HAW HAMBURG

BACHELOR THESIS
René Daniel Gyetvai

Effiziente Bereitstellungsprozesse in containerisierten Systemen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

René Daniel Gyetvai

Effiziente Bereitstellungsprozesse in containerisierten Systemen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Jan Oliver Sudeikat

Eingereicht am: 16. April 2024

René Daniel Gyetvai

Thema der Arbeit

Effiziente Bereitstellungsprozesse in containerisierten Systemen

Stichworte

CI/CD, Containerisierung, Deployment, Effizienz, Warteschlangentheorie

Kurzzusammenfassung

Trends im Bereich der künstlichen Intelligenz, die zunehmende Digitalisierung von Unternehmen wie auch die Transformation in die Cloud erfordern viele Ressourcen und führen zu unterschiedlichen Anforderungen an Computersysteme. Darüber hinaus steigt der Bedarf, Schritte zur Optimierung der Sicherheit von Software über den gesamten Entwicklungsprozess hinweg zu integrieren. Zur Bewältigung dieser Herausforderungen bietet die Automatisierung von Prozessen einige Lösungsansätze. Kombiniert mit modernen Konzepten wie CI/CD und der Containervirtualisierung können diese Ansätze zu effizienten Lösungen führen. Um mögliche Optimierungen zu identifizieren und eine effiziente Lösung darzustellen, erörtert diese Arbeit die Anwendung der Warteschlangentheorie und einiger damit verbundener Gesetzmäßigkeiten. Es wird anhand von Versuchen gezeigt, inwiefern sich vor allem die Gesetze Amdahls und Gustafsons auf die Architektur von CI/CD-Pipelines auswirken und welche Optimierungen durch sie möglich sind. Durch die Anpassung einer klassischen, seriellen Pipeline hin zu einer parallelen Pipeline sowie den Einsatz von Containern wird gezeigt, dass die Effizienz signifikant verbessert werden kann. Diese Verbesserung resultiert besonders aus der Verringerung der Durchlaufzeit und dem optimierten Einsatz von Ressourcen. Damit zeigen die Ergebnisse, dass die Optimierung der Architektur von CI/CD-Pipelines sinnvoll ist und zu einer effizienteren Ausführung beiträgt. Außerdem unterstreichen sie die Bedeutung der Warteschlangentheorie und der damit verbundenen Gesetzmäßigkeiten für die Optimierung von Prozessen innerhalb von CI/CD-Pipelines.

René Daniel Gyetvai

Title of Thesis

Efficient provisioning processes in containerized systems

Keywords

CI/CD, Containerization, Deployment, Efficiency, Queueing theory

Abstract

Trends in the field of artificial intelligence, the increasing digitalization of companies and the transformation to the cloud require many resources and lead to different requirements for computer systems. In addition, there is an increasing need to integrate steps to optimize the security of software throughout the entire development process. To overcome these challenges, the automation of processes offers some possible solutions. Combined with modern concepts such as CI/CD and container virtualization, these approaches can lead to efficient solutions. In order to identify possible optimizations and present an efficient solution, this thesis discusses the application of queueing theory and some associated laws. Experiments are used to show the extent to which Amdahl's and Gustafson's laws in particular affect the architecture of CI/CD pipelines and which optimizations are possible using them. By adapting a classic, serial pipeline to a parallel pipeline and using containers, it is shown that efficiency can be significantly improved. This improvement results in particular from the reduction in throughput time and the optimized use of resources. The results thus show that optimizing the architecture of CI/CD pipelines makes sense and contributes to more efficient execution. They also underline the importance of queueing theory and the associated laws for the optimization of processes within CI/CD pipelines.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In einer Zeit der großen Rechenzentren und eines steigenden Bedarfs an leistungsstarken Systemen stellt sich die Frage, wie Rechenleistungen und Auslastungen effizienter gestaltet werden können. [35] Dazu ergeben sich durch neu aufkommende Trends, wie im Bereich der künstlichen Intelligenz oder des Cloud-Computings, weiterhin steigende Anforderungen an die Entwicklung und Bereitstellung von Software. Diese Anforderungen umfassen allgemein eine stetige Anpassung von Business-Prozessen und Technologien sowie konkretere Systemanforderungen an unter anderem Flexibilität, Modularität, Integrierbarkeit und Sicherheit von Computersystemen. [27] Darüber hinaus spielt auch die Skalierbarkeit der Systeme eine wichtige Rolle, um der steigenden Anzahl an Nutzern und Anwendungen gerecht zu werden.

Zur Bewältigung der Systemanforderungen und der sich daraus ergebenden Problemstellungen wurden in der Vergangenheit viele neue Technologien und Praktiken erschlossen. So wurde unter anderem die Praktik der kontinuierlichen Integration (fortlaufend: engl. Continuous Integration, kurz: CI) und kontinuierlichen Auslieferung (fortlaufend: engl. Continuous Delivery, kurz: CD) etabliert, um wesentliche Lücken zwischen den Prozessen der Softwareentwicklung und Softwarebereitstellung zu schließen und diese Prozesse teils zu automatisieren. [1] Die Skalierungsanforderungen können hingegen durch den Einsatz moderner Technologien wie der Containervirtualisierung und dazugehöriger Orchestrierungswerkzeuge bewältigt werden.

Wie diese Praktiken genutzt werden können, um ein gutes Zusammenspiel von Effizienz und Sicherheit zu erreichen, ist jedoch noch nicht ausreichend untersucht worden. So gibt es im Softwareentwicklungsmodell DevOps bzw. DevSecOps, zusammengesetzt aus den englischen Begriffen „Development“, „Operation“ und „Security“, welche Ansätze bieten, um CI/CD-Prozesse sicherer und effizienter zu gestalten. Jedoch wird hierbei in vielen wissenschaftlichen Arbeiten entweder besonders der Sicherheitsgrad jeweiliger Maßnahmen betrachtet [18] oder aber, wie sich die Performanz und Effizienz durch Optimierungen an zugrunde liegenden Technologien, wie der Ausführungsumgebung der

Container, verbessern lassen [32]. Aktuellere Arbeiten, wie die von Morales et al. [18] zeigen dies sehr deutlich. Im letzten Teil ihres Fazits betonen die Autoren der Arbeit, dass zukünftige Forschungen sich weitergehend mit den realen und potenziellen Folgen suboptimaler Implementierungen von DevSecOps-Methoden befassen sollten. Dies folgerten sie insbesondere aus ihren eigenen Ergebnissen, die bereits die weitreichenden Auswirkungen auf die Sicherheit alleine, bei solch einer suboptimalen Implementierung, aufzeigen. Ebenso zeigt die Arbeit von Bezemer et al. [4] auf, wie wichtig die Entwicklung von Praktiken und Tools zur Verbesserung der Performanz von CI/CD-Pipelines allgemein ist. Das Fazit dieser Arbeit betont außerdem, dass weitere Forschung notwendig ist, um bestehende Praktiken des Performance-Engineerings in die Prozesse von CI/CD-Pipelines zu integrieren.

Vor diesem Hintergrund soll im Rahmen dieser Arbeit die effiziente Gestaltung von Bereitstellungsprozessen containerisierter Software, im Kontext aktueller CI/CD-Konzepte, durch gezielte Optimierung und Parallelisierung anhand theoretischer Ansätze der Warteschlangentheorie untersucht werden.

Mithilfe von drei beispielhaften Versuchsaufbauten soll dazu in Experimenten analysiert werden, wie sich das Verhalten des Bereitstellungsprozesses einer Software hinsichtlich der Laufzeit- und Speichereffizienz ändert, wenn insbesondere Prozessschritte für die Sicherheit der Software optimiert werden. Im ersten Versuchsaufbau wird eine einfache CI/CD-Pipeline modelliert, welche mithilfe eines Benchmark-Verfahrens das Gesetz von Amdahl und dessen Auswirkung im CI/CD-Kontext untersuchen soll. Im zweiten Versuchsaufbau wird ebenfalls eine einfache CI/CD-Pipeline modelliert, jedoch soll hier das Gesetz von Gustafson mithilfe des Benchmark-Verfahrens untersucht werden. Zuletzt wird im dritten Versuchsaufbau versucht, ein realistisches Softwareprojekt anhand der zuvor erprobten Gesetzmäßigkeiten zu optimieren. Dazu werden realistische Lasten erzeugt und die Auswirkungen der Optimierungen auf die Laufzeit- und Speichereffizienz untersucht. So werden die zunächst konzeptuellen Modelle aus dem ersten und zweiten Versuch im darauffolgenden dritten Versuch auf ihre Anwendbarkeit in der Praxis überprüft.

Um einen allgemeinen Überblick über das Thema zu geben, soll in Kapitel 2.1 und 2.2 zunächst tiefer auf die grundlegende Motivation und die sich ergebenden Anforderungen eingegangen werden. Anschließend soll in den Kapiteln 2.3, 2.4 und 2.5 ein Verständnis der relevanten theoretischen Grundlagen erschlossen werden. Darauf folgend werden dann in Kapitel 3 die verwendeten Technologien weiter betrachtet. Diese spiegeln

die konkrete Implementation der zuvor erläuterten Konzepte wider und werden für die Durchführung des Experiments genutzt. Außerdem sollen die theoretischen Grundlagen den Leser darauf vorbereiten, die Lösungsstrategie zur Verbesserung des Bereitstellungsprozesses im Experiment besser nachvollziehen zu können. Im darauf folgenden Kapitel 4 der Arbeit wird dann das Experiment als Teil der Evaluation ausführlicher erläutert. Zusätzlich wird in diesem Teil auf gemessene Daten und deren Auswertung eingegangen und die Überlegung angestrengt, wie sich die durchgeführten Experimente erweitern lassen würden. Zuletzt wird im Fazit in Kapitel 5 erläutert, welche Erkenntnisse aus der Arbeit erlangt wurden und wie die Ergebnisse eingeordnet werden können. Darauf folgend wird abschließend eine kritische Auseinandersetzung unternommen, um aufzuzeigen, inwiefern die Erkenntnisse aus den Experimenten in der Praxis anwendbar sind.

2 Theorie

Das folgende Kapitel soll einen Überblick über die theoretischen Grundlagen der Arbeit geben. Es soll dazu dienen Konzepte und Theorien zu erläutern, die zur Entwicklung einer effizienten Lösung relevant sind. Darüber hinaus wird vorab noch einmal erläutert, welche Motivationen und Anforderungen für die Entwicklung einer solchen Lösung bestehen.

2.1 Motivation

Als besondere Motivationen der Arbeit lassen sich im Wesentlichen drei Faktoren für die Ausarbeitung einer geeigneten Lösung benennen. Zunächst soll durch die Verbesserung des CI-Prozesses eine geringere Durchlaufzeit für die Ausführung einer definierten Menge an Tests und Sicherheitsscans erreicht werden. Dadurch kann folglich die Entwicklung neuer Versionen beschleunigt werden.

Darüber hinaus soll der Prozess effizienter gestaltet werden, indem die jeweiligen Tests und Sicherheitsscans eine bessere Auslastung der zur Verfügung gestellten Ressourcen erzielen. So soll zum einen sichergestellt werden, dass Rechenleistung nicht über die benötigten Anforderungen der Prozesse hinaus vergeben wird und dadurch ungewollt Ressourcen blockiert werden. Zum anderen sollen Prozesse möglichst wenig Zeit mit geringer Auslastung verbringen.

Übergeordnet wird die Entwicklung eines möglichst universal einsetzbaren Konzeptes angestrebt. Entsprechend sollen die Erkenntnisse der Arbeit auf unterschiedliche Systeme anwendbar sein.

2.2 Anforderungen

Um eine geeignete Lösung entwickeln zu können, ist es von zentraler Bedeutung, die Anforderungen an das System zu kennen. Im Wesentlichen kann hier zwischen zwei unterschiedlichen Arten von Anforderungen unterschieden werden. Zum einen können die nichtfunktionalen Anforderungen betrachtet werden. Konkret handelt es sich hierbei um Anforderungen, die beispielsweise die Leistungsfähigkeit, Sicherheit und Skalierbarkeit des Systems betreffen. Zum anderen können die funktionalen Anforderungen untersucht werden. Dies sind Anforderungen an ein System, die direkt mit seiner Funktionalität in Verbindung stehen. Konkret handelt es sich hierbei um Anforderungen an die Funktionalität des Systems oder die Interaktion mit dem Benutzer. Zur einfachen Übersicht und Strukturierung werden im Folgenden die Anforderungen in einer tabellarischen Form dargestellt. Die genannten Use-Cases werden im Anhang A.1 näher erläutert.

Nichtfunktional	Funktional	Use-Case
Skalierbarkeit	Anpassbarkeit durch Konfigurationsdateien	UC1
Performanz	Parallelität von Ausführungsabschnitten	UC2
Verfügbarkeit	Automatisches Ausführen von Builds	UC3
Fehlertoleranz	Erkennen von fehlerhaften Zuständen	UC4
Sicherheit	Automatische Tests und Sicherheitsscans	UC5

Tabelle 2.1: Anforderungen an das System

2.3 Technische Konzepte und Methoden

Um die in dieser Arbeit behandelten Themenfelder besser zu verstehen und einordnen zu können, bedarf es einiger Grundlagen. Diese Grundlagen spalten sich zum einen in technische Konzepte und zum anderen in theoretische Methoden und Gesetzmäßigkeiten auf. Im Folgenden wird ein Überblick über alle relevanten Themen gegeben und ein Bezug zu der späteren Anwendung im Rahmen des Versuchs hergestellt.

2.3.1 DevOps

Der Begriff *DevOps*, erstmalig 2009 während einer Konferenz in San Jose verwendet und später durch Patrick Debois geprägt [10], spielt in der heutigen IT-Welt eine wichtige Rolle und ist in vielen Unternehmen mittlerweile ein etablierter Bestandteil bei der Bereit-

stellung von Software. Inspiriert durch das weiterentwickelte Toyota Production System (TPS) von T. Ohno und die sich darauf beziehenden Konzepte des Lean Management [10] sowie andere Ansätze des Software-Engineerings [12] hat sich DevOps als eine Kombination aus den Begriffen *Development* und *Operations* etabliert. In diesem Zusammenhang beschreibt DevOps eine Kultur der Automatisierung wesentlicher Softwareentwicklungs- und Betriebsprozesse, um Lösungen kontinuierlich zu verbessern und zu optimieren. Es handelt sich dabei nicht um eine Technologie oder Software-Werkzeuge, sondern um eine Kultur, die auf Messbarkeit, schnellen Mehrwert und Transparenz in der Softwarebereitstellung abzielt [10][12]. Konkret bedeutet dies, dass Ergebnisse, die durch eine Art „Entwicklungs-Fließband“ erzeugt werden, stets durch einfache Metriken und Kennzahlen überwacht werden können. Außerdem wird der Mehrwert der Software möglichst schnell und effizient an den Kunden ausgeliefert, wobei durch kleinere Bereitstellungszyklen die Qualität der Software stetig verbessert wird [10][12]. Ausgehend von diesen Faktoren ist ein Großteil dieser Arbeit im Kontext von DevOps angesiedelt und beschäftigt sich eingehend mit konkreten Ansätzen, wie beispielsweise Continuous Integration (CI), um die Bereitstellungsprozesse von Software zu verbessern.

2.3.2 Continuous Integration, Delivery und Deployment

Continuous Integration, Delivery und Deployment sind Praktiken, welche mittlerweile in vielen Unternehmen etabliert sind. Besonders im Zusammenhang mit der Cloud werden sie oftmals verwendet, um die Bereitstellung von Software zu automatisieren und zu beschleunigen. Um jedoch zu verstehen, welchen Nutzen diese Praktiken haben, sollte zunächst eine Herleitung der Begriffe erfolgen.

Bei der Continuous Integration bezieht sich der Integrationsbegriff auf die Systemintegration, welche wesentliche Prozessschritte zum Zusammenfügen funktionsfähiger Systeme vereint [28]. Hierzu gehören beispielsweise das Installieren und Konfigurieren von Software, das Laden von Testdaten, das Kompilieren von Quellcode oder auch das Ausführen von Tests [28] (Abb.: 2.1).

Die Continuous Delivery bezieht sich hingegen auf die Auslieferung und das Continuous Deployment auf die Bereitstellung von Software. Hierbei muss betont werden, dass Continuous Delivery und Deployment ähnlich sind und je nach Quelle auch Überschneidungen haben.

Mit Blick auf andere Arbeiten findet man ähnliche Definitionen, wie von Fitzgerald et al. [7] und Weber et al. [34]. In diesen Arbeiten wird die Auslieferung (Delivery) von Software meist als der Prozess beschrieben, der auf die Continuous Integration folgt. Dieser Prozess dient der Sicherstellung eines guten, automatisierten Software-Builds für eine Umgebung, ist jedoch nicht unbedingt für den Endnutzer bestimmt. Insbesondere spielt auch der Gedanke eine Rolle, dass die Software zur Bereitstellung einer produktiven Umgebung andere Anforderungen erfüllen muss, als es in der im CI-Prozess bestehenden Entwicklungsumgebung der Fall ist. So muss das System nach Sommerville et al. [28] für die spätere Bereitstellung (vgl. Deployment [34][7]) bei einem Kunden angepasst werden und gegebenenfalls Tests wie beispielsweise Lasttests durchlaufen. Fitzgerald et al. [7] definieren hier hingegen, dass das Continuous Deployment die Aspekte der Continuous Delivery impliziert und generell für die kontinuierliche Bereitschaft zur Veröffentlichung der Software und zur Bereitstellung beim Kunden verantwortlich ist. Einen Überblick über die Zusammenhänge von Continuous Integration, Delivery und Deployment gibt Abbildung 2.1.

Abbildung 2.1: Schema – Continuous Integration, Delivery und Deployment [15]

Ordnet man beide Begriffe nun in den gesamten Entwicklungsprozess ein, so wird deutlich, dass die Continuous Integration besonders in der Entwicklungsphase eine Rolle spielt. Um hier eine Automatisierung zu erreichen, wird typischerweise Quellcode in Verzeichnissen in einem Versionsverwaltungssystem abgelegt. Bei Änderungen wird der entsprechende Integrationsserver benachrichtigt, welcher dann den Buildprozess auslöst. Dieser sorgt dafür, dass aus den Quelldateien eine funktionierende Software integriert wird. Dieser Buildprozess sowie die Ausführung von Tests werden durch eine Integrationspipeline repräsentiert [28]. Die Continuous Delivery hingegen spielt besonders in der Bereitstellungsphase eine Rolle. Hierbei wird die Software automatisiert getestet und eine Bereitschaft für das Deployment hergestellt [34][7]. In der Praxis heißt das, dass die Software bereitstellbar zum Beispiel in Form eines Containerabbilds vorliegt. Im Prozessabschnitt des Continuous Deployment wird dann sichergestellt, dass das Abbild der Software kontinuierlich beim Kunden bereitgestellt werden kann [34][7].

Um eine CI/CD-Pipeline möglichst effektiv zu gestalten, sollten grundlegende Faktoren berücksichtigt werden. So ist es beispielsweise wichtig, dass innerhalb des CI/CD-Prozesses der Buildprozess nicht immer wieder von vorne beginnt, sondern nur diejenigen Teile des Prozesses ausgeführt werden, die sich durch Änderungen im Quellcode ergeben. Allgemein kann also gesagt werden, dass mögliche Abhängigkeiten zwischen den einzelnen Prozessschritten berücksichtigt werden sollten, damit langsame Prozessabschnitte nicht immer wieder ausgeführt werden müssen [28]. Speziell die Einordnung in schnellere und langsamere Prozessabschnitte spielt im Rahmen dieser Arbeit eine wesentliche Rolle, um Verbesserungspotenziale zu identifizieren, wie im Folgenden noch genauer erläutert wird. Darüber hinaus konzentriert sich die Arbeit insbesondere auf den Teil der Continuous Integration und Delivery, da insbesondere hier interessante Ansätze zur Optimierung gefunden werden können.

2.3.3 Containervirtualisierung

Die in dieser Arbeit angestrebte Optimierung wird in einem Umfeld durchgeführt, in dem die Containervirtualisierung eine wesentliche Rolle spielt. Containervirtualisierung ist ein Konzept, bei dem Anwendungen in sogenannten Containern ausgeführt werden. Diese Container stellen dabei nach Tanenbaum [31] eine Sammlung von Binärdateien dar. Im gemeinsamen Kontext bieten sie die Softwareumgebung, die eine Anwendung benötigt, um ausgeführt werden zu können. Zudem bedient sich die Containervirtualisierung im Allgemeinen der Ressourcen ihres Host-Betriebssystems. Beispielsweise erfolgt dies in einem UNIX-System, indem sie auf Namespaces, das Dateisystem (UnionFS) und Kontrollgruppen (cgroups) zurückgreift und diese zur Gruppierung oder Isolation von Prozessen nutzt [31]. Vergleicht man die Containervirtualisierung mit anderen Virtualisierungstechniken, so können einige Unterschiede festgestellt werden. Um einen konkreteren Einblick in diese Unterschiede zu erhalten, kann eine systematische Studie wie von Sharma et al. [26] herangezogen werden, bei der Linux-Container (LXC) mit Linux-basierten virtuellen Maschinen (KVM) verglichen wurden. Diese Studie zeigt, dass Container in einem einfachen Vergleich, in dem eine Anwendung auf beiden Plattformen ausgeführt wurde, eine geringfügig [26][23] bessere Leistung aufweisen. Insbesondere Eingabe- und Ausgabevorgänge (I/O) waren hierbei signifikant schneller innerhalb der Container. Dies ist darauf zurückzuführen, dass Container im Gegensatz zu virtuellen Maschinen keine zusätzliche Betriebssystemebene benötigen, um Anwendungen auszuführen. Der Grund liegt darin, dass sie über die zuvor erläuterten Schnittstellen direkt

auf dem Host-Betriebssystem aufsetzen können.
Um den Aufbau eines Containers zu verdeutlichen, kann im Folgenden die Abbildung 2.2 betrachtet werden.

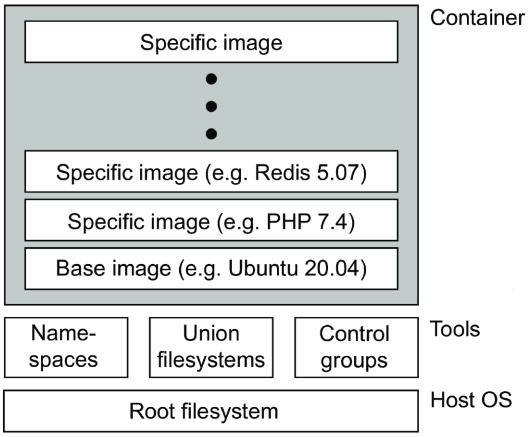

Abbildung 2.2: Container-Organisation innerhalb einer Hostumgebung [31]

Besonders interessant ist die Containervirtualisierung im Kontext der Arbeit aber auch deshalb, da sie Eigenschaften bietet, die sich in Kombination mit den zuvor beschriebenen CI/CD-Prozessen anwenden lassen. So können die leichtgewichtigen und schnell zu startenden Container dazu genutzt werden, bestimmte Abläufe innerhalb der CI/CD-Pipeline zu kapseln. So ist es denkbar, unter anderem Build- und Testumgebungen in Containern zu betreiben, um diese Prozessabschnitte wiederverwendbar und unabhängig von einem spezifischen Hostsystem zu machen. Darüber hinaus können je nach Containerumgebung weitere Vorteile wie beispielsweise die Skalierbarkeit von Containern genutzt werden, was im Weiteren noch vertieft wird.

2.3.4 Prozesse und Threads

Prozesse und Threads gehören zu den wesentlichen Bestandteilen eines Betriebssystems und sind von besonderer Bedeutung, wenn es um die Ausführung von Software geht. Um einen Einblick in die Funktionsweisen zu erhalten und den Zusammenhang mit der späteren Anwendung in CI/CD-Pipelines herzustellen, werden im Folgenden einige Grundlagen und Eigenschaften erläutert.

Funktionsweise in Betriebssystemen

Diese Arbeit untersucht Prozesse und Threads in einer verteilten Umgebung. Prozesse und Threads sind dabei zwei grundlegende Konzepte, die in einem Betriebssystem zur Ausführung von Software genutzt werden. Ein Prozess ist nach Tanenbaum [30] ein Programm in Ausführung, versehen mit einem eigenen Adressraum, welcher das ausführbare Programm, Programmdaten sowie einen Stack beinhaltet. Darüber hinaus verfügt jeder Prozess über einen Prozesstabelleneintrag. Wichtig ist dabei, dass ein Prozess unabhängig von anderen Prozessen existiert und somit keine Daten oder Ressourcen mit anderen Prozessen teilt. So beschreibt Tanenbaum beispielhaft, dass ein und dasselbe Programm zweifach ausgeführt werden kann, jedoch jede Ausführung einen eigenen, unabhängigen Prozess erzeugt [30]. Laufen mehrere Prozesse gleichzeitig auf einem System, so wird die Rechenzeit des Prozessors auf diese Prozesse aufgeteilt. Dies führt dazu, dass eine Quasiparallelität entsteht. Der diesem zugrunde liegende Schaltmechanismus des Prozessors wird als Multiprogrammierung bezeichnet, wobei mithilfe eines Zeitscheibenverfahrens und entsprechender Befehlszähler die Prozesse abwechselnd ausgeführt werden können [30]. Neben den Prozessen gibt es auch Threads, welche als kleinere Einheiten innerhalb eines Prozesses betrachtet werden können. Threads werden nach Tanenbaum [30] als quasiparallele Ausführungsfäden innerhalb desselben Adressraumes eines Prozesses beschrieben. Darüber hinaus hält jeder Thread einen eigenen Stack und einen eigenen Zustand, wobei mehrere Threads innerhalb eines Prozesses auf die gemeinsamen Ressourcen des überliegenden Prozesses zugreifen können. Dies ist insbesondere dann von Vorteil, wenn mehrere Threads innerhalb eines Prozesses gleichzeitig auf beispielsweise eine Datei zugreifen müssen. Dies liegt darin begründet, dass es die Kommunikation und Synchronisation vereinfachen kann [30]. Bei aller Funktionalität, die diese Konzepte mit sich bringen, sollte jedoch auch bedacht werden, dass zusätzlicher Synchronisationsaufwand entstehen kann, wenn mehrere Threads auf gemeinsame Ressourcen zuzugreifen versuchen. Dies kann dazu führen, dass die Ausführungsgeschwindigkeit des Programms sinkt, da die Threads, mithilfe von Synchronisationsmethoden wie beispielsweise Mutexen, aufeinander warten müssen [30].

Anknüpfend an die zuvor erläuterte Funktionsweise lässt sich die Bedeutung für die Effizienzsteigerung in CI/CD-Pipelines und damit auch das Experiment der Arbeit spezifizieren. In CI/CD-Umgebungen ermöglicht die intelligente Verteilung von Prozessen auf unterschiedliche Rechenknoten nicht nur eine optimierte Ausnutzung der verfügbaren Hardware-Ressourcen, sondern auch eine signifikante Beschleunigung der Durchlaufzei-

ten. Prozesse, die isoliert voneinander auf verschiedenen Knoten laufen, können unabhängige Teile der Pipeline parallel abarbeiten. Dies reduziert die Wartezeit zwischen den Schritten und erhöht die Gesamtperformance der Pipeline. [6]

Threads innerhalb dieser Prozesse eröffnen zusätzlich die Möglichkeit, feingranulare parallele Aufgaben innerhalb der Prozesse durchzuführen. So können beispielsweise Softwaretests parallelisiert werden. Der strategische Einsatz von Multi-Threading in Prozessen, die über die CI/CD-Pipeline verteilt sind, optimiert demnach die Ausnutzung der CPU-Ressourcen und minimiert die Ausführungszeit.

Durch diese Parallelisierungsansätze werden die in der Pipeline definierten Aufgaben, wie Builds oder Tests, nicht nur schneller, sondern auch flexibler hinsichtlich der Lastverteilung auf die Infrastruktur durchgeführt.

Mehrkernprozessoren und Multicomputer

Um zu verstehen, warum die parallele Ausführung von Prozessen und Threads in CI/CD-Pipelines besonders sinnvoll ist, sollte auch das Konzept der Mehrkernprozessoren und Multicomputer kurz beleuchtet werden (Abbildung 2.3). Bei den Mehrkernprozessoren wird die Rechenleistung mehrerer Rechenkerne genutzt, um Prozesse parallel auszuführen. Anders als bei Systemen mit einem einzigen Kern ist die Ausführungsplanung bei Mehrkernprozessoren wesentlich komplexer und bedarf unterschiedlicher Scheduling-Strategien, um eine effiziente Verteilung der einzelnen Prozesse zu erreichen [30]. Das im Rahmen dieser Arbeit durchgeführte Experiment basiert wesentlich auf der Nutzung von Mehrkernprozessoren und verwendet mehrere Rechenknoten mit virtualisierten Mehrkernprozessoren, um eine Art Multicomputer oder auch verteiltes System zu bilden.

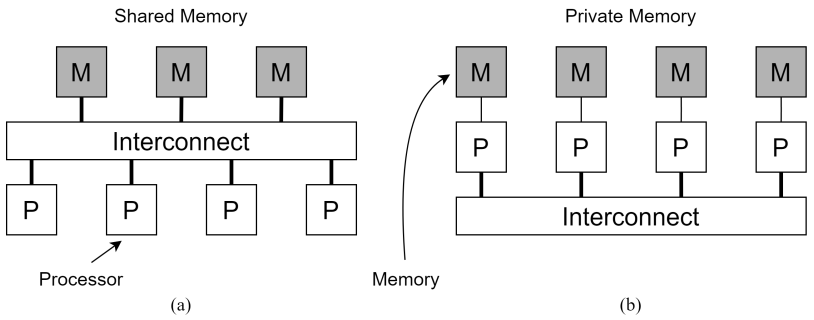

Abbildung 2.3: Vergleich – (a) Mehrkernprozessor und (b) Multicomputer nach [31]

2.3.5 Skalierbarkeit

Die Skalierbarkeit von CI/CD-Pipelines ist auch ein entscheidender Faktor, um die Effizienz und Leistungsfähigkeit bei der Bereitstellung von Software zu steigern. In diesem Abschnitt werden die Konzepte der vertikalen und horizontalen Skalierung vorgestellt und deren Anwendbarkeit auf CI/CD-Pipelines wird erläutert. Dabei liegt der Fokus auf den Vorteilen, die die jeweiligen Skalierungsstrategien bieten. Im Gegenzug werden darüber hinaus potenzielle Herausforderungen betrachtet, die bei der Implementierung und Anwendung dieser Strategien auftreten können.

Vertikale Skalierung

Bei der vertikalen Skalierung handelt es sich um ein Prinzip, bei dem die Leistungsfähigkeit eines einzelnen Servers durch die Verstärkung von Ressourcen gesteigert wird. Dies wird erreicht durch beispielsweise die Erhöhung der Rechenleistung, des Arbeitsspeichers oder der Speicherkapazität. Im Kontext von CI/CD-Pipelines bedeutet dies, dass die Verarbeitungsgeschwindigkeit der Pipeline durch die Erhöhung der Ressourcen auf einem einzelnen Server gesteigert werden sollten. Problematisch bei diesem Ansatz ist jedoch, dass die Leistungsfähigkeit eines einzelnen Servers begrenzt ist und somit die vertikale Skalierung nur bis zu einem gewissen Grad möglich ist. Darüber hinaus kann die vertikale Skalierung auch zu höheren Kosten führen, da die Anschaffung leistungsstärkerer Server teurer ist als die von weniger leistungsstarken Servern. Diese höheren Kosten können außerdem auch die Wartung betreffen, wenn beispielsweise Großrechner angeschafft werden, die durch speziell geschultes Personal gewartet werden müssen.

Für das Experiment der Arbeit ist die vertikale Skalierung insofern relevant, als die Leistungsfähigkeit der einzelnen Rechenknoten ein Parameter dafür ist, wie viele Threads und Prozesse parallel auf einem Knoten ausgeführt werden können.

Horizontale Skalierung

Im Gegensatz zur vertikalen Skalierung setzt die horizontale Skalierung auf eine Erhöhung der Leistungsfähigkeit durch die Erhöhung der Ressourcen in einem Gesamtsystem. Das bedeutet, dass mehrere Knoten dem System hinzugefügt werden, um die anfallende Arbeitslast weiter aufzuteilen. Möchte man eine CI/CD-Pipeline horizontal skalieren, so bedeutet dies, dass mehrere Server als Knoten zur Berechnung der Pipeline-Schritte

hinzugefügt werden. Interessant ist dieser Ansatz insbesondere dann, wenn besonders rechenintensive Abschnitte aus der Pipeline ausgelagert werden können. Somit können mehrere Knoten parallel an der Berechnung des Gesamtprozesses arbeiten. Ein weiterer interessanter Aspekt bei der horizontalen Skalierung ist, dass durch ein dynamisches Hinzufügen und Entfernen von Knoten die Skalierung an die aktuelle Arbeitslast angepasst werden kann. Dies bedeutet, dass die Ressourcen des Systems effizienter genutzt werden können und somit die Kosten für die Bereitstellung der Pipeline bestenfalls reduziert werden können.

Bezieht man dies auf das Experiment der Arbeit, so bedeutet dies, dass die horizontale Skalierung der Pipeline durch das Hinzufügen von weiteren Rechenknoten die Ausführungsgeschwindigkeit der Pipeline steigern kann. Darüber hinaus kann die horizontale Skalierung auch dazu genutzt werden, nur so viele Ressourcen zu nutzen, wie benötigt werden. Dies bedeutet, dass die Kosten für die Bereitstellung der Pipeline nur so hoch sind wie erforderlich.

2.4 Warteschlangen

Warteschlangen sind ein lang etabliertes Konzept der Informatik und spielen auch im Kontext von CI/CD-Pipelines eine interessante Rolle. Da die Warteschlangentheorie einige relevante Gesetzmäßigkeiten und Optimierungsmöglichkeiten bietet, wird im Folgenden ein Überblick über die grundlegenden Konzepte und Gesetzmäßigkeiten gegeben und deren Anwendbarkeit auf CI/CD-Pipelines erläutert.

2.4.1 Gesetzmäßigkeiten

Einige der wichtigsten Gesetzmäßigkeiten der Warteschlangentheorie sind Littles Gesetz, Amdahls Gesetz und Gustafsons Gesetz. Diese Gesetzmäßigkeiten beschreiben die Beziehung zwischen verschiedenen Parametern einer Warteschlange und ermöglichen es, die Leistungsfähigkeit von Warteschlangen zu analysieren und zu optimieren.

Littles Gesetz

Littles Gesetz [16], benannt nach John D. C. Little, ist eines der bekanntesten Gesetze der Warteschlangentheorie. Seine Formel

$$L = \lambda * W \tag{2.1}$$

beschreibt die Beziehung zwischen der Anzahl der Einheiten in einer Warteschlange (L), der durchschnittlichen Anzahl der Einheiten, die pro Zeiteinheit in das System eintreten (λ), und der durchschnittlichen Zeit, die eine Einheit im System verbringt (W). Das Gesetz von Little bietet die Grundlage für weitere Gesetzmäßigkeiten zur Analyse und Optimierung von Warteschlangen.

Amdahls Gesetz

Amdahls Gesetz [3], benannt nach Gene M. Amdahl, dient zur Vorhersage der maximal zu erwartenden Verbesserung in einem Gesamtsystem. Es beschreibt dazu im Bereich des parallelen Rechnens den maximal erreichbaren Speed-up, anhand der Anzahl der Prozessoren und des Anteils eines Programms, der sequentiell ausgeführt wird. Zur Herleitung des Gesetzes werden einige grundlegende Definitionen benötigt. So gilt für den sequentiellen Anteil f eines Algorithmus:

$$0 \geq f \geq 1 \tag{2.2}$$

Wenn demnach der sequentielle Anteil gleich 0 ist, so ist der Algorithmus vollständig parallelisierbar. Wenn der sequentielle Anteil jedoch 1 beträgt, so ist der Algorithmus nicht parallelisierbar. Daraus ergibt sich weiter der parallel ausführbare Anteil des Algorithmus als:

$$(1 - f) \tag{2.3}$$

Nun besagt Amdahls Gesetz, dass die Zeit zur Ausführung des Algorithmus auf einem Prozessor durch $T^*(n)$ definiert ist. Demnach entspricht $T'(n)$ der einfachen Ausführungszeit des Algorithmus. Die Gesamtlaufzeit des Algorithmus ergibt sich aus dem sequentiellen Anteil plus dem parallelen Anteil. Die gesamte parallele Laufzeit ist dann größer gleich der Summe der Zeit für den sequentiellen Teil und die parallelen Teile, dividiert durch die Anzahl p der Prozessoren:

$$T_p(n) \geq f * T'(n) + \frac{(1 - f) * T'(n)}{p} \tag{2.4}$$

Zuletzt folgt für die maximal erreichbare Beschleunigungsrate:

$$\begin{aligned} S_p(n) &= \frac{T'(n)}{T_p(n)} \\ &= \frac{T'(n)}{f * T'(n) + \frac{(1-f)*T'(n)}{p}} \\ &= \frac{1}{f + \frac{(1-f)}{p}} \qquad \textbf{(Amdahls Gesetz)} \end{aligned} \tag{2.5}$$

Darüber hinaus lässt sich für ein $f > 0$ und $p \to \infty$ zeigen, dass gilt:

$$S_p(n) \geq \frac{1}{f} \tag{2.6}$$

Gustafsons Gesetz

Gustafsons Gesetz [9], bennant nach John L. Gustafson, ist ein weiteres Gesetz, welches die Leistungsfähigkeit von parallelen Systemen beschreibt. Im Gegensatz zu Amdahls Gesetz bezieht sich Gustafsons Gesetz auf die Skalierbarkeit von parallelen Systemen, indem es die Größe des Problems mit der Anzahl der Prozessoren in Beziehung setzt. Konkret bedeutet dies, dass der verteilbare Anteil des Problems mit der Problemgröße und der Prozessoranzahl nach Gustafson linear skaliert. Das Gesetz kann hergeleitet werden, indem für den sequentiellen Anteil f eines Algorithmus gilt:

$$f = \frac{f_1}{p * (1 - f_1) + f_1} \tag{2.7}$$

In Bezug auf Amdahls Gesetz folgt:

$$\begin{aligned} S_p(n) &= \frac{1}{f + \frac{(1-f)}{p}} \\ &= \frac{1}{\frac{f_1}{p*(1-f_1)+f_1} + \frac{1-\frac{f_1}{p*(1-f_1)+f_1}}{p}} \\ &= p * (1 - f_1) + f_1 \qquad \textbf{(Gustafsons Gesetz)} \end{aligned} \tag{2.8}$$

Damit ergibt sich, dass bei einer festen Laufzeit f_1 und einer durch die Prozessorenanzahl p begrenzten Problemgröße die Optimierung mit einem konstanten sequentiellen Teil annähernd linear mit der Prozessoranzahl skaliert.

2.4.2 Anwendung der Warteschlangentheorie in CI/CD-Pipelines

Die Warteschlangentheorie bietet wertvolle Einsichten in das Management von Prozessen innerhalb von CI/CD-Pipelines. Insbesondere Littles Gesetz [16] und die Gesetze von Amdahl [3] und Gustafson [9] liefern Grundlagen für das Verständnis des Verhaltens von Warteschlangen und die Optimierung von Prozessen in verteilten Systemen. Durch die geschickte Parametrierung dieser Gesetzmäßigkeiten können Optimierungsstrategien entwickelt werden, die darauf abzielen, Engpässe zu identifizieren und zu beseitigen.

2.4.3 Zusammenhänge von Warteschlangen und Pipelines

Bei der Bezeichnung *CI/CD-Pipeline* ist die Analogie zum Pipeline-Entwurfsmuster [8] offensichtlich. Weniger offensichtlich ist jedoch die Tatsache, dass CI/CD-Pipelines auch als Warteschlangen angesehen werden können.

Betrachtet man zunächst das Entwurfsmuster der Pipeline [8], so lassen sich direkte Zusammenhänge mit den durch Little [16] beschriebenen Warteschlangen beobachten. So ist eine Pipeline nach Definition eine Abfolge von sequentiellen Verarbeitungsprozessen, die über ihre Ein- und Ausgabe gekoppelt sind. Darüber hinaus steht am Anfang einer Pipeline eine Datenquelle und am Ende eine Datensenke. Die Datenquelle sendet Daten in die Pipeline, wie in Littles Warteschlangenmodell die ankommenden Einheiten. Die aus der Datensenke ausgehenden Daten entsprechen demnach den abgearbeiteten Einheiten. Die Verarbeitungsschritte innerhalb der Pipeline entsprechen den Warteschlangenprozessen, die die ankommenden Einheiten verarbeiten und an die nächste Stelle in der Pipeline weitergeben.

Allerdings sollte an dieser Stelle auch differenziert werden, da CI/CD-Pipelines in der Praxis oft nicht nur auf die Eigenschaften des Pipeline-Entwurfsmusters beschränkt sind. So lassen sich, ähnlich wie in den Gesetzen von Amdahl und Gustafson beschrieben, auch parallele Verarbeitungsprozesse integrieren. Dies ist insbesondere dann der Fall, wenn die Verarbeitungsschritte, die man in der CI/CD-Pipeline abbilden möchte, unabhängig voneinander ausgeführt werden können. Folgt man dieser Argumentation, so

wären mehrere unabhängige Verarbeitungsprozesse, bestehend aus einzelnen Verarbeitungsschritten, aneinandergereiht, wie in der folgenden Abbildung 2.4 dargestellt. Bei den Bezeichnungen sollte an dieser Stelle darauf geachtet werden, dass mit dem Begriff des *Verarbeitungsprozesses* am ehesten eine Gruppe von Schritten gemeint ist. Diese werden in der Praxis oft auch als *Stages* bezeichnet.

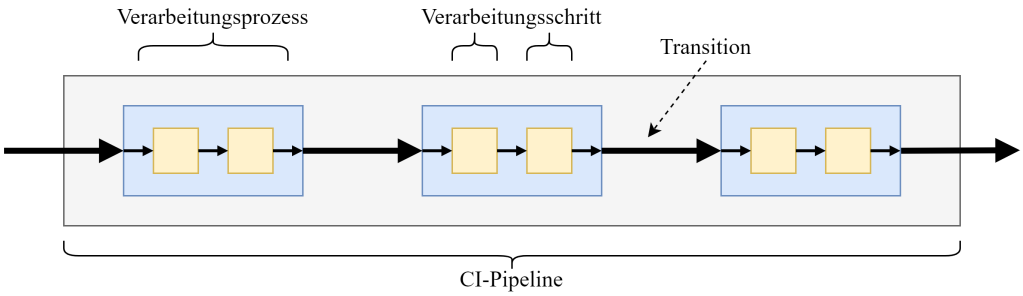

Abbildung 2.4: Aneinanderreihung von Verarbeitungsprozessen in einer CI/CD-Pipeline (eigene Darstellung)

2.5 Batchverfahren

Batching oder Stapelverarbeitung ist ein Konzept, das in der Informatik und insbesondere in der Verarbeitung von Daten und Aufgaben lange weit verbreitet ist. Im Folgenden wird ein kurzer Überblick über die Grundlagen und Anwendungen von Batchverfahren auf CI/CD-Pipelines gegeben.

2.5.1 Batching-Verfahren zur Prozessoptimierung

Ein weiterer effektiver Ansatz zur Optimierung von CI/CD-Pipelines ist die Implementierung von Batching-Verfahren. Diese Verfahren gruppieren mehrere Aufgaben zu einem Batch, bevor diese verarbeitet werden. Insbesondere kann dies bei Tests und dem Build-Prozess von Vorteil sein, da es die Anzahl der benötigten Operationen reduziert und die Auslastung der Ressourcen optimiert. Wie einige Studien zeigen, kann durch Batching die Gesamtbearbeitungszeit signifikant reduziert werden, indem die Zeit, die für die Initialisierung und den Abschluss von Prozessen benötigt wird, minimiert wird [2][6][19].

2.5.2 Implementierung von Batching-Verfahren in CI/CD-Pipelines

Die Implementierung von Batching-Verfahren in CI/CD-Pipelines erfordert eine sorgfältige Planung und Konfiguration. Es ist notwendig, ein Gleichgewicht zwischen der Größe der Batches und der Verfügbarkeit der Ressourcen zu finden. Zu große Batches können zu einer Überlastung der Ressourcen führen, während zu kleine Batches die Vorteile des Batching minimieren. Eine dynamische Anpassung der Batch-Größe, basierend auf der aktuellen Auslastung und Verfügbarkeit der Ressourcen, kann eine effektive Strategie sein, um die Vorteile von Batching zu maximieren, wie aus anderen Arbeiten hervorgeht [2][6][19]. Ein Beispiel für den Aufbau eines konkreten Batching-Verfahrens ist anhand des BatchBisect-Batching-Verfahrens aus der Arbeit von Beheshtian et al. [2] in Abbildung 2.5 zu sehen.

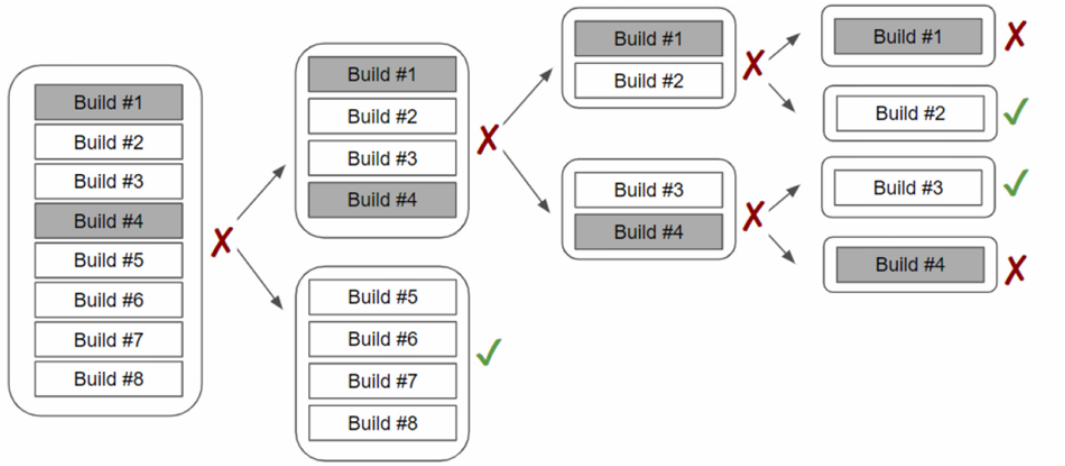

Abbildung 2.5: Beispiel – BatchBisect-Batching-Verfahren [2]

3 Verwendete Technologien

In diesem Kapitel werden die Technologien vorgestellt, welche benötigt werden, um das System für die folgenden Experimente aufzubauen. Dabei wird insbesondere auf die Funktionsweise und weitere Besonderheiten, die für die Experimente relevant sind, eingegangen.

3.1 Container

Container stellen eine wesentliche Grundlage für den Aufbau der späteren Versuche dar. Daher soll im folgenden die verwendete Container-Plattform Docker sowie die Container-Orchestrierungsplattform Kubernetes näher erläutert werden.

3.1.1 Docker

Als eine zentrale Technologie für die Bereitstellung von Containern wird die heute stark verbreitete Containerplattform Docker verwendet [14]. Docker ist eine Open-Source-Plattform, die es ermöglicht, Anwendungen isoliert in Containern auszuführen. Dazu wird gemäß dem Prinzip der Containervirtualisierung die Möglichkeit geboten, Container unabhängig vom Hostbetriebssystem auszuführen und zu verwalten. Um dies zu realisieren, greift Docker auf die Funktionalitäten des Linux-Kernels zurück. Damit wird ein möglichst geringer Overhead bezüglich der Performanz erzeugt. Allgemeiner basiert Docker auf einer Client-Server-Architektur und ist in drei wesentliche Bestandteile unterteilbar. Diese sind der Docker-Client, der Docker-Host, welcher den Docker-Daemon beinhaltet, und die Docker-Registry. Der Client dient für den Nutzer als Schnittstelle, um Befehle wie beispielsweise „docker run“ an den Daemon via RESTful-API zu senden. Auf der Seite des Docker-Hosts gibt es neben dem Daemon, welcher die Ausführung sämtlicher komplexer Befehle übernimmt, auch die Docker-Images und die Docker-Container.

Die Docker-Images dienen als Abbilder kompletter Systeme für die Erstellung der Container und werden in der Docker-Registry gespeichert. Die Docker-Registry dient zur Speicherung und Verteilung von Images. Sie kann sowohl als öffentlicher Dienst wie auch als private Instanz betrieben werden [36]. Konkret lässt sich die Funktionsweise an der folgenden Abbildung 3.1 verdeutlichen:

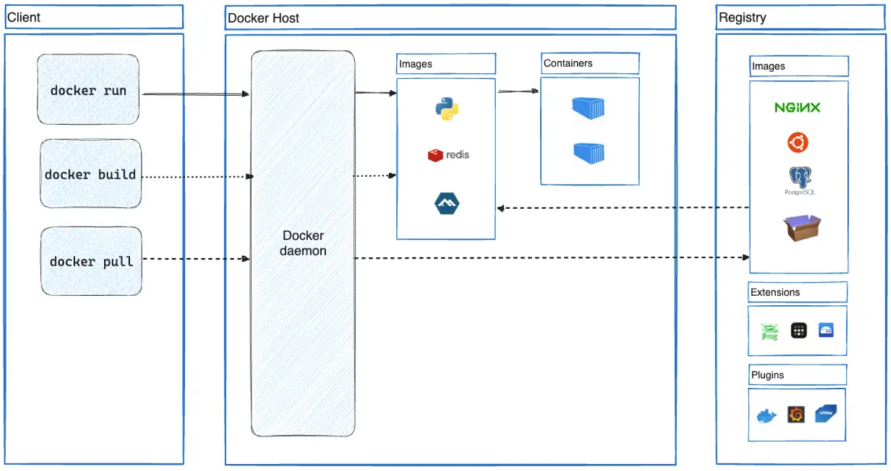

Abbildung 3.1: Schema – Docker-Architektur [11]

Für die Versuche der Arbeit spielen die Docker-Container eine zentrale Rolle. Sie dienen dazu, eine Ausführungsumgebung für Schritte innerhalb der CI/CD-Pipeline bereitzustellen. So gibt es beispielsweise einen Container, dessen Dockerfile alle notwendigen Softwarekomponenten beinhaltet, die für das Kompilieren und Ausführen von Tests in einem Maven Projekt notwendig sind. Ein weiterer Container beinhaltet hingegen einen Stresstest. Dieser zweite Container wird mehrfach parallel gestartet, um die Auswirkungen auf die Ressourcennutzung des Gesamtsystems zu untersuchen. Allgemein bieten die Container also eine Möglichkeit, verschiedene Softwarekomponenten für den jeweiligen Ausführungskontext zu kapseln und vom restlichen System zu isolieren. Die verwendeten Dockerfiles, die die Container abbilden, sind im Anhang A.4 zu finden.

3.1.2 Kubernetes

Kubernetes ist ein Container-Orchestrierungstool, welches von Google entwickelt wurde und heute von der Cloud Native Computing Foundation (CNCF) als Open-Source-Projekt verwaltet wird [24]. Zur Orchestrierung von Containern bietet Kubernetes eine

Vielzahl von Funktionalitäten [25]. Besonders relevant sind Funktionen wie das Bereitstellen von Containern in verteilten Clustern sowie die Fähigkeit zur automatischen Skalierung und Verwaltung von Containern. Zur Realisierung dieser Funktionalitäten basiert Kubernetes auf einem Kontrollknoten (Control-Plane) und meist mehreren Arbeiterknoten (Worker), welche die eigentliche Ausführung der Container übernehmen. Für die späteren Versuche wurde diese Struktur von Kontroll- und Arbeiterknoten virtualisiert.

Zur Ausführung der Container greift Kubernetes auf unterliegende Containerplattformen wie beispielsweise Docker zurück. Besonders ist hierbei, dass Kubernetes eine zusätzliche logische Einheit zur Strukturierung von Systemen verwendet, die sogenannten Pods. Ein Pod ist eine Gruppe von einem oder mehreren Containern, die gemeinsam auf einem Arbeiterknoten ausgeführt werden. Dabei teilen sich die Container eines Pods die gleiche Netzwerk- und Speicherumgebung. Dies ermöglicht es, dass Container innerhalb eines Pods miteinander kommunizieren können, ohne dass zusätzliche Konfigurationen notwendig sind. Zwischen den verschiedenen Knoten kommunizieren die Knoten über eine RESTful-API. Damit der Anwender mit Kubernetes interagieren kann, wird ein Client mithilfe einer Kommandozeilenschnittstelle, namens „kubectl“, bereitgestellt. Im späteren Experiment werden für jeden Versuch Pods durch den Jenkins-Server automatisiert erstellt und mit den zuvor beschriebenen Containern bestückt.

Weitere wichtige Details sind zum einen die Möglichkeit zur Definition von Ressourcen via, meist YAML-basierten, Konfigurationsdateien und zum anderen die Möglichkeit der Verwaltung von Ressourcen in Namespaces. Namespaces dienen dazu, Ressourcen zu gruppieren und zu isolieren. Dies ist insbesondere dann relevant, wenn mehrere Teams oder Anwendungen auf einer Kubernetes-Instanz betrieben werden. So wird beispielsweise auch in den Versuchen jeder Pod in einem Namespace betrieben, der an einen bestimmten Worker-Knoten gebunden ist. Dies ist wiederum, genau wie auch die Container der Pods, in einer Konfigurationsdatei definiert.

Weitere besonders relevante Ressourcen, die in Kubernetes definiert werden können, sind Services, Deployments, StatefulSets und DaemonSets, wobei auf StatefulSets und DaemonSets im Rahmen dieser Arbeit nicht weiter eingegangen wird. Services dienen im Wesentlichen dazu, die Kommunikationsinfrastruktur für die Pods bereitzustellen. Dazu gehört beispielsweise die Zuweisung von Hostnamen zu IP-Adressen oder das Load-Balancing. Auch Services werden für die Einrichtung der Infrastruktur im späteren Ex-

periment genutzt. Sie sind dafür zuständig, dass beispielsweise auf die Weboberflächen der Anwendungen zugegriffen werden kann.

Deployments dienen dazu, die Bereitstellung von Pods zu steuern. Sie ermöglichen es beispielsweise, die Anzahl der Replikate eines Pods zu definieren und sicherzustellen, dass die Pods in einer bestimmten, definierten Version bereitgestellt werden. Dies spielt ebenfalls für die späteren Versuche nur in Bezug auf die eigentliche Infrastruktur eine Rolle. Die Pipeline-Definitionen, die im Jenkins-Server gegeben werden, unterstützen keine Deployments.

Zum besseren Verständnis der Funktionsweise von Kubernetes kann die Abbildung 3.2 betrachtet werden.

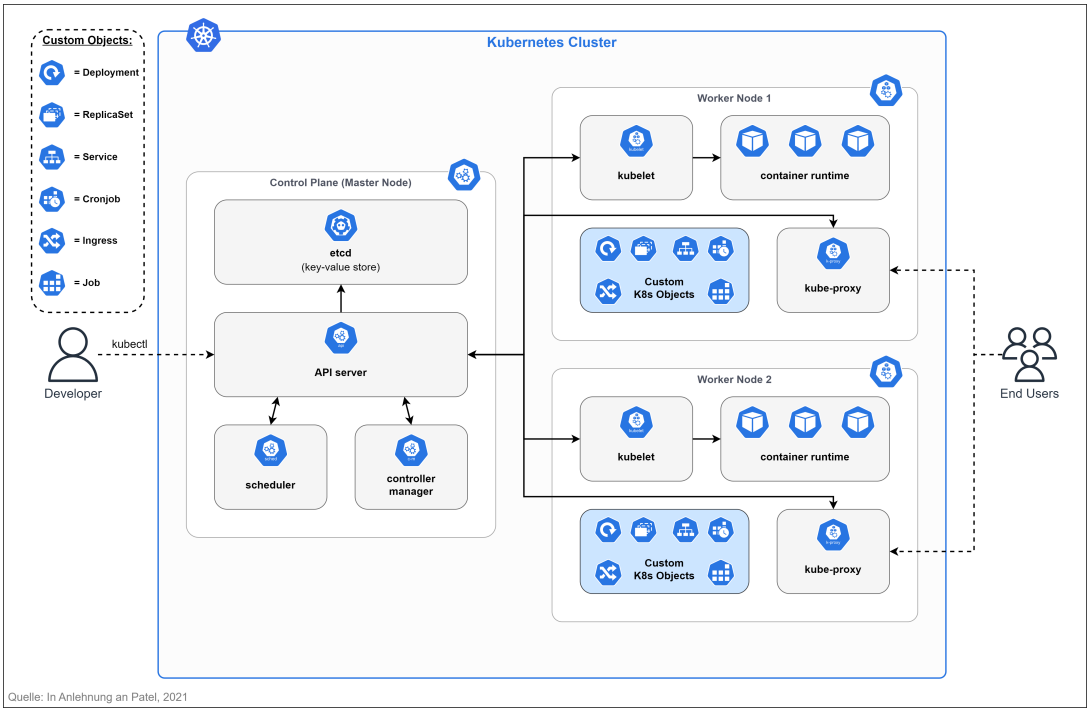

Abbildung 3.2: Lokale Kubernetes-Cluster-Architektur – in Anlehnung an [20]

3.2 Continuous Integration

Um die CI/CD-Pipelines für die späteren Versuche zu realisieren, wird Jenkins als zentrale Softwarelösung für die Steuerung der Pipelines verwendet. Im folgenden Abschnitt wird Jenkins näher erläutert.

3.2.1 Jenkins

Für die Integration von Continuous-Integration- und Delivery-Prozessen ist Jenkins eine der nach Bezemer et al. [4] am weitesten verbreiteten Softwarelösungen. Auch wenn seit 2019 die Verbreitung anderer Softwareprodukte dieser Kategorie zugenommen hat, gilt Jenkins immer noch als eine der wichtigsten und am häufigsten verwendeten CI/CD-Lösungen. Dies macht sie zur Technologie für die Steuerung der CI/CD-Pipelines im Rahmen der Versuche.

Jenkins ist eine Open-Source-Software und basiert auf einem Controller und einem oder mehreren Agent-Knoten (Master-Slave). Der Controller dient dabei als zentrale Instanz zur Steuerung und Verwaltung der Agent-Knoten. Er stellt außerdem beispielsweise auch die Weboberfläche für den Anwender zur Verfügung. Über diese Weboberfläche werden die CI/CD-Pipelines eingerichtet und gesteuert. Die Agent-Knoten hingegen übernehmen die eigentliche Ausführung der Pipelines. Im Rahmen der Versuche der Arbeit werden die Agent-Knoten dynamisch auf Basis von Kubernetes Pods verwaltet. So können die benötigten Ressourcen nach Bedarf der jeweiligen Pipeline skaliert werden.

Für die Erstellung von CI/CD-Pipelines nutzt Jenkins wahlweise eine eigene deklarative Syntax oder eine Groovy-basierte domainspezifische Sprache (DSL), welche zur Erstellung der sogenannten Jenkinsfiles genutzt wird. Für die Versuche wurde sowohl auf die deklarative Syntax als auch auf die Groovy-basierte DSL zurückgegriffen. Sogar die Kombination beider Syntaxen ist möglich und wurde auch in den Versuchen genutzt, um Prozessabläufe zu definieren, die nur mit der deklarativen Syntax nicht realisierbar gewesen wären. Besonders interessant an den Jenkinsfiles ist auch der Fakt, dass diese üblicherweise neben den Projektdateien in einem Versionierungssystem abgelegt werden. So kann der Jenkins-Server für eine konfigurierte Pipeline das entsprechende Jenkinsfile überwachen und bei Änderungen automatisch die Pipeline aktualisieren und aktivieren.

Eine weitere Stärke von Jenkins ist die Integration einer großen Anzahl von Plug-ins, mit denen die Plattform beliebig erweitert werden kann. Mithilfe solcher Plug-ins kann unter anderem auch der Einsatz von Docker und Kubernetes im Zusammenspiel mit den CI/CD-Pipelines realisiert werden. Dies wurde auch in den Versuchen der Arbeit genutzt, um die Container-Infrastruktur für die Ausführung der Schritte innerhalb der CI/CD-Pipelines bereitzustellen. Dabei erlaubt das Kubernetes-Plug-in für Jenkins die Definition der Kubernetes-Manifeste zur Erstellung der Pods direkt in den Jenkinsfiles.

Die Kommunikation zwischen den Jenkins-Knoten kann über unterschiedliche Technologien erfolgen. So nutzt Jenkins meist JNLP (Java Network Launch Protocol) für die Kommunikation zwischen dem Controller und den Agent-Knoten. Alternativ können auch SSH oder WebSockets verwendet werden. Aufgrund der Einfachheit und der standardmäßigen Konfiguration von JNLP wurde diese Technologie auch in den Versuchen der Arbeit genutzt.

3.3 Tests und Scanner

Das Testen und Scannen nach Sicherheitslücken sind wichtige Bestandteile der Softwareentwicklung. Um einen konkreteren Blick auf die im Rahmen der späteren Versuche verwendeten Test- und Scanverfahren zu geben, werden diese im folgenden Abschnitt erläutert.

3.3.1 Tests

Testen ist ein besonders wichtiger Teil der Softwareentwicklung. Im Rahmen der Arbeit soll ein besonderer Fokus auf den Teststufen der Komponenten- und Integrationstests liegen. Spillner et al. [29] bieten dazu eine gute Erläuterung der beiden Teststufen. Komponententests, welche oft auch Unittests genannt werden, dienen demnach dazu, die Funktionalität einzelner Komponenten zu überprüfen. Integrationstests hingegen haben den Zweck, die Interaktion zwischen verschiedenen Komponenten zu überprüfen. Dabei wird in der Regel ein komplettes Modul oder eine komplette Anwendung getestet. Im Rahmen von Continuous-Integration-Prozessen werden diese Tests in der Regel automatisiert durchgeführt.

Zur Realisierung der Komponenten- und Integrationstests wird im Rahmen der späteren Versuche die Verwendung von „JUnit“, welches ein bekanntes Test-Framework im Kontext von Java ist, sowie „Mockito“, das der Erstellung von Mock-Objekten zur Durchführung von REST-API-Tests dient, vorgesehen. JUnit bietet die Möglichkeit, Tests in Form von Testklassen zu definieren. Diese Testklassen werden dann über Build-Werkzeuge, wie Maven im Rahmen von Versuch 3, in CI/CD-Pipelines integriert. Darüber hinaus wird speziell für die Integrationstests auf die Software „Testcontainers“ zurückgegriffen. Testcontainers bietet die Möglichkeit, abhängige Software in Containern zu isolieren und so beispielsweise Datenbanken oder andere Dienste für Testzwecke bereitzustellen. Dies ist auch für die Datenbankoptionen des beispielhaften Softwareprojekts in Versuch 3 entsprechend integriert worden.

3.3.2 SCA-, SAST- und DAST-Scanner

Bei der automatisierten Bereitstellung von Software spielen auch Sicherheitsaspekte eine wichtige Rolle. Diese Rolle ist sogar so relevant, dass im Rahmen von DevOps das Konzept des DevSecOps etabliert wurde, welches sich tiefergehend mit dem Aspekt der Sicherheit im Rahmen von herkömmlichen DevOps auseinandersetzen soll. Um möglichst hohe Sicherheit bei der Bereitstellung von Software zu gewährleisten, bietet sich neben der Verwendung von klassischen Tests auch spezielle Software zum Scannen von Sicherheitslücken an. Im Bezug auf CI/CD-Pipelines, wie auch im Rahmen der Versuche dieser Arbeit, spielen hier insbesondere Scanner aus der Kategorie der „Software-Composition-Analysis“ (SCA), des „Static-Application-Security-Testing“ (SAST) und des „Dynamic-Application-Security-Testing“ (DAST) eine wichtige Rolle. Einen Überblick über die Funktionsweise und den Einsatzbereich dieser Scanner-Kategorien bieten unter anderem die Arbeiten von Rajapakse et al. [22], Marandi et al. [17] und Chen et al. [5]. Demnach legen SCA-Scanner ihren Fokus auf die Überprüfung von Sicherheitslücken in Abhängigkeiten, die in einem Software-Projekt verwendet werden. Dabei wird in der Regel auf Datenbanken von bekannten Sicherheitslücken in Abhängigkeiten zurückgegriffen. SAST-Scanner dienen dazu, Sicherheitslücken in den Quellcode-Dateien zu finden. Dabei beschränken sich die Scanner in der Praxis nicht nur auf das Finden von sogenannten „Common Vulnerabilities and Exposures“ (CVE), sondern bieten auch auf das Finden von allgemeinen Sicherheitslücken, die durch schlechten Quellcode entstehen können. DAST-Scanner hingegen dienen dazu, Sicherheitslücken in der laufenden Anwendung zu finden. So können sie den Ablauf der Anwendung untersuchen und bei-

spielsweise automatisierte Angriffszenarien durchführen. Alle diese Scanner-Kategorien sind besonders wichtig, um die Sicherheit von Software zu gewährleisten. Im Rahmen des späteren Experiments wird beispielhaft unter anderem die Verwendung von „Docker Scout“ als SCA-Scanner, „SonarQube“ als SAST-Scanner und „OWASP ZAP“ als DAST-Scanner vorgesehen. Docker Scout dient dazu, die abhängigen Softwarepakete in den späteren Containerabbildern, im Experiment, auf Sicherheitslücken zu überprüfen. SonarQube bietet hingegen die Möglichkeit, Sicherheitslücken in Quellcode-Dateien zu finden und zu beheben. OWASP ZAP eröffnet darüber hinaus die Option, Sicherheitslücken in laufenden Anwendungen zu finden und zu beheben. Eine Auflistung der verwendeten Scanner und von deren Schwerpunkten ist in der Tabelle A.12 im Anhang zu finden.

3.4 Monitoring

Der Aufbau eines geeigneten Monitoringsystems ist essentiell für die Überwachung der automatisierten Prozesse. Um die entstehenden Metriken und Daten der späteren Versuche zu sammeln und zu visualisieren, werden die im Folgenden vorgestellten Softwarelösungen verwendet.

3.4.1 cAdvisor

Ein besonderer Vorteil bei der Verwendung von Kubernetes ist die Möglichkeit, Container-Ressourcen zu überwachen. Eine der am meisten genutzten Softwarelösungen, die diese Funktionalität bereitstellen, ist cAdvisor. cAdvisor ist eine Open-Source-Software, die ursprünglich von Google entwickelt wurde. Besonders ist, dass cAdvisor bereits standardmäßig in der Binary von kubelet integriert ist. Die entstehenden Daten, die von cAdvisor überwacht werden, werden in der Regel in Form von Zeitreihendaten bereitgestellt. Diese können dann beispielsweise in Prometheus gesammelt und weiterverarbeitet werden. cAdvisor spielt für die späteren Versuche eine wichtige Rolle, um die Auswirkungen der parallelisierten Pipelines auf die Ressourcennutzung der Container zu überwachen.

3.4.2 Prometheus

Prometheus zeichnet sich durch seine effektive Sammlung und Verarbeitung von Metriken mithilfe eines HTTP-Pull-Modells aus und ist ein wichtiges Instrument für die Überwachung und Alarmierung in modernen IT-Umgebungen. Diese Metriken werden auf der Plattform als Zeitreihendaten gespeichert, die durch Zeitstempel und Labels ergänzt werden. Dies ermöglicht eine detaillierte Analyse und Visualisierung. Prometheus bietet zahlreiche Möglichkeiten zur Datenauswertung, da es die Abfragesprache PromQL unterstützt und mit Visualisierungstools wie Grafana kompatibel ist. Besonders ist die Fähigkeit, unabhängig und zuverlässig in dynamischen und serviceorientierten Architekturen zu funktionieren. Prometheus ist daher für die Leistungsüberwachung und Fehlerdiagnose in verteilten Systemen ein passendes Werkzeug [21]. Die Daten, die von cAdvisor gesammelt werden, können an Prometheus weitergereicht und anschließend in Grafana visualisiert werden. Damit kann im Rahmen der Versuche sichergestellt werden, dass die vom Kubernetes-Cluster gemessenen Metriken gesammelt und visualisiert werden können.

3.4.3 Grafana

Zur Visualisierung von unterschiedlichen Metriken im späteren Experiment ist eine passende Software notwendig, welche mit den entsprechenden Datenquellen integrierbar ist. Eine solche Software ist Grafana. Grafana bietet die Möglichkeit, Daten aus unterschiedlichen Datenquellen zu integrieren und zu visualisieren. Im Rahmen des Experiments wird Grafana genutzt, um die von Prometheus gesammelten Metriken zu visualisieren.

4 Evaluation

Im folgenden Kapitel wird die Evaluation der Versuche im Rahmen dieser Arbeit beschrieben. Dazu wird zunächst darauf eingegangen, welche Erwartungen, welcher Versuchsaufbau und welcher Ablauf der Versuche geplant wurden. Anschließend wird die Durchführung der Versuche beschrieben und die gemessenen Daten werden ausgewertet. Abschließend wird die Gesamtauswertung der Versuche vorgenommen und ein Bezug zu den theoretischen Grundlagen hergestellt.

4.1 Systemaufbau

Zur Durchführung der Versuche wurde ein System aufgebaut, welches aus einem einzelnen Rechner besteht. Dieser Rechner hat eine AMD Ryzen 7 3800X CPU mit 8 Kernen und 16 Threads sowie 32 GB DDR4 Arbeitsspeicher verbaut. Zur Simulation der verschiedenen Systemknoten wurde auf dem Rechner der Typ-1-Hypervisor VMware ESXi in der Version 8.0U1a genutzt. Auf diesem Hypervisor wurden vier virtuelle Maschinen mit Ubuntu 22.04.2 LTS als Betriebssystem erstellt und konfiguriert. Alle virtuellen Maschinen sind über ein lokales Netzwerk miteinander verbunden, welches über den Hypervisor konfiguriert wurde. Die Konfiguration der virtuellen Maschinen ist im Anhang A.14 bereitgestellt. Im Weiteren wurde auf einer der virtuellen Maschinen ein Docker Host eingerichtet, um einzelne Container unabhängig vom restlichen System zu betreiben. Die drei weiteren virtuellen Maschinen wurden für die Installation eines einfachen Kubernetes-Clusters verwendet. Dieses Cluster verfügt über eine einzelne Control-Plane und zwei Worker-Nodes. Die Installation des Clusters wurde mithilfe der Konfigurationsmanagement-Software „Ansible“ automatisiert. Die entsprechenden Ansible Playbooks sind im Anhang beigefügt A.3.2. Innerhalb des Kubernetes-Clusters wird ein Jenkins Server betrieben, der die CI/CD-Pipelines der verschiedenen Versuche ausführt. Die Metriken des Clusters oder konkreter der CI/CD-Pipelines werden durch

eine ebenfalls im Kubernetes-Cluster, jedoch auf einem dedizierten Worker-Node, betriebenen Überwachungslösung aufgezeichnet. Eine Übersicht des Systemaufbaus ist in der folgenden Abbildung 4.1 dargestellt.

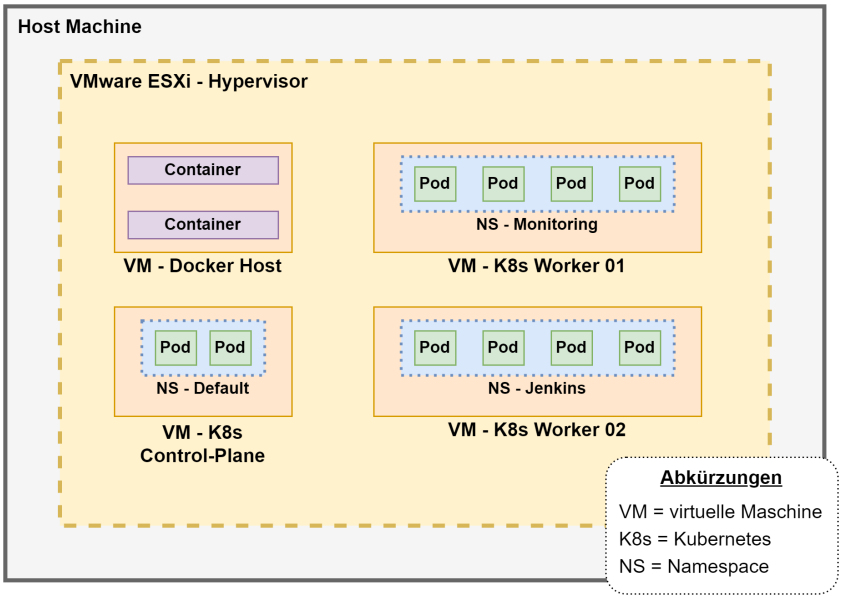

Abbildung 4.1: Systemaufbau zur Durchführung der Versuche

4.1.1 Allgemeine Softwarekonfiguration

Docker Host

Auf dem Docker Host wurde die SAST-Software „SonarQube“ als Docker-Container über das offizielle Docker-Abbild des Herstellers „Sonarsource“ installiert und über den Port 9000 von außerhalb verfügbar gemacht. Außerdem wurde die Umgebungsvariable „SONAR_ES_BOOTSTRAP_CHECKS_DISABLE“ auf den Wert „true“ gesetzt, um einige Prüfungen für eine produktive Bereitstellung zu überspringen. Diese waren für die Testumgebung nicht relevant und haben keinen Einfluss auf die Erhebung der Versuchsdaten. Die Installation erfolgte konkret mithilfe des folgenden Befehls:

Listing 4.1: SonarQube Installation

```
docker run -d --name sonarqube \
    -e SONAR_ES_BOOTSTRAP_CHECKS_DISABLE=true \
    -p 9000:9000 sonarqube:latest
```

Anschließend wurde in SonarQube ein API-Token für die Verbindung zum Jenkins Server erstellt. Dieser Token wurde in Jenkins als „Secret“ hinterlegt, um die Verbindung zu SonarQube aus den Pipelines zu ermöglichen.

Kubernetes-Cluster

Im Kubernetes-Cluster wurde die Software über unterschiedliche Wege bereitgestellt. Zum einen wurde über klassische Kubernetes-Manifest-Dateien die CI/CD-Software „Jenkins“ installiert. Die im Rahmen der Arbeit verwendeten Manifest-Dateien sind auf der Seite der offiziellen Anleitung zur Kubernetes-Installation von Jenkins [13] zu finden und beschreiben unterschiedliche Aspekte der installierten Jenkins-Instanz. Zum anderen wurde über ein sogenanntes „Helm Chart“ das Softwarepaket „kube-prometheus-stack“ installiert, welches durch das offizielle Prometheus-Projekt bereitgestellt wird. Dieses Paket stellt eine Überwachungslösung für die Kubernetes-Umgebung dar, die Prometheus (einschließlich der Software Alertmanager und Node Exporter) sowie Grafana umfasst. Die Installation erfolgte mithilfe der folgenden Befehle:

Listing 4.2: kube-prometheus-stack Installation

```
kubectl create namespace prometheus

helm install stable \
    prometheus-community/kube-prometheus-stack \
    -n prometheus
```

Darüber hinaus mussten die erstellten Services der einzelnen Anwendungen in Kubernetes vom Typ „ClusterIP“ auf „NodePort“ geändert werden, um die Anwendungen von außerhalb des Clusters im Netzwerk erreichbar zu machen.

Im Anschluss an die Installation wurde die Verbindung zwischen Jenkins und SonarQube über die Jenkins-Web-Oberfläche konfiguriert. Außerdem wurden die notwendigen

Plug-ins für Jenkins installiert, um unter anderem auf die Ressourcen des Kubernetes-Clusters zugreifen zu können und somit dynamisch Pods für neue Jenkins-„Agenten“ zu erzeugen, welche als Worker-Knoten die Pipelines verarbeiten. Die Installation der Plug-ins erfolgte ebenfalls über die Jenkins-Web-Oberfläche. Eine Liste der installierten Plug-ins ist im Anhang A.6 zu finden. Weiter wurde in der Grafana-Instanz ein Dashboard für die Überwachung der Kubernetes-Umgebung erstellt, welches in Abbildung 4.2 zu sehen ist. Dieses Dashboard wurde mithilfe der, im installierten Softwarepaket, vorkonfigurierten Prometheus-Datenquelle erstellt und zeigt die wichtigsten Metriken der Kubernetes-Umgebung sowie der Pods, die durch das Jenkins-Plug-in dynamisch erzeugt wurden, an. Zu den Metriken gehören CPU-, Arbeitsspeicher- und Festplattenmetriken auf Pod- und Node-Ebene sowie eine Vielzahl weiterer möglicher Metriken, die durch die entsprechende Software im Kubernetes-Cluster, wie beispielsweise hier cAdvisor, definiert sind. Die konkrete Konfiguration des Dashboards ist unter der Web-Adresse https://grafana.com/grafana/dashboards/20836 öffentlich verfügbar.

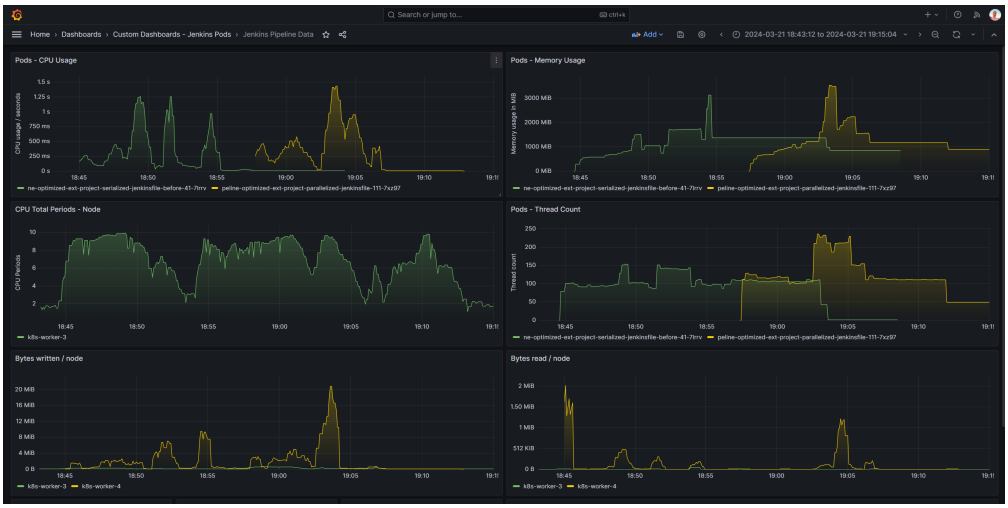

Abbildung 4.2: Grafana-Dashboard zur Überwachung der Kubernetes-Umgebung

4.1.2 Messung und Aufzeichnung der Metriken

Um die Metriken bezüglich CPU, Arbeitsspeicher, Festplattenaktivitäten sowie Durchlaufzeit zu erheben, wurden unterschiedliche Software-Werkzeuge eingesetzt. So wurden die Hardware-spezifischen Metriken, wie beispielsweise CPU- und Arbeitsspeichermetriken, durch die im Kubernetes „kubelet“ integrierte Software „cAdvisor“ erhoben. cAdvisor bietet dazu eine REST-API an, welche durch die zuvor erläuterte Prometheus-

Instanz abgefragt werden kann. Darüber hinaus kann Prometheus auch die Metriken des Kubernetes-Add-ons „kube-state-metrics“ abfragen, welches selbst Metriken zum Zustand der Cluster-Objekte durch das Belauschen der Kubernetes-API bereitstellt. Somit dient Prometheus als zentrale Datenquelle zur Datenaggregation und -speicherung. Neben diesen Daten werden zusätzlich mithilfe der Jenkins-Zeitstempelfunktion die Durchlaufzeiten der einzelnen Schritte der CI/CD-Pipelines erfasst. Diese Daten wurden manuell erhoben und als CSV-Datei zusammen mit den Metadaten der Pipelines gespeichert. Zur Auswertung der Metriken wurde zum einen, wie zuvor bereits beschrieben, die Software „Grafana“ verwendet und zum anderen wurden die Daten mittels selbst angefertigter Python-Skripte ausgewertet. Die verwendeten Skripte sind im Anhang A.5 zu finden.

4.1.3 Zeitlicher Ablauf

In der folgenden Abbildung 4.3 wird der zeitliche Ablauf einer Pipeline-Ausführung in Jenkins in dem zuvor beschriebenen System dargestellt. Diese Ausführungslogik trifft für alle im Folgenden beschriebenen Versuche zu. Im Anhang A.1 ist das Sequenzdiagramm in größerer Darstellung für eine bessere Lesbarkeit zu finden.

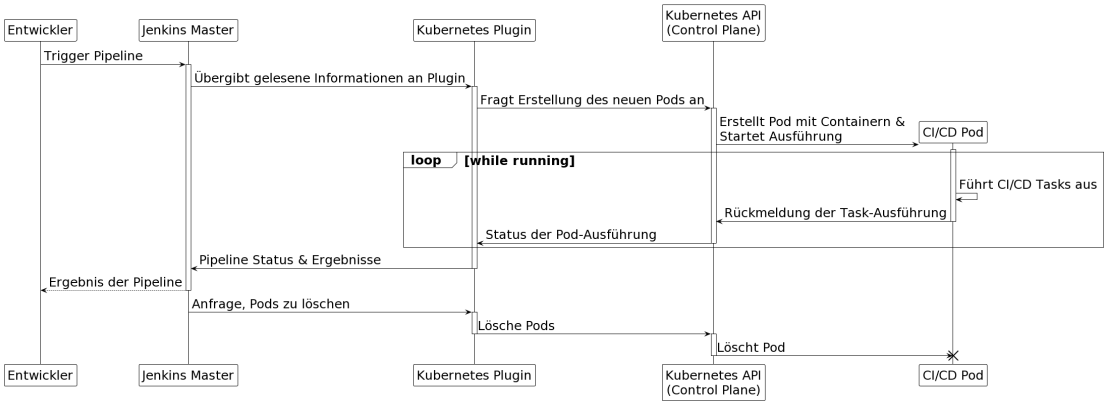

Abbildung 4.3: Sequenzdiagramm der Pipeline-Ausführung aus Systemsicht

4.2 Versuchsaufbau

Im folgenden Abschnitt werden die drei Versuche beschrieben, die im Rahmen dieser Arbeit durchgeführt wurden. Dabei wird auf die spezifische Softwarekonfiguration, den

zeitlichen Ablauf und die Erwartungen eingegangen. Zur Untersuchung der Anwendbarkeit der Gesetze von Amdahl (siehe Abschnitt 2.4.1) und Gustafson (siehe Abschnitt 2.4.1) auf parallelisierte CI/CD-Prozesse wurden unterschiedliche Konfigurationen für die CI/CD-Pipelines implementiert. Diese Vorgehensweise zielte darauf ab, zwei Hauptaspekte zu beleuchten: erstens, ob diese Gesetze auf CI/CD-Prozesse übertragbar sind, und zweitens, inwiefern sich diese Konzepte auf ein konkretes Beispiel aus der Softwareentwicklung anwenden lassen. Dabei stand insbesondere die Erstellung einer getesteten und sicheren Softwarebereitstellung im Fokus, um deren Einfluss auf die Effizienz und Durchlaufgeschwindigkeit des Gesamtprozesses zu bewerten. Die spezifischen Konfigurationen, die in den jeweiligen Versuchen angewendet wurden, werden im Folgenden detailliert beschrieben.

4.2.1 Aufbau – Versuch 1

Im ersten Versuch wurden als Basis drei Jenkinsfiles erstellt, welche Pipelines beschreiben, die jeweils in jedem Schritt der Pipeline eine simulierte Arbeitslast in einem Docker-Container ausführen. Als Software für diese Benchmarks wurde das Open-Source-Programm „stress-ng“ verwendet. Zur einfachen, immer selben Ausführung wurde der Aufruf des Benchmarks in einem Shell-Skript gekapselt und in einem Dockerfile eingebunden. Aufgabe der parallelen Benchmarks war es, die CPU zu strapazieren und eine gegebene Zielmenge an zu verarbeitenden Operationen auszuführen. Da der Rechner, auf dem die Versuche durchgeführt wurden, über 8 Kerne und 16 Threads verfügt, wurden die Benchmarks so konstruiert, dass in der Pipeline jeder Schritt einen Container mit 2 zugewiesenen CPU-Kernen und 2 GB Arbeitsspeicher erhielt. Die Benchmarks wurden so konfiguriert, dass sie pro Schritt 250 000 Operationen auf 2 CPU-Kernen berechnen sollten. Das bedeutet, bei vier parallelen Schritten wurden insgesamt 1 000 000 Operationen auf 8 CPU-Kernen in 2 Containern berechnet. Von diesem Standpunkt aus wurde bei einer festen Schrittmenge von 10 Pipeline-Schritten untersucht, wie sich eine steigende Anzahl an Workern auf die, in diesem Beispiel genannten, vier durch die Jenkinsfiles definierten Pipelines mit unterschiedlichen Anteilen an parallelen Schritten auswirkt. Die Anzahl der Worker wurde dabei bei einer Anzahl von n parallelen Schritten von 1 bis n erhöht. Diese Vorgehensweise sollte die Auswirkungen von Amdahls Gesetz auf die Durchlaufzeit der Pipelines zeigen, unter der Annahme, dass jeder Verarbeitungsschritt eine gleich starke CPU-Auslastung erzeugt. Die Jenkinsfiles sowie das Benchmark-Skript

sind im Anhang A.4.3 zu finden. Eine kurze Zusammenfassung der allgemeinen Konfigurationen ist in der folgenden Tabelle 4.1 dargestellt:

Pipeline	**Steps**	**Worker$_P$ (%)**	**Worker$_P$ (Abs.)**	**Operations**
A1.1	10	40%	2	2500000
A1.2	10	40%	3	2500000
A1.3	10	40%	4	2500000
A2.1	10	60%	2	2500000
A2.2	10	60%	3	2500000
A2.3	10	60%	4	2500000
A2.4	10	60%	5	2500000
A2.5	10	60%	6	2500000
A3.1	10	80%	2	2500000
A3.2	10	80%	3	2500000
A3.3	10	80%	4	2500000
A3.4	10	80%	5	2500000
A3.5	10	80%	6	2500000
A3.6	10	80%	7	2500000
A3.7	10	80%	8	2500000

Tabelle 4.1: Zusammenfassung der Konfigurationen für Versuch 1
(Worker$_P$ = parallele Worker, % = prozentual, Abs. = absolut)

Darüber hinaus soll die folgende Abbildung 4.4 weiter verdeutlichen, wie die Konfigurationen der Pipelines in Versuch 1 aufgebaut sind.

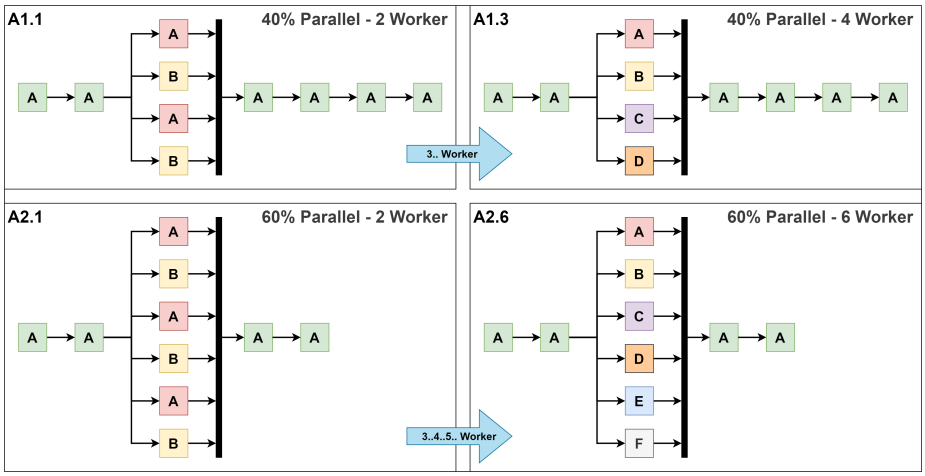

Abbildung 4.4: Versuch 1 – Minimal- nach Maximalkonfiguration (eigene Darstellung)

In der Abbildung zu sehen sind vier der insgesamt 17 Konfigurationen, die im Rahmen des Versuchs durchgeführt wurden. Auf der jeweils linken Seite sind die minimalen Konfigurationen für den entsprechenden Parallelanteil dargestellt, während auf der rechten Seite die maximalen Konfigurationen für den entsprechenden Parallelanteil angezeigt sind. Ein blauer Pfeil zwischen ihnen soll verdeutlichen, dass zwischen den beiden Konfigurationen einer Ebene weitere Konfigurationen existieren. Diese sind in der Abbildung nicht dargestellt und erhöhen entsprechend die Anzahl der Worker, bis die wieder dargestellte maximale Konfiguration erreicht ist. Grundsätzlich gilt außerdem, dass zuerst die Konfigurationen mit dem niedrigeren Benennungspräfix wie beispielsweise „A1“ vor denen mit dem höheren Präfix wie „A2“ ausgeführt wurden.

4.2.2 Aufbau – Versuch 2

Im zweiten Versuch war das Ziel, das Gesetz von Gustafson in einer CI/CD-Pipeline zu untersuchen. Hierzu wurde ebenfalls auf die simulierten Arbeitslasten aus dem ersten Versuch zurückgegriffen. Es wurden dazu erneut drei Jenkinsfiles erstellt, welche Pipelines beschreiben, die einen festen Anteil an seriellen Schritten haben. Darüber hinaus wurde dann in jeder Konfiguration eine unterschiedliche Anzahl an parallelen Schritten hinzugefügt. Die Anzahl der Gesamtschritte war dabei nicht fest, sondern entsprach der Summe der seriellen und parallelen Schritte. Durch diesen Aufbau konnte untersucht werden, wie sich die Durchlaufzeit und Beschleunigungsrate der Pipelines verhalten, wenn die Anzahl der parallelen Schritte proportional mit der Anzahl der zu verarbeitenden Operationen steigt, ohne den seriellen Anteil prozentual auf dem immer selben Niveau halten zu müssen. Dies wäre mit den 16 verfügbaren Threads nicht möglich gewesen, da die benötigten Ressourcen der parallelen Schritte die der verfügbaren Hardware überschritten hätten und somit CPU-gebundene Performanzverluste entstanden wären. Die Jenkinsfiles für den zweiten Versuch sind ebenfalls im Anhang A.4.4 zu finden. Eine kurze Zusammenfassung der allgemeinen Konfigurationen ist in der folgenden Tabelle 4.2 dargestellt:

Pipeline	Steps	Worker$_P$ (%)	Worker$_P$ (Abs.)	Operations
B1.1	4	50.00%	2	1000000
B1.2	6	66.66%	4	1500000
B1.3	8	75.00%	6	2000000
B1.4	10	80.00%	8	2500000
B2.1	6	33.33%	2	2000000
B2.2	8	50.00%	4	2500000
B2.3	10	60.00%	6	3000000
B2.4	12	66.66%	8	3500000
B3.1	8	25.00%	2	2500000
B3.2	10	40.00%	4	3000000
B3.3	12	50.00%	6	3500000
B3.4	14	57.14%	8	4000000

Tabelle 4.2: Zusammenfassung der Konfigurationen für Versuch 2
(Worker$_P$ = parallele Worker, % = prozentual, Abs. = absolut)

Darüber hinaus soll auch hier durch eine Abbildung weiter verdeutlicht werden, wie die Konfigurationen der Pipelines in Versuch 2 aufgebaut sind. Anders als im ersten Versuch kann hier gesehen werden, dass die Konfigurationen von links nach rechts die Anzahl der parallelen Worker erhöhen und gleichzeitig die Anzahl der insgesamt zu berechnenden Operationen pro Pipeline steigt. Die gelb hinterlegten Worker sollen verdeutlichen, dass diese im Vergleich zur vorherigen Konfiguration hinzugekommen sind.

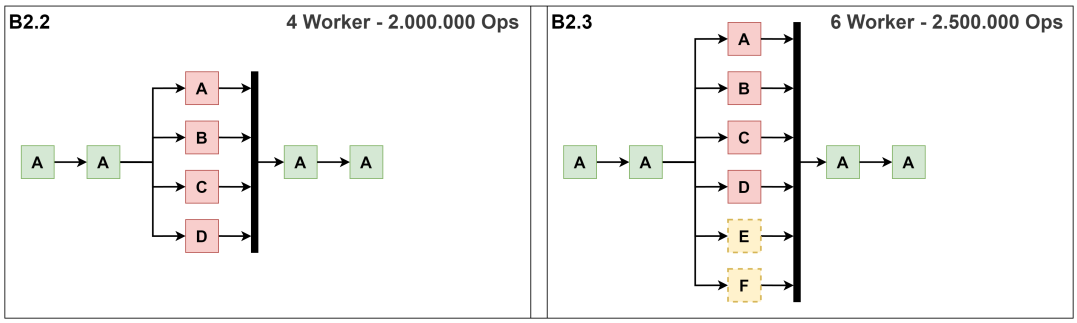

Abbildung 4.5: Versuch 2 – beispielhafte Konfiguration (eigene Darstellung)

4.2.3 Aufbau – Versuch 3

Im dritten Versuch wurde anders als in den ersten beiden Versuchen ein reales Softwareprojekt verwendet, um realistischere Arbeitslasten innerhalb der Pipeline zu erzeugen. Dies ist sinnvoll, da in einem realen Anwendungsfall nie alle Schritte einer Pipeline gleich CPU-intensiv sind. Konkret wurde als Softwareprojekt auf das Demonstrationsprojekt „Spring PetClinic“ [33] zurückgegriffen. Dieses Projekt ist ein Beispielprojekt des weit verbreiteten Spring-Frameworks und wird von der Spring-Community als Beispiel für die Entwicklung von Spring-Anwendungen als Open-Source-Projekt bereitgestellt. So bringt das Projekt bereits eine gewisse Zahl an Unit- und Integrationstests mit und greift auf unterschiedliche Softwareabhängigkeiten zurück. Für die Verwendung im Rahmen dieser Arbeit wurde ein Fork von diesem Projekt erzeugt und in einem eigenen Repository abgelegt. Dieser Fork basiert auf dem Commit mit dem Hash „5167226“. Zur Integration in die CI/CD-Umgebung wurden zwei Branches mit jeweils unterschiedlichen Jenkinsfiles erzeugt. Ein Branch bildet eine seriell ablaufende Pipeline ab, während der andere Branch eine Pipeline mit parallelen Schritten darstellt. Die konkreten Werte der Pipeline sind zur Vereinheitlichung in der folgenden Tabelle 4.3 zusammengefasst.

Pipeline	**Steps**	**Worker$_P$ (%)**	**Worker$_P$ (Abs.)**	**Operations**
Seriell	12	0%	0	*mixed tasks*
Parallel	12	50 - 70%	*not applicable*	*mixed tasks*

Tabelle 4.3: Zusammenfassung der Konfigurationen für Versuch 3
(Worker$_P$ = parallele Worker, % = prozentual, Abs. = absolut)

Darüber hinaus wurden die Konfigurationsdateien für das im Projekt verwendete „Checkstyle“-Plug-in angepasst, um einige Dateien innerhalb der CI/CD-Umgebung nicht in die Prüfung des Plug-ins einzubeziehen. Der Grund ist, dass dies zu unerwünschten Abbrüchen geführt hätte. Die Jenkinsfiles der beiden Konfigurationen sind im Anhang A.4.5 zu finden. Im Rahmen beider Jenkins-Pipelines wurde die Software zuerst aus dem Repository geklont, kompiliert und in einem lokalen Docker-Container eingebunden. Dann wurden sowohl Unit- als auch Integrationstests sowie SCA-, SAST- und DAST-Scans durchgeführt. Abschließend wurde das Abbild des lokalen Test-Containers in ein Release-Abbild überführt und in einer Container-Registry hochgeladen. Zuletzt wurden in beiden Fällen die erzeugten Artefakte der Pipeline aufgeräumt. Für einen besseren Überblick über die jeweiligen Abläufe kann die folgende Abbildung 4.6 betrachtet werden:

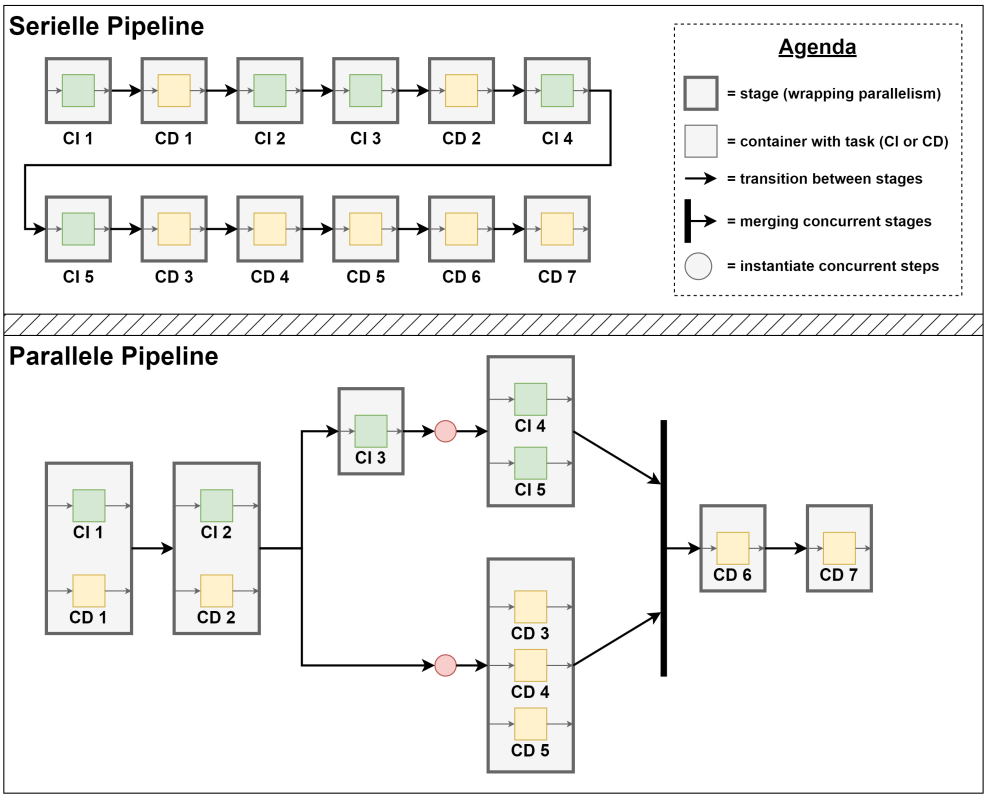

Abbildung 4.6: Versuch 3 – Aufbau serielle und parallele Pipeline (eigene Darstellung)

Die Abbildung zeigt, dass die Pipelines aus Schritten mit Aufgaben der „Kategorien“ Continuous Integration und Continuous Delivery bestehen. Während die serielle Pipeline die Schritte nacheinander abarbeitet, werden in der parallelen Pipeline die Schritte in zwei Gruppen aufgeteilt und parallel abgearbeitet. So können die Schritte der jeweiligen Kategorien unabhängig voneinander ausgeführt werden. Bei der seriellen Pipeline ist außerdem wichtig zu beachten, dass die Reihenfolge der CI- und CD-Aufgaben nicht relevant ist, solange sie innerhalb ihrer „Kategorie“ im zeitlichen Verlauf immer um eins inkrementiert werden. So kann beispielsweise der zweite CD-Schritt auch mit dem vierten CI-Schritt getauscht werden. Darüber hinaus sind in der parallelen Pipeline zwei rot eingezeichnete Kreise erkennbar, welche den Initialisierungsschritt der folgenden Stage mit parallelen Schritten darstellen sollen. Dieser Schritt enthält keine eigentliche Arbeitslast, muss aus Prozesssicht jedoch dargestellt werden, da auch die Initialisierung von Ressourcen selbst eine gewisse Zeit in Anspruch nimmt und explizit in der Pipeline definiert werden muss.

4.3 Erwartungen

Für die Ergebnisse der folgenden Versuche wird erwartet, dass die Gesetze von Amdahl und Gustafson grundsätzlich auf CI/CD-Prozesse anwendbar sind und somit eine Optimierung gegenüber rein seriellen Prozessen möglich ist. Es wird davon ausgegangen, dass dies bereits bei kleiner skalierten Hardwarekonfigurationen zu erkennen ist und eine signifikante Verbesserung erzielt werden kann. Darüber hinaus sollte neben der Verringerung der Durchlaufzeit auch eine Verbesserung der Effizienz der Prozesse erkennbar sein. Dies sollte sich in einer gleichmäßigeren Nutzung der verfügbaren Ressourcen und geringeren Leerlaufzeiten zeigen.

4.3.1 Erwartung – Versuch 1

Im ersten Versuch wird erwartet, dass durch die Inkrementierung des parallelen Anteils an Schritten innerhalb einer Pipeline mit fester Anzahl der Gesamtschritte die Durchlaufzeit der Pipeline sinkt. Dabei sollte die Durchlaufzeit in den Konfigurationen mit mehr parallelen Schritten im Vergleich zu den Konfigurationen mit weniger parallelen Schritten signifikant geringer sein. Außerdem sollte die Beschleunigungsrate besonders dann am höchsten innerhalb einer Jenkinsfile-Konfiguration sein, wenn pro parallelem Schritt ein dedizierter Worker eingesetzt wird. Die Effizienz der Prozesse sollte darüber hinaus besonders dann steigen, wenn die Anzahl der Worker im Verhältnis zur Anzahl der parallelen Schritte so skaliert wird, dass eine möglichst gleichmäßig hohe CPU-Auslastung erreicht wird, ohne die verfügbaren Ressourcen zu überlasten. Entsprechend sollte die Beschleunigungsrate bei Annäherung an die jeweilige Maximalkonfiguration abnehmen, ähnlich wie anhand der Formel von Amdahl in Abschnitt 2.4.1 beschrieben.

4.3.2 Erwartung – Versuch 2

Hinsichtlich des zweiten Versuchs wird erwartet, dass durch das gleichzeitige Inkrementieren von parallelen Workern sowie der zu verarbeitenden Problemgröße, also der Anzahl der Operationen, bei fester Anzahl der seriellen Schritte, eine gleichbleibende Durchlaufzeit der Pipeline erreicht werden kann. Demnach sollte eine größere Arbeitslast in derselben Zeit verarbeitet werden, wenn die Anzahl der parallelen Worker dazu proportional erhöht wird. Außerdem sollte hier die Beschleunigungsrate besonders abhängig von der Zahl der Worker und der anfallenden Problemgröße sein. Entsprechend wird erwartet,

dass die Beschleunigungsrate, wie in Gustafsons Gesetz in Abschnitt 2.4.1 beschrieben, linear mit der Anzahl der Worker steigt.

4.3.3 Erwartung – Versuch 3

Im dritten Versuch wird erwartet, dass durch die parallele Ausführung der realen Arbeitslasten der CI/CD-Pipeline ähnlich wie in den vorherigen Versuchen eine signifikante Verbesserung der Durchlaufzeit erreicht werden kann. Die Auslastung der CPU sollte darüber hinaus gleichmäßiger sein als in der seriellen Variante der Pipeline. Es wird darüber hinaus erwartet, dass die Beschleunigungsrate nicht so hoch abschneidet wie in den vorherigen Versuchen, da die Arbeitslasten keine optimalen Bedingungen für die Anwendung der Gesetze von Amdahl und Gustafson bieten. Dennoch sollte insgesamt eine Verbesserung der Effizienz und Durchlaufzeit erkennbar sein, darüber hinaus eine Tendenz zur Optimierung der Konfiguration.

4.4 Auswertung der Versuche

Zur Auswertung der Versuche wurden die gemessenen Daten der unterschiedlichen Systemmetriken mithilfe der Software Grafana zuerst über die Web-Oberfläche gesichtet. Anschließend wurden die mithilfe der Abfragesprache PromQL erstellten Datensätze im CSV-Format exportiert und für die weitere Auswertung in Python-Skripten aufbereitet. Primär wurden die Bibliotheken „Pandas“, „Matplotlib“, „Numpy“ und „Scipy“ verwendet, um die Daten zu analysieren und grafisch darzustellen. Die Ergebnisse der Auswertung werden im Folgenden für jeden Versuch einzeln erläutert. Die Graphen sind in englischer Sprache beschriftet.

4.4.1 Auswertung – Versuch 1

Die folgende Abbildung 4.7 zeigt die Beschleunigungsrate der verschiedenen Konfigurationen der Pipelines aus Versuch 1. Die jeweiligen Konfigurationen wurden durch die durchgängigen farbigen Linien dargestellt. Die entsprechenden theoretischen Beschleunigungsraten nach Amdahl sind hingegen in Grau gehalten. Darüber hinaus wurde eine logarithmische Regression durch die Punkte der jeweils gemessenen Daten berechnet und durch die farbig gepunktete Linie verdeutlicht.

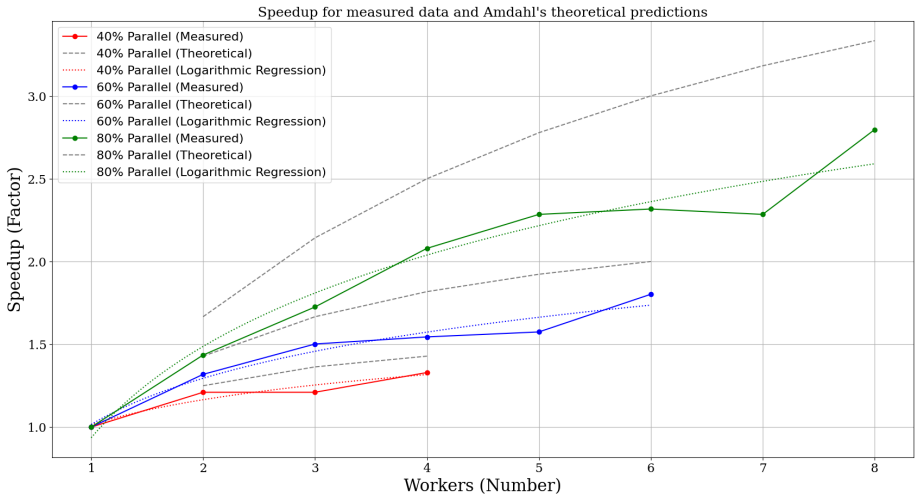

Abbildung 4.7: Versuch 1 – Auswertung des Speed-ups nach Amdahl

Bei der Betrachtung der Graphen fällt zunächst auf, dass die gemessenen Beschleunigungsraten keine konstante Kurve wie bei den theoretischen Werten von Amdahl bilden. Dies kann an Messfehlern liegen, jedoch fällt deutlich auf, dass sich bei einer gewissen Anzahl an Workern bei jeder der Konfigurationen eine Art Plateau bildet, indem die Beschleunigungsrate nicht signifikant zunimmt. Interessant an diesen Plateaus ist, dass sie insbesondere bei einer Workeranzahl entstehen, bei der die Anzahl der parallelen Schritte nicht gerade durch die Anzahl der Worker teilbar ist. Dies könnte darauf hindeuten, dass die CPU an solchen Stellen auf einzelne Berechnungen warten muss, da sie nicht ausreichend Ressourcen für weitere parallele Berechnungen zur Verfügung hat. Betrachtet man hier insbesondere die Konfiguration mit 80 % parallelem Anteil, so scheint es, als wären Konfigurationen mit einer gerade teilbaren Anzahl an parallelen Schritten und Workern effizienter als solche, die nicht gerade teilbar sind. Folgt man dieser Argumentation, könnte ein Zusammenhang außerdem daher entstehen, dass die Worker kein Scheduling durchführen, sondern jeder Prozess einem Worker statisch zugewiesen werden muss. Neben dieser Tatsache lässt sich durch die eingezeichnete Regression dennoch zeigen, dass die Entwicklung der Beschleunigungsraten demselben Schema folgt wie bei Amdahl. Jedoch kann man auch sehen, dass es im Verlauf zu signifikanten Performanzverlusten kommt. So ist die Beschleunigung bei beispielsweise der Konfiguration mit einem Anteil von 60 % Parallelisierung näher an den Werten der theoretischen Beschleunigungsrate von 40 % Parallelisierung. Eine derartige Abweichung könnte dadurch

entstehen, dass die Container für die Worker bei jedem Durchlauf erst dynamisch erstellt werden mussten. Derartige Managementprozesse benötigen Zeit und Ressourcen, die dann nicht für die eigentliche Berechnung genutzt werden können.

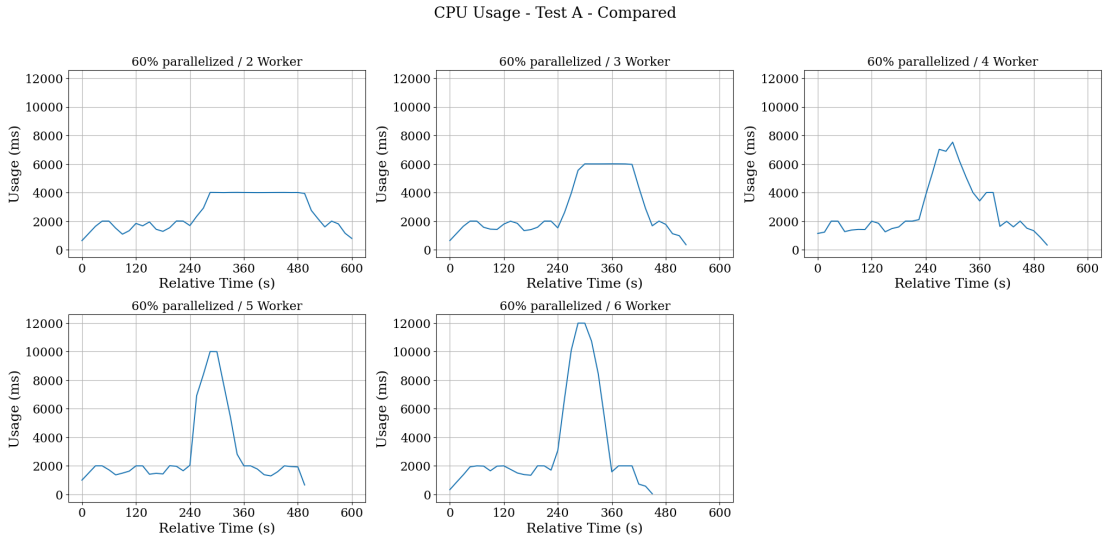

Abbildung 4.8: Versuch 1 – CPU-Auslastung bei 60 % Parallelisierung

In Abbildung 4.8 sind beispielhaft die Werte der CPU-Auslastung für die Pipeline-Konfiguration mit 60 % parallelem Anteil zu sehen. Diese Werte zeigen zunächst für jede Worker-Anzahl eine Aufteilung der CPU-Auslastung auf einem niedrigeren und einem höheren Level. Naheliegend ist hier, dass der schwankende Wert auf geringem Level den seriellen Anteil darstellt, während der Ausschlag auf eine höhere Auslastung den parallelen Anteil darstellt. Diese Annahme lässt sich damit stärken, dass der maximale Ausschlag jeweils bei der Summe der maximal zur Verfügung gestellten CPU-Ressourcen aller parallelen Worker-Container liegt. So ist bei der Konfiguration mit zwei Workern bei 60 % Parallelisierung ein Ausschlag von vier Sekunden zu sehen, was der Summe von zwei Containern mit einer maximalen Zuweisung von zwei Sekunden CPU-Zeit entspricht. Die Messung in Zeiteinheiten ist hierbei eine typische Darstellungsweise für Kubernetes. Sie wird darüber hinaus oft in Cloud-Umgebungen mit hohem Verteilungsgrad eingesetzt. Auch zu sehen ist in dieser Abbildung, dass die CPU-Auslastung bei einer höheren Anzahl an Workern entsprechend hoch ist. Die Dauer dieser Ausschläge wird jedoch bei steigender Anzahl an Workern kürzer. Dies könnte darauf hindeuten, dass die CPU-Auslastung bei einer höheren Anzahl an Workern zwar höher ist, jedoch die Berechnungen schneller abgeschlossen werden.

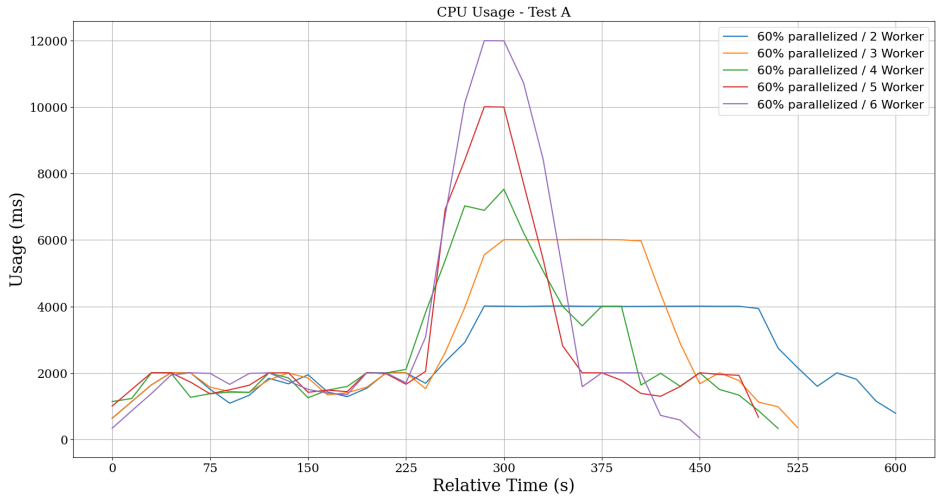

Abbildung 4.9: Versuch 1 – CPU Auslastung bei 60 % Parallelisierung im Vergleich

In Abbildung 4.9 kann die zuvor getroffene Annahme bestätigt werden, indem die einzelnen Verläufe zeitlich relativ zueinander übereinandergelegt dargestellt werden. Hier ist zu sehen, dass die CPU-Auslastung bei einer höheren Anzahl an Workern zwar für einen kurzen Moment höher ist, jedoch die Berechnungen auch zunehmend schneller abgeschlossen werden. Betrachtet man allerdings die Durchläufe mit drei, vier und fünf Workern, so ist zu sehen, dass es nur eine geringe Verbesserung der Durchlaufzeit gibt. Sollte demnach eine optimale Parallelisierung nicht erreicht werden können, so ist es nicht unbedingt sinnvoll, die Anzahl der Worker zu erhöhen. Der Grund liegt darin, dass dies zu einer ineffizienten Nutzung der Ressourcen führen kann.

4.4.2 Auswertung – Versuch 2

In der folgenden Abbildung 4.10 wurden die Beschleunigungsraten der verschiedenen Konfigurationen der Pipelines aus Versuch 2 nach Gustafsons Gesetz farblich dargestellt. Darüber hinaus wurden die theoretischen Beschleunigungsraten nach Gustafson für 20 bis 70 Prozent seriellen Anteil, mit einer Schrittweite von 10 Prozent, in grau abgestufter Farbe dargestellt.

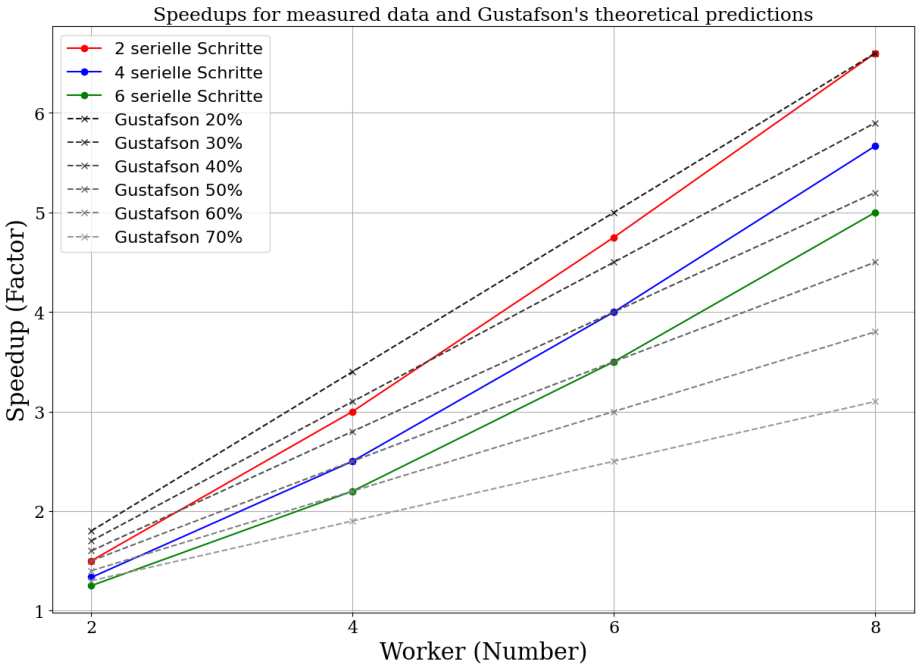

Abbildung 4.10: Versuch 2 – Auswertung des Speed-ups nach Gustafson

Aus dem Graphen lässt sich entnehmen, dass die Konfigurationen des Versuchs trotz der Tatsache, dass sie keinen prozentual konstant seriellen Anteil haben, dennoch der Gesetzmäßigkeit von Gustafson folgen. Dies lässt sich anhand der Schnittpunkte mit den theoretischen Werten erkennen. Bei den konfigurierten Pipelines ist die Beschleunigungsrate deshalb nicht linear, da mit einem festen Wert serieller Schritte in der Pipeline, anstelle von prozentualen Anteilen, gearbeitet wurde. Eine konstante prozentuale Verteilung hätte zu viele Ressourcen erfordert und wäre daher nicht umsetzbar gewesen.

In Abbildung 4.11 ist beispielhaft die CPU-Auslastung für die Pipeline-Konfiguration mit sechs seriellen Schritten zu sehen. Hier ist zu erkennen, dass die CPU-Auslastung bei einer höheren Anzahl an parallelen Schritten entsprechend höher ist. So ist auch hier wie in Versuch 1 zu sehen, dass das Maximum der jeweiligen Ausschläge bei der Summe der maximal zur Verfügung gestellten CPU-Ressourcen aller parallelen Worker liegt. Interessant ist außerdem, dass die Gesamtdauer aller Verläufe ähnlich lang ist. Dies könnte insbesondere im Zusammenhang mit Abbildung 4.10 darauf hindeuten, dass die Beschleunigungsraten sich wie von Gustafson beschrieben verhalten und linear mit der Anzahl der Worker steigen. Für eine genauere Analyse ist es auch hier sinnvoll, die Durchläufe im direkten Vergleich zu betrachten.

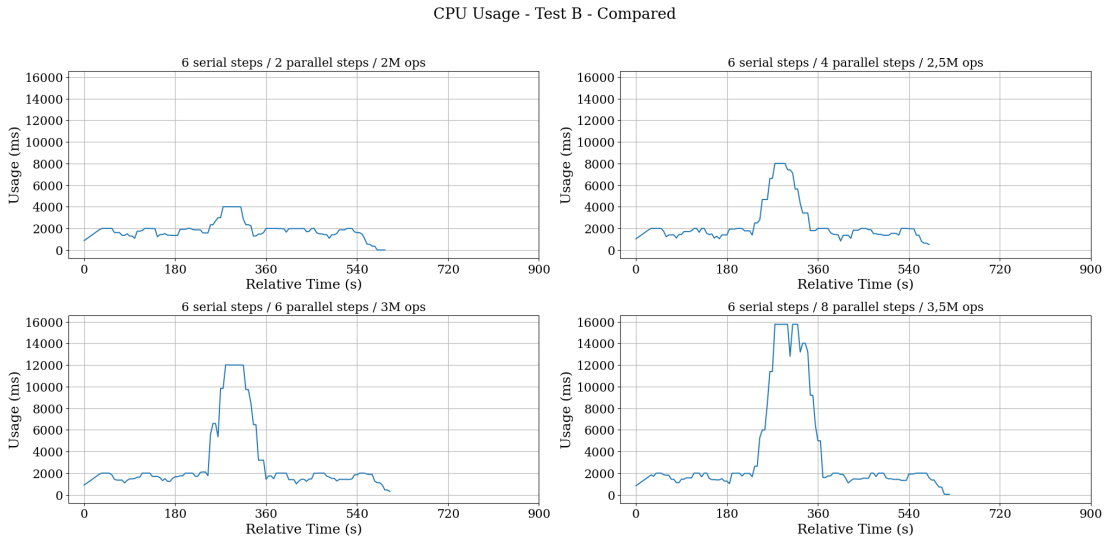

Abbildung 4.11: Versuch 2 – CPU-Auslastung bei 6 seriellen Schritten

Entsprechend werden in Abbildung 4.12 die Durchläufe der CPU-Auslastung bei sechs seriellen Schritten zeitlich relativ zueinander übereinandergelegt dargestellt. Hier ist zu sehen, dass es zwar eine ähnlich lange Durchlaufzeit gibt, jedoch die Dauer des parallelisierten Teils bei steigender Worker-Anzahl zunimmt. Auch ist erkennbar, dass sich dies zu einem gewissen Teil auf die Durchlaufzeit auszuwirken scheint und die Konfigurationen mit mehr Workern demnach etwas länger benötigen, um die Berechnungen abzuschließen. Auch dies könnte wie im ersten Versuch darauf hinweisen, dass die Managementprozesse der Container bei steigender Anzahl an Workern zu einer ineffizienten Nutzung der Ressourcen führen.

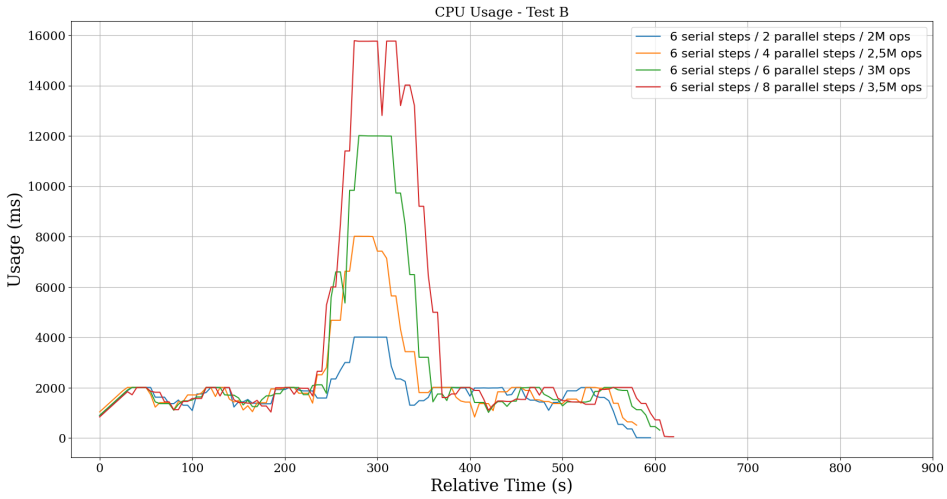

Abbildung 4.12: Versuch 2 – CPU-Auslastung bei 6 seriellen Schritten im Vergleich

Um die Zunahme der Durchlaufzeit bei steigender Anzahl an Workern zu verdeutlichen, wurden in Abbildung 4.13 die Durchlaufzeiten der verschiedenen Konfigurationen der Pipelines aus Versuch 2 dargestellt und es wurde eine Trendlinie mithilfe einer linearen Regression eingezeichnet.

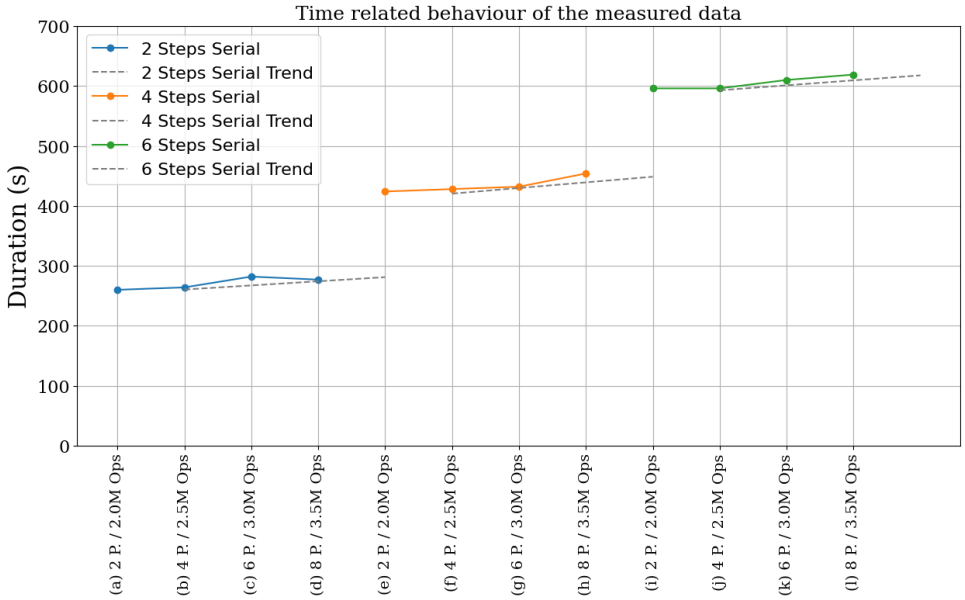

Abbildung 4.13: Versuch 2 – Auswertung des Zeitverhaltens der Konfigurationen

Bei Betrachtung der Graphen fällt auf, dass unabhängig vom seriellen Pipeline-Anteil die Durchlaufzeiten bei steigender Anzahl an Workern zunehmen. In der Konfiguration mit zwei seriellen Schritten ist jedoch auch zu erkennen, dass diese Zunahme nicht immer stattfinden muss. Auch in den Werten der Konfiguration mit vier und sechs seriellen Schritten können diese Beobachtungen gemacht werden. Dies kann so interpretiert werden, dass die Zunahme der Durchlaufzeit abhängig von der Auslastung des Gesamtsystems und den darin zusätzlich laufenden Prozessen ist.

4.4.3 Auswertung – Versuch 3

In Abbildung 4.14 ist die CPU-Auslastung der seriellen und parallelen Pipeline des dritten Versuchs dargestellt.

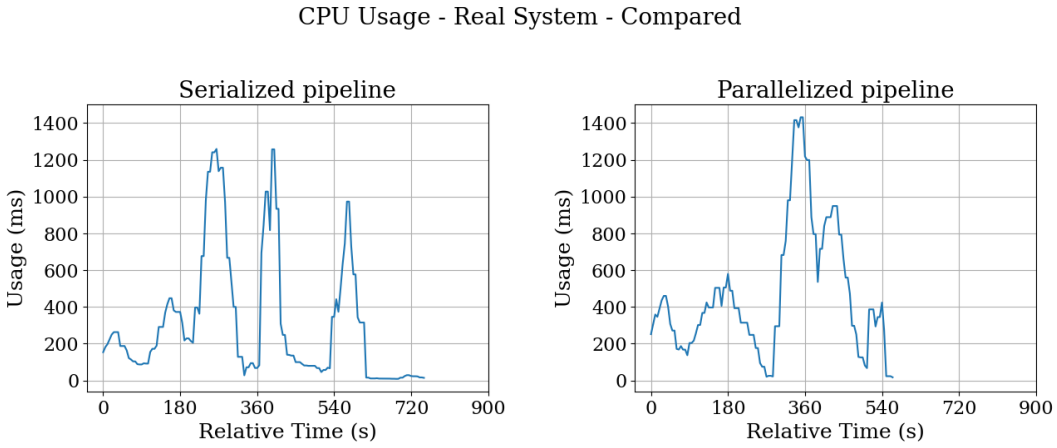

Abbildung 4.14: Versuch 3 – CPU-Auslastung – seriell vs. parallel

Bei der Betrachtung des Diagramms kann man zunächst erkennen, dass die Auslastung der CPU der parallelen Konfiguration einen höheren Maximalausschlag hat als die der seriellen Konfiguration. Dies ist darauf zurückzuführen, dass durch die parallele Ausführung der Schritte mehr Ressourcen gleichzeitig genutzt werden können. Weiter kann man erkennen, dass die serielle Konfiguration ebenfalls Ausschläge mit höherer CPU-Auslastung aufweist. Dies könnte damit zusammenhängen, dass der serielle und parallele Prozess dieselben Ressourcen zur Verfügung gestellt bekommen haben, wobei bei der parallelisierten Konfiguration die Ressourcen auf zwei Container gewichtet aufgeteilt wurden. Dementsprechend könnten die Ausschläge der seriellen Pipeline darauf hindeuten, dass einige der Schritte innerhalb des Worker-Containers die größere Anzahl

an CPU-Ressourcen ausnutzen konnten. Dies kann beispielsweise durch die parallele Ausführung von mehreren Prozessen oder Threads innerhalb des Containers durch eine Software geschehen. Somit ist der Pipeline-Prozess zwar seriell, aber eine Parallelisierung innerhalb der einzelnen Schritte denkbar. Selbiges würde auch für die parallele Pipeline gelten, bei der jedoch auch auf Ebene des Pipeline-Prozesses eine parallele Ausführung stattfindet. Hier fallen entsprechend parallele Ausführungen innerhalb der Container weniger ins Gewicht, da die Ressourcen auf mehrere Container aufgeteilt wurden und somit die Auslastung übergeordnet über die Prozessparallelisierung der Pipeline verteilt wird. Betrachtet man das Diagramm etwa bei Sekunde 1050 bis 1250, so ist eine längere Phase mit höherer CPU-Auslastung zu erkennen. Diese könnte zeitlich zu der parallelen Ausführung der SCA-, SAST- und DAST-Scans gehören. Bei der seriellen Pipeline scheinen diese Scans deutlich verteilter für hohe Lasten zu sorgen, da die CPU-Auslastung dementsprechend stark verteilt ist.

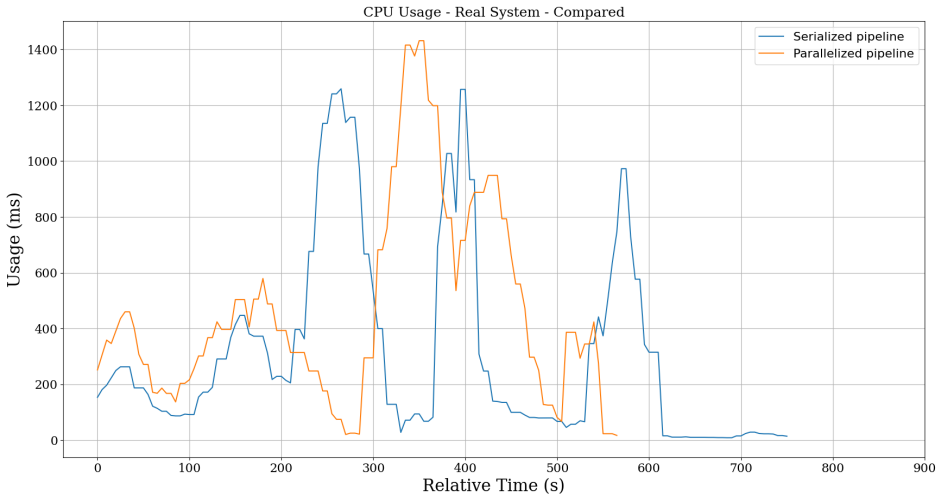

Abbildung 4.15: Versuch 3 – CPU-Auslastung seriell vs. parallel im Vergleich

Betrachtet man in Abbildung 4.15 die CPU-Auslastung der seriellen und parallelen Pipeline des dritten Versuchs im direkten Vergleich, so ist außerdem zu erkennen, dass der parallelisierte Prozess eine signifikant kürzere Durchlaufzeit erreicht. Darüber hinaus ist zu sehen, dass die CPU-Auslastung überwiegend leicht erhöht gegenüber der seriellen Pipeline ist. Auch sind die Ausschläge der parallelen Pipeline konstanter und anhaltender als die der seriellen Pipeline. Dies deutet darauf hin, dass durch eine sinnvollere Ver-

teilung der Ressourcen auf mehrere Container eine gleichmäßigere Auslastung erreicht werden konnte.

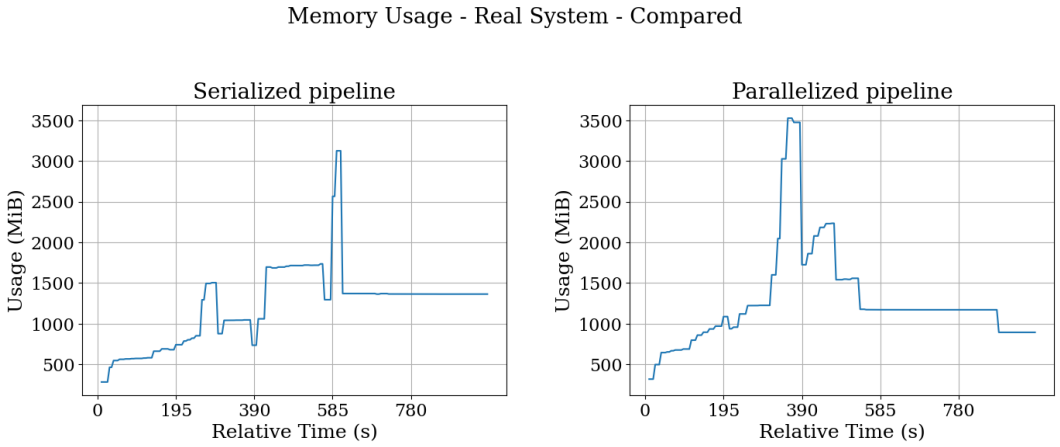

Abbildung 4.16: Versuch 3 – RAM-Auslastung – seriell vs. parallel

In Abbildung 4.16 ist die Auslastung des Arbeitsspeichers der seriellen und parallelen Pipeline des dritten Versuchs dargestellt. Hier ist zu erkennen, dass die parallele Pipeline einen höheren Maximalausschlag hat als die serielle Pipeline. Dies ist darauf zurückzuführen, dass durch die parallele Ausführung der Schritte mehr Ressourcen gleichzeitig genutzt werden. Weiter ist in beiden Verläufen zu erkennen, dass die Auslastung des Arbeitsspeichers über die Zeit hinweg ansteigt und zum Ende des Prozesses wieder leicht abfällt. Dies könnte darauf hindeuten, dass die Arbeitsspeicherressourcen der Worker-Container nach Beendigung einiger intensiver Berechnungen wieder freigegeben werden. Bei einem direkten Vergleich mit Abbildung 4.14 ist ferner zu sehen, dass die Auslastung des Arbeitsspeichers besonders dann ansteigt, wenn auch die CPU-Auslastung ansteigt. Dies ist ein Indiz dafür, dass die zu verarbeitenden Aufgaben gemischte Ressourcenanforderungen haben, was wiederum auf eine realistischere Arbeitslast hindeutet.

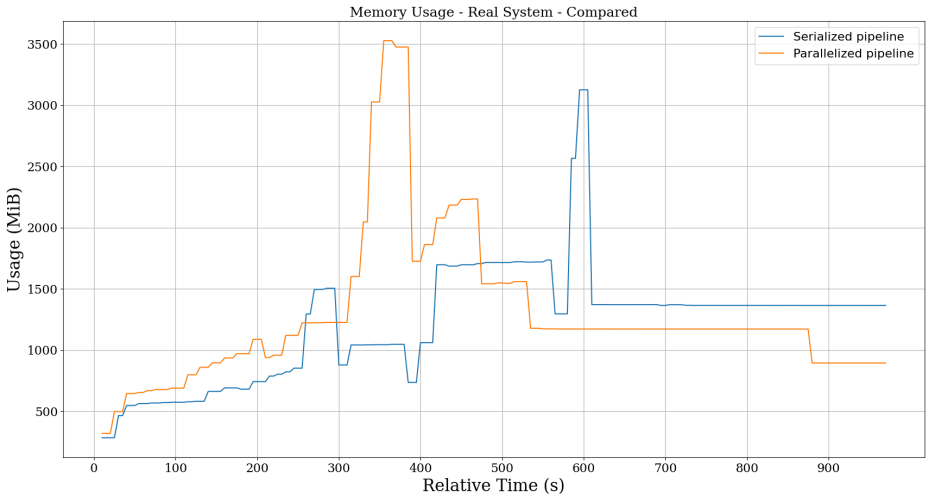

Abbildung 4.17: Versuch 3 – RAM-Auslastung seriell vs. parallel im Vergleich

Beim Vergleich der Arbeitsspeicher-Auslastung ist in Abbildung 4.17 zu erkennen, dass die parallele Pipeline ebenfalls, wie in den vorherigen Vergleichsdiagrammen, eine geringe Durchlaufzeit erzielt. Während die parallele Pipeline bei etwa Sekunde 530 das Ende erreicht hat, ist die serielle Pipeline erst zwischen Sekunde 600 und 700 abgeschlossen. Auch hier ist zu erkennen, dass die parallele Pipeline eine meist höhere Auslastung des Arbeitsspeichers hat als die serielle Pipeline. Erst zum Ende des Prozesses ist zu sehen, dass die Auslastung des Arbeitsspeichers der parallelen Pipeline abfällt, während die der seriellen Pipeline weiterhin konstant hoch bleibt. Dies lässt sich durch die parallele Ausführung der Schritte erklären, die eine schnellere Auslastung der Ressourcen erzeugt und somit auch schneller wieder Ressourcen freigeben kann.

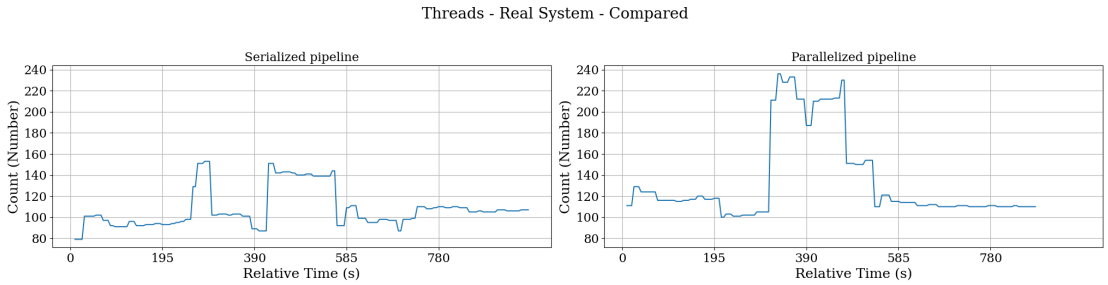

Abbildung 4.18: Versuch 3 – Anzahl der Threads – seriell vs. parallel

Betrachtet man zusätzlich zur CPU- und Arbeitsspeicherauslastung die Anzahl der Threads, so ist in Abbildung 4.18 zu erkennen, dass die parallele Pipeline eine signifikant höhere Anzahl an Threads verwendet als die serielle Pipeline. Dies ist darauf zurückzuführen, dass die parallele Pipeline deutlich mehr parallele Prozesse erzeugt und damit einhergehend auch mehr Threads.

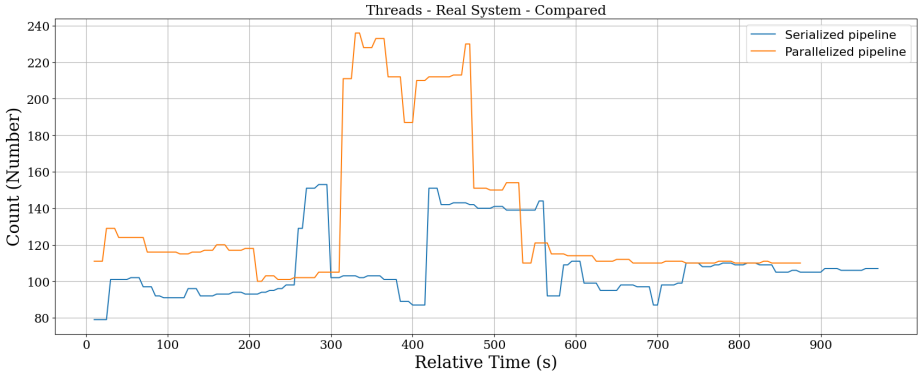

Abbildung 4.19: Versuch 3 – Anzahl der Threads – seriell vs. parallel im Vergleich

Legt man auch hier beide Verläufe im direkten Vergleich übereinander, so ist in Abbildung 4.19 zu erkennen, dass die parallele Pipeline in fast allen Phasen der Pipeline eine deutlich höhere Threadanzahl aufweist. Vergleicht man die beiden höchsten Ausschläge miteinander, so liegt die Anzahl der Threads bei der parallelen Pipeline etwa bei 150 % der Anzahl der Threads in der seriellen Pipeline.

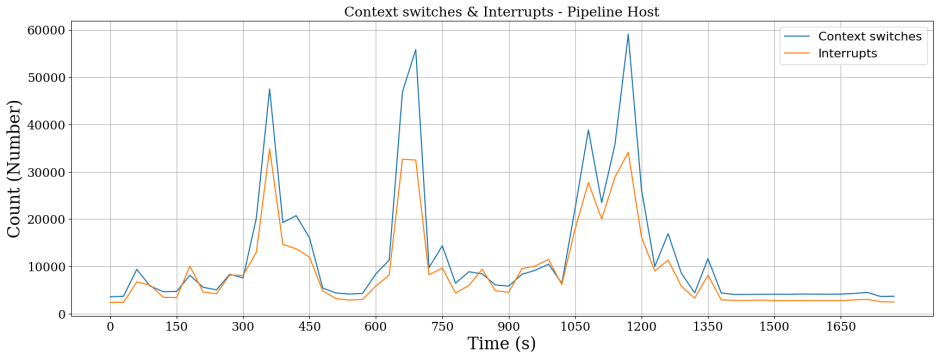

Abbildung 4.20: Versuch 3 – Context-Switches und Interrupts

Zuletzt ist in Abbildung 4.20 die Anzahl der Context-Switches und Interrupts der seriellen und parallelen Pipeline des dritten Versuchs dargestellt. Bei dieser Metrik wurde

das Hostsystem gemessen und nicht die Worker-Container. Dennoch kann anhand der Zeitachse erkannt werden, dass die zuerst gestartete serielle Pipeline zwei Ausschläge mit einer höheren Anzahl an Context-Switches und Interrupts aufweist und die parallele Pipeline einen länger anhaltenden Ausschlag. Bei genauerer Betrachtung kann auch bei dem Ausschlag der parallelen Pipeline ein doppelter Ausschlag erkannt werden. Jedoch gehen die Ausschläge ineinander über und sind nicht so deutlich voneinander abgegrenzt wie bei der seriellen Pipeline. Dies deutet darauf hin, dass die parallele Pipeline zeitlich gesehen nur zu einem Zeitpunkt eine höhere Anzahl an Context-Switches und Interrupts erzeugt und somit das Verhalten besser kalkulierbar ist sowie Ressourcen effizienter genutzt werden können.

4.5 Gesamtauswertung

Anhand der vorangegangenen Auswertung der Versuche können konkrete Empfehlungen für die Konfiguration von CI/CD-Pipelines abgeleitet werden. Zunächst kann anhand der Ergebnisse von Versuch 1 festgehalten werden, dass bei ausreichender Anzahl an Workern eine Parallelisierung der Pipeline-Schritte sinnvoll sein kann, um geringere Durchlaufzeiten zu erreichen. Um sicherzustellen, dass die gewählte Konfiguration nicht, wie in Versuch 1 gezeigt, durch eine suboptimale Verteilung der Worker auf die parallelen Schritte weniger effizient wird, sollten Vergleiche mit unterschiedlichen Anzahlen an Workern durchgeführt werden. Optimal ist demnach eine Konfiguration, bei der entweder jeder Schritt einen eigenen Worker erhalten kann oder die Anzahl der parallelen Schritte gerade durch die Anzahl der Worker teilbar ist. Darüber hinaus kann festgehalten werden, dass der Umfang der zu verarbeitenden Schritte gut skaliert, solange zusätzliche Schritte durch weitere Worker berechnet werden. In allen Fällen, in denen weitere Worker hinzugenommen werden, muss allerdings immer damit gerechnet werden, dass auch zusätzliche Arbeitslasten für das Management der Worker-Container entstehen. Diese zusätzlichen Arbeitslasten benötigen ebenfalls Zeit zur Verarbeitung und können besonders bei einer großen Zahl von Workern negativ ins Gewicht fallen. Hier sollte bei der Implementierung neuer Pipelines getestet werden, ab wann die Effizienz nicht weiter zunimmt, sondern gegebenenfalls sogar abnimmt. Auch sollte bedacht werden, dass die Erhöhung der Worker-Anzahl auch zu einer höheren Auslastung aller Ressourcen im System führen kann. Dies bedeutet, dass neben der CPU auch ein höherer Bedarf an beispielsweise Arbeitsspeicher entsteht.

4.6 Bezug auf theoretische Grundlagen

In Kapitel 2 wurden neben den Konzepten der Continuous Integration und Delivery sowie der Funktionsweise von Prozessen und Threads unter anderem auch die Gesetzmäßigkeiten von Amdahl und Gustafson erläutert.

Ausgehend von den in diesem Kapitel durchgeführten Versuchen wurde angenommen, dass sich die warteschlangentheoretischen Gesetze von den hardwarenahen Prozessen, wie im Falle von CPUs, auch auf andere Prozesse, in diesem Fall CI/CD-Pipelines, anwenden lassen und eine Beschleunigung erzielen können.

Die Ergebnisse der Versuche konnten diese Annahmen anhand unterschiedlicher Daten aufzeigen und bestätigen. So war eine der Thesen, dass die Variablen Amdahls und Gustafsons für die Hardware-Prozessoren durch die Anzahl der Worker, also Rechenknoten in einem verteilten System, ersetzt werden können. Dies traf in den durchgeführten Versuchen zu, jedoch mit der besonderen Beobachtung, dass die Dauer der Managementprozesse für das Erstellen und Verwalten der Worker-Container einen, je nach Skalierung, relevanten Einfluss auf die Durchlaufzeit der Pipelines hat. Eine Begründung dafür, dass diese Managementprozesse einen relevanten zeitlichen Einfluss haben, lässt sich unter anderem mithilfe des Aufbaus der Container-Orchestrierung erklären. Demnach müssen in einer Orchestrierungsumgebung wie Kubernetes aus Abschnitt 3.1.2 die Pods und Container von einer Reihe an Komponenten über die entsprechende API, welche in diesem Fall durch das installierte Jenkins-Plug-in ausgeführt wurde, erstellt und verwaltet werden. Die daraus entstehenden Kommunikations- und Synchronisationsaufgaben benötigen dann entsprechende zusätzliche Zeit und Ressourcen.

Neben diesen Beobachtungen konnte im Rahmen der Versuche weiter beobachtet werden, dass eine optimale Parallelisierung besonders von der vertikalen und horizontalen Skalierung der Worker abhängt. So können Aufgaben nur bei genügenden Workern mit entsprechender Hardwareausstattung effizient parallelisiert werden. Zusätzlich ist aber auch deutlich geworden, dass die horizontale Skalierung bessere Ergebnisse erzielen konnte als die vertikale Skalierung. Besonders deutlich wurde dies im dritten Versuch, bei dem dieselben Ressourcen in der parallelisierten Pipeline auf zwei Worker aufgeteilt wurden. Hier konnten eine signifikante Verbesserung der Durchlaufzeit und eine gleichmäßigere Auslastung der Ressourcen erreicht werden.

4.7 Erweiterungen und Anpassungen

Um das Optimierungspotenzial parallelisierter Pipelines für die Bereitstellung containerisierter Anwendungen weiter zu untersuchen, sind einige Erweiterungen und Anpassungen der vorgestellten Versuche denkbar.

Zunächst wäre es sinnvoll, die Versuche auf einer größeren Anzahl von Workern durchzuführen. Hier könnten zwei Strategien verfolgt und ausgewertet werden. Zum einen könnte die Rechenleistung der CPU erhöht werden, um weitere Worker zu simulieren. Zum anderen wäre es aber noch interessanter, die Versuche über eine größere Anzahl von physischen Maschinen zu verteilen. Dies würde eine realistischere Simulation eines verteilten Systems ermöglichen und die Auswirkungen von Netzwerklatenzen und -auslastungen auf die Durchlaufzeit der Pipelines zusätzlich untersuchbar machen.

Auch könnten die Versuche um eine Analyse der Wirtschaftlichkeit der verschiedenen Konfigurationen erweitert werden. Hierzu könnte eine Kosten-Nutzen-Rechnung aufgestellt werden. Dies würde dabei helfen, noch bessere Aussagen über die Wirtschaftlichkeit unterschiedlicher Konfigurationen zu treffen und einen Punkt zu bestimmen, an dem die Hinzunahme weiterer Worker zur Erhöhung der Beschleunigungsrate unwirtschaftlich würde.

In ähnlicher Weise wäre auch die Messung des Energieverbrauchs interessant, um auch hier klarere Handlungsempfehlungen ableiten zu können und die Auswirkungen auf die Umwelt näher zu untersuchen.

Eine aus technischer Sicht interessante Erweiterung wäre auch die Implementation eines Scheduling-Algorithmus, der die Schritte der Pipeline dynamisch auf die Kubernetes-Worker verteilt. Dies könnte eine deutliche Verbesserung ermöglichen, betrachtet man die Auswirkungen von Scheduling auf der Ebene von Prozessen und Threads wie in Abschnitt 2.3.4 beschrieben.

Zuletzt könnte auch ein kombiniertes Experiment sinnvoll sein, um die Erkenntnisse dieser Arbeit mit denen aus Abschnitt 2.5 zu kombinieren. So könnten die Durchlaufzeiten noch weiter optimiert und die Ressourcennutzung könnte noch effizienter gestaltet werden.

5 Fazit

In diesem Kapitel soll ein übergeordnetes Fazit zu den Ergebnissen der Arbeit gezogen werden. Dabei soll die Fragestellung beantwortet und ein kritischer Blick auf die Ergebnisse geworfen werden.

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde nach Optimierungspotenzial für einen effizienten Bereitstellungsprozess für Software im Kontext containerisierter Systeme gesucht. Dazu wurde die Funktionsweise verschiedener Technologien und Konzepte erläutert und auf die theoretischen Grundlagen der Warteschlangentheorie eingegangen. Insbesondere wurden die Gesetze von Amdahl und Gustafson dargelegt und deren Anwendbarkeit im Kontext der automatisierten Bereitstellung von Software in CI/CD-Pipelines wurde untersucht. Aus den Ergebnissen dieser Untersuchungen konnte die Anwendbarkeit der theoretischen Grundlagen gezeigt und ein effizienter Bereitstellungsprozess, durch die Parallelisierung von Prozessen, entwickelt werden. Durch die Untersuchung der theoretischen Grundlagen wurde außerdem dargestellt, welche limitierenden Faktoren bei der Parallelisierung nach Amdahl und Gustafson auftreten können und welchen Ursprung diese haben. Anhand dessen konnte gezeigt werden, dass die Parallelisierung von Prozessen nicht immer zu einer effizienteren Bereitstellung führt und dass die Anwendung der Gesetze von Amdahl und Gustafson im Kontext der Bereitstellung von Software in CI/CD-Pipelines nur unter Berücksichtigung bestimmter Faktoren zu den gewünschten Ergebnissen führt.

5.2 Beantwortung der Fragestellung

Ziel der Arbeit war es, einen effizienten Bereitstellungsprozess für Software im Kontext containerisierter Systeme zu entwickeln. Mithilfe der theoretischen Grundlagen aus Ka-

pitel 2 und deren Untersuchung in der Evaluation in Kapitel 4 konnte gezeigt werden, auf welche Weise sich Prozessschritte innerhalb einer CI/CD-Pipeline parallelisieren lassen und wie sich dadurch die Performanz und Effizienz steigern lassen. Im Fokus der Optimierung standen vor allem Prozessabschnitte, welche für das Testen und die Sicherheit der Software verantwortlich sind. Allgemeiner konnten solche Prozesse gut parallelisiert werden, die möglichst unabhängig von anderen Prozessen ausführbar sind.

Die Ergebnisse der Evaluation zeigen, dass durch die Parallelisierung der Schritte einer CI/CD-Pipeline eine signifikante Verbesserung der Durchlaufzeit erreicht werden kann. Im dritten Versuch, in welchem die CI/CD-Pipeline eines realistischen Softwareprojekts optimiert wurde, wurde eine Gesamtverbesserung der Durchlaufzeit von etwa 22 % erreicht. Das entspricht in diesem Fall einer Beschleunigungsrate von etwa $1,283$. Das bedeutet, dass insbesondere die Laufzeiteffizienz gesteigert werden kann. Anhand des dritten Versuchs konnte jedoch auch gezeigt werden, dass die Zunahme der Laufzeiteffizienz mit einem zeitweise erhöhten Arbeitsspeicheraufwand einhergeht. Vergleicht man beispielsweise die jeweils höchsten Ausschläge, so lag der Bedarf der parallelen Pipeline bei etwa 12,837 % über dem Arbeitsspeicherbedarf der seriellen Pipeline. Über die Gesamtlaufzeit der beiden Pipelines hinweg betrachtet wurde hier jedoch auch eine Verringerung von etwa 10 % bei der parallelen Pipeline erreicht.

Anhand der Versuche, besonders Versuch 3, wurde somit ein Beispiel für die effiziente Gestaltung eines Bereitstellungsprozesses in einem containerisierten System aufgezeigt. Parallelisierung und auch Skalierung sind wesentliche Faktoren, mithilfe derer die Effizienz, mit besonderem Fokus auf der Laufzeit, gesteigert werden kann.

5.3 Kritische Betrachtung

Wie bereits in Abschnitt 4.7 aufgezeigt, wurden die Versuche in einer virtuellen, teils simulierten Umgebung durchgeführt. Dies bedeutet, dass das untersuchte Verhalten auf mehreren physischen Knoten möglicherweise anders ausfallen könnte. Darüber hinaus war das verwendete System auch mit einer begrenzten Anzahl an Ressourcen ausgestattet, was nur eine begrenzte Skalierung der Prozesse ermöglichte. Somit waren keine Rückschlüsse auf sich wiederholendes Verhalten bezüglich der in beispielsweise Versuch 1 4.4.1 gesehenen Plateaus, bei der Auswertung der Beschleunigungsraten, möglich. Eine Erweiterung der Versuche auf mehrere Knoten und eine höhere Anzahl an Ressourcen könnten somit weitere Erkenntnisse liefern, die in dieser Arbeit nicht erlangt wurden.

Weiter könnten die Ergebnisse der Arbeit auch dadurch beeinflusst worden sein, dass die Ausführung der Prozesse nur auf einem spezifischen Softwarestack erfolgte. Eine Erweiterung der Versuche auf ein Vergleichssystem könnte sinnvoll sein, um die Ergebnisse zu verifizieren und mögliche Anomalien zu identifizieren.

Literaturverzeichnis

[1] Alnafessah, Ahmad ; Gias, Alim U. ; Wang, Runan ; Zhu, Lulai ; Casale, Giuliano ; Filieri, Antonio: Quality-Aware DevOps Research: Where do we stand? In: *IEEE Access* 9 (2021), 1, S. 44476–44489. – URL https://doi.org/10.1109/access.2021.3064867

[2] Beheshtian, Mohammad J. ; Bavand, Amir H. ; Rigby, Peter C.: Software Batch Testing to Save Build Test Resources and to Reduce Feedback Time. In: *IEEE Transactions on Software Engineering* 48 (2022), Nr. 8, S. 2784–2801

[3] Bengel, Günther ; Baun, Christian ; Kunze, M. ; Stucky, Karl-Uwe: *Masterkurs Parallele und verteilte Systeme*. Springer Vieweg, 1 2015. – URL https://doi.org/10.1007/978-3-8348-2151-5

[4] Bezemer, Cor-Paul ; Eismann, Simon ; Ferme, Vincenzo ; Grohmann, Johannes ; Heinrich, Robert ; Jamshidi, Pooyan ; Shang, Weiyi ; Hoorn, André van ; Villavicencio, Monica ; Walter, Jürgen ; Willnecker, Felix: How is Performance Addressed in DevOps? In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. New York, NY, USA : Association for Computing Machinery, 2019 (ICPE '19), S. 45–50. – URL https://doi.org/10.1145/3297663.3309672. – ISBN 9781450362399

[5] Chen, Shiang-Jiun ; Pan, Yu-Chun ; Ma, Yi-Wei ; Chiang, Cheng-Mou: The Impact of the Practical Security Test during the Software Development Lifecycle. In: *2022 24th International Conference on Advanced Communication Technology (ICACT)*, 2022, S. 313–316

[6] Fallahzadeh, Emad ; Bavand, Amir H. ; Rigby, Peter C.: Accelerating Continuous Integration with Parallel Batch Testing. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA : Association for Computing

Machinery, 2023 (ESEC/FSE 2023), S. 55–67. – URL https://doi.org/10.1145/3611643.3616255. – ISBN 9798400703270

[7] Fitzgerald, Brian ; Stol, Klaas-Jan: Continuous software engineering: A roadmap and agenda. In: *Journal of Systems and Software* 123 (2017), 1, S. 176–189. – URL https://doi.org/10.1016/j.jss.2015.06.063

[8] Goll, Joachim ; Dausmann, Manfred: *Architektur- und Entwurfsmuster der Softwaretechnik*. Springer Vieweg, 1 2014. – URL https://doi.org/10.1007/978-3-658-05532-5

[9] Gustafson, John L.: Reevaluating Amdahl's law. In: *Communications of The ACM* 31 (1988), 5, Nr. 5, S. 532–533. – URL https://doi.org/10.1145/42411.42415

[10] Halstenberg, Jürgen ; Pfitzinger, Bernd ; Jestädt, Thomas: *DevOps*. Springer Vieweg, 1 2020. – URL https://doi.org/10.1007/978-3-658-31405-7

[11] Inc., Docker: *Docker overview*. Website. – URL https://docs.docker.com/get-started/overview/. – Accessed: 2024-03-05

[12] Jabbari, Ramtin ; Ali, Nauman bin ; Petersen, Kai ; Tanveer, Binish: What is DevOps? A Systematic Mapping Study on Definitions and Practices. (2016). – URL https://doi.org/10.1145/2962695.2962707. ISBN 9781450341349

[13] Jenkins: *Installing Jenkins*. – URL https://www.jenkins.io/doc/book/installing/kubernetes/. – Accessed: 2024-04-03

[14] Kreisa, John: *Docker Index: Dramatic Growth in Docker Usage Affirms the Continued Rising Power of Developers | Docker*. Website. 7 2020. – URL https://www.docker.com/blog/docker-index-dramatic-growth-in-docker-usage-affirms-the-continued-rising-power-of-developers/. – Accessed: 2024-03-05

[15] Limited, Red H.: *Was ist CI/CD? Konzepte und CI/CD Tools im Überblick*. 2 2023. – URL https://www.redhat.com/de/topics/devops/what-is-ci-cd. – Accessed: 2024-03-10

[16] Little, John D. C.: A Proof for the Queuing Formula: L= λ W. In: *Operations Research* 9 (1961), Nr. 3, S. 383–387. – URL http://www.jstor.org/stable/167570

[17] Marandi, Manohar ; Bertia, A. ; Silas, Salaja: Implementing and Automating Security Scanning to a DevSecOps CI/CD Pipeline. In: *2023 World Conference on Communication and Computing (WCONF)*, 2023, S. 1–6

[18] Morales, Jose A. ; Scanlon, Thomas P. ; Volkmann, Aaron ; Yankel, Joseph ; Yasar, Hasan: Security impacts of sub-optimal DevSecOps implementations in a highly regulated environment. In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. New York, NY, USA : Association for Computing Machinery, 2020 (ARES '20). – URL https://doi.org/10.1145/3407023.3409186. – ISBN 9781450388337

[19] Najafi, Armin ; Rigby, Peter C. ; Shang, Weiyi: Bisecting commits and modeling commit risk during testing. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA : Association for Computing Machinery, 2019 (ESEC/FSE 2019), S. 279–289. – URL https://doi.org/10.1145/3338906.3338944. – ISBN 9781450355728

[20] Patel, Ashish: *Kubernetes — Architecture and Cluster Components Overview | DevOps Mojo*. 1 2022. – URL https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd. – Accessed: 2024-03-07

[21] Prometheus ; Foundation, The L.: *Overview | Prometheus*. – URL https://prometheus.io/docs/introduction/overview/. – o.D. ; Accessed: 2024-03-12

[22] Rajapakse, Roshan N. ; Zahedi, Mansooreh ; Babar, Muhammad A.: An Empirical Analysis of Practitioners' Perspectives on Security Tool Integration into DevOps. In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. New York, NY, USA : Association for Computing Machinery, 2021 (ESEM '21). – URL https://doi.org/10.1145/3475716.3475776. – ISBN 9781450386654

[23] Rijn, Vincent van ; Rellermeyer, Jan S.: A fresh look at the architecture and performance of contemporary isolation platforms. In: *Proceedings of the 22nd International Middleware Conference*. New York, NY, USA : Association for Computing Machinery, 2021 (Middleware '21), S. 323–335. – URL https://doi.org/10.1145/3464298.3493404. – ISBN 9781450385343

[24] Saito, Hideto ; Lee, Hui-Chuan C. ; Wu, Cheng-Yang: *Devops with Kubernetes*. Packt Publishing, 10 2017. – URL https://www.packtpub.com/product/devops-with-kubernetes/9781788396646

[25] Sayfan, Gigi: *Mastering Kubernetes*. 3. Packt Publishing Ltd., 6 2020

[26] Sharma, Prateek ; Chaufournier, Lucas ; Shenoy, Prashant ; Tay, Y. C.: Containers and Virtual Machines at Scale: A Comparative Study. In: *Proceedings of the 17th International Middleware Conference*. New York, NY, USA : Association for Computing Machinery, 2016 (Middleware '16). – URL https://doi.org/10.1145/2988336.2988337. – ISBN 9781450343008

[27] Sirkemaa, Seppo: Key perspectives in Information Technology Infrastructure Management. In: *Journal of Advances in Information Technology* 10 (2019), 1, Nr. 3, S. 100–103. – URL https://doi.org/10.12720/jait.10.3.100-103

[28] Sommerville, Ian: *Modernes Software-Engineering*. 1. Pearson, 1 2020. – URL https://elibrary.pearson.de/book/99.150005/9783863268923

[29] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest*. 6. dpunkt.verlag GmbH, 6 2019

[30] Tanenbaum, Andrew S. ; Bos, Herbert: *Moderne Betriebssysteme*. 4. Pearson, 1 2016

[31] Tanenbaum, Andrew S. ; Van Steen, Maarten: *Distributed systems*. 4.01. Maarten Van Steen, 1 2023

[32] Velp, Guillaume E. de ; Rivière, Etienne ; Sadre, Ramin: Understanding the performance of container execution environments. In: *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*. New York, NY, USA : Association for Computing Machinery, 2021 (WOC'20), S. 37–42. – URL https://doi.org/10.1145/3429885.3429967. – ISBN 9781450382090

[33] VMware Tanzu, Spring by: *GitHub - spring-projects/spring-petclinic: A sample Spring-based application*. – URL https://github.com/spring-projects/spring-petclinic. – Accessed: 2024-03-26

[34] Weber, Ingo ; Nepal, Surya ; Zhu, Liming: Developing Dependable and Secure Cloud Applications. In: *IEEE Internet Computing* 20 (2016), Nr. 3, S. 74–79

[35] Zakarya, Muhammad: Energy, performance and cost efficient datacenters: A survey. In: *Renewable and Sustainable Energy Reviews* 94 (2018), 10, S. 363–385. – URL https://doi.org/10.1016/j.rser.2018.06.005

[36] Öggl, Bernd ; Kofler, Michael: *Docker: Das Praxisbuch für Entwickler und DevOps-Teams*. 4. Rheinwerk Verlag, 10 2023. – ISBN 9783836296465

A Anhang

A.1 Use-Cases

A.1.1 UC1 – Anpassbarkeit durch Konfigurationsdateien

Abschnitt	**Inhalt**	
Bezeichner	**UC1**	
Name	Anpassbarkeit durch Konfigurationsdateien	
Priorität	Mittel	
Kurzbeschreibung	Die Ausführung der Pipeline hängt von der dazugehörigen Konfigurationsdatei ab	
Auslösendes Ereignis	Die ausführende Person lädt Konfigurationsdatei in den Verzeichnisserver hoch	
Akteure	Jenkins Server, Ausführende Person	
Vorbedingungen	Es wurde eine Konfigurationsdatei erstellt	
Nachbedingungen	Die korrekte Konfiguration bzw. Pipeline wurde ausgeführt	
Ergebnis	Die korrekte Pipeline ist ausgeführt worden	
Hauptszenario	**Schritt**	**Aktion**
	1	Pipeline mit Konfiguration A wird gestartet
	2	Jenkins liest Konfigurationsdatei
	3	Jenkins führt Pipeline A aus
Alternativszenarien	**Schritt**	**Abzweigende Aktion**
	AC1	Pipeline mit Konfiguration B wird gestartet
	AC3	Jenkins führt Pipeline B aus
Ausnahmeszenarien	**Schritt**	**Abzweigende Aktion**
	2a	Konfiguration ist nicht valide: Der Prozess wird abgebrochen

Tabelle A.2: Use-Case 1

A.1.2 UC2 – Parallelität von Ausführungsabschnitten

Abschnitt	**Inhalt**	
Bezeichner	**UC2**	
Name	Parallele Ausführungsabschnitte	
Priorität	Mittel	
Kurzbeschreibung	Beim Ausführen einer Pipeline sollen mehrere Abschnitte der Pipeline parallel verarbeitet werden	
Auslösendes Ereignis	Pipeline Ausführung wird gestartet	
Akteure	Jenkins Server, Kubernetes Server	
Vorbedingungen	Pipeline wurde konfiguriert und ist ausführbar	
Nachbedingungen	Abschnitte wurden verarbeitet	
Ergebnis	Abschnitte sind verarbeitet	
Hauptszenario	**Schritt**	**Aktion**
	1	Pipeline wird gestartet
	2	Jenkins liest Konfigurationsdateien
	3	Jenkins fragt Kubernetes nach Ressourcen
	4	Kubernetes stellt Ausführungsumgebungen bereit
Alternativszenarien	**Schritt**	**Abzweigende Aktion**
	-	-
Ausnahmeszenarien	**Schritt**	**Abzweigende Aktion**
	2a	Konfiguration ist nicht valide: Der Prozess wird abgebrochen
	3a	Kubernetes ist nicht erreichbar: Der Prozess wird abgebrochen
	4a	Bereitstellung ist fehlerhaft: Der Prozess wird abgebrochen

Tabelle A.4: Use-Case 2

A.1.3 UC3 – Automatisches Ausführen von Prozessschritten

Abschnitt	**Inhalt**	
Bezeichner	**UC3**	
Name	Automatische Ausführung	
Priorität	Mittel	
Kurzbeschreibung	Beim Ausführen einer Pipeline sollen alle Schritte automatisch durchgeführt werden	
Auslösendes Ereignis	Pipeline Ausführung wird gestartet	
Akteure	Jenkins Server, Kubernetes Server	
Vorbedingungen	Pipeline wurde konfiguriert und ist ausführbar	
Nachbedingungen	Gesamte Pipeline wurde automatisch durchgeführt	
Ergebnis	Gesamte Pipeline ist automatisch durchgeführt worden	
Hauptszenario	**Schritt**	**Aktion**
	1	Pipeline wird gestartet
	2	Jenkins liest Konfigurationsdateien
	3	Jenkins fragt Kubernetes nach Ressourcen
	4	Kubernetes stellt Buildumgebungen bereit
	5	Jenkins führt Pipeline schrittweise aus
	6	Pipeline wird beendet
	7	Ressourcen werden wieder freigegeben
Alternativszenarien	**Schritt**	**Abzweigende Aktion**
	-	-
Ausnahmeszenarien	**Schritt**	**Abzweigende Aktion**
	2a	Konfiguration ist nicht valide: Der Prozess wird abgebrochen
	3a	Kubernetes ist nicht erreichbar: Der Prozess wird abgebrochen
	4a	Bereitstellung ist fehlerhaft: Der Prozess wird abgebrochen
	5a	Schritt ist fehlerhaft: Der Prozess wird abgebrochen

Tabelle A.6: Use-Case 3

A.1.4 UC4 – Erkennen von fehlerhaften Zuständen

Abschnitt	**Inhalt**	
Bezeichner	**UC4**	
Name	Erkennen fehlerhafter Zustände	
Priorität	Mittel	
Kurzbeschreibung	Beim Ausführen einer Pipeline sollen fehlerhafte Zustände erkannt werden	
Auslösendes Ereignis	Pipeline Ausführung wird gestartet	
Akteure	Jenkins Server	
Vorbedingungen	Pipeline wurde konfiguriert und ist ausführbar	
Nachbedingungen	Fehlerhafte Zustände wurden geprüft	
Ergebnis	Fehlerhafte Zustände sind geprüft worden	
Hauptszenario	**Schritt**	**Aktion**
	1	Pipeline wird gestartet
	2	Pipeline führt Schritt aus
	3	Jenkins prüft auf fehlerhaften Zustand
	4	Jenkins erkennt einen fehlerhaften Zustand
	5	Pipeline wird beendet
Alternativszenarien	**Schritt**	**Abzweigende Aktion**
	AC4	Jenkins erkennt keinen fehlerhaften Zustand
Ausnahmeszenarien	**Schritt**	**Abzweigende Aktion**
	4a	Fehlerhafter Zustand erforder Abbruch: Der Prozess wird abgebrochen
	4b	Fehlerhafter Zustand kann ignoriert werden: Der Prozess läuft weiter

Tabelle A.8: Use-Case 4

A.1.5 UC5 – Automatische Tests und Sicherheitsscans

Abschnitt	**Inhalt**	
Bezeichner	**UC5**	
Name	Automatische Tests & Sicherheitsscans	
Priorität	Mittel	
Kurzbeschreibung	Beim Ausführen einer Pipeline sollen Tests und Scans zur Sicherheitsprüfung durchgeführt werden	
Auslösendes Ereignis	Pipeline Ausführung wird gestartet	
Akteure	Jenkins Server	
Vorbedingungen	Pipeline wurde konfiguriert und ist ausführbar	
Nachbedingungen	Software wurde mit Tests und Scans geprüft	
Ergebnis	Software ist mit Tests und Scans geprüft worden	
Hauptszenario	**Schritt**	**Aktion**
	1	Pipeline wird gestartet
	2	Pipeline führt Tests aus
	3	Pipeline führt Scans aus
	4	Pipeline wird beendet
Alternativszenarien	**Schritt**	**Abzweigende Aktion**
	-	-
Ausnahmeszenarien	**Schritt**	**Abzweigende Aktion**
	2a	Test schlägt fehl: Der Prozess wird abgebrochen
	3a	Scan schlägt fehl: Der Prozess wird abgebrochen

Tabelle A.10: Use-Case 5

A.2 Softwareübersicht

Software	Einsatzbereich	Abschnitt nach 4.6
Unit-Tests	COMMON TEST	CI
Integrationstests	COMMON TEST	CI
SonarQube	SAST SCAN	CI
Snyk	SAST SCAN	CI
OWASP ZAP	DAST SCAN	CD
Nikito	DAST SCAN	CD
Docker Scout	SCA SCAN	CD

Tabelle A.12: Verwendete Sicherheitslösungen

A.3 Versuch – Systemkonfigurationen

A.3.1 Virtuelle Maschinen

Virtuelle Maschine	CPU-Kerne	Arbeitsspeicher (GB)	Festplatte (GB)
k8s-master-1	2	4	20
k8s-worker-1	4	4	40
k8s-worker-2	16	12	80
docker-host-1	4	4	25

Tabelle A.14: Konfigurationen der virtuellen Maschinen

A.3.2 Ansible Playbooks

Listing A.1: Ansible Skript - k8s-master.yml

```
---
- name: Wrapper for Ansible Deployment.
    hosts: k8s_masters
    become: true
    remote_user: ubuntu

    tasks:
    - name: Import vars
        ansible.builtin.include_vars: vars.yml

    - name: Install dependecies
        ansible.builtin.include_tasks: pre-install-
          k8s_dependencies.yml

    - name: Install k8s-master
        ansible.builtin.include_tasks: 04-install-k8s_master.
          yml
```

Listing A.2: Ansible Skript - vars.yml

```
---
docker_plugin_dir: /home/ubuntu/.docker/cli-plugins
local_working_dir: /mnt/c/Users/rgyetvai/Coding/\
                   GIT/Personal/terraform_playground/ansible
supabase_dir: /home/ubuntu/supabase/docker
supabase_root_dir: /home/ubuntu/supabase
app_directory: /home/ubuntu/app
```

Listing A.3: Ansible Skript - pre-install-k8s_dependencies.yml

```
---
- name: Check if swap is on
  ansible.builtin.command: swapon --show
  register: swap_check
  changed_when: false
  failed_when: swap_check.rc not in [0, 1]

- name: Turn off swap.
  ansible.builtin.command: swapoff -a
  when: swap_check.stdout != ""

- name: Disable swap permanently.
  ansible.builtin.replace:
    path: /etc/fstab
    regexp: '^([^#].*?\sswap\s+sw\s+.*)$'
    replace: '# \1'

- name: Update apt cache.
  ansible.builtin.apt:
    update_cache: true

- name: Install required packages.
  ansible.builtin.apt:
    name:
      [
        "apt-transport-https",
        "ca-certificates",
        "curl",
        "gnupg",
        "lsb-release",
      ]
    state: present
```

```
- name: Download the GPG key for the official Docker repository
    .
    ansible.builtin.get_url:
    url: https://download.docker.com/linux/ubuntu/gpg
    dest: /tmp/docker.gpg
    mode: "0440"

- name: Add the GPG key to the keyring.
  ansible.builtin.command:
    cmd: |
      sudo gpg --dearmor \
      -o /usr/share/keyrings/docker-archive-keyring.gpg \
      /tmp/docker.gpg
    creates: /usr/share/keyrings/docker-archive-keyring.gpg

- name: Add the Docker repository to APT sources.
  ansible.builtin.apt_repository:
    repo: "deb [arch=amd64 signed-by=/usr/share/keyrings/\
      docker-archive-keyring.gpg] \
      https://download.docker.com/linux/ubuntu \
      {{ ansible_distribution_release }} stable"
    state: present
    update_cache: true

- name: Install Docker.
  ansible.builtin.apt:
    name: ["docker-ce", "docker-ce-cli", "containerd.io"]
    state: present
    update_cache: true

- name: Update Docker daemon configuration
  ansible.builtin.copy:
    dest: /etc/docker/daemon.json
    content: |
      {
        "exec-opts": ["native.cgroupdriver=systemd"],
```

```
        "log-driver": "json-file",
        "log-opts": {
            "max-size": "100m"
        },
        "storage-driver": "overlay2"
      }
    owner: root
    group: root
    mode: '0644'

- name: Download cri-dockerd .deb package
  ansible.builtin.get_url:
    url: "https://github.com/Mirantis/cri-dockerd/\
        releases/download/v0.3.4/\
        cri-dockerd_0.3.4.3-0.ubuntu-jammy_amd64.deb"
    dest: "/tmp/cri-dockerd_0.3.4.3-0.ubuntu-jammy_amd64.deb"
    mode: "0644"

- name: Install cri-dockerd package
  ansible.builtin.apt:
    deb: "/tmp/cri-dockerd_0.3.4.3-0.ubuntu-jammy_amd64.deb"

- name: Remove tmp file
  ansible.builtin.file:
    path: "/tmp/cri-dockerd_0.3.4.3-0.ubuntu-jammy_amd64.deb"
    state: absent

- name: Enable Docker service and reload daemon.
  ansible.builtin.systemd:
    name: docker
    enabled: true
    daemon_reload: true
    state: restarted

# Install shared Kubernetes components #
```

```
- name: Update apt cache.
  ansible.builtin.apt:
    update_cache: true

- name: Install required packages.
  ansible.builtin.apt:
    name: ["apt-transport-https", "ca-certificates", "curl"]
    state: present

- name: Add Kubernetes repo GPG key
  ansible.builtin.get_url:
    url: https://packages.cloud.google.com/apt/doc/apt-key.gpg
    dest: /tmp/kubernetes-archive-keyring.gpg
    mode: "0440"

- name: Add the GPG key to the keyring.
  ansible.builtin.command:
    cmd: sudo gpg --dearmor \
       -o /usr/share/keyrings/kubernetes-archive-keyring.gpg \
       /tmp/kubernetes-archive-keyring.gpg
    creates: /usr/share/keyrings/kubernetes-archive-keyring.gpg

- name: Add Kubernetes repository to APT sources.
  ansible.builtin.apt_repository:
    repo: "deb [signed-by=/usr/share/keyrings/\
        kubernetes-archive-keyring.gpg] \
        https://apt.kubernetes.io/ \
        kubernetes-xenial main"
    state: present
    update_cache: true

- name: Update apt cache.
  ansible.builtin.apt:
    update_cache: true

- name: Install required packages.
```

```
    ansible.builtin.apt:
      name: ["kubelet", "kubeadm", "kubectl"]
      state: present

  - name: Wait for dpkg lock to be released.
    ansible.builtin.shell: |
      while sudo fuser /var/lib/dpkg/lock >/dev/null 2>&1; do
         sleep 5; done;

  - name: Halten Sie die Version der Pakete.
    ansible.builtin.dpkg_selections:
      name: "{{ item }}"
      selection: hold
    loop: ["kubelet", "kubeadm", "kubectl"]

  - name: Enable to start on boot and reload the kubelet daemon.
    ansible.builtin.systemd:
      name: kubelet
      enabled: true
      state: restarted
      daemon_reload: true
```

Listing A.4: Ansible Skript - 04-install-k8s_master.yml

```
---
- name: Check if Kubernetes cluster is initialized
  ansible.builtin.shell: |
    kubectl cluster-info
  register: k8s_check
  changed_when: false
  failed_when: k8s_check.rc not in [0, 1]

- name: Initialize Kubernetes cluster with kubeadm
  ansible.builtin.shell: |
    sudo kubeadm init \
    --pod-network-cidr=10.244.0.0/16 \
    --cri-socket /var/run/cri-dockerd.sock
  when: k8s_check.rc == 1
  register: k8s_init_output

- name: Output kubeadm init output
  ansible.builtin.debug:
    msg: "{{ k8s_init_output.stdout_lines }}"

- name: Create .kube directory
  ansible.builtin.file:
    path: "/home/{{ ansible_user }}/.kube"
    state: "directory"
    owner: "{{ ansible_user }}"
    group: "{{ ansible_user }}"
    mode: "0755"

- name: Copy admin.conf to user's kube config
  ansible.builtin.copy:
    src: /etc/kubernetes/admin.conf
    dest: /home/{{ ansible_user }}/.kube/config
    remote_src: true
    owner: "{{ ansible_user }}"
```

```
    group: "{{ ansible_user }}"
    mode: "0644"

- name: Get user id
  ansible.builtin.command: id -u
  register: uid
  changed_when: false

- name: Get group id
  ansible.builtin.command: id -g
  register: gid
  changed_when: false

- name: Change owner of .kube/config
  ansible.builtin.file:
    path: "/home/{{ ansible_user }}/.kube/config"
    owner: "{{ uid.stdout }}"
    group: "{{ gid.stdout }}"

- name: Wait for master node to be ready
  ansible.builtin.wait_for:
    host: localhost
    port: 6443
    delay: 5
    timeout: 60
    state: started

- name: Check if Weave Net is installed
  ansible.builtin.command:
    kubectl get daemonset weave-net -n kube-system
  register: weave_check
  changed_when: false
  failed_when: weave_check.rc not in [0, 1]

- name: Install Weave Net
  ansible.builtin.shell: |
```

```
    kubectl apply -f https://github.com/weaveworks/\
    weave/releases/download/v2.8.1/weave-daemonset-k8s.yaml
  when: weave_check.rc == 1
```

Listing A.5: Ansible Skript - k8s-worker.yml

```
---
- name: Wrapper for Ansible Deployment.
  hosts: k8s-worker-04
  become: true
  remote_user: ubuntu

  tasks:
    - name: Import vars
      ansible.builtin.include_vars: vars.yml

    - name: Install dependecies
      ansible.builtin.include_tasks: pre-install-
          k8s_dependencies.yml

    - name: Install k8s-worker
      ansible.builtin.include_tasks: 05-install-k8s_worker.yml
```

Listing A.6: Ansible Skript - 05-install-k8s_worker.yml

```
---
- name: Join worker nodes to the Kubernetes cluster
  ansible.builtin.shell: >
    kubeadm join 10.0.10.50:6443 --token <TOKEN>
    --discovery-token-ca-cert-hash <HASH_VALUE>
    --cri-socket /var/run/cri-dockerd.sock
```

A.4 Jenkins- und Dockerfiles

A.4.1 Dockerfile – Docker in Docker Container

```
# Use Docker-in-Docker image
FROM docker:25.0.0-rc.3-dind

# Metadata
LABEL de.rgyetvai.version="1.0"
LABEL de.rgyetvai.release-date="2021-12-20"
LABEL type="Jenkins Agent"

# Update & install dependencies and packages
RUN apk update
RUN apk add --no-cache curl
RUN apk add --no-cache --repository=https://dl-cdn.alpinelinux.
    org/alpine/edge/community/x86_64/ openjdk21-jdk

# Upgrade packages and repositories
RUN apk upgrade --no-cache

# Download and install Maven
ARG MAVEN_VERSION=3.9.6
ARG USER_HOME_DIR="/root"
ARG SHA=706
    f01b20dec0305a822ab614d51f32b07ee11d0218175e55450242e49d2156386483b506b3a4

ARG BASE_URL=https://apache.osuosl.org/maven/maven-3/${
   MAVEN_VERSION}/binaries

# Create the directories, download maven, validate the download
    , install it, remove downloaded file and set links
RUN mkdir -p /usr/share/maven /usr/share/maven/ref \
    && echo "Downloading maven" \
    && curl -fsSL -o /tmp/apache-maven.tar.gz ${BASE_URL}/
        apache-maven-${MAVEN_VERSION}-bin.tar.gz \
```

```
    \
    && echo "Checking download hash" \
    && echo "${SHA} /tmp/apache-maven.tar.gz" | sha512sum -c -
       \
    \
    && echo "Unzipping maven" \
    && tar -xzf /tmp/apache-maven.tar.gz -C /usr/share/maven --
       strip-components=1 \
    \
    && echo "Cleaning and setting links" \
    && rm -f /tmp/apache-maven.tar.gz \
    && ln -s /usr/share/maven/bin/mvn /usr/bin/mvn

# Define environmental variables required by Maven, like
   Maven_Home directory and where the maven repo is located
ENV MAVEN_HOME /usr/share/maven
ENV MAVEN_CONFIG "$USER_HOME_DIR/.m2"

# Install trivy through script
RUN curl -sfL https://raw.githubusercontent.com/aquasecurity/
   trivy/main/contrib/install.sh | sh -s -- -b /usr/local/bin

# Create the temporary working directory
WORKDIR /tmp

# Define the entrypoint
CMD [""]
```

A.4.2 Dockerfile – Stresstest-Container

```
# Use of the official Alpine Linux image
FROM alpine:edge

# Update & install dependencies and packages
RUN apk update
RUN apk add --no-cache curl
RUN apk add --no-cache --repository=https://dl-cdn.alpinelinux.
    org/alpine/edge/community stress-ng

# Upgrade packages and repositories
RUN apk upgrade --no-cache

# Copy the script to the container
COPY simulate_load.sh /usr/local/bin/simulate_load.sh

# Execute the script
RUN chmod +x /usr/local/bin/simulate_load.sh

CMD ["/bin/sh"]
```

A.4.3 Bash Skript – Stresstest

```
#!/bin/sh

# Function to display help
show_help() {
    echo "Usage: $0 DURATION CPU_CORES CPU_OPS [-h | --help]"
    echo ""
    echo "Simulates CPU load based on the specified parameters."
    echo "Evaluates the execution time for a target number of CPU stress test cycles."
    echo ""
    echo "Arguments:"
    echo "  DURATION        Duration of the simulation in seconds"
    echo "  CPU_CORES       Number of CPU cores to be used"
    echo "  CPU_OPS         Target number of CPU stress test operations (bogo-ops)"
    echo ""
    echo "Options:"
    echo "  -h, --help      Shows this help message and exits."
}

# Check for help flag
if [ "$1" == "-h" ] || [ "$1" == "--help" ]; then
    show_help
    exit 0
fi

# Check if the correct number of arguments were passed
if [ "$#" -ne 3 ]; then
    show_help
    exit 1
fi
```

```
DURATION=$1
CPU_CORES=$2
CPU_OPS=$3

echo "Simulating CPU load for $DURATION seconds with $CPU_CORES
    cores targeting $CPU_OPS CPU stress test operations."

# Use stress-ng, if available
if
    command -v stress-ng &
    >/dev/null
then
    stress-ng --cpu $CPU_CORES --cpu-ops $CPU_OPS --timeout ${
       DURATION}s --metrics-brief
else
    echo "Error: stress-ng is not installed. This simulation
       requires stress-ng."
    exit 3
fi
```

A.4.4 Jenkinsfiles – Experiment 1

Serial – Jenkinsfile

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: jenkins-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
                      requiredDuringSchedulingIgnoredDuringExecution
                          :
                        nodeSelectorTerms:
                        - matchExpressions:
                          - key: kubernetes.io/hostname
                            operator: In
                            values:
                            - k8s-worker-4
                  containers:
                  - command:
                    - cat
                    name: custom-stress-testing-00
                    image: rgyetvai/custom-stress-testing:
                        latest
                    resources:
                      limits:
                        cpu: "2"
```

```
                    memory: 2Gi
                  requests:
                    cpu: 500m
                    memory: 500Mi
                tty: true
            '''
        }
    }
    options {
        buildDiscarder(logRotator(numToKeepStr: '5'))
        parallelsAlwaysFailFast()
    }
    stages {
        stage('Execute Load Simulation 1') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 2') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 3') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
```

```
                }
            }
        }
    }
    stage('Execute Load Simulation 4') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 5') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 6') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 7') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
```

```
                }
            }
        }
    }
    stage('Execute Load Simulation 8') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 9') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 10') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
}
post {
    success {
        echo 'Build Success'
    }
```

```
        failure {
            echo 'Build Failed'
        }
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

Konfiguration A1 - Jenkinsfile

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: jenkins-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
                      requiredDuringSchedulingIgnoredDuringExecution
                          :
                        nodeSelectorTerms:
                        - matchExpressions:
                          - key: kubernetes.io/hostname
                            operator: In
                            values:
```

```
                - k8s-worker-4
    containers:
    - command:
      - cat
      name: custom-stress-testing-00
      image: rgyetvai/custom-stress-testing:
          latest
      resources:
        limits:
          cpu: "2"
          memory: 2Gi
        requests:
          cpu: 500m
          memory: 500Mi
      tty: true
    - command:
      - cat
      name: custom-stress-testing-01
      image: rgyetvai/custom-stress-testing:
          latest
      resources:
        limits:
          cpu: "2"
          memory: 2Gi
        requests:
          cpu: 500m
          memory: 500Mi
      tty: true
    - command:
      - cat
      name: custom-stress-testing-02
      image: rgyetvai/custom-stress-testing:
          latest
      resources:
        limits:
          cpu: "2"
```

```
                    memory: 2Gi
                  requests:
                    cpu: 500m
                    memory: 500Mi
                tty: true
            '''
        }
    }
    options {
        buildDiscarder(logRotator(numToKeepStr: '5'))
        parallelsAlwaysFailFast()
    }
    stages {
        stage('Execute Load Simulation 1') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 2') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 3') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
```

```
                }
            }
        }
    }
    stage('Execute Load Simulation 4') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 5') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 6') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 7') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
```

```
                }
            }
        }
    }
    stage('Parallel Section') {
        parallel {
            stage('Execute Load Simulation 8') {
                steps {
                    container('custom-stress-testing-01') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
            stage('Execute Load Simulation 9') {
                steps {
                    container('custom-stress-testing-02') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
        }
    }
    stage('Execute Load Simulation 10') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
```

```
        }
    }
    post {
        success {
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
        }
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

Konfiguration A2 - Jenkinsfile

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: jenkins-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
```

```
        requiredDuringSchedulingIgnoredDuringExecution
            :
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
              - k8s-worker-4
  containers:
  - command:
    - cat
    name: custom-stress-testing-00
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-01
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
```

```
    - cat
    name: custom-stress-testing-02
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-03
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-04
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
```

```
                    memory: 500Mi
                tty: true
            '''
        }
    }
    options {
        buildDiscarder(logRotator(numToKeepStr: '5'))
        parallelsAlwaysFailFast()
    }
    stages {
        stage('Execute Load Simulation 1') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 2') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 3') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
```

```
        }
        stage('Execute Load Simulation 4') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 5') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Parallel Section') {
            parallel {
                stage('Execute Load Simulation 6') {
                    steps {
                        container('custom-stress-testing-01') {
                            script {
                                sh 'simulate_load.sh 600 2
                                    250000'
                            }
                        }
                    }
                }
                stage('Execute Load Simulation 7') {
                    steps {
                        container('custom-stress-testing-02') {
                            script {
```

```
                        sh 'simulate_load.sh 600 2
                            250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 8') {
            steps {
                container('custom-stress-testing-03') {
                    script {
                        sh 'simulate_load.sh 600 2
                            250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 9') {
            steps {
                container('custom-stress-testing-04') {
                    script {
                        sh 'simulate_load.sh 600 2
                            250000'
                    }
                }
            }
        }
    }
}
stage('Execute Load Simulation 10') {
    steps {
        container('custom-stress-testing-00') {
            script {
                sh 'simulate_load.sh 600 2 250000'
            }
        }
    }
```

```
        }
    }
    post {
        success {
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
        }
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

Konfiguration A3 - Jenkinsfile

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: jenkins-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
```

```
        requiredDuringSchedulingIgnoredDuringExecution
            :
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
              - k8s-worker-4
  containers:
  - command:
    - cat
    name: custom-stress-testing-00
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-01
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
```

```
    - cat
    name: custom-stress-testing-02
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-03
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-04
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
```

```
                    memory: 500Mi
                tty: true
              - command:
                - cat
                name: custom-stress-testing-05
                image: rgyetvai/custom-stress-testing:
                    latest
                resources:
                  limits:
                    cpu: "2"
                    memory: 2Gi
                  requests:
                    cpu: 500m
                    memory: 500Mi
                tty: true
              - command:
                - cat
                name: custom-stress-testing-06
                image: rgyetvai/custom-stress-testing:
                    latest
                resources:
                  limits:
                    cpu: "2"
                    memory: 2Gi
                  requests:
                    cpu: 500m
                    memory: 500Mi
                tty: true
            '''
    }
}
options {
    buildDiscarder(logRotator(numToKeepStr: '5'))
    parallelsAlwaysFailFast()
}
stages {
```

```
stage('Execute Load Simulation 1') {
    steps {
        container('custom-stress-testing-00') {
            script {
                sh 'simulate_load.sh 600 2 250000'
            }
        }
    }
}
stage('Execute Load Simulation 2') {
    steps {
        container('custom-stress-testing-00') {
            script {
                sh 'simulate_load.sh 600 2 250000'
            }
        }
    }
}
stage('Execute Load Simulation 3') {
    steps {
        container('custom-stress-testing-00') {
            script {
                sh 'simulate_load.sh 600 2 250000'
            }
        }
    }
}
stage('Parallel Section') {
    parallel {
        stage('Execute Load Simulation 4') {
            steps {
                container('custom-stress-testing-01') {
                    script {
                        sh 'simulate_load.sh 600 2
                            250000'
                    }
```

```
            }
        }
    }
    stage('Execute Load Simulation 5') {
        steps {
            container('custom-stress-testing-02') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 6') {
        steps {
            container('custom-stress-testing-03') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 7') {
        steps {
            container('custom-stress-testing-04') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 8') {
        steps {
            container('custom-stress-testing-05') {
```

```
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
                stage('Execute Load Simulation 9') {
                    steps {
                        container('custom-stress-testing-06') {
                            script {
                                sh 'simulate_load.sh 600 2
                                    250000'
                            }
                        }
                    }
                }
            }
        }
        stage('Execute Load Simulation 10') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
    }
    post {
        success {
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
        }
```

```
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

Konfiguration A4 - Jenkinsfile

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: jenkins-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
                      requiredDuringSchedulingIgnoredDuringExecution
                          :
                        nodeSelectorTerms:
                        - matchExpressions:
                          - key: kubernetes.io/hostname
                            operator: In
                            values:
                            - k8s-worker-4
                  containers:
                  - command:
```

```
    - cat
    name: custom-stress-testing-00
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-01
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-02
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
```

```
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-03
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-04
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-05
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
```

```
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-06
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-07
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-08
    image: rgyetvai/custom-stress-testing:
       latest
```

```
                    resources:
                      limits:
                        cpu: "2"
                        memory: 2Gi
                      requests:
                        cpu: 500m
                        memory: 500Mi
                    tty: true
                '''
        }
    }
    options {
        buildDiscarder(logRotator(numToKeepStr: '5'))
        parallelsAlwaysFailFast()
    }
    stages {
        stage('Execute Load Simulation 1') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Parallel Section') {
            parallel {
                stage('Execute Load Simulation 2') {
                    steps {
                        container('custom-stress-testing-01') {
                            script {
                                sh 'simulate_load.sh 600 2
                                    250000'
                            }
                        }
                    }
```

```
}
stage('Execute Load Simulation 3') {
    steps {
        container('custom-stress-testing-02') {
            script {
                sh 'simulate_load.sh 600 2
                    250000'
            }
        }
    }
}
stage('Execute Load Simulation 4') {
    steps {
        container('custom-stress-testing-03') {
            script {
                sh 'simulate_load.sh 600 2
                    250000'
            }
        }
    }
}
stage('Execute Load Simulation 5') {
    steps {
        container('custom-stress-testing-04') {
            script {
                sh 'simulate_load.sh 600 2
                    250000'
            }
        }
    }
}
stage('Execute Load Simulation 6') {
    steps {
        container('custom-stress-testing-05') {
            script {
```

```
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 7') {
        steps {
            container('custom-stress-testing-06') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 8') {
        steps {
            container('custom-stress-testing-07') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 9') {
        steps {
            container('custom-stress-testing-08') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
```

```
            }
        }
        stage('Execute Load Simulation 10') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
    }
    post {
        success {
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
        }
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

A.4.5 Jenkinsfiles - Experiment 2

Konfiguration B1 - Jenkinsfile

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: jenkins-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
                      requiredDuringSchedulingIgnoredDuringExecution
                          :
                        nodeSelectorTerms:
                        - matchExpressions:
                          - key: kubernetes.io/hostname
                            operator: In
                            values:
                            - k8s-worker-4
                  containers:
                  - command:
                    - cat
                    name: custom-stress-testing-00
                    image: rgyetvai/custom-stress-testing:
                        latest
                    resources:
                      limits:
                        cpu: "2"
```

```
                        memory: 2Gi
                    requests:
                        cpu: 500m
                        memory: 500Mi
                    tty: true
                - command:
                    - cat
                    name: custom-stress-testing-01
                    image: rgyetvai/custom-stress-testing:
                        latest
                    resources:
                        limits:
                            cpu: "2"
                            memory: 2Gi
                        requests:
                            cpu: 500m
                            memory: 500Mi
                    tty: true
                - command:
                    - cat
                    name: custom-stress-testing-02
                    image: rgyetvai/custom-stress-testing:
                        latest
                    resources:
                        limits:
                            cpu: "2"
                            memory: 2Gi
                        requests:
                            cpu: 500m
                            memory: 500Mi
                    tty: true
            '''
    }
}
options {
    buildDiscarder(logRotator(numToKeepStr: '5'))
```

```
        parallelsAlwaysFailFast()
    }
    stages {
        stage('Execute Load Simulation 1') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 2') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Parallel Section') {
            parallel {
                stage('Execute Load Simulation 3') {
                    steps {
                        container('custom-stress-testing-01') {
                            script {
                                sh 'simulate_load.sh 600 2
                                    250000'
                            }
                        }
                    }
                }
                stage('Execute Load Simulation 4') {
                    steps {
                        container('custom-stress-testing-02') {
```

```
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
        }
        stage('Execute Load Simulation 5') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 6') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
    }
    post {
        success {
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
        }
        unstable {
```

```
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

Konfiguration B2 - Jenkinsfile

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: jenkins-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
                      requiredDuringSchedulingIgnoredDuringExecution
                          :
                        nodeSelectorTerms:
                        - matchExpressions:
                          - key: kubernetes.io/hostname
                            operator: In
                            values:
                            - k8s-worker-4
                  containers:
                  - command:
                    - cat
```

```
    name: custom-stress-testing-00
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-01
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-02
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
```

```
                    tty: true
                  - command:
                    - cat
                    name: custom-stress-testing-03
                    image: rgyetvai/custom-stress-testing:
                        latest
                    resources:
                      limits:
                        cpu: "2"
                        memory: 2Gi
                      requests:
                        cpu: 500m
                        memory: 500Mi
                    tty: true
                  - command:
                    - cat
                    name: custom-stress-testing-04
                    image: rgyetvai/custom-stress-testing:
                        latest
                    resources:
                      limits:
                        cpu: "2"
                        memory: 2Gi
                      requests:
                        cpu: 500m
                        memory: 500Mi
                    tty: true
                '''
    }
}
options {
    buildDiscarder(logRotator(numToKeepStr: '5'))
    parallelsAlwaysFailFast()
}
stages {
    stage('Execute Load Simulation 1') {
```

```
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 2') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Parallel Section') {
        parallel {
            stage('Execute Load Simulation 3') {
                steps {
                    container('custom-stress-testing-01') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
            stage('Execute Load Simulation 4') {
                steps {
                    container('custom-stress-testing-02') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
```

```
                    }
                }
            }
            stage('Execute Load Simulation 5') {
                steps {
                    container('custom-stress-testing-03') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
            stage('Execute Load Simulation 6') {
                steps {
                    container('custom-stress-testing-04') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
        }
    }
    stage('Execute Load Simulation 7') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 8') {
        steps {
```

```
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
    }
    post {
        success {
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
        }
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

Konfiguration B3 - Jenkinsfile

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
```

```
    name: jenkins-build-pod
    namespace: devops-tools
  spec:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution
            :
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
              - k8s-worker-4
    containers:
    - command:
      - cat
      name: custom-stress-testing-00
      image: rgyetvai/custom-stress-testing:
          latest
      resources:
        limits:
          cpu: "2"
          memory: 2Gi
        requests:
          cpu: 500m
          memory: 500Mi
      tty: true
    - command:
      - cat
      name: custom-stress-testing-01
      image: rgyetvai/custom-stress-testing:
          latest
      resources:
        limits:
          cpu: "2"
          memory: 2Gi
```

```
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-02
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-03
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-04
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
```

```
                    limits:
                      cpu: "2"
                      memory: 2Gi
                    requests:
                      cpu: 500m
                      memory: 500Mi
                  tty: true
                - command:
                  - cat
                  name: custom-stress-testing-05
                  image: rgyetvai/custom-stress-testing:
                      latest
                  resources:
                    limits:
                      cpu: "2"
                      memory: 2Gi
                    requests:
                      cpu: 500m
                      memory: 500Mi
                  tty: true
                - command:
                  - cat
                  name: custom-stress-testing-06
                  image: rgyetvai/custom-stress-testing:
                      latest
                  resources:
                    limits:
                      cpu: "2"
                      memory: 2Gi
                    requests:
                      cpu: 500m
                      memory: 500Mi
                  tty: true
              '''
        }
    }
```

```
    options {
        buildDiscarder(logRotator(numToKeepStr: '5'))
        parallelsAlwaysFailFast()
    }
    stages {
        stage('Execute Load Simulation 1') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 2') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Parallel Section') {
            parallel {
                stage('Execute Load Simulation 3') {
                    steps {
                        container('custom-stress-testing-01') {
                            script {
                                sh 'simulate_load.sh 600 2
                                    250000'
                            }
                        }
                    }
                }
                stage('Execute Load Simulation 4') {
```

```
        steps {
            container('custom-stress-testing-02') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 5') {
        steps {
            container('custom-stress-testing-03') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 6') {
        steps {
            container('custom-stress-testing-04') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 7') {
        steps {
            container('custom-stress-testing-05') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
```

```
                    }
                }
            }
            stage('Execute Load Simulation 8') {
                steps {
                    container('custom-stress-testing-06') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
        }
        stage('Execute Load Simulation 9') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 10') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
    }
    post {
        success {
```

```
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
        }
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

Konfiguration B4 - Jenkinsfile

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: jenkins-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
                      requiredDuringSchedulingIgnoredDuringExecution
                          :
                        nodeSelectorTerms:
                        - matchExpressions:
                          - key: kubernetes.io/hostname
```

```
                operator: In
                values:
                - k8s-worker-4
    containers:
    - command:
      - cat
      name: custom-stress-testing-00
      image: rgyetvai/custom-stress-testing:
          latest
      resources:
        limits:
          cpu: "2"
          memory: 2Gi
        requests:
          cpu: 500m
          memory: 500Mi
      tty: true
    - command:
      - cat
      name: custom-stress-testing-01
      image: rgyetvai/custom-stress-testing:
          latest
      resources:
        limits:
          cpu: "2"
          memory: 2Gi
        requests:
          cpu: 500m
          memory: 500Mi
      tty: true
    - command:
      - cat
      name: custom-stress-testing-02
      image: rgyetvai/custom-stress-testing:
          latest
      resources:
```

```
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-03
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-04
    image: rgyetvai/custom-stress-testing:
       latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-05
```

```
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-06
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
  - command:
    - cat
    name: custom-stress-testing-07
    image: rgyetvai/custom-stress-testing:
        latest
    resources:
      limits:
        cpu: "2"
        memory: 2Gi
      requests:
        cpu: 500m
        memory: 500Mi
    tty: true
```

```
                - command:
                  - cat
                  name: custom-stress-testing-08
                  image: rgyetvai/custom-stress-testing:
                      latest
                  resources:
                    limits:
                      cpu: "2"
                      memory: 2Gi
                    requests:
                      cpu: 500m
                      memory: 500Mi
                  tty: true
              '''
    }
}
options {
    buildDiscarder(logRotator(numToKeepStr: '5'))
    parallelsAlwaysFailFast()
}
stages {
    stage('Execute Load Simulation 1') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 2') {
        steps {
            container('custom-stress-testing-00') {
                script {
                    sh 'simulate_load.sh 600 2 250000'
                }
```

```
            }
        }
    }
    stage('Parallel Section') {
        parallel {
            stage('Execute Load Simulation 3') {
                steps {
                    container('custom-stress-testing-01') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
            stage('Execute Load Simulation 4') {
                steps {
                    container('custom-stress-testing-02') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
            stage('Execute Load Simulation 5') {
                steps {
                    container('custom-stress-testing-03') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
            stage('Execute Load Simulation 6') {
```

```
        steps {
            container('custom-stress-testing-04') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 7') {
        steps {
            container('custom-stress-testing-05') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 8') {
        steps {
            container('custom-stress-testing-06') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
            }
        }
    }
    stage('Execute Load Simulation 9') {
        steps {
            container('custom-stress-testing-07') {
                script {
                    sh 'simulate_load.sh 600 2
                        250000'
                }
```

```
                    }
                }
            }
            stage('Execute Load Simulation 10') {
                steps {
                    container('custom-stress-testing-08') {
                        script {
                            sh 'simulate_load.sh 600 2
                                250000'
                        }
                    }
                }
            }
        }
        stage('Execute Load Simulation 11') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
        stage('Execute Load Simulation 12') {
            steps {
                container('custom-stress-testing-00') {
                    script {
                        sh 'simulate_load.sh 600 2 250000'
                    }
                }
            }
        }
    }
    post {
        success {
```

```
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
        }
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

A.4.6 Jenkinsfiles - Experiment 3

Realer Anwendungsfall - Jenkinsfile seriell

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: jenkins-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
                      requiredDuringSchedulingIgnoredDuringExecution
                          :
                        nodeSelectorTerms:
                        - matchExpressions:
                          - key: kubernetes.io/hostname
                            operator: In
                            values:
                            - k8s-worker-4
                  containers:
                  - command:
                    - cat
                    name: custom-dind
                    image: rgyetvai/custom-dind:latest
                    resources:
                      limits:
                        cpu: "3"
                        memory: 3Gi
```

```
                    requests:
                      cpu: 500m
                      memory: 500Mi
                  tty: true
                  volumeMounts:
                  - mountPath: /var/run/docker.sock
                    name: docker-sock-volume
                volumes:
                - hostPath:
                    path: /var/run/docker.sock
                    type: Socket
                  name: docker-sock-volume
              '''
    }
}
options {
    buildDiscarder(logRotator(numToKeepStr: '5'))
    parallelsAlwaysFailFast()
}
environment {
    SCANNER_HOME = tool 'sonar-scanner'
    DOCKERHUB_CREDENTIALS = credentials('rgyetvai-dockerhub
        ')
    IMAGE_TAG_TEST = 'rgyetvai/petclinic:testing'
    IMAGE_TAG_LATEST = 'rgyetvai/petclinic:latest'
}
stages {
    stage('Git Checkout') {
        steps {
            container('custom-dind') {
                script {
                    git branch: 'parallelized-jenkinsfile',
                        credentialsId: 'git_jenkins_ba_01',
                        url: 'git@github.com:renegyetvai/
                            spring-petclinic.git'
                }
```

```
            }
        }
    }
    stage('Docker Login') {
        steps {
            container('custom-dind') {
                script {
                    sh 'echo $DOCKERHUB_CREDENTIALS_PSW |
                        docker login -u
                        $DOCKERHUB_CREDENTIALS_USR --
                        password-stdin'
                }
            }
        }
    }
    stage('Update Dependencies + Clean, Validate & Compile
       Sources') {
        steps {
            container('custom-dind') {
                sh 'mvn --version'
                // Prepare the environment
                sh 'mvn dependency:purge-local-repository'
                sh 'mvn versions:use-latest-versions'
                sh 'mvn -U clean validate compile -
                    DskipTests'
            }
        }
    }
    stage('Execute Tests') {
        steps {
            container('custom-dind') {
                sh 'mvn test'
            }
        }
    }
    stage('Setup Test Instance') {
```

```
        steps {
            container('custom-dind') {
                sh 'docker rm -f petclinic-test'

                sh 'docker network rm -f zapnet'
                sh 'docker network create --driver=bridge
                   --subnet=172.16.0.0/24 zapnet'

                sh 'mvn spring-boot:build-image -D spring-
                   boot.build-image.imageName=spring-
                   petclinic -DskipTests'
                sh 'docker run -d --name temp_container
                   spring-petclinic:latest'
                sh 'docker commit temp_container
                   $IMAGE_TAG_TEST'
                sh 'docker rm -f temp_container'

                sh 'docker rm -f petclinic-test'
                sh 'docker run -d --name petclinic-test --
                   net zapnet --ip 172.16.0.2 -p 8080:8080
                   $IMAGE_TAG_TEST'
            }
        }
    }
    stage('SonarQube Scan') {
        steps {
            withSonarQubeEnv('sonarqube') {
                sh ''' $SCANNER_HOME/bin/sonar-scanner -
                   Dsonar.projectName=petclinic-example \
                -Dsonar.java.binaries=. \
                -Dsonar.projectKey=petclinic-example \
                -Dsonar.exclusions=dependency-check-report.
                   html '''
            }
        }
    }
```

```
        stage('Snyk Scan') {
            steps {
                snykSecurity severity: 'critical',
                   snykInstallation: 'snyk@latest', snykTokenId
                   : 'renegyetvai-snyk-api-token', failOnIssues
                   : 'false'
            }
        }
        stage('Docker Scout') {
            steps {
                container('custom-dind') {
                    catchError {
                        // Install Docker Scout
                        sh 'curl -sSfL https://raw.
                           githubusercontent.com/docker/scout-
                           cli/main/install.sh | sh -s -- -b /
                           usr/local/bin'

                        // Analyze and fail on critical or high
                            vulnerabilities
                        sh 'docker-scout cves $IMAGE_TAG_TEST
                           --exit-code --only-severity critical
                           '
                    }
                }
            }
        }
        stage('OWASP ZAP Scan') {
            steps {
                container('custom-dind') {
                    sh 'docker pull softwaresecurityproject/zap
                       -stable'
                    sh 'docker run --net zapnet --name zap --
                       user root -v $(pwd):/zap/wrk/:rw -t
                       softwaresecurityproject/zap-stable zap-
```

```
                    full-scan.py -t https://172.16.0.2:8080
                    -g gen.conf -r report.html -I'
                }
            }
        }
        stage('Nikto Scan') {
            steps {
                container('custom-dind') {
                    sh 'docker run --net zapnet --name nikto --
                       rm frapsoft/nikto -h 172.16.0.2 -p 8080'
                }
            }
        }
        stage('Docker Build') {
            steps {
                container('custom-dind') {
                    script {
                        sh 'docker run -d --name temp_container
                            spring-petclinic:latest'
                        sh 'docker commit temp_container
                           $IMAGE_TAG_LATEST'
                        sh 'docker rm -f temp_container'
                    }
                }
            }
        }
        stage('Docker Push') {
            steps {
                container('custom-dind') {
                    script {
                        sh 'docker push $IMAGE_TAG_LATEST'
                    }
                }
            }
        }
    }
```

```
    post {
        always {
            container('custom-dind') {
                sh 'docker logout'

                // Clean up all containers and networks
                sh 'docker stop petclinic-test'
                sh 'docker rm -f petclinic-test'

                sh 'docker rm -f zap'
                sh 'docker rm -f nikto'

                sh 'docker network rm -f zapnet'

                // Clean up all images
                sh 'docker rmi -f $IMAGE_TAG_TEST'
                sh 'docker rmi -f paketobuildpacks/builder-
                    jammy-base'
                sh 'docker rmi -f paketobuildpacks/run-jammy-
                    base'
                sh 'docker rmi -f spring-petclinic'
                sh 'docker rmi -f rgyetvai/petclinic'
                sh 'docker rmi -f petclinic'
                sh 'docker rmi -f rgyetvai/custom-dind'
                sh 'docker rmi -f custom-dind'
                sh 'docker rmi -f testcontainers/ryuk'
                sh 'docker rmi -f softwaresecurityproject/zap-
                    stable'
                sh 'docker rmi -f frapsoft/nikto'
            }
        }
        success {
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
```

```
        }
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

Realer Anwendungsfall - Jenkinsfile parallel

```
pipeline {
    agent {
        kubernetes {
            cloud 'K8s Cluster 01'
            slaveConnectTimeout 300
            idleMinutes 5
            yaml'''
                apiVersion: v1
                kind: Pod
                metadata:
                  name: custom-build-pod
                  namespace: devops-tools
                spec:
                  affinity:
                    nodeAffinity:
                      requiredDuringSchedulingIgnoredDuringExecution
                          :
                        nodeSelectorTerms:
                        - matchExpressions:
                          - key: kubernetes.io/hostname
                            operator: In
                            values:
                            - k8s-worker-4
                  containers:
```

```
- command:
  - cat
  name: custom-dind-01
  image: rgyetvai/custom-dind:latest
  resources:
    limits:
      cpu: "1"
      memory: 1Gi
    requests:
      cpu: 500m
      memory: 500Mi
  tty: true
  volumeMounts:
  - mountPath: /var/run/docker.sock
    name: docker-sock-volume
  - mountPath: /usr/share/git
    name: shared-workspace
- command:
  - cat
  name: custom-dind-02
  image: rgyetvai/custom-dind:latest
  resources:
    limits:
      cpu: "2"
      memory: 2Gi
    requests:
      cpu: 500m
      memory: 500Mi
  tty: true
  volumeMounts:
  - mountPath: /var/run/docker.sock
    name: docker-sock-volume
  - mountPath: /usr/share/git
    name: shared-workspace
volumes:
- hostPath:
```

```
                    path: /var/run/docker.sock
                    type: Socket
                name: docker-sock-volume
              - name: shared-workspace
                emptyDir: {}
            '''
        }
    }
    options {
        buildDiscarder(logRotator(numToKeepStr: '5'))
        parallelsAlwaysFailFast()
    }
    environment {
        SCANNER_HOME = tool 'sonar-scanner'
        DOCKERHUB_CREDENTIALS = credentials('rgyetvai-dockerhub
            ')
        IMAGE_TAG_TEST = 'rgyetvai/petclinic:testing'
        IMAGE_TAG_LATEST = 'rgyetvai/petclinic:latest'
    }
    stages {
        stage('Prepare Workspace') {
            parallel {
                stage('Git Checkout') {
                    steps {
                        container('custom-dind-01') {
                            script {
                                git branch: 'parallelized-
                                    jenkinsfile',
                                    credentialsId: '
                                        git_jenkins_ba_01',
                                    url: 'git@github.com:
                                        renegyetvai/spring-
                                        petclinic.git'
                                // copy the git repo to the
                                    shared workspace
                                sh 'cp -r ./* /usr/share/git'
```

```
                    }
                }
            }
        }
        stage('Docker Login') {
            steps {
                container('custom-dind-02') {
                    script {
                        sh 'echo
                            $DOCKERHUB_CREDENTIALS_PSW |
                             docker login -u
                            $DOCKERHUB_CREDENTIALS_USR
                            --password-stdin'
                    }
                }
            }
        }
    }
}
stage('Prepare Environment') {
    parallel {
        stage('Compile') {
            steps {
                container('custom-dind-01') {
                    sh 'mvn --version'
                    sh 'mvn -U clean validate compile -
                        DskipTests'
                }
            }
        }
        stage('Setup Test Instance') {
            steps {
                container('custom-dind-02') {
                    // Get shared workspace
                    sh 'cp -r /usr/share/git/* .'
```

```
                        sh 'docker rm -f petclinic-test'

                        sh 'docker network rm -f zapnet'
                        sh 'docker network create --driver=
                            bridge --subnet=172.16.0.0/24
                            zapnet'

                        sh 'mvn dependency:purge-local-
                            repository'
                        sh 'mvn versions:use-latest-
                            versions'
                        sh 'mvn spring-boot:build-image -D
                            spring-boot.build-image.
                            imageName=spring-petclinic -
                            DskipTests'
                        sh 'docker run -d --name
                            temp_container spring-petclinic:
                            latest'
                        sh 'docker commit temp_container
                            $IMAGE_TAG_TEST'
                        sh 'docker rm -f temp_container'

                        sh 'docker rm -f petclinic-test'
                        sh 'docker run -d --name petclinic-
                            test --net zapnet --ip
                            172.16.0.2 -p 8080:8080
                            $IMAGE_TAG_TEST'
                    }
                }
            }
        }
    }
    stage('Test & Scan Sources') {
        steps {
            script {
                getWrappedStages()
```

```
                }
            }
        }
        stage('Docker Build') {
            steps {
                container('custom-dind-02') {
                    script {
                        sh 'docker run -d --name temp_container
                            spring-petclinic:latest'
                        sh 'docker commit temp_container
                           $IMAGE_TAG_LATEST'
                        sh 'docker rm -f temp_container'
                    }
                }
            }
        }
        stage('Docker Push') {
            steps {
                container('custom-dind-02') {
                    script {
                        sh 'docker push $IMAGE_TAG_LATEST'
                    }
                }
            }
        }
    }
    post {
        always {
            container('custom-dind-02') {
                sh 'docker logout'

                // Clean up all containers and networks
                sh 'docker stop petclinic-test'
                sh 'docker rm -f petclinic-test'

                sh 'docker rm -f zap'
```

```
                sh 'docker rm -f nikto'

                sh 'docker network rm -f zapnet'

                // Clean up all images
                sh 'docker rmi -f $IMAGE_TAG_TEST'
                sh 'docker rmi -f paketobuildpacks/builder-
                    jammy-base'
                sh 'docker rmi -f paketobuildpacks/run-jammy-
                    base'
                sh 'docker rmi -f spring-petclinic'
                sh 'docker rmi -f rgyetvai/petclinic'
                sh 'docker rmi -f petclinic'
                sh 'docker rmi -f rgyetvai/custom-dind'
                sh 'docker rmi -f custom-dind'
                sh 'docker rmi -f testcontainers/ryuk'
                sh 'docker rmi -f softwaresecurityproject/zap-
                    stable'
                sh 'docker rmi -f frapsoft/nikto'
            }
        }
        success {
            echo 'Build Success'
        }
        failure {
            echo 'Build Failed'
        }
        unstable {
            echo 'Build Unstable'
        }
        changed {
            echo 'Build Changed'
        }
    }
}
```

```
def getWrappedStages() {
    stages = [:]
    stages["Tests & SAST"] = {
        stage('SAST') {
            container('custom-dind-01') {
                stage('Tests & SAST') {
                    sh """
                        echo "Executing Tests & SAST"
                    """
                }
                parallel nestedStagesOne()
            }
        }
    }
    stages["Snyk Scan"] = {
        // Snyks Jenkins plugin is not compatible with the
           container step
        stage('Snyk Scan') {
            snykSecurity severity: 'critical', snykInstallation
               : 'snyk@latest', snykTokenId: 'renegyetvai-snyk-
               api-token', failOnIssues: 'false'
        }
    }
    stages["Prepare, Build & Scan"] = {
        stage('DAST') {
            container('custom-dind-02') {
                stage('Prepare, Build & Scan') {
                    sh """
                        echo "Executing Prepare, Build & Scan"
                    """
                }
                parallel nestedStagesTwo()
            }
        }
    }
    parallel stages
```

```
}

def nestedStagesOne() {
    stages = [:]
    stages["Unit & Integration Tests"] = {
        stage('Unit & Integration Tests') {
            sh 'mvn test'
        }
    }
    stages["SonarQube Scan"] = {
        stage('SonarQube Scan') {
            withSonarQubeEnv('sonarqube') {
                sh ''' $SCANNER_HOME/bin/sonar-scanner -Dsonar.
                    projectName=petclinic-example \
                -Dsonar.java.binaries=. \
                -Dsonar.projectKey=petclinic-example \
                -Dsonar.exclusions=dependency-check-report.html
                    '''
            }
        }
    }
    return stages
}

def nestedStagesTwo() {
    stages = [:]
    stages["OWASP ZAP Scan"] = {
        stage('OWASP ZAP Scan') {
            sh 'docker pull softwaresecurityproject/zap-stable'
            sh 'docker run --net zapnet --name zap --user root
               -v $(pwd):/zap/wrk/:rw -t
                softwaresecurityproject/zap-stable zap-full-scan
                .py -t https://172.16.0.2:8080 -g gen.conf -r
                report.html -I'
        }
    }
```

```
    stages["Nikto Scan"] = {
        stage('Nikto Scan') {
            //sh 'docker run --net zapnet --name nikto --rm
                frapsoft/nikto -h https://172.16.0.2:8080'
            sh 'docker run --net zapnet --name nikto --rm
                frapsoft/nikto -h 172.16.0.2 -p 8080'
        }
    }
    stages["Docker Scout"] = {
        stage('Docker Scout') {
            try {
                // Install Docker Scout
                sh 'curl -sSfL https://raw.githubusercontent.
                    com/docker/scout-cli/main/install.sh | sh -s
                     -- -b /usr/local/bin'

                // Analyze and fail on critical or high
                    vulnerabilities
                sh 'docker-scout cves $IMAGE_TAG_TEST --exit-
                    code --only-severity critical'
            } catch (Exception e) {
                echo "Docker Scout failed: ${e}"
            }
        }
    }
    return stages
}
```

A.5 Python Skripte – Auswertung

A.5.1 Python Skript – Ressourcenauswertung

```
import os
import sys
from datetime import datetime

import matplotlib.pyplot as plt
import pandas as pd
from pandas import DataFrame

# Set the debug mode to True to see the debug messages
DEBUG = True
FILE_DIR_01 = "assets/data/board_a/test_a/"
FILE_DIR_02 = "assets/data/board_a/test_b/B3/"
FILE_DIR_03 = "assets/data/board_a/real/"
FILE_DIR_04 = "assets/data/board_b/Test_Real/"


def remove_data(data: DataFrame, lower_than: float) ->
    DataFrame:
     for column in data.columns:
          if column != 'Time':
              # Replace values lower than 'lower_than' with pd.NA
              data[column] = data[column].apply(
                  lambda x: pd.NA if x < lower_than else x)

     # remove empty columns
     data = data.dropna(axis=1, how='all')
     return data


def format_time(timestamps: list) -> list:
     dates = [
```

```
        datetime.fromtimestamp(ts / 1000).strftime("%Y-%m-%d %H
            :%M:%S")
        for ts in timestamps
    ]
    start_time = datetime.strptime(dates[0], "%Y-%m-%d %H:%M:%S
        ")
    time = [
        (datetime.strptime(date, "%Y-%m-%d %H:%M:%S") -
            start_time).seconds
        for date in dates
    ]
    return time


def trim_leading_zeros(df):
    # Iterate from the start to find the first row with a non-
        zero value in the second column
    for start in range(len(df)):
        if df.iloc[start, 1] != 0:
            break
    # Slice the DataFrame from the first non-zero row to the
        end
    return df.iloc[start:]


def trim_trailing_zeros(df):
    # Iterate from the end to find the last row with a non-zero
        value in the second column
    for end in range(len(df) - 1, -1, -1):
        if df.iloc[end, 1] != 0:
            break
    # Slice the DataFrame from the start to the last non-zero
        row
    return df.iloc[:end + 1]
```

```
def optimize_df(data: DataFrame) -> DataFrame:

    # create new dataframes for each column with the Time
        column
    dataframes = [data[["Time", column]] for column in data.
        columns[1:]]

    # remove all NA values from the dataframes
    dataframes = [df.dropna() for df in dataframes]

    # cut for each dataframe the zeros at the frond and back
    trimmed_dataframes = []
    for i, df in enumerate(dataframes):
        df = trim_leading_zeros(df)
        df = trim_trailing_zeros(df)
        trimmed_dataframes.append(df)
    dataframes = trimmed_dataframes

    # convert timestamps to relative seconds from the start of
        each dataframe
    for i, df in enumerate(dataframes):
        df["Time"] = format_time(df["Time"])

    # merge all dataframes to one dataframe by the Time column
    data = dataframes[0]

    for df in dataframes[1:]:
        data = pd.merge(data, df, on="Time")

    return data


def read_data(file: str, target_unit: str, compare: bool =
    False, min: int = 0, max: int = sys.maxsize, filter: str =
    "") -> any:
    with open(file, "r") as f:
```

```
        df = pd.read_csv(f, delimiter=",", header=1)

    if DEBUG:
        print(df)

    # Change all string 'undefined' to NaN
    df = df.replace("undefined", pd.NA)

    if compare:
        df = optimize_df(df)
    else:
        df["Time"] = format_time(df["Time"])

    # Fix the time as it is in milliseconds and we need it in
       relative seconds from the start
    # timeline = format_time(df["Time"])
    timeline = df["Time"]

    # Switch case to convert the data rows to the target unit
    if target_unit == "MiB":
        df = df.apply(pd.to_numeric, errors="coerce") / (1024
           ** 2)
    elif target_unit == "GiB":
        df = df.apply(pd.to_numeric, errors="coerce") / (1024
           ** 3)
    elif target_unit == "Milliseconds":
        df = df.apply(pd.to_numeric, errors="coerce") * 1000
    elif target_unit == "Percentage":
        df = df.apply(pd.to_numeric, errors="coerce") * 100
    elif target_unit == "Number":
        df = df.apply(pd.to_numeric, errors="coerce")
    else:
        raise ValueError("Invalid target unit")

    # replace time row with the new time since it gets changed
       in the switch case
```

```
    df["Time"] = timeline

    # Remove all rows with a time less than 'min' and greater
        than 'max'
    for column in df.columns:
        if column != 'Time':
            df = df.drop(df[(df["Time"] < min) | (df["Time"] >
                max)].index)

    # remove all columns after the Time column that not contain
        'filter' in their name
    if filter != "":
        for column in df.columns:
            if column != 'Time' and filter not in column:
                df = df.drop(column, axis=1)

    # remove empty columns
    df = df.dropna(axis=1, how='all')

    return df


def plot_data(data: DataFrame, title: str, ylabel: str, xlabel:
    str = "Time (s)", ticksize: int = 20) -> None:
    if DEBUG:
        print(data)

    fig, ax = plt.subplots(figsize=(10, 5))

    time_column = data["Time"]
    other_columns = data.columns[1:]

    # Dynamically plot each of the other columns against the '
        Time' column
    for column in other_columns:
        column = pd.to_numeric(data[column], errors='coerce')
```

```
        ax.plot(time_column, column)

    ax.set_title(title, fontsize=18, fontfamily='serif')
    ax.set_xlabel(xlabel, fontsize=22, fontfamily='serif')
    ax.set_ylabel(ylabel, fontsize=22, fontfamily='serif')
    # ax.set_xticks(data["Time"][::len(data) // 10])
    ax.set_xticks(data["Time"][::ticksize])

    ax.grid(True, axis="x", which="major")
    ax.grid(True, axis="y", which="major")

    # fig.legend(other_columns, loc='outside lower center')
    plt.xticks(fontsize=16, fontfamily='serif')
    plt.yticks(fontsize=16, fontfamily='serif')
    plt.legend(other_columns, fontsize=16)
    plt.show()


def plot_data_splitted(data: DataFrame, title: str, ylabel: str
    , xlabel: str = "Time (s)", ticksize: int = 20, cols: int =
    2) -> None:
    # Split the dataframe into n dataframes based on the number
        of columns, excluding the Time column
    dataframes = [data[["Time", column]] for column in data.
       columns[1:]]

    # Determine the number of rows and columns for the subplots
    total_plots = len(dataframes)
    ncols = cols  # Maximum number of columns per row
    nrows = total_plots // ncols + \
        (1 if total_plots % ncols > 0 else 0)  # Calculate rows
            needed

    # Adjust the figure size dynamically based on the number of
        subplots
    # Adjust width and height based on ncols and nrows
```

```
fig = plt.figure(figsize=(20, 5 * nrows))

# Adjust for the calculated layout
spec = fig.add_gridspec(ncols=ncols, nrows=nrows)

shared_ax = None  # Initialize variables for shared axes

# Loop through each dataframe and plot it in a separate
    subplot
for i, df in enumerate(dataframes):
    # Calculate row and column position for current subplot
    row = i // ncols
    col = i % ncols

    if shared_ax is None:
        # For the first subplot, just create it
        ax = fig.add_subplot(spec[row, col])
        shared_ax = ax  # Set the first subplot's axes as
            the shared axes
    else:
        # Subsequent plots will share axes
        ax = fig.add_subplot(
            spec[row, col], sharex=shared_ax, sharey=
                shared_ax)

    # Convert the non-Time column to numeric, handling
        errors by coercion to NA
    data_series = pd.to_numeric(df[df.columns[1]], errors='
        coerce')
    ax.plot(df["Time"], data_series)  # Plot data
    # Set title for each subplot
    ax.set_title(f"{df.columns[1]}", fontsize=16,
        fontfamily='serif')
    ax.set_xlabel(xlabel, fontsize=18, fontfamily='serif')
    ax.set_ylabel(ylabel, fontsize=18, fontfamily='serif')
    ax.grid(True)
```

```
            # Adjust the x-ticks to reduce crowding, if necessary
            tick_spacing = max(1, len(df["Time"]) // ticksize)
            ax.set_xticks(df["Time"][::tick_spacing])

            plt.xticks(fontsize=16, fontfamily='serif')
            plt.yticks(fontsize=16, fontfamily='serif')

    # Set a title for the whole figure
    plt.suptitle(title, fontsize=20, fontfamily='serif')

    # Adjust layout to add more space between plots, accounting
        for multiple rows
    fig.subplots_adjust(hspace=0.5, wspace=0.4)

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()


if __name__ == "__main__":
    # create list of file names from files inside the assets
       folder
    file_names = os.listdir(FILE_DIR_01)
    file_names.sort()

    dataframe = read_data(
        FILE_DIR_01 + file_names[4], "Milliseconds", False,
            2600, 6000)
    dataframe = remove_data(dataframe, 8)
    plot_data(dataframe, "CPU Usage - Test A", "Usage (ms)")

    dataframe = read_data(
        FILE_DIR_01 + file_names[4], "Milliseconds", True,
            filter="60%")
    dataframe = remove_data(dataframe, 8)
```

```
plot_data(dataframe, "CPU Usage - Test A - Compared", "
    Usage (ms)",
            xlabel="Relative Time (s)", ticksize=5)
plot_data_splitted(dataframe, "CPU Usage - Test A -
    Compared",
                    "Usage (ms)", xlabel="Relative Time (s)
                        ", ticksize=5)

# List all files in the directory
file_names = os.listdir(FILE_DIR_02)
file_names.sort()

dataframe = read_data(
    FILE_DIR_02 + file_names[4], "Milliseconds", False)
dataframe = remove_data(dataframe, 8)
plot_data(dataframe, "CPU Usage - Test B", "Usage (ms)",
    ticksize=40)

dataframe = read_data(
    FILE_DIR_02 + file_names[4], "Milliseconds", True,
        filter="6 serial steps")
dataframe = remove_data(dataframe, 8)
plot_data(dataframe, "CPU Usage - Test B - Compared",
          "Usage (ms)", xlabel="Relative Time (s)")
plot_data_splitted(dataframe, "CPU Usage - Test B -
    Compared",
                    "Usage (ms)", xlabel="Relative Time (s)
                        ", ticksize=5)

# List all files in the directory
file_names = os.listdir(FILE_DIR_03)
file_names.sort()

dataframe = read_data(
    FILE_DIR_03 + file_names[4], "Milliseconds")
dataframe = remove_data(dataframe, 8)
```

```
plot_data(dataframe, "CPU Usage - Real System", "Usage (ms)
    ")

dataframe = read_data(
    FILE_DIR_03 + file_names[4], "Milliseconds", True)
dataframe = remove_data(dataframe, 8)
plot_data(dataframe, "CPU Usage - Real System - Compared",
    "Usage (ms)",
          xlabel="Relative Time (s)")
plot_data_splitted(dataframe, "CPU Usage - Real System -
    Compared",
                   "Usage (ms)", xlabel="Relative Time (s)
                       ", ticksize=5)

dataframe = read_data(
    FILE_DIR_03 + file_names[5], "Number")  # Already in
        MiB from source
dataframe = remove_data(dataframe, 5)
plot_data(dataframe, "Memory Usage - Real System", "Usage (
    MiB)")

dataframe = read_data(
    # Already in MiB from source
    FILE_DIR_03 + file_names[5], "Number", True)
dataframe = remove_data(dataframe, 5)
plot_data(dataframe, "Memory Usage - Real System - Compared
    ",
          "Usage (MiB)", xlabel="Relative Time (s)")
plot_data_splitted(dataframe, "Memory Usage - Real System -
    Compared",
                   "Usage (MiB)", xlabel="Relative Time (s)
                       ", ticksize=5)

dataframe = read_data(
    FILE_DIR_03 + file_names[6], "Number")
dataframe = remove_data(dataframe, 50)
```

```
    plot_data(dataframe, "Threads - Real System", "Count (
        Number)")

    dataframe = read_data(
        FILE_DIR_03 + file_names[6], "Number", True)
    dataframe = remove_data(dataframe, 50)
    plot_data(dataframe, "Threads - Real System - Compared",
              "Count (Number)", xlabel="Relative Time (s)")
    plot_data_splitted(dataframe, "Threads - Real System -
        Compared",
                       "Count (Number)", xlabel="Relative Time
                           (s)", ticksize=5)

    # List all files in the directory
    file_names = os.listdir(FILE_DIR_04)
    file_names.sort()

    dataframe = read_data(
        FILE_DIR_04 + file_names[4], "Number")
    dataframe = remove_data(dataframe, 0.1)
    plot_data(dataframe, "Context switches & Interrupts -
        Pipeline Host",
              "Count (Number)", ticksize=5)
```

A.5.2 Python Skript – Versuch 1 – Amdahl Bezug

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Path to the CSV file
file_path = 'assets/data/general/test_a_general.csv'

# Read the CSV data into a DataFrame
with open(file_path, "r") as f:
```

```
    df = pd.read_csv(f, sep=',')

# Calculate measured speedup
T_serial = df[df["Test"] == "Serial"]["Dauer Gesamt (s)"].iloc
    [0]
df["Measured Speedup"] = T_serial / df["Dauer Gesamt (s)"]

# Amdahl's Law for theoretical speedup
df["Theoretical Speedup"] = df.apply(
    lambda row: 1 / ((1 - row["Anteil Parallel"]) + (row["
        Anteil Parallel"] / row["Worker Parallel"])), axis=1)

# Logarithmic function for regression


def log_func(x, a, b):
    return a + b * np.log(x)


# Plot setup
plt.figure(figsize=(14, 8))
colors = ['red', 'blue', 'green']
parallel_portions = df["Anteil Parallel"].unique()[1:]  #
    Exclude 0 portion
workers = df["Worker Parallel"].unique()

# Root point for logarithmic regression
root_point_x = [1]
root_point_y = [1]

# Plotting
for i, p in enumerate(parallel_portions):
    x_data_with_root = np.append(
        # Adding root point
        root_point_x, df[df["Anteil Parallel"] == p]["Worker
            Parallel"])
```

```
    y_data_with_root = np.append(
        root_point_y, df[df["Anteil Parallel"] == p]["Measured
            Speedup"])

    # Measured speedup and theoretical speedup plot
    plt.plot(x_data_with_root, y_data_with_root, 'o-',
             color=colors[i], label=f'{int(p*100)}% Parallel (
                 Measured)')
    plt.plot(df[df["Anteil Parallel"] == p]["Worker Parallel"],
        df[df["Anteil Parallel"] == p]
             ["Theoretical Speedup"], '--', color='grey', label
                =f'{int(p*100)}% Parallel (Theoretical)')

    # Logarithmic regression
    if len(x_data_with_root) > 1:
        popt, _ = curve_fit(log_func, x_data_with_root,
            y_data_with_root)
        x_fit = np.linspace(min(x_data_with_root), max(
            x_data_with_root), 100)
        y_fit = log_func(x_fit, *popt)
        plt.plot(x_fit, y_fit, ':', color=colors[i], label=f'{
                 int(p*100)}% Parallel (Logarithmic Regression)
                     ')

plt.title(
    'Speedup for measured data and Amdahl\'s theoretical
        predictions', fontsize=18, fontfamily='serif')
plt.xlabel('Workers (Number)', fontsize=22, fontfamily='serif')
plt.ylabel('Speedup (Factor)', fontsize=22, fontfamily='serif')
plt.xticks(np.append(root_point_x, workers), fontsize=16,
    fontfamily='serif')
plt.yticks(fontsize=16, fontfamily='serif')
plt.legend(fontsize=16)
plt.grid(True)
plt.show()
```

A.5.3 Python Skript – Versuch 2 – Gustafson Bezug

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Path to the CSV file
file_path = 'assets/data/general/test_b_general.csv'

# Read the CSV data into a DataFrame
with open(file_path, "r") as f:
    df = pd.read_csv(f, sep=',')

# Calculate speedup using the given formula
df['Speedup'] = 1 + (df['Steps / Worker Parallel'] - 1) * \
    (df['Anteil Parallel'] / 100)

# Prepare data for plotting
target_data = df.pivot_table(index='Steps / Worker Parallel',
                             columns='Steps Seriell', values='
                                Speedup', aggfunc='mean')
target_data.columns = [f"{c} serielle Schritte" for c in
    target_data.columns]

# Gustafson's speedups calculation for 20% to 70% in steps of
    10%
percentages_serial = np.arange(20, 80, 10) / 100
steps_parallel = target_data.index
gustafson_speedups = {s: [n + (1 - n) * s for n in
    steps_parallel]
                      for s in percentages_serial}

# Initialize the plot for combined measured data and Gustafson'
    s theoretical predictions
plt.figure(figsize=(10, 8))
```

```
# Plot the measured data
for column, color in zip(target_data.columns, ['red', 'blue', '
    green']):
     plt.plot(target_data.index,
              target_data[column], marker='o', label=column,
                  color=color)

# Define shades of grey for each Gustafson's theoretical
    speedups
# Darker to lighter shades of grey
grey_shades = ['0.1', '0.2', '0.3', '0.4', '0.5', '0.6']

# Plot Gustafson's theoretical speedups with different shades
    of grey
for (s, speedups), shade in zip(gustafson_speedups.items(),
    grey_shades):
     plt.plot(steps_parallel, speedups, label=f'Gustafson {
              int(s*100)}%', marker='x', color=shade, linestyle
                  ='--')

plt.title('Speedups for measured data and Gustafson\'s
    theoretical predictions', fontsize=18, fontfamily='serif')
plt.xlabel('Worker (Number)', fontsize=22, fontfamily='serif')
plt.ylabel('Speedup (Factor)', fontsize=22, fontfamily='serif')
plt.xticks(target_data.index, fontsize=16, fontfamily='serif')
plt.yticks(fontsize=16, fontfamily='serif')
plt.legend(fontsize=16)
plt.grid(True)
plt.show()
```

A.5.4 Python Skript – Versuch 2 – Zeitverlauf

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

```
# Path to the CSV file
file_path = 'assets/data/general/test_b_general.csv'

# Read the CSV data into a DataFrame
with open(file_path, "r") as f:
    df = pd.read_csv(f, sep=',')

# Mapping from Test labels to a numeric value for regression
    analysis
df['Test_Numeric'] = df.index + 1

# create sub dataframes for each serial step portion
df_2 = df[df['Steps Seriell'] == 2]
df_4 = df[df['Steps Seriell'] == 4]
df_6 = df[df['Steps Seriell'] == 6]

df_all = [df_2, df_4, df_6]
labels = ['2 Steps Serial', '4 Steps Serial', '6 Steps Serial']

# Initialize the plot for combined measured data
plt.figure(figsize=(10, 8))

# Plot the measured data and regression line
for df_subset, label in zip(df_all, labels):
    # Plotting the data points
    plt.plot(df_subset['Test'], df_subset['Dauer Gesamt'],
             marker='o', label=label)

    # Fitting a linear regression model
    model = np.polyfit(df_subset['Test_Numeric'], df_subset['
        Dauer Gesamt'], 1)
    predict = np.poly1d(model)

    # Generating x values for the regression line (from min to
        max Test_Numeric)
    x_regression = np.linspace(
```

```
            min(df_subset['Test_Numeric']), max(df_subset['
                Test_Numeric']), 100)
        # Generating y values for the regression line
        y_regression = predict(x_regression)

        # Plotting the regression line
        plt.plot(x_regression, y_regression, linestyle='--',
                 color='grey', label=f'{label} Trend')

plt.title('Time related behaviour of the measured data',
          fontsize=18, fontfamily='serif')
# plt.xlabel('Experiment', fontsize=22, fontfamily='serif')
plt.ylabel('Duration (s)', fontsize=22, fontfamily='serif')
plt.xticks(rotation=90, fontsize=14, fontfamily='serif')
plt.yticks(np.arange(0, max(df['Dauer Gesamt']) +
           100, 100), fontsize=16, fontfamily='serif')
plt.legend(fontsize=16)
plt.grid(True)
plt.tight_layout()  # Adjust layout to make room for the
   rotated x-axis labels
plt.show()
```

A.6 Jenkins Plugins

Software	Version
Eclipse Temurin installer Plugin	1.5
Ant Plugin	497.v94e7d9fffa_b_9
OWASP Markup Formatter Plugin	162.v0e6ec0fcfcf6
Apache HttpComponents Client 4.x API Plugin	4.5.14-208.v438351942757
Apache HttpComponents Client 5.x API Plugin	5.3.1-1.0
Authentication Tokens API Plugin	1.53.v1c90fd9191a_b_
Blue Ocean	1.27.11
Bitbucket Pipeline for Blue Ocean	1.27.11
Common API for Blue Ocean	1.27.11
Config API for Blue Ocean	1.27.11
Blue Ocean Core JS	1.27.11
Dashboard for Blue Ocean	1.27.11
Display URL for Blue Ocean	2.4.2
Events API for Blue Ocean	1.27.11
Git Pipeline for Blue Ocean	1.27.11
GitHub Pipeline for Blue Ocean	1.27.11
i18n for Blue Ocean	1.27.11
JWT for Blue Ocean	1.27.11
Personalization for Blue Ocean	1.27.11
Pipeline implementation for Blue Ocean	1.27.11
Blue Ocean Pipeline Editor	1.27.11
Pipeline SCM API for Blue Ocean	1.27.11
REST API for Blue Ocean	1.27.11
REST Implementation for Blue Ocean	1.27.11
Web for Blue Ocean	1.27.11
Bootstrap 5 API Plugin	5.3.2-3
bouncycastle API Plugin	2.30.1.77-225.v26ea_c9455fd9
Branch API Plugin	2.1152.v6f101e97dd77
Build Timeout	1.32
Caffeine API Plugin	3.1.8-133.v17b_1ff2e0599
Checks API plugin	2.0.2
Cloud Statistics Plugin	336.v788e4055508b_
Bitbucket Branch Source Plugin	880.vcf4056c5a_71f
Folders Plugin	6.858.v898218f3609d

Tabelle A.16: Verwendete Jenkins Plugins - Teil 1

Software	Version
Command Agent Launcher Plugin	107.v773860566e2e
commons-lang3 v3.x Jenkins API Plugin	3.13.0-62.v7d18e55f51e2
commons-text API Plugin	1.11.0-95.v22a_d30ee5d36
Credentials Plugin	1319.v7eb_51b_3a_c97b_
Credentials Binding Plugin	657.v2b_19db_7d6e6d
Dark Theme	416.v535839b_c4e88
OWASP Dependency-Check Plugin	5.5.0
Display URL API	2.200.vb_9327d658781
CloudBees Docker Build and Publish plugin	1.4.0
docker-build-step	2.11
Docker Commons Plugin	439.va_3cb_0a_6a_fb_29
Docker API Plugin	3.3.4-86.v39b_a_5ede342c
Docker plugin	1.5
Docker Pipeline	572.v950f58993843
Durable Task Plugin	550.v0930093c4b_a_6
ECharts API Plugin	5.4.3-3
Email Extension Plugin	2.105
Favorite	2.208.v91d65b_7792a_c
Font Awesome API Plugin	6.5.1-2
Git plugin	5.2.1
Git client plugin	4.7.0
GitHub plugin	1.38.0
GitHub API Plugin	1.318-461.v7a_c09c9fa_d63
GitHub Branch Source Plugin	1781.va_153cda_09d1b_
GitHub Integration Plugin	0.7.0
Gradle Plugin	2.10
Gson API Plugin	2.10.1-15.v0d99f670e0a_7
Handy Uri Templates 2.x API Plugin	2.1.8-30.v7e777411b_148
HTML Publisher plugin	1.33
Instance Identity	185.v303dc7c645f9
Ionicons API	56.v1b_1c8c49374e
Jackson 2 API Plugin	2.16.2-378.v7e79818f53ce
Jakarta Activation API	2.1.3-1
Jakarta Mail API	2.1.3-1

Tabelle A.18: Verwendete Jenkins Plugins - Teil 2

Software	Version
Javadoc Plugin	243.vb_b_503b_b_45537
JavaBeans Activation Framework (JAF) API	1.2.0-6
JavaMail API	1.6.2-9
JAXB plugin	2.3.9-1
Oracle Java SE Development Kit Installer Plugin	73.vddf737284550
Design Language	1.27.11
Java JSON Web Token (JJWT) Plugin	0.11.5-77.v646c772fddb_0
Joda Time API Plugin	2.12.7-29.v5a_b_e3a_82269a_
JQuery3 API Plugin	3.7.1-1
JSch dependency plugin	0.2.16-86.v42e010d9484b_
JSON Api Plugin	20240303-41.v94e11e6de726
JSON Path API Plugin	2.9.0-33.v2527142f2e1d
JUnit Plugin	1259.v65ffcef24a_88
Kubernetes plugin	4186.v1d804571d5d4
Kubernetes Client API Plugin	6.10.0-240.v57880ce8b_0b_2
Kubernetes Credentials Plugin	0.11
Mailer Plugin	470.vc91f60c5d8e2
Matrix Authorization Strategy Plugin	3.2.2
Matrix Project Plugin	822.824.v14451b_c0fd42
Maven Integration plugin	3.23
Metrics Plugin	4.2.21-449.v6960d7c54c69
Mina SSHD API :: Common	2.12.0-90.v9f7fb_9fa_3d3b_
Mina SSHD API :: Core	2.12.0-90.v9f7fb_9fa_3d3b_
OkHttp Plugin	4.11.0-172.vda_da_1feeb_c6e
PAM Authentication plugin	1.10
Pipeline: Build Step	540.vb_e8849e1a_b_d8
Pipeline: GitHub Groovy Libraries	42.v0739460cda_c4
Pipeline Graph Analysis Plugin	216.vfd8b_ece330ca_
Pipeline: Groovy Libraries	704.vc58b_8890a_384
Pipeline: Input Step	477.v339683a_8d55e
Pipeline: Milestone Step	111.v449306f708b_7
Pipeline: Model API	2.2184.v0b_358b_953e69
Pipeline: Declarative	2.2184.v0b_358b_953e69
Pipeline: Declarative Extension Points API	2.2184.v0b_358b_953e69
Pipeline: REST API Plugin	2.34

Tabelle A.20: Verwendete Jenkins Plugins - Teil 3

Software	Version
Pipeline: Stage Step	305.ve96d0205c1c6
Pipeline: Stage Tags Metadata	2.2184.v0b_358b_953e69
Pipeline: Stage View Plugin	2.34
Plain Credentials Plugin	179.vc5cb_98f6db_38
Plugin Utilities API Plugin	3.8.0
Prism API Plugin	1.29.0-11
Prometheus metrics plugin	2.5.1
Pub-Sub "light"Bus	1.18
Resource Disposer Plugin	0.23
SCM API Plugin	683.vb_16722fb_b_80b_
Script Security Plugin	1326.vdb_c154de8669
Simple Theme Plugin	176.v39740c03a_a_f5
SnakeYAML API Plugin	2.2-111.vc6598e30cc65
Snyk Security Plugin	4.0.2
SonarQube Scanner for Jenkins	2.17.2
Server Sent Events (SSE) Gateway Plugin	1.26
SSH Credentials Plugin	322.v124df57ed808
SSH Build Agents plugin	2.948.vb_8050d697fec
SSH server	3.322.v159e91f6a_550
Structs Plugin	337.v1b_04ea_4df7c8
Theme Manager	215.vc1ff18d67920
Timestamper	1.26
Token Macro Plugin	400.v35420b_922dcb_
Trilead API Plugin	2.142.v748523a_76693
Variant Plugin	60.v7290fc0eb_b_cd
Pipeline	596.v8c21c963d92d
Pipeline: API	1291.v51fd2a_625da_7
Pipeline: Basic Steps	1049.v257a_e6b_30fb_d
Pipeline: Groovy	3883.vb_3ff2a_e3eea_f
Pipeline: Nodes and Processes	1331.vc8c2fed35334
Pipeline: Job	1385.vb_58b_86ea_fff1
Pipeline: Multibranch	773.vc4fe1378f1d5
Pipeline: SCM Step	415.v434365564324
Pipeline: Step API	657.v03b_e8115821b_
Pipeline: Supporting APIs	865.v43e78cc44e0d
Workspace Cleanup Plugin	0.45

Tabelle A.22: Verwendete Jenkins Plugins - Teil 4

A.7 Versuch – Sequenzdiagramm

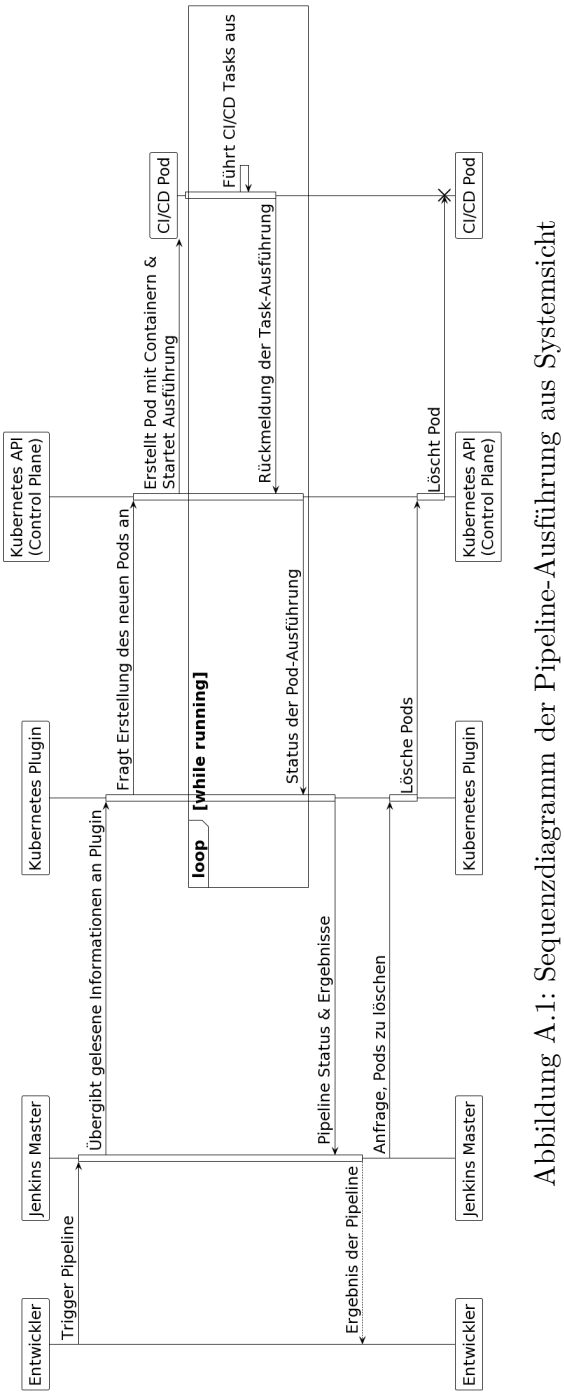

Abbildung A.1: Sequenzdiagramm der Pipeline-Ausführung aus Systemsicht

Glossar

Ansible Playbooks YAML-Dateien, die eine Reihe von Anweisungen für die Automatisierung von IT-Prozessen mit Ansible definieren. Playbooks beschreiben, welche Aufgaben auf Zielservern ausgeführt werden sollen, um Systemkonfigurationen, Softwarebereitstellung und andere automatisierte Prozesse zu verwalten..

API-Token Ein Sicherheitsschlüssel, der als Identifikations- und Authentifizierungsmittel bei der Interaktion mit einer API dient. Er ermöglicht es, autorisierte Anfragen an eine API zu senden und den Zugriff auf spezifische Ressourcen oder Dienste zu kontrollieren..

Batch Eine Sammlung von Daten oder Aufgaben, die als eine Einheit zur Verarbeitung eingereicht werden. In der Computertechnik bezieht sich Batchverarbeitung auf die Ausführung einer Serie von Jobs ohne manuelle Intervention..

cAdvisor Container Advisor ist ein Open-Source-Tool zur Überwachung der Ressourcennutzung und Leistungsdaten von laufenden Containern. cAdvisor sammelt, aggregiert und präsentiert Informationen über CPU, Speicher, Netzwerk und Dateisystemnutzung von Containern..

CI/CD-Pipelines Automatisierte Prozesse in der Softwareentwicklung, die Continuous Integration (CI) und Continuous Delivery bzw. Continuous Deployment (CD) umfassen. CI automatisiert das Zusammenführen und Testen von Codeänderungen, während CD den automatischen Transfer des getesteten Codes in verschiedene Umgebungen bis hin zur Produktion ermöglicht..

Datensenke Ein Zielort oder System, an das Daten zur Speicherung oder weiteren Verarbeitung gesendet werden. Datensenken können Datenbanken, Dateisysteme oder andere Datenspeicher- und Verarbeitungsdienste sein..

Deployments Eine Ressource in Kubernetes, die die Bereitstellung und das Update von Anwendungen oder Diensten steuert. Deployments ermöglichen die Deklaration des gewünschten Zustands für eine Gruppe von Pods und stellen sicher, dass der tatsächliche Zustand diesem entspricht..

DevOps DevOps ist ein Ansatz, der die Entwicklung und den Betrieb von Software näher zusammenbringt.

DevSecOps DevSecOps ist ein Ansatz, der die Entwicklung, den Betrieb und die Sicherheit von Software näher zusammenbringt.

Dockerfile Eine Textdatei, die eine Reihe von Befehlen enthält, um ein Docker-Image zu erstellen. Diese Befehle können das Hinzufügen von Dateien, das Setzen von Umgebungsvariablen und das Ausführen von Befehlen innerhalb des Containers umfassen..

Helm Chart Helm Charts sind Sammlungen von Dateien, die es ermöglichen, vorkonfigurierte Kubernetes-Ressourcen als eine einzige Einheit zu verwalten. Sie erleichtern die Installation, das Upgrade und die Verwaltung von Kubernetes-Anwendungen..

Jenkinsfiles Ein Jenkinsfile ist eine Textdatei, die in der Jenkins-Pipeline-Syntax geschrieben ist und definiert, wie die CI/CD-Pipeline für ein Projekt automatisiert wird. Es ermöglicht die Versionierung des Pipeline-Codes zusammen mit dem Anwendungscode..

Maven Ein Werkzeug zur Automatisierung des Build-Prozesses in der Softwareentwicklung, insbesondere für Java-Projekte. Es hilft bei der Verwaltung von Projektabhängigkeiten, Build-Prozessen und anderen Aspekten..

Pods Die kleinste und einfachste Einheit im Kubernetes-Ökosystem, die eine oder mehrere Container enthält, die auf demselben Host gemeinsame Ressourcen teilen und gemeinsam verwaltet werden..

Services Eine Ressource in Kubernetes, die eine logische Gruppe von Pods definiert und einen stabilen Endpunkt für den Zugriff auf diese Pods bereitstellt. Services ermöglichen die Skalierung und den Lastenausgleich von Anwendungen in Kubernetes..

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original