HAW HAMBURG

BACHELOR THESIS
Simon Schwarzkopf

Energiesparpotentiale in CI/CD-Pipelines: Vergleichende Analyse ausgewählter Best Practices

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

Simon Schwarzkopf

Energiesparpotentiale in CI/CD-Pipelines: Vergleichende Analyse ausgewählter Best Practices

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

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Klaus-Peter Kossakowski

Eingereicht am: 23. September 2025

Simon Schwarzkopf

Thema der Arbeit

Energiesparpotentiale in CI/CD-Pipelines: Vergleichende Analyse ausgewählter Best Practices

Stichworte

CI/CD, Energieverbrauch, Best Practices, Caching, Parallelisierung, GitLab, Green Software Engineering, Ressourcenoptimierung, Energieeffizienz, Green in ICT

Kurzzusammenfassung

Die Optimierung von CI/CD-Pipelines birgt ein erhebliches Energiesparpotential. Daher untersucht diese Bachelorarbeit, inwiefern der Einsatz von Dependency-Caching und Parallelisierung zu Effizienzgewinnen führen kann. Als methodische Grundlage dient eine vergleichende Analyse der Optimierungsstrategien in einer kontrollierten Versuchsumgebung anhand einer Beispiel-Pipeline aus der Praxis. Die Ergebnisse zeigen, dass durch die Maßnahmen sowohl die Laufzeit als auch der Energieverbrauch der Pipeline signifikant reduziert wird. Insbesondere die Kombination beider Praktiken erzielt durch Synergieeffekte die stärkste Effizienzsteigerung. Durch ihre Erkenntnisse leistet die Arbeit einen Beitrag zur Quantifizierung der Wirksamkeit ressourcenschonender Maßnahmen in der Softwareentwicklung und fördert darüber hinaus die Sichtbarkeit des durch den ICT-Sektor verursachten Energieverbrauchs.

Simon Schwarzkopf

Title of Thesis

Energy saving opportunities in CI/CD pipelines: Comparative analysis of selected best practices

Keywords

CI/CD, Energy Consumption, Best Practices, Caching, Parallelization, GitLab, Green Software Engineering, Resource Optimization, Energy Efficiency, Green in ICT

Abstract

The optimization of CI/CD pipelines offers considerable potential for energy savings. This bachelor's thesis therefore examines the extent to which the use of dependency caching and parallelization can lead to efficiency gains. The methodological basis is a comparative analysis of the optimization strategies in a controlled test environment using a sample pipeline from practice. The results show that these measures significantly reduce both the runtime and energy consumption of the pipeline. In particular, the combination of both practices achieves the greatest efficiency gains through synergy effects. With its findings, this thesis contributes to quantifying the effectiveness of resource-saving measures in software development and also promotes the visibility of energy consumed by the ICT sector.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **viii**

1 Einleitung **1**

2 Stand der Forschung **3**

3 Grundlagen **6**

3.1 Energieverbrauch durch Rechenzentren 7

3.2 Continuous Integration, Delivery und Deployment 9

3.2.1 Relevanz 10

3.2.2 Messen des Energieverbrauchs 11

3.2.3 Strategien zur Optimierung des Energieverbrauchs 13

3.2.4 Umsetzung in GitLab 20

4 Methodik **23**

4.1 Versuchsumgebung 23

4.1.1 Die Referenzpipeline 23

4.1.2 Hard- und Softwareumgebung 25

4.2 Vorstellung und Umsetzung der Szenarien 25

4.3 Datenerhebung und -auswertung 27

4.3.1 Prozess der Datenerhebung 27

4.3.2 Datenaufbereitung und Auswertungsmethoden 29

5 Ergebnisse **32**

5.1 Performance der Referenzpipeline 32

5.2 H1: Dependency-Caching 36

5.2.1 Cache-Aufbau vs. Basis 36

5.2.2 Cache-Nutzung vs. Cache-Aufbau 39

5.2.3 Cache-Nutzung vs. Basis 42
5.3 H2: Parallelisierung 45
5.4 H3: Caching und Parallelisierung 48
5.4.1 Cache-Aufbau und Parallelisierung 49
5.4.2 Cache-Nutzung und Parallelisierung 52
5.5 Gesamtbewertung der Optimierungsstrategien 57

6 Diskussion **61**
6.1 H1: Dependency-Caching 61
6.2 H2: Parallelisierung 63
6.3 H3: Kombination der Optimierungsstrategien 64
6.4 Methodenkritik und Limitationen 66

7 Fazit **69**

Literaturverzeichnis **71**

A Anhang **78**
A.1 Statistische Übersicht über die Szenarien 78

Selbstständigkeitserklärung **80**

Abbildungsverzeichnis

3.1 Optimierungsarten in CI/CD-Pipelines 14

4.1 Pseudo-Code-Darstellung des automatisierten Messablaufs 28

5.1 Ressourcenverteilung im Basis-Szenario je Job 33
5.2 Zeitlicher Vergleich des Cache-Aufbau- und Basis-Szenarios 35
5.3 Abweichung der Ressourcen in den Caching-Szenarien vom Basis-Szenario 37
5.4 Zeitlicher Vergleich der Caching-Szenarien 41
5.5 Zeitlicher Vergleich des Cache-Nutzungs- und Basis-Szenarios 44
5.6 Zeitlicher Vergleich des Parallel- und Basis-Szenarios 47
5.7 Zeitlicher Vergleich des Parallel-Aufbau- und Cache-Aufbau-Szenarios . . 51
5.8 Zeitlicher Vergleich des Parallel-Aufbau- und Parallel-Szenarios 51
5.9 Zeitlicher Vergleich des Parallel-Nutzungs- und Cache-Nutzungs-Szenarios 55
5.10 Zeitlicher Vergleich des Parallel-Nutzungs- und Parallel-Szenarios 56
5.11 Relativer Vergleich aller Szenarien mit dem Basis-Szenario 57

Tabellenverzeichnis

A.1 Statistische Kenngrößen je Szenario und Metrik 79

1 Einleitung

Der weltweite Energieverbrauch ist mit einem Anteil von 73 % die Hauptursache des menschengemachten Klimawandels [61]. Daher wurde 2015 im *Sustainable Development Goal* 7 von den Mitgliedsstaaten der Vereinten Nationen gefordert, den Ausbau erneuerbarer Energien bis zum Zieljahr 2030 deutlich voranzutreiben [61]. Mit dem Unterziel 7.3 haben sich die Staaten zu einer Verdoppelung der Energieeffizienz seit 2010 bis zum Zieljahr verschrieben, da allein hierdurch 40 % der Emissionseinsparungen erreicht werden könnten. Da die erforderliche jährliche Effizienzsteigerung jedoch mit Ausnahme 2015 konsequent verfehlt wurde, wurde sich 2023 im Rahmen der weltweiten Klimakonferenz *COP28* darauf geeinigt, die Bemühungen nochmals zu verdoppeln [39, 38].

Während die Digitalisierung oft als Ermöglicherin von Energieeffizienzsteigerungen erachtet wird, ist der Sektor der Informations- und Telekommunikationstechnologien (ICT) selbst ein bedeutender Treiber des globalen Energieverbrauchs [35]. Auf ihn entfallen mindestens 4,7 % des globalen Stromverbrauchs. Hiervon ist ein wesentlicher Anteil dem Betrieb von Rechenzentren zuzuschreiben, für dessen Stromverbrauch durch fortschreitende Entwicklungen in der KI eine Verdopplung bis 2030 prognostiziert wird [37, 2]. In den Rechenzentren sind wiederum Server für den Großteil des Energiebedarfs verantwortlich [37]. Ihr Stromverbrauch hängt neben der verwendeten Hardware stark von der Auslastung durch laufende Anwendungen ab, weshalb in der Optimierung von Software ein hohes Einsparpotenzial liegt [56, 44].

Ein zentraler Bestandteil in der Nutzung von Serverressourcen sind Praktiken, die unter dem Begriff Continuous Integration, Delivery und Deployment (CI/CD) zusammengefasst werden. CI/CD-Pipelines automatisieren Aufgaben wie das Build, Testen und Deployment von Software und werden besonders im Rahmen agiler Entwicklungspraktiken häufig ausgeführt. Darüber hinaus wird ihre Anwendung durch Plattformen wie GitLab.com und GitHub.com stark vereinfacht. Es ist daher von einem hohen Energieverbrauch durch die wiederholte Ausführung von CI/CD-Pipelines auszugehen. Wenngleich er vor den Entwickler:innen hinter der Simplizität der Umsetzung verborgen bleibt [51].

Um einen Beitrag zur Reduktion und Sichtbarmachung des Energieverbrauchs im ICT-Sektor zu leisten, ist es daher notwendig, die Energieeffizienz von CI/CD-Pipelines und die Auswirkungen von Optimierungsstrategien auf diese zu untersuchen. Aus diesem Grund werden in dieser Arbeit die Auswirkungen der beiden Best Practices *Dependency-Caching* und *Job-Parallelisierung* auf den Energieverbrauch einer beispielhaften CI/CD-Pipeline untersucht, die in ihrer grundlegenden Form in einem Projekt des Leibniz-Informationszentrums Wirtschaft (ZBW) verwendet wird. Hierzu sollen die folgenden drei Hypothesen überprüft werden:

H1 (Caching) Die Initialisierung der Dependency-Caches steigert den Energieverbrauch, die CPU-Auslastung, die RAM-Auslastung und die Laufzeit eines einzelnen, durch die Initialisierung betroffenen Pipeline-Durchlaufs. Nachfolgende Durchläufe profitieren jedoch vom Caching und weisen bezüglich der Metriken deutlich geringere Werte auf als eine Implementierung ohne Caching, wodurch die Strategie über mehrere Pipeline-Durchläufe gemittelt zu einer Netto-Reduzierung der Metriken führt.

H2 (Parallelisierung) Die Einführung einer Stage-übergreifenden Parallelisierung führt zu signifikant höheren Spitzenauslastungen von CPU und RAM während eines Pipeline-Durchlaufs, reduziert aber dennoch ihren Energieverbrauch und ihre Laufzeit.

H3 (Kombination) Die kombinierte Anwendung der beiden Best Practices stellt die effizienteste Gesamtstrategie dar, indem sie im Vergleich zu allen Einzelstrategien die Laufzeit und den Energieverbrauch gemittelt über mehrere Pipeline-Durchläufe maximal reduziert.

Zur Überprüfung der Hypothesen wird, aufbauend auf dem Stand der Forschung (Kapitel 2), in Kapitel 3 zunächst der Energieverbrauch von Rechenzentren beleuchtet. Daran anschließend wird die Softwareengineering-Praxis CI/CD samt der Möglichkeiten zur Messung und Optimierung ihres Energieverbrauchs erläutert. Im Anschluss wird in Kapitel 4 die Methodik der Untersuchung vorgestellt, welche die Versuchsumgebung, die Umsetzung der Szenarien sowie die Datenerhebung und -auswertung umfasst. Die Ergebnisse der durchgeführten Experimente werden in Kapitel 5 präsentiert und dann in Kapitel 6 diskutiert, in den wissenschaftlichen Kontext eingeordnet sowie vor dem Hintergrund der verwendeten Methodik kritisch reflektiert. Abschließend fasst Kapitel 7 die zentralen Erkenntnisse der Arbeit zusammen und ordnet ihren Beitrag zur Forschung ein.

2 Stand der Forschung

Im Folgenden wird der aktuelle Stand der Forschung im Bereich des Energieverbrauchs von CI/CD-Pipelines und der zugehörigen Optimierungsstrategien dargelegt. Zuerst werden Arbeiten vorgestellt, die sich allgemein mit dem Energieverbrauch von Pipelines befassen. Dann folgen Studien, die zusätzlich oder ausschließlich spezifische Best Practices und deren Auswirkungen untersuchen.

Perez et al. [51] messen in ihrer Untersuchung den Energieverbrauch von über 800 CI/CD-Pipelines aus knapp 400 gängigen Java-Projekten auf GitHub.com. Ihren Ergebnissen zufolge beträgt der durchschnittliche Energieverbrauch einer einzelnen Pipeline 10,2 Wh. Erscheint der Verbrauch pro Ausführung gering, so summiert er sich über die Menge der Ausführungen auf einen Mittelwert von 22 kWh je Projekt. Jedoch, so stellen die Autor:innen fest, variiert der Energieverbrauch der Pipelines stark. Den Median für den Energieverbrauch eines Pipeline-Durchlaufs beziffern sie auf unter 1 Wh, während z. B. acht der Pipelines je über 100 Wh für einen Durchlauf benötigen.

Limbrunner [45] untersucht den Energieverbrauch von CI/CD-Pipelines aus 2507 GitHub-Repositorys und unterscheidet dabei zwischen dem Gesamtverbrauch und dem Verbrauch pro Stage, Job und Step. Die durchschnittliche Leistungsaufnahme einer Pipeline liegt bei circa 31 Watt bei einer Laufzeit von rund 14 Minuten, was einem Energieverbrauch von durchschnittlich 7 Wh entspricht. Limbrunner stellt jedoch fest, dass die Laufzeiten stark variieren. Aus seiner Analyse geht weiter hervor, dass auf Job-Ebene die Release-Jobs den höchsten Energieverbrauch aufweisen, während auf Step-Ebene die Build-Steps am meisten verbrauchen. Build-Jobs selbst benötigen dabei deutlich weniger Energie als Release-Jobs, was darauf hindeuten könnte, dass viele Release-Jobs vorbereitende Build-Steps beinhalten

Bouzenia und Pradel [3] haben eine empirische Studie zur Ressourcennutzung und den Optimierungspotenzialen von CI/CD-Pipelines auf GitHub.com durchgeführt. Als Ressource definieren sie die Laufzeit eine Pipeline sowie die anfallenden Kosten. Insgesamt

haben sie 1,3 Millionen Pipeline-Durchläufe in über 900 Open-Source-Repositorys untersucht. Ihre Analyse ergibt, dass Build- und Test-Aufgaben zusammen über 90 % der Laufzeit beanspruchen, während Deployment-Aufgaben weniger als einen Prozent ausmachen.
Darüber hinaus untersuchen sie sechs Optimierungsstrategien und stellten fest, dass die Nutzung von Caches die Laufzeit der Pipelines im Schnitt um 3,4 % bei kostenlosen Repositorys und um 6,9 % bei kostenpflichtigen Repositorys verringert. Insgesamt bemerken die Autor:innen eine geringe Akzeptanz der vorhandenen Optimierungsverfahren, was sie teilweise auf mangelhafte Dokumentation und unzureichende Unterstützung bei der Implementierung zurückführen.

Gallaba et al. [20] analysieren in einer Studie das Laufzeitverhalten verschiedener Pipeline-Stages und kamen zu dem Ergebnis, dass Kompilierungs- und Testphasen im Median jeweils knapp ein Drittel der Laufzeit ausmachen, gefolgt von Dependency-Downloads mit knapp einem Fünftel. In einer weiteren Arbeit stellen Gallaba et al. [21] KOTINOS vor, einen Ansatz zur Beschleunigung von CI-Builds durch Environment Caching und das Überspringen nicht betroffener Build-Steps. In ihrer Untersuchung von insgesamt zehn Projekten konnten sie durch die Einführung von KOTINOS signifikante Laufzeitverbesserungen bei minimalem zusätzlichem Ressourcenbedarf nachweisen.

Mathew und S R [46] haben eine Mixed-Method-Studie in 50 beteiligten Unternehmen durchgeführt, um die Effekte von Parallelisierung, verschiedener Caching-Strategien und Testoptimierung auf die Performance von CI/CD-Pipelines zu bewerten. Aus dem quantitativen Teil der Studie geht hervor, dass Parallelisierung zu einer durchschnittlichen Laufzeitreduzierung von circa 40 % führte. Gleichzeitig wurde durch eine verbesserte Lastenverteilung die CPU-Auslastung um 38,7 % und durch eine effizientere Speicherzuweisung die RAM-Auslastung um 35,2 % verringert. Der Einsatz von Dependency-Caching reduzierte die Build-Zeit um 52,4 % und führte durch eine bessere Ausnutzung des Speicherplatzes zu einer Verringerung der Speichernutzung um 45,6 %. Die Kombination aller Optimierungsstrategien bewirkte mit einer Reduzierung der Pipeline-Laufzeit um 58,7 % die größte Zeitersparnis und optimierte zugleich die Ressourcennutzung.

In ihrer Arbeit stellen Nayak et al. [49] ein Verfahren zur Reduzierung von Testdurchführungen vor, indem nur Tests für von Änderungen betroffenen Code-Abschnitten ausgeführt werden. Zur Überprüfung ihres Ansatzes haben sie zunächst über ein Quartal lang den Energieverbrauch einer Pipeline bei vollständiger Ausführung der Tests gemessen und dann über denselben Zeitraum unter Anwendung ihres Verfahrens. Die wesentli-

che Erkenntnis ist, dass durch die reduzierte Testausführung der Energieverbrauch pro Quartal um fast 90 % gesenkt werden konnte.

Die vorliegende Arbeit knüpft an die obigen Studien an, indem sie die Effekte ausgewählter Optimierungsstrategien auf den Ressourcenverbrauch von CI/CD-Pipelines untersucht. In Kontrast untersucht sie die Auswirkung der Best Practices jedoch nicht unter Produktivbedingungen und in der bisherigen Laufzeitumgebung der Pipeline. Stattdessen werden die Pipeline-Durchläufe gezielt für das Experiment und in einer dezidierten Versuchsumgebung ausgeführt.

3 Grundlagen

Die wissenschaftliche Auseinandersetzung mit dem globalen Energieverbrauch der Informations- und Telekommunikationstechnologien (ICT) ist durch eine uneinheitliche Datenlage gekennzeichnet, da je nach Studie unterschiedliche Aspekte betrachtet werden und Unternehmen relevante Informationen nicht weitergeben, sodass Berechnungen auf verschiedenen Annahmen beruhen und teilweise Aktualität einbüßen [23, 2, 33]. Dennoch ist allen Studien gemein, dass sie einen Anstieg des Stromverbrauchs für Rechenzentren und Datennetze prognostizieren [36].

Allein auf den ICT-Sektor entfallen einem Bericht der Internationalen Fernmeldeunion (ITU) und Weltbank [2] zufolge mindestens 1,7 % (2022) der globalen Treibhausgasemissionen und 4,7 % (2021) des globalen Stromverbrauchs. Gelenbe [23] beziffert hingegen den Anteil des Stromverbrauchs des ICT-Sektors auf Basis von Angaben der Internationen Energieagentur (IEA) für das Jahr 2019 auf 8,5 % – von einem Rückgang ist jedoch nicht auszugehen.
Ca. 16,4 % (2022) des Stromverbrauchs des ICT-Sektors lassen sich hierbei auf Rechenzentren zurückführen, näherungsweise 0,7 % des gesamten globalen Stromverbrauchs 2022 [2]. Für das Jahr 2024 hat die IEA [37] einen Anteil der Rechenzentren am globalen Stromverbrauch von 1,5 % ermittelt. Sie geht von einem jährlichen Anstieg um +15 % bis 2030 auf 3 % des globalen Stromverbrauchs aus. Dies entspräche einer Verdoppelung und einem viermal so schnellen Anstieg im Vergleich zu allen anderen Sektoren [37]. Trotzdem ein Anteil am globalen Stromverbrauch von 3 % noch vergleichsweise gering wäre, unterstreicht diese Zunahme die Rolle, die Rechenzentren bereits jetzt und zukünftig einnehmen. Insbesondere in den USA haben Rechenzentren Stand 2024 einen Anteil von 45 % am nationalen Stromverbrauch [37].

ICT-Infrastrukturen sind jedoch nicht nur Treiberin des weltweiten Energieverbrauchs, sondern indem sie die Effizienz von Prozessen steigern, können sie den Verbrauch potenziell senken [35, 6]. Grob lässt sich unterscheiden zwischen den Handlungsfeldern *Green in ICT* und *Green by ICT* [35]. Da diese Arbeit jedoch die Perspektive *Green in ICT*

einnimmt und spezieller den Energieverbrauch durch *Continuos Integration, Delivery und Deployment* (CI/CD) untersucht, wird im Folgenden zunächst der Energieverbrauch durch Rechenzentren näher beleuchtet und anschließend die Softwareengineering-Praxis CI/CD erläutert mit einem Fokus auf ihre Relevanz bezüglich des Energieverbrauchs.

3.1 Energieverbrauch durch Rechenzentren

Als Rechenzentrum bezeichnet das Bundesamt für Sicherheit in der Informationstechnik [4] zentrale „IT-Betriebsbereiche“ und die notwendigen technischen Supportbereiche. IT-Betriebsbereiche definiert es als Räume, „[...] in denen die Hardware aufgebaut ist und betrieben wird, die der Bereitstellung von Diensten und Daten dient“ [4]. Die IT-Hardware umfasst die Server, Datenträger und Netzwerkgeräte wie Switches und Router [56]. Zu den technischen Supportbereichen eines Rechenzentrums zählen u. a. die Stromversorgung, Kühlung, Löschtechnik sowie Sicherheitstechnik [4]. Die weitere Klassifikation nach Art der Nutzung sowie Anzahl und Art der Nutzenden ist hier nicht relevant. Der Definition folgend können bereits kleinere Serverschränke als Rechenzentrum klassifiziert werden. Für den Zeitraum von 2010 bis 2019 in Deutschland gehen Grünwald und Caviezel [33] konstant von ungefähr 125 Tausend Rechenzentren in der Größenordnung von bis zu 10 m^2 aus. Eine Zunahme ist hingegen bzgl. größerer Rechenzentren zu verzeichnen, sowohl in Deutschland als auch global [33, 37].
Eine weitere Klassifikation von Rechenzentren unterteilt diese in Unternehmens-, Colocation- und Dienstleistungs- sowie Hochleistungsrechenzentren. Unternehmensrechenzentren werden üblicherweise von Unternehmen zu eigenen Zwecken verwendet und sind meist kleiner sowie weniger energieeffizient als andere Rechenzentrumsarten. Colocation-Rechenzentren vermieten nur die Fläche, auf der verschiedene Kund:innen ihre eigenen Server und Datenträger unterbringen können, wohingegen Dienstleistungsrechenzentren zusätzlich zur Fläche die gesamte IT-Hardware anbieten. Hochleistungsrechenzentren werden von großen Technologiekonzernen betrieben und beinhalten höchst effiziente IT-Hardware für Webhosting sowie Cloud- und KI-Dienstleistungen [37].

Unterschiedliche Faktoren führen zum weiteren Anwachsen der Rechenzentren sowohl in ihrer Anzahl als auch in ihrer Serverkapazität und in der Folge zu einem Anstieg ihres Stromverbrauchs. Einige dieser Faktoren sind die weltweite Digitalisierung, die zunehmende Verwendung von Cloud-Computing, die Zunahme des Streamings von Medien,

die Ausweitung der Nutzung sozialer Medien und zuletzt der Vormarsch künstlicher Intelligenz [37, 2]. Dieser ist eng an die Fortschritte bei der Weiterentwicklung von GPUs und dem damit einhergehenden Kostenrückgang für GPUs gekoppelt [37]. In der Folge werden zunehmend Hochleistungsserver mit GPUs verwendet, deren Stromverbrauch höher ist als der konventioneller Server [37]. Auf künstliche Intelligenz spezialisierte Hochleistungsrechenzentren verbrauchen daher bis zu fünfmal so viel Strom wie konventionelle Rechenzentren [37]. Zudem sind aktuell Rechenzentren geplant mit einem 50 Mal so hohen Verbrauch [37].

Drei Viertel des Stromverbrauchs von Hochleistungsrechenzentren ist auf den Betrieb ihrer Server zurückzuführen (Stand 2024) [37]. Bei anderen Rechenzentrenarten liegt dieser Wert zwischen ca. 45 % bis 55 % [37]. Positiv geframet, sind Hochleistungsrechenzentren damit bezogen auf ihre Aufgaben effizienter [37]. Da sie jedoch insgesamt mehr Strom verbrauchen als andere Rechenzentrenarten (s. o.), ist eher davon auszugehen, dass der Verbrauch der anderen Stromabnehmer (Datenspeicher, Netzwerkgeräte, Kühlanlage etc.) nicht wesentlich angestiegen ist und diese somit einen geringeren Anteil am Gesamtverbrauch ausmachen. Die Optimierung von Servern und ihrer Nutzung birgt folglich enorme energetische Einsparpotenziale [18, 15]. Rankings wie *Green500* zeigen die Bemühungen, die diesbezüglich auf Hardwareebene geleistet werden [60]. Allein zwischen November 2021 und November 2024 ist die Anzahl der maximal gemessenen Gigaflops pro Watt um mehr als 80 % angestiegen [60, 59].

Jedoch hängt der Stromverbrauch von Servern auch signifikant von der Auslastung durch laufende Anwendungen ab – eine geringere Auslastung korreliert mit geringerem Stromverbrauch –, sodass verschiedene Autor:innen davon ausgehen, dass in der Optimierungen von Software ein höheres Einsparpotenzial liegt als in der Optimierung der Hardware [56, 15, 18]. Claßen et al. [7] zufolge ist ein signifikanter Anteil der Prozesse von Servern auf Programme zurückzuführen, die automatisiertes Testen, Builds und Deployment von Software ermöglichen.
Im Folgenden soll auf diese automatisierten Praktiken, die oft unter dem Term CI/CD zusammengefasst werden, eingegangen werden, da sie ein zentraler Bestandteil in der Nutzung von Rechenzentrumsressourcen sind.

3.2 Continuous Integration, Delivery und Deployment

Über vier Fünftel der Softwareentwickler:innen ist auf die ein oder andere Art und Weise in DevOps-Tätigkeiten involviert [11]. Während es viele verschiedene Definitionen von DevOps gibt, soll DevOps hier als inhaltliche und personelle Kombination von Softwareentwicklung und Systemadministration verstanden werden. Hierbei umfasst DevOps verschiedene Methoden und Tools, die bei Tätigkeiten an der Schnittstelle beider Bereiche unterstützen [14]. CI/CD reiht sich unter den obersten fünf der bekanntesten DevOps-Tätigkeiten ein [11]. Während die Abkürzung CI/CD bereits früher in dieser Arbeit eingeführt wurde, soll im Folgenden auf die zwei Bestandteile eingegangen werden, aus denen sich das Tätigkeitsfeld zusammensetzt.

Continuous Integration (CI) beschreibt die regelmäßige Integration lokaler Entwicklungsstände in einen globalen Entwicklungsstand mit dem Ziel, immer einen funktionsfähigen und aktuellen globalen Stand der Entwicklung zu haben [42, 53]. Typischerweise wird für die Verwaltung der Stände eine Versionsverwaltung w. z. B. Git verwendet [57]. Der Integrationsprozess umfasst dabei vor allem – aber nicht ausschließlich – das Erstellen von Builds, das Ausführen statischer und dynamischer Codeanalysen sowie Tests [19, 53]. Ziel dieser Prozesskette ist es, den verantwortlichen Entwickler:innen ein unmittelbares Feedback über die Funktionalität des zur Verfügung gestellten Codes zu geben und dabei gleichzeitig den globalen Stand möglichst fehlerfrei zu halten [19, 42]. Damit Fehler zeitnah behoben werden können, wird oft eine tägliche bis mehrmals tägliche Integration angestrebt [19, 57].

Continuous Delivery (CD) ist das automatisierte Deployment einer lauffähigen Softwareversion in eine beliebige Umgebung, jedoch angestoßen durch einen Menschen [62, 19]. CD ist somit der CI nachgelagert, da es die grundsätzliche Funktionsfähigkeit der Software voraussetzt, welche durch CI sichergestellt wird [57]. Continuous Deployment (CDP) setzt weiterführend auf CD auf, indem es das gänzlich automatisierte Deployment einer Softwareversion in eine Produktivumgebung beschreibt, sobald die CI erfolgreich durchlaufen wurde [53, 62]. CD kann in Abgrenzung zu CDP auch als Pull-basierter Ansatz bezeichnet werden, bei dem eine Person entscheidet, welche Version einer Software ausgerollt wird [57]. CDP ist dementsprechend Push-basiert, da das Auslösen und erfolgreiche Durchlaufen der CI-Prozesse automatisch zu einem Deployment der integrierten, neuen Version führt [57]. Im Rahmen dieser Arbeit wird, sofern nicht explizit anders angemerkt, mit der Abkürzung CI/CD sowohl Continuous Delivery als auch Continuous Deployment bezeichnet, da die genaue Differenzierung hier nicht weiter zielführend ist.

Nachdem das Konzept von CI/CD in den Grundzügen erläutert wurde, wird im Folgenden die aktuelle und zukünftige Relevanz von CI/CD mit einem Fokus auf den Energieverbrauch aufgezeigt. In den daran anschließenden zwei Unterkapiteln wird skizziert, wie sich der Energieverbrauch von CI/CD-Pipelines messen lässt und welche Ansätze es gibt, ihren Energie- und Ressourcenverbrauch zu reduzieren. Abschließend wird die technische Umsetzung von CI/CD-Pipelines in GitLab vorgestellt.

3.2.1 Relevanz

In einer im Auftrag der Continuous Delivery Foundation [41] durchgeführten Umfrage aus dem Jahr 2022 gaben fast die Hälfte der befragten Entwickler:innen an, entweder CI oder CD einzusetzen. Die zunehmende Verwendung von CI/CD-Praktiken hängt stark zusammen mit der Beliebtheit agiler Vorgehensmodelle in der Softwareentwicklung, da sie die Notwendigkeit mit sich bringen, neue Softwareversionen in kürzeren Zeitabschnitten zur Verfügung zu stellen [22, 57]. CI/CD-Praktiken versprechen diesbezüglich, dass durch häufigere Releases Feedback von Entwickler:innen und User:innen schneller eingeholt und eingearbeitet werden kann [57]. Dies soll darüber hinaus zu einem höheren Produktvertrauen seitens der User:innen führen [57].

GitLab und GitHub zählen zu den meist verwendeten Plattformen, die Versionsverwaltung und DevOps-Lösungen kombiniert anbieten [29]. Nach eigenen Angaben hat GitLab.com 04/2025 über 50 Millionen [28] und GitHub.com über 150 Millionen User:innen [25]. Durch diverse Angebote auf den Plattformen wird das Erstellen und Ausführen von CI/CD-Pipelines immer mehr Entwickler:innen ermöglicht und zudem durch frei verfügbare Vorlagen für Standardprozesse stark vereinfacht [51]. Mehr als 30 % der Open-Source-Projekte auf GitHub.com verwenden *GitHub Actions*, GitHubs Tool zur Erstellung von CI/CD-Pipelines [3]. Ein treibender Grund hierfür ist sicherlich auch das kostenlose Anbieten von Rechenleistung auf geteilten, von GitHub gehosteten Servern bis zu einer jährlichen Gesamtlaufzeit von knapp über einer halben Stunde [3]. Darüber hinaus haben Golzadeh et al. [29] in einer Langzeitstudie zur Verwendung von CI/CD in GitHub-Repositorys über einen Zeitraum von neun Jahren festgestellt, dass in den Repository in der Regel mehr als nur eine CI/CD-Pipeline implementiert ist. Ebenso zeigen ihre Daten, dass die Anzahl der CI/CD-Pipelines in gleichem Maße wie die Anzahl der Repositorys stark zunimmt.

Die Menge der Pipeline-Ausführungen hängt von der Menge der Aktionen (z. B. Commits, Pull-Requests, aber auch Pipeline-Ausführungen selbst) ab, die in einem Repository ausgeführt werden, da Pipelines durch ebensolche Aktionen ausgelöst werden können [51]. Bouzenia und Pradel [3] identifizieren Commits und Pull-Requests als die zwei häufigsten Auslöser für CI/CD-Pipelines. Eine weitere Studie von Fairbanks et al. [16] basierend auf ca. zwölftausend öffentlich verfügbaren Repositorys auf GitHub.com und GitLab.com zeigt, dass die Verwendung von CI/CD-Pipelines mit einer höheren Commit-Häufigkeit einhergeht. Andersherum, der Erkenntnis von Bouzenia und Pradel [3] folgend, müsste dies zu einer häufigeren Ausführung der CI/CD-Pipelines führen.

Im Anschluss an die Studienergebnisse, die im Kapitel zum Forschungsstand skizziert wurden und mit Hinblick auf die hohe Anzahl an Nutzer:innen von GitLab und GitHub lässt sich ein hoher Gesamtenergieverbrauch durch das häufige Ausführen von CI/CD-Pipelines erahnen. Hinzu kommt, dass agile Entwicklungspraktiken durch kürzere Entwicklungszyklen zu gesteigerter Aktivität in Repositorys führen und in der Folge zu häufigeren Pipeline-Ausführungen [51]. Der Energieverbrauch von CI/CD-Pipelines wird jedoch vor den Entwickler:innen hinter der Simplizität der Umsetzung verborgen, so Perez et al. [51]. Umso wichtiger ist es, Methoden und Möglichkeiten herauszuarbeiten, um den Energieverbrauch von CI/CD-Pipelines zu messen und dadurch sichtbar zu machen.

3.2.2 Messen des Energieverbrauchs

CPU, Arbeitsspeicher, Speichermedien und Netzwerkadapter sind gemeinsam für den Großteil des Energieverbrauchs von Computersystemen verantwortlich [44]. CPUs sind hierbei die stärksten Verbraucher, die zwischen 60 % und 69 % des Energieverbrauchs zu verantworten haben [51]. Durch die Verwendung von Arbeitsspeicher-Ressourcen entstehen immerhin noch 6 % bis 19 % [51]. Hinzu kommt, dass der Zugriff auf den Arbeitsspeicher in der stark verbreiteten Von-Naumann-Architektur einen Performance- und energietechnischen Flaschenhals darstellt, der entsprechend als Von-Naumann-Flaschenhals bekannt ist [48]. In der Vergangenheit lag der Fokus von Forschung oft darauf, wie etwaige Performance-Probleme durch Hardware-seitige Optimierungen gelöst werden können [44, 24]. Mittlerweile gibt es jedoch ausreichend Nachweise, dass die auf der Hardware laufende Software relevanten Einfluss auf den Energieverbrauch hat [44]. Der Energieverbrauch von Software zieht sich über den gesamten Softwareentwicklungszyklus hinweg [44]. In der hier fokussierten Nutzung wird vor allem Energie für Berechnungen (CPU/GPU),

Lesen und Schreiben von Daten (Arbeitsspeicher, Speichermedien) sowie für Netzwerkkommunikation (Netzwerkadapter) benötigt [24].

Grundsätzlich kann zwischen hardware- und softwarebasierten Ansätzen zur Messung des Energieverbrauchs von Computersystemen unterschieden werden [44, 5]. Hardwarebasierte Techniken messen Energie durch die direkte Erfassung elektrischer Parameter wie Strom und Spannung [5]. Es kann weiter differenziert werden zwischen externen und integrierten Messungen [5, 9]. Externe Messungen werden durch Leistungsmesser gewährleistet, die zwischen das Host-System und dessen Stromversorgung geschaltet werden [9]. Diese Messart auf Systemebene ist messtechnisch zwar sehr genau, allerdings lassen sich durch sie keine Aussagen über den Energieverbrauch einzelner virtueller Maschinen (VM) auf dem Host-System treffen [9, 45]. In vielen Kontexten scheitert eine solche Messart zudem bereits am mangelnden Zugriff auf die Stromversorgung des Host-Systems.
Integrierte Messungen basieren in der Regel auf herstellerseitig verbauten Sensoren in u. a. CPUs und liefern darüber Informationen über den Energieverbrauch des Systems [9]. Das Auslesen der Daten ist z. B. über Hardwaretreiber möglich [9]. Intel-Prozessoren (Sandy Bridge oder neuer) verfügen über das integrierte Messtool *Running Average Power Limit* (RAPL), dessen Energieverbrauchsmessungen für die CPU inkl. integrierter Grafikarten und für den Arbeitsspeicher auf internen Leistungszählern und einem proprietären Leistungsmodell basieren [43]. Khan et al. [43] haben die Messgenauigkeit von RAPL in ihrer Studie untersucht und konstatieren, dass insbesondere seit Intels Prozessorarchitektur *Haswell* die Messungen vergleichbar genau sind mit denen externer Messungen. Eine Möglichkeit die zur Verfügung gestellten Daten unter Linux auszuwerten, ist es die Dateien im Verzeichnis `/sys/class/powercap/intel-rapl/` auszulesen [45]. Bzgl. der Verwendung in VMs und Containern bemängelt Limbrunner [45] jedoch die Komplexität der Anwendbarkeit. Positiv ist hervorzuheben, dass durch die Hardwarenähe von RAPL letztlich nur durch das Auslesen selbst zusätzlicher Overhead verursacht wird [45, 32].

Softwarebasierte Messtools können den Energieverbrauch des Gesamtsystems, einzelner Systemkomponenten (CPU, Arbeitsspeicher etc.) oder einzelner Prozesse ermitteln. Viele der verfügbaren Tools ermitteln den Energieverbrauch, indem sie jeweils den Verbrauch des Arbeitsspeichers und der CPU modellieren und darüber den Gesamtverbrauch berechnen [44].
Eine andere Option ist es, Machine-Learning-Modelle zu verwenden [44]. Diese benötigen Werte über die Ressourcennutzung eines Systems oder Prozesses sowie relevante Informationen über die Architektur des Systems als Eingabe und ermitteln auf dieser

Basis Energiewerte [44]. Die Genauigkeit der Modelle hängt stark von der Qualität der Trainingsdaten und der Korrektheit sowie Verfügbarkeit der Eingabewerte ab [44]. Als Trainingsdaten kommen z. B. Benchmarks von Servern zum Einsatz, die deren Stromverbrauch bei Vollast und Leerlauf sowie deren Architektur-Spezifikation enthalten [54]. Für das Zusammentragen der einzugebenden Ressourcenwerte kann zwischen verschiedenen Tools ausgewählt werden [32]. Eine Vielzahl der Tools bezieht die Ressourceninformationen von Zählern auf Kernel-Ebene [32]. Der Zugriff auf diese Zähler erfolgt in Linux-Distributionen über das Dateisystem, z. B. über das `/proc`- (prozessweit) oder das `/sys`-Dateisystem (prozess- und systemweit) [52, 32]. Ähnlich wie bei RAPL verbraucht lediglich der Lesezugriff auf die Zähler zusätzliche, vernachlässigbare Ressourcen [32]. Unter Vorbehalt der Qualitätseinschränkungen durch mangelhafte Eingabewerte birgt diese Vorgehensweise zum Messen des Energieverbrauchs von Software den Vorteil, dass sie sich umstandslos in VMs und Container-Technologien wie Docker integrieren lässt [34].

Eine nicht zu vernachlässigende Kennzahl für die (relative) Energieeffizienz eines Prozesses oder einer Software ist zudem deren Laufzeit: „[...] there is a strong correlation between time and energy consumption for a given platform running a single computation thread“ [13]. Dies ist einerseits bedingt durch die geringere Anzahl an Maschinenbefehlen, die in der kürzeren Laufzeit ausgeführt werden können und andererseits kann bei kürzerer Laufzeit der Prozessor früher in einen energiesparenden Leerlaufmodus übergehen [13].

In diesem Unterkapitel wurde der Fokus auf die größten Energieverbraucher von Computersystemen gelenkt und einen Überblick über mögliche Messmethoden gegeben. Das nächste Unterkapitel wird daran anknüpfen, indem es verschiedene Strategien vorstellt, um die Energie- und Ressourcennutzung von CI/CD-Pipelines zu optimieren.

3.2.3 Strategien zur Optimierung des Energieverbrauchs

Der Energieverbrauch von Software lässt sich reduzieren, indem die Ressourcennutzung minimiert wird: Weniger (kostenintensive) Berechnungen durchführen, Zugriffe auf den Arbeitsspeicher und Speicher reduzieren sowie Netzwerkkommunikation vermeiden [24]. Um die Wirkungsweise solcher Optimierungen zu verstehen, soll zunächst die Unterscheidung zwischen CPU- und I/O-gebundenen Prozessen eingeführt werden, um dann nachfolgend verschiedene Optimierungsstrategien im Kontext von CI/CD vorzustellen.

Ein Prozess gilt als CPU-gebunden, wenn seine Ausführungsgeschwindigkeit primär durch die Rechenleistung des Prozessors limitiert ist [32]. Demgegenüber steht ein I/O-gebun-

dener Prozess, welcher die meiste Zeit darauf wartet, dass I/O-Operationen, wie beispielsweise das Lesen von einer Festplatte oder das Laden von Daten aus dem Netzwerk, abgeschlossen werden [32]. Die Zeit, die ein Prozessor auf I/O-Operation wartet (*IOWait*), ist eine relevante Quelle des Energieverbrauchs, da die CPU nicht in ihren stark energiesparenden Leerlauf-Modus (*idle state*) übergehen kann und somit der Anteil unproduktiv verbrauchter Energie durch die CPU steigt [17, 8]. Zudem benötigen die I/O-Aktivitäten, die in dieser Phase ablaufen, ebenfalls Energie, die nicht zu vernachlässigen ist [17, 8]. Zwar wird die Leistungsaufnahme der CPU während solcher *IOWait*-Phasen in modernen CPU-Architekturen reguliert, sodass weniger Energie durch die CPU verbraucht wird als in CPU-gebundenen Prozessen, allerdings liegen die so erzielten Leistungswerte weiterhin über den Werten im Leerlauf [12]. Entsprechend zeigen Dorier et al. [12], dass Systeme mit vielen auf I/O-Operationen wartenden Prozessen eine niedrigere durchschnittliche Leistungsaufnahme aufweisen, jedoch durch die verlängerte Laufzeit in Summe mehr Energie verbrauchen.

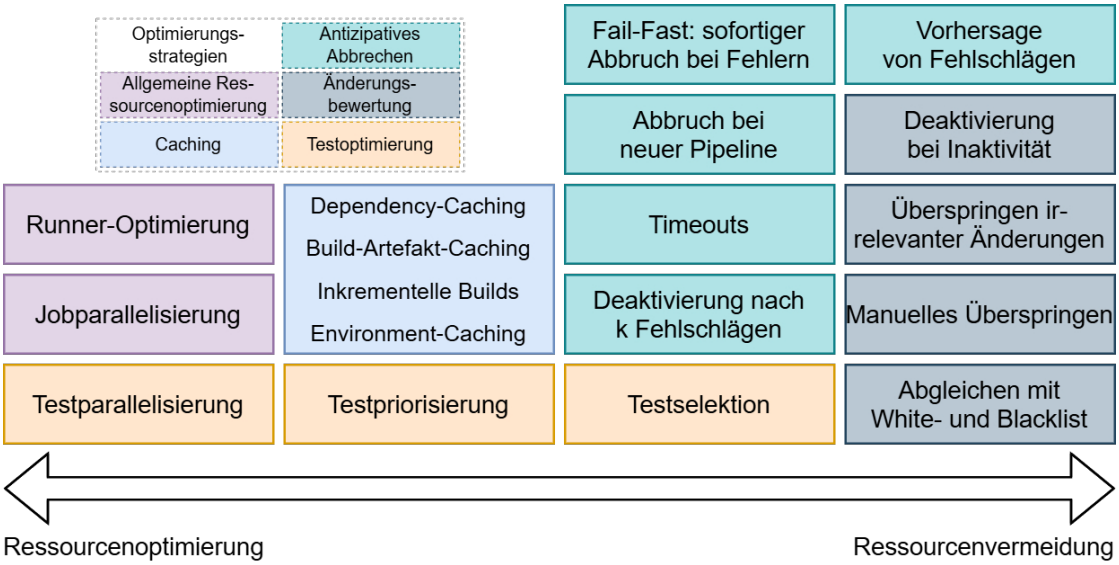

Abbildung 3.1: Überblick über Optimierungsarten in CI/CD-Pipelines geordnet nach Grad der Ressourcenoptimierung bzw. -vermeidung.

Auf der Grundlage verschiedener Autor:innen [3, 20, 21, 45, 46, 47, 55, 58] wurden Best Practices zur Optimierung der Ressourcennutzung und damit einhergehend des Energieverbrauchs von CI/CD-Pipelines identifiziert in fünf übergeordnete Strategien unterteilt: *Caching*, *Änderungsbewertung*, *Testoptimierung*, *antizipatives Abbrechen* und *allgemeine Ressourcenoptimierung* (*Abb. 3.1*). Die Praktiken lassen sich danach ordnen, ob ihre

Optimierung darin besteht die Nutzung von Ressourcen zu vermeiden, indem Pipelines beispielsweise gar nicht erst ausgeführt werden, oder ob sie die Nutzung der verfügbaren Ressourcen optimieren. Es sei darauf hingewiesen, dass diese Anordnung primär als Orientierungshilfe dient und sich nicht jede Praktik umstandslos auf der Skala einordnen lässt. Nachfolgend wird auf die übergeordneten Strategien und, wo angebracht, beispielhaft auf die einzelnen Praktiken eingegangen.

Caching

Caching-Mechanismen können in CI/CD-Pipelines allgemein verwendet werden, um Dateien für nachfolgende Pipeline-Durchläufe zur Verfügung zu stellen. Sofern sich die Anforderungen an die Dateien (z. B. geänderte Versionsnummer) nicht verändert haben, müssen diese nicht erneut erstellt oder heruntergeladen werden, wodurch die I/O- sowie die Netzwerklast reduziert wird [3]. Auf die enorme Effektivität von Caching bei der Einsparung von Ressourcen und Reduzierung der Laufzeit wurde im Kapitel zum Stand der Forschung (*Kap. 2*) unter anderem mit Verweis auf die Arbeiten von [3] und [21] hingewiesen.

Caching als Optimierungsstrategie für CI/CD-Pipelines wird in den hier untersuchten Studien primär in Bezug auf Dependencies, Build-Komponenten und Laufzeitumgebungen angewandt [21, 46]. Dependencies oder Laufzeitumgebungen zu cachen, lohnt sich Gallaba et al. [21] zufolge besonders, da sich Dependencies zwischen Builds nur gelegentlich verändern und ebenso Laufzeitumgebungen eher selten rekonfiguriert werden. Laufzeitumgebungen können z. B. in Form von Docker-Images gecacht werden [21]. Dependencies sind u. a. Softwarebibliotheken, die für das Build oder Testen einer Softwware benötigt und heruntergeladen werden.
Das Caching von Build-Komponenten verhindert vollständige Rebuilds, indem nur veränderte Komponenten erneut gebaut werden, wodurch die Pipeline-Laufzeit reduziert und Ressourcen effizienter genutzt werden [46]. In der Literatur und Praxis wird diese Vorgehensweise häufig als *incremental build* [46, 47] oder *build caching* [45] referenziert. Am Beispiel eines Repositorys des Mozilla-Firefox-Browsers haben Maudoux und Mens [47] gezeigt, dass es in knapp 16 % der analysierten Pipeline-Durchläufe zu keiner Änderung der Build-Dateien kam, sodass in diesen Fällen durch Build-Caching gegenüber einem Rebuild extreme Einsparung von Ressourcen möglich waren. In ihrer beispielhaften Untersuchung konnten so durchschnittlich 90 % CPU-Zeit eingespart werden.

Bei der Anwendung von Caching-Strategien ist sicherzustellen, dass bei Veränderungen der Anforderungen an die gecachten Dateien nicht mehr auf die gecachte Version zurückgegriffen wird, sondern die neuen Dateien erstellt oder heruntergeladen werden und der Cache entsprechend erneuert wird.

Alle Caching-Praktiken wurden hier primär der Ressourcenoptimierung zugeordnet, da sie darauf abzielen, vorhandene Artefakte wiederzuverwenden und somit die Effizienz laufender Pipelines steigern, anstatt eine Ausführung gänzlich zu verhindern (*Abb. 3.1*).

Änderungsbewertung

In der Kategorie *Änderungsbewertung* werden manuelle und automatisierte Optimierungspraktiken zusammengefasst, die bei fehlenden oder als irrelevant eingestuften Änderungen die Ausführung einer Pipeline verhindern. Offensichtlich birgt das Nicht-Ausführen von Pipelines im Vergleich zur Ausführung ein maximales Einsparpotenzial [51].

Die einfachste, manuelle Umsetzung besteht darin, dass Entwickler:innen vor dem Pushen eines Commits selbst entscheiden, ob die geplante Änderung den Durchlauf einer Pipeline erfordert, da z. B. Quellcode geändert wurde und dieser getestet werden muss und ein Rebuild erfordert. Soll der Commit explizit keine Pipeline auslösen, kann durch Hinzufügen von Schlüsselwörtern w. z. B. *ci-skip* in den meisten Versionsverwaltungssystemen die Ausführung unterbunden werden. Trotz des maximalen Einsparpotenzials, wird die Technik nicht häufig angewandt, was Bouzenia und Pradel auf eine mangelnde Bekanntheit des Features zurückführen [3].

Eine teilautomatisierte Form der manuellen Vorgehensweise beruht auf dem Hinterlegen von Dateien in White- oder Blacklists. Vor der Ausführung einer Pipeline wird abgeglichen, inwiefern für die geänderten Dateien Einträge in einer der Listen existieren. Grundsätzlich können zwei Fälle unterschieden werden: Sofern alle geänderten Dateien in einer Whitelist enthalten sind, wird keine Pipeline ausgelöst. Sollte jedoch eine Datei nicht in der Whitelist stehen oder aber auf einer Blacklist stehen, wird die Pipeline ausgelöst. Die Problematik dieser Praktik liegt in ihrer Absolutheit, da es selten vorkommt, dass alle Dateien eines Commits einheitlich in Whitelists enthalten sind, sodass nur selten Pipeline-Ausführungen übersprungen werden [3].

Abdalkareem et al. [1] haben 58 Java-Projekte dahingehend analysiert, welche Charakteristika Commits aufweisen, für die Entwickler:innen manuell die Ausführung einer Pi-

peline unterbunden haben. Auf Basis ihrer Erkenntnisse haben die Autoren ein Modell entwickelt, das regelbasiert entscheidet, ob ein Commit eine Pipeline auslöst oder nicht. Ihre Analyse zeigt, dass nahezu ein Fünftel der Commits zum Überspringen einer Pipeline führte. Dieses Ergebnis wird im ähnlichen Maße durch eine Studie von Jin und Servant [40] belegt. Abdalkareem et. al [1] merken in Hinblick auf ihre Automatisierung jedoch an, dass die erzielte Güte stark von der in den Commits enthaltenen Dateiarten abhängt. Commits, die ausschließlich Nicht-Quellcode-Dateien enthalten, wurden demzufolge häufiger korrekt als überspringenswürdig erkannt als Commits mit einer gemischten Art von Dateien. Dementsprechend muss je Projekt entschieden werden, ob ein derartiges, regelbasiertes Modell gewünschte Effekte erzeugt oder ob falsche Klassifizierungen von Commits einen zu hohen Schaden anrichten könnten [21].

Eine weitere Optimierungspraktik bezieht sich auf geplante Pipeline-Ausführungen, die beispielsweise zu einer bestimmten Zeit oder durch regelmäßige Ereignisse ausgelöst werden. Die Praktik sieht vor, geplante Pipeline-Ausführungen auszusetzen, sobald in einer Codebasis über einen definierten Zeitraum keinerlei Änderungen stattgefunden haben. Den bei GitHub voreingestellten Zeitraum von sechs Monaten halten Bouzenia und Pradel [3] zu lang für die meisten Projekte.

Die Strategien der Änderungsbewertung wurden einheitlich der Ressourcenvermeidung zugeordnet, da sie die Ausführung von Pipelines gänzlich unterbinden, wenn Änderungen als nicht relevant eingestuft werden oder keine Änderungen zu verzeichnen sind (*Abb. 3.1*).

Testoptimierung

Mathew und S R [46] zeigen in ihrer Studie, dass die Laufzeit von CI/CD-Pipelines durch Testoptimierungsstrategien wie Testselektion, -priorisierung und -parallelisierung signifikant verringert und die Ressourcennutzung um ca. ein Drittel optimiert werden kann. Nachfolgend werden die beiden Praktiken Testselektion und -priorisierung in aller Kürze skizziert.

Bei der Testselektion geht es darum, nur Tests auszuführen, die von veränderten Codeteilen betroffen sind. S R und Mathew arbeiten in einer weiteren gemeinsamen Veröffentlichung heraus, dass hierdurch eine Laufzeitreduktion um 30 % erreicht werden kann. Trotz der Nicht-Ausführung von Tests stellen sie weiterhin eine hohe Testabdeckung fest [55].

Die Testpriorisierung ermöglicht es, Tests, die kritische Funktionen eines Systems abprüfen, als Erstes ausführen zu lassen, sodass nicht unnötig Ressourcen für weniger wichtige Tests aufgewandt werden [45, 46].
Jin und Servant [40] nennen zudem die Möglichkeit, Tests dahingehend zu priorisieren, ob sie mit einer höheren Wahrscheinlichkeit fehlschlagen oder nicht. Indem Tests, die wahrscheinlich scheitern, da sie z. B. in den vorherigen Pipeline-Durchläufen gescheitert sind, priorisiert ausgeführt werden, kann ggf. die Ausführung voraussichtlich erfolgreich ablaufender Tests verhindert werden. Eine gesteigerte Variante dieser Priorisierung sieht vor, die wahrscheinlich erfolgreich ablaufenden Tests, gänzlich zu überspringen. Wichtig ist hier die Genauigkeit des zugrundeliegenden Modells und ab welcher Wahrscheinlichkeit Tests als erfolgreich und somit nicht-auszuführen klassifiziert werden [40].

Die Testoptimierungsstrategien teilen sich zwischen Ressourcenoptimierung und -vermeidung auf. Während die Testparallelisierung die vorhandenen Ressourcen effizienter nutzt, bewegen sich Testpriorisierung und -selektion eher in Richtung Ressourcenvermeidung, indem sie ein frühzeitiges Abbrechen der Pipeline provozieren oder die Menge der auszuführenden Tests reduzieren (*Abb. 3.1*).

Antizipatives Abbrechen

Neben der Testpriorisierung gibt es weitere Praktiken, die, sofern es zu einem Fehlschlagen der Pipeline kommt, den Abbruch der Pipeline möglichst früh forcieren, sodass möglichst keine Ressourcen durch die erfolgreichen Bestandteile der Pipeline beansprucht werden.

So gibt es z. B. die Möglichkeit bestimmte Jobs einer Pipeline priorisiert auszuführen, wenn sie in einem vorherigen Durchlauf fehlgeschlagen sind. Auf Basis ihrer Daten haben Bouzenia und Pradel festgestellt, dass ca. 30 % der Jobs, die bereits zuvor fehlgeschlagen sind, in nachfolgenden Durchläufen erneut fehlschlagen [3].

In GitHub führt standardmäßig das Fehlschlagen eines einzelnen Jobs zum Abbruch der gesamten Pipeline. Dieser Mechanismus wird in der Literatur meist als *fail fast* referenziert. Bouzenia und Pradel konnten durch diesen Mechanismus die Laufzeit der getesteten Pipelines um ca. 1,5 % bis 2 % verkürzen[1]. Aufgrund der Niedrigschwelligkeit in der Um-

setzung und da es eine Standardeinstellung ist, empfehlen die Autoren diese Strategie in GitHub-Repositorys umzusetzen [3].

Insbesondere in hochfrequenten Repositorys mit mehreren Entwickler:innen könnte Bouzenia und Pradel zufolge sinnvoll sein, laufende Pipelines zu Gunsten von neueren Pipelines in der Warteschlange abzubrechen. In ihrer Analyse konnte die Gesamtlaufzeit so um knapp 4 % verkürzt werden. Wichtig hierbei sei allerdings, nur Pipelines vorzuziehen, die sich z. B. auf denselben Pull-Request oder Branch beziehen, da Entwickler:innen ansonsten falsches Feedback erhalten könnten. Die Autoren vermuten, dass die geringe Verwendung dieser Strategie auf ebendiese Herausforderungen zurückzuführen sind [3].

Sowohl auf Pipeline- als auch auf Job-Ebene kann die unnötige Nutzung von Ressourcen verhindert werden, indem spezifische Timeouts definiert werden [20]. Auch wenn die festgelegten Timeouts selten ausgereizt werden, verhindern sie dennoch den maximalen Verbrauch von Ressourcen [20]. Bouzenia und Pradel [3] schlagen vor, die Länge der Timeouts empirisch auf Basis der bisherigen Maxima der Pipeline-Durchläufe zu ermitteln und sicherheitshalber einen Puffer von 10 % hinzuzufügen. Für ihre Daten ermitteln sie dadurch eine Zeitersparnis von etwa 8 %.

Die bisherigen antizipativen Abbruchpraktiken brechen eine Pipeline frühzeitig während ihrer Laufzeit ab. Ähnlich wie beim Tracking der Inaktivität eines Repositorys zur Unterbindung geplanter Pipeline-Ausführungen gibt es weitere Praktiken, die die Ausführung einer Pipeline gänzlich vermeiden und somit in Bezug auf eine einzelne Ausführung maximal Ressourcen einsparen. Diese Praktiken agieren auf Basis von Heuristiken über vergangene Pipeline-Ausführungen. Eine dieser Herangehensweise ist es, alle weiteren Ausführungen zu unterbinden und verantwortliche Personen zu benachrichtigen, sobald k vorherige Pipelines fehlgeschlagen sind [3]. Zusätzlich gibt es verschiedene Ansätze basierend auf Machine-Learning-Modellen oder neuronalen Netzwerken, die ersuchen, das Fehlschlagen einer Pipeline vorherzusagen, um ihre Ausführung vorzeitig zu unterbinden, bis die Pipeline wieder freigegeben wird [55].

Die Praktiken des antizipativen Abbrechens eint, dass sie Ressourcen vermeiden, indem sie den Abbruch einer Pipeline hervorrufen oder ihren Start aufgrund einer hohen Fehlschlagwahrscheinlichkeit verhindern. Daher wurden sie im Spektrum zwischen Ressourcenoptimierung und -vermeidung vor allem der Vermeidung zugeordnet (*Abb. 3.1*).

[1] Interessanterweise wurde das Feature in bezahlten GitHub-Repositorys öfter deaktiviert als in unbezahlten [3], was die Frage nach den Beweggründen aufwirft.

Allgemeine Ressourcenoptimierung: Job-Parallelisierung

Alle bislang angeführten Strategien haben gemein, dass sie sich auf die Optimierung des sequentiellen Programmablaufs beziehen und im besten Fall die Zeit, die die Prozesse die CPU beanspruchen (CPU-Zeit) reduzieren. Durch die hingegen parallele Ausführung von Jobs kann die tatsächliche Laufzeit reduziert werden [58].

Die Parallelisierung von Jobs ist besonders effektiv bei zeitintensiven Jobs wie Testen oder Deployment [58]. Sofern dadurch Ressourcen besser ausgenutzt werden und Geräte früher in einen Leerlaufbetrieb übergehen oder abgeschaltet werden können, kann Parallelisierung den Energieverbrauch einer Pipeline reduzieren [45].
Gegenteilig ist der Energieverbrauch einer Pipeline höher anzunehmen, wenn die bereitgestellten Ressourcen (CPU, RAM etc.) überdimensioniert gewählt sind [45]. Aufgrund ihrer schnellen horizontalen Skalierfähigkeit lohnt es sich daher, Cloud- und Container-Lösungen zu verwenden, um während einer Pipeline-Ausführung die optimalen Ressourcen zur Verfügung zu stellen und diese danach auf ein Minimum zu reduzieren, da selbst sich im Leerlauf befindende Ressourcen weiterhin Energie verbrauchen [58]. Die Schwierigkeit bei der Parallelisierung von Jobs besteht jedoch darin, die Abhängigkeiten zwischen ihnen zu koordinieren, um Konflikte zu vermeiden [46].

Die Job-Parallelisierung und auch die dynamische Zuteilung von Ressourcen sind klar der Ressourcenoptimierung zuzuordnen. Durch die parallele Ausführung und die dynamische horizontale Skalierung wird die Auslastung maximiert und die Gesamtlaufzeit verkürzt (*Abb. 3.1*).

3.2.4 Umsetzung in GitLab

GitLab ist im Kern eine Versionsverwaltung für Programmcode, unterstützt jedoch durch verschiedenen Tools breite Teile des gesamten Software-Development-Zyklus und so auch CI/CD-Prozesse. Im Folgenden wird die technische Umsetzung dieser Prozesse in GitLab dargelegt, sofern sie für die spätere Auswertung relevant und dem Verständnis dienlich sind.

CI/CD-Pipelines werden in GitLab in YAML-Syntax in `gitlab-ci.yml`-Dateien notiert und setzen sich grob strukturiert zusammen aus *Stages* und *Jobs*. Ein Job definiert auszuführende Aufgaben w. z. B. die Kompilierung von Quellcode oder die Ausführung von Testsuiten. Es können zudem Bedingungen angegeben werden, unter denen ein Job

ausgeführt wird oder nicht. Stages umfassen einen oder mehrere Jobs und definieren auf einer höheren Ebene die logische Reihenfolge, in der Jobs ausgeführt werden und die logische Zugehörigkeit der Jobs. Typische Stages sind `build`, `test` und `deploy`. Die Jobs innerhalb einer Stage können potenziell parallel ausgeführt werden, sofern die Konfiguration entsprechend angepasst wurde und genügend Kapazitäten auf dem Runner-Host (s. u.) vorhanden sind. Weiter können die auszuführenden Befehle im Rahmen eines Jobs in drei Skript-Sektionen unterteilt werden: `before_script`, `script` und `after_script`. Die in `before_script` definierten Befehle werden zeitlich vor denen in `script` ausgeführt, während `after_script`-Befehle am Ende laufen [50].

Jeder Job wird von einem Runner ausgeführt. Ein Runner ist eine Applikation, die CI/CD-Jobs von einer GitLab-Instanz empfängt und abarbeitet. Um einen Job auszuführen, verwendet der Runner einen sogenannten Executor, der die Zielumgebung des Jobs bestimmt. Eine häufig genutzte Konfiguration ist der Docker-Executor, welche Jobs in isolierten Docker-Containern startet. Runner sind eigene Prozesse, die auf einem anderen Server als die GitLab-Instanz selbst laufen müssen. Die Konfiguration eines Runners, einschließlich des gewählten Executors, wird in der Datei `config.toml` auf dem Runner-Server vorgenommen [50].

Die Ausführung eines Jobs durch einen Docker-Executor lässt sich vereinfacht in eine Vorbereitungsphase (Pre-Job-Phase), die eigentliche Job-Ausführung (Job-Phase) und eine Nachbereitungsphase (Post-Job-Phase) gliedern. Die Pre-Job- und Post-Job-Phase laufen in speziellen Hilfs-Containern ab. Während der Pre-Job-Phase werden die benötigten Repositorys geklont sowie, sofern konfiguriert, Caches und Artefakte geladen. In der Post-Job-Phase werden eventuelle Caches erstellt und Artefakte für nachfolgende Jobs gespeichert. Die eigentliche Job-Ausführung, welche die oben genannten Skript-Sektionen abarbeitet, erfolgt in der Job-Phase und in einem Container basierend auf dem in der `.gitlab-ci.yml` für den Job definierten Image. Wenn kein Image angegeben wurde, greift der Docker-Executor auf ein Standard-Image zurück [27].

In Kombination mit dem Docker-Executor nutzt GitLab für das lokale Caching Docker-Volumes, um die Cache-Verzeichnisse zu speichern [26]. Obwohl diese Verzeichnisse in den Job-Container eingebunden werden, liegen die Daten physisch auf dem Dateisystem des Runner-Hosts [26]. Ein zusätzlicher Effizienzvorteil entsteht durch die Nutzung des Docker-Speichertreibers `overlay2`, der eine gemeinsame Verwendung des Page-Caches ermöglicht [10]. Nachdem die Cache-Daten einmalig von der Festplatte gelesen wurden, hält das Betriebssystem des Hosts eine Kopie dieser Daten im Page-Cache des

Arbeitsspeichers vor [32]. Nachfolgende Lesezugriffe können somit direkt aus dem Arbeitsspeicher bedient werden, wodurch die I/O-Aktivität auf der Festplatte reduziert wird.

Die in Kapitel 3.2.3 vorgestellten Best Practices eröffnen ein breites Spektrum an Möglichkeiten zur Optimierung und Vermeidung von Ressourcen bei der Verwendung von CI/CD-Pipelines. Die nachfolgende empirische Analyse konzentriert sich auf die beiden Praktiken *Parallelisierung* und *Dependency-Caching*, die zuvor primär der Ressourcenoptimierung zugeordnet wurden 3.1. Wie aus den Grundlagen und im Kapitel zum Stand der Forschung herausgearbeitet, versprechen die ausgewählten Strategien ein enormes Energiesparpotenzial. Während Dependency-Caching die Optimierung der als besonders ressourcenintensiv identifizierten Build- und Test-Jobs anvisiert, wird durch Job-Parallelisierung die optimierte Nutzung der auf dem Runner-Host verfügbaren Ressourcen angestrebt. Aufbauend auf den theoretischen Grundlagen wird im folgenden Kapitel die Methodik zur systematischen Untersuchung der Auswirkungen beider Strategien sowie ihrer Kombination in einer kontrollierten Versuchsumgebung dargelegt.

4 Methodik

Bevor die Ergebnisse der Untersuchung präsentiert werden, wird nachfolgend auf ihre methodische Grundlage eingegangen. Zunächst wird die Versuchsumgebung vorgestellt, in dem auf die Referenzpipeline sowie die Hard- und Softwareumgebung der Untersuchung eingegangen wird. Daran anknüpfend werden die verschiedenen Untersuchungsszenarien und ihre technische Umsetzung beschrieben. Abschließend wird die Vorgehensweise bei der Datenerhebung und -auswertung, einschließlich der verwendeten Werkzeuge und Methoden erläutert.

4.1 Versuchsumgebung

Nachfolgend wird zunächst die für alle durchgeführten Experimente als Ausgangszustand dienende Referenzpipeline vorgestellt. Daran anschließend werden die für das Experimentdesign relevanten Hard- und Softwarekonfigurationen präsentiert.

4.1.1 Die Referenzpipeline

Die Referenzpipeline setzt sich aus den drei Stages `build`, `test` und `deploy` zusammen. Die Stages werden in der genannten Reihenfolge sequentiell ausgeführt, d. h., dass Jobs einer nachfolgenden Stage erst ausgeführt werden, sobald alle Jobs der vorherigen Stage erfolgreich abgeschlossen wurden. Dies wird einerseits durch die standardmäßige Konfiguration in GitLab-CI/CD-Pipelines sichergestellt und andererseits, indem in der Runner-Konfiguration des GitLab-Docker-Runners (`config.toml`) die Anzahl der maximal parallel laufenden Jobs auf 1 (`concurrent=1`) gesetzt wurde. Zur Gewährleistung identischer und reproduzierbarer Umgebungsbedingungen werden alle Jobs in Docker-Containern ausgeführt.

Innerhalb der `build`-Stage sind die beiden Jobs `build_api` und `build_frontend`, nachfolgend auch Build-API-Job und Build-Frontend-Job genannt, angesiedelt. Ersterer kompiliert eine auf dem Spring-Boot-Framework basierende Java-Anwendung mittels *Maven*. Der Build-Frontend-Job ist für den Build-Prozess eines auf Vue.js basierenden Frontends zuständig und nutzt hierfür eine Node.js-Laufzeitumgebung sowie den Node-Package-Manager *npm*. Beide Jobs sind geeignet für die Untersuchung von Dependency-Caching (H1), da sie durch die Befehle `mvn clean package` bzw. `npm install` eine Vielzahl von Dependencies herunterladen. Da beide Jobs in derselben Stage angesiedelt sind und keine Abhängigkeiten zueinander aufweisen, eignen sie sich zudem für die parallele Ausführung, was die Grundlage für die Untersuchung der Hypothese *H2* schafft.

An die Build-Phase schließt sich die `test`-Stage an, welche den Job `test_api` (Test-API-Job oder auch Test-Job) zur Ausführung der Unit-Tests der Backend-Anwendung enthält. Dieser Job ist explizit vom Build-API-Job abhängig, da er dessen erzeugte Artefakte für die Testdurchführung benötigt und somit erst nach dessen Abschluss starten kann. Zusätzlich zu den Build-Dependencies lädt der Test-Job weitere Dependencies für die Durchführung der Unit-Tests herunter. Indem der Test-Job nicht vom Frontend-Build-Job abhängig ist, ist er, sofern der Build-API- schneller abläuft als der Build-Frontend-Job, potenziell Untersuchungsgegenstand der zweiten Hypothese. Durch das Herunterladen der Dependencies ist er in jedem Fall Teil der Untersuchung der ersten Hypothese.

Den Abschluss der Pipeline bildet die `deploy`-Stage mit dem `deploy`-Job (Deployment-Job). Seine Aufgabe auf dem GitLab-Runner beschränkt sich auf das Kopieren der von den Build-Jobs bereitgestellten Build-Artefakte auf einen Ziel-Server mittels `SCP` und das Anstoßen von Remote-Befehlen via `SSH`. Der ressourcenintensive Teil des Deployments, w. z. B. das Bauen von Docker-Images, findet folglich auf dem Ziel-Server statt und ist nicht Teil der Messungen. Da der Deployment-Job auf die (überprüften) Artefakte aus der Build-Stage angewiesen ist, ist er vom Test-API- und vom Build-Frontend-Job abhängig und kann entsprechend nicht parallelisiert werden. Ebenso eröffnen sich keine Möglichkeiten für Dependency-Caching, da der Job bereits mit den fertigen Artefakten versorgt wird.

4.1.2 Hard- und Softwareumgebung

Für die Durchführung der Experimente wurde ein eigener GitLab-Runner-Host in Form einer VM auf einem Server in der ZBW eingerichtet. Dieser Schritt soll die Vergleichbarkeit und Reproduzierbarkeit der Messergebnisse gewährleisten.

Der VM wurden 8 vCPUs und 16 GB RAM zugewiesen. Als Betriebssystem kommt Ubuntu in der Version 24.04.3 LTS zum Einsatz. Der darunterliegende Server basiert auf der Intel Cascade Lake-Architektur von 2019 und ist mit zwei Intel Xeon Gold 6246 Prozessoren ausgestattet. Diese verfügen zusammen über 24 physischen Kernen (48 Threads) bei einer Basistaktrate von 3.30GHz. Ergänzt wird das Setup durch 384 GB Arbeitsspeicher.

Die CI/CD-Prozesse wurden über die API einer in der ZBW gehosteten GitLab-Instanz in der Version 18.0 gesteuert. Für die Ausführung der Pipeline-Jobs kam der GitLab-Runner in der Version 18.1.1 zum Einsatz, der so konfiguriert wurde, dass er den Docker-Executor verwendet. Docker selbst war in der Version 28.3.0 auf dem Runner-Host installiert und mit dem standardmäßigen Speichertreiber *overlay2* konfiguriert.

4.2 Vorstellung und Umsetzung der Szenarien

Die Untersuchung der aufgestellten Hypothese fundiert auf der Durchführung der Messung in verschiedenen Szenarien. Das Basis-Szenario – die Referenzpipeline – wurde bereits vorgestellt. Dessen Messwerte dienen grundlegend als Referenzpunkte für die Bewertung der anderen Szenarien, deren Funktion im Rahmen der Arbeit und deren technische Umsetzung im Folgenden kurz vorgestellt wird.

Cache-Aufbau- und Cache-Nutzungs-Szenario

Die aufgestellte Hypothese *H1* bringt die Notwendigkeit mit sich, die Effekte des Cachings nicht als ein Ganzes zu untersuchen, sondern die Anwendung von Caching zu unterteilen in den Aufbau und die Nutzung von Caches. Diese Unterteilung wird gewährleistet, indem die Referenzpipeline so angepasst wurde, dass immer zwei aufeinanderfolgende Pipeline-Durchläufe sich einen Cache teilen, wobei der erste Pipeline-Durchlauf den Cache aufbaut und der zweite ihn nutzt.

Technisch wurde dies umgesetzt, indem beim Start jeder Pipeline der eindeutige Name des zu verwendenden Caches übergeben wurde. Der Name setzt sich zusammen aus dem Namen des aktuellen Szenarios und der aufgerundeten, durch zwei geteilten aktuellen Iterationsnummer. Durch diese Namenskonvention wird sichergestellt, dass genau zwei aufeinanderfolgende Pipeline-Durchläufe sich dieselben Dependency-Caches teilen: Immer der erste der beiden Durchläufe baut die Caches auf und der zweite nutzt sie. Um die Caches den Jobs zuzuordnen, wird der Cache-Namen-Variable zudem der Kurzname des jeweils verwendeten Package-Managers (npm, mvn) als Präfix vorangestellt und das Resultat im Cache-Abschnitt des Jobs als Schlüssel für den Cache verwendet. Damit wird zudem erreicht, dass der Build-API- und der Test-API-Job, die zum Großteil dieselben Dependencies verwenden, innerhalb einer Pipeline auf denselben Cache zugreifen.
Die Aufteilung der Messdaten auf das Cache-Aufbau- sowie das Cache-Nutzungs-Szenario erfolgt postum auf Basis der Iterationsnummern.

Parallel-Szenario

Der standardmäßige Aufbau einer GitLab-Pipeline sieht vor, dass Stages und die in ihnen gebündelten Jobs sequentiell ablaufen. Für die Untersuchung der in Hypothese *H2* geforderten Stage-übergreifenden parallelen Ausführung von Jobs wurde zunächst in allen Jobs über das `needs`-Schlüsselwort festgelegt, von welchen anderen Jobs sie abhängig sind. Da die beiden Build-Jobs keine Abhängigkeiten zu anderen Jobs aufweisen, wurde dies entsprechend über ein leeres Array deklariert. Für den Test-API-Job wurde der Build-API-Job und für den Deployment-Job der Test-API- sowie der Build-Frontend-Job als Abhängigkeiten angegeben. Aus diesen Abhängigkeitsbeziehungen ergibt sich eine potenzielle Parallelisierung der beiden Build-Jobs sowie des Test-API- und des Build-Frontend-Jobs. Für die letztliche Aktivierung der Parallelisierung wurde für alle Szenarien mit aktivierter Parallelisierung der Wert für `concurrent` in der Runner-Konfiguration (`config.toml`) von 1 auf 8 gesetzt. Dies entspricht der Anzahl der maximal verfügbaren vCPUs.

Parallel-Aufbau- und Parallel-Nutzungs-Szenario

Die beiden Szenarien Parallel-Aufbau und Parallel-Nutzung dienen der Untersuchung der kombinierten Verwendung von Caching und Parallelisierung, die aus der dritten Hypothese hervorgeht. Wie die Namen vermuten lassen, untersucht das Parallel-Aufbau-Szenario

die Kombination aus Parallelisierung und Cache-Aufbau und das Parallel-Nutzungs-Szenario die aus Parallelisierung und Cache-Nutzung. Auch technisch entspricht die Umsetzung einer Kombination der oben beschriebenen Vorgehensweisen für Caching und Parallelisierung.

4.3 Datenerhebung und -auswertung

Im Folgenden wird das Vorgehen bei der Datenerhebung und -auswertung beschrieben. Hierfür wird zunächst der Prozess zur Sammlung der Messdaten vorgestellt. Anschließend werden die Methoden zur Aufbereitung und statistischen Auswertung der erhobenen Daten dargelegt.

4.3.1 Prozess der Datenerhebung

Um eine systematische und reproduzierbare Erfassung der für diese Arbeit relevanten Daten zu gewährleisten, wurde ein mehrteiliger, automatisierter Messprozess konzipiert. Dieser umfasst Messungen auf dem Runner-Host und in den einzelnen Pipeline-Jobs sowie Abfragen von Metadaten mittels der GitLab-API.

Zur Sammlung der Ressourcenwerte wurde das Werkzeug `sar` (System Activity Reporter) aus dem `sysstat`-Paket eingesetzt. Hierbei handelt es sich um ein softwarebasiertes Tool für die Messung systemweiter Ressourcen, wie sie im Grundlagenkapitel (*Kap. 3.2.2*) vorgestellt werden. `sar` wurde ausgewählt, da es sich als etablierte Anwendung unkompliziert sowohl auf dem Runner-Host als auch in die Pipeline-Jobs einbinden lässt. Zudem wird durch diese Art der Messung kaum zusätzlicher Overhead verursacht (*Kap. 3.2.2*). Für die Untersuchung wurden unter anderem die Metriken zur CPU-Auslastung (`%user`, `%system`, `%idle`), RAM-Auslastung (`%memused`) sowie die I/O-Last (`bread/s`, `bwrtn/s`) sekündlich mittels `sar` aufgezeichnet. Die Messungen erfolgten hierbei auf zwei Ebenen. Zum einen wurde eine Blackbox-Messung auf dem GitLab-Runner-Host durchgeführt, um die Gesamtressourcennutzung während eines vollständigen Pipeline-Durchlaufs zu erfassen. Zum anderen wurde `sar` in die Pipeline-Jobs integriert, indem die Messungen im `before_script`-Abschnitt der `.gitlab-ci.yml` gestartet und im `after_script`-Abschnitt beendet wurden. Die resultierenden Daten wurden als Job-Artefakte gespeichert, um die Job-spezifische Analyse zu ermöglichen.
Zusätzlich zu den geschilderten Messungen wurden relevante Metadaten der Pipeline und

```
PROGRAMMSTART(branch, szenario, n_iter, caching_aktiv)

FÜR i = 1 BIS n_iterationen:
1. MESSUNG STARTEN:
    - Starte sar-Skript auf Runner-Host
    - WENN caching_aktiviert:
        - cache_name = ERMITTLE_CACHE_NAME(szenario, i)
        - Starte Pipeline(branch, cache_name)
    - SONST:
        - Starte Pipeline(branch)

2. PIPELINE ÜBERWACHEN:
    - WARTE_AUF_PIPELINE_ENDE(timeout=7min)
    - Stoppe sar-Skript auf Runner-Host

3. ERGEBNISSE VERARBEITEN:
    - WENN Pipeline ERFOLGREICH:
        - Lade Host-Messdaten & Job-Artefakte
        - Lade Pipeline- & Job-Metadaten
        - Speichere alle Daten für Iteration i
        - Passe adaptive Wartezeit an
    - SONST:
        - Protokolliere Fehler
        - Abbruch nach 3 Fehlversuchen
```

Abbildung 4.1: Pseudo-Code-Darstellung des automatisierten Messablaufs

der Jobs über die GitLab-API abgefragt. Diese umfassen unter anderem die Pipeline- und Job-ID, den Status der Pipeline, den Namen des Pipeline-Durchlaufs sowie die genauen Start- und End-Zeitpunkte.

Der grundsätzliche Ablauf einer Messreihe für ein Szenario wurde durch ein lokales Python-Skript automatisiert und folgt dem in Abbildung 4.1 skizzierten Schema. Für jedes Szenario wurde eine eigene Git-Branch angelegt, auf der die `gitlab-ci.yml` der Referenzpipeline den Kriterien aus Kapitel 4.2 entsprechend angepasst wurde. Der automatisierte Ablauf beginnt mit dem Start des Mess-Skripts, das für eine übergebene Anzahl an Iterationen die Messungen anstößt. In jeder Iteration wird zunächst die `sar`-Messung auf dem Runner-Host per SSH gestartet, bevor eine neue GitLab-Pipeline auf dem entsprechenden Branch getriggert wird. Falls Caching für das Szenario aktiviert ist, wird der Cache-Name generiert und an die Pipeline übergeben (*Kap. 4.2*). Anschließend überwacht das Skript den Fortschritt der Pipeline und stoppt die Host-Messung nach Beendigung der Pipeline. Bei einem erfolgreichen Durchlauf werden alle erzeugten Daten – die Host-Metriken, Pipeline- und Job-Metadaten sowie die als Artefakte gespeicherten Job-Messungen – heruntergeladen und lokal für die spätere Auswertung abgelegt.

Bei Fehlschlägen wird der Fehler protokolliert und die Messreihe nach drei aufeinanderfolgenden Fehlversuchen abgebrochen.

Die automatisierte Messreihe wurde für Szenarien mit Caching 120 Mal durchgeführt, wobei sich dies auf 60 Iterationen mit Cache-Aufbau und 60 Iterationen mit Cache-Nutzung aufteilt. Für das Basis- sowie das Parallel-Szenario wurden gleichermaßen jeweils 60 Messungen durchgeführt.

4.3.2 Datenaufbereitung und Auswertungsmethoden

Nach Vollendung der Messreihen wurden die resultierenden CSV- und JSON-Dateien jeder Iteration mithilfe von Python-Skripten in Jupyter-Notebooks weiterverarbeitet. Initial wurden die Daten in vier CSV-Dateien je Szenario überführt, welche die Pipeline-Metadaten, Job-Metadaten sowie die Messergebnisse des `sar`-Tools auf Job- und Host-Ebene enthalten. Jede Zeile wurde dabei, sofern zutreffend, um die entsprechenden Informationen zu Szenario, Job und Iterationsnummer angereichert. Um eine detaillierte Analyse der Ressourcennutzung auf Job-Ebene zu ermöglichen, wurden die Host-Messungen über ein Mapping der Start- und Endzeitstempel aus den Job-Metadaten dem jeweils laufenden Job zugeordnet. Zudem wurden die Messdaten jeder Iteration über die Zeitangaben in den Pipeline-Metadaten auf den tatsächlichen Zeitraum der Pipeline eingeschränkt.

Des Weiteren wurden die `sar`-Messungen um Leistungswerte in Watt erweitert. In Orientierung an der Vorgehensweise im Projekt *Eco CI* [31] wurden die Werte mittels des quelloffenen Machine-Learning-Modells *Cloud Energy Model* der Firma Green Coding Solutions GmbH berechnet [30]. Hierbei handelt es sich um ein XGBoost-Modell, das speziell für die Schätzung der Leistungsaufnahme in Umgebungen wie Cloud- oder virtualisierten Systemen entwickelt wurde, in denen direkte Leistungsmessungen nur bedingt möglich sind [34]. Als Trainingsgrundlage des Modells diente der SPECPower-Benchmark-Datensatz, der detaillierte Leistungs- und Verbrauchsdaten diverser Serverkonfigurationen enthält [34, 54].
Einziger obligatorischer Eingabeparameter für das Modell ist ein Stream von CPU-Auslastungswerten [34]. Um eine möglichst hohe Genauigkeit zu gewährleisten, wurde das Modell aber durch die Übergabe aller optionalen, hardwarespezifischen Merkmale an die Versuchsumgebung angepasst: `-cpu-chips 2`, `-cpu-freq 3300`, `-cpu-threads`

`48`, `-cpu-cores 24`, `-release-year 2019`, `-tdp 165`, `-ram 384`, `-architecture cascadelake`, `-cpu-make intel` und `-vhost-ratio 0.166`. Der Parameter `vhost-ratio` dient hierbei als Korrekturfaktor, um der VM nur den anteiligen Energieverbrauch zuzuschreiben. Mit Blick auf die Genauigkeit der absoluten Werte ist hierbei problematisch, dass von einem ausbalancierten Verhältnis von vCPUs und RAM ausgegangen wird, der im hiesigen Setting nicht gegeben ist [30].
In einer internen Untersuchung erreichte das Cloud Energy Model laut Guldner et al. [34] eine Genauigkeit, die mit dem von Rteil et al. [54] vorgestellten Modell vergleichbar ist, bei einem Fehler von etwa 10 Watt (ca. 10 %). Auch dieses Modell basiert auf dem SPECPower-Benchmark-Datensatz. In der Methodenkritik (*Kap. 6.4*) wird auf die Einschränkungen der hier vorgestellten Berechnung der Leistungswerte weiter eingegangen.

Da sich das im vorherigen Absatz genannte Messtool Eco CI [31] mit wenigen Konfigurationsschritten über die Skript-Sektionen auf Job-Ebene in CI/CD-Pipelines einbinden lässt, wurde dessen Verwendung für die Arbeit in Erwägung gezogen. Jedoch gibt es die gemessenen Werte nur in aggregierter Form aus und hätte folglich in Weiten Teilen für die Verwendung angepasst werden müssen, um Einzelmessungen zu erhalten, zumal ausschließlich die CPU-Auslastung gemessen und darauf basierend die Leistung ermittelt wird. Außerdem spiegeln die Messungen durch Eco CI, ebenso wie die hier verwendeten In-Job-Messungen, lediglich die Job-Phase der Jobs wider und lassen somit die Prozesse der Pre-Job- und Post-Job-Phase, die u. a. das Laden und Speichern von Caches enthalten, gänzlich außer Acht. Letztendlich hat sich daher die doppelte und einheitliche Verwendung von `sar` als der passendere Ansatz für die hier anvisierte detaillierte Analyse herauskristallisiert.

Um die Auswirkungen systeminterner Cachingvorgänge zu minimieren, wurde jeweils die erste Messung jedes Szenarios entfernt. Nach einer tiefergehenden Untersuchung der Datenbasis wurden des Weiteren auf Basis der direkt gemessenen Ressourcenmetriken (I/O-Operationen, CPU- und RAM-Auslastung) sowie der Laufzeit Ausreißer identifiziert. Konkret wurde für jeden Pipeline-Job innerhalb eines Szenarios der Wertebereich für Ausreißer anhand des Interquartilsabstands (IQR) bestimmt. Hierbei wurden die unteren und oberen Grenzen so gesetzt, dass Werte außerhalb des Intervalls $(Q1-3*IQR, Q3+3*IQR)$ als Ausreißer klassifiziert werden. Zusätzlich wurde eine Mindestabweichung von 5 % um den Median definiert, um kleine Schwankungen nicht fälschlich als Ausreißer zu werten. Durch diese Vorgehensweise wurden Messungen, die extrem von einer Norm abweichen, aus der Auswertung entfernt, während gleichzeitig legitime Abweichungen erhalten blei-

ben.
Nach diesen Bereinigungsschritten enthalten die Szenarien noch mindestens 46 Messungen. Beim Basis-Szenario wurde lediglich eine Messung als Ausreißer klassifiziert, sodass hier inklusive der entfernten ersten Messung 58 der ursprünglich 60 Messungen in die Auswertung eingehen. Die genaue Anzahl der resultierenden Messungen je Szenario kann der Tabelle A.1 entnommen werden.

Für die Feststellung von Unterschieden zwischen Messwerten wurde ein Signifikanzniveau von 95 % ($p \leq 0,05$) angesetzt. Die Signifikanz wurde in Abhängigkeit davon, ob beide Vergleichsgruppen normalverteilt sind oder nicht, mittels des „Welch t“-Tests (normalverteilt) oder des zweiseitigen „Mann-Whitney U“-Tests (nicht normalverteilt) ermittelt. Auf Normalverteilung wurde mit dem „Shapiro-Wilk“-Test geprüft. Im Ergebnisteil angeführte Unterscheidungen zwischen Vergleichsgruppen sind, sofern nicht anders angegeben, als signifikant anzunehmen.

5 Ergebnisse

Im Folgenden werden die Messergebnisse der durchgeführten Experimente präsentiert. Als Erstes wird die Referenzpipeline anhand ihrer Metriken charakterisiert. Daran anschließend werden die Auswirkungen von Cache-Aufbau und Cache-Nutzungen auf die Pipeline beleuchtet (H1), gefolgt von einer Aufschlüsselung der Effekte der Parallelisierung der Pipeline-Jobs (H2). Zuletzt wird die Kombination der Praktiken in Betracht genommen sowie alle Szenarien gegenübergestellt (H3).

5.1 Performance der Referenzpipeline

Die Referenzpipeline weist im Schnitt eine Laufzeit von 2:41 Minuten auf, von der sie in aller Regel nicht mehr als ca. 3 Sekunden abweicht (*Tab. A.1*). In ihren Extremfällen erreicht sie eine Laufzeit von 2:32 Minuten (Min.) bzw. 2:49 Minuten (Max.). Die CPU-Auslastung liegt im Mittel bei 40,75 % und weist eine nur geringe Schwankung auf (SD: 1,33 %). Unterstützend liegen das Minimum und Maximum der CPU-Auslastung lediglich 5,59 Prozentpunkte auseinander.
Der Unterschied zwischen der minimal und der maximal verzeichneten I/O-Nutzung beträgt absolut 230,47 MiB und somit weniger als 1/7 der mittleren I/O-Nutzung von 1722,56 MiB. Die RAM-Auslastung der Referenzpipeline liegt im Durchschnitt bei 17,06 % und zeigt mit einer Standardabweichung von unter einem halben Prozent eine über die Messdurchläufe hinweg konstante Nutzung des Arbeitsspeichers. Ähnlich konstant verhält sich der Energieverbrauch der verschiedenen Pipeline-Durchläufe, der sich maximal zwischen 1,36 Wh und 1,54 Wh bewegt und einen Mittelwert von 1,45 Wh bei einer standardmäßigen Abweichung von 0,05 Wh annimmt. Zumindest die Pipeline als Ganzes betrachtend, verstärkt sich das Bild, dass die Referenzpipeline durch eine hohe Stabilität und Vorhersagbarkeit in den Metriken geprägt ist.

Ein ähnliches Bild spiegelt sich wider in der Betrachtung der einzelnen Jobs. Auch hier ist über die Boxplots in *Abb 5.1* ersichtlich, dass die Jobs je Metrik nur in geringem

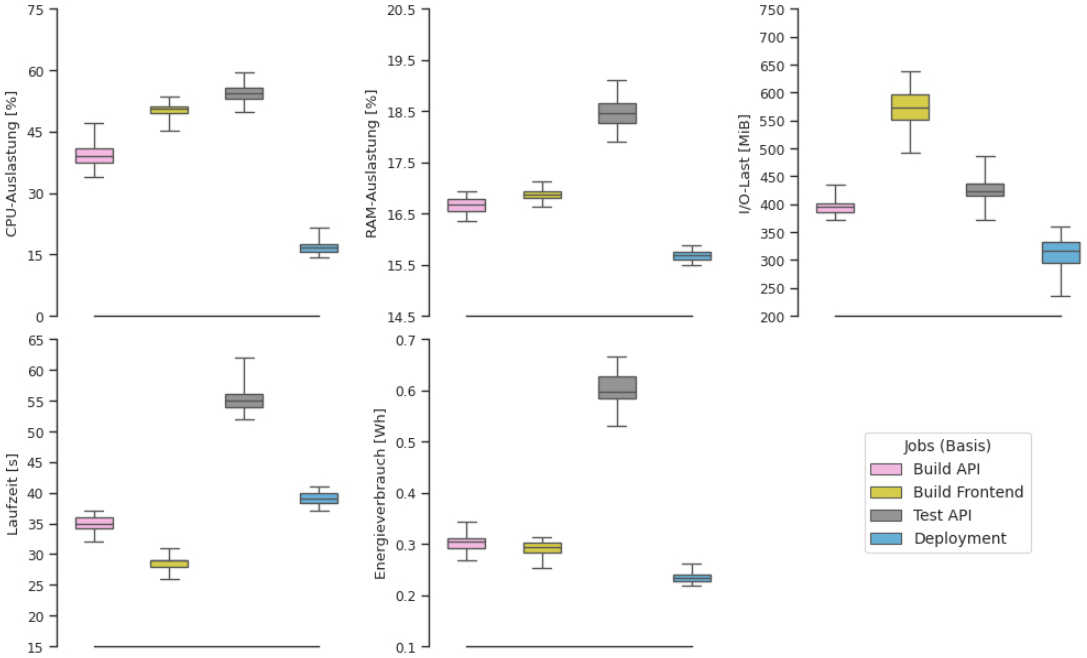

Abbildung 5.1: Ressourcenverteilung im Basis-Szenario je Job. Die Whiskers sind auf das Dreifache des IQR gesetzt.

Maße Schwankungen aufweisen. Der Deployment-Job verzeichnet über alle Ressourcenmetriken hinweg im Mittel die geringsten Werte, was aufgrund der Tatsache, dass der ressourcenintensive Anteil des Deployments auf einem externen Deployment-Server geschieht, nicht weiter verwunderlich ist (*Kap. 4.1*). Konkret ist der minimale Energiebedarf des Deployment-Jobs im Basis-Szenario lediglich um 0,05 Wh höher angesiedelt als der durchschnittliche Energiebedarf im Leerlauf über die minimale Laufzeit des Deployment-Jobs von 38 Sekunden (Leerlauf: 0,15 Wh, Deployment: 0,2 Wh).
Konträr zum Deployment benötigt das Testen der API mit Ausnahme der Menge der I/O-Operationen über alle Metriken hinweg im Schnitt die meisten Ressourcen und hat die längste Laufzeit von durchschnittlich 55,12 Sekunden. Damit macht der Test-Job ca. ein Drittel der Gesamtlaufzeit der Referenzpipeline aus und benötigt fast doppelt so viel Zeit wie die mittlere Laufzeit des Frontend-Builds, welches mit 28,59 Sekunden die geringste Laufzeit verzeichnet. Auch das Build der API ist in der Regel knapp 16 Sekunden schneller als der Test-Job. Ebenso drastisch sind die Unterschiede zwischen dem Test der API und den beiden Build-Jobs bezüglich ihres Energiebedarfs. Das Testen der API benötigt im Schnitt 0,6 Wh, (mehr als) doppelt so viel wie das Frontend- oder das API-Build (Frontend: 0,29 Wh, API: 0,3 Wh).

Der Energiebedarf der beiden Build-Jobs unterscheidet sich nur marginal um 3,77 %. Interessant ist hierbei, dass das Frontend-Build allerdings um ein Drittel mehr I/O-Operationen durchführt als das API-Build, ebenso eine um knapp ein Viertel gesteigerte CPU-Auslastung und eine etwas gesteigerte RAM-Auslastung aufweist, aber dennoch um fast ein Viertel schneller ist als das API-Build.

Ein breiteres Verständnis dahingehend, wie sich die Ressourcenwerte über den Zeitraum einer Pipeline entwickeln und wie sich die einzelnen Jobs auf die Entwicklung der Ressourcenwerte auswirken, kann aus der Abbildung 5.2 gewonnen werden. Die Abbildung zeigt vergleichend die Entwicklung der Messwerte je Messung für die Referenzpipeline (rechts) und die Pipeline mit Cache-Aufbau (links). Für den direkten Vergleich zwischen den Szenarien wurden die Werte auf einen Wertebereich zwischen 0 und 1 normiert (y-Achse). Hierbei entspricht 1 dem Maximum der jeweiligen Metrik über beide Szenarien hinweg. Jede dünne Linie im Diagramm repräsentiert die Entwicklung der Messwerte für einen Messdurchlauf, während die dick hervorgehobenen Linien in der jeweils selben Farbe den gemittelten Verlauf über alle Messungen markieren. Die Zeitangaben wurden je Szenario auf Basis der mittleren Laufzeit normiert (x-Achse). Auf den Vergleich zwischen den beiden Szenarien wird an späterer Stelle im Kontext der Analyse der Effekte des Cache-Aufbaus eingegangen. An dieser Stelle ist zunächst nur das rechte Zeitreihen-Diagramm für das Basis-Szenario von Interesse.

Im zeitlichen Verlauf der Referenzpipeline lassen sich ausgeprägte Schwankungen aller betrachteten Metriken erkennen (*Abb. 5.2*). Aufgrund der Tatsache, dass hier die gesamte Pipeline betrachtet wird, unter der vier verschiedene Jobs subsumiert werden, ist dies ein erwartbares Bild. Weiter ist zu beobachten, dass die gemessenen Watt-Werte der Entwicklung der CPU-Auslastung sehr stark folgen. Auffällig sind in diesem Zusammenhang die drei ausgeprägten Spitzen der CPU-Auslastung und respektive auch der Leistungsaufnahme (Sekunde: 22, 50, 99). Diesen Spitzen folgen mit kleinem zeitlichen Versatz Spitzen der RAM-Auslastung (Sekunde: 26, 55, 109), welche z. T. wiederum in erhöhte I/O-Werte münden (Sekunde: 30, 62), da z. B. vom RAM in den Speicher geschrieben wird und in der Folge die RAM-Auslastung abnimmt, jedoch die I/O-Last zunimmt.

Jede der ausgemachten CPU-, Watt- und RAM-Spitzen lässt sich einem der Build- oder Test-Jobs zuordnen. Bei mittlerer Laufzeit spiegelt der Bereich von Sekunde 0–35 die Entwicklung des API-Builds wider, Sekunde 36–65 Frontend-Build und Sekunde 66–121 das Testen der API. Das Deployment ist folglich dem verbliebenen Bereich von Sekunde 122–161 zuzuordnen. Zwar ist auch hier ein Höhepunkt der CPU-Auslastung und Leis-

tungsaufnahme erkennbar (Sekunde 131), allerdings ist dieser nur unwesentlich höher als die Minima während der anderen Jobs.
Mit dem Wissen über die Verteilung der Jobs lassen sich Muster in der Entwicklung der I/O-Operationen erkennen: In den ersten 9–10 Sekunden eines Jobs wir das lokal gespeicherte Docker-Image für den Container geladen, sowie der Docker-Container gestartet. Dies ist vor allem ca. 1–2 Sekunden nach Beginn der beiden Build-Jobs sowie des Deployments an dem synchronen CPU-, Watt- und relativ kleinen I/O-Peak zu erkennen. Als Nächstes folgt innerhalb dieser Vorbereitungsphase der Jobs das Klonen des GitLab-Projekts, markiert durch einen leicht höheren I/O-Höhepunkt ca. 7–8 Sekunden nach Job-Start, der bei allen Jobs ersichtlich ist. Beim Test- sowie beim Deployment-Job ist dieser Höhepunkt etwas höher und breiter ausgeprägt, da bei ersterem zusätzlich die bereitgestellten Artefakte des API-Builds geladen werden und beim letzteren die Artefakte beider Builds. Das Abspeichern der Build-Artefakte ist den Höhepunkten der I/O jeweils am Ende der Build-Jobs zu entnehmen (Sekunde: 30, 62). Der zentrale Höhepunkt der I/O im Build-Job des Frontends ist dem `npm`-Build zuzuschreiben und dem damit einhergehenden Schreiben des Ausgabeverzeichnisses `dist`. Zuletzt lassen sich die I/O-Bewegungen zwischen Sekunde 89 und 109 während der Job-Phase des Test-Jobs dem wiederholten Laden des Spring-`ApplicationContext` aufgrund von `DirtiesContext`-Annotationen in manchen Tests zuzuordnen.

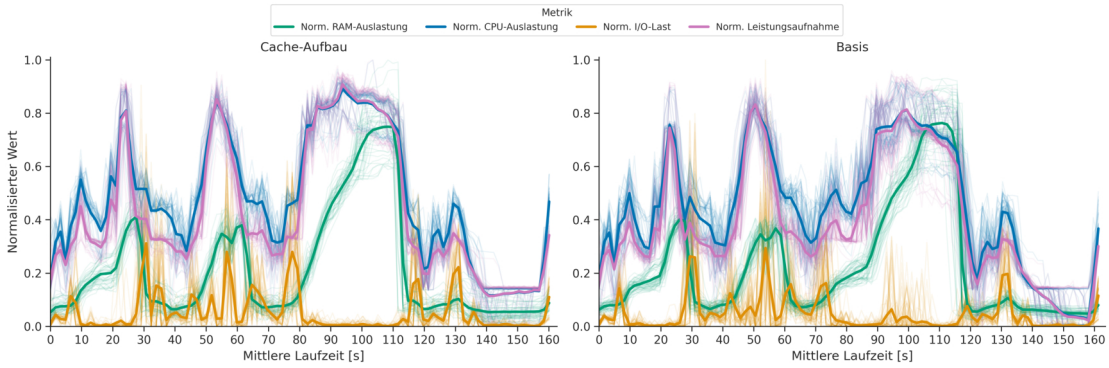

Abbildung 5.2: Vergleich des zeitlichen Verlaufs des Cache-Aufbau- (links) und des Basis-Szenarios (rechts) anhand der zentralen Messwerte. Messwerte wurden für den Vergleich normiert. Zeitwerte geben die mittlere Laufzeit an. Hervorgehobene Linien: mittlerer Verlauf der jeweiligen Ressource.

5.2 H1: Dependency-Caching

Die Auswirkungen von Dependency-Caching auf die Ressourcennutzung und Laufzeit wurden anhand der beiden Build-Jobs und des Test-Jobs untersucht. Wie in der Methodik beschrieben, ist für die Betrachtung der Effekte von Caching die Unterteilung in Cache-Aufbau und Cache-Nutzung wichtig. Die folgende Analyse orientiert sich an Hypothese *H1* und vergleicht die Ressourcennutzung der Caching-Szenarien untereinander sowie mit dem Basis-Szenario. In einer Art Zirkelschluss wird zuerst das Basis- mit dem Cache-Aufbau-Szenario, dann das Cache-Aufbau- mit dem Cache-Nutzungs-Szenario und zuletzt, den Kreis schließend, das Cache-Nutzungs- mit dem Basis-Szenario verglichen. Wo angebracht, wird von der Betrachtung auf Pipeline-Ebene in die Job-Ebene gewechselt, um die Zusammenhänge besser zu veranschaulichen.

5.2.1 Cache-Aufbau vs. Basis

Das Cache-Aufbau-Szenario unterscheidet sich vom Basis-Szenario vor allem durch eine um 8,25 % gesteigerte CPU-Auslastung (MW: 44,11 %) sowie einer Steigerung der I/O-Operationen um knapp ein Siebtel (MW: 1976,86 MiB).

Ein Pipeline-Durchlauf mit Cache-Aufbau benötigt im Schnitt 2:40 Minuten und ist damit knapp eine Sekunde schneller als die Referenz-Pipeline (nicht signifikant), bei geringerer, jedoch ähnlich starker Standardabweichung von 3,22 Sekunden (*Tab. A.1*). Trotz fehlender Signifikanz ist die im Mittel verringerte Laufzeit durchaus plausibel, da das Testen der API im Schnitt um 10 % (5,5 Sekunden) früher abgeschlossen ist (*Abb. 5.3*). Dass sich dies nur bedingt auf die Gesamtlaufzeit der Pipeline niederschlägt, hängt unter anderem mit der gegenläufig gesteigerten Laufzeit des Frontend- (ca. 1 Sekunde, +2,8 %) und des API-Builds (ca. 3 Sekunden, +9,32 %) zusammen. Beim Testen der API steigen zudem die CPU-Auslastung (+12,3 %) und I/O-Last (+23,85 %) deutlich an.
Ebenso wie die Gesamtlaufzeit hat die mittlere RAM-Auslastung im Cache-Aufbau-Szenario im Vergleich zum Basis-Szenario leicht abgenommen (-0,36 %). Dieser Unterschied ist ebenfalls nicht-signifikant. Entsprechend der gesteigerten CPU-Auslastung und I/O-Operationen sowie der geringfügigen Abweichungen vom Basis-Szenario in den anderen Vergleichswerten nimmt die Energienutzung durchschnittlich um 4,57 % (MW: 1,52 Wh) zu.

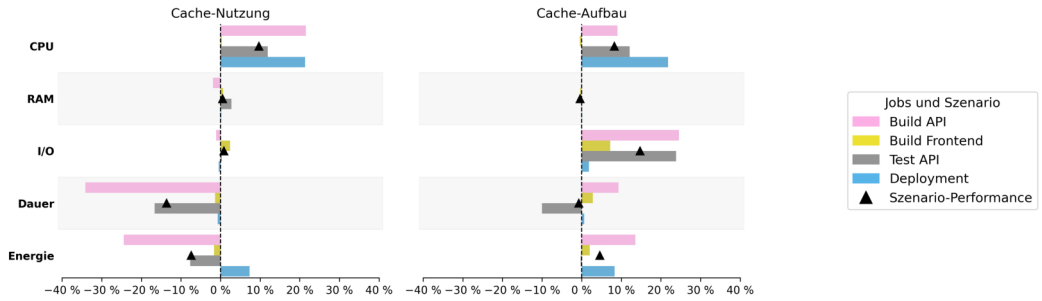

Abbildung 5.3: Rel. mittlere Abweichung der Ressourcen vom Basis-Szenario durch Cache-Nutzung (links) und -Aufbau (rechts). Schwarze Dreiecke: rel. mittlere Abweichung des jew. Cachings-Szenario vom Basis-Szenario

Die insgesamt gesteigerte Energienutzung ist vor allem auf die um 13,53 % höhere Energienutzung des Build-API-Jobs sowie teilweise auf die um 8,34 % gesteigerte Energienutzung des Deployments zurückzuführen. Während ersteres aufgrund des Mehraufwands durch den Cache-Aufbau erwartbar ist, verwundert letzteres, da das Design des Deployment-Jobs über alle Szenarien nicht verändert wird, und bedarf daher im Folgenden einer erweiterten Analyse.

Ein Blick in die Messungen des Deployment-Jobs im Cache-Aufbau-Szenario zeigt, dass dessen mittlere CPU-Auslastung im Vergleich zum Deployment im Basis-Szenario um mehr als ein Fünftel gestiegen ist. Gleichzeitig weichen RAM-Auslastung, die Menge der I/O-Operationen und die Dauer nur unwesentlich und nicht-signifikant vom Basis-Szenario ab. Doch auch die weiteren statistischen Kennzahlen wie Minimum, Maximum und das erste bis dritte Quartil weisen bzgl. der CPU-Auslastung einen Zuwachs um 18,44 % bis 25,98 % auf. Absolut entspricht dies jedoch lediglich einer Zunahme von 3,51 bis 4,06 Prozentpunkten. Aus dem zeitlichen Vergleich in *Abb. 5.2* tritt zudem hervor, dass dieser Zuwachs primär auf die Prozesse während der Job-Phase im Zeitabschnitt von Sekunde 140 bis 158 zurückzuführen ist: Während die CPU-Auslastung im Basis-Szenario hier bis auf ihr globales Minimum sinkt, bleibt sie im Cache-Aufbau-Szenario mehr oder weniger auf demselben Niveau. Allerdings ist an der Häufung der Linien von Einzel-Messungen im Basis-Szenario erkennbar, dass ein ähnlicher Verlauf wie im Cache-Aufbau-Szenario hier ebenfalls mehrfach vorkommt.

Aus den geschilderten Gesichtspunkten wird die stärkere Abweichung der CPU-Auslastung als normale Schwankung deklariert. Ihr Einfluss auf die Gesamtenergienutzung der Pipeline ist vernachlässigbar, was u. a. daran zu erkennen ist, dass der Anteil des Deployment-Jobs an der Gesamtenergie der Pipeline im Cache-Aufbau-Szenario lediglich um 0,5 Prozentpunkte (+3,07 %) im Vergleich zum Basis-Szenario zugenommen hat (Cache-Aufbau:

16,78 %, Basis: 16,28 %). Zusätzlich fällt diese Zunahme im Deployment- im Vergleich zum Build-API-Job mit 0,019 Wh gegenüber 0,041 Wh um über die Hälfte (-53,66 %) geringer aus, was diesbezüglich den dominierenden Einfluss des Build-API-Jobs unterstreicht.

Oberflächlich betrachtet ähnelt die zeitliche Entwicklung der Ressourcenmessungen im Cache-Aufbau-Szenario stark derer des Basis-Szenarios (*Abb. 5.2*). Insbesondere der Verlauf der Leistungsaufnahme zeichnet den bekannten Verlauf der CPU-Auslastung nach. Dennoch sind deutliche Unterschiede zu verzeichnen, die bei der genaueren Betrachtung ins Auge fallen. Zur besseren Orientierung im Folgenden, seien an dieser Stelle die zeitlichen Abschnitte der Jobs genannt: Build-API (Sekunde 0–38), Build-Frontend (Sekunde 39–69), Test-API (Sekunde 70–120) und Deployment (Sekunde 121–160). Alle Zeitbereiche sind gemittelt und auf volle Sekunden gerundet.
Im Vergleich zur Referenzpipeline ist die erste und die letzte der drei markanten CPU-Spitzen, die sich der zeitlichen Reihenfolge nach dem Build und dem Testen der API zuordnen lassen, um 6,8 % (Sekunde 24) bzw. 10,7 % (Sekunde 94) höher angesiedelt, allerdings im Falle des Builds um ca. zwei Sekunden nach hinten verschoben und im Falle des Test-Jobs um ca. 5 Sekunden vorgezogen. Die prägnanten RAM-Spitzen treten ebenfalls leicht zeitlich versetzt auf (Build-API: +1 Sekunde, Build-Frontend: +3 Sekunden, Test-API: -3 Sekunden), unterscheiden sich jedoch nicht wesentlich in ihren Maxima von denen des Basis-Szenarios.

Die Reduktion der Laufzeit im Test-Job im Gegensatz zur Zunahme in den anderen Jobs, die hier erneut hervorsticht, wird noch eindrücklicher, wenn lediglich die Job-Phase losgelöst von der Pre- und der Post-Job-Phase des Test-Jobs betrachtet wird: Diese fällt im Cache-Aufbau- im Schnitt 9,8 Sekunden kürzer aus als im Basis-Szenario, was einer Reduzierung der Laufzeit der Job-Phase um über ein Viertel entspricht. Dennoch lohnt sich der Blick auf die Pre- und Post-Job-Phasen, da diese 35,78 % (17,75 Sekunden) der Laufzeit des Test-Jobs im Cache-Aufbau-Szenario ausmachen, im Basis-Szenario hingegen 24,42 % (13,46 Sekunden). Somit dauert die Vor- und Nachbereitung im Schnitt 4,29 Sekunden länger (Pre-Job: +1,73 Sekunden, Post-Job: +2,57 Sekunden), während der eigentliche Job fast zehn Sekunden weniger Zeit benötigt.

In Bezug auf die I/O-Last gibt es eine zusätzliche I/O-Spitze im API-Build (Sekunde 35) sowie eine im Vergleich zur Referenzpipeline deutlich ausgeprägtere I/O-Spitze im Test-Job (Sekunde 118), jeweils in der Post-Job-Phase. Etwas weniger offensichtlich ist die zeitliche Streckung der I/O-Lastspitze im Übergang vom Frontend-Build (Sekunde 65)

zum Testen der API um ca. zwei Sekunden. Alle diese Veränderungen sind der zusätzlichen I/O-Last durch das Speichern der Caches zuzuordnen.
Andersherum ist die um 73,2 % gesteigerte I/O-Lastspitze in der Pre-Job-Phase des Test-Jobs (Sekunde 78) dem Laden des durch das Build der API bereitgestellten Caches zuzuordnen. Während im Basis-Szenario auf die genannte I/O-Spitze kurz darauf ein lokales CPU-Maximum (Sekunde 77) folgt und die CPU-Auslastung daraufhin für ca. 3 Sekunden zunächst bis auf ein lokales Minimum abfällt, ist im Cache-Aufbau-Szenario der Anstieg der CPU-Auslastung lediglich kurzzeitig gehemmt und fällt bis zum Erreichen des globalen Maximums nicht mehr ab. Die Abnahme der CPU-Auslastung im Basis-Szenario einerseits und ihr Wegfall im Cache-Aufbau-Szenario andererseits ist auf die erhöhte Netzwerklast im Basis-Szenario durch das Herunterladen der Dependencies für den Test-Job zurückzuführen. Indem im Cache-Aufbau-Szenario dem Test-Job bereits die kompletten Build-Dependencies über den Cache zur Verfügung gestellt werden, müssen hier ausschließlich Test-Dependencies heruntergeladen werden. Letzteres ist der Auslöser für die leichte Hemmung des CPU-Anstiegs.
Eine weitere Auffälligkeit im Vergleich der Test-Jobs ist das Ausbleiben der I/O-Schwankungen während der zentralen CPU-Belastungsphase (ca. Sekunde 80–110). Wie in Kapitel 5.1 erläutert, sind diese Schwankungen auf das mehrfache Neustarten des Spring-`ApplicationContext` zurückzuführen.

5.2.2 Cache-Nutzung vs. Cache-Aufbau

Das verwendete Experiment-Design, durch das auf einen Pipeline-Durchlauf mit Cache-Aufbau immer ein Pipeline-Durchlauf mit der dazugehörigen Cache-Nutzung folgt, ermöglicht den direkten Vergleich der entsprechenden Cache-Aufbau- und Cache-Nutzungs-Paare. Aus diesen Vergleichen geht hervor, dass in sämtlichen Fällen die Cache-Nutzung eine geringere I/O-Last, eine reduzierte Laufzeit und einen niedrigeren Energiebedarf hat. Konträr ist sowohl die RAM- als auch die CPU-Auslastung in der überwiegenden Anzahl der Vergleiche erhöht gegenüber dem Cache-Aufbau-Szenario (RAM: 97,83 %, CPU: 71,74 %).
Der Gesamtvergleich aller Cache-Aufbau- und Cache-Nutzung-Messungen miteinander führt zu einer noch stärkeren Aussage, da die maximale Laufzeit einer Cache-Nutzungs-Pipeline (2:27 Minuten) sieben Sekunden kürzer ist als die minimale Laufzeit im Cache-Aufbau-Szenario (2:34 Minuten), womit im hiesigen Experiment jede Pipeline im Cache-Nutzungs-Szenario schneller war als eine beliebige Pipeline des Cache-Aufbau-Szena-

rios (*Tab. A.1*). Analog unterschreitet der maximale Energiebedarf einer Pipeline im Cache-Nutzungs-Szenario (1,42 Wh) den minimalen Bedarf einer Pipeline im Cache-Aufbau-Szenario (1,43 Wh) um 0,01 Wh.

Eine Betrachtung der mittleren Performance-Werte beider Szenarien unterstreicht die genannten Funde. Die durchschnittliche Laufzeit einer Pipeline sinkt bei Cache-Nutzung um knapp 20 Sekunden von 2:40 auf 2:20 Minuten, eine Reduktion um 12,59 %. Besonders deutlich tritt dieser Effekt beim Build der API hervor, dessen Gesamtlaufzeit sich um fast 40 % (-15,26 Sekunden) und dessen reine Job-Laufzeit sich von 20,61 Sekunden (Cache-Aufbau) auf 6,92 Sekunden um fast zwei Drittel verkürzt. Gleichzeitig reduziert sich die mittlere Laufzeit der Pre-Job- und Post-Job-Phase um knapp 1,62 Sekunden.
Deutlich geringer ausgeprägt und ebenfalls zu Gunsten des Cache-Nutzungs-Szenarios, ist bei den weiteren drei Jobs eine Reduzierung der Gesamtlaufzeit zu verzeichnen (Frontend-Build: -4 %, API-Test: -7,36 %, Deployment: 1,38 %). Die absolute Laufzeit der Job-Phase des Test-Jobs bleibt im Vergleich nahezu konstant (-0,8 Sekunden), sodass die beobachtete Verkürzung um insgesamt 3,65 Sekunden auf eine effizientere Pre-Job- und Post-Job-Phase zurückzuführen ist. Die zeitliche Verkürzung des Deployments ist zwar signifikant, beläuft sich absolut jedoch auf lediglich knapp eine halbe Sekunde.

Bezüglich der I/O-Last kommt es im Mittel zu einer Verringerung um 239,78 MiB (-12,13 %) im Vergleich des Cache-Nutzungs- gegenüber dem Cache-Aufbau-Szenario. In erster Linie ist dies gekoppelt an die starken Einsparungen im Build-API- (-101,53 MiB) und im Test-API-Job (-100,18 MiB) um jeweils ca. ein Fünftel. Zusätzlich lässt sich ein moderater Beitrag zur Reduzierung der Gesamt-I/O-Last um 4,46 % (-27,46 MiB) im Frontend-Build feststellen. Die Veränderung der I/O-Last während des Deployments ist statistisch nicht-signifikant und verbleibt im Bereich zufälliger Schwankungen.
Bei der RAM-Nutzung lässt sich trotz einer signifikanten Zunahme um 0,91 % absolut lediglich ein Anstieg um 0,16 Prozentpunkte ausmachen, die sich begründet in einer Reduktion im Build-API-Job (-1,8 %) und einer Zunahme sowohl im Build des Frontends (+1,02 %) als auch im Test-API-Job (+2,59 %). Die RAM-Nutzung des Deployment-Jobs bleibt nahezu unverändert.
Ein vergleichbares Verhalten zeigt die CPU-Auslastung, die im Mittel um 1,34 % zunimmt im Vergleich zum Cache-Aufbau-Szenario (von 44,11 % auf 44,7 %). Dieses Plus ist vor allem der Zunahme der CPU-Auslastung im Build-API-Job zuzuschreiben (+11,47 %), wohingegen die CPU-Auslastung in den anderen Jobs nur marginal und nicht-signifikant abweicht.
Ähnlich fällt auch der Energiebedarf der Pipeline im Cache-Nutzungs-Szenario um 11,43 %

von 1,52 Wh auf 1,34 Wh und spiegelt damit die starke Reduktion im Build-API-Job um 33,44 % wider. Der Rückgang im Deployment-Job bewegt sich im nicht-signifikanten Bereich.

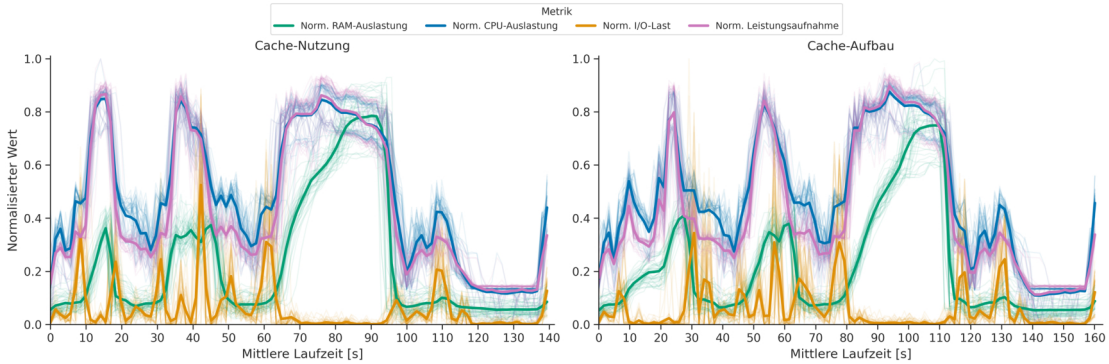

Abbildung 5.4: Vergleich des zeitlichen Verlaufs des Cache-Nutzungs- (links) und des Cache-Aufbau-Szenarios (rechts) anhand der zentralen Messwerte. Messwerte wurden für den Vergleich normiert. Zeitwerte geben die mittlere Laufzeit an. Hervorgehobene Linien: mittlerer Verlauf der jeweiligen Ressource.

Wie schon zuvor für die Referenz-Pipeline und das Cache-Aufbau-Szenario wird nachfolgend der zeitliche Verlauf der Ressourcenmessungen in den beiden Caching-Szenarien hinsichtlich charakteristischer Merkmale und Unterschiede analysiert (*Abb. 5.4*). Die Start- und Endzeiten der Jobs im Cache-Nutzungs-Szenario lassen sich wie folgt den Sekundenwerten der mittleren Laufzeit zuordnen (Zeitspanne der jeweiligen Job-Phase in Klammern): API-Build von Sekunde 0–23 (10–17), Frontend-Build von Sekunde 24–52 (33–47), API-Test von Sekunde 53–99 (65–96) sowie Deployment von Sekunde 100–139 (110–137).

Grundlegend bestätigen die zeitlichen Verläufe das aus dem Cache-Aufbau-Szenario bekannte Muster mit den drei markanten Höhepunkten der CPU-Auslastung und Leistungsaufnahme, wobei die Leistungsaufnahme wiederum stark dem Verlauf der CPU-Auslastung folgt. Gleichermaßen lassen sich die Peaks den beiden Build-Jobs und dem Test-Job zuordnen. Trotz dieser oberflächlichen Gemeinsamkeiten sind einige Unterschiede im zeitlichen Verlauf auszumachen, die an dieser Stelle in chronologischer Reihenfolge der Jobs betrachtet werden sollen.
Im Cache-Nutzungs-Szenario steigt die CPU-Auslastung nach ca. fünf Sekunden mehr oder weniger kontinuierlich bis auf ihr Maximum (Sekunde 15) während des Builds der API an. Hingegen steigt die CPU-Auslastung im Cache-Aufbau-Szenario zunächst auf ein lokales Maximum (Sekunde 10) und fällt dann innerhalb von fünf Sekunden auf ein

lokales Minimum. Erst dann steigt sie wie im Cache-Nutzungs-Szenario auf das Maximum der CPU-Auslastung während des API-Builds. Dieser lokale Peak bleibt im Cache-Nutzungs-Szenario folglich gänzlich aus. Weiter fällt auf, dass der Peak der I/O-Last in der Pre-Job-Phase (Sekunde 7) des Build-Jobs um 150 % höher ist, während der am Ende der Job-Phase (Sekunde 17) um 31 % kleiner ausfällt. Der im Cache-Aufbau-Szenario darauffolgende I/O-Höhepunkt in der Post-Job-Phase des API-Builds (Sekunde: 34) ist im Cache-Nutzungs-Szenario nicht auszumachen. Zudem zeigt sich im Cache-Nutzungs-Szenario ein deutlich schnellerer Anstieg der RAM-Auslastung, die das etwa 11 % niedrigere Maximum ca. 13 Sekunden vorher erreicht (Sekunde 15).
Auch während des Frontend-Build-Jobs treten auffällige Unterschiede im zeitlichen Verlauf der Ressourcenmessungen zwischen den beiden Caching-Szenarien zutage. In der Pre-Job-Phase zeigt sich ein um ca. 72 % erhöhter Höhepunkt der I/O-Last (Sekunde 31) und zum Ende der Job-Phase hin ein weiterer, um 69 % erhöhter I/O-Höhepunkt (Sekunde 42). Parallel dazu verlängert sich die Dauer der erhöhten RAM-Nutzung um ca. drei Sekunden (Sekunde 35–45). Die I/O-Last in der Post-Job-Phase verringert sich ebenfalls, erkennbar an dem ca. ein bis zwei Sekunden verkürzten Ausschlag in der I/O-Last (Sekunde 45–52).
Während des Test-API-Jobs verändert sich die I/O-Last in der Pre-Job- und in der Job-Phase nicht merklich. Dafür tritt der I/O-Höhepunkt in der Post-Job-Phase jedoch um ca. 50 % reduziert auf. Außerdem sinkt das Maximum der CPU-Auslastung (Sekunde 76) um ca. 4 %, während das der RAM-Auslastung (Sekunde 91) sich um ca. 5 % erhöht.
Das Deployment schließlich weist kaum markante Veränderungen auf. Lediglich der I/O-Peak innerhalb der Job-Phase (Sekunde 109) geht im Cache-Nutzungs-Szenario um knapp 14 % zurück.

5.2.3 Cache-Nutzung vs. Basis

Indem in den vorangegangenen zwei Unterkapiteln zunächst das Cache-Aufbau- mit dem Basis- und dann mit dem Cache-Nutzungs-Szenario verglichen wurde, ergeben sich bereits transitiv einige Erkenntnisse über das Verhältnis des Cache-Nutzungs- zum Basis-Szenario. Dennoch soll im Folgenden – wenn auch nicht in derselben Breite wie in den vorangegangenen Kapiteln – analysierte werden, inwiefern sich die Messungen der beiden Szenarien voneinander unterscheiden.

Die Analyse des direkten Vergleichs zwischen dem Cache-Nutzungs- und dem Basis-Szenario offenbart einige deutliche Effizienzsteigerungen zu Gunsten des Ersteren. Im Mittel

verkürzt sich die Laufzeit der Pipeline um 13,6 %, wobei sämtliche Cache-Nutzungs-Durchläufe eher abgeschlossen werden als der langsamste Durchlauf im Basis-Szenario: Die maximale Laufzeit im Cache-Nutzungs-Szenario beträgt 2:27 Minuten und bleibt damit fünf Sekunden unter dem Minimum der Referenzpipeline (*Tab. A.1*). Während die Cache-Nutzung gegenüber dem Cache-Aufbau stets mit einer niedrigeren Energienutzung verbunden war, kann dies im Vergleich zum Basis-Szenario nicht bestätigt werden, da der minimale Energieverbrauch der Referenzpipeline 0,06 Wh unterhalb des Maximums des Cache-Nutzungs-Szenarios liegt.

Ein Blick auf die Laufzeiten je Job unterstreicht die soeben beschriebenen zeitlichen Einsparungen (*Abb. 5.3*). Im Build-API-Job reduziert sich die Gesamtlaufzeit um 34,12 % (12 Sekunden), wobei die Dauer der Job-Phase um nahezu zwei Drittel geringer ausfällt (-13,37 Sekunden) und somit eine ähnliche Reduktion aufweist wie im Vergleich der beiden Caching-Szenarien. Die Pre-Job-Phase verlängert sich stattdessen um 17,23 % und die Post-Job-Phase verkürzt sich um 3,31 %. Beim Frontend-Build nimmt die Laufzeit zwar ebenfalls ab, jedoch nur sehr geringfügig (-1,36 %) und nicht-signifikant. Dennoch nimmt die Laufzeit der Job-Phase um 7,06 % ab. Dass sich dies nicht auf die Gesamtlaufzeit niederschlägt, liegt an der Verlängerung der Pre- (5,16 %) und Post-Job-Phase (6,17 %). Beim Test-API-Job verkürzt sich die mittlere Gesamtlaufzeit um 16,62 % (9,16 Sekunden), was vor allem aus einer Reduktion der Job-Phase von knapp einem Viertel (-10,6 Sekunden) hervorgeht. Einer verlängerten Pre-Job-Phase (+15,09 %) steht eine mäßige Verkürzung der Post-Job-Phase (-5,59 %) gegenüber. Für das Deployment lassen sich abgesehen von minimalen Verschiebungen keine signifikanten Einflüsse beobachten.

Die weiteren Ressourcen wie CPU-, RAM-Auslastung und I/O-Last treten im Mittel nur bedingt bis gar nicht verändert in Erscheinung. So ist bzgl. der I/O-Last eine sehr geringe, statistisch nicht signifikante Zunahme um 0,84 % (14,52 MiB) festzustellen. Auch auf der Ebene der Einzeljobs dominieren sehr kleine Veränderungen das Bild und lediglich das Frontend-Build weist eine minimale, statistisch signifikante Zunahme der I/O-Aktivität um 2,45 % auf.
Ähnlich der I/O-Last bleibt das RAM-Niveau insgesamt nahezu identisch (+0,55 %, nicht signifikant), allerdings ist auf Job-Ebene lediglich die Unterscheidung im Deployment nicht signifikant.
Eine merkbare und signifikante Veränderung verzeichnet hingegen die CPU-Auslastung, die im Mittel um 9,71 % gegenüber dem Basis-Szenario zugenommen hat. Dieser Zuwachs erklärt sich, in Teilen analog zur früheren Beobachtung im Vergleich des Cache-Aufbau- mit dem Basis-Szenario (*Kap. 5.2.1*), durch einen Zuwachs der CPU-Auslastung um mehr

als ein Fünftel im Build-API-Job, einer merkbaren Zunahme im Test-Job (+11,93 %) sowie durch das Ausbleiben des verstärkten CPU-Rückgangs im Deployment-Job.
Die mittlere Energienutzung der Pipeline nimmt trotz der gesteigerten CPU-Auslastung um 7,38 %ab. Dies ist im Wesentlichen auf die starke Senkung im Build-API- um knapp ein Viertel (-0,07 Wh) sowie um 7,64 % im Test-Job (-0,05 Wh) zurückzuführen. Eine leichte Zunahme um 7,33 % (0,02 Wh) im Deployment wirkt dem teilweise entgegen.

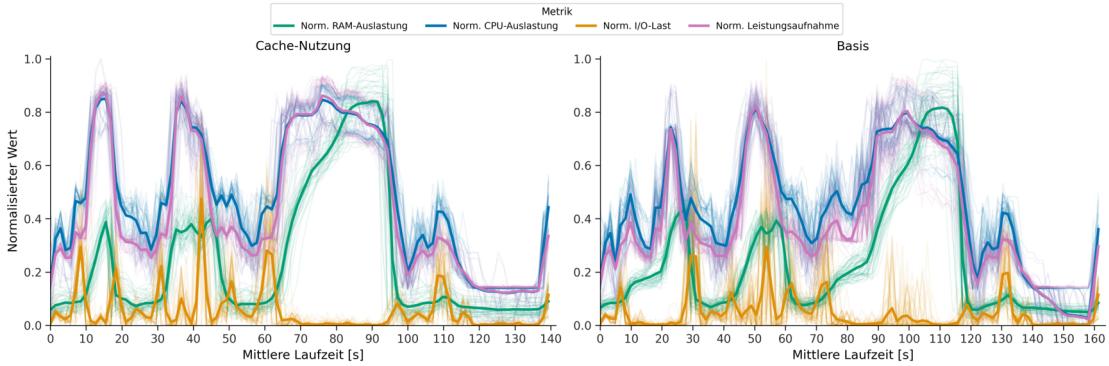

Abbildung 5.5: Vergleich des zeitlichen Verlaufs des Cache-Nutzungs- (links) und des Basis-Szenarios (rechts) anhand der zentralen Messwerte. Messwerte wurden für den Vergleich normiert. Zeitwerte geben die mittlere Laufzeit an. Hervorgehobene Linien: mittlerer Verlauf der jeweiligen Ressource.

Wie zuvor sollen die herausgearbeiteten Beobachtungen auf Basis der mittleren Performance der Pipelines im folgenden Abschnitt mit zusätzlichen Beobachtungen aus der vergleichenden Analyse des zeitlichen Verlaufs der Szenarien untermauert werden *Abb. 5.5*. Die Zeitabschnitte der Jobs im Cache-Nutzungs-Szenario können dem entsprechenden Abschnitt im Kapitel 5.2.2 entnommen werden.
Im Build-API ist ein um 110 % gesteigerter I/O-Höhepunkt in der Pre-Job-Phase (Sekunde 8) zu beobachten. Wie bereits im Vergleich zum Cache-Aufbau beschrieben, fällt der lokale CPU-Höhepunkt danach aus, stattdessen verläuft der Anstieg zum CPU-Maximum (Sekunde 15) des Build-API-Jobs mehr oder minder direkt. Das CPU-Maximum ist zudem um 15 % höher. Der Anstieg zum nahezu parallel auftretenden lokalen RAM-Maximum erfolgt ebenfalls direkter, jedoch ist das Maximum um ca. 10 % niedriger. Am Ende des Build-API-Jobs, in der Post-Job-Phase, fällt der I/O-Peak (Sekunde 18) um rund 19 % geringer aus.
Beim Frontend-Build liegt der I/O-Peak (Sekunde 31) in der Pre-Job-Phase um ca. 82 % höher. Ebenso ist der zentrale I/O-Höhepunkt (Sekunde 42) in der Job-Phase um 62 % erhöht. Hingegen bleibt I/O-Last in der Post-Job-Phase mehr oder weniger unverändert.

Das zentrale RAM-Hoch (Sekunde 35–45) dauert etwa drei Sekunden länger, verbleibt jedoch auf demselben Level.
Im Test-API-Job ist ein um drei Viertel gesteigerter I/O-Peak in der Pre-Job-Phase (Sekunde 60) zu beobachten. Ebenso wie beim Build-API-Job entfällt hier zu Beginn der Job-Phase der lokale CPU-Höhepunkt, der im Basis-Szenarios bei Sekunde 77 auftritt. Die im Basis-Szenario parallel zum Maximum der CPU-Auslastung auftretenden I/O-Schwankungen (Sekunde 90–100) entfallen ebenfalls. Dahingegen ist das Maximum um etwa 7 % leicht gesteigert. Keine Veränderungen weist der I/O-Höhepunkt in der Post-Job-Phase des Cache-Nutzungs-Szenarios auf (Sekunde 97).
Das Deployment verläuft bis auf die bereits erwähnte phasenweise Stagnation der CPU-Auslastung im Zeitraum zwischen Sekunde 121 und 137 mehr oder weniger identisch zum Basis-Szenario.

5.3 H2: Parallelisierung

In diesem Kapitel werden die Auswirkungen der aktivierten Parallelisierung im Parallel-Szenario systematisch untersucht und mit der sequentiellen Ausführung im Basis-Szenario verglichen. Ergänzend zur globalen Analyse werden die Abläufe auf Ebene einzelner Jobs betrachtet, mit einem Fokus auf parallel ablaufende Jobs.

Die Aktivierung der Parallelisierung führt zu einer Reduzierung der mittleren Gesamtlaufzeit im Vergleich zum Basis-Szenario um 16,84 % auf knapp 2:14 Minuten (*Tab. A.1*). Dies entspricht einer absoluten Abnahme von ca. 27 Sekunden. Darüber hinaus weisen die Laufzeiten der einzelnen Pipeline-Durchläufe im Szenario-Vergleich eine geringere Variabilität auf, was u. a. an der um neun Sekunden geringeren Distanz der Extremwerte sowie der um eine Sekunde geringeren Standardabweichung zu erkennen ist.
Folglich nicht ganz unerwartet, liegt das Maximum der Gesamtlaufzeit im Parallel-Szenario mit 2:19 Minuten 13 Sekunden (-8,55 %) unterhalb des Minimums im Basis-Szenario, womit alle Pipeline-Durchläufe schneller waren als im Basis-Szenario. Weniger extrem entspricht das Maximum der Energienutzung (1,36 Wh) dem Minimum des Basis-Szenarios, sodass alle Pipeline-Durchläufe maximal so viel Energie benötigt haben, wie die sparsamste Pipeline im Basis-Szenario. Im Mittel ist eine Abnahme der Energienutzung um 13,4 % (-0,19 Wh) zu verzeichnen.

Die weiteren Ressourcenwerte verzeichnen durchschnittlich eine geringfügige Zunahme: Die CPU-Auslastung steigt um 4,57 %, was allerdings nur 1,86 Prozentpunkten ent-

spricht. Die RAM-Auslastung steigt um 1,99 % bzw. 0,34 Prozentpunkte und die I/O-Last um 5,56 % bzw. 95,79 MiB.
An dieser Stelle sei angemerkt, dass alle Werte der RAM-Auslastung im Parallel-Szenario um 1,2 Prozentpunkte angehoben wurden, da im Vergleich zum Basis-Szenario und auch zu den anderen beiden Szenarien mit aktivierter Parallelisierung ein entsprechend niedrigeres Gesamtniveau der RAM-Auslastung festgestellt wurde, das sich folglich nicht durch Besonderheiten der Parallelisierung erklären lässt und somit auf externe Einwirkungen zurückzuführen ist.

Aus der Analyse geht hervor, dass die beiden Build-Jobs in 100 % der Fälle parallel ablaufen. Der Test-Job läuft hingegen nie parallel zum Build-Frontend-Job ab. Ebenso läuft das Deployment per Design nie parallel zu den anderen Jobs ab.
Ein Blick auf die Laufzeiten je Job zeigt, dass der Build-API-Job mit 38,07 Sekunden eine um 8,34 % längere Laufzeit (+2,93 Sekunden) aufweist. Dies ist primär durch eine 2,26 Sekunden längere Job-Phase (+11,13 %) bedingt, die in etwa drei Viertel der Laufzeitverlängerung ausmacht, während die Pre-Job-Phase sowie die Post-Job-Phase um maximal eine halbe Sekunde länger andauern. Gleichermaßen weist der Build-Frontend-Job mit 30,37 Sekunden eine um 6,23 % längere Laufzeit (+1,78 Sekunden) auf, die sich fast gänzlich (zu 92,2 %) auf die verlängerte Job-Phase beläuft. Zusammengenommen ergibt sich, dass die beiden Build-Jobs für die gesamte Laufzeit des Frontend-Build-Jobs parallel ablaufen. Der Build-API-Job läuft nach der parallelen Phase durchschnittlich zusätzliche 7,7 Sekunden sequentiell ab.
Zwar zeigt sich absolut gesehen beim Test-API-Job eine signifikante Reduktion der Laufzeit in ähnlicher Größenordnung der Zunahme beim Frontend (-1,31 Sekunden), relativ betrachtet ist dies jedoch lediglich ein Minus von 1,21 %. Da der Test-Job in diesem Szenario nicht parallel abläuft, wird diese Abweichung vom Basis-Szenario den erwartbaren Schwankungen zugeordnet. Die Laufzeit des Deployment-Jobs bleibt nahezu unverändert bzw. jegliche Änderung ist nicht-signifikant.

Die Ressourcennutzung parallel ausgeführter Jobs lässt sich aufgrund des gewählten Experimentdesigns nicht auf Ebene der einzelnen Jobs vergleichen, da die gemessenen Werte die Ressourcenbeanspruchung des Host-Systems abbilden und nicht die spezifische Nutzung der einzelnen Docker-Hosts. Daher wird an dieser Stelle der gesamte Zeitraum der parallelen Ausführung mit der sequentiellen Ausführung derselben Programmsequenzen im Basis-Szenario verglichen: Während der parallelen Ausführung der Build-Jobs ist eine insgesamt höhere Ressourcennutzung über alle Metriken hinweg zu beobachten. Dies zeigt sich in den aggregierten CPU-Werten der parallelen Sequenz, die durchgängig über

denen der sequentiellen Durchführung liegen. Die mittlere CPU-Auslastung ist um rund die Hälfte gesteigert (+50,23 %, +22,12 Prozentpunkte), während die RAM-Auslastung um 7,83 % (+1,31 Prozentpunkte) zunimmt. Gleichzeitig sinkt die mittlere Energienutzung trotz erhöhter Ressourcenauslastung um 26,24 % (-0,14 Wh), was auf die deutlich reduzierte Laufzeit (-46,91 %) der Sequenz durch die parallele Ausführung zurückzuführen ist.
Auffällig ist außerdem, dass die Maxima der CPU- und RAM-Auslastung im Basis-Szenario unterhalb der Minima im Parallel-Szenario liegen, während im Gegenzug die Energienutzung in der Parallelisierung permanent geringer ausfällt. Das durchschnittliche Minimum der parallelen Ausführung liegt dabei 17,31 % unterhalb des durchschnittlichen Maximums im Basis-Szenario. Die I/O-Last verändert sich hingegen nicht wesentlich und bleibt ohne signifikante Unterschiede.
Bezüglich der sequentiell ablaufenden Jobs ist eine Reduktion der mittleren CPU-Auslastung zu beobachten (Test-API: -8,51 %, Deployment: -13 %). Ebenso fällt beide Male der Energiebedarf geringer aus (Test-API: -9 %, Deployment: -4,68 %). Die I/O-Last nimmt zwar jeweils zu, allerdings absolut um maximal 20 MiB.

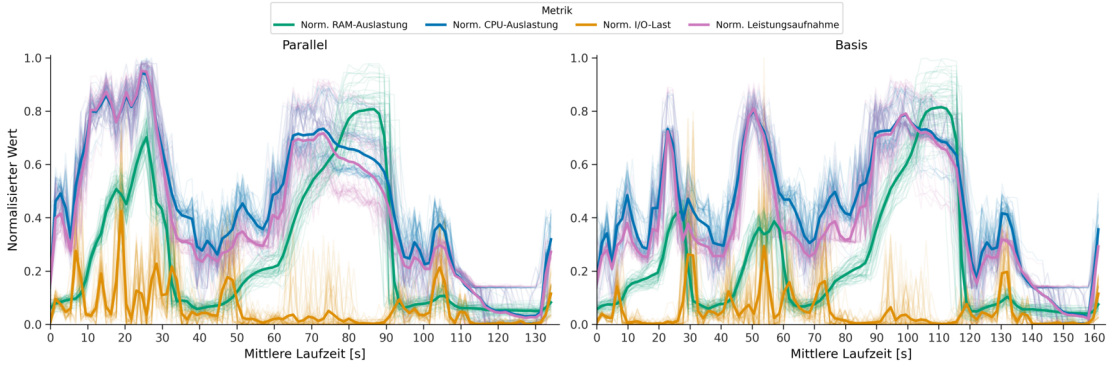

Abbildung 5.6: Vergleich des zeitlichen Verlaufs des Parallel- (links) und des Basis-Szenarios (rechts) anhand der zentralen Messwerte. Messwerte wurden für den Vergleich normiert. Zeitwerte geben die mittlere Laufzeit an. Hervorgehobene Linien: mittlerer Verlauf der jeweiligen Ressource.

Im zeitlichen Verlauf des Parallel-Szenarios (*Abb. 5.6*) sind im Gegensatz zum Basis-Szenario nur zwei und nicht drei zentrale Höhepunkte in der CPU-Auslastung zu vernehmen (Sekunde: 25, 72), da die Höhepunkte der beiden Build-Jobs durch die Parallelisierung zu einem gemeinsamen zusammenfallen. Wie gehabt, folgt der Verlauf der Leistungsaufnahme in starkem Ausmaß dem der CPU-Auslastung. Die Jobs im Parallel-Szenario lassen sich den folgenden Zeitmarken (auf ganze Sekunden gerundet) zuordnen,

wobei der zeitliche Abschnitt der Job-Phasen jeweils in Klammern folgt: Build-Frontend von Sekunde 0 bis 30 (9–27), Build-API von Sekunde 0 bis 38 (10–32), Test-API von Sekunde 39 bis 93 (48–90) und Deployment von Sekunde 94 bis 134 (104–132).
Die CPU-Auslastung und die I/O-Last nehmen in der Pre-Job-Phase (Sekunde 0–10) einen ähnlichen Verlauf wie bei den einzelnen Build-Jobs im Basis-Szenario an, jedoch im Vergleich zu den Einzeljobs stark erhöht. Verglichen mit den aufsummierten Werten der jeweiligen Höhepunkte in den beiden Build-Jobs, bewegen sich die Veränderungen der zwei I/O-Höhepunkte in der Pre-Job-Phase im niedrigen, einstelligen Prozentbereich. Da in der Job-Phase des API-Builds im Basis-Szenario nahezu keine I/O-Aktivitäten stattfinden, stechen bei paralleler Ausführung zum Frontend-Build dessen starken I/O-Schwankungen in der Job-Phase hervor, die z. T. deutlich erhöht sind (Sekunde 10–25). Genauso lassen sich die beiden I/O-Höhepunkte am Ende der parallelen Ausführung (Sekunde 28, 33) ihrer Reihenfolge nach den Post-Job-Phasen des Frontend- und des API-Build-Jobs zuordnen.
Im Anschluss an die I/O-Betrachtung lassen sich auch in der RAM-Auslastung Zuordnungen zwischen dem Basis- und dem Parallel-Szenario herstellen. Die erste Spitze der RAM-Auslastung in der Job-Phase des Parallel-Szenarios (Sekunde 18) fällt zusammen mit dem Beginn des RAM-Hochs des Frontend-Builds im Basis-Szenario (Sekunde 52). Weiter vereint die zweite Spitze (Sekunde 25) das RAM-Hoch des Frontend-Builds mit dem Höhepunkt der RAM-Auslastung des API-Builds, das im Basis-Szenario zum Vorschein tritt (Sekunde 25). Ähnlich werden die CPU-Spitzen der einzelnen Build-Jobs in einem längeren CPU-Hoch (Sekunde 11-25) vereint.
Der sequentielle Verlauf ab Sekunde 39 gleicht dem des Basis-Szenarios sehr, mit nur kleineren Ausnahmen. So ist die CPU während des Test-Jobs auf einem insgesamt niedrigeren Niveau ausgelastet, unter anderem daran erkenntlich, dass die beiden lokalen CPU-Maxima während des Test-Jobs (Sekunde 52, 72) jeweils ca. 5 Prozentpunkte unter ihren Pendants im Basis-Szenario liegen. Das Deployment weist einen um 10 % höheren CPU-Peak auf, was absolut ca. 4 Prozentpunkten entspricht.

5.4 H3: Caching und Parallelisierung

Nach der separaten Betrachtung der Caching- und Parallelisierungsstrategien folgt in diesem Kapitel die Analyse ihrer kombinierten Anwendung. Dazu werden die beiden Szenarien mit Parallelisierung und Cache-Aufbau bzw. Cache-Nutzung – im Folgenden vereinfacht „Parallel-Aufbau-Szenario“ und „Parallel-Nutzungs-Szenario“ genannt – mit

den bereits vorgestellten, jeweils enthaltenen Einzelstrategien verglichen. Diese vergleichende Analyse dient der Überprüfung von Hypothese *H3*.

5.4.1 Cache-Aufbau und Parallelisierung

In diesem Kapitel wird das Parallel-Aufbau-Szenario sowohl mit dem Cache-Aufbau-Szenario als auch mit dem Parallel-Szenario verglichen, mit dem Ziel, die kombinierten Effekte der beiden Optimierungsstrategien zu untersuchen.

Mit einer mittleren Gesamtlaufzeit von 2:09 Minuten (*Tab. A.1*) ist das Parallel-Aufbau-Szenario etwa 31 Sekunden schneller als eine durchschnittliche Pipeline im Cache-Aufbau-Szenario (-19,3 %) und knapp 5 Sekunden schneller als eine im Parallel-Szenario (-3,68 %). Nicht unerwartet, da ebenso im Vergleich zwischen dem Parallel- und dem Basis-Szenario herausgearbeitet, wurden alle Pipeline-Durchläufe im Parallel-Aufbau-Szenario schneller abgeschlossen als im Cache-Aufbau-Szenario: Das Maximum von 2:20 Minuten liegt 14 Sekunden unter dem Minimum des Cache-Aufbau-Szenarios.
Auf Pipeline-Ebene zeigt der Vergleich zwischen dem Parallel-Aufbau- und dem Cache-Aufbau-Szenario eine Reduktion der CPU-Auslastung um 1,58 % (-0,7 Prozentpunkte), der I/O-Operationen um 1,55 % (-30,65 MiB) und des Energieverbrauchs um 18,52 % (-0,28 Wh). Lediglich die RAM-Auslastung ist um 1,38 % (+0,24 Prozentpunkte) höher. Im Vergleich zum Parallel-Szenario ist die CPU-Auslastung nicht signifikant um 1,89 % gesteigert. Die RAM-Auslastung erhöht sich um 6,4 % (+1,04 Prozentpunkte) und die I/O-Last um 7,03 % (+127,86 MiB). Der Energieverbrauch ist mit einer Reduktion von 1,62 % (-0,02 Wh) nicht signifikant verringert.

Wie im Parallel-Szenario laufen die beiden Build-Jobs in 100 % der Fälle parallel. Da der Build-API-Job eine längere Laufzeit als der Build-Frontend-Job hat und der Test-Job vom Build-API-Job abhängt, läuft der Test-Job ebenfalls nie parallel zum Frontend-Build-Job.
Bei der Betrachtung der Job-Laufzeiten zeigt sich, dass der Build-API-Job sowohl im Vergleich zum Cache-Aufbau-Szenario (+2,53 %, +0,97 Sekunden) als auch zum Parallel-Szenario (+4,44 %, +1,31 Sekunden) mit einer um ca. eine Sekunde längeren Laufzeit abschneidet. Im Vergleich zum Parallel-Szenario ist die Post-Job-Phase um 2,85 Sekunden (+48,39 %) deutlich verlängert, während die Pre-Job-Phase um 1,06 Sekunden (-11,01 %) verkürzt ist. Gegenüber dem Cache-Aufbau-Szenario ist vor allem eine

Zunahme in der Job- (+7,08 %, +1,46 Sekunden) und eine Abnahme in der Pre-Job-Phase (-8,83 %, -0,83 Sekunden) zu verzeichnen.
Der Build-Frontend-Job weist eine nur unwesentlich veränderte Gesamtlaufzeit im Vergleich zum Parallel-Szenario auf. Ähnlich wie beim Build-API-Job findet im Vergleich zum Parallel-Szenario eine Verlagerung der Phasen-Anteile an der Gesamtlaufzeit von der Pre- (-0,92 Sekunden, -10,02 %) hin zur Post-Job-Phase (+1,54 Sekunden, +41,62 %) statt und im Vergleich zum Cache-Aufbau-Szenario von der Pre-Job- (-10,99 %, -1,02 Sekunden) zur Job-Phase (+9,28 %, +1,43 Sekunden).
Die zeitlichen Veränderungen des Test-API-Jobs im Vergleich zum Parallel- und Cache-Aufbau-Szenario weisen Parallelen zu den Vergleichen der beiden Szenarien mit dem Basis-Szenario auf: Die Gesamtlaufzeit verkürzt sich im Vergleich zum Cache-Aufbau-Szenario relativ betrachtet um 3,75 % (-1,86 Sekunden) und ist damit zu vernachlässigen. Im Vergleich zum Parallel-Szenario ist die Gesamtlaufzeit hingegen um 11,23 % (-6,04 Sekunden) verkürzt. Die hierzu führenden Prozesse sind dieselben wie im Cache-Aufbau-Szenario (*Kap. 5.2.1*) und letztlich treibende Ursache für die insgesamt kürzere Laufzeit der Pipeline im Parallel-Aufbau-Szenario im Vergleich zum Parallel-Szenario.
Die Reduzierung der Laufzeit im Deployment-Job ist sowohl absolut als auch relativ betrachtet geringfügig, weshalb hier festgestellt wird, dass sie im Vergleich zum Parallel- (-1,31 %, -0,51 Sekunden) und zum Cache-Aufbau-Szenario (-2,34 %, -0,92 Sekunden) nahezu unverändert bleibt.

Aufgrund des Experimentdesigns lässt sich die Ressourcenbeanspruchung parallel ablaufender Jobs nicht auf Job-Ebene mit der Beanspruchung in anderen Szenarien vergleichen. An dieser Stelle soll stattdessen jobübergreifend die Ressourcennutzung während paralleler Sequenzen verglichen werden. Der Vergleich mit dem Cache-Aufbau-Szenario erfolgt, indem die sequentielle Ausführung der untersuchten parallelen Phase als Referenz angenommen wird.
Während der ca. 30 Sekunden andauernden parallelen Ausführung der Build-Jobs im Parallel-Aufbau-Szenario liegt die durchschnittliche CPU-Auslastung bei 63,43 % und die RAM-Auslastung bei 17,89 %. Die I/O-Operationen belaufen sich auf 849,08 MiB und der Energieverbrauch auf 0,39 Wh.
Im Vergleich zur sequentiellen Ausführung der entsprechenden Abschnitte im Cache-Aufbau-Szenario ist die RAM- und CPU-Auslastung immer höher, sodass die minimal gemessenen Durchschnittswerte im Parallel-Aufbau-Szenario (RAM: 17,47 %, CPU: 57,57 %) über den maximalen Werten des Cache-Aufbau-Szenarios (RAM: 17,36 %, CPU: 49,0 %) liegen. Die mittlere Auslastung betrachtend, ist die CPU-Auslastung um 37,5 %

und die RAM-Auslastung um 6,85 % höher angesiedelt. Dahingegen ist die I/O-Last leicht um 6,42 % (-61,3 MiB) und der Energieverbrauch deutlich um 35,76 % (-0,22 Wh) gesunken.
Gegenüber der parallelen Phase im Parallel-Szenario zeigt sich eine minimale Reduktion der RAM-Auslastung um 0,92 % (-0,17 Prozentpunkte) sowie der CPU-Auslastung um 4,13 % (-2,73 Prozentpunkte). Letztlich erhöht sich die I/O-Last um 6,78 % (+56,76 MiB), während der Energieverbrauch um 3,83 % (+0,01 Wh) leicht sinkt. Die Länge der parallelen Sequenz hat sich nicht signifikant verändert.

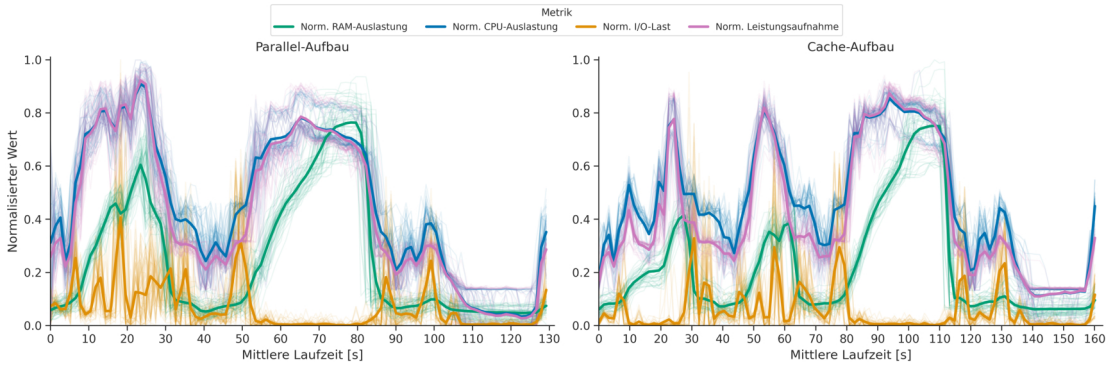

Abbildung 5.7: Vergleich des zeitlichen Verlaufs des Parallel-Aufbau- (links) und des Cache-Aufbau-Szenarios (rechts) anhand der zentralen Messwerte. Messwerte wurden für den Vergleich normiert. Zeitwerte geben die mittlere Laufzeit an. Hervorgehobene Linien: mittlerer Verlauf der jeweiligen Ressource.

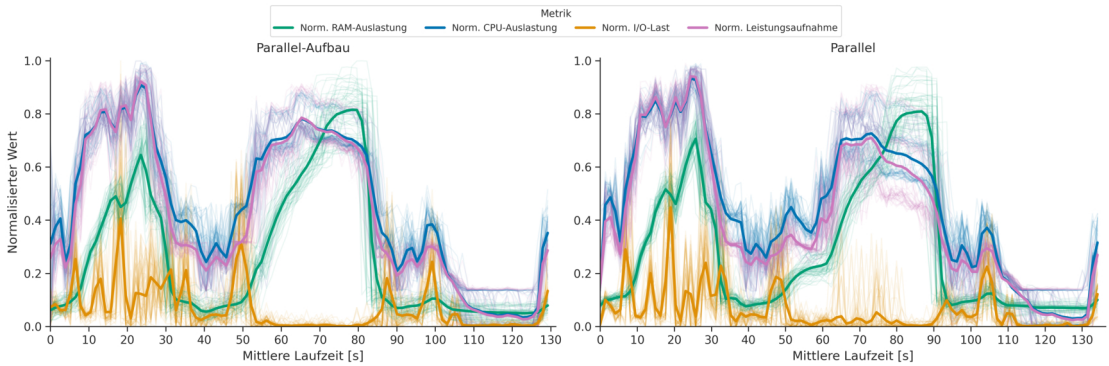

Abbildung 5.8: Vergleich des zeitlichen Verlaufs des Parallel-Aufbau- (links) und des Parallel-Szenarios (rechts) anhand der zentralen Messwerte. Messwerte wurden für den Vergleich normiert. Zeitwerte geben die mittlere Laufzeit an. Hervorgehobene Linien: mittlerer Verlauf der jeweiligen Ressource.

Die vergleichende Analyse des zeitlichen Verlaufs zeigt, wie sich das hier untersuchte Szenario puzzelartig aus den beiden Strategien Caching und Parallelisierung zusammen-

setzt (*Abb. 5.7, 5.8*). Oberflächlich betrachtet nimmt der Verlauf die Gestalt des Parallel-Szenarios mit den zwei zentralen CPU-Höhepunkten an. Wieder folgt die Entwicklung der Leistungsaufnahme eng derer der CPU-Auslastung. Die Jobs lassen sich den folgenden Zeitmarken zuordnen (in Klammern die jeweilige Zeitspanne der Job-Phase): Build-Frontend von Sekunde 0 bis 30 (8–25), Build-API von Sekunde 0 bis 40 (9–31), Test-API von Sekunde 41 bis 89 (52–83) und Deployment von Sekunde 90 bis 129 (99–126).
Während der parallelen Ausführung der Build-Jobs ist der Verlauf von RAM und CPU ähnlich dem des Parallel-Szenarios, jedoch mit einem um 10 % (RAM) bzw. 3 % (CPU) niedrigeren Maximum (Sekunde 23). Damit spiegeln sich auch hier die obigen Abweichungen in den Mittelwerten wider. Durch das Erstellen der Caches kommt es ab der Post-Job-Phase des Frontend-Builds (Sekunde 26) zu einer vergleichsweise längeren Phase der erhöhten I/O-Aktivität.
Im Vergleich zum Cache-Aufbau-Szenario fallen die Ressourcen-Peaks der Build-Jobs während der parallelen Ausführung zusammen, was zu dem oben beschriebenen deutlich erhöhten Gesamt-Niveau von CPU- und RAM-Auslastung führt. Ebenso treten die I/O-Aktivitäten von Frontend- und API-Build vereint auf.
Der Analyse der sequentiell ablaufenden Jobs (Test-API, Deployment) sind keine Erkenntnisse zu entnehmen, die über die aus den Einzelanalysen gewonnenen Erkenntnisse hinausgehen. Ein markanter Unterschied im Deployment-Job ist abermals, dass die CPU während der Job-Phase des Deployments auf ein tieferes Niveau sinkt als im Cache-Aufbau-Szenario.

5.4.2 Cache-Nutzung und Parallelisierung

Dieses Kapitel untersucht die kombinierten Effekte der Cache-Nutzung und Parallelisierung. Hierzu werden die Analyseergebnisse des Parallel-Nutzungs- mit denen des Cache-Nutzungs- und des Parallel-Szenarios verglichen.

Die Gesamtlaufzeit im Parallel-Nutzungs-Szenario beträgt 1:52 Minuten (*Tab. A.1*). Im Vergleich zum Parallel-Szenario ergibt dies eine Beschleunigung um 22 Sekunden und somit eine Reduktion der Laufzeit um 16,41 %. Gegenüber dem Cache-Nutzungs-Szenario fällt die Laufzeit mit einer Differenz von 26 Sekunden (-19,37 %) nochmal geringer aus.

Bei der Betrachtung der mittleren Ressourcen auf Pipeline-Ebene im Vergleich zum Cache-Nutzungs-Szenario zeigt sich ein gemischtes Bild. Die CPU-Auslastung ist um 1,07 Prozentpunkte (+2,39 %) höher angesiedelt, jedoch nicht signifikant, während die

RAM-Auslastung signifikant, jedoch ebenfalls nur geringfügig um 0,32 Prozentpunkte (+1,87 %) zunimmt. Demgegenüber stehen eine um 53,15 MiB (-3,06 %) leicht reduzierte I/O-Last und eine deutliche Energieeinsparung von 0,23 Wh (-17,4 %).
Ein ähnliches Muster bei den Ressourcenveränderungen, wenn auch mit abweichenden Werten, zeigt sich im Vergleich zum Parallel-Szenario. Die CPU-Auslastung steigt um 3,16 Prozentpunkte (+7,42 %), die RAM-Auslastung um 0,08 Prozentpunkte (+0,43 %) an und die I/O-Last sinkt um 134,42 MiB (-7,39 %) sowie der Energieverbrauch um 0,15 Wh (-11,66 %).

Wie bereits im Parallel-Szenario laufen die Build-Jobs für die gesamte Dauer des Build-API-Jobs parallel. Da sich die Laufzeit des Build-API-Jobs durch die Cache-Nutzung drastisch reduziert hat und die des Frontend-Build-Jobs unterschreitet, beginnt der Test-API-Job frühzeitiger und läuft für durchschnittlich 4,72 Sekunden parallel zum Build-Frontend-Job. Die restliche Zeit läuft der Test-Job wie gehabt sequentiell ab.
Hinsichtlich der einzelnen Job-Laufzeiten lassen sich differenziertere Ergebnisse beobachten. Die Gesamtlaufzeit des Build-API-Jobs hat sich im Vergleich zum Cache-Nutzungs-Szenario um 3 Sekunden (+12,96 %) verlängert. Dies geht vor allem auf eine um 2,7 Sekunden verlängerte Job-Phase (+39,16 %) sowie eine um 1,1 Sekunden verlängerte Post-Job-Phase (+12,96 %) zurück, wohingegen sich die Pre-Job-Phase um etwas unter einer Sekunde verkürzt. Gegenüber dem Parallel-Szenario ergibt sich eine deutliche Verkürzung der Gesamtlaufzeit des Jobs um 11,92 Sekunden (-31,31 %). Die primäre Ursache hierfür liegt in der mehr als halbierten Laufzeit der Job-Phase, die um 12,92 Sekunden (-57,29 %) sinkt. Demgegenüber weisen die anderen beiden Phasen gemeinsam lediglich eine absolute Verlängerung um eine Sekunde (+6,44 %) auf.
Auch der Build-Frontend-Job verzeichnet im Vergleich zum Cache-Nutzungs-Szenario eine Zunahme der Gesamtlaufzeit um absolut 2,67 Sekunden (+9,47 %). Dieser zeitliche Zuwachs ist ebenfalls der Job-Phase zuzuordnen, die sich um 3,08 Sekunden (+21,26 %) verlängert, während die anderen Phasen sich nur marginal um jeweils unter einer Sekunde verändern. Aus dem Vergleich des Parallel-Nutzungs- mit dem Parallel-Szenario geht hervor, dass sich die Laufzeit des Build-Frontend-Jobs nur in sehr geringem Maße um eine halbe Sekunde (+5,45 %) verlängert. Während sich die Job-Phase zeitlich nahezu nicht verändert, reduziert sich die Laufzeit der Pre-Job-Phase um 0,53 Sekunden (-5,77 %), und verlängert sich die Post-Job-Phase um 0,95 Sekunden (+10,35 %).
Der Test-API-Job weist gegenüber dem Cache-Nutzungs-Szenario eine um 1,43 Sekunden (-3,11 %) reduzierte Laufzeit auf. Es zeigt sich, dass die Reduzierung absolut betrachtet vor allem aus der Job-Phase hervorgeht (-1,01 Sekunden, -3,25 %). Zusätzlich

beschleunigt sich die Pre-Job-Phase um 0,85 Sekunden (-6,97 %), wohingegen die Post-Job-Phase um etwas unter einer halben Sekunde länger andauert (+15,93 %), was sich aufgrund der insgesamt kurzen Dauer dieser Phase relativ betrachtet stark auswirkt. Im Vergleich zum Parallel-Szenario kommt es zu einer deutlichen Reduzierung der Gesamtlaufzeit um 9,26 Sekunden (-17,22 %). Dies ist auf eine um ein Viertel schnellere Job-Laufzeit zurückzuführen (-10,3 Sekunden, -25,53 %). Die anderen Phasen ändern sich absolut nur unwesentlich, mit einer jeweiligen Zunahme um circa eine halbe Sekunde, was in der Post-Job-Phase jedoch einer relativen Zunahme um 13 % entspricht.
Für den Deploy-Job reduziert sich in beiden Vergleichen die Pre-Job-Phase um circa eine Sekunde bei gleichzeitiger Verlängerung der Post-Job-Phase um etwa eine halbe Sekunde, während die Job-Phase jeweils nahezu unverändert bleibt.

Wie in den vorherigen vergleichenden Analysen der Szenarien mit aktivierter Parallelisierung konzentriert sich die Untersuchung der Job-Performance hinsichtlich der Ressourcenmessungen auf den parallelisierten Zeitabschnitt und vergleicht diesen mit der sequentiellen Ausführung im Cache-Nutzungs-Szenario und der parallelen Ausführung im Parallel-Szenario. Beim Vergleich mit dem Parallel-Szenario werden zusätzlich die ersten fünf Sekunden des Test-Jobs mitbetrachtet, die im Parallel-Nutzungs-Szenario parallel zum Build-Frontend-Job ablaufen.
In der parallelen Phase des Parallel-Nutzungs-Szenarios ist die CPU-Auslastung im Mittel um 28,62 % höher als im Cache-Nutzungs-Szenario. Außerdem zeigt sich eine um 37,29 % höhere durchschnittliche Maximalbelastung von 68,81 % sowie eine um 30,5 % höhere Minimalbelastung. Die durchschnittliche RAM-Auslastung ist mit einem Plus von 0,88 Prozentpunkten nur um 5,32 % höher, und die durchschnittliche Maximalbelastung steigt um 4,51 %. Gleichzeitig ist die I/O-Last leicht um 4,34 % (-43,26 MiB) reduziert, und der Energieverbrauch sinkt deutlich um fast ein Drittel (-31,13 %, -0,17 Wh).
Gegenüber der parallelen Phase im Parallel-Szenario ergibt sich eine um 5,16 % höhere mittlere CPU-Auslastung, eine um 9,14 % höhere durchschnittliche Maximal-Auslastung sowie eine um 4,8 % höhere durchschnittliche Minimalbelastung. Der durchschnittliche RAM-Bedarf ist mit einer Reduktion um 0,06 Prozentpunkte nur unwesentlich geringer (-0,32 %), und die durchschnittliche Maximal-Auslastung ist nahezu identisch. Die I/O-Last ist um 6,23 % reduziert (-64,89 MiB) und der Energieverbrauch sinkt deutlich um fast ein Viertel (-23,38 %, -0,12 Wh).

Die Jobs lassen sich folgenden Zeitabschnitten im Zeitreihendiagramm zuordnen, wobei die Zeitangaben auf ganze Sekunden gerundet sind und der Abschnitt der Job-Phase jeweils in Klammern angegeben ist: Build-API von Sekunde 0 bis 26 (10–19), Build-

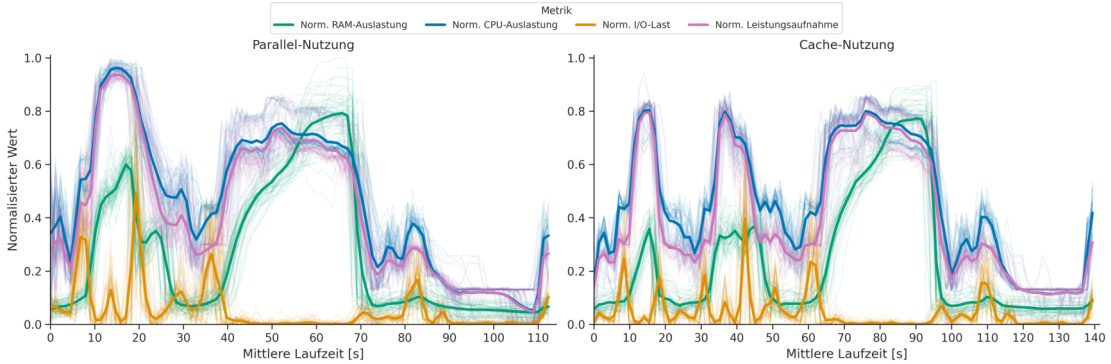

Abbildung 5.9: Vergleich des zeitlichen Verlaufs des Parallel-Nutzungs- (links) und des Cache-Nutzungs-Szenarios (rechts) anhand der zentralen Messwerte. Messwerte wurden für den Vergleich normiert. Zeitwerte geben die mittlere Laufzeit an. Hervorgehobene Linien: mittlerer Verlauf der jeweiligen Ressource.

Frontend von Sekunde 0 bis 31 (9–27), Test-API von Sekunde 27 bis 73 (40–70) und Deployment von Sekunde 74 bis 112 (83–109). Wie in den zuvor vorgestellten Szenarien mit aktivierter Parallelisierung sind auch hier wieder zwei statt der bei sequentiell ablaufenden Pipelines drei CPU- und RAM-Peaks sichtbar (*Abb. 5.7, 5.8*). Genau wie in allen Szenarien folgt die Leistungsaufnahme stark der CPU-Auslastung.

Im Vergleich zum Parallel-Szenario fällt eine geringere Anzahl markanter I/O-Peaks in der parallel verlaufenden Phase auf, die dafür teilweise deutlich höher ausfallen. Die anzahlmäßige Verringerung der I/O-Peaks ist bereits aus dem Vergleich zwischen Cache-Nutzungs-Szenario und Basis-Szenario bekannt. Das RAM-Maximum während der parallelen Phase wird bereits bei Sekunde 17 erreicht und damit acht Sekunden früher als im Parallel-Szenario bei Sekunde 25. Infolge der verkürzten Build-API-Job-Laufzeit kommt es zu einer Vertauschung der beiden lokalen RAM-Maxima aus dem Parallel-Szenario (Sekunde 18 und 25). Das RAM-Maximum ist zudem um circa 9 % niedriger als das Maximum in der parallelen Phase des Parallel-Szenarios. Weiter ist das CPU-Hoch zwischen Sekunde 11 und 18 deutlich konstanter und um sieben Sekunden gegenüber dem CPU-Hoch im Parallel-Szenario verkürzt, da sich die CPU-lastigen Build-Prozesse stärker überlagern. Das CPU-Maximum bei Sekunde 15 ist hingegen nur um knapp 7 % höher.

Eine Folge des Cachings, die ebenfalls im Vergleich des Cache-Nutzungs- und des Basis-Szenarios erläutert wurde, ist das Ausbleiben des CPU-Peaks zu Beginn der Job-Phase des Test-API-Jobs aufgrund der Cache-Nutzung. Im Deployment ist die CPU-Auslastung während der Job-Phase wieder über eine längere Phase erhöht.

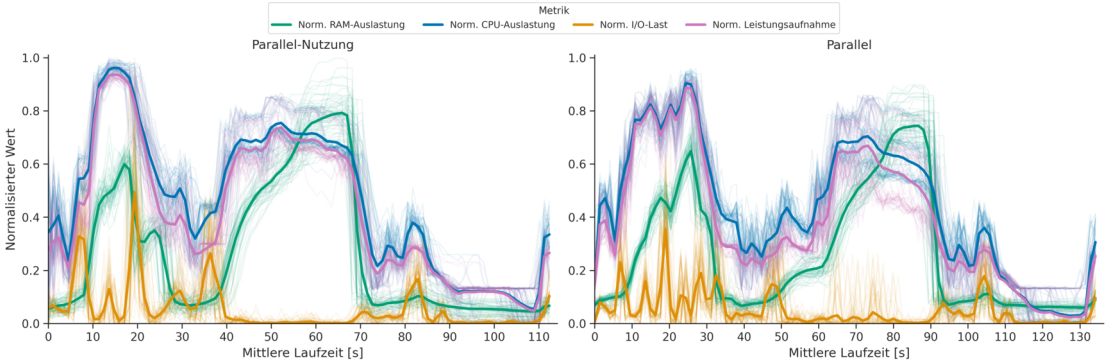

Abbildung 5.10: Vergleich des zeitlichen Verlaufs des Parallel-Nutzungs- (links) und des Parallel-Szenarios (rechts) anhand der zentralen Messwerte. Messwerte wurden für den Vergleich normiert. Zeitwerte geben die mittlere Laufzeit an. Hervorgehobene Linien: mittlerer Verlauf der jeweiligen Ressource.

Gegenüber dem Cache-Nutzungs-Szenario zeigt sich, dass die CPU- und RAM-Auslastung während der parallelen Verarbeitung deutlich höher ausfällt. Darüber hinaus ist erkennbar, dass sich der maximale I/O-Last-Höhepunkt bei Sekunde 20 aus den I/O-Peaks der beiden Build-Jobs im Cache-Nutzungs-Szenario bei Sekunde 19 und 42 zusammensetzt. Ansonsten zeigen sich keine weiteren Auffälligkeiten, die sich nicht bereits aus dem Vergleich des Parallel- mit dem Basis-Szenario ergeben.

Zuletzt wird die zeitliche und energetische Amortisierung der kombinierten Strategie durch die Nutzung der Caches untersucht. Die Kosten für das Aufsetzen des Caches im Parallel-Aufbau-Szenario sind negativ und belaufen sich auf durchschnittlich -0,02 Wh beziehungsweise -4,93 Sekunden. Das bedeutet, dass sich die Verwendung von Caching unabhängig von der Nutzung in einem Folgedurchlauf durchschnittlich schon im Cache-aufbauenden Pipeline-Durchlauf amortisiert. Für das detailliertere Verständnis hilft an dieser Stelle ein Blick auf die Laufzeit weiter: Obwohl für den Cache-Aufbau im Build-API-Job ein zeitlicher Mehraufwand von 1,3 Sekunden im Vergleich zum Parallel-Szenario anfällt, amortisiert sich dieser durch die Einsparungen von durchschnittlich 11,92 Sekunden bei der ersten Nutzung im Parallel-Nutzungs-Szenario. Dieser Effekt ist auch innerhalb des Cache-aufbauenden Pipeline-Durchlaufs zu sehen, da der Test-API-Job bereits hier auf den vorgebauten Cache zurückgreift. Wegen dieser Weitergabe lohnt sich auch das Updaten des Caches im Test-API-Job bereits im selben Durchlauf, da die Kosten für das Cache-Update negativ sind und die Einsparungen mit über 9 Sekunden vergleichbar mit denen im Build-API-Job ausfallen.
Interessanterweise fallen die Kosten für den Cache-Aufbau im Frontend-Build-Job mit

0,81 Sekunden zwar gering aus, es kommt durch die Nutzung jedoch – anders als im Vergleich der Caching-Szenarien ohne Parallelisierung – im Schnitt nicht zu Einsparungen, sondern sogar zu zusätzlichen Kosten von 0,5 Sekunden. Durch die hohen Einsparungen seitens der Cache-Nutzung im Build-API- und Test-API-Job wirkt sich dies jedoch nur unwesentlich auf die Einsparungen der Gesamtpipeline aus.

5.5 Gesamtbewertung der Optimierungsstrategien

Nachdem in den vorangegangenen Kapiteln die Effekte der Caching- und Parallelisierungsstrategien isoliert und in Kombination analysiert wurden, führt dieses Kapitel die Ergebnisse zusammen. Ziel ist eine vergleichende Gesamtbewertung, um die effektivste Optimierungsstrategie hinsichtlich des Energieverbrauchs und der Laufzeit zu identifizieren (H3) und eine kurze Übersicht über die Kernerkenntnisse aus den vorherigen Ergebniskapiteln zu geben. Als Grundlage für den übergreifenden Vergleich dient Abbildung *5.11*, welche die prozentualen Abweichungen der zentralen Metriken für jedes Szenario gegenüber dem Basis-Szenario visualisiert und die Szenarien entsprechend der erreichten Energieeinsparung anordnet.

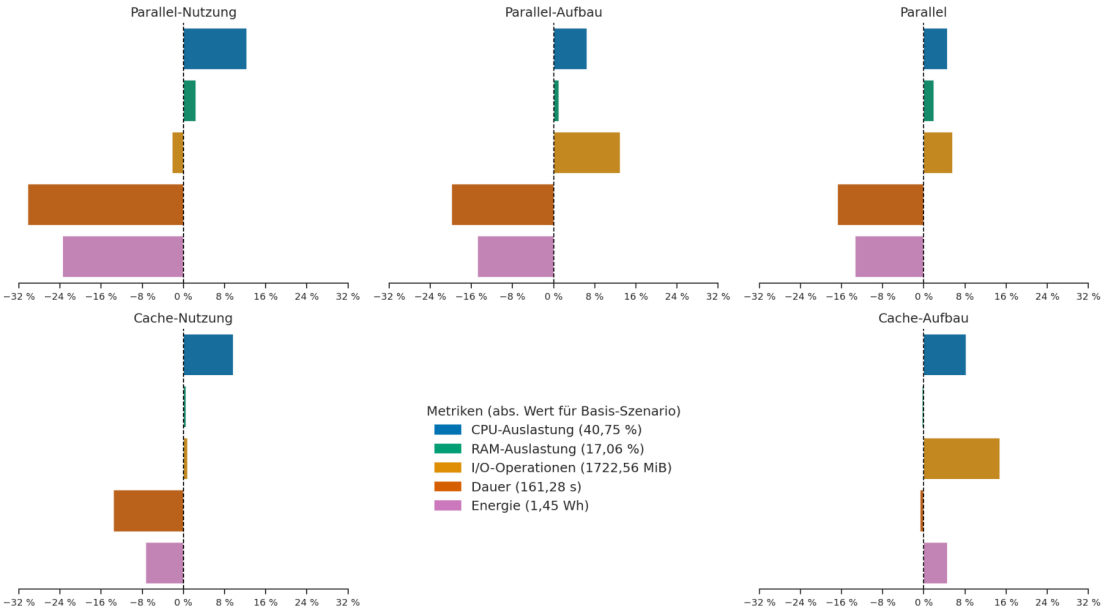

Abbildung 5.11: Relative mittlere Abweichung der zentralen Metriken je Szenario im Vergleich zum Basis-Szenario. Die Szenarien sind, oben links beginnend, absteigend nach ihrem mittleren Energieverbrauch sortiert.

Die Abbildung *5.11* verdeutlicht die unterschiedliche Wirksamkeit der angewandten Strategien. Während das reine Cache-Aufbau-Szenario als einzige Strategie zu einem leichten Anstieg des Energieverbrauchs (+4,57 %) gegenüber dem Basis-Szenario führt, erzielen alle anderen Szenarien signifikante Einsparungen bzgl. sowohl des Energieverbrauchs als auch der Laufzeit.
Am stärksten zeigt sich dieser Effekt im Parallel-Nutzungs-Szenario, dessen Pipelines durchschnittlich 30,43 % schneller sind als die des Basis-Szenarios. Selbst das energetisch und laufzeittechnisch zweit-effizienteste Szenario, das Parallel-Aufbau-Szenario, wird durch das Parallel-Nutzungs-Szenario mit einer 13,03 % schnelleren Laufzeit (-16,83 Sekunden) sowie einem um 10,48 % (-0,13 Wh) geringeren Energiebedarf deutlich übertrumpft.
Aus der Anordnung der Szenarien nach ihrer Energieeinsparung im Vergleich zum Basis-Szenario geht hervor, dass alle Szenarien mit aktivierter Parallelisierung sowohl energetisch als auch zeitlich einen geringeren Fußabdruck aufweisen als Szenarien ohne Parallelisierung. Etwas differenzierter geht hervor, dass zum einen die kombinierte Anwendung von Caching und Parallelisierung energetisch und zeitlich effizienter ist als die alleinige Anwendung einer einzelnen Strategie und zum anderen die Cache-Nutzung zeitlich und energetisch effizienter ist als der Cache-Aufbau oder eine Pipeline ohne Caching.

Als ein wiederkehrendes Muster über alle Szenarien hinweg wurde die enge Beziehung zwischen CPU-Auslastung, Laufzeit und Energieverbrauch herausgearbeitet. Der zeitliche Verlauf der Leistungsaufnahme folgt stets stark dem der CPU-Auslastung, wodurch die finalen Energiewerte maßgeblich durch die Intensität und Dauer der CPU-Nutzung bestimmt werden. Dies wird auch durch die Untersuchung des Zusammenhangs des Energiebedarfs und der Laufzeit aller Pipelines über die Szenarien hinweg bestätigt. Die Korrelation auf Basis von Spearman ergibt eine sehr starke positive Korrelation ($r_s = 0,944$) zwischen dem Energiebedarf und der Laufzeit, die zudem hochsignifikant ist ($p < 0,001$). Wenn auch dies nicht weiter verwunderlich ist, da Energie letztlich die Leistung über Zeit widerspiegelt, so ist interessant, dass die CPU- und die Watt-Messwerte ebenfalls sehr stark, sogar nahezu perfekt positiv miteinander korrelieren ($r_s = 1,000$, $p < 0,001$). Zusammengenommen bedeutet dies, dass der Energiebedarf maßgeblich von der Höhe und Dauer der CPU-Auslastung abhängt.

Die Analyse der Caching-Strategie (H1) ohne Parallelisierung zeigt, dass der initiale Cache-Aufbau eine Investition darstellt. Er führt zu einer erhöhten CPU-Auslastung um 8,25 %, einer Steigerung der I/O-Operationen um 14,76 % und einem um 4,57 % höheren Energiebedarf im Vergleich zum Basis-Szenario. Der Anstieg der CPU-Auslastung sowie

des Energiebedarfs ist dabei primär auf den Mehraufwand im Build-API-Job zurückzuführen. Eine ebenfalls beobachtete, erhöhte CPU-Last im Deployment-Job stellt sich bei näherer Analyse als normale Schwankung mit vernachlässigbarem Einfluss heraus, da sich der Anteil des Deployment-Jobs am Gesamtenergiebedarf der Pipeline im Vergleich zum Basis-Szenario lediglich um 0,5 Prozentpunkte erhöht, und die absolute Energiezunahme nur halb so groß ist wie jene des Build-API-Jobs.
Die energietechnischen Kosten des Cache-Aufbaus amortisieren sich bereits mit der ersten nachfolgenden Cache-Nutzung. Hier sinken die Laufzeit um 13,6 % und der Energieverbrauch um 7,38 % gegenüber einer Pipeline im Basis-Szenario. Die Effizienzsteigerung resultiert primär aus der drastischen Verkürzung des Build-API-Jobs, dessen Job-Phase sich um fast zwei Drittel reduziert, sowie des Test-API-Jobs, der um 16,62 % beschleunigt wird. Ein Nebeneffekt ist die um 9,71 % höhere durchschnittliche CPU-Auslastung bei der Cache-Nutzung. Dieser Zuwachs erklärt sich durch das intensivere Arbeiten der CPU in den Build- und Test-Jobs, da weniger auf I/O-Operationen gewartet wird.

Die Untersuchung der Parallelisierung (H2) untermauert die starke Korrelation von Laufzeit und Energieverbrauch. Durch die parallele Ausführung der beiden Build-Jobs wird die Gesamtlaufzeit der Pipeline um 16,84 % verkürzt. Während der parallelen Phase steigt die Ressourcennutzung im Vergleich zur sequentiellen Ausführung deutlich an (CPU-Auslastung: +50,23 %, RAM-Auslastung: +7,83 %). Trotz dieser höheren Auslastung sinkt der Energieverbrauch hier um 26,24 %, da die Ausführungsdauer fast halbiert wird (-46,91 %).

Die Kombination beider Strategien (H3) maximiert die Energie- und Laufzeiteinsparungen durch vorteilhafte Synergieeffekt, weshalb das Parallel-Nutzungs-Szenario gegenüber allen anderen Konfiguration überlegend ist. Konkret verbindet es die durch Parallelisierung verkürzte Gesamtlaufzeit mit den durch Caching beschleunigten Einzeljobs. Ein entscheidender Vorteil entsteht dadurch, dass der durch die Cache-Nutzung stark beschleunigte Build-API-Job früher fertiggestellt wird als der Build-Frontend-Job. Dies ermöglicht dem Test-API-Job einen früheren Start, sodass er für durchschnittlich 4,72 Sekunden parallel zum noch laufenden Frontend-Build-Job ausgeführt wird, was die Gesamtlaufzeit weiter reduziert.
Die Amortisierung der Cache-Aufbau-Kosten erfolgt im Parallel-Kontext besonders schnell. Der Mehraufwand für den Cache-Aufbau im Build-API-Job wird noch innerhalb desselben Pipeline-Durchlaufs durch die Einsparungen im nachfolgenden Test-API-Job (ca. 9 Sekunden) kompensiert. Dies führt dazu, dass bereits das Parallel-Aufbau-Szenario effizienter ist als das Parallel-Szenario.

Zusammenfassend lässt sich festhalten, dass die Kombination aus Parallelisierung und Caching die wirkungsvollste Strategie zur Reduzierung des Energieverbrauchs darstellt, da sich beide positiv auf die Reduzierung der Laufzeit auswirken und infolgedessen den Energieverbrauch minimieren. Die Kosten hierfür sind eine durchschnittlich deutlich erhöhte I/O-Last durch den Cache-Aufbau und phasenweise stark gesteigerte Ressourcenwerte während der parallelen Ausführung der Jobs.

6 Diskussion

Nachdem im vorangegangenen Kapitel die empirischen Ergebnisse der Untersuchung detailliert vorgestellt wurden, widmet sich dieses Kapitel ihrer Diskussion und Interpretation. Ziel ist es, die quantitativen Resultate zu den aufgestellten Hypothesen (*H1–H3*) zu erörtern, die dahinterliegenden technischen Mechanismen zu analysieren und die Erkenntnisse in den vorgestellten wissenschaftlichen Kontext einzuordnen. Abschließend erfolgt eine kritische Reflexion der gewählten Methodik, um die Validität sowie die Generalisierbarkeit der gemessenen Resultate zu bewerten.

6.1 H1: Dependency-Caching

Die Analyse der Ergebnisse bestätigt Hypothese *H1* überwiegend in ihren beiden zentralen Annahmen. Der erste Teil der Hypothese, wonach die Cache-Initialisierung zu einem Anstieg in der Ressourcennutzung und der Laufzeit führt, wird durch die Ergebnisse weitestgehend belegt: Im Cache-Aufbau-Szenario sind die durchschnittlichen Werte der CPU-Auslastung (+8,25 %), I/O-Last (+14,76 %) und des Energieverbrauchs (+4,57 %) im Vergleich zum Basis-Szenario deutlich gestiegen. Dieser Mehraufwand lässt sich durch den zusätzlichen, I/O- und CPU-intensiven Prozess des Erstellens, Komprimierens und Schreibens der Cache-Archive erklären (*Kap. 3.2.3*). Abweichend hiervon hat sich die RAM-Auslastung nicht signifikant verändert.

Während der Cache-Aufbau zu einer mehrheitlichen Steigerung der gemessenen Ressourcennutzung auf Pipeline-Ebene führt, ist der Effekt auf die Laufzeit differenzierter zu betrachten. Die Laufzeit der Pipeline-Durchläufe im Cache-Aufbau-Szenario hat sich lediglich nicht-signifikant im Vergleich zur Referenzpipeline verlängert. Allerdings wurde als eine Besonderheit das Caching-Verhalten im Test-API-Job herausgearbeitet, da dieser keinen gänzlich neuen Cache erstellt, sondern den vom Build-Api-Job erstellten Cache um die spezifischen Test-Dependencies erweitert und ansonsten auf die bereitgestellten Build-Dependencies zugreift. Dieser Vorgang führt zu einer signifikanten und

deutlichen Verkürzung der Job-Laufzeit um 11,22 % (-5,51 Sekunden), die noch im selben Pipeline-Durchlauf zur Kompensierung der zeitlichen Cache-Aufbaukosten (+4,08 Sekunden) durch die beiden Build-Jobs führt. Vor diesem Hintergrund ist die Annahme, dass der Cache-Aufbau zu einer erhöhten Laufzeit führt, ebenfalls zu bestätigen.

Die zweite zentrale Annahme der Hypothese, der zufolge nachfolgende Pipelines von der Cache-Nutzung durch eine vergleichsweise geringere Ressourcennutzung und Laufzeit profitieren, wird durch die Ergebnisse weitestgehend gestützt.
Die Cache-Nutzung führt im Vergleich zum Basis-Szenario zu einer drastischen Reduzierung der Laufzeit (-13,6 %) und des Energieverbrauchs (-7,38 %), während die RAM-Nutzung und die I/O-Last keine signifikanten Veränderungen aufweisen. Dieses Ergebnis untermauert quantitativ die von Gallaba et al. [21] beschriebenen Vorteile des Cachings und übertrifft die von Bouzenia und Pradel [3] für GitHub-Projekte ermittelte Laufzeitreduktion zwischen 3 und 6 % deutlich.
Entgegen der Erwartung einer generellen Ressourcenreduktion steht jedoch die signifikant erhöhte durchschnittliche CPU-Auslastung (+9,71 %). Dieses scheinbare Paradoxon lässt sich durch den in den Grundlagen erläuterten Wandel des Prozesses von einem zumindest in Teilen I/O-gebundenen zu einem CPU-gebunden Prozess erklären [12, 17, 8]. Anstatt auf den zeitintensiven Download von Dependencies aus dem Netzwerk zu warten, lädt die Pipeline den lokalen Cache und greift darüber auf die Dependencies zu. Dieser erlangte Zeitvorteil zeigt sich eindrücklich in der gemessenen Reduzierung der Job-Phase im Test-API-Job um mehr als ein Viertel. Durch den Entfall der Wartezeit kommt es durchschnittlich zu einer höheren Auslastung, jedoch in einer insgesamt kürzeren Zeit, sodass letztlich weniger Energie verbraucht wird, da der Effekt der verkürzten Laufzeit den der erhöhten CPU-Auslastung übersteigert.
Eine weitere Abweichung von der Annahme zeigt sich im Build-Frontend-Job, bei dem die Effekte der Cache-Nutzung sich nur unwesentlich auf dessen aggregierten Ressourcenwerte und Laufzeit auswirken. Dies ist vermutlich auf die geringe Anzahl und Größe der Dependencies in diesem Job zurückführen. Ein Beitragen der Netzwerk-Last wurde in dieser Arbeit nicht untersucht.

Dass die I/O-Last im Cache-Nutzungs-Szenario entgegen der Erwartung im Vergleich zum Basis-Szenario nicht gesunken ist, erklärt sich durch GitLabs technische Implementierung des lokalen Cachings in Form von Docker-Volumes, die als Verzeichnisse im Dateisystem des Host-Systems eingebunden sind (*Kap. 3.2.4*). Anstelle der netzwerkbasierten Downloads tritt daher eine erhöhte Festplatten-I/O-Last. Dieser Austausch der I/O-Art manifestiert sich in den Ergebnissen durch ausgeprägte I/O-Spitzen in den

Pre-Job-Phasen der Jobs. Gleichermaßen ist im Falle des Cache-Aufbaus eine erhöhte I/O-Last in den Post-Job-Phasen zu verzeichnen.
Ein zentrales Ergebnis der Cache-Nutzung über Docker-Volumes ist außerdem das Ausbleiben der I/O-Schwankungen während der Testausführung, die im Basis-Szenario durch das Neuladen des Spring-`ApplicationContext` entstehen. Dieses Phänomen lässt sich durch das in den Grundlagen erläuterte Zusammenspiel der Docker-Volumes mit dem Page-Cache des Host-Betriebssystems erklären(*Kap. 3.2.4*): Trotzdem das Lesen der lokalen Caches initial die I/O-Last erhöht, werden sie nach dem Lokalitätsprinzip im RAM angesiedelten Page-Cache des Host-Systems vorgehalten. Nachfolgende Lesezugriffe auf dieselben Dateien werden somit potenziell aus dem extrem schnellen RAM bedient, sodass entsprechende I/O-Aktivitäten entfallen.

Im Grundlagenkapitel werden alle Caching-Praktiken und so auch das Dependency-Caching auf dem Spektrum zwischen Ressourcenoptimierung und -vermeidung eher der Ressourcenoptimierung zugeordnet (*Kap. 3.2.3*). Die hier empirisch herausgearbeiteten Erkenntnisse unterstreichen diesen teils hybriden Charakter. Auf der einen Seite werden durch die Cache-Nutzung wiederholte, zeitintensive Netzwerkzugriffe vermieden. Auf der anderen Seite wird jedoch weiterhin die gleiche Anzahl an Aufgaben ausgeführt und lediglich der Download der Dependencies über das Netzwerk durch das Laden der Dependencies aus dem lokalen Speicher ausgetauscht, was eine Optimierung darstellt. Aus ebendiesem Grund, dass es durch das Dependency-Caching nicht zu einer Reduzierung der Menge der Aufgaben kommt, sondern vielmehr eine schnellere I/O-Quelle verwendet wird, wird die in den Grundlagen vorgenommene Einordnung nahe der Ressourcenoptimierung hier empirisch bestätigt.

Zusammenfassend lässt sich festhalten, dass die Kosten des Cache-Aufbaus real sind, sich aber bereits bei der ersten nachfolgenden Nutzung durch signifikante Einsparungen bei Laufzeit und Energieverbrauch amortisieren. Obwohl der lokale Festplatten-I/O die Netzwerk-Last ersetzt und die CPU-Auslastung durch wegfallende Wartezeiten sogar steigt, kompensiert der massive Zeitgewinn bei Weitem die Effekte der kurzzeitig höheren Leistungsaufnahme.

6.2 H2: Parallelisierung

Die Untersuchung der Parallelisierungsstrategie in Kapitel 6.2 bestätigt die Hypothese *H2* vollständig. Durch die parallele Ausführung der Build-Jobs wird die Gesamtlaufzeit der

Pipeline um durchschnittlich 16,84 % verkürzt und ihr Energieverbrauch um 13,4 % gesenkt. Trotz einer um über 50 % gesteigerten CPU-Auslastung während der parallelen Ausführung sinkt der Energieverbrauch hier um mehr als ein Viertel aufgrund der nahezu halbierten Laufzeit im Vergleich zur sequentiellen Ausführung. Somit bekräftigen die Messungen die im Grundlagenkapitel angeführten Überlegungen von Tatineni [58] und Limbrunner [45], die postulieren, dass Parallelisierung den Energieverbrauch senken kann, sofern Ressourcen dadurch effizienter ausgelastet werden und die Hardware früher in den Leerlauf übergehen kann. Die erhöhte Spitzenauslastung ist folglich kein Nachteil, sondern ein Beleg für eine gesteigerte Effizienz.

Die erzielte Laufzeitreduzierung untermauert die Wirksamkeit der Parallelisierung, erreicht jedoch nicht die bei Mathew und S R [46] vorgestellte durchschnittliche Reduktion von rund 40 %. Dieser Unterschied lässt sich darauf zurückführen, dass Mathew und S R die Ressourcennutzung am Beispiel verschiedener, deutlich umfangreicherer Unternehmenspipelines untersucht haben, während in dieser Arbeit eine einzelne, vergleichsweise kleine Pipeline untersucht wurde. Daher bergen die Pipelines bei Mathew und S R durch eine höhere Anzahl an zudem länger dauernden Jobs ein größeres Paral-lelisierungspotenzial in sich, das zu den gezeigten höheren Laufzeiteinsparungen führte.
Die beobachtete leichte Verlängerung der Laufzeiten der einzelnen parallelisierten Jobs lässt sich auf die erhöhte Ressourcenkonkurrenz zurückführen, bei der mehrere Prozesse gleichzeitig um CPU-Zeit und andere Systemressourcen konkurrieren (*Kap. 3.2.3*).

Zusammenfassend ist die bewusste Inkaufnahme höherer Leistungsspitzen eine effektive Strategie zur Energie- und Laufzeitoptimierung. Die im Grundlagenkapitel vorgenommene Klassifizierung der Parallelisierung als ressourcenoptimierende Best Practice (*Abb. 3.1*) ist folglich auf Basis der empirischen Ergebnisse zu bestätigen, da die Parallelisierung von Jobs den Umfang der auszuführenden Jobs nicht reduziert, sondern ihre Ausführungsreihenfolge auf einer höheren Ebene restrukturiert und so zu einer optimierten Auslastung der Systemressourcen beiträgt. Voraussetzung ist, dass die zugrundeliegende Hardware über ausreichende Kapazitäten verfügt, um die erhöhte Last während der parallelen Ausführung zu bewältigen.

6.3 H3: Kombination der Optimierungsstrategien

Die Untersuchung der kombinierten Nutzung von Caching und Parallelisierung bestätigt die Gültigkeit der Hypothese *H3*: Aus den Ergebnissen geht hervor, dass die Verbin-

dung beider Strategien die wirkungsvollste Methode zur Reduzierung von Laufzeit und Energieverbrauch ist und die Effekte der Einzelstrategien übertrifft.

Nicht nur die beiden kombinierten Strategien sind zeitlich und energetisch effizienter als die Referenzpipeline, sondern bereits das Cache-Nutzungs- sowie das Parallel-Szenario. Während das Cache-Aufbau-Szenario das einzige Szenario darstellt, dass bezüglich der genannten Effizienzmaße ineffizienter ist als das Basis-Szenario, führt die Kombination von Parallelisierung und Caching dazu, dass der kostenintensive Cache-Aufbau relativ gesehen an Bedeutung verliert. Folglich weist das Parallel-Aufbau-Szenario trotz des Cache-Aufbaus eine deutlich geringere mittlere Gesamtlaufzeit (-19,9 %) und geringeren Energiebedarf (-14,48 %) auf als das Basis-Szenario.

Elementar hervor gehen die positiven Effekte jedoch aus der Analyse des Parallel-Nutzungs-Szenarios. Hier ist nicht nur jeder Pipeline-Durchlauf um mindestens ein Fünftel schneller als der schnellste Durchlauf der Referenzpipeline, sondern auch um knapp ein Zehntel energiesparender. Noch eindrücklicher sind die mittleren Einsparungen, die eine Laufzeitreduzierung um fast ein Drittel sowie Energieeinsparungen um ein Viertel aufzeigen.

Der entscheidende Grund für diese Effizienzsteigerung liegt in den Synergieeffekten der beiden Strategien. Die extreme Beschleunigung des Build-API-Jobs durch die Cache-Nutzung ermöglicht die kurzzeitige Parallelisierung des Test-API- und des Build-Frontend-Jobs. Dieser Effekt, bei dem die durch Caching gewonnene Zeit direkt in weitere Parallelisierung umgesetzt wird, führt zu einer noch effizienteren Ressourcenauslastung durch Maximierung der Ressourcennutzung bei gleichzeitiger Minimierung der Gesamtlaufzeit.

Die vorgestellten Erkenntnisse stehen im Einklang mit den Resultaten von Mathew und S R [46], die in ihrer Studie ebenfalls die kombinierte Anwendung von Best Practices als wirkungsvollste Methode identifizierten. In ihrer Untersuchung, die Parallelisierung und verschiedene Caching- sowie Testoptimierungs-Methoden umfasste, wurde einer Reduzierung der Pipeline-Laufzeit um 58,7 % erzielt. Die vorliegende Arbeit bestätigt diesen Befund, indem sie durch die alleinige Kombination von Parallelisierung und einer der Caching-Strategien bereits eine Laufzeitreduzierung von fast einem Drittel erreicht.

Indem die Kombination von Parallelisierung und Caching das Potenzial beider Strategien durch synergetische Effekte maximiert, stellen die erzielten Optimierungen letztlich mehr dar als nur die Summe ihrer Teile.

6.4 Methodenkritik und Limitationen

Nach der Präsentation und Interpretation der empirisch ermittelten Ergebnisse folgt an dieser Stelle die kritische Reflexion der zugrundeliegenden Methodik, um die Aussagekraft der Ergebnisse einzuordnen.

Die vorliegende Untersuchung ist als Einzelfallstudie konzipiert, deren Experimente anhand eines einzelnen Softwareprojekts mit spezifischen Technologie-Stack und auf einer einzigen Hardware-Konfiguration durchgeführt wurden. Daher sind die quantitativen Ergebnisse, wie beispielsweise die genauen prozentualen Laufzeit- und Energieeinsparungen, nicht direkt auf andere Konfigurationen übertragbar. Die nachgewiesenen Wirkmechanismen, etwa der Wandel von I/O- zu CPU-gebundenen Prozessen durch Caching, bestätigen jedoch grundlegende, in der wissenschaftlichen Literatur beschriebene Prinzipien. Weshalb anzunehmen ist, dass die grundlegenden Erkenntnisse über die Optimierungseffekte von Caching und Parallelisierung sowie ihre synergetischen Effekte trotzdem auf andere Projekte mit ähnlicher technologischer Ausrichtung übertragbar sind.

Ein weiterer wichtiger Aspekt, der die Aussagekraft der Ergebnisse einschränkt, ist die vergleichsweise geringe Größe der eingesetzten Pipeline sowohl hinsichtlich der Anzahl als auch der Größe der Jobs. Die Referenzpipeline benötigt mit einer durchschnittlichen Laufzeit von nur 2:41 Minuten und einem Energieverbrauch von 1,45 Wh deutlich weniger Ressourcen als Pipelines anderer Studien. Zum Beispiel gibt Limbrunner [45] für seine Messungen eine durchschnittliche Pipeline-Laufzeit von etwa 14 Minuten mit einem Energieverbrauch von circa 7 Wh an. Diese geringe Skalierung wirkt sich vermutlich insbesondere auf die Effizienzsteigerungen durch Caching aus, die im Build-Frontend-Job minimal bis nicht erkennbar sind, da dort weniger und kleinere Dependencies zu verarbeiten sind, was den Nutzen von Caching einschränkt.

Während der Messungen konnten nicht alle externen Einflüsse vollständig kontrolliert werden. Dementsprechend gab es Prozesse auf dem GitLab-Runner-Host, die die Messergebnisse in geringem Maße beeinflusst haben. Ein solcher Effekt ist beispielsweise bei der Analyse des Parallel-Szenarios zu beobachten, weshalb die dort erhobenen RAM-Werte für die Auswertung korrigiert wurden. Ebenso wurden im Deployment-Job teilweise erhöhte CPU-Werte registriert. Wie die Analyse in Kapitel 6.1 jedoch gezeigt hat, wirken sich diese Abweichungen nur unwesentlich auf die Gesamtperformance der Pipeline-Durchläufe aus. Zudem legen die hohe Stabilität und geringe Varianz der Messwerte über

die meisten Szenarien und Jobs hinweg nahe, dass externe Einflüsse insgesamt nur einen vernachlässigbaren Einfluss auf die Resultate hatten.

Die Validität der Ergebnisse hängt entscheidend von der gewählten Messmethode ab. Die in dieser Arbeit ermittelten Energiewerte basieren auf einem Machine-Learning-Modell, das primär die CPU-Auslastung als Eingangsvariable nutzt. Wie im Methodik-Kapitel beschrieben, wurde dieses Modell auf einem Datensatz trainiert, der fast ausschließlich CPU-intensive Systeme umfasst. Dies schränkt seine Genauigkeit für I/O-lastige Prozesse potenziell ein. Daher sind die absoluten Energiewerte lediglich als Schätzungen zu betrachten. Die Aussagekraft über die relativen Energieunterschiede, die einen zentralen Bestandteil dieser Arbeit darstellt, bleibt hiervon jedoch unberührt, da aus den Messergebnissen hervorgeht, dass die gemessenen Prozesse überwiegend CPU-gebunden sind. Dass eine exemplarische Hochrechnung der gemessenen Durchschnittswerte nur unwesentlich von den durch Limbrunner [45] ermittelten Energiewerten abweicht, stützt diese Annahme zusätzlich. Nichtsdestotrotz wären direktere Energiemessungen, die zudem sensibel für RAM-Auslastung und I/O-Last sind, vom Vorteil.

Eine weitere Einschränkung stellt die unvollständige Erfassung der Pipeline dar. Der Energieverbrauch des Deployment-Jobs, der das Deployment auf einem externen Ziel-Server anstößt, wurde nur auf dem GitLab-Runner-Host gemessen, nicht jedoch auf dem Ziel-Server. Die ermittelten Gesamtenergiewerte der Pipelines sind somit systematisch unvollständig. Da die Konfiguration des Deployment-Jobs aber in allen Szenarien unverändert blieb und er keine zentrale Rolle im Experimentdesign spielt, ist die Vergleichbarkeit der Szenarien untereinander weiterhin vollständig gegeben.

Des Weiteren wurde keine Messung der Netzwerk-I/O-Last durchgeführt. Eine solche Messung hätte u. a. die Aussage, dass der Download von Dependencies aus dem Netzwerk zeitintensiver ist als das Laden aus einem lokalen Cache, untermauern können.

Der initiale Versuchsaufbau, nach dem die Messungen durchgeführt wurden, sah Messungen sowohl in den Job-Containern als auch auf dem GitLab-Runner-Host vor. Durch eine Vereinfachung des Designs auf Basis der Synchronisation von Zeitstempeln finden sich die Container-Messungen in den Ergebnissen nur noch zur Identifizierung der Start- und Endzeiten der Job-Phasen wieder. Für diesen Zweck hätte ein leichtgewichtigeres Setup ausgereicht. Da die Container-Messungen gleichermaßen in allen Szenarien und Jobs durchgeführt wurden, schränken sie die Vergleichbarkeit der Daten jedoch nicht ein. Die Synchronisation von Zeitstempeln in verteilten Systemen stellt grundsätzlich eine Herausforderung dar, da es keine gemeinsame Zeit der Systeme gibt. In der vorliegenden

Arbeit wurde die Zeit-Synchronisation zwischen dem GitLab-Server und dem GitLab-Runner-Server aufgrund der Nutzung gemeinsamer lokaler NTP-Server als ausreichend präzise eingestuft, da die Zeiten der Hosts in aller Regel auf Millisekunden genau übereinstimmen, die Messungen aber nur sekündlich erfolgten. Die Annahme wurde weiter validiert durch die Prüfung der Daten, die diesbezüglich keine Inkonsistenzen aufzeigen.

Zusammenfassend lässt sich festhalten, dass die herausgearbeiteten Ergebnisse aufgrund der genannten Einschränkungen zwar nicht in ihrer absoluten Form auf andere Untersuchungen übertragbar sind, die grundlegenden Erkenntnisse aber dennoch als valide und als eine für die Praxis relevante Orientierungshilfe einzuordnen sind.

7 Fazit

In der Einleitung zu dieser Arbeit wurde die Notwendigkeit herausgearbeitet, die Energieeffizienz von CI/CD-Pipelines zu untersuchen, da sie einerseits durch die häufige Ausführung spürbar zum Energieverbrauch von Servern beitragen, aber andererseits ein hohes Optimierungspotenzial bergen. Aus diesem Grund wurden die Auswirkungen der beiden Best Practices Dependency-Caching und Parallelisierung auf den Energieverbrauch einer exemplarischen Pipeline analysiert. Die Prüfung der eingangs aufgestellten Hypothesen ergibt, dass die kombinierte Anwendung beider Best Practices die effizienteste Strategie darstellt, um die Ressourcen optimal auszunutzen und den Energieverbrauch und die Laufzeit von CI/CD-Pipelines zu reduzieren.[1]

Die Ergebnisse bestätigen die erste Hypothese, wonach der Aufbau eines Dependency-Caches den Ressourcenverbrauch temporär steigert, sich dieser Mehraufwand durch die nachfolgenden Nutzungen jedoch amortisiert. Pipeline-Durchläufe mit Cache-Nutzung weisen eine signifikant geringere Laufzeit und einen signifikant reduzierten Energieverbrauch gegenüber Cache-aufbauenden Pipeline-Durchläufen sowie Implementierungen ohne Caching auf.
Ebenso wird die zweite Hypothese gestützt, da die Parallelisierung von Pipeline-Jobs zwar kurzfristig zu höheren Spitzenauslastungen von CPU und RAM führt, aber dennoch den Gesamtenergieverbrauch und die Laufzeit deutlich senkt.
Die dritte Hypothese, welche die höchste Effizienz durch die kombinierte Anwendung beider Praktiken postuliert, wird ebenfalls bestätigt. Das Zusammenspiel von Parallelisierung und Dependency-Caching bringt Synergieeffekte hervor, sodass durch die Parallelisierung eine Cache-aufbauende Pipeline, indem Caches job-übergreifend innerhalb eines Durchlaufs weiterverwendet werden, bereits energetisch und zeitlich effizienter ist als eine Pipeline mit nur Parallelisierung. Diese Effekte werden durch anschließende Pipeline-Durchläufe mit Parallelisierung und ausschließlicher Cache-Nutzung nochmals deutlich übertrumpft.

[1] Der im Rahmen dieser Arbeit erstellte Datensatz ist zur weiteren Nutzung und Nachvollziehbarkeit öffentlich zugänglich gemacht worden: `https://doi.org/10.5281/zenodo.17175371`.

Die hiesige Arbeitet leistet einen Beitrag zur Forschung im Bereich *Green in ICT*, indem sie die relativen Einsparpotenziale etablierter Optimierungsstrategien in CI/CD-Pipelines quantifiziert. Sie verdeutlicht, dass einfache, gezielte Anpassungen der Pipeline-Konfiguration zu erhebliche Effizienzgewinnen führen. Darüber hinaus trägt sie dazu bei, die versteckten Energieverbräuche in der Softwareentwicklung sichtbarer zu machen.

Aus der Arbeit ergeben sich verschiedene weiterführende Forschungsvorhaben. Künftige Forschungsarbeiten könnten untersuchen, inwiefern sich eine dynamische Ressourcenzuweisung auf das Energiesparpotenzial auswirkt. Eine Erweiterung der Analyse auf Netzwerklasten und den Energieverbrauch von Remote-Prozessen würde zudem eine umfassendere Energiebilanz von Pipelines ermöglichen. Zur Steigerung der Messgenauigkeit könnte die modellbasierte Energiemessung durch direkt erfasste Hardwaremessungen ersetzt werden. Zuletzt könnte eine qualitative Untersuchung Aufschluss über die Motive von Entwickler:innen bezüglich der (Nicht-)Anwendung von Best Practices geben, mit dem Ziel, Implementierungshemmnisse zu minimieren.

Literaturverzeichnis

[1] R. Abdalkareem et al. „Which Commits Can Be CI Skipped?“ In: *IEEE Transactions on Software Engineering* 47.3 (März 2021), S. 448–463. DOI: 10.1109/TSE.2019.2897300.

[2] S. Ayers et al. *Measuring the Emissions and Energy Footprint of the ICT Sector: Implications for Climate Action*. Hrsg. von World Bank Group und International Telecommunication Union. Washington D. C., 2024. ISBN: 978-92-61-38541-5. URL: http://documents.worldbank.org/curated/en/099121223165540890.

[3] I. Bouzenia und M. Pradel. „Resource Usage and Optimization Opportunities in Workflows of GitHub Actions“. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. New York, NY, USA: Association for Computing Machinery, Feb. 2024, S. 1–12. DOI: 10.1145/3597503.3623303.

[4] Bundesamt für Sicherheit in der Informationstechnik. *Definition Eines Rechenzentrums*. URL: https://www.bsi.bund.de/dok/RZ-Definition (besucht am 20. 05. 2025).

[5] G. Calandrini et al. „Power Measurement Methods for Energy Efficient Applications“. In: *Sensors* 13.6 (Juni 2013), S. 7786–7796. DOI: 10.3390/s130607786.

[6] C. Calero, M. A. Moraga und M. Piattini. „Introduction to Software Sustainability“. In: *Software Sustainability*. Cham: Springer International Publishing, 2021, S. 1–15. DOI: 10.1007/978-3-030-69970-3_1.

[7] H. Claßen et al. *Carbon-Awareness in CI/CD*. Okt. 2023. DOI: 10.48550/arXiv.2310.18718.

[8] T. Coleman et al. „Evaluating Energy-Aware Scheduling Algorithms for I/O-Intensive Scientific Workflows“. In: *Computational Science – ICCS 2021*. Hrsg. von M. Paszynski et al. Bd. 12742. Cham: Springer International Publishing, 2021, S. 183–197. DOI: 10.1007/978-3-030-77961-0_16.

[9] G. Da Costa, J.-M. Pierson und L. Fontoura-Cupertino. „Mastering System and Power Measures for Servers in Datacenter“. In: *Sustainable Computing: Informatics and Systems* 15 (Sep. 2017), S. 28–38. DOI: 10.1016/j.suscom.2017.05.003.

[10] Docker Inc. *OverlayFS Storage Driver*. Docker Documentation. URL: https://docs.docker.com/engine/storage/drivers/overlayfs-driver/ (besucht am 15. 09. 2025).

[11] L. Dodd und B. Noll. *State of CI/CD Report 2024: The Evolution of Software Delivery Performance*. Report. Apr. 2024. URL: https://cd.foundation/state-of-cicd-2024/ (besucht am 24. 03. 2025).

[12] M. Dorier et al. „On the Energy Footprint of I/O Management in Exascale HPC Systems“. In: *Future Generation Computer Systems* 62 (Sep. 2016), S. 17–28. DOI: 10.1016/j.future.2016.03.002.

[13] K. Eder et al. „Energy-Aware Software Engineering“. In: *ICT - Energy Concepts for Energy Efficiency and Sustainability*. IntechOpen, März 2017. DOI: 10.5772/65985.

[14] F. M. A. Erich, C. Amrit und M. Daneva. „A Qualitative Study of DevOps Usage in Practice“. In: *Journal of Software: Evolution and Process* 29.6 (2017), e1885. DOI: 10.1002/smr.1885.

[15] G. Fagas et al. „Energy Challenges for ICT“. In: *ICT - Energy Concepts for Energy Efficiency and Sustainability*. IntechOpen, März 2017. DOI: 10.5772/66678.

[16] J. Fairbanks, A. Tharigonda und N. U. Eisty. „Analyzing the Effects of CI/CD on Open Source Repositories in GitHub and GitLab“. In: *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA)*. Mai 2023, S. 176–181. DOI: 10.1109/SERA57763.2023.10197778.

[17] R. Ferreira Da Silva et al. „Accurately Simulating Energy Consumption of I/O-Intensive Scientific Workflows“. In: *Computational Science – ICCS 2019*. Hrsg. von J. M. F. Rodrigues et al. Bd. 11536. Cham: Springer International Publishing, 2019, S. 138–152. DOI: 10.1007/978-3-030-22734-0_11.

[18] T. Fischbach, E. Kieffer und P. Bouvry. *Challenges in Automatic Software Optimization: The Energy Efficiency Case*. Mai 2023. DOI: 10.48550/arXiv.2305.06397.

[19] B. Fitzgerald und K.-J. Stol. „Continuous Software Engineering: A Roadmap and Agenda“. In: *Journal of Systems and Software* 123 (Jan. 2017), S. 176–189. DOI: 10.1016/j.jss.2015.06.063.

[20] K. Gallaba, M. Lamothe und S. McIntosh. „Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI“. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. New York, NY, USA: Association for Computing Machinery, Juli 2022, S. 1330–1342. DOI: 10.1145/3510003.3510211.

[21] K. Gallaba et al. „Accelerating Continuous Integration by Caching Environments and Inferring Dependencies“. In: *IEEE Transactions on Software Engineering* 48.6 (Juni 2022), S. 2040–2052. DOI: 10.1109/TSE.2020.3048335.

[22] S. J. Gami, C. R. Katru und K. N. Shah. „Enhancing Software Reliability: The Role of Automated Continuous Integration and Continuous Delivery“. In: *International Journal of Computer Applications* 187.1 (Mai 2025), S. 57–62. DOI: 10.5120/ijca202592478.

[23] E. Gelenbe. „Electricity Consumption by ICT: Facts, Trends, and Measurements“. In: *Ubiquity* 2023.August (Aug. 2023), 1:1–1:15. DOI: 10.1145/3613207.

[24] S. Georgiou, S. Rizou und D. Spinellis. „Software Development Lifecycle for Energy Efficiency: Techniques and Tools“. In: *ACM Comput. Surv.* 52.4 (Aug. 2019), 81:1–81:33. DOI: 10.1145/3337773.

[25] GitHub.com. *Build Software Better, Together*. URL: https://github.com (besucht am 06. 05. 2025).

[26] GitLab.com. *Caching in GitLab CI/CD*. GitLab Documentation. GitLab Docs. URL: https://docs.gitlab.com/ci/caching/ (besucht am 15. 09. 2025).

[27] GitLab.com. *Docker executor*. GitLab Documentation. GitLab Docs. URL: https://docs.gitlab.com/runner/executors/docker/ (besucht am 15. 09. 2025).

[28] Gitlab.com. *About GitLab*. URL: https://about.gitlab.com/company/ (besucht am 05. 06. 2025).

[29] M. Golzadeh, A. Decan und T. Mens. „On the Rise and Fall of CI Services in GitHub“. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. März 2022, S. 662–672. DOI: 10.1109/SANER53432.2022.00084.

[30] Green Coding Solutions GmbH. *Cloud-Energy*. Green Coding Solutions GmbH. März 2025. URL: https://github.com/green-coding-solutions/cloud-energy (besucht am 14. 04. 2025).

[31] Green Coding Solutions GmbH. *Eco CI*. URL: https://www.green-coding.io/products/eco-ci/ (besucht am 24. 03. 2025).

[32] B. Gregg. *Systems Performance: Enterprise and the Cloud*. Second edition. Addison-Wesley Professional Computing Series. Boston, Columbus, New York, San Francisco, Amsterdam, Cape Town, Dubai, London, Madrid, Milan, Munich: Addison-Wesley, 2021. ISBN: 978-0-13-682015-4.

[33] R. Grünwald und C. Caviezel. *Energieverbrauch der IKT-Infrastruktur. Endbericht zum TA-Projekt*. Techn. Ber. Büro für Technikfolgen-Abschätzung beim Deutschen Bundestag (TAB), 2022. DOI: 10.5445/IR/1000151164.

[34] A. Guldner et al. „Development and Evaluation of a Reference Measurement Model for Assessing the Resource and Energy Efficiency of Software Products and Components—Green Software Measurement Model (GSMM)“. In: *Future Generation Computer Systems* 155 (Juni 2024), S. 402–418. DOI: 10.1016/j.future.2024.01.033.

[35] L. M. Hilty und B. Aebischer. „ICT for Sustainability: An Emerging Research Field“. In: *ICT Innovations for Sustainability*. Hrsg. von L. M. Hilty und B. Aebischer. Bd. 310. Cham: Springer International Publishing, 2015, S. 3–36. DOI: 10.1007/978-3-319-09228-7_1.

[36] R. Hintemann et al. *Neue Energiebedarfe Digitaler Technologien – Untersuchung von Schlüsseltechnologien Für Die Zukünftige Entwicklung Des IKT-bedingten Energiebedarfs*. Techn. Ber. Deutsche Energie-Agentur, Sep. 2023.

[37] International Energy Agency. *Energy and AI*. Report. Paris, 2025. URL: https://www.iea.org/reports/energy-and-ai.

[38] International Energy Agency et al. *Tracking SDG7: The Energy Progress Report*. Report. Washington DC, 2024. URL: https://www.iea.org/reports/tracking-sdg7-the-energy-progress-report-2024,.

[39] International Renewable Energy Agency. *Tracking COP28 Outcomes: Tripling Renewable Power Capacity by 2030*. Report Summary. 2024. URL: https://www.irena.org/Digital-Report/Tracking-COP28-outcomes-Tripling-renewable-power-capacity-by-2030 (besucht am 18. 09. 2025).

[40] X. Jin und F. Servant. *What Helped, and What Did Not? An Evaluation of the Strategies to Improve Continuous Integration*. Feb. 2021. DOI: 10.48550/arXiv.2102.06666.

[41] S. Jones und K. Korakitis. *State of Continuous Delivery Report Report: The Evolution of Software Delivery Performance*. Report. Juni 2022. URL: https://cd.foundation/state-of-cd-june-2022/ (besucht am 06. 09. 2025).

[42] A. Kasoju und T. Vishwakarma. „The Role of Continuous Integration and Deployment in Improving Software Quality“. In: 7.5 (Mai 2025), S. 19–30. ISSN: 2360-821X. URL: https://www.ajmrd.com/vol-7-issue-5/.

[43] K. N. Khan et al. „RAPL in Action: Experiences in Using RAPL for Power Measurements“. In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (März 2018), 9:1–9:26. DOI: 10.1145/3177754.

[44] A. Kruglov, G. Succi und G. Dlamini. „System Energy Consumption Measurement“. In: *Developing Sustainable and Energy-Efficient Software Systems*. Hrsg. von A. Kruglov und G. Succi. Cham: Springer International Publishing, 2023, S. 27–38. DOI: 10.1007/978-3-031-11658-2_3.

[45] N. Limbrunner. „Dynamic Macro to Micro Scale Calculation of Energy Consumption in CI/CD Pipelines“. Masterarbeit. Stockholm, Schweden: School of Electrical Engineering und Computer Science, 2023. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-337164.

[46] J. Mathew und D. S R. *Enhancing DevOps Pipeline Efficiency Through Modern Practices*. SSRN Scholarly Paper. Rochester, NY, Feb. 2025. DOI: 10.2139/ssrn.5143363.

[47] G. Maudoux und K. Mens. „Bringing Incremental Builds to Continuous Integration“. In: *SaTToSE – Seminar Series on Advanced Techniques & Tools for Software Evolution*. 2017. URL: http://hdl.handle.net/2078.1/189543.

[48] R. Muralidhar, R. Borovica-Gajic und R. Buyya. „Energy Efficient Computing Systems: Architectures, Abstractions and Modeling to Techniques and Standards“. In: *ACM Comput. Surv.* 54.11s (Sep. 2022), 236:1–236:37. DOI: 10.1145/3511094.

[49] K. Nayak et al. „Sustainable Continuous Testing in DevOps Pipeline“. In: *2024 1st International Conference on Communications and Computer Science (InCCCS)*. Mai 2024, S. 1–6. DOI: 10.1109/InCCCS60947.2024.10593566.

[50] J. Painter. *Practical GitLab Services: A Complete DevOps Guide for Developers and Administrators*. Berkeley, CA: Apress, 2024. DOI: 10.1007/979-8-8688-0427-4.

[51] Q. Perez et al. *Software Frugality in an Accelerating World: The Case of Continuous Integration*. Okt. 2024. DOI: 10.48550/arXiv.2410.15816.

[52] P. F. Popiolek und O. M. Mendizabal. „Monitoring and Analysis of Performance Impact in Virtualized Environments“. In: *Journal of Applied Computing Research* 2.2 (2012), S. 75–82. DOI: 10.4013/jacr.2012.22.03.

[53] A. Proulx et al. *Problems and Solutions of Continuous Deployment: A Systematic Review*. Dez. 2018. DOI: 10.48550/arXiv.1812.08939.

[54] N. Rteil et al. „Interact: IT Infrastructure Energy and Cost Analyzer Tool for Data Centers“. In: *Sustainable Computing: Informatics and Systems* 33 (Jan. 2022), S. 100618. DOI: 10.1016/j.suscom.2021.100618.

[55] D. S R und J. Mathew. „Optimizing Continuous Integration and Continuous Deployment Pipelines with Machine Learning: Enhancing Performance and Predicting Failures“. In: *Advances in Science and Technology Research Journal* 19.3 (März 2025), S. 108–120. DOI: 10.12913/22998624/197406.

[56] G. Schomaker, S. Janacek und D. Schlitt. „The Energy Demand of Data Centers“. In: *ICT Innovations for Sustainability*. Hrsg. von L. M. Hilty und B. Aebischer. Bd. 310. Cham: Springer International Publishing, 2015, S. 113–124. DOI: 10.1007/978-3-319-09228-7_6.

[57] M. Shahin, M. Ali Babar und L. Zhu. „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“. In: *IEEE Access* 5 (2017), S. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.

[58] S. Tatineni. „Optimizing Continuous Integration and Continuous Deployment Pipelines in DevOps Environments“. In: *International Journal of Computer Engineering and Technology (IJCET)* 13.3 (Dez. 2022), S. 95–101. ISSN: 0976-6375. URL: https://iaeme.com/Home/issue/IJCET?Volume=13&Issue=3.

[59] TOP500. *November 2021*. URL: https://top500.org/lists/green500/2021/11/ (besucht am 22. 05. 2025).

[60] TOP500. *November 2024*. URL: https://top500.org/lists/green500/2024/11/ (besucht am 22. 05. 2025).

[61] United Nations Development Programme. *Goal 7: Affordable and Clean Energy*. URL: https://www.undp.org/sustainable-development-goals/affordable-and-clean-energy (besucht am 08. 04. 2025).

[62] M. van Belzen, J. Trienekens und R. Kusters. „Critical Success Factors of Continuous Practices in a DevOps Context“. In: *Information Systems Development: Information Systems Beyond 2020 (ISD2019 Proceedings)* (2019). Hrsg. von A. Siarheyeva et al. URL: https://aisel.aisnet.org/cgi/viewcontent.cgi?article=1278&context=isd2014.

A Anhang

A.1 Statistische Übersicht über die Szenarien

Tabelle A.1 listet die statistischen Kenngrößen Mittelwert (MW), Median (MD), Minimum (Min), Maximum (Max) und Standardabweichung (SD) je Szenario und über aller Pipeline-Durchläufe aggregiert auf. Die Kenngrößen für RAM und CPU basieren auf den Mittelwerten der Pipeline-Durchläufe.

Tabelle A.1: Übersicht der statistischen Kenngrößen je Szenario und Metrik (aggregiert pro Pipeline-Durchlauf).

Szenario	Statistik	RAM [%]	CPU [%]	I/O [MiB]	Energie [Wh]	Laufzeit [s]
Basis (N=58)						
	MW	17,06	40,75	1722,56	1,45	161,28
	Md	17,06	40,70	1716,44	1,45	162,00
	Min	16,73	38,57	1582,91	1,36	152,00
	Max	17,33	44,16	1813,38	1,54	169,00
	SD	0,14	1,33	50,22	0,05	3,31
Cache-Aufbau (N=46)						
	MW	17,00	44,11	1976,86	1,52	160,09
	Md	16,94	44,48	1973,90	1,52	160,00
	Min	16,65	41,96	1827,05	1,43	154,00
	Max	17,59	46,69	2096,34	1,61	166,00
	SD	0,24	1,16	67,58	0,04	3,22
Cache-Nutzung (N=46)						
	MW	17,16	44,70	1737,08	1,34	139,35
	Md	17,11	45,21	1732,67	1,34	140,00
	Min	16,84	41,16	1577,30	1,22	134,00
	Max	17,72	46,93	1861,88	1,42	147,00
	SD	0,24	1,57	67,61	0,05	3,02
Parallel (N=57)						
	MW	17,40	42,61	1818,35	1,26	134,12
	Md	17,41	42,18	1820,19	1,25	134,00
	Min	16,92	39,21	1670,90	1,18	131,00
	Max	17,67	46,21	1984,76	1,36	139,00
	SD	0,15	1,73	70,54	0,05	2,10
Parallel-Aufbau (N=53)						
	MW	17,24	43,41	1946,21	1,24	129,19
	Md	17,23	42,23	1943,51	1,23	129,00
	Min	16,84	40,53	1852,64	1,14	122,00
	Max	17,62	49,17	2029,23	1,43	140,00
	SD	0,19	2,57	46,37	0,06	3,71
Parallel-Nutzung (N=53)						
	MW	17,48	45,77	1683,93	1,11	112,36
	Md	17,46	44,90	1680,53	1,09	112,00
	Min	17,11	41,70	1551,89	1,04	108,00
	Max	17,85	51,97	1786,21	1,21	121,00
	SD	0,20	2,45	55,68	0,05	3,01

Erklärung zur selbständigen 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