Bachelorarbeit

Skalierbares Lernen in der Cloud

Scalable Learning in the Cloud

von
Judith Herrmann

Betreuung
Dr. Michael Perscheid
Dr. Rainer Schlosser
Johannes Huegle
Alexander Kastius
Enterprise Platform and Integration Concepts

Hasso-Plattner-Institut an der Universität Potsdam

30. Juni 2022

Zusammenfassung

Cloud-Umgebungen gewinnen immer mehr an Bedeutung. Gerade im Bereich des Machine Learnings bieten sie die Möglichkeit, den Lernprozess effizienter zu gestalten. Die bessere Hardware, die einfache und andauernde Verfügbarkeit macht diese entfernten Rechner für moderne Systeme unverzichtbar. Von großen Anbietern wird oft eine Komplettlösung verkauft, die alle Werkzeuge beinhaltet, die für Forschende zum Arbeiten notwendig sind. Dabei sind jedoch nur einige wenige auf Reinforcement Learning spezialisiert. In dieser Arbeit wird eine Cloud-Architektur vorgestellt, mit der Reinforcement Learning Agenten trainiert werden können. Sie basiert auf einer Marktsimulation, in der diese eine möglichst optimale Preisstrategien lernen sollen. Die Anforderungen und die Umsetzungen werden thematisiert. Außerdem wird die Skalierbarkeit der Architektur anhand durchgeführter Experimente mit der Marktsimulation gezeigt.

Abstract

Cloud environments are becoming more and more important. Especially in the field of machine learning, they offer the possibility of making the learning process more efficient. The better hardware, the easy and continuous availability makes these remote computers indispensable for modern systems. Large providers often sell a complete solution that includes all the tools researchers need to work. However, only a few specialise in reinforcement learning. This paper presents a cloud architecture that can be used to train reinforcement learning agents. It is based on a market simulation in which the agents can learn the best possible pricing strategies. The requirements and the implementation are discussed. In addition, the scalability of the architecture will be shown on the basis of experiments carried out with the market simulation.

Inhaltsverzeichnis

1 Hintergrund

In diesem Kapitel soll zunächst die Nutzung der Cloud im Reinforcement Learning (RL) Bereich motiviert werden. Dazu wollen wir uns kurz grundlegende RL Konzepte anschauen. Anschließend werden wir in die Problemdomäne des Anwendungsfalls einsteigen. Mit dieser hat sich das Bachelorprojekt *Online Marketplace Simulation a Testbed for Self-Learning Agents in Recommerce* im Wintersemester 2021/22 und Sommersemester 2022 am Enterprise Platform and Integration Concepts Lehrstuhl des Hasso-Plattner-Instituts an der Universität Potsdam beschäftigt. Es wird die von dem Team entwickelte Marktsimulation kurz vorgestellt und gezeigt, warum dafür die Nutzung einer Cloud Infrastruktur sinnvoll erscheint.

1.1 Reinforcement Learning

Aus Fehlern lernt man. Das lehrt uns ein altes deutsches Sprichwort schon seit vielen Jahren. Es eignet sich jedoch auch, um ein grundlegendes Konzept von Reinforcement Learning zu verstehen. Reinforcement Learning ist eine KI Technologie, die aus 'Erfahrung' lernt. Abbildung 1.1 zeigt, wie dies funktioniert. Ein sogenannter Agent wird mit einer, ihm zunächst unbekannten Umgebung konfrontiert. In jeder Runde muss er Aktionen ausführen. Die Entscheidung, welche Aktion er ausführt, basiert er auf dem eingegebenen Zustand der Umgebung. Wie gut diese Aktion war, wird dem Agenten am Ende der Runde zusammen mit dem neuen Zustand der Umgebung übergeben. Dies wird über den sogenannten Reward abgebildet. Ziel für den Agenten ist es, eine möglichst gute Strategie zu erlernen, mit der dieser Reward maximiert werden kann. Dabei muss er zunächst nach dem *Trial-and-Error-Prinzip* vorgehen, da ihm nichts über die Umgebung bekannt ist. Im Laufe der Zeit kann der Agent jedoch besser werden, indem er aus den Ergebnissen vergangener Entscheidungen lernt. Für den Prozess der Entscheidungsfindung gibt es verschiedene Algorithmen, die genau definieren, wie der Agent agieren soll. Mehr Informationen siehe [30].

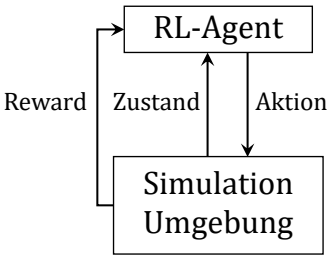


Abbildung 1.1: Grundlegendes RL Konzept

Allen Algorithmen ist jedoch gleich, dass sie tausende von Beispieldaten benötigen, bis sie eine performante Strategie erlernt haben. Um eine so große Menge an Daten produzieren zu können, trainieren RL-Agenten vorwiegend in einer Simulation. In dieser können die Agenten ihre Strategien durch iteratives Ausprobieren und Feedback bekommen erlernen. Die RL-Algorithmen können über sogenannte Hyperparameter von Forschenden konfiguriert werden. Diese können z.B. angeben, wie stark der Agent zukünftige Rewards kurzfristigen Rewards vorzieht, oder wie viel Prozent der Entscheidungen zufällig getroffen werden [30].

Abhängig vom gewählten Algorithmus haben diese Parameter unterschiedlichen Einfluss auf den Lernprozess. Die Wahl der richtigen Hyperparameter ist für Forschende, im RL-Bereich eine große Herausforderung. Siehe [19]. Ziel dabei ist es, immer eine Kombination zu

finden, mit der der Agent eine Strategie erlernen kann, die einen möglichst großen Reward erzielt. Ist eine solche Parameterkombination gefunden, so muss diese durch mehrfache Ausführung der gleichen Kombination verifiziert werden. Dies ist notwendig, da die RL-Agenten einen Teil ihrer Entscheidungen zufällig treffen. Es muss demnach sichergestellt werden, dass diese Ergebnisse nicht nur zufälligerweise gut waren.

Die Verarbeitung dieser großen Menge an Daten und die Mehrfachausführung des Trainings bedarf viel Zeit und Rechenleistung, siehe [13]. Der Zeitaufwand kann mit besserer Hardware verringert werden [1]. Hierbei kann eine Cloud-Umgebung für die Berechnungen helfen. Entfernte Server können Tag und Nacht laufen. Außerdem bieten sie oftmals eine bessere Rechenleistung als der eigene PC oder Laptop. Forschenden wird so die Möglichkeit des effizienten Arbeitens gewährt. Eine weitere Verbesserung wird geboten, wenn mehrere Trainingsdurchläufe parallel gestartet werden können. Eine geeignete Cloud-Architektur sollte demnach skalierbar sein.

Diese Arbeit schlägt eine solche Architektur vor. Entworfen ist sie für einen speziellen Anwendungsfall von Reinforcement Learning. Dem, vom Bachelorprojekt entwickeltem, `recommerce` Framework, entstammt die für RL genutzte Simulation. Das Framework dient Forschenden dazu, Marktprozesse und Preisfindungsstrategien in verschiedenen synthetischen Märkten zu studieren. Besonderes Augenmerk legt das Framework auf die Simulation sogenannter Kreislaufwirtschaften. Warum genau diese im Fokus stehen, und wie die Simulation funktioniert wird in den folgenden Abschnitten erklärt. Ziel ist es einen Eindruck zu gewinnen, vor welchen Herausforderungen die Agenten in den gegebenen Szenarien stehen.

1.2 Kreislaufwirtschaft

Lebten alle so, wie die Menschen in Deutschland, hätten wir im Jahr 2022 ab dem 04. Mai eine neue Erde benötigt. Die Ressourcen dieser Erde wären ab diesem Tag aufgebraucht. Siehe [71]. Auch wenn die Daten durch die COVID-19 Pandemie beeinträchtigt und eventuell verschoben sind, ist dennoch klar, dass die Bedeutung von Nachhaltigkeit weiter steigen muss. Gerade Nachhaltigkeit im Einzelhandel ist dabei ein wichtiger Aspekt [28]. Eine Möglichkeit, dem Problem der Rohstoffknappheit zu begegnen, ist die Kreislaufwirtschaft.

Oft wird sie unterschiedlich definiert, es liegen aber immer die gleichen Grundkonzepte vor. Siehe [2]. Ein Prinzip ist es Müll und Verschmutzung zu reduzieren. Anstatt Produkte in den Müll zu werfen sind sie so lange wie möglich zu gebrauchen. Das besagt das zweite Prinzip. Wenn Materialien nicht mehr weitergenutzt werden können, so besagt das dritte Prinzip, sind diese zu recyclen. So können aus den Rohstoffen, die in diesen Produkten stecken, neue Produkte gewonnen werden. Der Begriff Kreislaufwirtschaft ist dabei sehr weit gefasst und bezieht sich z.B. auch auf die Wiederaufbereitung von Müll.

Die Idee der zirkulären Wirtschaft ist nicht neu. Forschende haben Spuren erster Kompostieranlagen gefunden, die sich auf ca. 6000 bis 8000 v.Chr. einordnen lassen, siehe [3]. Auch Deutschland hat bereits 1996 ein Kreislaufwirtschafts- und Abfallgesetz (Closed Substance Cycle Waste Management Act) [3] beschlossen. Das Gesetz wurde seitdem immer wieder angepasst und regelt die Abfallwiederverwertung in Deutschland noch immer

Heutzutage gilt es, die Menge an produziertem Abfall stark zu verringern. Daher ist es für viele Kunden wichtig z.B. ihre Kleidung aus zweiter Hand zu kaufen. Unternehmen entdecken diese Art von Nachhaltigkeit ebenfalls für sich und integrieren einen Rückverkaufskanal in ihre Geschäftsprozesse. So bieten Konzerne wie z.B. Zalando [74], C&A [63] oder The North

Face [67, 65] ihren Kunden an, die genutzten Produkte zurückzunehmen. Die Entlohnung erfolgt meist mit Gutscheinen, für das jeweilige Unternehmen, oder kleineren Geldbeträgen.

Wie so ein Markt grundsätzlich funktioniert ist in Abbildung 1.2 gezeigt. Die Verkaufenden (Agenten) bieten das gleiche Produkt in zwei verschiedenen Produktlinien an: neu und gebraucht. Die Kunden können sich entscheiden, ob sie das gebrauchte (eventuell wieder aufbereitete) oder das neue Produkt kaufen. Haben sie keine Verwendung mehr für ihre gekaufte Ware, so können sie die Produkte wieder zurückverkaufen. Der Verkaufende muss dann die Wiederaufarbeitung des Artikels veranlassen, wenn dies beschädigt ist. Andere Kunden können es anschließend als gebrauchtes Produkt kaufen. Ist eine weitere Nutzung des Produktes nicht mehr wirtschaftlich vertretbar, so müssen die Bestandteile des Produkts nach Möglichkeit recycelt werden. Ziel ist es, dass nur wenige bis gar keine neuen Rohstoffe in diesen Zyklus kommen müssen, um so die auf der Erde natürlich vorkommenden Rohstoffe zu erhalten, weitere Informationen siehe [28].

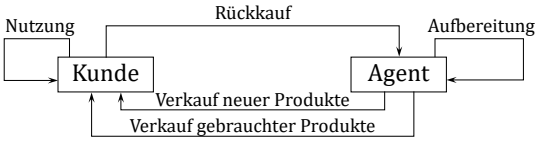


Abbildung 1.2: Durch die Simulation abgebildete Kreislaufwirtschaft

Solche Märkte sind sehr komplex. Gerade die Preisfindung wird immer schwieriger, je mehr Produkte angeboten werden sollen und je öfter die Preise geändert werden können. Pro Produkt muss der Händler drei Preise setzen: den Neupreis, den Gebrauchtpreis und den Rückkaufpreis. Alle drei Preise beeinflussen sich gegenseitig. Hierbei kann es zu sogenannten Kannibalisierungseffekten kommen, wodurch bspw. ein zu hoher Neupreis im Gegensatz zum Gebrauchtpreis den Verkauf von Neuprodukten hemmt.

Hinzu kommt, dass die meisten Unternehmen inzwischen auch im Online-Handel unterwegs sind [10]. Dort ändern sich Preise regelmäßig und die Verkaufenden werden dazu gezwungen ihre Preise immer häufiger anzupassen. Bieten die Unternehmen zusätzlich den Rückverkaufskanal an, so wird die Preisfindung für sie vielschichtiger. Online Märkte mit Rückkauf bezeichnen wir auch als Recommerce. Gerade in einer solchen komplexen Umgebung ist es wünschenswert, verschiedene Preisstrategien ausprobieren zu können, ohne den Profit des Unternehmens zu beeinflussen. Einen Preis zu setzen, der dann nicht die gewünschten Verkaufszahlen bringt, kann jedoch auf einem echten Markt gravierende Folgen haben. Hier kann eine möglichst reale Marktsimulation helfen.

1.3 Marktsimulation

In diesem Abschnitt wird die von dem Bachelorprojekt entwickelte Marktsimulation des `recommerce` Frameworks vorgestellt. Diese Simulation ist eine synthetisch entwickelte Marktumgebung, weshalb einige Marktprozesse von der Realität teilweise abstrahiert sind. Schulze Tast untersucht, wie die Simulation mit Zuhilfenahme von Marktdaten aus der realen Welt verbessert werden kann [23]. Im Framework existieren verschiedene Marktszenarien. Zwei der Szenarien sind lineare Marktplätze, also Marktplätze, in denen ein Produkt nur einmal verkauft und nach dem Gebrauch entsorgt wird. Diese eignen sich, um das grundlegende Konzept der Simulation zu verstehen und zu zeigen, wie viele Ressourcen verbraucht würden, wenn dies die hauptsächliche Wirtschaftsform bliebe.

Den größeren Teil nehmen die nachhaltigeren Recommerce Märkte ein. Grundlegend funktioniert die Simulation dabei wie folgt: In jeder simulierten Runde kommen Kunden, die sich entscheiden können, ob sie eines der angebotenen Produkte kaufen. Gibt es mehrere Verkaufende, können sie sich zudem noch entscheiden, bei wem sie dieses kaufen wollen. Kunden können dabei unterschiedliches Kaufverhalten haben. Eine bestimmte Anzahl an Kunden möchte außerdem, in jeder Runde, ein Produkt an einen Verkaufenden zurückverkaufen. (Genauere Analyse der Marktprozesse: [4])

Die Recommerce Märkte gibt es in zwei Varianten. In der einen Variante wird mit einem Rückkaufpreis gearbeitet. Wie oben beschrieben kauft der Verkaufende die genutzten Produkte für kleines Geld dafür von den Kunden zurück. Die zweite Variante ist ein Spezialfall der ersten Variante. Bei dieser beträgt der Rückkaufpreis null. Beide Varianten gibt es in drei verschiedenen Ausprägungen: Monopol, Duopol, Oligopol. Beim Monopol tritt nur ein Verkaufender (Agent) an, beim Duopol zwei und beim Oligopol im Allgemeinen einige wenige. Für jeden Markt gibt es verschiedene Agenten, die eingesetzt werden können, z.B. regelbasierte Agenten. Darüber hinaus existieren Agenten, die ihre Preise immer gleich setzen, und händisch steuerbare Agenten. (Für weitere Informationen zu den Agenten [4, 16]) Zu den möglichen einsetzbaren Agenten, zählen auch die oben beschriebenen RL-Agenten.

Diese können in der Simulation trainiert werden. In jeder Runde müssen sie dabei je nach Marktszenario die Preise für ihre Produkte setzen. In einer Kreislaufwirtschaft mit Rückkaufpreis beispielsweise ist die Aktion, die der Agent am Anfang jeder Runde ausführen muss, das Setzen der drei Preise (Neupreis, Gebrauchtpreis, Rückkaufpreis). Der Reward des Agenten berechnet sich aus der Menge der verkauften Produkte, der wieder eingekauften Produkte, der Lagerkosten und der Produktionskosten, die ein Verkaufender hat. Der Zustand des Marktes ist je nach Szenario und Ausprägung ein bisschen unterschiedlich. Er beinhaltet z.B. den Füllstand des eigenen Lagers und die Preise der konkurrierenden Agenten. Die Simulation des `recommerce` Frameworks gibt Forschenden die Möglichkeit, Preisstrategien in Kreislaufwirtschaften mit Reinforcement Learning zu erforschen. Um die Ausführung der Trainingsläufe effizienter zu machen, braucht es eine Cloud Architektur, die es ermöglicht mehrere parallele Trainingsläufe auf einer entfernten Maschine zu starten.

1.4 Gliederung

Diese Arbeit stellt eine Erweiterung des `recommerce` Frameworks um eine Netzwerkarchitektur vor. Zunächst erfolgt in Kapitel 2 eine Einordnung dieser Architektur. Dazu werden verschiedene Skalierbarkeitsansätze für Machine Learning Algorithmen vorgestellt. Anschließend werden wir in Kapitel 3 die Anforderungen an die Architektur bezogen auf die Simulation herausarbeiten. In Kapitel 4 wird eine Möglichkeit vorgestellt, diese Anforderungen umzusetzen. Kapitel 5 beschreibt die Technologien der genutzten Beispielimplementierung.

Wie bereits vorgestellt ist es für Forschende, die mit dem Framework arbeiten sehr wichtig, dass das Training gut skaliert. Daher wird die Skalierbarkeit der vorgestellten Architektur in Kapitel 6 untersucht. Ziel ist es über die Architektur mehrere Trainingsdurchläufe zu starten. Wir wollen ermitteln, wie die Ausführungsdauer von der Anzahl an parallelen Trainings abhängt und wie sich diese Abhängigkeit anhand des Ressourcenverbrauchs erklären lässt. Abgeschlossen wird die Arbeit in Kapitel 7 mit einem Ausblick über Erweiterungsmöglichkeiten der Netzwerkarchitektur und einer kurzen Zusammenfassung. Im Anhang befindet sich ein Glossar, welches einige der verwendeten Begriffe erklärt.

2 Einordnung

Der Umsatz und die Nachfrage für Cloud Computing ist in den letzten Jahren stark gestiegen. Eine Umfrage [66], vom April 2022, zeigt, dass die Bedeutung von *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)* oder *Software as a Service (SaaS)* auch weiterhin zunehmen wird. Die Verfügbarkeit und die größere Rechenleistung macht die Cloud auch für Machine Learning Prozesse attraktiv. Forschende beschäftigen sich damit, wie die Berechnungen besser skalieren können und wie die Cloud dabei genutzt werden kann. Als Cloud bezeichnen wir im Folgenden Rechen- und/oder Speicherkapazitäten auf einer oder mehreren entfernten Maschinen.

2.1 Skalierbare Machine Learning Frameworks

Für Machine Learning gibt es viele verschiedene Plattformen. Verbraeken [24] vergleicht einige dieser Frameworks. Er stellt dabei u.a. Python Bibliotheken wie `TensorFlow` [31] oder `PyTorch` [54] vor. Diese ermöglichen es dem Nutzenden, die Algorithmen lokal auf ihrem Gerät zu trainieren. Im Gegensatz dazu gibt es auch Frameworks, die sich speziell auf die Ausführung von verteiltem Machine Learning fokussieren.

Ein Ansatz dieses Problem zu lösen stellen Li et. al. [11, 12] vor. Mit Ihrem Framework, können Machine Learning Algorithmen verteilt lernen. Die Architektur besteht aus Parameterservern und Arbeiter-Knoten. Die Parameterserver beinhalten jeweils einen Teil der Parameter, die für das Training benötigt werden. Li et. al. gruppieren die Knoten und verteilen dann die Berechnungen so, dass der zu der Gruppe assoziierte Parameterserver, die entsprechend passenden Parameter gespeichert hat. In einer früheren Version des Framework wurden die Parameter in einem klassischen *Key-Value Store* gespeichert. Sie führen jedoch aus, dass das Speichern der Werte auch performanter geht. Dazu speichern sie die Werte in Parametermatrizen und Parametervektoren. Somit können Li et. al. eine bessere Skalierbarkeit erreichen.

Dieser Parameterserver ist imstande verschiedene Algorithmen gleichzeitig laufen zu lassen. Die gewünschte Parallelisierung wäre mit diesem Framework möglich. Jedoch hat sich dieses Forschungsteam auf Machine Learning Ansätze konzentriert, bei denen große Mengen an Daten bereits vor Beginn des Trainings vorhanden sind. Dies trifft auf das `recommerce` Framework nicht zu, da die Daten erst im Laufe einer ausgeführten Simulation entstehen.

Im Gegensatz zu dem vorgestellten Parameterserver, ist die Bibliothek `RLlib` [57] auf Reinforcement Learning spezialisiert. Liang et. al. [13] stellen in ihrer Arbeit diese Bibliothek vor. `RLlib` unterstützt sowohl `TensorFlow`, als auch `PyTorch`. Der Fokus wird hier darauf gelegt RL auf mehreren Knoten ausführen zu können. Dabei gibt es eine zentrale Instanz, die den Kontrollfluss überwacht und die Aufgaben delegiert. Je nach CPU bzw. GPU Auslastung kann diese Instanz dann Berechnungen auf andere Knoten verteilen. Liang et. al. schlagen dabei eine hierarchische Delegation der Aufgaben vor. Heißt, die zentrale Instanz vergibt die Arbeitslast an andere Instanzen, die wiederum die Arbeit weitergeben, bis sie nicht mehr

weiter geteilt werden kann. Für Nutzende der Bibliothek wird von dieser Verteilung der Berechnungen vollständig abstrahiert. Es kann über einen Parameter eingestellt werden, wie viele Knoten für die Verteilung genutzt werden.

`RLlib` parallelisiert auf einer anderen Ebene als der hier vorgestellte Ansatz für das `recommerce` Framework. Die Bibliothek parallelisiert die Berechnungen der RL-Algorithmen. Der Ansatz, den wir in dieser Arbeit untersuchen wollen, parallelisiert die gesamte Anwendung. Das heißt, mehrere Simulationen müssen gleichzeitig und isoliert voneinander laufen können.

2.2 Machine Learning in der Cloud

Verbraeken [24] stellt auch Machine Learning Plattformen vor, die in der Cloud laufen. Meist stellen große Unternehmen wie z.B. Amazon, Microsoft oder Google Hardware aber auch Software für die Ausführung von Machine Learning Algorithmen bereit. Bei `Jupyter Notebook` [45] beispielsweise wird die gesamte Umgebung, die zur Ausführung von Code notwendig ist, für Nutzende zur Verfügung gestellt. Auch für Frameworks, die darauf ausgelegt sind auf großen Computer Clustern zu laufen, wie z.B. `Apache Spark` gibt es Machine Learning Bibliotheken. `Apache Spark` nennt seine Bibliothek beispielsweise `MLlib` [56].

In der Literatur gibt es auch einige Arbeiten, die diese großen Frameworks nutzen und dann auf ihren Anwendungsfall anpassen, siehe [1]. Für die Erweiterung, des `recommerce` Frameworks, soll erreicht werden, dass diese von großer Infrastruktur unabhängig ist.

García et. al. [14] stellen einen solchen Ansatz vor. Sie haben sich zum Ziel gesetzt, *Machine Learning as a Service* anzubieten. Mit ihrem Framework `DEEP` bieten sie Entwickelnden eine Trainings-, sowie eine Ausführungsumgebung für Machine Learning Modelle. Die Modelle werden wie bei anderen vorgestellten Arbeiten, aus der vorherigen Sektion, über mehrere Knoten verteilt trainiert. Im Lernprozess müssen immer wieder Daten ausgetauscht werden. Für jeden Schritt dieses Austauschs haben sie ein *Application Programming Interface (API)* entwickelt. Anders als z.B. `RLib` sind hier die Komponenten untereinander für den Austausch und die Verteilung zuständig. Aufbauend auf diesem Framework implementieren Sassu et. al. [21] eine Anwendung zur Echtzeitanalyse von Video Streams.

`DEEP` bietet außerdem die Infrastruktur die gelernten Modelle in der Praxis einsetzen zu können. In dieser Arbeit werden wir diese Möglichkeit nicht betrachten. Forschende wollen die Modelle zwar meist auswerten, gleichzeitig benötigen sie aber kein laufendes, über eine *API* ansprechbares, vollständig trainiertes Modell, wie `DEEP` es bereitstellt. `DEEP` konzentriert sich auch auf Machine Learning Ansätze, die viele Daten für das Training benötigen. Diese Daten liegen bei Nutzung dieses Frameworks bereits zu Beginn eines Trainings vor. So haben sie z.B. Modelle auf Grundlage von Bilddaten von *iNaturalist* [68] mit dieser Architektur entwickelt. Auf den `recommerce` Anwendungsfall lässt sich dies nicht direkt übertragen, da die Simulation die Trainingsdaten dynamisch generiert. Die Agenten lernen, indem sie auf die Veränderungen reagieren.

Neben der Notwendigkeit RL Training zu unterstützen, soll die Architektur unabhängig von Infrastruktur Dritter sein. Auch die Menge an genutzten Knoten unterscheidet sich deutlich von den in diesem Kapitel betrachteten Ansätzen. Das Training wird hier nur auf einer Maschine ausgeführt. Daher werden wir die optimale Verteilung der Rechnungen nicht weiter betrachten.

3 Anforderungen

In der implementierten `recommerce` Simulation können lokal Verkaufsstrategien trainiert werden. Das Framework soll nun so erweitert werden, dass dies auch auf der remote Maschine möglich ist. Dafür wird eine passende Architektur benötigt. Die Menge an Funktionalität, die Nutzende lokal haben, soll auch auf der entfernten Maschine existieren. Daher muss vor dem Entwurf der Architektur analysiert werden, welche Anforderungen die Ausführung der Experimente mit dem Framework mit sich bringen. Dieses Kapitel legt die verschiedenen Anforderungen an die Simulation dar.

3.1 Konfiguration

Um ein effektives wissenschaftliches Arbeiten mit dem Framework zu ermöglichen, ist es sehr wichtig, dass Nutzende Experimente mit verschiedenen Konfigurationen durchlaufen lassen können. Dies dient auf der einen Seite dazu, gelernte RL-Strategien zu verifizieren (vgl. 1.1). Auf der anderen Seite soll das Framework Forschenden die Möglichkeit geben, verschiedene Verkaufsstrategien auf unterschiedlichen Märkten auszuprobieren.

Eine Konfiguration bezeichnet hier eine Menge von Konstanten, die zur Ausführung eines Experiments relevant sind. Große Teile der Simulation sollen durch `JSON` Dateien konfigurierbar sein. Eine solche Datei hat den Vorteil, dass nicht vor jedem Experiment der Code angepasst werden muss. Die notwendigen Konstanten werden einmal zu Beginn eines Simulationsdurchlaufs aus den Dokumenten gelesen und anschließend gesetzt.

Das Framework akzeptiert drei Dateien im `JSON` Format: `sim_market_config`, `rl_config` und die `environment_config`. In diesen Dateien stehen alle Konstanten, mit denen die Simulation beeinflusst werden kann. In welcher Weise diese Werte den Markt, die Simulation oder die RL-Agenten beeinflussen wird hier nicht weiter erläutert. Weitere Informationen zu den Einflüssen auf den Markt lassen sich in [4, 6] finden. Die `environment_config` wird in [16] beschrieben. Für die RL Parameter ist [9], ein Fachbuch wie z.B. [30] oder die Dokumentation der Algorithmen, wie z.B. [32] zu empfehlen. Um dennoch ein Gefühl für den Inhalt der drei Dateien zu bekommen, wird dieser hier kurz zusammengefasst.

`sim_market_config` Diese Konfigurationsdatei beinhaltet alle Parameter, die für die Kalibrierung des simulierten Marktes zuständig sind. Das inkludiert z.B. die Anzahl der Kunden, die sich pro Runde für eine der Aktionen (nichts tun, Neukauf oder Gebrauchtkauf) entscheiden. Andere Parameter sind z.B. die Lager- oder die Produktionskosten.

`rl_config` In der Datei befinden sich Konstanten, die für den RL-Agenten zum Erlernen der Verkaufsstrategien relevant sind. Diese Parameter sind von dem gewählten RL-Algorithmus abhängig. Für QLearning [26] beispielsweise ist ein Parameter `gamma` (Diskontierungsfaktor). Dieser liegt zwischen null und eins und gibt an, wie wichtig dem Agenten zukünftige Gewinne sein sollen. Ist der Wert näher an null, so wird der Agent sich eher für kurzfristige Gewinne entscheiden. Ist der Wert näher an eins, so ist

der Agent bereit seinen Gewinn zu verzögern, um dann später einen eventuell höheren Gewinn zu erzielen.

`environment_config` Hier befinden sich alle sonstigen Parameter. Ein wichtiger Parameter ist z.B. der `marketplace`. Dieser gibt an, welcher Markt von der Simulation genutzt werden soll. Abhängig von diesem können dann auch entsprechende Agenten über das Schlüsselwort `agents` gesetzt werden. Die unterschiedlichen Agenten und Märkte werden in [4, 6, 16] beschrieben. Ein weiterer bedeutsamer Wert ist der `task`. Dieser wird von den Nutzenden festgelegt und bestimmt welche Aktion das Framework ausführen soll. Es sind drei `tasks` definiert: `training`, `exampleprinter` und `agent_monitoring`.

Um RL Agenten trainieren zu können muss der `task` auf `training` gesetzt sein. Das Framework wird dann ein Training starten. Welcher RL-Algorithmus hier gewählt werden soll, wird ebenfalls über die `environment_config` geregelt, über den Parameter `agents`. Die Konstanten für das Training stehen dann im `rl` Teil der Konfigurationsdateien. Neben dem Training bietet das Framework auch Werkzeuge an die Strategien von Agenten zu evaluieren. Für diese Evaluierungsaufgaben sind die `tasks` `exampleprinter` und `agent_monitoring` vorgesehen. Eine ausführliche Beschreibung und Analyse dieser `tasks` finden sich in [16].

Durch diese verschiedenen Aufgaben kann das Framework vielseitig eingesetzt werden. In dieser Arbeit werden wir uns bei den Skalierbarkeits-Experimenten (vgl. Kapitel 6) ausschließlich mit der Aufgabe `training` beschäftigen. Diese ist, von allen vorgestellten Aufgaben, die mit dem größten Rechenaufwand. Die vollständige Durchführung einer dieser Aufgaben wird im Folgenden immer als Experiment bezeichnet.

Für die lokale Ausführung werden die drei beschriebenen `JSON` Dateien benötigt. In der hier vorgestellten Netzwerkarchitektur werden die Dateien zu einer zusammengefasst. Damit muss nur eine Datei auf die entfernte Maschine gebracht werden. Um sie anschließend wieder separieren zu können, werden die Dateien `sim_market_config` und `rl_config` unter dem Schlüsselwort `hyperparameter` (und dann die jeweilige Dateibezeichnung) zusammengefasst. Die Werte der `environment_config` werden unter dem Schlüsselbegriff `environment` abgebildet. Eine beispielhafte zusammengesetzte Konfigurationsdatei befindet sich in Quelltext 6.1. Die Zusammenfassung der `sim_market_config` und `rl_config` unter dem Schlüsselwort `hyperparameter` liegt darin begründet, dass diese beiden Dateien Parameter enthalten, die den Markt, bzw. die RL-Agenten konfigurieren. Sie sind abhängig von dem gewählten Marktszenario und dem RL-Algorithmus. Die Parameter in `environment_config` hingegen beziehen sich ausschließlich auf die Ausführungsumgebung.

3.2 Während der Ausführung

Im folgenden Abschnitt werden die Anforderung immer anhand des Trainings beschrieben. Dieselben Anforderungen gelten analog auch für `exampleprinter` und `agent_monitoring`.

Wie bereits erwähnt, ist die Wahl der richtigen Hyperparameter für die RL-Algorithmen eine Herausforderung für Forschende in diesem Gebiet. Eine Konfiguration muss außerdem durch mehrfache Ausführung des Trainings mit derselben Konfiguration verifiziert werden. Um effizient mit dem Framework arbeiten zu können und möglichst kurz auf die Ergebnisse der Experimente oder deren Verifizierung warten zu müssen, ist es wünschenswert, dass diese parallel ausgeführt werden. Dabei müssen die Experimente isoliert voneinander ablaufen.

Ansonsten könnten sie sich gegenseitig beeinflussen. Eine solche Beeinflussung macht die Ergebnisse unbrauchbar.

Reinforcement Learning kann, trotz Parallelisierung, gerade bei weniger guter Hardware sehr lange dauern. Daher soll es Forschenden ermöglicht werden Experimente über Nacht laufen lassen zu können. Wichtig dabei ist, dass das eigene Gerät nur zum Starten dieses Vorgangs benötigt wird und während der Experiment Durchläufe auch ausgeschaltet sein kann. Ist die angedachte Laufzeit für ein größeres Experiment mehrere Stunden / Tage, so ist es sehr sinnvoll mit Hilfe einer Benachrichtigung zu erfahren, ob das Experiment beendet oder abgebrochen wurde. So kann frühzeitig ein neuer Durchlauf gestartet werden.

Während eines Experiments gibt die Simulation regelmäßig Ausgaben auf der Kommandozeile aus. Diese Ausgaben bestätigen dem Nutzenden, dass das Training noch läuft. Sobald das Training abbricht, muss der Nutzende eventuell Schritte in die Wege leiten, um z.B. das Training neu zu starten. Bei dem Trainieren auf einer remote Maschine ist somit zu beachten, dass die Ausgaben auch dem Nutzenden zugänglich gemacht werden sollten. Auch wenn dies nicht erfüllt werden kann, sollte es eine minimale Anforderung sein, dass es dem Nutzenden möglich ist herauszufinden, ob das angefangene Training noch läuft. Soll ein Training abgebrochen werden, so kann lokal der Prozess mit z.B. `Ctrl + C` abgebrochen werden. Für remote laufende Prozesse muss dagegen diese Schnittstelle erst noch implementiert werden.

Des Weiteren unterstützt das Framework `TensorBoard` [58]. Es ist das Visualisierungtool der Python Bibliothek `TensorFlow` [31]. Mit Hilfe dessen können Forschende einen Einblick in den aktuellen Stand des Trainings bekommen. Während eines Trainings werden verschiedene Metriken, wie z.B. die Profite oder die Verkaufszahlen in eine Datei geschrieben. `TensorBoard` kann diese auswerten und als Grafiken in einer Weboberfläche anzeigen. Eine beispielhafte Bildschirmaufnahme findet sich in Abbildung 3.1. Die Bildschirmaufnahme zeigt einen Ausschnitt des `TensorBoards` während eines Trainings in der `recommerce` Simulation. Diese Visualisierungen sollen den Nutzenden ebenfalls zur Verfügung stehen.

3.3 Nach der Ausführung

Während der `tasks`, die das Framework bearbeiten kann, fallen Daten an. Bei einem Training sind das z.B. verschiedene Agenten, die in verschieden Stadien des Trainings gespeichert werden. Nach einem Training wird außerdem automatisch die Aufgabe `agent_monitoring` ausgeführt. Bei dieser werden verschiedene Diagramme zu der Performance des Agents in dem Markt erstellt. Ein paar beispielhafte Grafiken sind in Abbildung 3.2 zu sehen. Mollenhauer [16] beschreibt alle Grafiken ausführlich.

Wird eine der Aufgaben lokal ausgeführt, so speichert das Framework alle Daten in einem Ordner. In diesem Ordner befinden sich die Grafiken des `agent_monitoring`, die Dateien für das `TensorBoard` und die während des Trainings gespeicherten RL-Modelle.

Die Anforderung an die Netzwerkarchitektur besagt, dass dieses Verzeichnis zur Verfügung stehen soll. Die Dateien lokal auf den eigenen Rechner zu bekommen und auswerten/verwerten zu können ist für das wissenschaftliche Arbeiten wichtig.

Alle hier beschriebenen Anforderungen müssen von der Erweiterung des `recommerce` Frameworks erfüllt werden können.

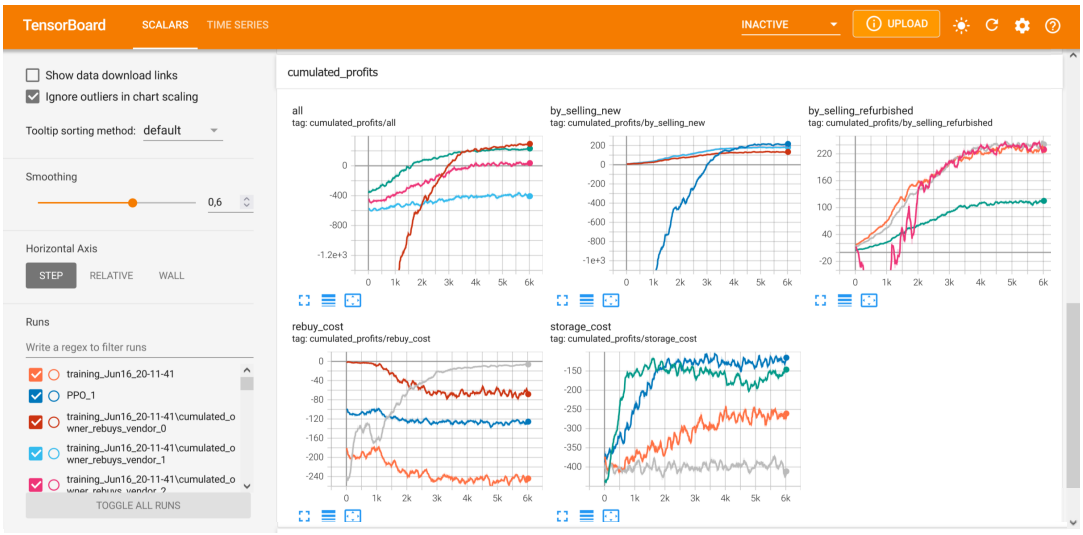


Abbildung 3.1: Beispielhaftes `TensorBoard` - Training eines Stable Baselines PPO Agenten [64] im Oligopoly mit drei regelbasierten Konkurrenten

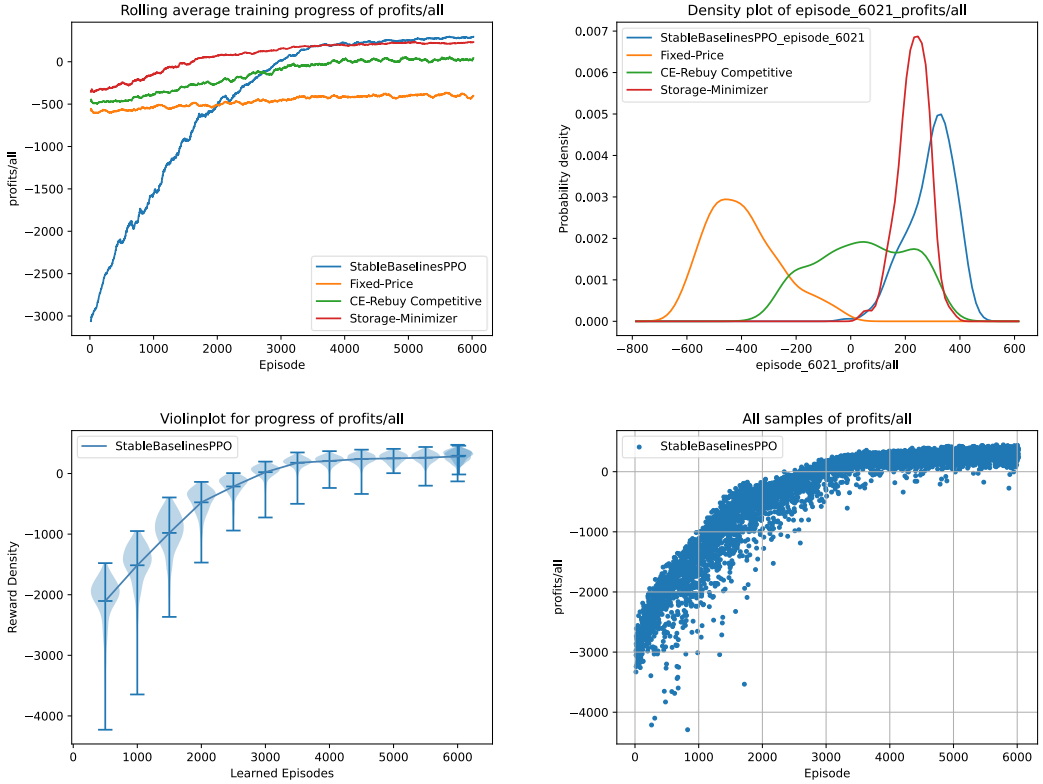


Abbildung 3.2: Beispielhafte Diagramme, die mit `agent_monitoring` erzeugt wurden. In diesen Grafiken: Stable Baselines PPO Agent [64] im Oligopoly mit drei regelbasierten Konkurrenten

4 Netzwerkarchitektur

Die im vorherigen Kapitel ausgearbeiteten Anforderungen prägen den Entwurf der Netzwerkarchitektur. Die entstandene Architektur besteht aus drei Komponenten: den Containern, der *API* und dem Webserver, siehe Abbildung 4.1. Container werden genutzt, um die unabhängige Parallelisierung der Experimente zu gewährleisten. Eine *REST API*, die auch auf der remote Maschine laufen muss, steuert die Container. Der Webserver dient als grafische Nutzeroberfläche, um alle Anforderungen aus dem vorherigen Kapitel umsetzen zu können. Gleichzeitig ist er auch *Client* für die *API* und übersetzt die Befehle des Nutzenden. Alle Komponenten der Architektur sind einzeln austauschbar. Untereinander kommunizieren sie nur über die vorgesehenen Schnittstellen.

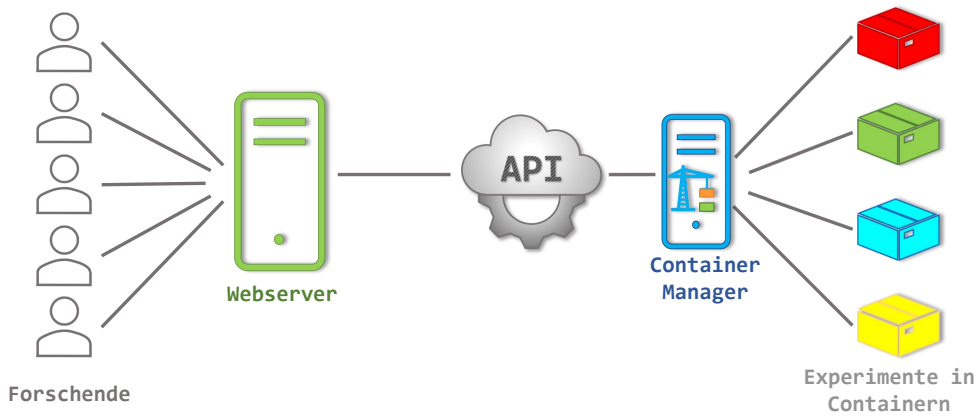


Abbildung 4.1: Schematische Darstellung der Netzwerkarchitektur

Alternativ zu der Netzwerkarchitektur können Forschende die Experimente z.B. auch mit *SSH* (Secure Shell) auf der entfernten Maschine ausführen. Mit Hilfe von *SCP* kommen die benötigten Konfigurationsdateien auf die Maschine und die Daten der Simulation zurück zum Nutzenden. Über *SSH* kann das `recommerce` Framework genau wie lokal genutzt werden. Dieser naive und direkte Weg bietet den Vorteil, dass keine zusätzliche Infrastruktur implementiert werden muss. Im Folgenden wird dieser Lösungsansatz '*SSH*-Ansatz' genannt. Außerdem werden einige Punkte aufgezeigt, bei denen dieser Ansatz der gewählten Architektur unterlegen ist.

4.1 Container

In dem Framework soll es möglich sein, mehrere Experimente parallel und voneinander isoliert laufen lassen zu können. Dabei gibt es verschiedene Möglichkeiten, um eine solche Isolation zu gewährleisten. Die wohl bekannteste Möglichkeit ist dabei die Virtualisierung. Der traditionelle Weg ist dafür einen *Hypervisor* zu nutzen. Dieser kann virtuelle Maschinen

(VMs) erstellen und betreiben. Dabei bildet er eine Schicht zwischen der echten Hardware und der virtuellen Maschine. Seine Hauptaufgabe besteht darin, die Ressourcen und das Betriebssystem der VM zu isolieren. Vom Host-System unterstützt werden muss beispielsweise die `Betriebssystem-Kern-basierte virtuelle Maschine (KVM)` [46]. Mit Hilfe der `KVM` kann man auf einer Linux Maschine, mehrere virtuelle Maschinen, die jeweils eigene virtualisierte Hardware, Netzwerkarten etc. besitzen, betreiben. Der *Hypervisor* für die Maschinen ist dabei der Kernel selbst. Externe *Hypervisor* nutzen z.B. andere prominente VMs wie z.B. `VMWare` [62], `Virtual Box` [51] oder `Xen` [53].

Laut Scheepers [22] bringt die Virtualisierung mit Hilfe eines *Hypervisor* die von uns gewünschte Isolation. Weiterhin arbeitet er auch heraus, dass die Container-basierte Virtualisierung ebenfalls eine solche Isolation bringen kann. In den Containern können Programme oder ganze Betriebssysteme laufen. Dazu werden alle für die Ausführung des Inhalts wichtigen Dateien und Bibliotheken zusammengesammelt und dann ausgeführt. Die Container nutzen direkt die Hardware der Maschine, sodass diese nicht erst noch, wie bei VMs, emuliert werden muss. Scheepers schlägt vor, Container hauptsächlich für *Platform as a Service (PaaS)* einzusetzen. Dua et. al. [7] vergleichen in ihrer Arbeit Container und virtuelle Maschinen für die Anwendung bei *PaaS*. Wie auch schon Scheepers kommen sie zu dem Schluss, dass Container für *PaaS* gut geeignet sind, da sie u.a. weniger Mehraufwand zum Starten benötigen, weniger Speicherplatz nutzen und somit die Ressourcen der Host Maschine besser ausnutzen.

Neben Scheepers und Dua et. al. haben viele andere Arbeiten, die Performance verschiedener Visualisierungstechniken verglichen. So haben z.B. Potdar et. al. [18] in ihrer Arbeit die Performance von Containern mit der von virtuellen Maschinen verglichen. Als Containertechnologie nutzten sie dabei `Docker` [43]. Sie kommen zu dem Schluss, dass `Docker` der virtuellen Maschine in allen gemessenen Performance Punkten überlegen ist, dazu gehörten u.a. CPU, Memory und Disk I/O. Auch Rad et. al. [20] analysieren die Performance von `Docker` Containern und stellen fest, dass diese der `KVM` überlegen ist.

Die hier vorgestellte Architektur nutzt ebenfalls Container. Die Möglichkeit der Isolation und der vergleichsweise geringere Mehraufwand für das Starten der Container ist für diesen Anwendungsfall überzeugend.

4.2 API

Die *API* soll es einem *Client* möglich machen, Experimente auf der entfernten Maschine zu starten, zu beenden und zu überwachen. Dafür muss sie ein über `HTTP`-Requests erreichbares Interface bereitstellen.

Dabei implementiert die *API* größtenteils die von Fiedling [27] eingeführten Prinzipien der *REST (Representational State Transfer)* Architektur:

Client-Server Der *Client* kann bei Bedarf, auf Wunsch des Nutzenden, Informationen über laufende Experimente bei der *API* anfragen.

Zustandslosigkeit Bei jeder Anfrage, die der *Client* stellt, muss die ID, des betroffenen Containers mitgegeben werden. Auch die gewünschte Aktion ist Teil der Anfrage. Weder *Client* noch *API* speichern sich Information zu den gestellten Anfragen.

Caching Caching ist nicht implementiert, da zunächst nicht mit mehreren Tausend Anfragen gerechnet werden kann. Bei der Auswahl der Technologie sollte darauf geachtet werden, dass der für die *API* ausgewählte Server dies unterstützt.

Einheitiche Schnittstelle Dafür gibt es die spezielle Rückgabe im `JSON` Format. Siehe 5.3.2.

Mehrschichtiges System Die *API* bietet lediglich die `HTTP` Schnittstelle an, die Kommunikation mit den Containern bleibt dem *Client* gegenüber verborgen.

Der *Client* sollte mit Hilfe der *API* nur so wenig Kontrolle wie möglich über die Maschine bekommen, sodass er die Ressourcen der Maschine, nicht anders als vom Besitzenden gewollt, ausnutzt. Gleichzeitig muss der Nutzende aber so viel Kontrolle wie nötig über die Container haben, um die gestellten Anforderungen zu erfüllen. Um die Anforderungen umzusetzen, gibt es einige Routen, die meist über die `HTTP`-Methode `GET` erreichbar sind.

`/start`
: Diese Route ist die einzige, die lediglich die Methode `POST` erlaubt, da sie eine Konfigurationsdatei im Anforderungstext erwartet. Außerdem hat die Route `start` einen Parameter, `num_experiments`, mit dem die Anzahl der zu startenden Experimente definiert werden kann. Wird diese Route vom *Client* aufgerufen, so wird einer oder mehrere neue Container gestartet. In diesen läuft das gewünschte Experiment mit der gegebenen Konfigurationsdatei. Zurückgegeben werden jeweils alle Container IDs der gestarteten Container. Diese IDs können genutzt werden, um weiterhin über die *API* mit dem Container zu interagieren.

`/health`
: Diese Route benötigt den Parameter `container_id`. Die übergebene ID muss zu einem laufenden Container passen. Passt sie, so wird der Status des Containers überprüft. Der entsprechende Status wird dem *Client* wieder zurückgegeben. Mögliche Status sind dabei alle `Docker` Status: `created`, `restarting`, `running`, `removing`, `paused`, `exited`, `dead`. Weitere Informationen dazu in [29, S. 26].

`/logs`
: Auch diese Route ist Container-spezifisch und bedarf somit einer `container_id`. Bei dieser Route wird die Kommandozeilenausgabe des angefragten Experiments zurückgegeben.

`/data`
: Diese Route bezieht sich ebenfalls auf einen existierenden Container. Zurückgegeben wird hier das Verzeichnis, als `tar`-Archiv, mit den Ergebnissen der aktuellen Aufgabe. Damit wird die Anforderung die Daten, die ein Experiment produziert, einsehen zu können erfüllt.

`/data/tensorboard`
: Diese Route ist Container-spezifisch. Für den angefragten Container gibt sie zurück, auf welchem Port das `TensorBoard` gerade läuft. Mit dieser Route wird die Unterstützung des Visualisierungstools im Framework umgesetzt.

`/pause`
: Mit dieser Route kann, der angefragte Container angehalten werden. Damit könnten z.B. Ressourcen die dieser Container einnimmt teilweise wieder freigegeben werden.

`/unpause`
: Das Gegenstück zu `pause`, ist Container-spezifisch und nur für pausierte Container wirksam. Diese werden wieder fortgeführt. Die Funktionalität Experimente anzuhalten gibt es in dem `recommerce` Framework, wenn es lokal ausgeführt wird nicht. Somit ist es möglich, die Ressourcen auf der remote Maschine besser einzuteilen.

`/remove`
: Auch nach Abschluss eines Experiments bleibt der Container bestehen, um z.B. später die Daten herunterladen zu können. Soll der Container vollständig entfernt werden, so kann diese Container-spezifische Route genutzt werden.

Mit diesen Routen ist es uns nun möglich, die Experimente zu starten und zu überwachen. Damit sind fast alle der gestellten Anforderungen erfüllt. Die *API* muss auf der Maschine laufen, auf der auch die Container arbeiten, da direkt mit ihnen kommuniziert wird.

Ein großer Vorteil der *API* besteht darin, dass sie von der gesamten Kommunikation mit den Containern abstrahiert. Soll beispielsweise eine andere Container- bzw. Virtualisierungstechnologie eingesetzt werden, so ist dies einfach möglich. Bleiben die Routen und der Rückgabetyp für diese gleich, so muss keiner der *Clients* verändert werden.

4.3 Webserver

Um die Experimente als Nutzender, auch ohne Wissen über Code, starten zu können, gibt es eine grafische Weboberfläche. Über diese sollen alle Anforderungen, die an die Interaktion mit der Simulation gestellt werden, durch einfaches Klicken erfüllbar sein. So kann man damit Experimente konfigurieren, sie beobachten, abbrechen oder die Ergebnisse herunterladen. Der Webserver kann auf einer anderen Maschine als die *API* laufen. Die *API* sollte jedoch für ihn erreichbar sein. Auf der Maschine muss außerdem das `recommerce` Framework installiert sein. Die im Framework eingebaute Validierung wird ebenfalls vom Webserver zur Validierung eingegebener Konfigurationen genutzt.

Der Webserver verfügt über eine Nutzerverwaltung, sodass Forschende sich vor Benutzung des Webservers einloggen müssen. Das hat den Vorteil, dass ein Forschender nicht versehentlich ein wichtiges Experiment eines anderen entfernt.

Nach dem Anmeldevorgang wird man auf das Dashboard weitergeleitet. Über das können die verschiedenen Seiten aufgerufen werden. Es gibt eine Webseite (`observe`). Auf dieser lässt sich eine Übersicht über alle von dem Webserver aus gestartete Experimente finden. Von dort aus können die verschiedenen Routen der *API* angesprochen werden. Möglich sind hier: `/health`, `/pause`, `/unpause`, `/data/tensorboard`, `/remove`. Außerdem gibt es für jedes gestartete Experiment eine Detailseite, auf der neben den bereits genannten Funktionalitäten, auch noch die `/logs`, `/data` Routen funktionieren. Unter `upload` können auf dem Webserver neue Konfigurationsdateien hochgeladen werden. Das Formular auf der Seite `configurator` kann ebenfalls zur Erstellung von Konfigurationen genutzt werden. Es gibt auch die Möglichkeit, dieses Formular mit einer hochgeladenen oder bereits genutzten Konfigurationsdatei vorzubefüllen. Auf der Seite können somit verschiedene Teilkonfigurationen zu Einer zusammengesetzt werden. Die hier beschriebenen Seiten sind in Abbildung A.2 und Abbildung A.1 dargestellt. Eine ausführliche Beschreibung eines Arbeitsablaufs findet sich in [6].

Der Webserver bietet den Nutzenden den Vorteil, dass sie eine grafische Möglichkeit haben die Simulation zu konfigurieren. Außerdem müssen sie keine lokale Python Installation besitzen um ihre Experimente ausführen zu können. Im Vergleich zum *SSH*-Ansatz, bietet der Webserver die Möglichkeit, Experimente starten zu können, ohne die Kommandozeile nutzen zu müssen. Läuft der Webserver auf einer remote Maschine, so kann er auch von anderen Endgeräten, wie z.B. dem Smartphone oder dem Tablet genutzt werden. Das ist insbesondere dann praktisch, wenn man z.B. von unterwegs sicherstellen möchte, ob die Experimente noch laufen. Diese Überprüfung ist mit dem *SSH*-Ansatz nicht so einfach möglich.

5 Technologien

In diesem Kapitel werden die Technologien, die in dieser Implementierung, für die einzelnen Komponenten ausgewählt wurden, vorgestellt. Außerdem wird begründet, warum sie zum Einsatz gekommen sind. Die Entscheidungen wurden hierbei im Bachelorprojekt-Team getroffen.

5.1 Webserver

Es gibt sehr viele verschiedene Webframeworks, die eingesetzt werden können. Iwan Vosloo und Derrick G. Kourie [25] haben 2008 bereits 80 verschiedene aufgelistet. Die Auswahl ist somit sehr groß. Im Team ist es wichtig gewesen bereits vorhandenes Wissen, auch bei Entwicklung dieser Komponente, nutzen zu können. Die Simulation selbst ist in Python geschrieben. Daher ist dies die präferierte Sprache für die Auswahl des Frameworks geworden. Des Weiteren hat es in Python bereits Coding Standards gegeben, sodass der Aufwand für die Erarbeitung neuer Standards nicht zu hoch war. Erfahrung mit `Django` [38] hat ein Teammitglied bereits mitgebracht. Da es auf Python basiert, ist dies das genutzte Framework für den Webserver geworden.

`Django` ist ein gut dokumentiertes Open-Source Webframework, welches Entwickelnden viele Möglichkeiten bietet und einiges an Funktionalität mitbringt. So gibt es u.a. Maßnahmen gegen einige Cyber Attacken, wie z.B. *Cross-Site Scripting (XSS)*, *SQL Injection*, *Clickjacking* oder *Cross-Site Request Forgery (CSRF)*, siehe [8]. Für die Nutzung in der Erweiterung des `recommerce` Frameworks ist eine *Django App* entstanden. Eine der Funktionalitäten des Webframeworks, ist die schon implementierte Nutzerverwaltung. Die `users` *App* kommt auch in dieser Implementierung zum Einsatz, um die Experimente verschiedener Forschenden isolieren zu können. Außerdem schützt ein Login vor unbefugten Zugriffen Dritter.

`Django` folgt dem *Model-View-Controller (MVC)* [5] Prinzip. Dies ist ein Gestaltungsmuster der Softwarearchitektur. Es besteht aus den drei Komponenten: *Modell*, *View* und *Controller*. Das *Modell* beinhaltet alle für die Anwendung wichtigen Daten. Die *View* beschreibt die Darstellung, also alles womit Nutzende interagieren können. Der *Controller* verwaltet die Daten und verfügt über große Teile der Anwendungslogik.

Der Webserver besitzt ebenfalls einige *Modell* Klassen: ein *Modell*, welches einen Container auf der entfernten Maschine repräsentiert. In diesem ist u.a. die ID dieses Containers sowie die verwendete Konfiguration gespeichert. Die Konfigurationsdatei wird in den anderen *Modellen* abgebildet. Sie repräsentieren den hierarchischen Aufbau dieser Datei. Gespeichert werden die Daten in der bei `Django` standardmäßig vorgesehenen `sqlite` Datenbank. Die *Views* bestehen aus mehreren `HTML` Dokumenten. Für das Layout, grafische Designelemente und die einfachere Handhabung von Javascript sind im Webserver noch `Bootstrap` [33] und `jQuery` [39] eingebunden.

Die in 3.2 geforderte Benachrichtigung über die Beendigung von Experimenten soll natürlich auch umgesetzt werden. Dazu müssen live Informationen über die laufenden Container

zu dem Webserver gelangen. Der klassische Ansatz dabei ist vom Webserver aus in definierten Zeitintervallen die Route `/health` für jeden laufenden Container abzufragen (`HTTP`-Polling). Laut Murley et. al. [17] ist dieser Ansatz in vielen Echtzeitanwendungen der gebräuchlichste. In dieser Implementierung wird das, laut den Autoren weniger gebräuchliche, *Websocket*-Protokoll [37] verwendet.

Die *Websocket* ermöglicht die Kommunikation zwischen dem Browser des Nutzenden und der *API*. Nach der erfolgreichen Verbindung sendet die *API* dem Browser die IDs der beendeten Experimente. Der Browser leitet diese Nachricht an den Webserver weiter. Dieser überprüft die IDs und veranlasst, wenn nötig, dass eine Benachrichtigung angezeigt wird. Nötig ist eine solche Benachrichtigung genau dann, wenn in der Datenbank dieser Container noch nicht als beendet markiert ist. Ein großer Vorteil dieser Technologie ist es, dass der Webserver nicht regelmäßig Anfragen stellen muss. Diese würden die Performance des Webservers beeinflussen.

Diese Komponente der Architektur nutzt `Django` 4.0.1, `Bootstrap` 5.1, `jQuery` 3.6 in Verbindung mit Python 3.8.

5.2 Container

Die Technologie, die bei den Containern zum Einsatz kommt, nennt sich `Docker` [43]. Dies ist eine Containervirtualisierungstechnologie, mit der eine leichtgewichtige Virtualisierung umgesetzt werden kann. `Docker` nutzt sogenannte *Images*, um Container zu bauen. Das *Dockerfile*, welches von den Entwickelnden geschrieben wird ist die Bauanleitung, für ein solches *Image*. Einzelne Schritte können die Installation von Software oder das Ausführen von bestimmten Kommandozeilenbefehlen beinhalten. Eigene *Images* können auf *Images* anderer basieren. So muss nicht jeder, der z.B. Ubuntu in dem Container benötigt, komplett alle Schritte zur Installation dieses Betriebssystems beschreiben. Im *Dockerfile* kann angegeben werden, dass das *Image* auf einem anderen basieren soll. Die *Registry* ist eine Sammlung solcher *Images*. Ein sehr prominentes Beispiel für eine Registry ist der `Docker` Hub [42]. Das ist eine Cloud Plattform mit aktuell mehr als 9 Mio. verfügbarer *Images*, auf die das eigene aufgebaut werden kann.

Ist ein solches *Image* gebaut, so können wir von diesem *Image* aus einen oder mehrere Container starten. Für die Verwaltung der Container ist der sogenannte *Docker Daemon* zuständig. Die Kommunikation mit diesem läuft über eine *REST API*. Über die Kommandozeile können Nutzende mit ihm interagieren. Er übernimmt das Starten, Stoppen und die allgemeine Verwaltung der Container. Mit seiner Hilfe können wir uns z.B. auch den Status der Container anschauen, diese sind in [43] erklärt. Bekommt der *Daemon* den Befehl einen Container zu starten, so wird er alle im *Image* definierten Schritte ausführen. Soll Code nach dem Start ausgeführt werden, so muss ein passender *Entrypoint* gewählt werden. Im Kontext der `recommerce` Implementierung wird der *Entrypoint* genutzt, um die entsprechenden `tasks` ausführen zu können. Der *Daemon* kann jedoch nur lokal angesprochen werden. Um eine remote *API* schreiben zu können, bietet `Docker` *Software Development Kits (SDK)* für verschiedene Sprachen an. In dieser Implementierung wird die `Docker SDK for Python` [44] genutzt. Diese Bibliothek stellt Python-Aufrufe für die Kommunikation mit dem *Daemon* bereit. Weitere Informationen zu Docker finden sich z.B. in [43, 29].

Für die Ausführung der Container auf `NVIDIA` *Grafikkarten* (GPUs), wird das `CUDA` Toolkit [35] benötigt. Dies übernimmt die Kommunikation mit der Hardware. Um das Toolkit nicht

eigenständig installieren zu müssen baut das genutzte *Image* auf dem `NVIDIA CUDA Image` (`nvidia/cuda:11.3.0-base-ubuntu20.04` [36]) auf. Auf diesem *Image* aufbauend wird Python installiert. Anschließend werden, die für die Installation des `recommerce` Frameworks, wichtigen Dateien in den Container kopiert. Dieses kann nun installiert werden und der entsprechende `task` wird ausgeführt.

Es wird `Docker` 20.10.15 genutzt. Die angesprochene `Docker SDK for Python` wird in der Version 5.0.3 in Verbindung mit Python 3.8 verwendet.

5.3 API

Die *REST API* hat die Aufgabe das Interface für die Kommunikation mit `Docker` nach außen bereitzustellen.

5.3.1 fastapi

Auch bei der Entscheidung für das Framework der *API* sollte das Wissen des Teams ausgenutzt werden. Daher kam auch hier nur ein Python Framework in Frage. Mit dem `django_restframework` [47] wäre es möglich die *API* auch mit `Django` umzusetzen. Dennoch ist die Entscheidung explizit gegen `Django` getroffen worden. `Django` wurde hier als zu komplex empfunden, um diese kleine *API* bereitzustellen. Daher ist die *API* mit dem `fastapi` [59] Framework umgesetzt worden.

Wie der Name impliziert ist dies ein sehr schlankes Framework, mit dem schnell *APIs* entwickelt werden können. Wie `Django` ist es gut dokumentiert. Hier galt es eine schnelle, einfache, mit wenigen Zeilen Code umsetzbare *API* zu entwickeln. Überzeugt hat, dass Routen in nur drei Zeilen Code definiert werden können. Daher ist die Wahl auf dieses Framework gefallen. `Fastapi` wird, wie es in der Dokumentation vorgeschlagen, in Verbindung mit einem `uvicorn` [61] Server genutzt. `Uvicorn` ist eine für Python speziell entwickelte Serverapplikation, mit deren Hilfe Frameworks wie z.B. `fastapi` laufen können. Es wird `fastapi` 0.73 in Verbindung mit Python 3.8 und `uvicorn` 0.17 genutzt.

5.3.2 Rückgabe

Um dem *REST*-Prinzip *einheitliche Schnittstelle* zu folgen ist die Rückgabe im `JSON` Format spezifiziert. Eine beispielhafte Rückgabe ist in Quelltext 5.1 zu sehen.

Quelltext 5.1: Beispielhafte Rückgabe der API, hier auf die Anfrage `/health`

```
{
    'id': '2ec012f244616a4d6e42b82571438597a256081ce01070e10dfe226d6b066296',
    'status': 'running',
    'data': None,
    'stream': None
}
```

In dem `id`-Feld steht die ID des Containers, dies ist gerade bei Container-spezifischen Routen wichtig. Damit ist es dem *Client* möglich, einen Integritätscheck zu machen. Er kann überprüfen, ob die empfangene Antwort auch zu der gesendeten Anfrage passt. Das `status`-Feld beinhaltet den aktuellen Status des angefragten Containers. Interessante Daten, wie z.B.

der Port für das `TensorBoard` stehen bei Anfrage der entsprechenden Routen im `data`-Feld. Der `stream` ist dann relevant, wenn z.B. die `/data` Route angefragt wird. Das `tar`-Archiv mit den Ergebnissen der Simulation wird hier als Stream zurückgegeben. Einzig die `/start` Route weicht ein wenig von diesem Muster ab. Bei `/start` können mehrere Container gestartet werden. Daher besteht die Rückgabe bei dieser Route aus mehreren solcher, in Quelltext 5.1 abgebildeter, Teilelementen.

Eine `HTTP`-Response beinhaltet auch immer einen `status_code` also eine Zahl, die angibt, wie die Verarbeitung der Anfrage gelaufen ist. Bei der Rückgabe der `status_codes` wird sich auf [34] bezogen. Eine erfolgreiche Anfrage hat immer den `status_code` 200 in der Antwort. Das heißt, der Container wurde gefunden und die angeforderte Aktion ist ohne Fehler durchgelaufen. Bekommt man `status_code` 404, so heißt das meist, dass der angefragte Container nicht gefunden werden konnte. Dies kann insbesondere bei Container spezifischen Routen auftreten. Es kann aber auch sein, dass ein Fehler während der Ausführung der gewünschten Operation aufgetreten ist. Die Ursache für den Fehler steht immer im `status` Feld der Antwort. Nachteilig hierbei ist, dass auch eine falsche Route den Code 404 liefert. Das wird bei dieser Implementierung nicht unterschieden. Der `status_code` 403 bedeutet, dass der *Client* nicht autorisiert ist die *API* zu nutzen (siehe 5.3.3).

Die *API* sollte im Idealfall so sicher gebaut sein, dass nie ein Fehlercode $\geq$ 500 auftritt. Sollte es dennoch sein, so ist die *API* fehlerhaft und das Problem sollte von den Entwickelnden oder Administratoren schnellstmöglich behoben werden.

5.3.3 Sicherheit

Damit nicht jeder beliebige *Client* mit der *API* sprechen kann, gibt es ein *API*-Token, welches bei jeder Anfrage im Authorization Header mitgeschickt werden muss. Zur Berechnung des Tokens wird ein sogenanntes Secret benötigt. Das Secret ist eine lange randomisierte Zeichenkette. Dies wird zwischen der *API* und dem *Client* geteilt. Das Token berechnet sich aus diesem Secret und der aktuellen Uhrzeit gehasht mit `sha256`. Ein Token ist maximal zwei Stunden gültig, sodass, sollte es mal verloren gehen, der Angreifende nicht so viel Zeit hat die *API* zu nutzen.

Kommt das Geheimnis an die Öffentlichkeit (z.B. es wurde auf GitHub gepusht), so muss es unbedingt zurückgezogen werden. Das heißt, ein neues geheimes Secret muss eingeführt werden. Sonst könnten sich Unbefugte unbemerkt Zugang zu der *API* verschaffen. Aus einer sicherheitstechnischen Perspektive ist der Verlust des geheimen Secrets ein großes Problem. Aber gerade bei unverschlüsselten Anfragen kann eine *man-in-the-middle* Attacke erfolgreich das Geheimnis mitlesen.

Als Mitigation wird hier die Möglichkeit der verschlüsselten Kommunikation geboten. Um diese zu gewährleisten, akzeptiert die *API* nur `HTTPS` Anfragen. Dafür sind eigene *SSL Zertifikate* mit der Hilfe von `OpenSSL` [52] erstellt worden. Für Pythons `request` Bibliothek gilt, dass diese die *SSL Zertifikate* der angefragten Routen standardmäßig überprüft. Ein in Python geschriebener Client, der die `request` Bibliothek nutzt, wie z.B. der Webserver, muss daher auch über das öffentliche Zertifikat der *API* verfügen. Auch einige Browser überprüfen die Zertifikate, sodass es auch hier notwendig sein kann, das selbst signierte Zertifikat hinzuzufügen.

Erst mit den `HTTPS` Anfragen kann die *API*-Token-Technologie sicher werden. Die angesprochene *man-in-the-middle* Attacke kann das Token so nicht lesen und damit eventuell das Secret entschlüsseln.

6 Skalierbarkeit

In diesem Kapitel soll nun konkret untersucht werden, wie gut die in den vorherigen Kapiteln vorgestellte Architektur skalieren kann. Dazu wollen wir $n \in \mathbb{N}$ parallele Experimente über die vorgestellte Architektur starten lassen. Es soll untersucht werden, wie sich die Ausführungszeit der Container in Abhängigkeit der Anzahl laufender Container entwickelt. Außerdem betrachten wir, wie sich die Ressourcennutzung verändert, wenn mehrere parallele Container gestartet werden.

Quelltext 6.1: Genutzte Konfigurationsdatei für die Skalierbarkeits-Experimente

```
{
    "environment": {
        "task": "training",
        "marketplace": "CircularEconomyMonopoly",
        "agents": [
            {
                "name": "CE Rebuy Agent (QLearning)",
                "agent_class": "QLearningAgent",
                "argument": ""
            }
        ]
    },
    "hyperparameter": {
        "rl": {
            "gamma" : 0.99,
            "batch_size" : 32,
            "replay_size" : 100000,
            "learning_rate" : 1e-6,
            "sync_target_frames" : 1000,
            "replay_start_size" : 10000,
            "epsilon_decay_last_frame" : 75000,
            "epsilon_start" : 1.0,
            "epsilon_final" : 0.1
        },
        "sim_market": {
            "max_storage": 100,
            "episode_length": 50,
            "max_price": 10,
            "max_quality": 50,
            "number_of_customers": 20,
            "production_price": 3,
            "storage_cost_per_product": 0.1
        }
    }
}
```

6.1 Aufbau

Um die Skalierbarkeit untersuchen zu können, müssen einige Messdaten erhoben werden. Diese müssen sowohl Auskunft über das System geben, als auch über die laufenden Container, sodass gut nachvollzogen werden kann, womit sich die beobachtbaren Ergebnisse begründen lassen können. Für die durchgeführten Experimente wird immer dieselbe Konfigurationsdatei verwendet. Diese ist in Quelltext 6.1 abgebildet.

6.1.1 Datenerhebung

Die Daten über das System müssen auf dem System selbst erhoben werden. Für die Daten über die Laufzeiten der Container, könnte auch in Erwägung gezogen werden, diese auf der Seite des *API-Clients* zu erheben. Dabei werden jedoch auch immer Netzwerklatenzen mit einbezogen. Um die Messdaten davon unabhängig zu machen, werden die Daten für diese Experimente auch auf der Seite der *API* erfasst. Dabei wird ein kleiner Teil der Ressourcen der Maschine für die Datenerfassung benötigt. Dieser ist jedoch im Vergleich minimal, sodass wir ihn ignorieren können. Des Weiteren wird die Ressourcennutzung auch für alle gestarteten Container gleich sein.

Bei der Laufzeit eines Containers muss nicht nur die reine Laufzeit der Simulation betrachtet werden. Vielmehr, soll auch die Dauer für das Starten von n Containern in die Ergebnisse mit einfließen. Daher wird als Startzeit für die Container immer die Zeit genommen, zu der der Aufruf an der *API* ankam. Beim Starten wird ebenfalls mitgeschrieben, wie viele Container in einer Gruppe gleichzeitig gestartet wurden. Für diese Gruppen wird außerdem eine *uuid*, also eine eindeutige ID vergeben. Mit dieser kann in der Auswertung erkannt werden, welche Container zusammen gestartet wurden. Für alle anderen Container-spezifischen Routen (siehe 4.2) wird die Zeit mitgeschrieben, zu der dieser Aufruf erfolgreich durchgeführt wurde. Wird der gleiche Aufruf mehrfach, auf denselben Container ausgeführt, so werden alle Aufrufe dokumentiert.

Um eine möglichst genaue Gesamtlaufzeit des Containers zu erhalten, gibt es auf der *API* Seite einen Prozess, der alle fünf Sekunden die Container überprüft und mitschreibt, welche Container bereits beendet sind und mit welchem Statuscode diese geendet haben. Später können wir so, alle Container, die nicht erfolgreich durchgelaufen sind von der Auswertung ausschließen. Wird ein Container manuell gestoppt, so wird zusätzlich mitgeschrieben, ob dieser vor dem Stoppen noch lief oder nicht. Alle Container, die abgebrochen wurden können keine zuverlässigen Daten über die Gesamtlaufzeit beisteuern. Eine genauere Übersicht über alle mitgeschriebenen Daten befindet sich in Tabelle A.1

Neben den Daten über die Container werden alle fünf Minuten wichtige Messwerte des Systems erhoben: CPU Nutzung, RAM Nutzung, GPU Nutzung und GPU Speichernutzung. Das Fünf-Minuten-Intervall liegt darin begründet, dass die Ausführung eines Containers mehr als fünf Minuten dauert. Dieses Intervall garantiert mindestens einen Systemmesswert pro Experiment. Gleichzeitig sorgt das Intervall auch dafür, dass nicht mehr als 288 Messwerte pro Tag anfallen. Somit wird die Menge an Speicherplatz, für die Datenerhebung reduziert.

Hinterher können wir die Systemdaten und die Containerdaten zusammenführen und herausfinden wie lange die Ausführung von n parallelen Experimenten dauert und welche Ressourcen dabei verbraucht werden. Außerdem können so Aussagen zur maximalen Anzahl paralleler Container mit der genutzten Konfiguration auf der genutzten Hardware gemacht werden.

6.1.1 Technologien

Um die Ergebnisse reproduzierbar zu machen, werden in diesem Abschnitt die verwendeten Technologien eingeführt. Die Technologien der Netzwerkarchitektur wurden bereits im vorherigen Kapitel beschrieben. Die für die Überwachung genutzten Softwaretechnologien, sowie die genutzte Hardware wird hier beschrieben.

Hardware

Für die Durchführung der Experimente nutzen wir eine virtuelle Maschine mit dem Betriebssystem Ubuntu 20.04.4 LTS [60]. Die Maschine verfügt über eine `NVIDIA` A40 GPU [49] mit 46068 MiB ($\approx$ 48 GB) Grafikspeicher. Außerdem stehen acht virtuelle CPU-Kerne zur Verfügung. In der verwendeten virtuellen Maschine existieren 100 GB RAM. Die Lese/Schreibgeschwindigkeit beträgt ca. 1.7 GB/s. Um dies zu messen, wurde mit Hilfe des Linux Kommandos `dd` eine 200 GB große Datei kopiert. Die Datei muss in etwa die doppelte Größe des Hauptspeichers haben, um so *Memory Caching* Effekte zu verhindern.

Software

Zum Mitschreiben auf der *API* Seite wird eine kleine `SQLight3` Datenbank genutzt, die über die Python Bibliothek `sqlite3` [40] angesprochen werden kann. Diese besteht aus zwei Tabellen. In der einen Tabelle sind die Informationen zu den Containern, (vgl. Tabelle A.1) gespeichert. Die zweite Tabelle beinhaltet die Daten über das System. Diese werden unabhängig von den Aktivitäten der *API* alle fünf Minuten gemessen. Um die aktuelle RAM und CPU Nutzung des Systems herauszufinden, wird die `psutil` Bibliothek [55] von Python genutzt. Diese unterstützt jedoch nicht das Auslesen der Daten für die `NVIDIA` Grafikkarte. Daher wird hier das `NVIDIA System Management Interface` genutzt. Mit `nvidia-smi` werden Informationen über die GPUs und ihre aktuelle Auslastung ausgegeben. Die Ausgabe dieses Kommandozeilenwerkzeugs wird ebenfalls in die Datenbank eingetragen.

6.2 Durchführung und Erwartungen

Während die Experimente ausgeführt wurden, liefen alle im letzten Abschnitt beschriebenen Werkzeuge für die Überwachung des Systems und der Container. Für die automatisierte Durchführung der Messungen wurde ein eigener *API Client* geschrieben. Dieser startet nacheinander über die *API* bis zu 40 Container ($n \leq 40$). Anschließend überprüft er alle fünf Minuten, ob die Container noch laufen. Container, die beendet sind, werden gestoppt. Sind alle Container gestoppt, wird 5,1 Minuten gewartet. Dann startet der *Client* erneut mehrere Container. Zwischen dem Starten von zwei verschiedenen Mengen an Containern wird immer etwas mehr als fünf Minuten gewartet. Damit ist sichergestellt, dass es immer mindestens einen Messwert des weniger belasteten Systems gibt.

Es ist zu erwarten, dass es eine Zahl $i \leq n$ geben wird, die aussagt, wie viele Container parallel laufen können, ohne sich gegenseitig Ressourcen wegzunehmen. Für diese Container wird die Ausführungsdauer in etwa gleich sein. Auch die Nutzung der Ressourcen wird hier noch nicht vollständig ausgelastet sein. Ab diesem i wird die Nutzung von CPU und GPU vermutlich bei 100 % liegen. Die Maschine, auf der die Experimente ausgeführt werden, ist sehr leistungsstark. Kann jeder CPU-Kern mit einem Container umgehen, so können wir

erwarten, dass dieses i in unserem Fall acht sein wird. Um zu verifizieren, dass wir mit dieser Annahme arbeiten können, wurde ein einzelner Container mit der Konfiguration gestartet. Mit `htop` [41] konnte daraufhin die CPU Auslastung ausgewertet werden. Beobachtbar war, dass immer eine der acht CPU-Kerne vollständig ausgelastet ist.

Wir nutzen dies im Folgenden als Annahme. Werden mehr als acht Container gleichzeitig gestartet, so ist die CPU der Annahme folgend ausgelastet. Es ist zu erwarten, dass die Ausführungsdauer von dort an linear zunimmt. Die Container können nun nicht mehr vollständig parallel laufen, sondern müssen sequentiell laufen.

Sei t_8 die Ausführungsdauer eines Containers bei acht parallel gestartete Containern. Für null parallel gestartete Container legen wir fest, dass die Ausführungszeit null Sekunden beträgt. Mit dem Punkt $(0, 0)$ kann nun eine lineare Beziehung aufgestellt werden. Es ergibt sich für die erwartete Ausführungsdauer t_n^e:

$$\forall n \in \mathbb{N} : t_n^e = \begin{cases} t_8 & \text{falls } 1 \leq n \leq 8 \\ \frac{t_8}{8} \cdot n & \text{sonst} \end{cases} \tag{6.1}$$

6.3 Auswertung

Für die Auswertung der Daten müssen die Daten über das System sowie die Daten über die Container zusammengeführt werden. Dazu werden die Datenbanktabellen als `CSV` exportiert und können anschließend ausgewertet werden. Die Menge aller gestarteten Container bezeichnen wir im Folgenden mit C. Wir nennen $M_n \subseteq C$ ($n \in \{1, ..., 40\}$) ein Set von n gleichzeitig gestarteten Containern. M_n bezeichnet also alle n Container, die mit dem Aufruf `/start?num_experiments=`n gestartet wurden. Mit anderen Worten in M_n befinden sich n parallel gestartete Container.

Bevor die Sets ausgewertet werden können, muss sichergestellt werden, dass diese gültig sind. Gültig ist ein Set M_n dann, wenn die folgenden Kriterien erfüllt sind:

1. Das Set M_n beinhaltet nur Experimente, die mit der Konfigurationsdatei aus Quelltext 5.1 gestartet wurden.
2. Während der Durchführung der Experimente dieses Sets wurden keine anderen Experimente über die *API* gestartet.
3. Kein Container aus M_n wurde vor Beendigung der Berechnungen abgebrochen.
4. Es existiert kein Container in M_n der pausiert wurde.
5. Für alle Container aus M_n gilt, haben sie sich selbstständig beendet, so ist die Zeit zwischen dem Starten des Containers und seiner Beendigung kleiner als eine Minute.

Erfüllt ein Set diese Anforderungen nicht, so wird es nicht ausgewertet. Die *API* kann von mehreren Forschenden genutzt werden. Daher kann es dazu kommen, dass eine der ersten vier Kriterien verletzt wird.

Mehrere Durchläufe und Experimente haben ergeben, dass mit dieser Konfiguration auf dieser Hardware die maximale Größe eines Sets 37 beträgt. Werden mehr als 37 Container gestartet, so beenden sich einige.

Die Ursache für diese Beendigung kann gefunden werden, wenn auf einen dieser Container die `/logs` Route aufgerufen wird. Die Ausgabe der Kommandozeile zeigt, dass das Experiment wegen des `RuntimeError: CUDA error: out of memory` Fehlers beendet wurde.

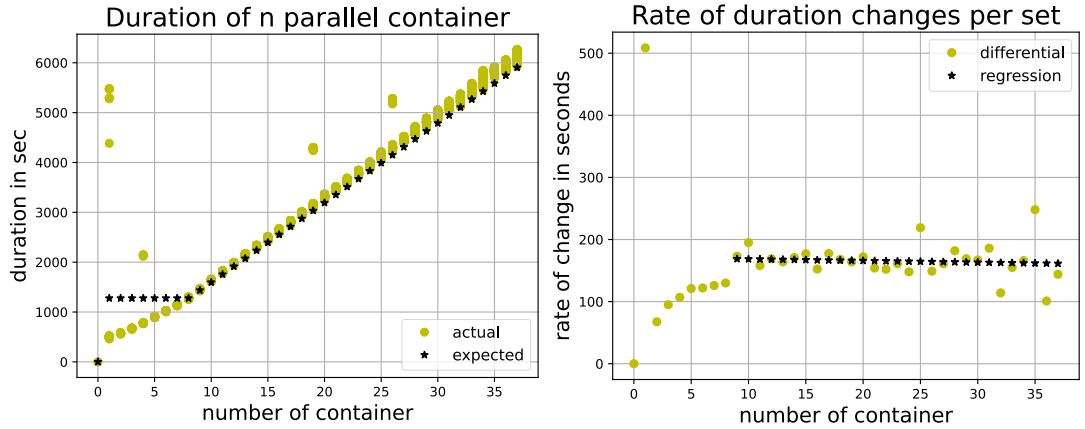
Das bedeutet, dass der Speicherplatz der Grafikkarte vollständig ausgenutzt ist. Diese vorzeitigen Beendigungen bleiben jedoch noch im Rahmen des fünften Kriteriums. Für jedes Set M_n muss demnach die Anzahl der tatsächlich gelaufenen Experimente $n' \leq n$ bestimmt werden. Ist das getan, so können die Sets $M_{n'}$ ausgewertet werden.

Die Ergebnisse der Auswertung, in Bezug auf die Dauer und den Ressourcenverbrauch der Experimente, werden wir in den folgenden Abschnitten betrachten. Die Beschränkung durch I/O (Eingabe/Ausgabe) wird hier nicht betrachtet werden.

Für diese Betrachtung wurden die Daten von insgesamt 12.161 Experimenten erfasst. Nach Anwendung der oben beschriebenen Kriterien für die Auswahl gültiger Sets, gingen davon 9.620 Experimente in die Auswertung ein, vgl. Tabelle A.2.

6.3.1 Dauer der Experimente

Nachdem die gültigen Sets $M_{n'}$ bestimmt wurden, muss noch die Ausführungsdauer für jedes Experiment ermittelt werden. Anschließend können die Ergebnisse in ein Streudiagramm eingetragen werden. Siehe Abbildung 6.1a. Auf der x-Achse ist die Anzahl der parallelen Container n' abgetragen. Die y-Achse beschreibt die Ausführungsdauer in Sekunden. Jeder grüne Punkt in dem Diagramm stellt die Ausführungsdauer eines Experiments dar.



(a) Dauer eines Sets M_n bestehend aus n parallelen Containern. Die erwarteten Werte sind nach Gleichung 6.2 berechnet worden.

(b) Änderungsrate der Medianwerte für jedes n

Abbildung 6.1: Diagramme zur Ausführungsdauer der Sets

Die Anzahl, der gewerteten Stichproben pro Set, kann in Tabelle A.2 nachgeschlagen werden. In Tabelle A.3 können die genauen Medianwerte für die Dauer eines Sets M_n nachgelesen werden.

Natürlich wollen wir die gemessenen Werte mit unseren Erwartungen vergleichen. Für die Gleichung 6.1 gilt $t_8 = 1277$. Wir erhalten:

$$\forall n \in \mathbb{N} : t_n^e = \begin{cases} 1277 & \text{falls } 1 \leq n \leq 8 \\ 159,625 \cdot n & \text{sonst} \end{cases} \tag{6.2}$$

Diese erwartete Beziehung ist mit schwarzen Sternen ebenfalls im Streudiagramm eingetragen.

Die Grafik zeigt, dass mit dieser Konfiguration maximal 37 Experimente parallel laufen können. Im Streudiagramm lassen sich einzelne Ausreißer finden. Auf der Maschine arbeiten mehrere Forschenden. Daher stehen die Ressourcen nicht nur den hier vorgestellten Experimenten zur Verfügung. Nutzen andere Menschen die Maschine, während die Container laufen, so hat dies Auswirkung auf die Laufzeit der hier vorgestellten Experimente. Wir können jedoch sehen, dass diese Fälle eher selten vorkommen.

Um die Veränderung der Ausführungsdauer besser beurteilen zu können, kann Abbildung 6.1b herangezogen werden. Betrachtet wird der Median pro Größe n eines Sets (vgl. Tabelle A.3). Für jedes mögliche n wird die Änderungsrate zu $n-1$ berechnet und dann in dieses Diagramm eingetragen. Das Diagramm lässt sich grob in drei Intervalle einteilen:

$1 < n < 5$ Betrachten wir den Wert für $n = 1$ als Ausreißer, so ist die Steigung für dieses Intervall linear. Damit steigt die Ausführungsdauer der Container hier parabelförmig an.

$5 \leq n \leq 8$ In diesem Intervall lässt sich beobachten, dass die Rate nicht anwächst, sondern ca. bei 120 Sekunden stagniert. Bezogen auf die Ausführungsdauer der Container bedeutet dies, dass diese in diesem Bereich linear anwächst.

$n < 8 \leq 37$ Die Änderungsrate in diesem Bereich scheint konstant zu sein. Für $n > 30$ schwankt sie jedoch. Daher ist zusätzlich, mit den schwarzen Sternen, eine lineare Regression über diese Datenpunkte im Diagramm eingetragen. Die Regression wurde mit der `polyfit` [48] Funktion der Python Bibliothek `numpy`, mit dem Grad eins, ausgeführt. Die regressierten Werte bilden eine konstante Funktion. Bezogen auf die Ausführungsdauer der Container bedeutet das, dass diese in diesem Bereich wieder linear steigen. Dennoch ist der Steigungsfaktor hier größer, als im vorherigen Intervall.

In Bezug auf unsere Annahme, sehen wir, dass unter acht parallelen Containern die Ausführungsdauer der meisten Container deutlich unter der von uns erwarteten Dauer liegt. Anders als ursprünglich angenommen, ist auch die Ausführungsdauer, für weniger als acht Experimente, nicht konstant. Vielmehr steigt sie bis zu vier Experimenten zunächst parabelförmig und bis zu acht Containern dann weiter linear. Für Sets mit einem $n' > 8$, stimmt jedoch unsere Annahme, dass die Ausführungsdauer mit Anzahl an parallel gestarteten Containern linear zunimmt. Um diese Beobachtungen und insbesondere die Diskrepanzen zur Annahme zu verstehen, muss die Ressourcennutzung des Systems genauer anschaut werden.

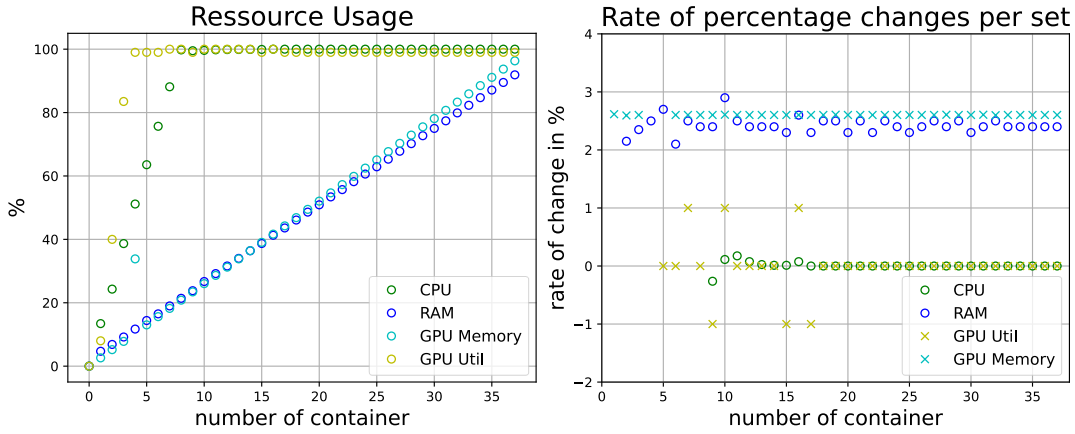
6.3.2 Ressourcennutzung

Für jedes Set $M_{n'}$ wurde erfasst, wann die Experimente auf der entfernten Maschine begannen und beendet wurden. Daraus ergeben sich Zeitspannen, in denen jeweils n' Container gleichzeitig liefen. Anhand dieser Intervalle wurden die Messwerte des Systems gefiltert. Passt ein Messwert in ein entsprechende Intervall, so wird er ausgewertet.

Die erfassten Metriken: CPU Auslastung, RAM Nutzung, GPU Auslastung und GPU Speicher Nutzung werden jeweils in Prozent angegeben. Die Werte für GPU und CPU werden in gleich großen Stichprobenintervallen erfasst. Dabei gibt die Prozentzahl an, wie viel Prozent der Zeit des letzten Intervalls die GPU bzw. CPU genutzt wurde. Eine solche Zeitspanne ist zwischen $1/6$ und einer Sekunde lang, siehe [50, 55]. Die genutzte Funktion misst den CPU Verbrauch für jeden der acht CPU-Kerne. Daher wird der Durchschnitt dieser Werte gebildet um einen Messwert pro Stichprobe zu erhalten. Bei der RAM und GPU Speicher Nutzung gibt die Prozentzahl den genutzten Speicher zu diesem Zeitpunkt an.

Für die Laufzeit eines Experiments fallen somit mehrere Stichproben an. Für jedes mögliche n' werden diese Stichproben pro Metrik gesammelt. Aus diesen Daten wird für alle n' der Median der jeweiligen Werte berechnet. Diese Medianwerte werden dann in die Grafik Abbildung 6.2a eingetragen. Die x-Achse bildet hierbei wieder die Anzahl paralleler Container n' ab. Die y-Achse geht von 0 bis 100 und gibt immer die Prozentzahl der Metriken für ein Set $M_{n'}$ an.

In dunkelgrün ist die CPU Auslastung dargestellt. Hellgrün stellt den prozentualen GPU Verbrauch dar. Die dunkelblauen Kreise bezeichnen den Anteil an genutztem RAM pro Set. Hellblaue stehen für die prozentuale Auslastung des GPU Speichers. Wie auch schon im vorherigen Kapitel wurden die Änderungsraten in Abbildung 6.2b aufgetragen.



(a) GPU, GPU Speicher, CPU und RAM die für die Ausführung eines Sets M_n von n parallelen Containern benötigt werden

(b) Änderungsrate der Medianwerte pro gemessener Metrik

Abbildung 6.2: Diagramme zur Ressourcennutzung der Sets

Abbildung 6.2a zeigt, dass die GPU ab vier parallelen Experimenten zu 100% ausgelastet ist. Die CPU ist erst ab ca. acht Containern ausgelastet. Dies entspricht unserer Annahme. Die Grafik bietet auch eine Erklärung für die Diskrepanz zwischen Annahme und Beobachtung für $n \leq 8$. Die, im vorherigen Kapitel definierten, Intervalle lassen sich auch hier finden.

Das erste Intervall ($1 < n < 5$) beschreibt die Ausführungsdauer mit einer weniger stark ausgelasteten GPU. Da diese jedoch immer stärker ausgelastet wird, verlängert sich die Laufzeit der Container. Das zeigt, dass diese Experimente stark von der Leistung der GPU abhängig sind. Für das zweite Intervall ($5 \leq n \leq 8$) bietet die CPU Auslastung eine Begründung. Mit steigender CPU Auslastung verlängert sich auch die Laufzeit linear. Über die Experimente sagt das aus, dass die CPU wenig genutzt wird, da ihre Auslastung nur einen linearen Einfluss auf die Ausführungsdauer besitzt. Sind CPU und GPU vollständig ausgelastet, so ist die Laufzeit immer noch vom verfügbaren RAM und GPU Speicher abhängig. In Abbildung 6.2b sehen wir, dass die Änderungsrate für diese über alle Intervalle konstant ist. Damit steigt die Auslastung von GPU Speicher und RAM mit zunehmender Zahl an parallelen Containern linear an. Der lineare Anstieg im dritten Intervall ($8 < n \leq 37$) ist somit belegt.

CPU und GPU sind also zuerst ausgelastet. Die Simulation ist somit GPU beschränkt. Die maximale Anzahl an Containern wird hauptsächlich durch die Menge an verfügbarem

Speicherplatz bestimmt. Abbildung 6.2a zeigt, dass ab 37 Containern der GPU Speicher zu voll ist. Im Median ist dieser bei 37 Containern zu 96.3% ausgelastet.

Um die Maschine vor Überlastung zu schützen werden Prozesse unter Linux beendet, sobald der Speicherplatz knapp ist. Für den ausgelasteten RAM kann unter Linux der sogenannte *Swap Space* genutzt werden. Das ist ein Speicherbereich auf der Festplatte, in den die Prozesse ausgelagert werden können, sobald der RAM voll ist. Für die `NVIDIA` GPU ist ein solches Konzept nicht vorgesehen. Trotzdem kann die Anzahl der maximal parallel ausführbaren Experimente noch etwas gesteigert werden, siehe 7.1.

6.4 Limitationen

Die Experimente sind mit Limitationen verbunden, die hier explizit aufgezeigt werden sollen. Eine Limitation des Versuchsaufbaus ist, dass für alle Experimente nur eine Konfiguration verwendet wurde. Insbesondere wurden nur die Trainingszeiten von QLearning berücksichtigt. Andere Reinforcement Learning Algorithmen besitzen eine andere Komplexität als QLearning und haben somit auch eine abweichende Trainingsdauer.

Es gibt sehr viele Parameter, die die Komplexität der Simulation und damit ihre Laufzeit und Ressourcennutzung beeinflussen. In dieser Arbeit werden wir nicht alle diese Parameter ausführlich diskutiert können. Gerade bei den verschiedenen RL-Algorithmen setzt dies, teilweise, ein tiefes Verständnis der Implementierung voraus, welches außerhalb des Rahmens dieser Arbeit liegt. Dennoch soll für einige Parameter aufgezeigt und erklärt werden, wie diese die Komplexität beeinflussen.

Betrachten wir einige ausgewählte Parameter, die Auswirkungen auf die Simulation haben und über die Konfigurationsdatei gesetzt werden können.

`marketplace` Zunächst einmal spielt der gewählte Marktplatz eine Rolle. Je nach Szenario befinden sich einer bis mehrere Agenten auf dem gewählten Markt. Jeder zusätzliche Agent erhöht die Zahl der gesetzten Preise, die sowohl die Kunden, als auch der RL-Agent betrachten muss. Dabei macht es dann auch noch einen Unterschied, ob es auf dem Markt einen Rückkaufpreis gibt oder nicht. Gibt es diesen, so muss der Agent statt zwei Preisen drei setzen. Außerdem muss er, statt zwei Preisen von jedem gegnerischen Agenten, nun drei Preise in den Entscheidungsprozess mit einbeziehen.

`max_price` Der Preis, der maximal gesetzt werden kann, ist für die RL-Agenten ebenfalls ein Faktor, der die Gesamtkomplexität erhöht und die Laufzeit beeinflusst. Durch einen höheren maximalen Preis erhöht sich die Anzahl der möglichen Preise, die ein Agent setzen kann. In einem diskreten Aktionsraum könnte dem Agenten z.B. nur möglich sein ganzzahlige Preise zu setzen. Beträgt der maximale Preis beispielsweise fünf, so kann er nur die Preise eins, zwei, drei, vier und fünf setzen. In einem Szenario mit Rückkaufpreis ergibt sich die Anzahl der möglichen Kombinationen diese drei Preise zu setzen als $5 \cdot 5 \cdot 5 = 125$. Wird der maximale Preis auf zehn gesetzt, so beträgt die Anzahl an möglichen Aktionen bereits $10 \cdot 10 \cdot 10 = 1000$. Über bestimmte Regeln, wie z.B. der Rückkaufpreis muss immer niedriger sein, als der Gebrauchtpreis kann diese Menge um einen Faktor verringert werden. Dennoch ist klar, dass mit steigendem maximalen Preis, die Anzahl an Möglichkeiten Preise zu setzen für den Agenten exponentiell zunimmt. Damit ist dies ein für die Laufzeit und Ressourcennutzung einflussreicher Parameter.

`number_of_customers` Ein weiterer Parameter, der die Komplexität des Marktes bestimmt, die Anzahl der Kunden. Kommen mehr Kunden, so müssen mehr Kaufentscheidungen getroffen werden. Die Kunden werden dabei über eine Wahrscheinlichkeitsverteilung modelliert. Dennoch muss auch diese mit steigender Anzahl an Kunden entsprechend häufiger gesampelt werden. Weitere Informationen zur Komplexität des Kundenverhaltens und dessen Auswirkung auf den Markt finden sich in [4].

Neben den Parametern, die das Framework direkt beeinflussen, spielt natürlich auch die Wahl der Technologien eine Rolle. Da die Zeit zum Starten eines Experiments ebenfalls in die Ausführungsdauer einberechnet wird, macht es einen Unterschied, welche Art der Virtualisierung genutzt wird. Wie bereits in 4.1 beschrieben ist z.B. die Startzeit von Containern im Vergleich zu der Startzeit von virtuellen Maschinen geringer.

Eine weitere Frage, die gestellt werden muss bezieht sich auf die Aktionen, die auf die laufenden Container ausgeführt werden können. So wurde in dieser Arbeit nicht untersucht, welchen Einfluss es auf die Performance eines Experiments hat, wenn z.B. jede Minute die `/data` Route auf den Container aufgerufen wird.

Die Anzahl der parallel gestarteten Container ist ebenfalls eine Limitation dieser Arbeit. Startet der Nutzende 50 Experimente, so sollen dabei auch 50 verwertbare Ergebnisse entstehen. Bei in der hier vorgestellten Architektur, mit der für die Skalierbarkeits-Experimente genutzten Konfiguration, wäre dies nicht möglich. In 7.1 werden wir auf eine mögliche Lösung für dieses Problem eingehen. Es ist dennoch nicht ausführlich untersucht und ausgewertet worden, sodass wir es hier in die Liste der Limitationen dieser Arbeit aufnehmen.

6.5 Fazit

Die hier genutzte Maschine kann bis zu 37 Container der Simulation mit der in Quelltext 6.1 vorgestellten Konfiguration gleichzeitig laufen lassen. Dann ist die Menge an verfügbarem GPU Speicher aufgebraucht.

Wir haben ermittelt, wie sich die Ausführungsdauer in Bezug auf die Anzahl an parallelen Containern auswirkt. (vgl. Abbildung 6.1) Bei bis zu vier parallelen Experimenten steigt die Dauer parabelförmig, ab dann linear. Dabei sind zwei verschiedene lineare Faktoren zu beobachten. Der für mehr als acht Container ist dieser lineare Faktor etwa 50 größer, als für weniger als acht Experimente.

Die Ergebnisse können unter Zuhilfenahme von Abbildung 6.2 erklärt werden. Es hat sich gezeigt, dass die Simulation für diese Konfiguration stark von der GPU abhängt. Die größte Limitation der Experimente ist die verwendete Konfiguration. Es wird nur eine Variante der Konfigurationsdatei genutzt. Daher kann das Ergebnis nicht als repräsentativ angesehen werden. Wird beispielsweise einer der beschriebenen Faktoren aus 6.4 verändert, so können die Ergebnisse stark abweichen. Dennoch kann das Fazit gezogen werden, dass mit Hilfe dieser Architektur, mehrere Experimente parallel gestartet werden können. Dies ist für das effiziente wissenschaftliche Arbeiten sehr wichtig.

Mit dem *SSH*-Ansatz ist das Starten mehrerer isolierter Experimente nicht ohne weiteres möglich. Das `recommerce` Framework bietet diese Parallelisierung nicht an. Mit dem *SSH*-Ansatz müsste der `task` manuell mehrfach gestartet werden. Dabei muss dann auch darauf geachtet werden, dass zwei gleichzeitig laufende Experimente, nicht auf dieselben Dateien mit Schreibberechtigung zugreifen wollen. Das Betriebssystem würde dies verhindern und somit wären die Ergebnisse der `tasks` eventuell unbrauchbar.

Bei der Untersuchung der Skalierbarkeit der vorgestellten Architektur ergeben sich durchaus weitere Forschungsfragen. Diese beschäftigen sich u.a. damit, wie sich verschiedene Faktoren auf die Laufzeit und Ressourcennutzung der Experimente auswirken, vgl. 6.4. Um die Beschränkung der Anzahl maximal parallel laufender Experimente noch ein wenig auszudehnen, wird eine mögliche Erweiterung der genuzten Implementierung in 7.1 vorgestellt.

7 Ausblick

Dieses Kapitel soll dazu dienen Erweiterungen, die die Architektur bereichern vorzustellen. Außerdem soll eine Zusammenfassung der Ergebnisse dieser Arbeit gegeben werden.

7.1 Erweiterung der Architektur

Bei dieser Konfiguration ist ab 37 parallelen Experimenten GPU Speicherplatz verbraucht. Nutzende wollen bei 50 gestarteten Experimenten auch 50 Ergebnisse erhalten. Auch für andere Konfigurationen ist zu erwarten, dass die Ressource GPU Speicher für eine bestimmte Menge an parallelen Containern erschöpft ist. Im Folgenden werden wir immer auf den GPU Speicher eingehen. Für RAM funktionieren die Ansätze analog. Hierfür könnte jedoch auch, der unter Linux konfigurierbare, *Swap Space* ausgenutzt werden.

Um zu erreichen, dass trotz mangelndem Speicher die Experimente durchlaufen, gibt es mehrere Möglichkeiten. Eine Möglichkeit besteht darin, die Menge an zur Verfügung stehender Hardware zu erweitern. Dabei kann entweder eine Maschine mit mehr Ressourcen genutzt werden (*Scale-Up*) oder eine weitere Maschine könnte die Genutzte ergänzen (*Scale-Out*).

Bei einem *Scale-Out* muss ein weiterer Knoten dem System hinzugefügt werden. Sind mehrere Rechner vorhanden, so müssen die Container verteilt werden, um eine optimale Ausnutzung der Ressourcen zu ermöglichen. In Kapitel 2 sind für dieses Problem bereits Lösungsansätze vorgestellt worden.

Eine andere Möglichkeit wäre es, die Container erst dann zu starten, wenn ausreichend Speicherplatz verfügbar ist. Es würde also lediglich die Information, dass ein Experiment gestartet werden soll, inklusive der dazugehörigen Konfiguration, gespeichert werden. Vor jedem zu startenden Experiment würde nun geprüft werden müssen, ob ausreichend GPU Speicher verfügbar ist. Ist das der Fall, so kann der Container gestartet werden. Gibt es jedoch nicht ausreichend Speicherplatz für einen weiteren Container, so muss dieser Start verzögert werden, bis wieder ausreichend Platz vorhanden ist.

Der tatsächliche Start der Berechnungen kann so also stark in die Länge gezogen werden. Dabei ist zu beachten, dass der *Client* der *API* die Antwort, mit den Referenzen zu den Experimenten zeitnah bekommen sollte. In den hier gezeigten Beispielen müsste ein *Client*, der mehr als 37 Experimente gleichzeitig starten möchte, mindestens 6000 Sekunden (100 Minuten) warten um die Antwort zu bekommen. Zu diesem Zeitpunkt sind einige der Container bereits beendet und dem *Client* ist keine Möglichkeit geboten worden, live den Container zu überwachen. Der Prozess des Startens muss also asynchron funktionieren. So kann der Prozess angestoßen werden, während gleichzeitig die Antwort an den *Client* zurückgegeben wird.

Mit mehr Hardware oder einer kleinen Softwareänderung kann so die Menge der Experimente, die auf der entfernten Maschine laufen erhöht werden. Gerade bei dem zweiten Ansatz muss mit einer deutlichen Steigerung der Ausführungsdauer gerechnet werden. Bei

der Softwarelösung ist jedoch auch zu bedenken, dass auch diese auf das Problem mangelnder Ressourcen treffen wird. Um jedoch eine kleinere Steigung für die maximale Anzahl an Containern zu erreichen, kann diese implementiert werden.

7.2 Zusammenfassung

In dieser Arbeit wurde eine Netzwerkarchitektur vorgestellt, die das `recommerce` Framework erweitert. Diese Komponente können Forschende, im Bereich der dynamischen Preisfindung nutzen, um effizient Reinforcement Learning Agenten zu trainieren. Sie besteht aus einem Webserver, der gleichzeitig die grafische Benutzeroberfläche zum Konfigurieren und Verwalten der Experimente darstellt. Außerdem dient er als *Client* für die *REST API*. Mit dieser können auf der remote Maschine die Experimente in `Docker` Containern gestartet werden. Mit der Architektur können alle in Kapitel 3 erwarteten Anforderungen umgesetzt werden. Sie ist unabhängig von der Infrastruktur Dritter. Stattdessen bringt sie alle Strukturen auf Basis von Python mit.

Alle Komponenten der Architektur können einzeln ausgetauscht werden, da sie jeweils nur über eine definierte Schnittstelle miteinander kommunizieren. Dadurch ist eine optimale Modularität und Wartbarkeit gegeben. Es könnte z.B. ein anderer Webserver in einer anderen Sprache oder mit einem anderen Framework geschrieben werden, ohne dass sich an der eigentlichen Ausführung der Experimente etwas änderte. Auch ist es möglich beispielsweise die Container durch eine andere Virtualisierungstechnologie auszutauschen. Wir haben gesehen, dass die Architektur dem naiven *SSH*-Ansatz überlegen ist. Dazu zählt u.a. dass die Architektur es Forschenden möglich macht die Simulation ohne Kommandozeile nutzen zu können. Ist der Webserver deployt, so kann er von jedem beliebigen Endgerät aus genutzt werden, welches ihn erreichen kann. Das ist bei dem *SSH*-Ansatz nicht so einfach möglich. Der wohl wichtigste Punkt, bei dem die Architektur der Arbeit mit *SSH* überlegen ist, ist die Skalierbarkeit. Mit Hilfe der Architektur ist es einfach möglich viele Experimente gleichzeitig starten zu können.

Für eine Konfigurationsdatei wurde diese Skalierbarkeit gemessen. Es ergab, dass die gestarteten Experimente durch die GPU limitiert sind. Ein zweiter limitierender Faktor, der sich negativ auf die Laufzeit auswirkt, ist die CPU. Die Anzahl der Container, die maximal parallel laufen können, beträgt für diese Konfiguration 37. Es wurde jedoch auch beschrieben, wie verschiedene Parameter Einfluss auf die Komplexität der Simulation haben können, sodass sich hier weitere spannende Forschungsfragen zu den Einflüssen der verschiedenen Faktoren auf die Experimente ergeben.

Literaturverzeichnis

Andere Veröffentlichungen

[1] Zeeshan Ahmed, Khalid Mohamed, Saman Zeeshan und XinQi Dong. „Artificial intelligence with multi-functional machine learning platform development for better healthcare and precision medicine“. In: *Database* 2020 (März 2020). baaa010. ISSN: 1758-0463. DOI: 10.1093/database/baaa010. eprint: https://academic.oup.com/database/article-pdf/doi/10.1093/database/baaa010/32923892/baaa010.pdf. URL: https://doi.org/10.1093/database/baaa010 (besucht am 17. Juni 2022).

[2] Saleh Md Arman und Cecilia Mark-Herbert. „Re-Commerce to Ensure Circular Economy from Consumer Perspective“. In: *Sustainability* 13.18 (2021), Seite 10242. URL: https://www.mdpi.com/2071-1050/13/18/10242 (besucht am 10. Juni 2022).

[3] Annukka Berg, Riina Antikainen, Ernesto Hartikainen, Sari Kauppi, Petrus Kautto, David Lazarevic, Sandra Piesik und Laura Saikku. „Circular Economy for Sustainable Development“. en. 2018. URL: http://hdl.handle.net/10138/251516 (besucht am 17. Juni 2022).

[4] Nick Bessin. „Der Marktplatz der Zukunft, Simulation von Marktprozessen im Re-Commerce“. Universität Potsdam: Hasso Plattner Institut, Enterprise Platform und Integration Concepts, 2022.

[5] John Deacon. „Model-view-controller (mvc) architecture“. In: *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf* 28 (2009).

[6] Leonard Dreeßen. „Preisfindung in der Recommerce-Domäne: Analyse von Preisstrategien mithilfe einer Online-Marktsimulation“. Universität Potsdam: Hasso Plattner Institut, Enterprise Platform und Integration Concepts, 2022.

[7] R. Dua, A. Raja und D. Kakadia. „Virtualization vs Containerization to Support PaaS“. In: *2014 IEEE International Conference on Cloud Engineering (IC2E)*. Los Alamitos, CA, USA: IEEE Computer Society, März 2014, Seiten 610–614. DOI: 10.1109/IC2E.2014.41. URL: https://doi.ieeecomputersociety.org/10.1109/IC2E.2014.41 (besucht am 15. Juni 2022).

[8] Devndra Ghimire. „Comparative study on Python web frameworks: Flask and Django“. Metropolia University of Applied Sciences, 2020.

[9] Jan Niklas Groeneveld. „A comparison of reinforcement learning algorithms for dynamic pricing in recommerce markets“. 2022.

[10] Manuel Hope. „Online Atmospherics in Second-hand Retail“. 2022. URL: https://dc.etsu.edu/honors/709 (besucht am 17. Mai 2022).

[11] Mu Li, David G. Andersen, Jun Woo Park, Alex Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita und Bor-Yiing Su. „Scaling Distributed Machine Learning with the Parameter Server“. 2014. URL: https://www.cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf (besucht am 16. Juni 2022).

[12] Mu Li, David G. Andersen, Alexander Smola und Kai Yu. „Communication Efficient Distributed Machine Learning with the Parameter Server“. In: NIPS'14 (2014), Seiten 19–27. URL: https://proceedings.neurips.cc/paper/2014/file/1ff1de774005f8da13f42943881c655f-Paper.pdf (besucht am 16. Juni 2022).

[13] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan und Ion Stoica. „RLlib: Abstractions for Distributed Reinforcement Learning“. In: Proceedings of Machine Learning Research 80 (Juli 2018). Herausgegeben von Jennifer Dy und Andreas Krause, Seiten 3053–3062. URL: https://proceedings.mlr.press/v80/liang18b.html (besucht am 14. Juni 2022).

[14] Álvaro López García u. a. „A Cloud-Based Framework for Machine Learning Workloads and Applications“. In: *IEEE Access* 8 (2020), Seiten 18681–18692. DOI: 10.1109/ACCESS.2020.2964386. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8950411 (besucht am 17. Juni 2022).

[15] Chnar Mustafa Mohammed, Subhi RM Zeebaree u. a. „Sufficient comparison among cloud computing services: IaaS, PaaS, and SaaS: A review“. In: *International Journal of Science and Business* 5.2 (2021), Seiten 17–30.

[16] Nikkel Mollenhauer. „Monitoring of Agents for Dynamic Pricing in different ReCommerce Markets“. Universität Potsdam: Hasso Plattner Institut, Enterprise Platform und Integration Concepts, 2022.

[17] Paul Murley, Zane Ma, Joshua Mason, Michael Bailey und Amin Kharraz. „WebSocket Adoption and the Landscape of the Real-Time Web“. In: *Proceedings of the Web Conference 2021*. WWW '21. Ljubljana, Slovenia: Association for Computing Machinery, 2021, Seiten 1192–1203. ISBN: 9781450383127. DOI: 10.1145/3442381.3450063. URL: https://doi.org/10.1145/3442381.3450063.

[18] Amit M Potdar, Narayan D G, Shivaraj Kengond und Mohammed Moin Mulla. „Performance Evaluation of Docker Container and Virtual Machine“. In: *Procedia Computer Science* 171 (2020). Third International Conference on Computing and Network Communications (CoCoNet'19), Seiten 1419–1428. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.04.152. URL: https://www.sciencedirect.com/science/article/pii/S1877050920311315 (besucht am 14. Juni 2022).

[19] Davy Preuveneers, Ilias Tsingenopoulos und Wouter Joosen. „Resource Usage and Performance Trade-offs for Machine Learning Models in Smart Environments“. In: *Sensors* 20.4 (2020). ISSN: 1424-8220. DOI: 10.3390/s20041176. URL: https://www.mdpi.com/1424-8220/20/4/1176 (besucht am 14. Juni 2022).

[20] Babak Bashari Rad, Harrison John Bhatti und Mohammad Ahmadi. „An introduction to docker and analysis of its performance“. In: *International Journal of Computer Science and Network Security (IJCSNS)* 17.3 (2017), Seite 228.

[21] Alessandro Sassu, Jose Francisco Saenz-Cogollo und Maurizio Agelli. „Deep-Framework: A Distributed, Scalable, and Edge-Oriented Framework for Real-Time Analysis of Video Streams“. In: *Sensors* 21.12 (2021). ISSN: 1424-8220. DOI: 10.3390/s21124045. URL: https://www.mdpi.com/1424-8220/21/12/4045 (besucht am 23. Juni 2022).

[22] Mathijs Jeroen Scheepers. „Virtualization and containerization of application infrastructure: A comparison“. In: *21st twente student conference on IT*. Band 21. 2014. URL: https://thijs.ai/papers/scheepers-virtualization-containerization.pdf (besucht am 15. Juni 2022).

[23] Johann Schulze Tast. „Pre-training RL-based pricing agents for Re-Commerce Applications using historical market data“. Universität Potsdam: Hasso Plattner Institut, Enterprise Platform und Integration Concepts, 2022.

[24] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen und Jan S. Rellermeyer. „A Survey on Distributed Machine Learning“. In: *ACM Comput. Surv.* 53.2 (März 2020). ISSN: 0360-0300. DOI: 10.1145/3377454. URL: https://doi.org/10.1145/3377454.

[25] Iwan Vosloo und Derrick G. Kourie. „Server-Centric Web Frameworks: An Overview“. In: *ACM Computing Surveys* 40.2 (Mai 2008). ISSN: 0360-0300. DOI: 10.1145/1348246.1348247. URL: https://doi.org/10.1145/1348246.1348247 (besucht am 17. Juni 2022).

[26] Christopher J. C. H. Watkins und Peter Dayan. „Q-learning“. In: *Machine Learning* 8.3 (Mai 1992), Seiten 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698 (besucht am 19. Juni 2022).

Bücher

[27] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[28] F. Moser. *Kreislaufwirtschaft und nachhaltige Entwicklung*. Herausgegeben von Heinz Brauer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, Seiten 1059–1153. ISBN: 978-3-642-60943-5. DOI: 10.1007/978-3-642-60943-5_18. URL: https://doi.org/10.1007/978-3-642-60943-5_18 (besucht am 9. Juni 2022).

[29] Adrian Mouat. *Using Docker - Developing and deploying software with container*. " O'Reilly Media, Inc.", 2015.

[30] Richard S. Sutton und Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.

Dokumentationen

[31] Martín Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (besucht am 17. Juni 2022).

[32] Stable Baselines3. *Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations*. 2020. URL: https://stable-baselines3.readthedocs.io/en/master/ (besucht am 24. Juni 2022).

[33] Bootstrap. *Build fast, responsive sites with Bootstrap*. 2022. URL: https://getbootstrap.com/ (besucht am 24. Mai 2022).

[34] MDN contributors. *HTTP response status codes*. 2022. URL: https://developer.mozilla.org/de/docs/Web/HTTP/Status (besucht am 7. Juni 2022).

[35] NVIDIA Corporation. *CUDA Toolkit*. 2022. URL: https://developer.nvidia.com/cuda-toolkit (besucht am 18. Juni 2022).

[36] nvidia Docker Inc. *NVIDIA CUDA*. 2022. URL: https://hub.docker.com/r/nvidia/cuda (besucht am 24. Mai 2022).

[37] Ian Fette und Alexey Melnikov. *The websocket protocol*. 2011.

[38] Django Software Foundation. *Django: The web framework for perfectionists with deadlines*. 2022. URL: https://www.djangoproject.com/ (besucht am 21. Mai 2022).

[39] OpenJS Foundation. *JQuery write less, do more*. 2022. URL: https://jquery.com/ (besucht am 15. Juni 2022).

[40] Python Software Foundation. *DB-API 2.0 interface for SQLite databases*. 2022. URL: https://docs.python.org/3/library/sqlite3.html (besucht am 3. Juni 2022).

[41] HTOP. *htop - an interactive process viewer*. 2022. URL: https://htop.dev/ (besucht am 25. Juni 2022).

[42] Docker Inc. *Build and Ship any Application Anywhere*. 2022. URL: https://hub.docker.com/ (besucht am 16. Mai 2022).

[43] Docker Inc. *Docker Dokumentation*. 2022. URL: https://docs.docker.com/ (besucht am 16. Mai 2022).

[44] Docker Inc. *Docker SDK for Python*. 2022. URL: https://docker-py.readthedocs.io/en/stable/ (besucht am 19. Mai 2022).

[45] Jupyter. *Jupyter*. 2022. URL: https://jupyter.org/ (besucht am 25. Juni 2022).

[46] KVM. *Main Page — KVM*, [Online; accessed 14-June-2022]. 2016. URL: https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792.

[47] Encode OSS Ltd. *Django REST framework*. 2022. URL: https://www.django-rest-framework.org/ (besucht am 7. Juni 2022).

[48] NumPy. *numpy.polyfit*. 2022. URL: https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html (besucht am 23. Juni 2022).

[49] nvidia. *NVIDIA A40*. 2022. URL: https://www.nvidia.com/de-de/data-center/a40/ (besucht am 3. Juni 2022).

[50] nvidia. *Useful nvidia-smi Queries*. 2022. URL: https://nvidia.custhelp.com/app/answers/detail/a_id/3751/%7E/useful-nvidia-smi-queries (besucht am 3. Juni 2022).

[51] Oracle. *Virtual Box*. 2022. URL: https://www.virtualbox.org/ (besucht am 15. Juni 2022).

[52] OpenSSL Project. *OpenSSL Cryptography and SSL/TLS Toolkit*. 2021. URL: https://www.openssl.org/ (besucht am 25. Juni 2022).

[53] Xen Project. *Xen Project brings the power of virtualization everywhere*. 2022. URL: https://xenproject.org/ (besucht am 15. Juni 2022).

[54] PyTorch. *PyTorch from research to production*. 2022. URL: https://pytorch.org/ (besucht am 25. Juni 2022).

[55] Giampaolo Rodola. *psutil documentation*. 2022. URL: https://psutil.readthedocs.io/en/latest/ (besucht am 3. Juni 2022).

[56] Apache Spark. *MLlib is Apache Spark's scalable machine learning library.* 2022. URL: https://spark.apache.org/mllib/ (besucht am 16. Juni 2022).

[57] The Ray Team. *RLlib: Industry-Grade Reinforcement Learning*. 2022. URL: https://docs.ray.io/en/latest/rllib/index.html (besucht am 17. Juni 2022).

[58] Tensorboard. *TensorBoard: TensorFlow's visualization toolkit*. 2022. URL: https://www.tensorflow.org/tensorboard?hl=en (besucht am 24. Mai 2022).

[59] tiangolo/fastapi. *FastAPI FastAPI framework, high performance, easy to learn, fast to code, ready for production*. 2022. URL: https://fastapi.tiangolo.com/ (besucht am 24. Mai 2022).

[60] Canonical Ltd. Ubuntu. *Ubuntu*. URL: https://ubuntu.com (besucht am 27. Juni 2022).

[61] uvicorn. *uvicorn. An ASGI web server, for Python.* 2022. URL: https://www.uvicorn.org/ (besucht am 26. Mai 2022).

[62] vmware. *vmware*. 2022. URL: https://www.vmware.com/de.html (besucht am 15. Juni 2022).

Weitere Quellen

[63] C & A. *Was wir mögen: Wenn Kleidung zurückkommt*. 2022. URL: https://www.c-and-a.com/de/de/shop/kreislaufmode (besucht am 7. Juni 2022).

[64] Stable Baselines3. *PPO*. 2022. URL: https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html (besucht am 18. Juni 2022).

[65] The North Face. *Same, Same. But better*. 2022. URL: https://www.thenorthface.de/de_de/sustainability.html#banner=summer22.Duffels.discoverourjourney.HPMain1 (besucht am 17. Mai 2022).

[66] Gartner. *Umsatz mit Cloud Computing** weltweit von 2010 bis 2021 und Prognose bis 2023*. 2022. URL: https://de.statista.com/statistik/daten/studie/195760/umfrage/umsatz-mit-cloud-computing-weltweit/ (besucht am 16. Juni 2022).

[67] Regina Henkel. *The North Face startet mit Recommerce-Plattform in Deutschland*. 22.04.2022. URL: https://fashionunited.de/nachrichten/einzelhandel/the-north-face-startet-mit-recommerce-plattform-in-deutschland/2021042240596 (besucht am 17. Mai 2022).

[68] iNaturalist. *iNaturalist*. 2022. URL: https://www.inaturalist.org/ (besucht am 17. Juni 2022).

[69] kingthorin. *SQL Injection*. Online. 2022. URL: https://owasp.org/www-community/attacks/SQL_Injection (besucht am 25. Juni 2022).

[70] KirstenS. *Cross Site Scripting (XSS)*. Online. 2022. URL: https://owasp.org/www-community/attacks/xss/ (besucht am 21. Juni 2022).

[71] Global Footprint Network. *Country Overshoot Days*. 2022. URL: https://www.overshootday.org/newsroom/country-overshoot-days/ (besucht am 17. Mai 2022).

[72] Gustav Rydstedt. *Clickjacking*. Online. 2022. URL: https://owasp.org/www-community/attacks/Clickjacking (besucht am 21. Juni 2022).

[73] Tatu Ylonen und Chris Lonvick. *The secure shell (SSH) protocol architecture*. Online. 2006. URL: https://www.rfc-editor.org/rfc/rfc4251n (besucht am 27. Juni 2022).

[74] Zalando Zircle. *Zircle*. 2022. URL: https://www.zircle.de/ (besucht am 17. Mai 2022).

Abbildungsverzeichnis

A Anhang

Glossar

Application Programming Interface (API) Eine *API* verbindet Computerprogramme untereinander, um Informationen zwischen diesen auszutauschen.

Clickjacking Cyberangriff, der häufig auf Webseiten vorkommt. Der Angreifende legt z.B. über einen Button, der eine bestimmt Funktion hat einen weiteres unsichtbares, klickbares Element. Klicken Nutzende auf den Button, in dem Glauben die Funktionalität des Buttons auszuführen, so wird stattdessen der Code des Angreifenden ausgeführt. Dieser hat den Klick des Nutzers ausgenutzt, weitere Informationen siehe [72].

Client Computerprogramm oder Mensch, der die Dienste eines Servers in Anspruch nimmt.

Cross-Site Scripting (XSS) Attacke, bei der bösartiger Code vom Angreifenden in z.B. eine vertrauensvolle Webanwendung eingeschleust wird. Dieser Code wird vom Browser des Nutzenden ausgeführt. Er wird ausgeführt, da es für den Browser so aussieht, als ob der Code von der vertrauensvollen Webseite kommt, weitere Informationen siehe [70].

Cross-Site Request Forgery (CSRF) Attacke, bei der Nutzende dazu gezwungen wird über eine Seite, bei der dieser authentifiziert ist vom Angreifenden gewünschte Aktionen auszuführen.

Django App Eine Web Applikation innerhalb des `Django` Frameworks, die für eine Aufgabe bestimmt ist z.B. ein Blog oder eine Umfrageanwendung. Eine Website kann aus mehrerer solcher *Apps* bestehen, siehe [38].

Docker Daemon Verwaltet `Docker` Container.

Dockerfile Datei, die die Schritte enthält, die nötig sind ein `Docker` *Image* zu erstellen, weitere Informationen siehe [43].

Docker Image Ein `Docker` *Image* wird aus einem *Dockerfile* erstellt. Es enthält alle Schritte, die notwendig sind einen vollständigen Container laufen zu lassen, weitere Informationen siehe [43].

Docker Registry Als `Docker` *Registry* wird eine Ansammlung von *Images* bezeichnet. Sie kann auch in der Cloud öffentlich Zugänglich sein, wie z.B. `Docker` Hub [42], weitere Informationen siehe [43].

Hypervisor Ein *Hypervisor* erstellt und betreibt virtuelle Maschinen. Er isoliert die Ressourcen und das Betriebssystem einer VM.

Infrastructure as a Service (IaaS) Cloud-Computing Modell, bei dem der Kunde Rechenleistung und Speicherplatz gestellt bekommt. Alle benötigte Software kann dieser dann selbst installieren und verwalten, siehe [15].

Key-Value Store Speichermöglichkeit, bei dem jeder gespeicherte Wert einem eindeutigen Schlüssel zuzuordnen ist. Über diesen kann der Wert auch abgefragt werden.

man-in-the-middle Attacke Attacke, bei der ein Angreifender die Kommunikation zwischen zwei Parteien abfängt. Die abgefangenen Nachrichten können vom Angreifenden z.B. verändert werden.

Memory Caching Technik, mit der Daten zwischenzeitlich im RAM gespeichert werden, um die Zugriffszeiten für diese Daten zu verkürzen. Z.B. wenn eine Datei kopiert wird, kann ihr Inhalt im RAM behalten werden, sodass beim anschließenden Öffnen der Datei die Daten nicht erst von der Festplatte gelesen werden müssen.

Model-View-Controller (MVC) Entwurfsmuster, welches die Daten, von der Oberfläche und der Datenverwaltung trennt. Es besteht aus drei Komponenten: *Modell*, *View* und *Controller*. Das *Modell* beinhaltet die Daten, die *View* deren Anzeige und der *Controller* implementiert die Verarbeitungslogik, weitere Informationen siehe [5].

Platform as a Service (PaaS) Cloud-Computing Modell, welches dem Kunden ein Framework bereitstellt, mit dem die Software dann entwickelt werden und verteilt werden kann, siehe [15].

Representational State Transfer (REST) Architektur, die beschreibt, wie entfernte Systeme miteinander kommunizieren können. Ist kein Standard, bedient sich aber standardisierten Methoden wie `JSON`, `HTTP`. Verfolgt fünf Prinzipien: Client-Server, Zustandslosigkeit, Caching, einheitliche Schnittstelle und mehrschichtige Systeme, siehe [27].

Scale-Out Erweiterung der bestehenden Architektur um eine weitere Maschine.

Scale-Up Verbesserung der Hardware der genutzten Maschinen.

Secure Copy (SCP) Programm, welches Daten zwischen zwei Rechnern verschlüsselt übertragen kann.

Secure Shell (SSH) Protokoll, welches eine sichere Kommunikation zu einer entfernten Maschine über ein unsicheres Netzwerk bereitstellt, siehe [73].

Software as a Service (SaaS) Cloud-Computing Modell, bei dem der Kunde die bereitgestellte Software nutzen kann.

Software Development Kit (SDK) Eine Sammlung von Funktionen und Bibliotheken, die beim Programmieren helfen.

SSL Zertifikat Datei, die einen Schlüssel enthält, der Informationen über den Betreiber der Webseite enthält. Sie werden benötigt, um z.B. `HTTPS` umzusetzen.

SQL Injection Injektion einer SQL Abfrage in ein bestehendes Datenbanksystem. Durch eine *SQL Injection* können sensitive Daten gelesen, gelöscht oder verändert werden, weitere Informationen siehe [69].

Swap Space Ist der RAM in einem Linux System voll, so kann der Kernel diesen reservierten Speicherbereich nutzen, um Prozesse teilweise dorthin auszulagern, um wieder Platz im RAM zu schaffen.

Trial-and-Error-Prinzip Das *Trial-and-Error-Prinzip* ist eine Methode zur Lösung eines Problems. Dabei werden so lange wie nötig zulässige Lösungen ausprobiert, bis die richtige, bzw. eine gute gefunden wird.

Universally Unique Identifier (uuid) 128-Bit Zahl, die eine Ressource eindeutig identifiziert.

Bildschirmaufnahme der Webserver UI

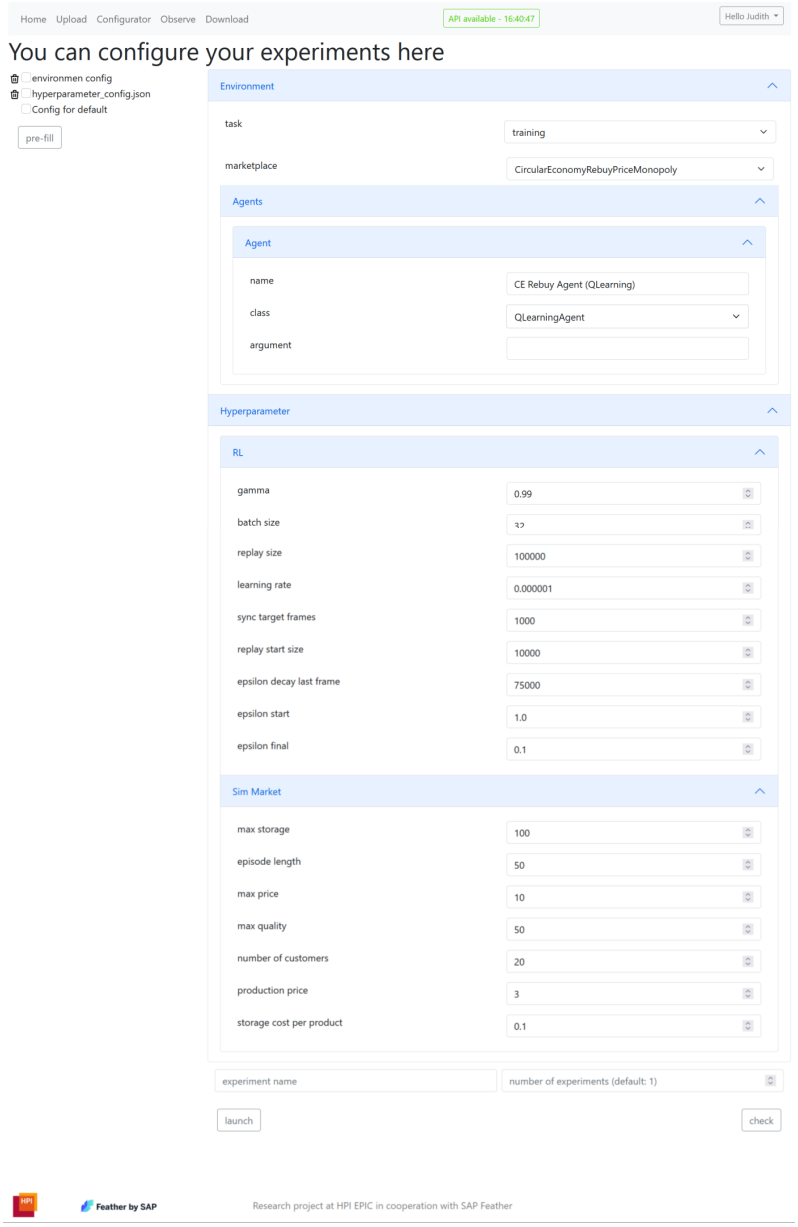


Abbildung A.1: Bildschirmaufnahmen der `configurator` Seite des Webservers

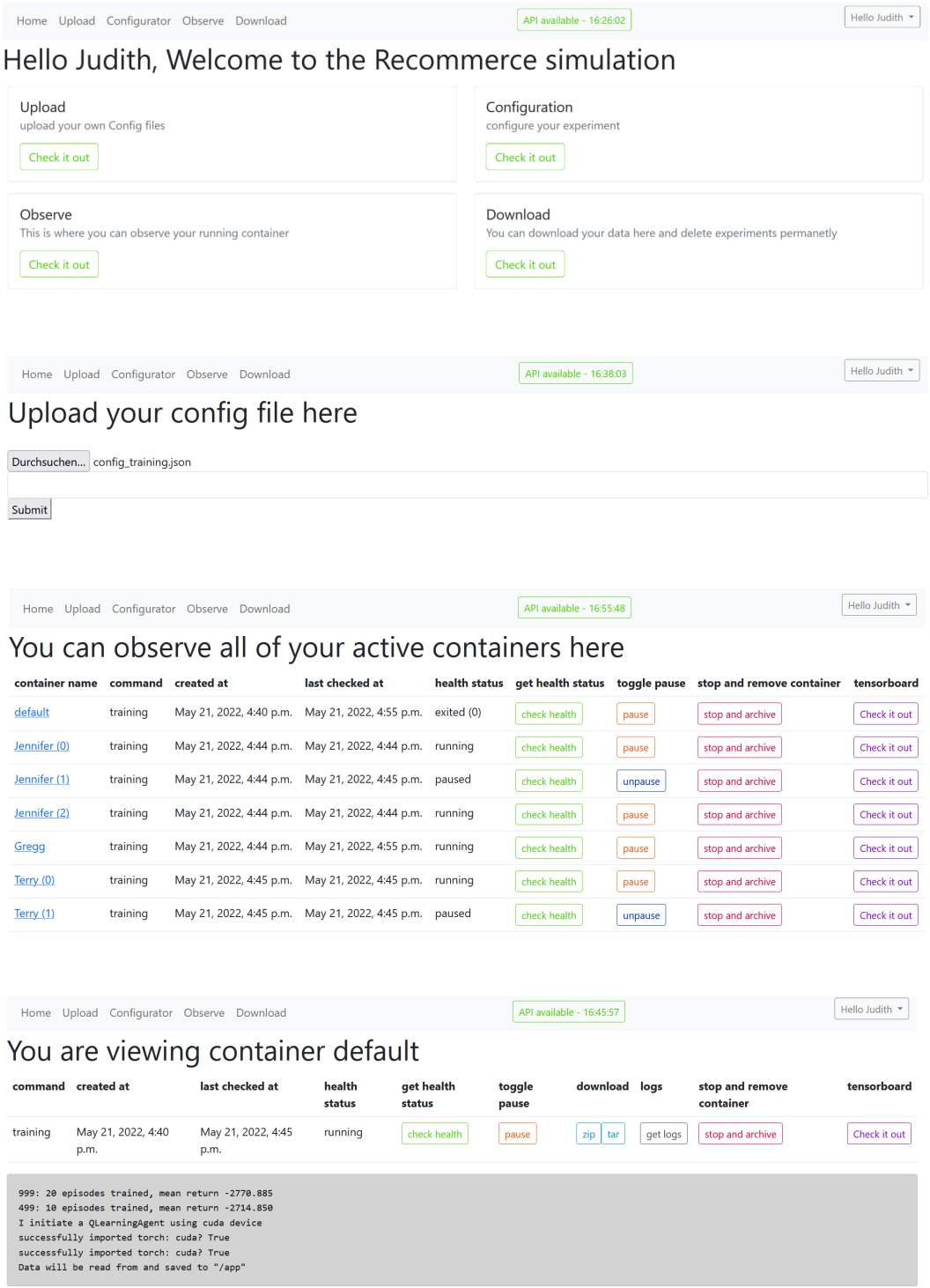


Abbildung A.2: Bildschirmaufnahmen der Seiten des Webservers von oben nach unten: `dashboard`, `upload`, `observe`, `details`

Zusätzliche Daten Skalierbarkeits-Experimente

Zur besseren Einordnung und Reproduzierbarkeit der Experimente in Kapitel 6, werden hier weitere interessante Daten angeführt.

Tabelle A.1: Erklärung der Datenbankspalten. Die Datenbank speichert die Informationen über die Aktionen, die auf die Container ausgeführt wurden.

Tabellenspalte	Beschreibung
`container_id`	Die ID des Container, die auch für Container-spezifische Routen in der *API* verwendet wird.
`config`	Die für den Container verwendete Konfigurationsdatei als Zeichenkette.
`started_at`	Die Uhrzeit zu der der Container gestartet wurde.
`started_by`	Ist der Container von einem Development-Server gestartet worden oder über den deployten Webserver. Beide haben verschiedene Access-Token für die API.
`group_id`	Eindeutige ID für alle zu der Zeit parallel gestarteten Container.
`group_members`	Anzahl der Container in dieser Gruppe.
`stopped_at`	Uhrzeit zu der der Container über die *API* Route `/remove` gestoppt wurde.
`force_stop`	Wahrheitswert, der angibt ob der Container zum Zeitpunkt des Stoppens noch lief oder schon beendet war.
`exited_at`	Zeit an der der Container sich selbst beendet hat.
`exit_status`	Der Status mit dem der Container geendet ist. Kann nur einer der `Docker` Status sein, vgl. [43].
`health`	Zeit zu der über die *API* `/health` auf den Container aufgerufen wurde. Bei mehreren Aufrufen sind diese durch Komma getrennt.
`paused`	Zeit zu der über die *API* `/pause` auf den Container aufgerufen wurde. Bei mehreren Aufrufen sind diese durch Komma getrennt.
`resumed`	Zeit zu der über die *API* `/unpause` auf den Container aufgerufen wurde. Bei mehreren Aufrufen sind diese durch Komma getrennt.
`tensorboard`	Zeit zu der über die *API* `/tensorboard` auf den Container aufgerufen wurde. Bei mehreren Aufrufen sind diese durch Komma getrennt.
`logs`	Zeit zu der über die *API* `/logs` auf den Container aufgerufen wurde. Bei mehreren Aufrufen sind diese durch Komma getrennt.
`data`	Zeit zu der über die *API* `/data` auf den Container aufgerufen wurde. Bei mehreren Aufrufen sind diese durch Komma getrennt.

Tabelle A.2: Anzahl der Stichproben pro Anzahl paralleler Container für das Messen der Ausführungsdauer

parallele Container	Anzahl der Stichproben
1	54
2	14
3	14
4	14
5	21
6	12
7	11
8	11
9	12
10	12
11	22
12	22
13	13
14	22
15	16
16	22
17	18
18	17
19	16
20	13
21	26
22	10
23	10
24	10
25	10
26	12
27	10
28	16
29	16
30	13
31	13
32	12
33	10
34	10
35	14
36	12
37	10

Tabelle A.3: Median der Ausführungsdauer pro Anzahl paralleler Container in Sekunden

parallele Container	Median der Ausführungsdauer in Sekunden
1	508.5
2	576.0
3	671.0
4	778.0
5	899.0
6	1021.0
7	1147.0
8	1277.0
9	1450.0
10	1645.0
11	1803.0
12	1973.0
13	2136.0
14	2308.0
15	2484.0
16	2638.0
17	2814.0
18	2983.0
19	3146.0
20	3318.0
21	3472.5
22	3624.0
23	3785.0
24	3933.0
25	4152.0
26	4301.0
27	4462.0
28	4645.0
29	4813.0
30	4980.0
31	5166.0
32	5280.0
33	5435.0
34	5601.0
35	5849.0
36	5950.0
37	6094.0

Tabelle A.4: Prozentualer Medianwert für CPU Auslastung, GPU Auslastung, RAM Nutzung und GPU Speichernutzung

parallele Container	CPU	RAM	GPU	GPU Speicher
1	13.4 %	4.7 %	8.0 %	2.6 %
2	24.3 %	6.8 %	40.0 %	5.2 %
3	38.7 %	9.2 %	83.5 %	7.8 %
4	99.9 %	34.1 %	99.0 %	83.3 %
5	63.5 %	14.4 %	99.0 %	13.0 %
6	75.7 %	16.5 %	99.0 %	15.6 %
7	88.1 %	19.0 %	100.0 %	18.2 %
8	99.8 %	21.4 %	100.0 %	20.8 %
9	99.5 %	23.8 %	99.0 %	23.4 %
10	99.6 %	26.7 %	100.0 %	26.0 %
11	99.8 %	29.2 %	100.0 %	28.6 %
12	99.9 %	31.6 %	100.0 %	31.2 %
13	99.9 %	34.0 %	100.0 %	33.8 %
14	99.9 %	36.5 %	100.0 %	36.4 %
15	99.9 %	38.7 %	99.0 %	39.0 %
16	100.0 %	41.3 %	100.0 %	41.7 %
17	100.0 %	43.7 %	99.0 %	44.3 %
18	100.0 %	46.1 %	99.0 %	46.9 %
19	100.0 %	48.6 %	99.0 %	49.5 %
20	100.0 %	50.9 %	99.0 %	52.1 %
21	100.0 %	53.4 %	99.0 %	54.7 %
22	100.0 %	55.7 %	99.0 %	57.3 %
23	100.0 %	58.2 %	99.0 %	59.9 %
24	100.0 %	60.6 %	99.0 %	62.5 %
25	100.0 %	62.9 %	99.0 %	65.1 %
26	100.0 %	65.3 %	99.0 %	67.7 %
27	100.0 %	67.8 %	99.0 %	70.3 %
28	100.0 %	70.2 %	99.0 %	72.9 %
29	100.0 %	72.7 %	99.0 %	75.5 %
30	100.0 %	75.0 %	99.0 %	78.1 %
31	100.0 %	77.4 %	99.0 %	80.7 %
32	100.0 %	79.9 %	99.0 %	83.3 %
33	100.0 %	82.3 %	99.0 %	85.9 %
34	100.0 %	84.7 %	99.0 %	88.5 %
35	100.0 %	87.1 %	99.0 %	91.1 %
36	100.0 %	89.5 %	99.0 %	93.7 %
37	100.0 %	91.9 %	99.0 %	96.3 %

Danksagung

Die Arbeit ist auf Grundlage des Bachelorprojekts *Online Marketplace Simulation a Testbed for Self-Learning Agents in Reommerce* im Wintersemester 2021/22 und Sommersemester 2022 am Enterprise Platform and Integration Concepts Lehrstuhl des Hasso-Plattner-Instituts an der Universität Potsdam entstanden.

Mein Dank gilt unseren Projektbetreuer Dr. Michael Perscheid, Dr. Rainer Schlosser, Alexander Kastius und Johannes Huegle für die Überlassung des Themas, sowie der mit am EPIC Lehrstuhl zuteil gewordenen Unterstützung.

Ferner bedanke ich mich bei meinem Team, bestehend aus Nick Bessin, Leonard Dreeßen, Jan Niklas Groeneveld, Nikkel Mollenhauer und Johann Schulze Tast, für die gemeinsame Arbeit am `recommerce` Framework.

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Bachelorarbeit „Skalierbares Lernen in der Cloud“ („Scalable Learning in the Cloud“) selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, den 30. Juni 2022,

_____

(Judith Herrmann)