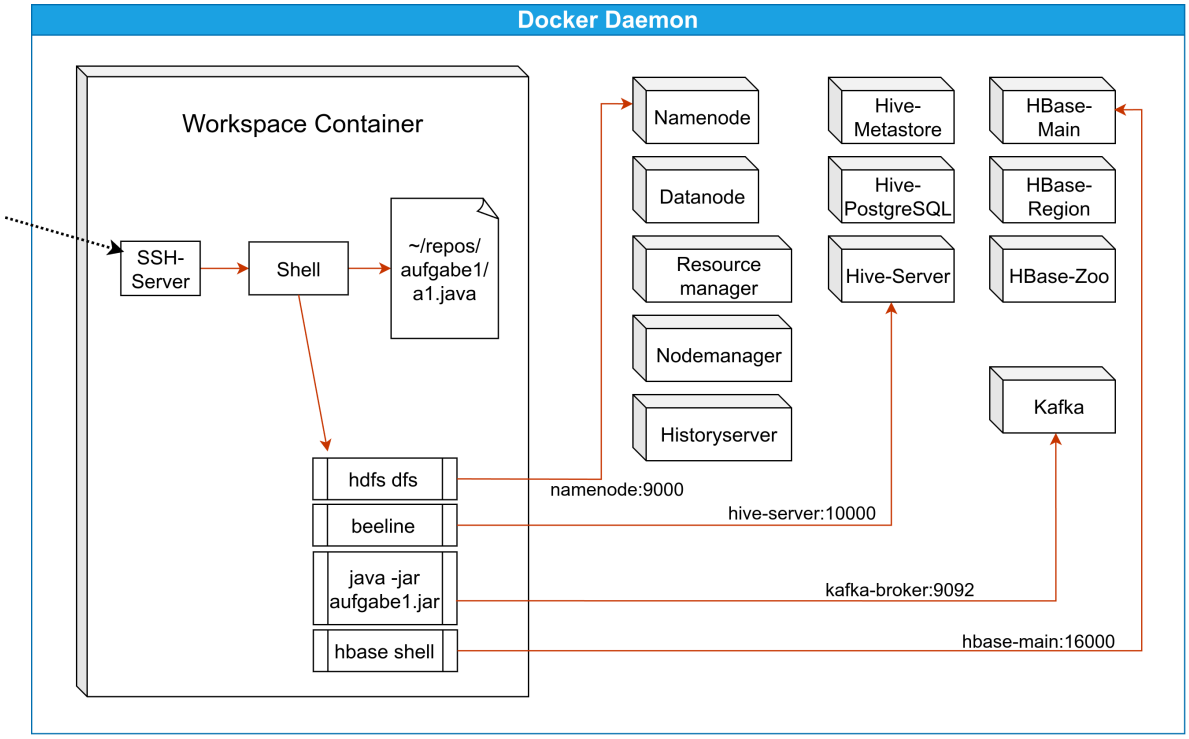

Abbildung 3.2: Container Architektur

3.5 Editoren

Die vorliegende Entwicklungsumgebung beinhaltet keinen festgelegten Texteditor. Es wird erwartet, dass Nutzer:innen bereits einen favorisierten Editor auf ihrem Gerät installiert haben. Daher liegt der Fokus des vorliegenden Projektes eher darauf, Zugangsmöglichkeiten für bereits installierte Editoren zu schaffen.
Dafür wird das Konzept des *Remote Developments* angewendet. Die grafische Oberfläche von Editoren wie *VSCodium*, *IntelliJ IDEA* oder *Eclipse* können dabei direkt auf dem Hostsystem und damit auch auf anderen Betriebssystemen laufen. Die genannten Editoren bieten, meist durch Erweiterungen, die Möglichkeit in dem Workspace einen Server zu installieren. Über diesen Server können die Editoren auf das Dateisystem, eine Shell und andere Systeme des Workspace zugreifen. Die Kommunikation zwischen der Oberfläche und dem Server des Editors erfolgt über das SSH-Protokoll.

Eine weitere Möglichkeit besteht darin, ein Verzeichnis vom Host-Gerät in die Ubuntu-VM zu mounten und dieses Verzeichnis in der VM dann in den Workspace-Container zu mounten. Dann können Dateien von außen auf dem Host-Gerät editiert werden und gleichzeitig innerhalb des Containers gebaut und ausgeführt werden. Dadurch geht allerdings die Funktion innerhalb des Containers Prozesse, etwa für einen *Language-Server*, auszuführen verloren.

Kommandozeilenwerkzeuge wie `vim`, `sed` oder `nano` können direkt im Workspace-Container installiert und über die Shell verwendet werden um Quellcode zu bearbeiten.

Die als Container implementierten Serverkomponenten, der Workspace Container, die virtuelle Maschine auf der die Container laufen, die Editoren und die Host-Maschine ergeben zusammen die in der folgenden Abbildung 3.3 dargestellte Infrastruktur.

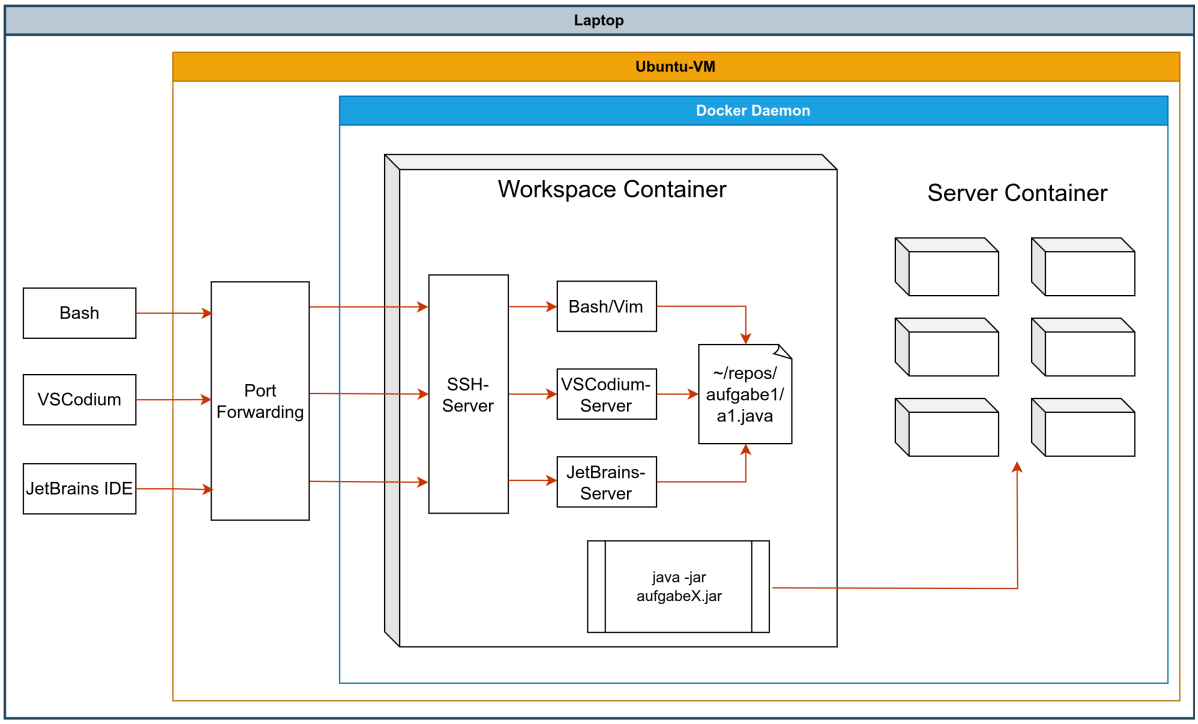

Abbildung 3.3: Architektur der Entwicklungsumgebung

3.6 Steuerung der Entwicklungsumgebung

Das Starten und Stoppen der Entwicklungsumgebung und der einzelnen Module sind in mehreren Bash-Skripten implementiert. Das Hauptskript `envctl.sh` kann mit den Kommandos `start`, `stop`, `restart` und optional einem oder mehreren Modulnamen aufgerufen werden: `bin/envctl.sh start hive hbase`.
Dabei kennt `envctl.sh` die einzelnen Module nicht, sondern schließt anhand des Modulnamens auf ein Skript zum Steuern des Moduls. In diesem Skript sind die Namen und Startoptionen von allen dazugehörigen Containern und die Reihenfolge beschrieben, in der sie gestartet werden müssen.
In der Abbildung 3.4 ist abgebildet wie das Skript zum Steuern der Entwicklungsumgebung die verschiedenen Skripte für die Module aufruft. Diese starten und stoppen die einzelnen Container.
Der Startvorgang der Module kann nicht weiter verallgemeinert werden, da innerhalb der Module unterschiedliche Abhängigkeiten und Startreihenfolgen beachtet werden müssen. Zum Beispiel können alle Container für das *Hive* Modul parallel starten, während die Container für das HDFS nacheinander starten müssen.

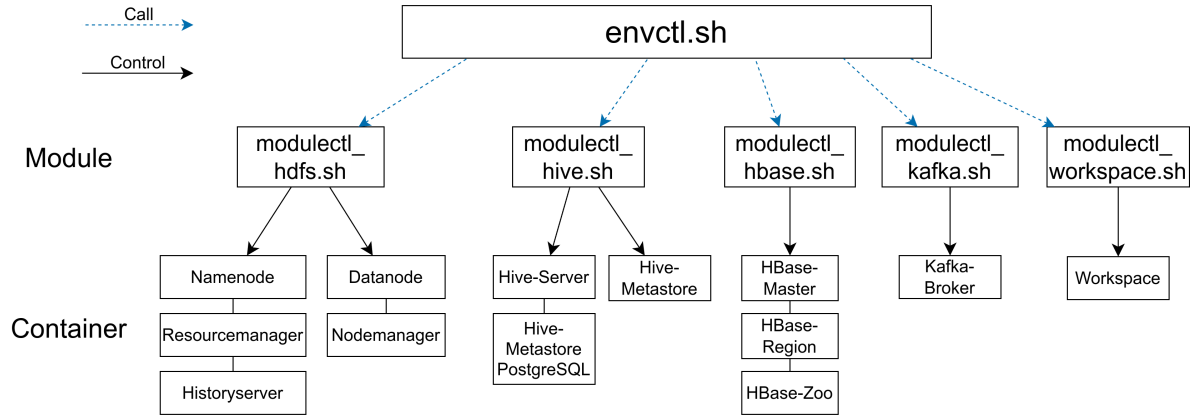

Abbildung 3.4: Steuerung der Entwicklungsumgebung mit Bash-Skripten

Die Schnittstelle zwischen den Skripten ist mit den Kommandos `start`, `stop`, `restart` zum Steuern der Module sowie `container` und `dependencies` festgelegt.
Über das Kommando `container` werden alle an dem Modul beteiligten Containernamen ausgegeben. Diese Funktion wird genutzt, um alle laufenden Container der Entwicklungsumgebung anzuzeigen.
Die Abhängigkeiten eines Moduls sind in dem Modulsteuerungs Skript beschrieben und werden mit dem Kommando `dependencies` ausgegeben. Bevor das Hauptskript ein Modul startet werden diese Abhängigkeiten abgefragt, und die Module in der richtigen Reihenfolge gestartet. Wie in der Terminalausgabe in Listing 3.1 zu sehen ist, werden für den oben genannten Befehl `bin/envctl.sh start hive hbase` nicht nur die Module *Hive* und *HBase* gestartet, sondern zunächst das Modul *HDFS* und der Workspace, von denen beide Module abhängig sind.
Der Workspace Container ist als ein eigenes Modul implementiert und als Abhängigkeit in allen Modulen eingetragen. So wird er automatisch immer mitgestartet.

```
1  $ bin/envctl.sh start hive hbase
2  starting workspace
3  starting namenode
4  starting datanode
5  waiting for hdfs safemode to turn off
6  hdfs safemode is off
7  starting resourcemanager
8  waiting for resourcemanager
9  starting nodemanager
10 starting historyserver
11 hdfs started
12 starting hive-metastore-postgresql
13 starting hive-metastore
14 starting hive-server
```

```
15 hive started
16 starting hbase-zoo
17 starting hbase-master
18 starting hbase-region
19 hbase started
```

Listing 3.1: Terminal Output: Starten der Module Hive und HBase

4 Diskussion der Ergebnisse

In diesem Kapitel werden die Ergebnisse der Implementierung analysiert und anhand der in Kapitel 2.4 definierten nicht-funktionalen Anforderungen Transparenz, Wartbarkeit und Nachhaltigkeit bewertet.

Für die Analyse der Wartbarkeit werden die Abhängigkeiten der verwendeten Technologien analysiert, um potenzielle Wartungsaufwände zu identifizieren. Anhand der Aktualisierungszyklen der einzelnen Komponenten wird ein Wartungszyklus abgeleitet, der sicherstellt, dass die Entwicklungsumgebung langfristig stabil und einsatzbereit bleibt.
Weiterhin liegt der Fokus auf der Plattformunabhängigkeit, die durch das Bauen von Multi-Plattform-Images erreicht wird. Diese Methode ermöglicht es, die Entwicklungsumgebung für unterschiedliche CPU-Architekturen wie x86_64 und ARM64 bereitzustellen.
Die Modularität der Architektur wird beispielhaft an der Integration eines neuen Moduls demonstriert.
Zudem wird diskutiert, welche Herausforderungen sich während der Entwicklung ergaben und welche Lösungsansätze gewählt wurden, um diese zu bewältigen.

4.1 Wartbarkeit

Für die Analyse der Wartbarkeit werden im Folgenden die Stellen identifiziert die durch externe Updates, Änderungen der Anforderungen in der langfristigen Nutzung angepasst werden müssen.

Hadoop wurde seit der Erstellung der Entwicklungsumgebung von der Version 3.2.1 auf die neueste Version 3.4.2 aktualisiert, ohne dass Auswirkung auf die genutzten Funktionen festgestellt werden konnten.
Hadoop 2, die vorherige Hauptversion vom 31. Mai 2022 wird zum Zeitpunkt dieser Arbeit immer noch offiziell unterstützt und es ist keine weitere Hauptversion angekündigt [35]. Es kann also davon ausgegangen werden, dass die Hauptversion 3 noch mehrere Jahre unterstützt wird und somit keine Schnittstellen verändernde Änderungen auftreten.

Hive wurde zunächst mit der Version 2.3.2 in die Entwicklungsumgebung aufgenommen. Im Mai 2024 hat *Apache* die Version 2 als End-of-Life erklärt, im Oktober auch die Version 3 [36]. Daraufhin wurde die Version in der Entwicklungsumgebung auf die derzeit Version 4.1 aktualisiert. Bei dem Update sind einige Änderungen aufgefallen:

Unter anderem wird *MapReduce* als Engine die *Hive* standardmäßig zur Ausführung von Jobs nutzt durch *Tez* ersetzt. Das sollte zwar das Interface für Benutzer:innen nicht verändern, ist aber für die Dokumentation und die Lehrmaterialien relevant.
MapReduce ist durch die Verwendung im HDFS bereits bekannt. Dadurch ist es einfacher zu Verstehen wie *Hive* die Abfragen ausführt. Außerdem ist das Aufsetzen von *Hive* auf *MapReduce*

ein gutes Beispiel wie durch ein neues Programm die Funktion eines anderen erweitert wird ohne es verändern zu müssen. *Tez* ist zwar in vielen Fällen performanter [37], aber die Einfachheit und Bekanntheit von *MapReduce* ist in der Lehre von größerer Bedeutung.
Deswegen wird in der Entwicklungsumgebung weiterhin *MapReduce* für die Ausführung der *Hive* Abfragen verwendet, solange es unterstützt wird. Eine entsprechende Zeile wurde in die Konfigurationsdatei eingefügt.

Seit der Version 4.0 unterstützt *Hive* offiziell nur die Java Version 17. Mit Java 8 für *Hadoop* und Java 21 für *Kafka* muss damit die dritte Java Version im *Workspace* installiert werden.
Hier besteht noch Potenzial den *Workspace* zu vereinfachen. Durch zusätzliche Tests könnte gezeigt werden, dass zumindest die Anwendungen im *Workspace* mit einer gemeinsamen Java Version kompatibel sind, auch wenn in den Containern der Serverkomponenten weiterhin unterschiedliche Versionen installiert sind.

HBase ist in der aktuellsten stabilen Version 2.5.12 installiert.
Die nächste Hauptversion 3.0.0 ist noch in der Beta-Phase [38] und konnte bisher nicht erfolgreich in der Entwicklungsumgebung getestet werden. Log-Ausgaben des *Regionservers* in Listing 4.1 zeigen den Fehler in der Kommunikation der *HBase* Server.
Auf der offiziellen End-of-Life Website [39] gibt es keine Informationen darüber wie lange die Version 2.5 noch offiziell unterstützt wird.

```
1 INFO  regionserver.HRegionServer: reportForDuty to master=hbase-master,16000
2 ERROR regionserver.HRegionServer: Master passed us a different hostname to use;
  ↪  was=hbase-region, but now=hbase-region.db2-devenv-net
3 INFO  regionserver.HRegionServer: ***** STOPPING region server
  ↪  'hbase-region.db2-devenv-net' *****
```

Listing 4.1: Log Ausgabe mit Fehler bei HBase 3.0.0

Apache Kafka ist in der Version 4.1.0 installiert und zum Zeitpunkt dieser Thesis auf dem aktuellsten Stand.
Apache plant für *Kafka* drei Releases im Jahr und unterstützt im Moment die letzten 3 Versionen 3.9.1, 4.0.0 und 4.1.0 [40]. Um immer eine unterstützte Version zu verwenden, sollte daher mindestens einmal im Jahr die neueste Version installiert werden.
Im Verlauf dieses Projektes wurde *Kafka* von der Version 3.9.1 auf 4.1.0 aktualisiert. Dabei konnten, für die einfachen Beispielanwendung, die der Entwicklungsumgebung beiliegen, keine relevanten Änderungen mit festgestellt werden.

Der *Workspace* basiert auf Debian 13 und hat bei dem Update von Debian 12 auf 13 keine Anpassungen benötigt. Debian 13 wird für mindestens fünf Jahre unterstützt [41].
Die im *Workspace* installierten Java Versionen 8, 17 und 21 werden von *Temurin* mindestens bis 2027 unterstützt [42].
Die meisten Werkzeuge im *Workspace Container* sind langjährig erprobt und alle Werkzeuge

können bei endendem Support ausgetauscht werden. Damit ist der *Workspace* einer der stabilsten Komponenten in der Entwicklungsumgebung.

Zusammenfassend lässt sich sagen, dass alle verwendeten Komponenten in der aktuellsten Version verwendet werden und keine End-of-Life Termine, die große Änderungen mit sich bringen könnten, bekannt sind. Dennoch bringt eine Entwicklungsumgebung in der unterschiedliche Technologien kennengelernt werden sollen, zwangsläufig viele Abhängigkeiten mit sich.

4.2 Modularität

Wie in Kapitel 3.3 beschrieben sind die einzelnen Technologien in Modulen organisiert. Die Vorteile der Modularität der Entwicklungsumgebung lassen sich beispielhaft anhand des folgenden Prozesses zeigen, in dem das Modul *Neo4j* nachträglich in die Umgebung mit aufgenommen wird.

Neo4j ist ein Graphendatenbank-Management-System, das speziell für die Speicherung, Abfrage und Verwaltung von stark vernetzten Daten konzipiert ist.
Im Gegensatz zu relationalen Datenbanken organisiert *Neo4j* Daten nicht in Zeilen und Spalten, sondern in Knoten, Kanten und Eigenschaften [43]. Diese Struktur ermöglicht eine effiziente Abbildung von Beziehungen zwischen Entitäten, etwa in sozialen Netzwerken, Empfehlungssystemen oder komplexen Abhängigkeitsgraphen.
Für *Neo4j* wird ein offizielles Docker-Image angeboten [44], wodurch es sich sehr einfach in bestehende Container Infrastrukturen einbinden lässt.

Analog zu den bereits bestehenden Modulen wird für das neue Modul ein Skript geschrieben welches die in Kapitel 3.6 beschriebene Schnittstelle implementiert.
Bei einem einfachen Modul mit nur einem Server Container besteht dieses Skript fast nur aus dem in Listing 4.2 abgebildeten Start Befehl für diesen Container. Hier wird ein Volumen für die persistente Datenbank angegeben und der Port für die Weboberfläche an den Docker-Host durchgereicht.

```
1  start(){
2      if ! container_running neo4j then
3          echo "starting neo4j"
4          docker container run -d \
5              --name  neo4j \
6              --hostname neo4j
7              --network db2-devenv-net \
8              --volume neo4j-data \
9              --publish 7474:7474 \
10             neo4j \
11             > /dev/null
```

```
12      else
13          echo "neo4j already running"
14      fi
15  }
```

Listing 4.2: Ausschnitt aus Skript für die Steuerung des Moduls Neo4j

Durch die Port Freigabe von Docker ist die Weboberfläche von *Neo4j* auf dem Docker-Host unter `localhost:7474` abrufbar. Bei automatischer Port Weiterleitung der genutzten Virtualisierungssoftware kann der Port auch von einem Browser auf dem VM Host erreicht werden.

Für die Entwicklung von Programmen, die mit einer Java API auf die *Neo4j* Datenbank zugreifen, wird die Bibliothek *neo4j-java-driver* benötigt. Im Listing 4.3 ist der Ausschnitt des Dockerfiles für den *Workspace* zu sehen, in dem diese Bibliothek aus der Maven Repository heruntergeladen wird.

```
1  FROM debian:13 AS workspace-db2
2  # ...
3  RUN mkdir -p /opt/neo4j/lib
4  RUN curl -o /opt/neo4j/lib/neo4j-java-driver-6.0.1.jar \
5    https://repo1.maven.org/maven2/org/neo4j/driver/neo4j-java-driver/6.0.1/neo4j-java
     ↪ -driver-6.0.1.jar
6
```

Listing 4.3: Ausschnitt aus dem Dockerfile des Workspace Containers

Die Java API benötigt mindestens Java 17, ist aber auch mit Java 21 kompatibel, was bereits installiert ist. Für das Kompilieren und Ausführen der Java Programme im *Workspace* muss lediglich die Umgebungsvariable `JAVA_HOME` entsprechend gesetzt werden.

Ein minimales Beispielprogramm wird im Repository hinzugefügt unter in den *Workspace Container* eingebunden. Aus dem Container kann der *Neo4j* Server unter der URL `neo4j://neo4j:7687` erreicht werden.

Dieses Beispiel zeigt wie die modulare Architektur das Hinzufügen von neue Technologien in die Entwicklungsumgebung vereinfacht. Die notwendigen Schritte sind durch die Vorgaben an ein Modul vordefiniert:
Mit einem Dockerfile und einem Eintrag in das Build Skript wird das benötigte Image zur Entwicklungsumgebung hinzugefügt. Ein Skript zum Kontrollieren des Moduls enthält alle Informationen um aus dem Image einen oder mehrere Server zu erzeugen. Die benötigten Bibliotheken, Werkzeuge und Beispiele für die Entwicklung werden im *Workspace Container* installiert.

Die Serverkomponenten der verschiedenen Module sind durch die Implementation in eigenen Containern voneinander isoliert und beeinflussen sich nicht gegenseitig. Es konnte jedoch keine Möglichkeit gefunden werden diese Modularität auch für den *Workspace* Container umzusetzen. Alle Datenbankentechnologien sollten aus dem einen *Workspace* zu bedienen sein, um die Verbindung von mehreren Datenbankservern in einem Programm zu ermöglichen. Das führt dazu, dass für jedes Modul zusätzliche Bibliotheken und andere Software im *Workspace* installiert werden müssen, die einander beeinflussen könnten.
Zum Beispiel sind die Module nur mit unterschiedlichen Java Versionen kompatibel. Im *Workspace* müssen daher mehrere Versionen installiert sein und für jedes Projekt bewusst entschieden werden welche Version verwendet wird.

Zusammenfassend veranschaulicht die nachträgliche Aufnahme des Moduls *Neo4j*, dass durch die klare Trennung der einzelnen Module die Entwicklungsumgebung flexibel erweitert werden kann, ohne dass bestehende Komponenten angepasst werden müssen.
Durch das Abstrahieren von Funktionen auf eine Modulebene, können diese Funktionen, wie das Starten und Stoppen der Container, wiederverwendet werden. Es gibt zwar eine erzwungene Überschneidung der Module durch den geteilten *Workspace* aber die modulare Architektur und das Denken in expliziten Schnittstellen hilft Verflechtungen der Module zu verringern.

4.3 Build System

Durch die virtuelle Maschine mit Ubuntu als Basis ist die Entwicklungsumgebung unabhängig vom unterliegenden Betriebssystem. Für die Plattformunabhängigkeit muss zusätzlich die Kompatibilität mit den CPU-Architekturen x86_64 und ARM64 gewährleistet sein.
Dazu müssen alle verwendeten Docker-Images für diese Architekturen vorhanden sein und für Nutzer:innen bereitgestellt werden. Darüber hinaus sollten die beiden Versionen der Images in der Bedienung keine Unterschiede aufweisen, um die Lehre zu vereinfachen.
Die verwendeten offiziellen Images für *Zookeeper*, *Kafka* und *Neo4j* sind für diese Architekturen auf Docker Hub verfügbar. Für die eigens gebauten Docker Images kommt das Multi-Plattform Build Verfahren von Docker zum Einsatz.

Docker Images bestehen aus einem Manifest, welches auf die einzelnen *Layer* zeigt, die das Dateisystem des Images bilden. Ein Multi-Plattform-Image benötigt ein Manifest pro Plattform die unterstützt wird. Um diese zu Verwalten, gibt es eine übergeordnete Listenstruktur die auf die verschiedenen Manifeste verweist [45].
Die verschiedenen Versionen für die unterschiedlichen Architekturen können so in demselben Image in einer Registry gespeichert werden. Wenn Docker ein Multi-Plattform-Image mit `docker pull` herunterlädt, wird zunächst nur die Listenstruktur geladen. Anhand der Liste wird das passende Manifest ausgewählt und damit die benötigten Layer identifiziert. Die lokale Architektur wird dabei automatisch erkannt und muss nicht explizit angegeben werden.

Die Images für die Entwicklungsumgebung werden über eine Docker Registry und über *Tar*-Dateien auf einem Hochschulserver verteilt.
Für die Verteilung über einen Hochschulserver wird jedes Image separat für die jeweilige Architektur in eine *Tar*-Datei gepackt und in ein architekturspezifisches Verzeichnis abgelegt. Nutzer:innen laden anschließend alle *Tar*-Dateien aus dem Verzeichnis herunter, das ihrer eigenen CPU-Architektur entspricht, und importieren diese direkt in ihren Docker-Host.
Dieser Ansatz vermeidet zusätzliche Logik zur Plattformererkennung im Build- oder Bereitstellungsprozess und ermöglicht eine einfache, fehlerresistente und trotzdem ressourcensparsame Verteilung der Images.
Der Build Prozess muss also das Bauen von Multi-Plattform-Images unterstützen, die in eine Docker Registry geladen werden. Außerdem müssen diese Images pro Plattform als *Tar*-Datei exportiert werden.

Die Standardinstallation der Docker Engine unterstützt keine Multi-Plattform-Images, da der mitgelieferte *Image-Store* nicht mit ihnen kompatibel ist. Der *Image-Store* ist der interne Speicher, in dem Docker die mit `docker pull` geladenen oder mit `docker build` erzeugte Images speichert.
Die Dokumentation [45] zeigt zwei Möglichkeiten, um die Docker Umgebung für den Multi-Plattform-Build Prozess vorzubereiten.
Die erste Möglichkeit ist die Verwendung eines anderen Build-Treibers. Der Standard Build-Treiber *docker* arbeitet auf der Ebene der Docker Engine und lädt und speichert Images direkt im *Image-Store*. Durch diese Kopplung können keine Multi-Plattform-Images erzeugt werden. Der alternative Build-Treiber *docker-container* arbeitet in einem Container und hat keinen Zugriff auf den *Image-Store* der Docker Engine. Stattdessen werden benötigte und fertiggestellte Images direkt mit einer Docker Registry ausgetauscht.
Durch das Fehlen des internen *Image-Stores* in dem gebaute Images für weitere `FROM`-Anweisungen zwischengespeichert werden entsteht ein hoher Netzwerkverkehr. Für diese Variante sollte daher eine lokale Docker Registry aufgesetzt werden.

Zu Beachten ist auch, dass *Tar*-Dateien die beim Erstellen eines Multi-Plattform-Image exportiert werden, nur im OCI Format gespeichert werden können. Ein OCI Image enthält keine Metadaten, sondern nur das Dateisystem, mit dem ein Container erzeugt werden kann. Beim Importieren eines OCI Image muss daher der Image-Tag mit angegeben werden:
`docker image import oci-image.tar newtag:latest`.
Images im Docker Format sind dagegen ein Archiv eines kompletten Docker Images. Sie enthalten Informationen zu den einzelnen Layern, mit denen sie erzeugt wurden und Metadaten wie zum Beispiel den Tag. Daher können sie ohne weitere Angaben geladen werden:
`docker image load -i image.tar`.
Für eine effiziente Verteilung der Images in *Tar*-Format können die Images plattformspezifisch aus der genutzten Registry exportiert werden. Hierzu wird ein zusätzliches Werkzeug, wie z. B. *regctl* [46] benötigt.

Eine andere Möglichkeit ist es, den *Image-Store* auszutauschen und dann mit dem *docker*

Build-Treiber zu bauen.
Der *Containerd Image-Store* unterstützt Images mit einer Listenstruktur für Manifeste und kann somit auch Multi-Plattform-Images verwalten [47]. Während dieser Store in der Software *Docker Desktop* bereits der Standard ist, zählt er in der *Docker Engine* noch als experimentelles Feature. Mit der in Listing 4.4 aufgeführten Änderung in der Konfigurationsdatei in der Datei `/etc/docker/daemon.json` lässt sich der *Containerd Image-Store* aktivieren:

```
{
  "features": {
    "containerd-snapshotter": true
  }
}
```

Listing 4.4: Konfigurationsdatei /etc/docker/daemon.json

Nach einem Neustart des Docker Service mit `systemctl restart docker` kann mit dem Befehl in Listing 4.5 und dem Standard Build-Treiber ein Multi-Plattform-Image gebaut werden:

```
docker build \
    -t "newtag:latest" \
    --platform="linux/amd64,linux/arm64" \
    folder/with/dockerfile/
```

Listing 4.5: Multi-Plattform-Image mit docker build

Das Herunterladen aller benötigten Images aus einer Docker Registry verursacht 4.5 GB Traffic. Dieselben Images als *Tar*-Dateien sind 5.9 GB groß.
Die Differenz ergibt sich aus den Layern die in mehreren Images vorkommen. Aus der Registry werden diese nur einmal heruntergeladen, in den *Tar*-Dateien müssen sie dagegen in jeder Datei vorkommen und werden mehrfach heruntergeladen.

Die Images werden unabhängig von der CPU-Architektur mit denselben Dockerfiles gebaut. In den Images `hadoop-hadoop`, `hadoop-hive` und `hadoop-hbase` gibt es in den Dockerfiles keine Unterschiede zwischen den CPU-Architekturen, da hier lediglich Java-Programme und Shell Skripte installiert werden.
Das Basisimages `debian:13`, `apache/kafka`, `zookeeper` und `zookeeper` existieren für beide Architekturen und werden im Build Prozess automatisch für die korrekte Architektur heruntergeladen.
Da das JDK von Java 8, aufgrund des Alters, nicht mehr in den offiziellen Paketquellen für Debian verfügbar ist, werden die Binaries aus dem Projekt *Adoptium* [48] von Eclipse installiert. Hierfür werden, innerhalb des Dockerfiles, die zur CPU-Architektur passenden Binaries heruntergeladen und installiert. Wie in Listing 4.6 zu sehen ist, erfolgt die Auswahl der Version über

die Umgebungsvariable `TARGETPLATFORM`, welche automatisch von `docker build` gesetzt wird.

```
1 ARG TARGETPLATFORM
2 ARG BUILDPLATFORM
3 RUN echo "I am running on $BUILDPLATFORM, building for $TARGETPLATFORM"
4 RUN if [ $TARGETPLATFORM = linux/amd64 ]; then curl -Lo
  ↪ /usr/lib/jvm/jdk8/jdk8.tar.gz https://github.com/adoptium/temurin8-binaries/rele␣
  ↪ ases/download/jdk8u432-b06/OpenJDK8U-jdk_x64_linux_hotspot_8u432b06.tar.gz; fi
5 RUN if [ $TARGETPLATFORM = linux/arm64 ]; then curl -Lo
  ↪ /usr/lib/jvm/jdk8/jdk8.tar.gz https://github.com/adoptium/temurin8-binaries/rele␣
  ↪ ases/download/jdk8u432-b06/OpenJDK8U-jdk_aarch64_linux_hotspot_8u432b06.tar.gz;
  ↪ fi
6 RUN tar -xz --strip-components=1 -f jdk8.tar.gz
```

Listing 4.6: Terminal Output

4.4 Deklarative vs. Imperative Beschreibung der Infrastruktur

Die hier verwendete Methode die Containerinfrastruktur zu beschreiben kann als *imperativ* verstanden werden. In einem Bash-Skript werden die einzelnen Schritte beschrieben, mit denen die Anwendung gestartet wird.

Zu Beginn des Projektes wurden die Container und ihre Startvorgänge mit *Docker Compose* beschrieben. Das ist ein Werkzeug zum Definieren und Administrieren von Anwendungen mit mehreren Containern. Dabei werden die einzelnen Container und die Infrastruktur, die diese miteinander oder mit dem Host-System verbindet, in einem *Composefile* im YAML-Format beschrieben.
Dieser Ansatz ist *deklarativ*, da der gewünschte Endzustand der Infrastruktur beschrieben wird [49, S. 12-13]. *Docker Compose* bietet die Möglichkeiten das gesamte Projekt, oder einzelne Container, hoch- und runterzufahren und kümmert sich dabei automatisch um das Aufräumen von alten Containern und das Erstellen von Volumen und Netzwerken. *Compose* bietet auch eine einfache Möglichkeit Log-Dateien von allen Containern anzeigen zu lassen.
Durch diese Annehmlichkeiten und die übersichtliche Beschreibung aller benötigten Container schien es zunächst naheliegend, *Docker Compose* für die Containerverwaltung zu verwenden. An mehreren Stellen wurden allerdings Nachteile des deklarativen Ansatzes festgestellt.

Zum Beispiel kann die Startreihenfolge der Container nicht sinnvoll umgesetzt werden. *Compose* bietet mit dem Keyword depends_on die Möglichkeit den Start eines Containers von anderen Containern abhängig zu machen. Dazu kann ein *Healthcheck* definiert werden, der periodisch ausgeführt wird, um den Zustand des Containers zu überprüfen. Das Ergebnis dieser

Überprüfung kann als Startvoraussetzung für andere Container genutzt werden.
Für das fehlerfreie Starten der Anwendung war es notwendig auf unterschiedliche Zustände eines Containers zu warten.
Beim Starten des *Hadoop Distributed Filesystem* wird zunächst der *Namenode* gestartet. Wenn der *Namenode* bereit ist Verbindungen auf Port 9000 anzunehmen werden alle *Datanodes* gestartet. Erst wenn sich genügend Datenknoten bei dem *Namenode* angemeldet haben, kann das Cluster den schreibgeschützten Modus verlassen und die weiteren Container können gestartet werden.
Damit ist der Startvorgang von zwei unterschiedlichen Zuständen des *Namenodes* abhängig und kann nicht einfach und transparent mit *Docker Compose* dargestellt werden.
Ebenso ist das Starten von einer variablen Anzahl an Datenknoten nicht sinnvoll im *Composefile* abzubilden. In der iterativen Lösung können dagegen beliebig viele Datenknoten in einer einfachen Schleife gestartet werden.
Diesen Verlust der Kontrolle über einzelne Schritte im Deployment-Prozess beschrieben auch die Autor:innen von „Introduction to Infrastructure as Code“ [49, S. 13] als Nachteil eines deklarativen Ansatzes.

Außerdem lässt sich der modulare Aufbau der Entwicklungsumgebung nicht zufriedenstellend mit *Docker Compose* darstellen. Zwar können einzelne Compose-Dateien der Module mit dem Schlüsselwort *include* zu einem Projekt zusammengesetzt werden, allerdings kann dieses nur als ein Ganzes gesteuert werden. Das Starten und Stoppen einzelner Module wäre für Studierende, die nicht mit *Docker Compose* vertraut sind, nicht offensichtlich und sollte daher durch Wrapper Skripte vereinfacht werden.
Dieses Mischen der deklarativen und iterativen Beschreibung der Containerinfrastruktur bringt erhöhte Komplexität mit sich und erschwert das Verstehen der Entwicklungsumgebung.

Docker Compose ist eine Abstraktionsebene, die auf der Docker Engine aufsetzt und mit der versucht wird die genauen Schritte zum Anlegen und Starten von Containern zu verallgemeinern. Das kann in vielen Anwendungen zu mehr Übersichtlichkeit und weniger Boilerplate-Code führen. In einigen Fällen kann ein gewünschter Prozess nicht durch die Abstraktion abgebildet werden und muss durch ein Durchbrechen der Abstraktionsschicht gelöst werden.
Joel Sporsky, einer der Gründer von *StackOverflow*, beschreibt dieses Phänomen in dem Artikel *The Law of Leaky Abstractions (2002)* [50] und argumentiert, dass es zur Beherrschung einer Abstraktionsebene unabdingbar sei die darunter liegenden Konzepte zu verstehen.

Diese Probleme haben zu der Entscheidung geführt, für die Beschreibung der Container Infrastruktur auf *Docker Compose* zu verzichten und stattdessen die genauen Schritte zum Starten und Stoppen der Entwicklungsumgebung in Bash-Skripten zu implementieren.

5 Resümee und Ausblick

Die vorliegende Bachelorthesis widmet sich der Konzeption und Umsetzung einer plattformunabhängigen, containerisierten Entwicklungsumgebung für die Lehre, die am Beispiel verteilter Datenbanksysteme evaluiert wurde. Ausgangspunkt der Arbeit war die Beobachtung, dass bestehende Lösungen aufgrund proprietärer Lizenzen, hoher Komplexität und eingeschränkter Flexibilität nur bedingt für den Einsatz in der Hochschullehre geeignet sind. Vor diesem Hintergrund wurde eine offene, modulare und nachhaltige Alternative entwickelt, die Studierenden den Zugang zu Technologien wie *Hadoop HDFS*, *Hive*, *HBase* und *Kafka* ermöglicht, ohne dabei auf spezifische Hardware oder Betriebssysteme angewiesen zu sein.

Ein zentrales Ergebnis der Arbeit ist die erfolgreiche Kombination von Virtualisierung und Containerisierung, um eine einheitliche Entwicklungsumgebung auf heterogener Hardware bereitzustellen. Durch den Einsatz einer virtuellen Maschine als Docker-Host wird eine gemeinsame Basis geschaffen, auf der Docker-Container mit den benötigten Technologien betrieben werden können. Diese Architektur ermöglicht es, die Entwicklungsumgebung auf verschiedenen Betriebssystemen, wie Linux, macOS und Windows zu nutzen. Durch das Erstellen von Multi-Plattform-Images für Docker können auch unterschiedlichen CPU-Architekturen, wie zum Beispiel x86_64 und ARM64, unterstützt werden.

Ein weiterer Schwerpunkt der Arbeit lag auf der Transparenz und Wartbarkeit der Entwicklungsumgebung durch eine modulare Architektur. Diese Modularität findet sich zum einen in den einzelnen Werkzeugen wieder. Inspiriert von der Unix-Philosophie sind Komponenten wie Editoren, Compiler, Datenbanken und Shell-Werkzeuge orthogonal konzipiert und über klare Schnittstellen miteinander verbunden. Zudem wurden die Datenbanksysteme, die in der Lehre verwendet werden sollen, in Modulen organisiert. Diese Herangehensweise ermöglicht die Wartung, das Austauschen und das Hinzufügen von einzelnen Modulen und Werkzeugen, ohne die gesamte Entwicklungsumgebung zu beeinflussen. Die Verwendung von Open-Source-Software und die Reduzierung externer Abhängigkeiten tragen zusätzlich dazu bei, den Wartungsaufwand gering zu halten und die digitale Souveränität der Hochschule und der Studierenden zu stärken.

Die praktische Umsetzung der Entwicklungsumgebung erfolgt durch die Erstellung von Docker-Containern für die einzelnen Technologien sowie die Bereitstellung eines Workspace-Containers, der alle notwendigen Werkzeuge für die Entwicklung und das Testen von Anwendungen enthält. Das Steuern der Umgebung erfolgt über Bash-Skripte, die das Starten und Stoppen der Module ermöglichen. Durch die Integration von Remote-Development-Funktionalitäten können Studierende ihre bevorzugten Editoren auf dem Hostsystem nutzen, während die eigentliche Entwicklungsumgebung in einem Linux Container läuft. Dies vereinfacht nicht nur die Bedienung, sondern ermöglicht auch eine nahtlose Integration in bestehende Arbeitsabläufe.

Für zukünftige Arbeiten bietet die entwickelte Umgebung eine stabile Grundlage, die um weitere Technologien, z. B. Spark, Flink oder Vektordatenbanken, erweitert werden könnte.

Auch die Erweiterung zu einem echten verteilten Cluster wäre ein vielversprechender nächster Schritt, um die Umgebung noch näher an reale Produktionsszenarien heranzuführen. Durch Technologien wie Docker Swarm oder Kubernetes, können dieselben Docker Images verwendet werden, um die Entwicklungsumgebung auf mehrere Server auszuweiten. Der Weg zu einem optimierten, leistungsfähigen Cluster dürfte viele Erkenntnisse zum Verarbeiten von großen Datenmengen in einem verteilten System bringen.
Zudem wäre eine automatisierte Bereitstellung durch Continuous Integration/Continuous Deployment (CI/CD) denkbar, um die Wartung weiter zu vereinfachen.
Die Entwicklungsumgebung wurde von Beginn an für die Lehre von verteilten Datenbanksystemen konzipiert. Es bleibt die Frage, wie gut sich das Konzept für andere Hochschulmodule anpassen ließe. Besonders interessant wäre die Anpassung für Module in denen Netzwerktechniken im Vordergrund stehen, da unter Verwendung des IPvlan Treibers das Einbinden von Containern in externe Netzwerke möglich wird [51].

Die Arbeit zeigt, dass durch den gezielten Einsatz von Virtualisierung, Containerisierung und einer modularen Architektur, eine zukunftsfähige Entwicklungsumgebung geschaffen werden kann, die sich leicht an veränderte Lehrinhalte oder neue Technologien anpassen lässt.
Gleichzeitig wird durch den Verzicht auf proprietäre Software und den Einsatz offener Standards die digitale Souveränität der Hochschule und der Studierenden gestärkt.
Abschließend lässt sich festhalten, dass die entwickelte Lösung nicht nur die technischen und didaktischen Anforderungen einer modernen Informatiklehre erfüllt, sondern auch ein Beitrag zur Nachhaltigkeit in der digitalen Bildung darstellt. Durch den Verzicht auf proprietäre Software und die Förderung offener Standards wird die Abhängigkeit von Technologiekonzernen reduziert, während gleichzeitig die Zugänglichkeit und Flexibilität der Lehre erhöht wird. Besonders hervorzuheben ist dabei die Ressourceneffizienz der Lösung, die auch auf leistungsschwächeren Geräten, wie sie Studierende häufig verwenden, stabil läuft. Dies trägt nicht nur zur ökologischen Nachhaltigkeit bei, indem die Lebensdauer älterer Hardware verlängert wird, sondern fördert auch die soziale Inklusion, da keine zusätzlichen Anschaffungen erforderlich sind.

Literaturverzeichnis

[1] E. D. Team. „Breaking Down The Numbers: How Much Data Does The World Create Daily in 2024?“ en, besucht am 10. Okt. 2025. Adresse: https://edgedelta.com/company/blog/how-much-data-is-created-per-day.

[2] N. Kumar. „Facebook Users Statistics (2025).“ en, besucht am 10. Okt. 2025. Adresse: https://www.demandsage.com/facebook-statistics/.

[3] N. Khan u. a., „Big Data: Survey, Technologies, Opportunities, and Challenges,“ *The Scientific World Journal*, Jg. 2014, Nr. 1, S. 712 826, 2014. DOI: https://doi.org/10.1155/2014/712826. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2014/712826. Adresse: https://onlinelibrary.wiley.com/doi/abs/10.1155/2014/712826.

[4] D. Borthakur. „Facebook has the world's largest Hadoop cluster!“ en, besucht am 10. Okt. 2025. Adresse: https://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html.

[5] Oracle. „Oracle Big Data Lite Virtual Machine.“ en, besucht am 10. Okt. 2025. Adresse: https://www.oracle.com/database/technologies/bigdatalite-v411.html.

[6] Oracle. „Oracle Big Data Lite Virtual Machine Agreement.“ en, besucht am 10. Okt. 2025. Adresse: https://www.oracle.com/downloads/licenses/bigdata-lite-virtual-machine-agreement.html.

[7] D. Jungmannn, *Virtuelle Maschinen für Anwendungssoftware-Kompatibilität und zukunftsorientierte Lehre*, Informatik in der DDR - Grundlagen und Anwendungen, 2008.

[8] V. Dakic, P. Mukhedkar, A. Vettathu und H. D. Chirammal, *Mastering KVM virtualization : design expert data center virtualization solutions with the power of linux KVM*, 2. ed. Birmingham: Packt Publishing, Limited, 2020, xiii, 663 Seiten : Illustrationen, ISBN: 9781838828714. Adresse: https://suche.suub.uni-bremen.de/peid=B188044838.

[9] R. McDougall und J. Anderson, „Virtualization performance: perspectives and challenges ahead,“ *SIGOPS Oper. Syst. Rev.*, Jg. 44, Nr. 4, S. 40–56, Dez. 2010, ISSN: 0163-5980. DOI: 10.1145/1899928.1899933. Adresse: https://doi.org/10.1145/1899928.1899933.

[10] C. Baun, M. Kappes, H.-N. Cocos und M. M. Koch, „Eine Plattform zur Erstellung und Verwendung komplexer virtualisierter IT-Strukturen in Lehre und Forschung mit Open Source Software,“ *Informatik Spektrum*, Jg. 47, Nr. 1, S. 38–45, Apr. 2024, ISSN: 1432-122X. DOI: 10.1007/s00287-024-01559-x. Adresse: https://doi.org/10.1007/s00287-024-01559-x.

[11] V. Danciu, T. Guggemos und D. Kranzlmüller, „Schichtung virtueller Maschinen zu Labor- und Lehrinfrastruktur," in *9. DFN-Forum Kommunikations-technologien*, Bonn: Gesellschaft für Informatik e.V., 2016, S. 35–44, ISBN: 978-3-88579-651-0.

[12] T. Hildred. „The History of Containers." en, besucht am 22. Sep. 2025. Adresse: https://www.redhat.com/en/blog/history-containers.

[13] A. Mouat, *Docker - Software entwickeln und deployen mit Containern*. dpunkt.verlag, 2016.

[14] H. Aljafer und G. Cantrell, „Learning and Teaching Undergraduate Introductory Programming Courses in Java – The Use of an IDE VS Command Line," in *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2020, S. 992–997. DOI: 10.1109/CSCI51800.2020.00184.

[15] E. S. Raymond. „The Art of Unix Programming. "Adresse: http://www.faqs.org/docs/artu/.

[16] VSCodium. „VSCodium." en, besucht am 8. Okt. 2025. Adresse: https://vscodium.com/.

[17] J. Pierre. „Open Remote - SSH." en, besucht am 8. Okt. 2025. Adresse: https://github.com/jeanp413/open-remote-ssh.

[18] Microsoft. „VS Code Remote Development." en, besucht am 8. Okt. 2025. Adresse: https://code.visualstudio.com/docs/remote/remote-overview.

[19] „Nachhaltige IT-Infrastrukturen - Ansätze für ressourcenbewusste Softwareentwicklung: SSH & Vim & VSCodium," *FifF Kommunikation 1/2025*, Nr. 1/2025, S. 18–24, 2025.

[20] GitHub. „What are GitHub Codespaces." en, besucht am 8. Okt. 2025. Adresse: https://docs.github.com/en/codespaces/about-codespaces/what-are-codespaces.

[21] JetBrains. „Remote development overview." en, besucht am 8. Okt. 2025. Adresse: https://www.jetbrains.com/help/idea/remote-development-overview.html?keymap=KDE.

[22] Microsoft. „Advanced settings configuration in WSL | Microsoft Learn." en, besucht am 10. Sep. 2025. Adresse: https://learn.microsoft.com/en-us/windows/wsl/wsl-config.

[23] Lima. „Port Forwarding | Lima." en, besucht am 10. Sep. 2025. Adresse: https://lima-vm.io/docs/config/port/.

[24] F. Fieber und M.-F. Wendland, *Basiswissen Abnahmetest*, 1. ed. Heidelberg: dpunkt.verlag, 2021, ISBN: 9783864908293.

[25] „IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, S. 1–84, 1990. DOI: 10.1109/IEEESTD.1990.101064.

[26] S. Prakash, F. Antony, D. A. Köhler und R. Liu, *Ökologische und ökonomische Aspekte beim Vergleich von Arbeitsplatzcomputern für den Einsatz in Behörden unter Einbeziehung des Nutzerverhaltens (Öko-APC)*. 06844 Dessau-Roßlau: Umweltbundesamt, 2016.

[27] K. Carillo und C. Okoli, „The open source movement: A revolution in software development," *Journal of Computer Information Systems*, Jg. 49, S. 1–9, Dez. 2008.

[28] Apache. „Apache HDFS Architecture." en, besucht am 9. Okt. 2025. Adresse: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[29] Apache. „Apache HDFS User Guide." en, besucht am 22. Mai 2025. Adresse: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html.

[30] J. Dean und S. Ghemawat, „MapReduce: simplified data processing on large clusters," *Commun. ACM*, Jg. 51, Nr. 1, S. 107–113, Jan. 2008, ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. Adresse: https://doi.org/10.1145/1327452.1327492.

[31] Apache. „Apache Hive." en, besucht am 14. Mai 2025. Adresse: https://hive.apache.org/.

[32] Apache. „Apache Hive." en, besucht am 9. Okt. 2025. Adresse: https://hive.apache.org/docs/latest/admin/adminmanual-metastore-administration/.

[33] Apache. „Apache HBase." en, besucht am 14. Mai 2025. Adresse: https://hbase.apache.org/.

[34] Apache. „Apache HBase Reference Guide." en, besucht am 9. Okt. 2025. Adresse: https://hbase.apache.org/book.html.

[35] Apache. „Apache Hadoop." en, besucht am 7. Okt. 2025. Adresse: https://hadoop.apache.org/releases.html.

[36] Apache. „Apache Hive Downloads." en, besucht am 7. Okt. 2025. Adresse: https://hive.apache.org/general/downloads.

[37] S. Ibtisum, S. M. A. Rahman und s. M. S. Hossain, „Comparative analysis of MapReduce and Apache Tez Performance in Multinode clusters with data compression," *World Journal of Advanced Research and Reviews*, Jg. 20, S. 519–526, Dez. 2023. DOI: 10.30574/wjarr.2023.20.3.2486.

[38] Apache. „Apache HBase." en, besucht am 10. Sep. 2025. Adresse: https://hbase.apache.org/downloads.html.

[39] Apache. „Apache HBase - endoflife.date." en, besucht am 19. Okt. 2025. Adresse: https://endoflife.date/hbase.

[40] Apache. „Apache Kafka Download." en, besucht am 10. Sep. 2025. Adresse: https://kafka.apache.org/downloads.

[41] Debian. „Debian 13 ”trixie”released.“ en, besucht am 13. Okt. 2025. Adresse: https://www.debian.org/News/2025/20250809.

[42] Adoptium. „Temurin Support.“ en, besucht am 13. Okt. 2025. Adresse: https://adoptium.net/support.

[43] Neo4j. „What is a graph database.“ en, besucht am 13. Okt. 2025. Adresse: https://neo4j.com/docs/getting-started/graph-database/.

[44] Neo4j. „Neo4j - Docker Hub.“ en, besucht am 13. Okt. 2025. Adresse: https://hub.docker.com/_/neo4j/.

[45] Docker. „Multi-platform | Docker Docs.“ en, besucht am 14. Okt. 2025. Adresse: https://docs.docker.com/build/building/multi-platform/.

[46] regclient. „Regclient.“ en, besucht am 14. Okt. 2025. Adresse: https://regclient.org/cli/regctl/.

[47] Docker. „Containerd Image Store.“ de, besucht am 19. Sep. 2025. Adresse: https://docs.docker.com/desktop/features/containerd/.

[48] E. Adoptium. „Temurin8 Binaries.“ en, besucht am 10. Okt. 2025. Adresse: https://github.com/adoptium/temurin8-binaries.

[49] S. Pandya und R. Guha Thakurta, „Introduction to Infrastructure as Code,“ in *Introduction to Infrastructure as Code: A Brief Guide to the Future of DevOps*. Berkeley, CA: Apress, 2022, S. 3–17, ISBN: 978-1-4842-8689-0. DOI: 10.1007/978-1-4842-8689-0_1. Adresse: https://doi.org/10.1007/978-1-4842-8689-0_1.

[50] J. Spolsky. „The Law of Leaky Abstractions.“ en, besucht am 8. Okt. 2025. Adresse: https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

[51] Docker. „IPvlan network driver.“ en, besucht am 8. Okt. 2025. Adresse: https://docs.docker.com/engine/network/drivers/ipvlan/.

Abbildungsverzeichnis

Listingverzeichnis

Anhang

I Git-Repository

Das Git-Repository, über das die Entwicklungsumgebung bereitgestellt wird, ist auf dem Gitlab Server der Hochschule gehostet und kann unter der URL https://gitlab.informatik.hs-bremerhaven.de/lsteffen/hadoop-aio-docker abgerufen werden.

II Messen des Downloads von Docker Images

Für das Herunterladen der *Tar*-Dateien vom Hochschulserver lässt sich die Menge der heruntergeladenen Daten anhand der Größe der *Tar*-Dateien erkennen. Der verursachte Netzwerkverkehr bei einem `docker pull` Befehl ist dagegebn nicht offensichtlich.
Um ihn zu ermitteln wurde mit dem Programm *nethogs* der Netzwerkverkehr des *Docker Daemons* während des Herunterladens gemessen. Die Ausgabe im Listing II.1 zeigt unter anderem alle gesendeten und empfangenen Daten.

```
1 PROGRAM                                                                 DEV
  ↪  SENT      RECEIVED
2 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock  wlp3s0
  ↪  64.894    4499.713 MB
```

Listing II.1: Nethogs Ausgabe nach docker pull

Selbstständigkeitserklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit habe ich mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegt.

Bremerhaven, den 29. Oktober 2025 Unterschrift: