HAW HAMBURG

Masterarbeit

Fabian Beck

Simulation und KPI-basierte Analyse von Geschäftsprozessen in Anwendungslandschaften

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Fabian Beck

Simulation und KPI-basierte Analyse von Geschäftsprozessen in Anwendungslandschaften

Mastertarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 28. Februar 2019

Fabian Beck

Thema der Arbeit

Simulation und KPI-basierte Analyse von Geschäftsprozessen in Anwendungslandschaften

Stichworte

Performance-Testen, Agentenbasierte Simulation, Anwendungslandschaft

Kurzzusammenfassung

Durch die verteilte Natur von Anwendungslandschaften und durch Fehler im Entwicklungsprozess von Systemen können während des Betriebs Fehler oder Flaschenhälse auftreten. Diese können schnell einen negativen Einfluss auf die Zufriedenheit der Nutzerbasis zur Folge haben. Eine Möglichkeit, um zu prüfen ob Probleme vorliegen, sind Performance-Tests. Dabei wird eine Anwendung oder ganze Anwendungslandschaften mit einer synthetischen Last belegt und währenddessen werden KPIs gesammelt. So können Fehler schon vor dem Produktivbetrieb entdeckt werden. In dieser Arbeit wurden zwei verschiedene Performance-Test-Szenarien verwendet, um zwei Anwendungslandschaften zu untersuchen. Die synthetische Last wurde in dieser Arbeit durch eine agentenbasierte Simulation generiert. Die implementierten Landschaften basierten beide auf einem beispielhaften Geschäftsprozess. Die eine Landschaft wurde HTTP-basiert und die andere Event-basiert implementiert. Während der Tests wurde für das Erfassen der Metriken Prometheus und der ELK Stack verwendet. Durch die Analyse der KPIs konnten in beiden Anwendungslandschaften Flaschenhälse identifiziert werden.

Fabian Beck

Title of Thesis

Simulation and KPI-based analysis of business processes in application landscapes

Keywords

performance-testing, agent-based simulation, application landscape

Abstract

The distributed nature of application landscapes and errors in the development process of systems can result in errors or bottlenecks in productive use. This can have a negative effect to the customer satisfaction and could lead to a negative impact on sales. Performance tests can be used to test systems or application landscapes and locate those problems before they are in a productive environment. During the tests a synthetic load profile is used to simulate the usage of the system. For analysis of the tests some KPIs are collected during the test. The KPIs can be used to find errors or bottlenecks before production. In this work two different performance test scenarios were used to test two application landscapes wich implemented the same business process. One landscape was based on HTTP and the other one was event-based. The synthetic workload in this work was generated with an agent-based simulation. The KPIs were collected with Prometheus and the ELK Stack. Due to the analysis of the KPIs it was possible to find bottlenecks in both landscapes.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ABM Agent-based-Model.

ABS Agent-based-Simulation.

BR Bottleneck Resource.

CR Capture and Replay.

CRM Customer-Relationship-Management.

EST Empty Semi Trucks.

GOAP Goal-Oriented Action Planning.

ICC Informatik Compute Cloud.

KPI Key Performance Indicator.

MARS Multi-Agent Research and Simulation.

MAS Multiagentensystem.

MBT Model-Based Testing.

MQ Message Queue.

OLB One Lane Bridge.

PTA Probabilistic Timed Automata.

PVC Persistent Volume Claim.

REST Representational State Transfer.

SUT System Under Test.

1 Einleitung

IT-Systeme lassen sich in der heutigen Zeit meist nicht mehr isoliert als einzelnes System betreiben. Stattdessen müssen sie häufig mit anderen Systemen interagieren, um beispielsweise einen Geschäftsprozess abbilden zu können. Dadurch entstehen Anwendungslandschaften, die über einige wenige Service bis zu hunderten oder sogar noch weit mehr Services enthalten können. Dies kann zum Beispiel auf die zugrunde liegende Architektur, die Verbreitung von Cloud-Plattformen oder das organische Wachsen der Anwendungslandschaft zurückgeführt werden. Beispielsweise entstanden bei der Gilt Group alleine durch das Auftrennen einer monolitischen Applikation in Microservices 156 Services [16].

Durch die Verteilung können im Fall von Microservices verschiedene Probleme von monolitischen Systemen gelöst werden. Allerdings entstehen dadurch auch neue Probleme. Organisch gewachsene Anwendungslandschaften haben jedoch meist viele Nachteile. Ein Hauptproblem des organischen Wachstums ist, dass die Dokumentation der gesamten Anwendungslandschaft mit der Zeit veraltet. Dabei kann das Wissen über die Zusammenhänge zwischen den einzelnen Anwendungen verloren gehen. Unabhängig vom Grund für die Verteilung von Systemen in Anwendungslandschaften, müssen die in den Anwendungslandschaften realisierten Geschäftsprozessen trotzdem in der durch die Nutzer geforderten Qualität bereitgestellt werden. So kann laut Schulz et al. und Chen et al. eine schlechte Performance einen nicht zu vernachlässigenden Einfluss auf den Umsatz eines Unternehmens haben [42, 15]. Um die nötige Service-Qualität in Anwendungslandschaften bieten zu können, ist es daher wichtig, die Anwendungslandschaften beispielsweise mithilfe von Performance-Tests zu analysieren.

In dieser Masterarbeit sollen Anwendungslandschaften mithilfe von Performance-Tests auf Basis von Agent-based-Simulation (ABS) untersucht werden. Insbesondere soll der Einfluss von synchroner HTTP-basierter Kommunikation bzw. asynchroner Event-basierter Kommunikation auf die Leistungsfähigkeit der Anwendungslandschaften mithilfe von verschiedenen Key Performance Indicators (KPIs) näher beleuchtet werden. Die KPIs

sollen hierfür mit zwei verschiedenen Messsystemen erhoben werden. Das Ziel der Arbeit ist das Auffinden von Flaschenhälsen innerhalb der Anwendungslandschaften und ein abschließender Vergleich, bei dem die eingesetzten Messsysteme bewertet werden.

Als Grundlage für die Anwendungslandschaften soll ein Geschäftsprozess einer beispielhaften Bank verwendet werden. Über diesen Geschäftsprozess können die Kunden der Bank die Überweisung eines Kredits auf ihr Konto beantragen. Um für die Simulation Last auf den Anwendungslandschaften während der Performance-Tests generieren zu können, soll Multi-Agent Research and Simulation (MARS) oder ein System, dem MARS zugrunde liegt, verwendet werden. Hierfür muss der Geschäftsprozess in einem Agentenmodell realisiert werden. In der Untersuchung sollen als Messsysteme Prometheus und der ELK Stack verwendet werden.

Diese Arbeit besteht aus sieben Kapiteln. Nach dieser Einführung, werden in Kapitel 2 Grundlagen die wichtigsten in dieser Arbeit behandelten Themengebiete vorgestellt. Dabei werden Agent-based-Model (ABM), Performance-Testing, dazu gehörende Forschungsergebnisse und in der Arbeit verwendete Frameworks beschrieben. Kapitel 3 Aufbau befasst sich mit dem Aufbau der verwendeten Systeme in der Informatik Compute Cloud (ICC). In Kapitel 4 Szenario wird der untersuchte Geschäftsprozess, seine Umsetzung in zwei Anwendungslandschaften und eine Umsetzung eines auf ihm basierenden Agentenmodells vorgestellt. Darüber hinaus werden die Test-Szenarien und die während der Tests gesammelten KPIs erläutert. Auf dieser Basis werden die wichtigsten Ergebnisse der Tests in Kapitel 5 Ergebnisse vorgestellt und beschrieben. In Kapitel 6 Diskussion werden die Ergebnisse der Tests genauer erläutert und analysiert. Mit Kapitel 7 Zusammenfassung und Fazit wird diese Arbeit abgeschlossen. Dabei wird die Arbeit zusammengefasst und ein Ausblick auf mögliche weitere Vorhaben gegeben.

2 Grundlagen

2.1 Agentenbasierte Simulation

Simulations- und Berechnungsframeworks, mit denen dynamische Prozesse unter Beteiligung von autonomen Agenten abgebildet oder simuliert werden können, werden als ABM oder ABS bezeichnet [33]. Über diese Frameworks werden beispielsweise die individuelle Entscheidungsfindung und soziales und organisatorisches Verhalten von Agenten abgebildet und anschließend simuliert [10]. ABM verfügen meist über drei wichtige Bausteine, die laut Macal und North die typischen Eigenschaften solcher Frameworks sind [31]:

- Das Framework arbeitet mit Agenten
- Agenten haben Beziehungen und können interagieren
- Agenten sind in eine Umgebung eingebettet

Die Existenz von Agenten ist dabei die wichtigste Eigenschaft in einem ABM, da diese in der Regel die aktiven Elemente einer Simulation darstellen. So starten sie beispielsweise Interaktionen mit anderen Entitäten während einer Simulation. Diese Interaktionen können mit einem anderen Agenten oder mit der Umgebung ausgeführt werden. Manche Modelle verfügen über spezielle Fälle, in denen Ereignisse oder Interaktionen durch die Umgebung angestoßen werden können. Dies kann beispielsweise vorkommen, wenn in einer Simulation zu einer bestimmten Zeit ein wichtiges Ereignis auftritt.

In einem Großteil der Literatur werden Agenten mit der Möglichkeit zur Ausführung von Interaktionen definiert [13, 10, 31]. Darüber hinaus teilen sich die Ansichten, welche Entitäten in einem Modell als Agenten bezeichnet werden sollen. Laut Bonabeau sind dies alle unabhängigen Komponenten in einem ABM [10]. Bei Casti werden nur die Komponenten, die ihr Verhalten adaptieren können, als Agenten bezeichnet [13]. In dieser Arbeit werden alle Komponenten, von denen Interaktionen ausgehen, als Agenten verstanden. Außerdem weisen sie die von Macal und North definierten, zentralen Agenteneigenschaften auf [31]:

- Agenten sind autonom und selbstgesteuert
- Agenten sind modular und in sich abgeschlossen
- Agenten sind sozial und kommunizieren untereinander
- Agenten besitzen einen zeitlich veränderbaren Zustand

Neben diesen zentralen Eigenschaften können Agenten nach Macal und North noch weitere Eigenschaften besitzen. Diese sind beispielsweise die Möglichkeit Ziele zu verfolgen, lernfähig reagieren zu können oder der Besitz von Ressourcen [31]. Die in dieser Arbeit verwendeten Agenten verfügen über die zentralen Eigenschaften ohne die Möglichkeit untereinander zu kommunizieren. Darüber hinaus können sie jedoch auch noch Ziele verfolgen und besitzen eigene Ressourcen.

Werden die in einem ABM definierten Agenten in einer ABS simuliert kann durch das Zusammenspiel der Agenten Emergenz entstehen. Diese Eigenschaft von ABS ist eines der zentralen Features von ABS [11, 32]. Selbst einfachste Agenten mit eingeschränkten Verhaltensmöglichkeiten können Emergenz während der Simulation hervorrufen [33]. Bei der Simulation von komplexen Agenten entsteht oft Emergenz, welche nicht immer erwünscht ist. Eine erwünschte Emergenz wäre der Fluss einer Menschenmasse bei der Evakuierung eines Schiffes in einer Schiffs-Evakuierungs-Simulation[49]. Treten unerwünschte Muster in komplexen ABS auf, können diese den Einsatz des der Simulation zugrunde liegenden ABM erschweren oder sogar ganz verhindern.

2.2 Performance-Testing

Um ein System hinsichtlich seiner Leistungsgrenzen zu untersuchen, können Performance-Tests verwendet werden. Das kann sinnvoll sein, wenn festgestellt werden soll, ob ein System die geplante Anzahl von Nutzern oder Zugriffen bewältigen kann [46]. Daher werden bei einem Test Anfragen an das System Under Test (SUT) gestellt, sodass Last generiert wird und verschiedene Metriken gemessen werden, um eine quantitative Auswertung zu ermöglichen. Mithilfe von Performance-Tests werden funktionale und nicht-funktionale Anforderungen überprüft, um Missstände von Anforderungen aufzuzeigen [46, 15]. Es ist z. B. möglich Dead-Locks, die bei hoher Parallelität auftreten können, zu entdecken [15]. Es kann aber auch überprüft werden, ob in Anforderungen festgelegte Antwortzeiten eingehalten werden. Das Überprüfen von nicht-funktionalen Anforderungen, wie der Antwortzeit, darf nicht vernachlässigt werden, da beispielsweise bei Webshops

jede Sekunde längere Ladezeit ein schlechteres Ergebnis bei der Kaufentscheidung der Kunden nach sich ziehen kann [15].

Es gibt verschiedene Aspekte, die mit unterschiedlichen Arten von Performance-Tests untersucht werden können. Aus dieser unterschiedlichen Zielsetzung der Tests ergeben sich unterschiedliche Konstellationen bezügliche generierter Anfragen, der Testdauer, des Ziel-Systemzustands und der erhobenen Metriken [35]. In der Literatur werden eine Reihe dieser Test-Arten beschrieben, wobei für einzelne Definitionen je nach Autor unterschiedliche Namen verwendet werden [35, 52, 34, 14]. Um in dieser Arbeit eine einheitliche Terminologie zu verwenden, werden die verwendeten Test-Arten mit Bezügen zur Literatur definiert.

Volume-Test

Bei dieser Test-Art wird die geplante Auslastung eines Systems getestet. Um dies zu erreichen, wird die im realen Betrieb erwartete Menge an parallelen Anfragen pro Sekunde generiert und an das SUT gesendet. Diese Art der Performance-Tests entspricht am ehesten der alltäglichen Nutzung eines Systems in der Realität. Um die Anfragen möglichst realitätsnah gestalten zu können, werden für diese Tests meist reale Think Time und Pacing verwendet. Diese Performance-Tests werden auch als Load-Tests bezeichnet [34]. [35]

Stress-Test

Mithilfe von Stress-Tests werden Systeme auf ihre Grenzen hin getestet. Hierfür werden während eines Tests mit fortschreitender Zeit immer mehr Anfragen an das SUT gestellt. Die Anzahl der parallelen Anfragen kann hierbei weit über die geplante Auslastung hinaus skaliert werden [14]. Diese Tests werden entweder für eine definierte Zeit ausgeführt oder bis das SUT unter der Last der Anfragen versagt. Zeichen für das Versagen eines System können die Überschreitung einer bestimmten Antwortzeit sein oder ein Timeout des Systems. Stress-Tests werden in der Literatur einheitlich benannt und beschrieben. [35, 34]

Stability-Test

Mit diesen Tests wird das Verhalten eines System während einer langen Laufzeit untersucht. Ein solcher Test kann mehrere Stunden bis Tage dauern [14]. So können beispielsweise Memory Leaks oder unvorhergesehene Limitationen bei der Ausführung eines Use-Cases identifiziert werden. [34]

Die Last bei der Ausführung eines Tests kann auf unterschiedlichen Wegen generiert werden. Zwei dieser Wege sind Capture and Replay (CR) und Model-Based Testing

(MBT). Bei CR basierten Ansätzen wird reale Last aufgenommen und gespeichert, um sie während des Tests wieder abspielen zu können. Die Aufzeichnung für solche Tests kann im realen System mit realen Anfragen erfolgen. Es können jedoch auch Use-Cases ausgeführt werden, um sie für spätere Tests aufzunehmen. Wird MBT verwendet, müssen keine Lastprofile aufgezeichnet werden, sondern es wird ein Modell für die Generierung von Anfragen erstellt und während der Tests ausgeführt. [9]

Um die in einem Performance-Test verwendeten Lastmodelle besser beschreiben zu können, werden zwei allgemeine Größen verwendet. Diese beziehen sich auf die Menge von Anfragen, die während eines Tests an das System gestellt werden. Sie werden unabhängig von der Art der Lastgenerierung angewendet. So können Vergleiche zwischen einzelnen Tests, wie z. B. der Menge der Anfragen pro Sekunde, angestellt werden.

Think Time

Think Time repräsentiert die Wartezeiten und Pausen die durch einen Nutzer verursacht werden, wenn er mit dem SUT interagiert. So überlegt sich ein Ebay-Nutzer beispielsweise welche Höhe sein Gebot haben soll, bevor er es über die Webseite abschickt. Anhand dieser Größe kann die Realität besser in der Simulation abgebildet werden, indem reale Zeiten verwendet werden. [34]

Pacing

Pacing ist im Vergleich zur Think Time nicht die Wartezeit zwischen einzelnen Aktionen, sondern der Abstand der Wiederholung ganzer Aktionsfolgen. Das kann gesteuert werden, indem die Wartezeit zwischen Wiederholungen einer Aktionsfolge definiert wird. [34]

2.3 Related Works

In der Literatur gibt es eine Vielzahl an Arbeiten zum Thema Performance-Testen. Ein Survey über Performance- und Load-Testing von Jiang und Hassan beschreibt die verschiedenen Wege, auf denen diese Test-Arten verwendet werden können und welche Entwicklungen bis 2015 vorhanden waren [28]. Ein weiterer Survey von Arora und Bhatia stellt die Fortschritte agentenbasierter Tests und verschiedene Wege, auf denen Last generiert werden kann, dar [2]. In den Surveys wurde gezeigt, dass diese Art zu testen ein sehr breites Spektrum umfasst. Aufgrund der in der vorliegenden Arbeit verwendeten Systeme und Frameworks wird die Thematik auf Arbeiten mit Peer-to-Peer-artigen Generatoren und MBT-basiertem Workload eingeschränkt.

Ahamand et al. haben beschrieben, wie sie, auf der Grundlage von Probabilistic Timed Automata (PTA), ein Tool für Performance-Testing entwickelten. Über das Tool kann Last anhand von Eingabeparametern erstellt, KPIs gesammelt und ein abschließender Testbericht erstellt werden. In einem PTA können die Übergänge zwischen verschiedenen Aktionen eines Benutzers modelliert werden. Die Lastgenerierung während eines Tests geschieht durch die Simulation einzelner Nutzer, die jeweils den PTA durchlaufen. So versuchten sie ein reales Verhalten nachzuahmen. Das Tool verfügt über verschiedene Komponenten. Einen Master-Knoten, der die Simulation verwaltet und auswertet und beliebig viele Slave-Knoten, die die einzelnen Nutzer simulieren und Anfragen an das SUT stellen [1]. Auch Hao et al. haben ein Framework entwickelt, mit dessen Hilfe Anfragen in einem verteilten System, das Nutzer simuliert, generiert werden. In diesem Framework gibt es einen zentralen Manager, der den Nutzern Aufgaben zuweist, die sie während des Tests verfolgen. Darüber hinaus sammelt der Manager die durch den Test erstellten Daten, um sie den testenden Benutzern aufbereitet anzeigen zu können [21].

Während der Tests ist es laut Jiang und Hassan wichtig, einen möglichst realistischen Workload zu generieren. Um dies zu ermöglichen, kann der Workload Use-Case-basiert erstellt werden [28]. Es gibt aber auch weitere Ansätze, in denen die Basis für das Workload-Modell beispielsweise Logs aus der realen Nutzung des SUT sind [15, 24, 41]. Unabhängig vom gewählten Weg haben Huaji und Huarui beschrieben, wie wichtig es ist die Aktionen aus denen der Workload besteht, in einer sinnvollen Reihenfolge ablaufen zu lassen [25]. Für einen noch realistischeren Workload hat Ramakrishnan einen Algorithmus entwickelt, über den die Think Time der simulierten Nutzer anhand von vorhandenen Logs bestimmt werden kann [39]. So hat er beschrieben, dass in vielen Tests nur ein gewisses Aktionsverhältnis eingehalten wird, wodurch im SUT kein normaler nutzerbasierter Prozess abläuft. Diese Arbeit sich auf eine durch Agenten repräsentierte, nutzerzentrierte Simulation für die Generierung eines möglichst realistischen Workloads.

Für das Testen mithilfe von Agenten werden im Allgemeinen die drei Tools JADE, Aglet und Springs verwendet. Es gibt noch weitere Ansätze, die jedoch nur selten in Studien eingesetzt werden. Von den drei aufgezählten Tools wird JADE in den meisten Fällen verwendet [2]. JADE ist ein in Java implementiertes Framework, mit dessen Hilfe Agenten verteilt simuliert werden können [8]. Laut Ma et al. ist die Verwendung von agentenbasierten Simulationen für Tests sinnvoll, da Agenten autonom, kollaborativ und reaktiv agieren können. Dies haben sie in ihrer Studie zu adaptiven Performance-Testing von Web Services beschrieben. Dabei haben sie ein System vorgestellt, mit dem sie die Agentensimulation für einen Performance-Test umgesetzt haben [30].

Xu beschreibt in einer Studie die Verwendung von JADE als Basis für ein Performance-Testing System. Dabei zeigte Xu auf wie das verwendete System im Ganzen aufgebaut ist. Die JADE Agenten werden auf einzelnen Simulationsknoten simuliert. Diese sogenannten Last-Generatoren wurden in einer zentralen Instanz verwaltet [5]. Ma et al. haben ebenfalls eine Studien zu agentenbasierten Test Szenarien veröffentlicht.

Jiang und Hassan haben in ihrem Survey auch die in vielen Studien gesammelten Metriken betrachtet und haben die drei am häufigsten verwendeten Metriken identifiziert[28]. Dabei handelt es sich um die Anzahl von erfolgreichen und fehlgeschlagenen Anfragen, die Ende-zu-Ende Antwortzeiten und den Ressourcenverbrauch. Ansonsten werden in vielen Studien Applikations-Logs gesammelt und ausgewertet. Hierfür müssen die SUT jedoch sinnvolle Logs generieren, die im Anschluss ausgewertet werden können.

Für die Analyse von Flaschenhälsen gibt es einige Arbeiten, die die Grundlagen hierfür legen [44, 45, 43, 17]. So wurde in diesen Arbeiten definiert, welche Ursachen Flaschenhälse haben können und was ein Entwickler dagegen tun kann. In den letzten Jahren wurden diese Arbeiten wieder aufgegriffen und dafür verwendet, mithilfe der allgemein gesammelten Metriken Flaschenhälse in Systemen zu identifizieren. Wert et al. haben in verschiedenen Arbeiten gezeigt, wie dies automatisiert geschehen kann [50, 51].

2.4 Messsysteme

2.4.1 ELK Stack

Mit der Bezeichnung ELK Stack wurden ursprünglich die drei Applikationen Elasticsearch, Logstash und Kibana beschrieben [40, 18]. 2015 wurde der ELK Stack um Beats erweitert [18]. Mit dieser Erweiterung steht der ELK Stack für eine allumfassende Lösung, um Applikationen zu überwachen. Der Nutzen des ELK Stack ist jedoch nicht nur auf das Sammeln von Logs und Monitoring beschränkt. Einzelne Komponenten können auch für andere Zwecke verwendet werden. Das ist z. B. bei Elasticsearch der Fall, da neben Logs auch andere Daten gespeichert werden können.

Abbildung 2.1 zeigt die aktuellen Komponenten des ELK Stacks. Im Folgenden werden die einzelnen Komponenten näher beschrieben, um deren Anwendungsbereiche besser abgrenzen zu können.

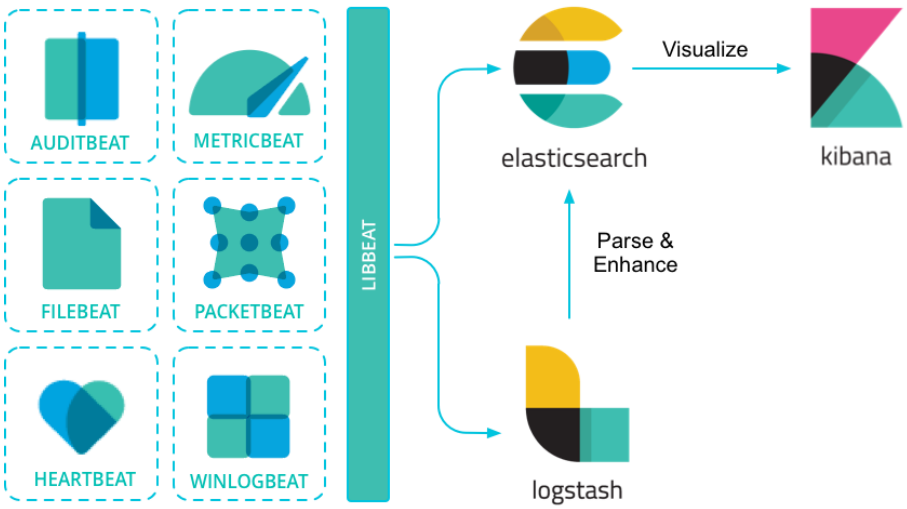


Abbildung 2.1: ELK Stack Übersicht [18]

Beats

Elastic bietet mit der Beat-Plattform ein Go-Framework an, mit dem leichtgewichtige Applikationen erstellt werden können, die Daten in den ELK Stack einspeisen [20]. Eine auf dem Beat-Framework basierende Applikation wird auch Beat genannt. Elastic bietet aktuell verschiedenste Beats an, davon sind sechs auf der linken Seite in Abbildung 2.1 aufgeführt. Daneben gibt es eine Vielzahl von durch die Community erstellten Beats. Ein Beat hat die Möglichkeit, Daten erst an Logstash zu schicken, wenn sie vor der Indexierung in Elasticsearch noch verarbeitet werden sollen. Er kann sie aber auch direkt an Elasticsearch schicken[40]. [18]

Logstash

Logstash ist eine Java Applikation mit deren Hilfe Datenströme verarbeitet und normalisiert werden können, bevor sie in Elasticsearch eingespeist werden [37, 3].

Elasticsearch

Elasticsearch ist eine in Java entwickelte und auf Apache Lucence basierende verteilte Datenbank [4]. Elasticsearch speichert JSON-Dokumente und wurde speziell dafür entwickelt, semi-strukturierte Daten zu analysieren und zu durchsuchen [20]. Die gespeicherten Daten und Ergebnisse werden von Elasticsearch über eine Representational State Transfer (REST)-API zur Verfügung gestellt. Im Normalfall wird Elasticsearch in einem Cluster betrieben. In dieser Betriebsart werden die

Daten zwischen den verschiedenen Elasticsearch-Knoten verteilt und gespiegelt, sodass eine Ausfallsicherheit gegeben ist. [18]

Kibana

Kibana ist das Visualisierungstool des ELK Stacks. Es ist Web-basiert und eng an Elasticsearch gebunden. Mithilfe von Kibana können auf Elasticsearch basierende Dashboards erstellt werden. Es bietet aber auch eine Oberfläche für X-Pack Features. Unter X-Pack sind verschiedene Erweiterungen des ELK Stacks zusammengefasst, die beispielsweise den Betrieb durch Monitoring der einzelnen Komponenten erleichtern, oder neue Möglichkeiten der Datenanalyse mithilfe von AI bieten. [18]

2.4.2 Prometheus

Prometheus ist ein Open-Source Monitoring und Alerting Toolkit, das ursprünglich bei Soundcloud entwickelt wurde. Nach der Veröffentlichung wurde Prometheus der Cloud Nativ Computing Foundation hinzugefügt und war dort nach Kubernetes das zweite Projekt. [38]

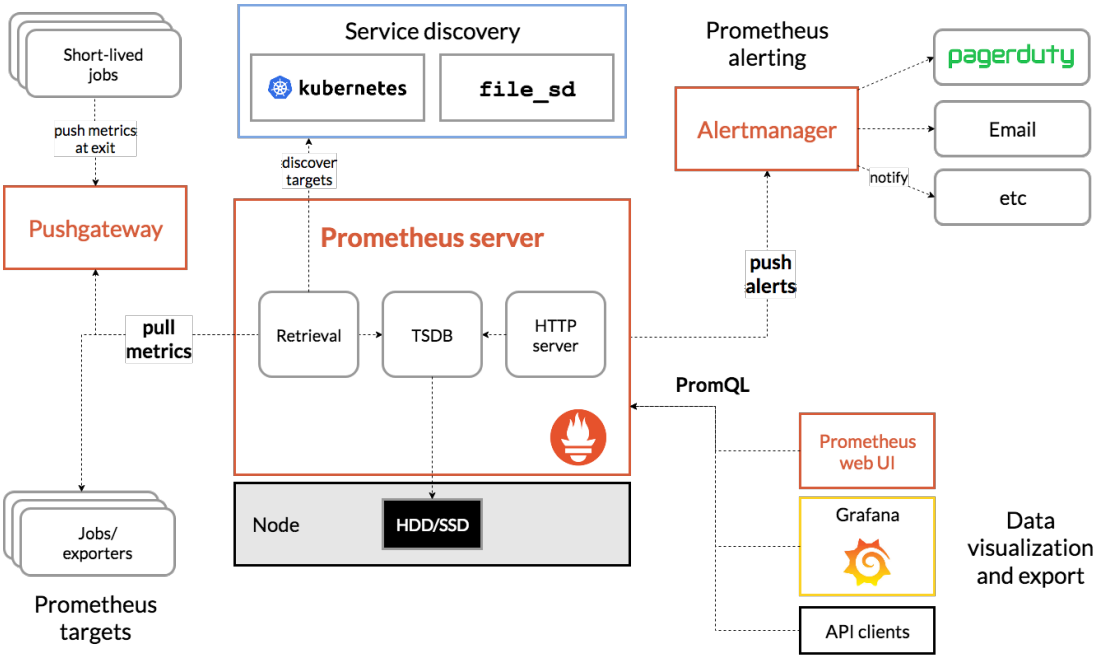


Abbildung 2.2: Übersicht des Prometheus Ökosystems [38]

Prometheus ist, wie in Abbildung 2.2 dargestellt, als zentraler Server konzipiert, der das Monitoring und Alerting verwaltet [12]. Für das Alerting ist ein zusätzlicher Manager vorhanden, über den verschiedenste Protokolle angesprochen werden können. Um gesammelte Metriken darzustellen und erste Auswertungen durchzuführen, besitzt der Prometheus Server eine integriertes Web UI. Für komplexere Use-Cases wie Dashboards ist das integrierte UI jedoch nicht mehr ausreichend und muss durch ein weiteres Tool wie beispielsweise Grafana ersetzt werden [12]. Prometheus sammelt Daten über des „Pull"-Prinzip. Das bedeutet, dass der Prometheus Server innerhalb eines festgelegten Intervalls von allen Zielen, von denen Metriken erhoben werden sollen, die bereitgestellten Daten abruft. Dieser Vorgang wird „scrapping" genannt. Für das Scrapping muss jedes Ziel eine REST-API anbieten, über die Metriken bereitgestellt werden. Ziele können entweder mithilfe von externen Discovery Services wie beispielsweise Kubernetes oder direkt per Konfiguration eingerichtet werden. [38]

Das Datenmodell von Prometheus basiert auf Zeitreihen [12]. Hierbei haben die Zeitstempel der Zeitreihen eine Auflösung von einer Millisekunde. Die Metriken werden immer mithilfe von „float64" gespeichert. Der Name einer Metrik sollte laut Prometheus immer so gewählt werden, dass er die zu messende Größe und das System, in dem sie gemessen wird, enthält. Um Metriken besser analysieren zu können, gibt es darüber hinaus die Möglichkeit, Werte mit beliebigen Labels zu versehen. Labels sind hierbei Key/Value Paare. Prometheus unterstützt verschiedene Typen von Metriken. [38]

Die drei in dieser Arbeit verwendeten Typen sind:

Counter
: Dies ist eine Metrik, die durch einen monoton steigenden Zähler repräsentiert wird. Der Zähler kann nur erhöht oder bei einem Neustart auf Null zurückgesetzt werden. [38]

Gauge
: Dabei handelt es sich um eine Metrik, die einen numerischen Wert repräsentiert. Im Vergleich zum Counter kann dieser Wert jedoch größer oder kleiner werden. [38]

Histogramm
: Diese Metrik beschreibt Beobachtungen, wie beispielsweise die Dauer einer Anfrage und teilt diese konfigurierbaren Buckets zu. Daneben bietet sie eine Summe über alle beobachteten Werte, die über einen Counter erfasst wurden. [38]

2.5 Frameworks

2.5.1 Application Framework

Das Application Framework wurde in vorangegangenen Arbeiten erstellt und bietet die Möglichkeit, gleichförmige Services für ein Microservice basiertes System auf der Basis von GoLang zu erstellen. Hierbei müssen, wie in einer serverlosen Umgebung, nur Funktionen definiert werden. Um diese Funktionen verwenden zu können, müssen sie dann verschiedenen Services zugeordnet werden. Damit ist es möglich, alle Services ohne großen Aufwand mit Monitoring- und Logging-Funktionalität auszustatten. Außerdem muss nur die Applikations-Logik erstellt werden, da die Infrastruktur durch das Framework bereitgestellt wird. Die Zusammensetzung eines Services kann hierbei über eine Konfigurationsdatei erstellt werden.

Um mithilfe dieses Frameworks einen Service zu implementieren, müssen in einer Application-Komponente Funktionen erstellt werden, die die Geschäftslogik enthalten. In einer Application-Komponente werden Funktion-Interfaces implementiert. In der verwendeten Version können Jobs, REST-Handler und Message Queue (MQ) -Handler definiert werden. Monitoring und Logging wird über sogenannte Observer-Komponenten bereitgestellt. Dies geschieht ebenfalls über definierte Interfaces, wodurch Observer erweitert werden können. Die Zusammenführung der Observer-Funktionen und der Application-Funktionen geschieht per Konfiguration über das Decorator-Pattern. In der aktuellen Version des Frameworks sind Observer für Logging, Prometheus und Zipkin vorhanden.

Abbildung 2.3 zeigt den Initialisierungsablauf eines Services, der auf Basis des Application Frameworks erstellt wurde. Für ein besseres Verständnis der Zusammenhänge werden die Aufgaben der wichtigsten Komponenten im Folgenden kurz beschrieben.

Morph

Dies ist die zentrale Komponente des Frameworks. Durch sie wird die Initialisierung der anderen Komponenten gesteuert.

Application Manager

Mit dieser Komponente werden die einzelnen Application-Komponenten verwaltet. Sie identifiziert alle benötigten Komponenten, initialisiert sie und reicht die vorhandenen Funktionen an die Morph-Komponente weiter. So können Funktionen für die Geschäftslogik dekoriert und zugreifbar gemacht werden.

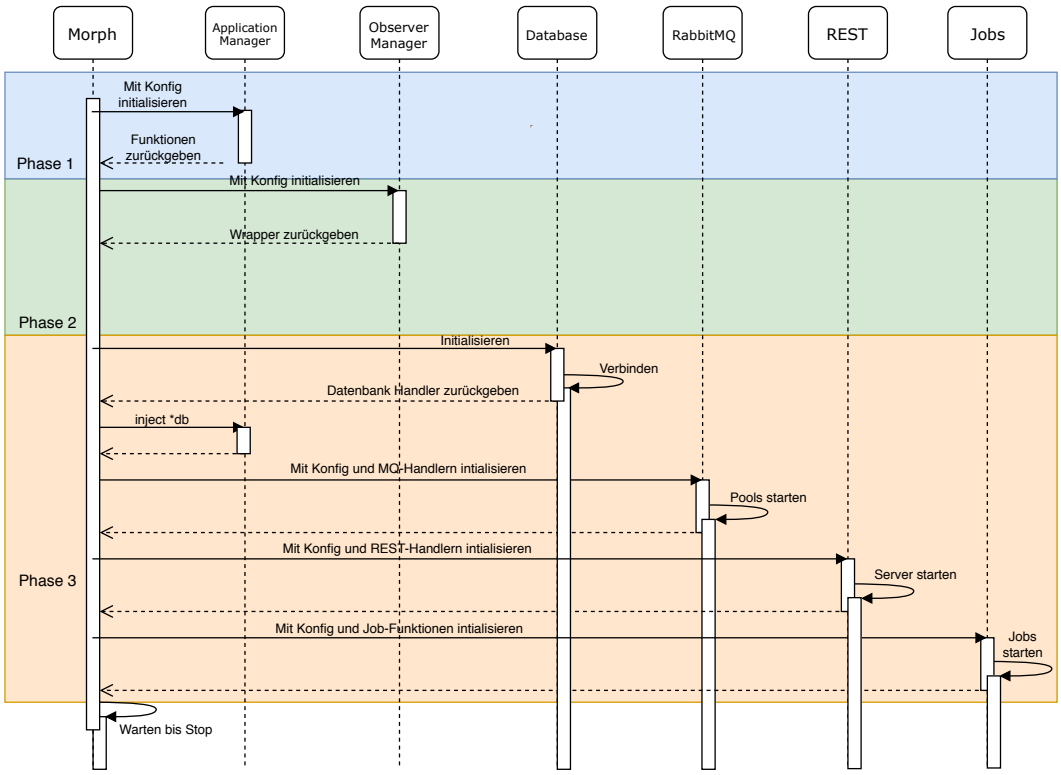


Abbildung 2.3: Ablauf der Initialisierung

Observer Manager

Diese Komponente hat die gleiche Aufgabe wie die Application-Manager-Komponente, mit dem Unterschied, dass sie Observer-Komponenten verwaltet.

Database

Über diese Komponente wird dem Framework eine Verbindung zu einer Datenbank bereitgestellt. In der aktuellen Version wird nur eine MySQL Datenbank unterstützt.

REST

Über die REST-Komponente kann eine REST-API zur Verfügung gestellt und konsumiert werden. Mithilfe der definierten REST-Handler wird per Konfiguration die REST-API erstellt.

RabbitMQ

Diese Komponente stellt, wie die REST-Komponente, eine Möglichkeit zur Kommunikation mit anderen Services zur Verfügung. Hierfür bietet sie dem Framework eine Möglichkeit, Nachrichten über RabbitMQ zu senden. Für die Verarbeitung von Nachrichten werden die implementierten und konfigurierten MQ-Handler verwendet.

Für die Verarbeitung von über RabbitMQ empfangene Nachrichten stellt diese Komponente einen Worker-Pool zur verfügung. In der verwendeten Version des Frameworks konnte die Größe dieses Pools jedoch nicht automatisch skaliert werden. Damit konnte die Größe des Pools nur durch einen Eintrag in der Konfiguration geändert werden kann.

Job

In dieser Komponente wird der zeitabhängige Start von Jobs gesteuert. Um Jobs in einem Service verwenden zu können, müssen Job-Funktionen implementiert werden.

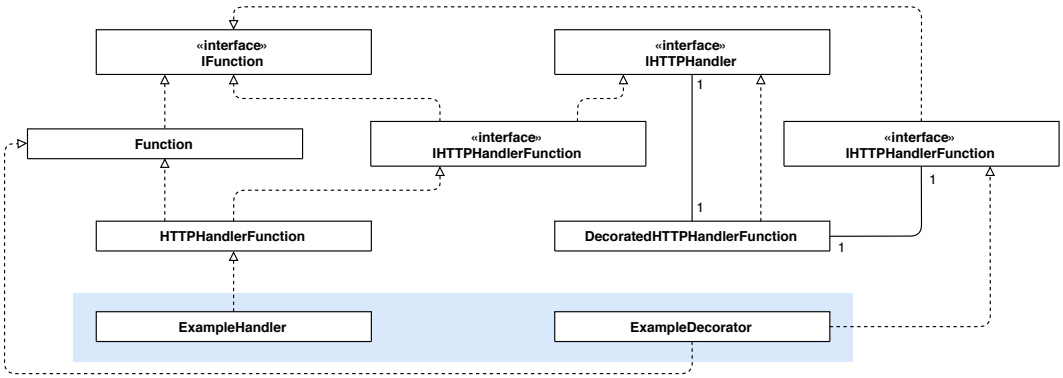


Abbildung 2.4: Decorator Zusammenhänge

Abbildung 2.4 zeigt die Implementierung des Decorator-Patterns anhand des REST-Handler Interfaces. Das Pattern muss für jedes Funktions-Interface separat implementiert werden, da GoLang aktuell keine Generics unterstützt. In blau sind die Klassen hinterlegt, die entweder in einer Application- oder einer Observer-Komponente implementiert werden müssen. Durch die Klasse „DecoratedHTTPHandlerFunction“ wird ein Decorator mit der zu dekorierenden Funktion verbunden. Da die dekorierte Funktion wieder das benötigte Decorator-Interface anbietet, können Funktionen beliebig oft dekoriert werden. Durch das „IFunction“ Interface und die dazugehörige Klasse werden Tags für Funktionen und Dekorationen zur Verfügung gestellt. So kann in der Konfiguration gesteuert werden, welche Funktion mit welchen Dekorationen ausgestattet werden soll.

In der Konfiguration können neben den Vorgaben zum Dekorieren der Funktionen noch weitere Parameter hinterlegt werden, die für das Funktionieren eines mit dem Framework erstellten Services unabdingbar sind. So müssen die Funktionen definiert werden, die im Service verwendet werden sollen. Dabei können Parameter an die Funktionen übergeben werden. Um eine REST-API anzubieten, können die zu den Handler-Funktionen gehörenden Pfade und der Port, auf dem die API angeboten werden soll, konfiguriert werden.

Um Nachrichten zu empfangen, müssen die Handler auf die gewünschten Nachrichten-Schlüssel gemappt werden. Darüber hinaus können noch einige spezifische RabbitMQ Einstellungen vorgenommen werden. Wird der Service in Kubernetes verwendet, kann über die Konfiguration Probing bereitgestellt werden. Soll eine Verbindung zu einer Datenbank aufgebaut werden, kann die Datenbank ebenso über die Konfiguration definiert werden.

2.5.2 Agent Framework

In vorangegangenen Arbeiten wurde ein Framework für die Umsetzung eines ABM implementiert [6, 7]. In diesem Framework kann eine ABS mit C# erstellt werden. Dazu ist nur die Implementierung der modellspezifischen Logik nötig. Die restlichen Komponenten werden durch das Framework bereitgestellt. Um möglichst wenige Änderungen im Code vornehmen zu müssen, können Agenten umfassend über eine Konfigurationsdatei angepasst werden. Eine Simulation mit diesem Framework kann über MARS oder über eine extra entwickelte rudimentäre Umgebung ausgeführt werden. Das MARS System wird an der Hochschule für Angewandte Wissenschaften Hamburg durch die MARS-Forschungsgruppe entwickelt [26]. Es ist ein cloud-basiertes massives Multiagentensystem (MAS) [27]. Die rudimentäre Simulationsumgebung wurde entwickelt, um einfache Simulationen ausführen zu können, die die vielfältigen Möglichkeiten von MARS nicht benötigen.

Das Framework setzt sich aus den in Abbildung 2.5 nicht blau hinterlegten Komponenten zusammen. Die blau umschlossenen Komponenten müssen für die Realisierung eines ABM mit dem Framework implementiert werden. Alle in grün eingefassten Komponenten gehören zur im Framework umgesetzten Agentenabstraktion. Die orange eingefassten Komponenten sind für das Ausführen von Interactions zuständig.

Knowledge

Diese Komponente ist für das Verwalten von Agenten-Daten während der Simulation verantwortlich. So werden beispielsweise die Zustände der Agenten durch diese Komponente verwaltet. Dies erfolgt in der aktuellen Version des Frameworks in Form einer Baumstruktur. Dabei ist es möglich Zustandsvariablen dynamisch zu ändern oder neu zu erstellen.

Planer

In dieser Komponente werden, auf Basis des aktuellen Zustands, Aktionen und Ziele ausgewählt, die ausgeführt und verfolgt werden sollen. Für die Auswahl wird Goal-Oriented Action Planning (GOAP) verwendet. Um auf den aktuellen Zustand

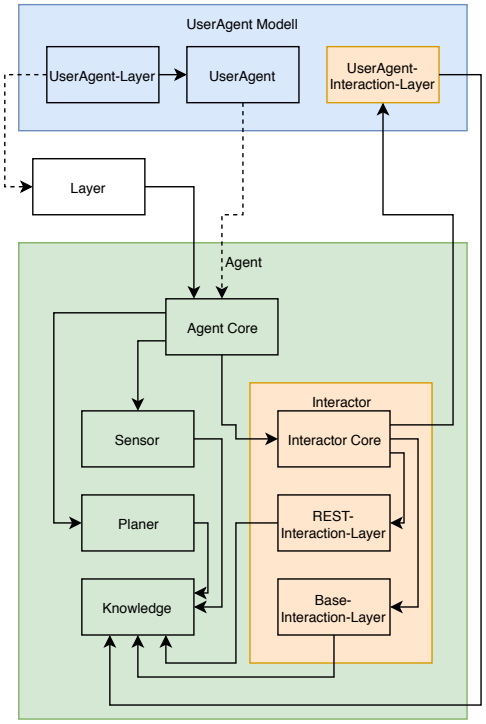


Abbildung 2.5: Abstrahierter Zusammenhang der Komponenten des Agent Frameworks

und alle vorhandenen Aktionen zugreifen zu können, hat diese Komponente Zugriff auf die Knowledge-Komponente.

Interaction Core

Dies ist die dritte Komponente ohne deren Funktionalität ein Agent nicht simuliert werden könnte. Sie übersetzt die durch die Planer-Komponente ausgewählten Actions in Interactions. Hierfür benötigt auch diese Komponente Zugriff auf die Knowledge-Komponente, da Interactions auf Basis des aktuellen Zustandes ausgeführt werden.

Sensor

Über diese Komponente werden alle Zustandsänderungen der Agenten vorgenommen. Aus diesem Grund benötigt sie Zugriff auf die Knowledge-Komponente. Im Zuge der Änderungen am Agentenzustand verwaltet sie auch alle aktuell ausgeführten Interactions und überführt ihre Ergebnisse bei Beendigung in den Agentenzustand. Hierfür hat die Sensor-Komponente Zugriff auf die Interaction-Komponente.

Base-Interaction-Layer

Diese Interaction-Layer stellt die Basisfunktionalität für die Simulation eines Agenten

bereit. Darunter fallen Interactions, wie das „Warten“ und das „Beenden eines Agenten“.

REST-Interaction-Layer

Auf Basis dieser Interaction-Layer können REST Interactions implementiert und ausgeführt werden. Hierfür enthält die Komponente beispielsweise eine Sensor-Schnittstelle, die REST Antworten verarbeiten kann.

Agent Core

Der Agent Core entspricht dem Bindeglied zwischen den Komponenten des Frameworks und hat deshalb Zugriff auf alle wichtigen Komponenten. In dieser Komponente werden die ausgewählten Interactions gestartet.

Layer

Diese Komponente hat keine Verbindung zu den Interaction-Layers. Ihr Ursprung liegt bei MARS, da dort Agenten oder Umwelteigenschaften in Layer gegliedert werden. Layer-Komponenten sind eine Abstraktion dieses Konzeptes und werden für die Verwaltung der Agenten während der Simulation implementiert. So können beispielsweise neue Agenten erstellt der Ereignisse in der Simulation ausgelöst werden.

Mit diesem Framework wird die Agentenlogik in einer Interaction-Layer implementiert und für das Framework bereitgestellt. Alle wichtigen Komponenten bieten Interfaces, über die sich neue Komponenten in den jeweiligen Ablauf einklinken können. Die Sensor-Komponente bietet ein Sensoren-Interface, über das Interaction-Layer-Sensors implementiert werden können. Diese werden in den Sensor-Zyklus eingebunden und können so auf für die Layer spezifische Dinge reagieren.

Der Ablauf einer ABS lässt sich in Ticks gliedern. Ein Tick entspricht hierbei dem Verstreichen einer festgelegten konstanten Zeit in der Simulation. Während eines Ticks werden der Status und die nächste Aktion jedes Agenten errechnet. Wie lange die Berechnung eines Ticks in einer normalen ABS benötigt, hat keinen Einfluss auf die Zeit, die ein Tick in dem Simulations-Modell darstellt. In einer Simulation mit dem Framework ist die vergangene Zeit an den Fluss der realen Zeit gebunden. Deshalb entspricht ein Framework-Tick in der Regel der Zeit, die für die Berechnung des Ticks benötigt wird. Um über das Framework trotzdem eine annähernd konstante Tick-Zeit zu ermöglichen, haben Ticks eine Mindestdauer. Um die geplante Tick-Dauer nicht zu oft zu überschreiten, muss darauf geachtet werden, nicht zu viele Agenten mit zu wenig Ressourcen zu simulieren.

Der Tick eines Agenten auf Basis des Frameworks läuft nach einem festen Schema ab. Die einzelnen Schritte spiegeln teilweise die Komponenten des Frameworks wieder. Abbildung 2.6 zeigt den Ablauf auf einer hohen Abstraktionsebene.

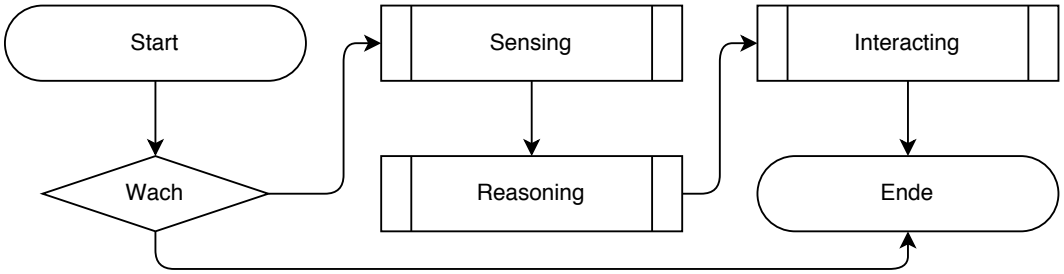


Abbildung 2.6: Der abstrakte Ablauf eines Ticks im Agent Framework

Der Tick eines Agenten wird durch die Simulationsumgebung gestartet. Hierfür wird auf die Komponente Agent Core zugegriffen, da sie die zentrale Komponente des Agenten ist und in ihr die Logik eines Ticks implementiert ist. Von ihr wird daraufhin als erstes das „Sensing“ gestartet, das auf der Sensor-Komponente basiert. Hier werden alle Änderungen, die beispielsweise durch das Beenden einer Interaktion oder das Erreichen von Zielen verursacht wurden, in den Zustand des Agenten übertragen. Das ist der einzige Zeitpunkt während eines Ticks, in dem der Agentenzustand verändert werden kann. So werden alle Entscheidungen in einem Tick auf der gleichen Grundlage getroffen. Wurde keine für die Simulation wichtige Änderung des Zustandes festgestellt, wird der restliche Tick des Agenten nicht mehr berechnet, da in diesem Fall keine sinnvolle neue Aktion ausgewählt werden kann. Nach einer Zustandsänderung wird mit dem „Reasoning“ fortgefahren. Hierfür wird die Planer-Komponente verwendet. Während des „Reasonings“ wird erst, wenn nötig, ein neues Ziel und danach eine passende Aktion ausgewählt. Die ausgewählte Aktion wird daraufhin in der „Interacting“-Phase über die Interaction-Komponente einer Interaktion zugeordnet, die anschließend gestartet wird. Damit ist der Tick des Agenten beendet.

3 Aufbau

Für die Untersuchung in dieser Arbeit müssen verschiedene Voraussetzungen erfüllt sein. Neben den zentralen Punkten, wie einem Geschäftsprozess, dem darauf basierenden Agentenmodell und der daraus resultierenden Anwendungslandschaft, werden noch weitere Komponenten benötigt, um die Untersuchung durchzuführen. So wird eine Umgebung benötigt, in der die Simulation ausgeführt und in die die Anwendungslandschaft eingebettet werden können und in der ausreichend Ressourcen zur Verfügung stehen. Zusätzlich muss auch ein Konzept für den Systemaufbau erstellt werden. Darüber hinaus werden Systeme für das Sammeln und Speichern der generierten Daten benötigt, um daraus KPIs berechnen zu können. Hierfür muss feststehen, auf welche Weise aus gesammelten Daten KPIs berechnet werden. Für ein besseres Verständnis der Untersuchung werden diese Punkte im folgenden Kapitel näher erläutert.

3.1 Umgebung

Das AI Labor der HAW Hamburg stellt auf Grundlage von Kubernetes die ICC zur Verfügung. Diese ist mit dem ebenfalls vom AI Labor betriebenen GitLab verbunden. So hat jeder Benutzer und jedes Projekt in GitLab einen eigenen Namespace in der ICC. Namespaces, die einem Projekt in GitLab zugeordnet sind, können alle Benutzer, die Zugriff auf das Repository haben, verwenden. Um Datenbanken oder Ähnliches zu betreiben, stellt die ICC persistenten Speicher zur Verfügung. Das Cluster und andere Benutzer werden geschützt, indem normale Benutzer in der ICC keine Rechte besitzen, um Änderungen an Einstellungen vorzunehmen oder DaemonSets oder Service Accounts anzulegen oder Pods im „priviliged“ Modus zu starten. [22]

3.2 Messsysteme

3.2.1 ELK Stack

Der ELK Stack wird anhand der Vorgaben von Elastic betrieben. So werden die Applikationen, wie in Abbildung 3.1 dargestellt, verwendet, um Daten zu sammeln. Nach dem Abgreifen der Daten mithilfe der Beats, werden sie entweder direkt in Elasticsearch eingespeist oder zuerst mithilfe von Logstash veredelt und danach zu Elasticsearch geschickt. Die Daten können daraufhin mithilfe von Kibana oder Grafana ausgewertet werden. Beide Lösungen greifen hierfür auf Elasticsearch zu.

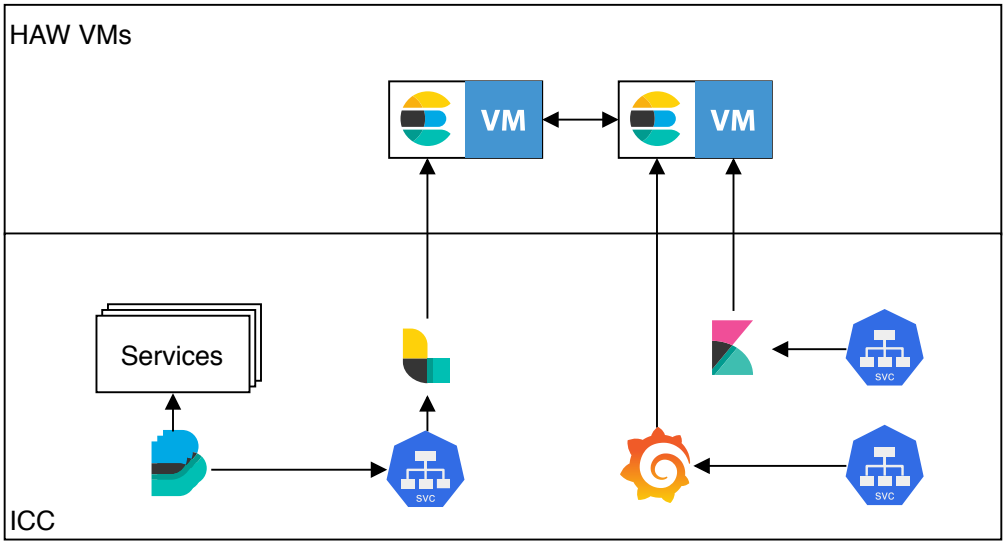


Abbildung 3.1: Verknüpfungen der verwendeten ELK Stack Applikationen in der ICC

Von Elastic wird vorgeschlagen, Beats mithilfe eines DeamonSets in Kubernetes zu betreiben, sodass auf jedem Kubernetes-Knoten ein entsprechender Beat vorhanden ist. Dieses Vorgehen konnte aufgrund der Rechteverteilung der ICC nicht gewählt werden, da die Rechte normaler Nutzer begrenzt sind. So können beispielsweise DeamonSets nicht durch normale Nutzer erstellt werden.

Logstash, Kibana und Grafana sind eigenständige Deployments in der ICC. Hierbei werden Kibana und Grafana immer nur in jeweils einem Pod betrieben, da nur von einer Person auf diese Dienste zugegriffen werden kann. Für die vorliegende Arbeit war geplant, Logstash aufgrund der hohen Last die für einen Logstash-Pod entstehen kann, wenn viele Daten verarbeitet werden müssen skalierbar zu betreiben. Der Engpass kann sehr schnell

entstehen, da Logstash eine ressourcenhungrige Java Applikation ist. Der Plan musste jedoch aufgrund der nicht vorhandenen Rechte aufgegeben werden.

Um Elasticsearch in Kubernetes betreiben zu können, muss für die Konfiguration der Kubernetes-Knoten auf DeamonSets mit privilegierten Pods zurückgegriffen werden. Aus diesem Grund war es nicht möglich Elasticsearch in einer sinnvollen Konfiguration in der ICC auszuführen. So werden stattdessen zwei Elasticsearch-Knoten auf zwei Virtuellen Maschienen des AI Labores betrieben. Hierbei kann nicht auf die komfortablen Features von Kubernetes zurückgegriffen werden, man kann jedoch auf diesem Weg ein Cluster betreiben. Alle in der ICC vorhandenen ELK Stack Komponenten mussten mit einer direkten IP-Adresse auf einem Elasticsearch Knoten bekannt gemacht werden.

3.2.2 Prometheus

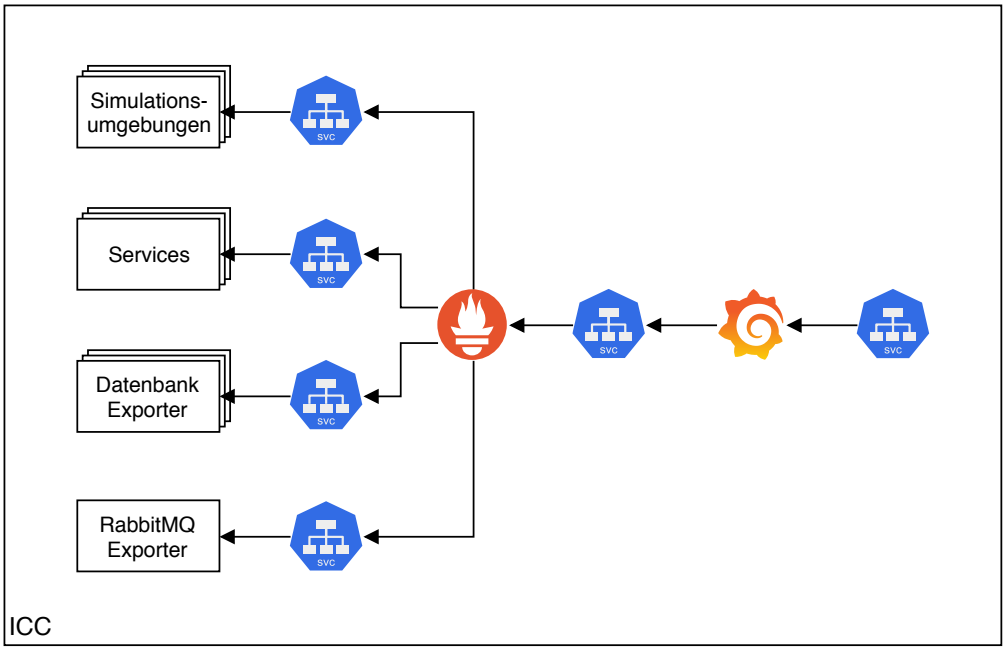


Abbildung 3.2: Darstellung wie mit Prometheus Daten erhoben werden können und wie auf Metriken in Prometheus zugegriffen werden kann

Um Prometheus zu verwenden, wird, wie in Unterabschnitt 2.4.2 vorgestellt, grundsätzlich nur der zentrale Prometheus Server benötigt. Weitere Komponenten, die neben dem zentralen Server in der ICC verwendet werden, um Applikationen zu monitoren, werden in Abbildung 3.2 dargestellt. Für den Betrieb und die Konfiguration der Scrape-Ziele wurde

der Kubernetes Prometheus-Operator verwendet. Dies ist ein Plugin für Kubernetes über das, mithilfe einer normalen Kubernetes-Konfiguration, ein Prometheus Server erstellt werden kann. Der Server wird dabei zentral verwaltet, ohne dass der Nutzer sich um Images, Speicher oder die Integration der Konfiguration in den Prometheus Pod kümmern muss. Die Scrape-Ziele können, wie der Server selbst, über eine Kubernetes-Konfiguration erstellt und verwaltet werden. Um die Definition der Ziele zu vereinfachen, werden gleichartige Ziele über Kubernetes-Services gebündelt. So muss nur ein Scrape-Ziel pro Kubernetes-Service definiert werden.

Prometheus bietet zwar die Möglichkeit, Metriken auf der Webseite des Prometheus Servers darzustellen. Eine Darstellung von Dashboards ist dabei jedoch nicht möglich. Für das Erstellen von Dashboards zur Überwachung der Simulation wird deshalb Grafana verwendet. Grafana ist als Deployment in Kubernetes vorhanden und so konfiguriert, dass es auf Prometheus zugreifen kann.

3.3 Infrastruktur-Komponenten

Für die Implementierung des Prozesses werden unterschiedliche Infrastruktur-Komponenten benötigt. Die Einsatzweise aller Infrastruktur-Komponenten ist vereinheitlicht. Sie können ohne weitere Konfiguration für jede der Service-Implementierungen verwendet werden. Die genaue Konfiguration, wie beispielsweiße das Datenmodell einer Datenbank, muss dabei jedoch durch die Applikation selbst definiert werden. Dies hat den Vorteil, dass alle Komponenten die gleiche Grundkonfiguration haben und sehr einfach verwaltet werden können . Des Weiteren ist es so möglich, die Komponenten einheitlich an die Messsysteme anzubinden, um weitere Daten sammeln zu können.

Alle Komponenten werden innerhalb der ICC ausgeführt. Aufgrund der gleichen Konfiguration kann für alle Komponenten ein einheitlicher Pod-Aufbau erstellt werden. Im nachfolgenden werden die Komponenten und deren Konfiguration innerhalb der ICC beschrieben.

3.3.1 Datenbank

Als Datenbank wird MySQL verwendet, da nur diese vom Framework unterstützt wird. Hierfür wird sie als StatefulSet in der ICC gestartet. Es wird kein Deployment verwendet,

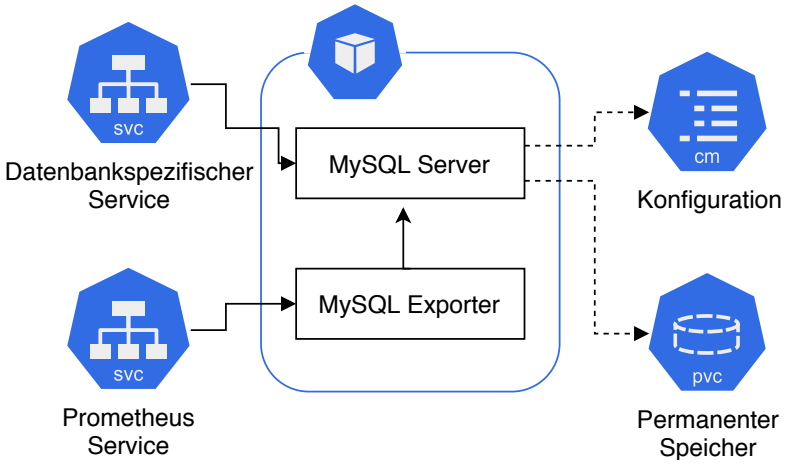


Abbildung 3.3: Repräsentation der MySQL Datenbank in der ICC

da mithilfe eines StatefulSets die Daten der Datenbank auch über den Lebenszyklus eines Pods hinaus bestehen und so im Normalfall keine Daten verloren gehen können. Über das StatefulSet ist nur ein MySQL Pod definiert, da eine einzelne Datenbank pro Service ausreicht und der Aufbau eines MySQL-Clusters in der ICC zu komplex wäre. Abbildung 3.3 zeigt einen MySQL Pod. Im Pod sind ein MySQL Server und ein MySQL Exporter vorhanden. Der Exporter verbindet sich über das lokale Netzwerk des Pods mit dem MySQL Server und kann so die Metriken des Servers auslesen und über eine API für Prometheus bereitstellen.

3.3.2 Message Queue

Als Message Queue unterstützt das Framework nur RabbitMQ. Für die Untersuchung wird RabbitMQ als einzelner Server innerhalb eines StatefulSets in der ICC betrieben, da so über einen Pod-Lebenszyklus alle Daten gespeichert werden. Damit gehen die bereits enthaltenen Nachrichten im Falle eines Absturzes nicht verloren und können nach einem Neustart des RabbitMQ-Containers ohne manuelle oder programmatische Eingriffe wieder hergestellt werden. Der Pod, in dem der RabbitMQ Server gestartet wird, enthält zwei Container. In einem Container ist der eigentliche RabbitMQ Server lokalisiert, in dem zweiten der Exporter. Der Exporter wird dazu verwendet, Metriken für Prometheus zugänglich zu machen.

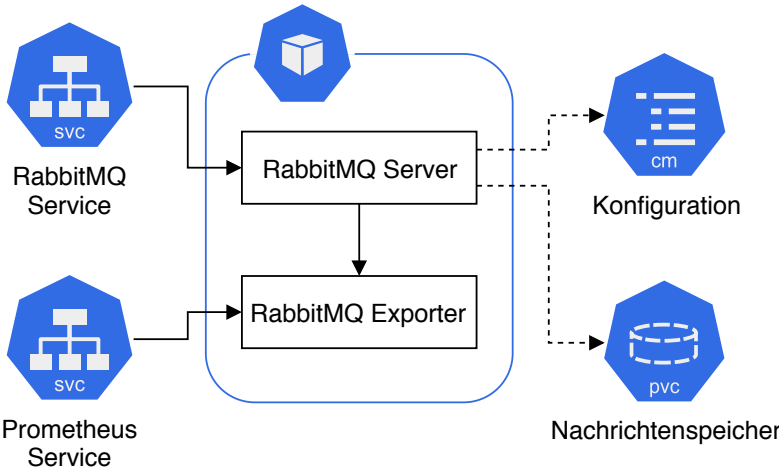


Abbildung 3.4: Repräsentation eines RabbitMQ-Servers in der ICC

3.4 Simulations-Komponenten

Auch die in der Simulation verwendeten Komponenten werden in der ICC betrieben. So können dynamisch Ressourcen für die Simulation allokiert und nach Ende der Simulation wieder freigegeben werden. Darüber hinaus können so alle Komponenten der Simulation mit geringem Aufwand skaliert werden. Wenn z. B. mehr Agenten simuliert werden müssen, können der Simulationsumgebung entweder mehr Ressourcen zugeteilt oder zusätzliche Simulationsumgebungen gestartet werden. Die verwendeten Komponenten lassen sich in zwei Bereiche gliedern. Ein Bereich steht für die Services des nachgestellten Geschäftsprozesses. Diese Services entsprechen den in der Realität eingesetzten Servern und Applikationen und können auf die in Abschnitt 3.3 vorgestellten Infrastruktur-Komponenten zugreifen. Der zweite Bereich steht für die Simulationsumgebung und ihre Komponenten.

Abbildung 3.5 gibt einen Überblick über das Zusammenspiel der Simulations-Komponenten mit der Applikation und den Messsystemen. Es sind die zwei Wege der Datenerfassung durch die Messsysteme erkennbar. Der Zugriff aus den Messsystemen erfolgt durch Prometheus während über den ELK Stack Daten in die Messsysteme gesendet werden. Der genaue konzeptionelle Aufbau der Simulations-Komponenten in der ICC wird im Folgenden beschrieben. Die Implementierung eines Szenarios wird in Kapitel 4 aufgezeigt.

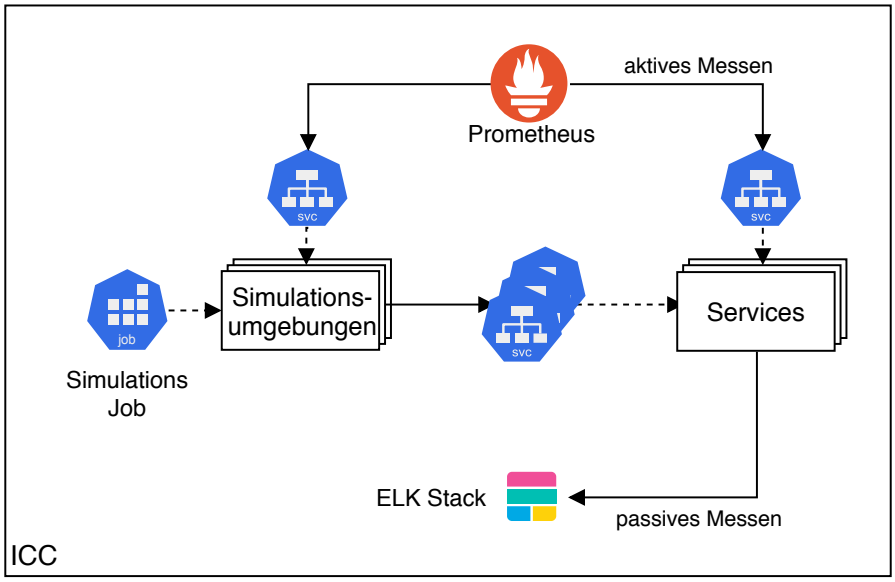


Abbildung 3.5: Zusammenspiel der Komponenten in der ICC

3.4.1 Prozess-Komponente

Für die Umsetzung der Prozesse wird das in Unterabschnitt 2.5.1 vorgestellte Application Framework verwendet. Mit diesem Framework werden, in Anlehnung an serverloses-Programmieren, nur Funktionen definiert. Diese können für die Ausführung in einem Service gruppiert werden. Für die Simulation wird jeder Service in einem Pod in der ICC ausgeführt. So wird für die Untersuchung eine Serviceoriented- beziehungsweise Microservice-Architektur umgesetzt, in der alle Services mit der gleichen Pod-Konfiguration betrieben werden. Abbildung 3.6 zeigt den standardisierten Service Pod.

Der Pod enthält die Konfigurationsdatei, über die der Service definiert wird. Über den Filebeat-Container werden die durch den Service geschriebenen Logs an den ELK Stack gesendet. Ein Exporter für Prometheus-Daten wird nicht benötigt, da das Framework so definiert wurde, dass es eine Prometheus-API bereitstellt. Die Pods werden mit Deployments erstellt, die nicht automatisch skalieren. So bleibt die Anzahl aller Pods über den Verlauf einer Simulation konstant. Es wurde festgelegt, dass jeder Service immer nur in einem Pod betrieben wird. Für eine Untersuchung mit skalierenden Deployments fehlen die benötigten Rechte in der ICC.

Für die Kommunikation zwischen den Services kann RabbitMQ verwendet werden. Hierfür müssen keine weiteren Änderungen an der ICC-Konfiguration vorgenommen werden, da

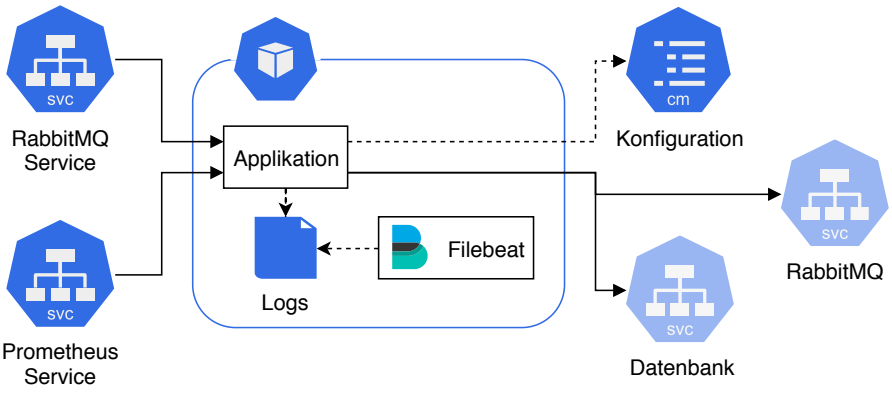


Abbildung 3.6: Repräsentation eines Services auf Basis des Application Frameworks in der ICC

nur auf den RabbitMQ-Service zugegriffen werden muss. Für die Kommunikation über REST muss in der ICC für jede Service-Gruppe ein eigener Service erstellt werden. So können andere Pods mithilfe des Service-Namens auf den oder die Pods der Service-Gruppe zugreifen. Die Endpunkte von RabbitMQ, anderen Service-Gruppen oder Datenbanken werden in der Service-Konfiguration hinterlegt.

3.4.2 Agenten

Die mit dem Agent Framework umgesetzten Modelle werden für die Untersuchung in der rudimentären Simulationsumgebung ausgeführt. Sollen sehr viele Agenten simuliert werden, können mehrere Umgebungen gleichzeitig gestartet werden. Um mit dieser Umgebung möglichst reproduzierbare Simulationen zu erhalten, wird zusätzlich zu den Simulationsumgebungen ein zentraler Server für die Verwaltung der Agentenkonfigurationen verwendet. Dieser ist auf Basis des Application Frameworks implementiert. Er bietet eine API, über das die Simulationsumgebungen Agentenkonfigurationen anfragen können. Bereitgestellte Konfigurationen werden in einer Datenbank gespeichert, um sie für spätere Simulationen wieder verwenden zu können.

Da die Simulation in der ICC ausgeführt wird, müssen die Simulationsumgebungen auch in einem Pod ausgeführt werden. Die hierfür verwendete Konfiguration ist in Abbildung 3.7 dargestellt. Im Pod ist nur die Simulationsumgebung als Container vorhanden. Es werden keine Exporter benötigt, da die Simulationsumgebung selbst ein Prometheus API bereitstellt. Es werden zwei Konfigurationsdateien benötigt. Eine definiert das Logging, über die zweite werden die Umgebung selbst, sowie die in dieser Umgebung verwendeten Modelle

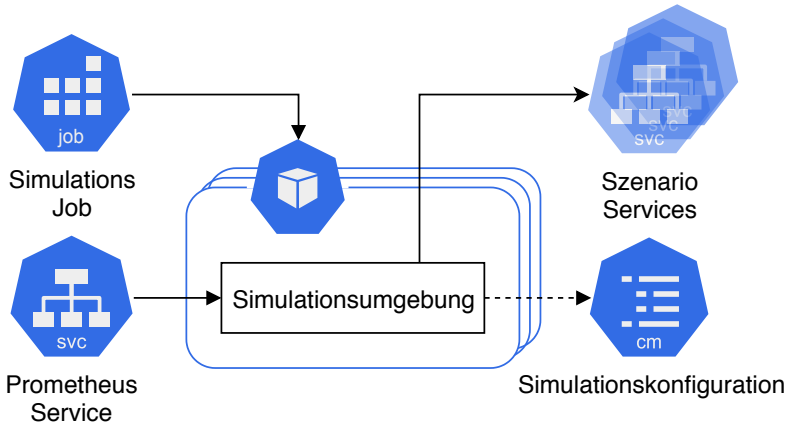


Abbildung 3.7: Repräsentation einer Simulationsumgebung in der ICC

konfiguriert. Die so definierten Pods werden für die Simulation über Kubernetes Jobs verwaltet. Es werden Jobs verwendet, da eine Simulation nur einmal definiert, gestartet und ausgeführt werden soll. Muss ein Pod aus irgendeinem Grund gestoppt werden, ist der Job so konfiguriert, dass kein neuer Pod gestartet wird. In der Untersuchung werden Simulationen, bei denen dies geschieht, nicht berücksichtigt. Der Server zur Verwaltung der Agentenkonfgurationen wird auf die gleiche Weise wie die Prozess-Komponenten in der ICC betrieben. Die Definition hierfür ist in Unterabschnitt 3.4.1 dargelegt.

4 Szenario

4.1 Prozess

In dieser Arbeit wurde der Geschäftsprozess einer beispielhaften Sekundärbank untersucht. Die Geschäftstätigkeiten der Bank umfassen die Bereiche Konsumentenkredite und Forderungsmanagement. Für die Verwaltung von Kreditkarten wird den Kunden ein Portal sowie eine Smartphone-App geboten. Mit der App können verschiedenste Dienste in Anspruch genommen werden. So kann beispielsweise eine aktuelle Kreditübersicht angezeigt oder eine Überweisung von der Kreditkarte auf das hinterlegte Konto angestoßen werden.

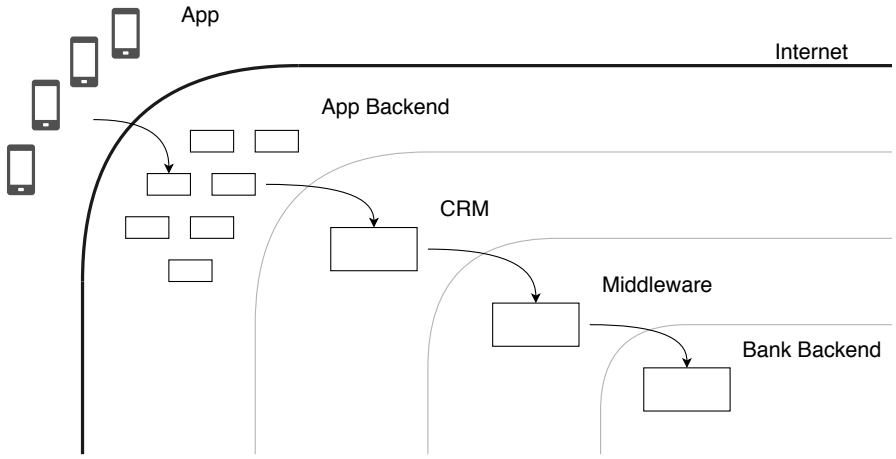


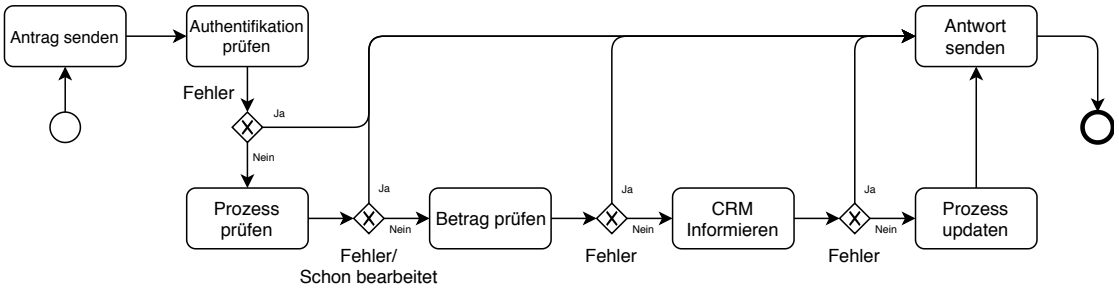
Abbildung 4.1: Beispielhafte Übersicht der abstrahierten IT-Architektur

Abbildung 4.1 zeigt die beispielhafte und abstrahierte IT-Architektur auf der die App der Bank basiert. Hierbei ist zu sehen, dass der App-Service über vier Schichten realisiert wurde. Die beiden unteren Schichten sind für das zentrale Bankgeschäft zuständig. Die dritte Ebene von unten enthält ein Customer-Relationship-Management (CRM). Dieses wird für die Verwaltung der Kunden verwendet. Auf der obersten Ebene befindet sich

das eigentliche Backend der App. Die auf dem Smartphone der Kunden installierte App kommuniziert mit den in der obersten Ebene laufenden Services.

Über den in dieser Arbeit verwendeten Geschäftsprozess können Kunden die Überweisung eines Kredites auf das hinterlegte Bankkonto beantragen. In der App kann der Kunde angeben welchen Betrag er benötigt. Der Antrag wird daraufhin an das App-Backend gesendet. Dieses speichert den Antrag und leitet ihn an die weiteren Systeme in den tieferen Schichten weiter. Als Antwort bekommt die App mitgeteilt, ob der Antrag für die Bearbeitung angenommen werden konnte. Nach einer erfolgreichen Antwort wartet die App eine gewisse Zeit, um danach den Status des Antrags anzufragen. Hierfür wird eine Anfrage mit einem Verweis auf den vorherigen Antrag an das Backend gesendet. Dieses antwortet mit dem aktuellen Antragsstatus. Liegt noch kein endgültigen Status vor, wird nach einer definierten Wartezeit erneut durch die App angefragt. Das wird so lange wiederholt, bis ein endgültiger Status zurückgegeben wird.

(a) Geschäftsprozessteil „a“ - Kreditantrag



(b) Geschäftsprozessteil „b“ - Statusanfrage

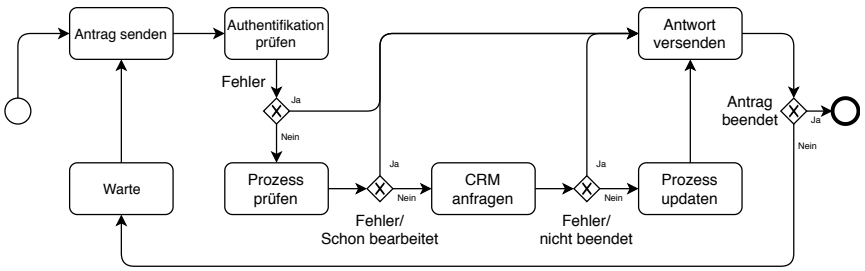


Abbildung 4.2: Der verwendete Geschäftsprozess

Abbildung 4.2 zeigt das Modell des verwendeten Geschäftsprozesses. Der Geschäftsprozess konnte in zwei Teile aufgeteilt werden. Geschäftsprozessteil „a“ entspricht dem Stellen des Kreditantrags an das Backend. Geschäftsprozessteil „b“ repräsentiert das Einholen des Status, bis ein endgültiger Status vorhanden ist. In beiden Teilen muss die Anfrage

vor einer weiteren Bearbeitung autorisiert werden. Ist dies in Teil „a“ geschehen, wird im Backend überprüft, ob es sich um einen neuen Antrag handelt. Ist das der Fall, wird in einem im Backend gespeicherten Konten-Snapshot des zentralen Banking-Systems geprüft, ob der Antrag genehmigt werden kann. War die Prüfung erfolgreich wird der Antrag an das CRM weitergeleitet, welches den Antrag speichert und weiterverarbeitet. Das CRM gibt eine Antrags-ID zurück, die im Backend gespeichert wird. Diese wird zusammen mit dem aktuellen Status zurück an die App gesendet. Nach der Authentifizierung in Teil „b“ wird im Backend geprüft, ob ein abschließender Status vorhanden ist. Ist dies der Fall wird der Status zurückgegeben. War die Prüfung nicht erfolgreich, wird anhand der gespeicherten Antrags-ID der aktuelle Status im CRM angefragt. Wurde er abschließend bearbeitet, wird der Status im Backend gespeichert und an die App zurückgegeben.

4.2 Applikationslandschaft

Für die Untersuchung wurde der in Abschnitt 4.1 beschriebene Geschäftsprozess einmal Event-driven und einmal über REST implementiert. Das Event-driven Szenario wird in dieser Arbeit „Szenario 1“ und das REST Szenario „Szenario 2“ genannt. Für die Implementierung der Services wurde das in Unterabschnitt 2.5.1 vorgestellte Framework verwendet, da mit ihm einfach neue Funktionalität implementiert und verwendet werden kann und durch Änderung der Konfiguration neue Metriken gemessen werden können.

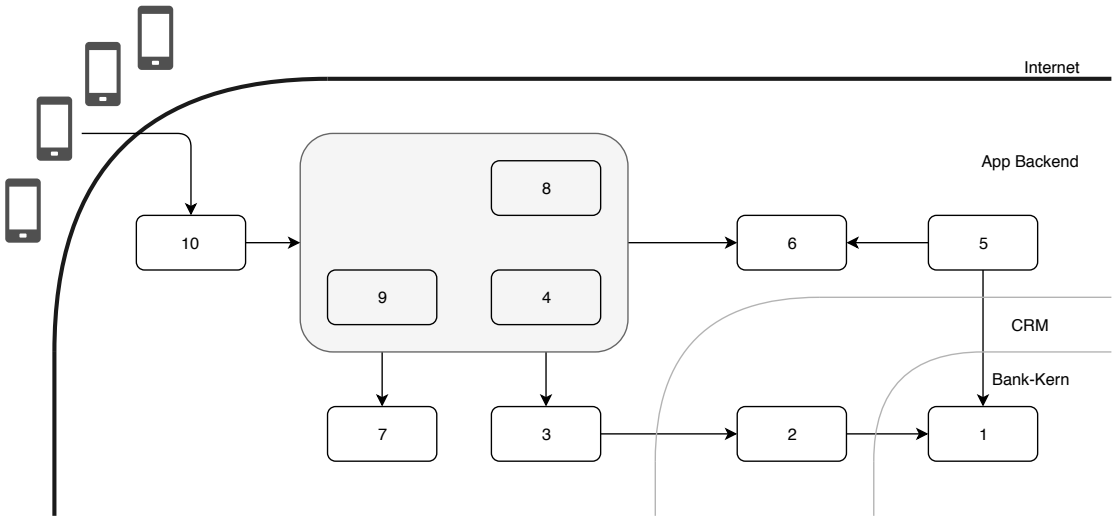


Abbildung 4.3: Übersicht der verwendeten Services. Die Definitionen zu den jeweiligen Nummern sind in Tabelle 4.1 aufgeführt.

Die in Abbildung 4.1 vorgestellte allgemeine IT-Architektur der Bank wurde für die Umsetzung des Geschäftsprozesses berücksichtigt. Abbildung 4.3 zeigt eine für das Szenario entworfene Version der IT-Architektur, wie sie in beiden Szenarien in jeweils leicht abgewandelter Form verwendet wurde. Hierbei wurde darauf geachtet, das alle wichtigen vorgestellten Applikationen der allgemeinen IT-Architektur über Services repräsentiert werden. Die wichtigsten Unterschiede der Szenarien treten bei den Services auf, die im grauen Kasten aufgeführt sind. Die Services in Abbildung 4.3 können anhand ihrer Nummer identifiziert werden. Tabelle Tabelle 4.1 enthält eine Beschreibung aller verwendeten Services.

Tabelle 4.1: Beschreibung der verwendeten Services

Service Nr. Name	**Beschreibung**
1 Banking Mock	Dieser Service repräsentiert die Kern-Komponenten der Bank aus Abbildung 4.1. Die Komponenten wurden zusammengefasst, da der Fokus der Untersuchung auf dem eigentlichen Backend der App liegt. Der Service verwaltet im System die eigentlichen Bank-Entitäten, wie beispielsweise die Konten.
2 CRM Mock	Dieser Service repräsentiert das CRM aus Abbildung 4.1. In ihm werden alle Kunden und deren Anfragen mit weiteren Parametern gespeichert. Dieser Service dient als Bindeglied zwischen App Backend und Service „1“.
3 CRM Client Service	Dieser Service ist eine Abstraktion des CRMs und bietet so einen zentralen CRM-Zugriffspunkt im App Backend. So ist das Backend vom CRM entkoppelt und es kann gut auf Änderungen des CRMs reagiert werden. Die Aufgabe des Services ist in erster Linie, das Weiterleiten von Anfragen aus dem Backend an das CRM.
4 Process Service	Jede an das Backend gesendete REST-Anfrage wird durch die App mit einem eindeutigen Identifier ausgestattet. Anhand des Identifiers erstellt dieser Service einen Systemprozess, der mit der weiteren Verarbeitung der Anfrage immer wieder aktualisiert wird. So muss bei einem Abbruch durch einen Applikationsfehler nicht mehr der komplette Prozess durchlaufen werden, sondern nur die noch fehlenden Schritte. Daneben kann die Verarbeitung von doppelten Anfragen erkannt werden.

Fortsetzung nächste Seite

Tabelle 4.1 – Fortsetzung vorherige Seite

5 Updater Service	Über diesen Service wird der Snapshot der Konten in Service „6“ regelmäßig aktualisiert.
6 Banking Service	Dieser Service enthält einen Snapshot aller Konten aus Service „1“. In der Realität ist dieser Snapshot stunden- oder tages-aktuell und bietet so den Services des Backends einen schnellen Zugriff auf die Kontendaten. Dies ist nötig, da die Bereitstellung durch die zentralen Bank-Applikationen in der Realität so lange dauern würden, dass die daraus resultierende Antwortzeit nicht mehr akzeptabel wäre. So kann im Backend eine Vorauswahl getroffen werden, ob ein Antrag mit großer Wahrscheinlichkeit im weiteren Verlauf angenommen wird oder nicht. Wird davon ausgegangen, dass der Antrag nicht angenommen wird, kann er schon an dieser Stelle abgelehnt werden. Hierbei kann es jedoch vorkommen, dass Anfragen aufgrund veralteter Daten in diesem Service abgelehnt werden.
7 Authentication Service	Um das Backend zu schützen werden, Anfragen über diesen Service authentifiziert.
8 Credit Request	Dieser Service wird nur in Szenario 2 verwendet. Er verwaltet den Ablauf von Kredit-Anfragen. Eine genauere Beschreibung wird in Abschnitt 4.2.4 gegeben.
9 Credit State Service	Dieser Service wird nur in Szenario 2 verwendet. Über ihn wird der Ablauf von Status-Anfragen gesteuert. Eine genauere Beschreibung wird in Abschnitt 4.2.4 gegeben.
10 Gateway Service	Dieser Service fungiert als Fassade des Backends gegenüber der App. Die Implementierung unterscheidet sich in beiden Szenarios, weshalb die Funktionsweise der Fassade in den folgenden Abschnitten noch genauer dargelegt wird.

Alle Services, die eine Datenbank benötigten, verwenden eine MySQL Datenbank, da das Framework nur diese unterstützt. Jeder Service verfügt dabei über eine exklusive Datenbank. Diese wurden in der ICC mit der in Unterabschnitt 3.3.1 vorgestellten Konfiguration betrieben. Außerdem wurden die Services, bedingt durch das Framework, mithilfe von Handlern und Jobs definiert. Diese wurden während der Untersuchung analysiert. Um die Handler immer eindeutig zuordnen zu können, wurden sie nach einem

bestimmten Schema nummeriert. Die Nummer besteht aus den drei Zahlen „S“, „T“ und „Z“ und wird durch die Aneinanderreihung der entsprechenden Zahlen gebildet.

S steht für den Service zu dem die Funktion gehört. Die Zahl hierfür ist die in Tabelle 4.1 verwendete Service-Nummer.

T repräsentiert den Typ der Funktion. In dieser Arbeit werden drei Typen verwendet.
Job: 0
HTTP-Handler: 1
MQ-Handler: 2

Z ist ein Zähler, um die Funktionen desselben Services und Typs zu unterscheiden. Diese Zahl entspricht der Position der Funktion, nachdem alle Funktionen eines Services mit dem gleichen Typ anhand ihres Names alphabetisch sortiert wurden.

Wird dieses Schema beispielsweise auf den einzigen Job des Services „5“ angewendet ergibt sich daraus der Identifier „501“. Das Schema kann in dieser Arbeit verwendet werden, da pro Service und Typ nie mehr als neun Funktionen vorhanden sind.

4.2.1 Unabhängige Handler

Als unabhängige Handler wurden die Handler definiert, die in der gleichen Form in beiden Szenarien eingesetzt wurden. Diese Handler basieren in beiden Szenarien auf REST. Abbildung 4.4 gibt eine Übersicht über die unabhängigen Handler. Tabelle 4.2 beschreibt die Zugehörigkeiten und die Aufgaben der Handler. In der Abbildung sind alle unabhängigen Handler in rot dargestellt. Die Vergabe der Handler-Nummern erfolgte anhand des beschriebenen Schemas zur Benennung von Funktionen.

In Abbildung 4.4 ist zu sehen, dass sich die unabhängigen REST-Handler auf die Kern-Services und den Fassaden-Service beschränken. REST-Handler werden in Abbildung 4.4 rot dargestellt. Die Kern-Services entsprechen in diesem Geschäftsprozess den legacy Systemen, die nicht ohne weiteres geändert werden können. Aus diesem Grund wurde angenommen, dass die verwendete Technologie unabhängig vom Backend der App ist. Die Technologie der Handler des Fassaden-Services wurde in beiden Systemen beibehalten, da sonst zwei verschiedene Simulationsumgebungen nötig gewesen wären. Die Implementierung der Fassaden-Handler änderte sich jedoch abhängig vom Szenario. In Tabelle 4.2 wird die Aufgabe der Fassaden-Handler beschrieben. Wie sie in den Szenarien implementiert wurden, wird in Unterabschnitt 4.2.3 und 4.2.4 dargelegt.

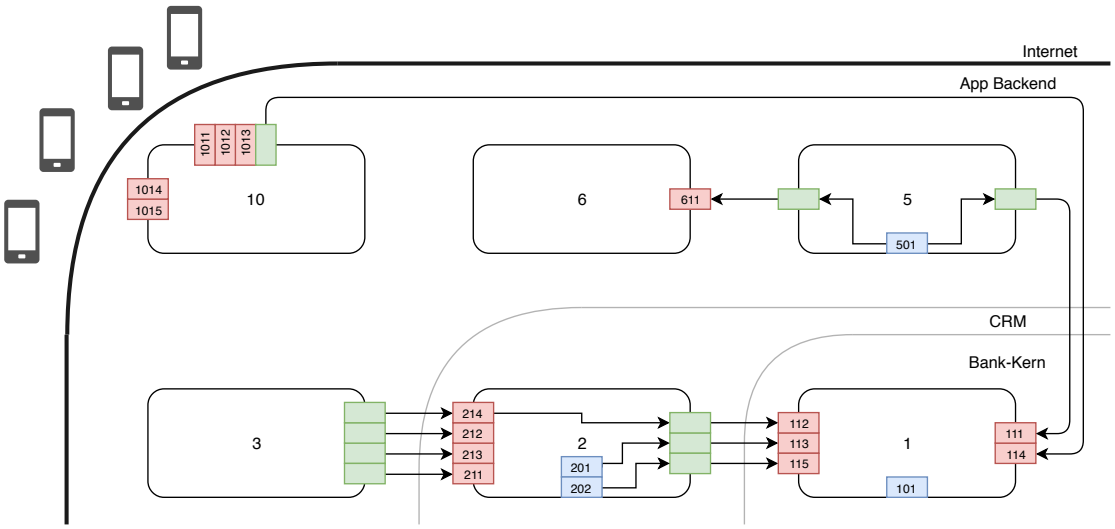


Abbildung 4.4: Übersicht der Zusammenhänge der unabhängigen Handler. In rot werden unabhängige REST-Handler und in blau Job-Funktionen dargestellt. Grün repräsentiert REST-Client Funktionen.

Tabelle 4.2: Beschreibung der unabhängigen REST-Handler

Funktion Nr. Name	**Definition**
111 AccountState	Über diesen Handler können Kontenstände abgefragt werden, die sich in einem in der Anfrage angegebenen Zeitraum geändert haben. Dies wird für die Erstellung des Snapshots in Service „6“ benötigt.
112 CreateAccount	Hierüber kann ein Benutzer mit Konto im Bank System angelegt werden.
113 Request	Dies ist eine Schnittstelle für Massendaten, über die eine Liste von Kreditanträge an das Banksystem übermittelt werden. Jeder Antrag wird hierbei durch das System geprüft und für die weitere Verarbeitung gespeichert. In der Antwort sind alle Prüfungsergebnisse enthalten, sodass der aufrufende Service informiert wird, welche Anträge fehlgeschlagen und welche Anträge weiterverarbeitet werden.
114 Pay	Dieser Handler bietet eine Schnittstelle, um einen Betrag auf ein Konto einzuzahlen und so Kredite zu begleichen.

Fortsetzung nächste Seite

Tabelle 4.2 – Fortsetzung vorherige Seite

115 RequestState	Diese Bulk Schnittstelle bietet die Möglichkeit den Status von Kreditanträgen anzufragen. Die durch den, Handler generierte Antwort enthält zu jedem angefragten Antrag eine Antwort.
211 CheckLogin	In diesem Handler werden von einem Nutzer gesendete Login-Daten mit den für den spezifizierten Account hinterlegten Login-Daten abgeglichen.
212 CreditRequest	Über diesen Handler wird ein Kreditantrag im CRM angelegt, um ihn im späteren Verlauf weiter bearbeiten zu können. Die Antwort des Handlers beschreibt, ob der Antrag gespeichert wurde. Ist dies der Fall, wird eine durch das CRM generierte Antrags-ID mit verschickt. Dieser Handler entspricht dem Prozess-Schritt „CRM Informieren“ aus Abbildung 4.2.
213 CreditState	Dieser Handler bietet die Möglichkeit, den Status eines Kreditantrags anhand der durch das CRM vergebenen Antrags-ID zu ermitteln. Das Ergebnis ist der aktuelle Stand des Antrags. Dieser Handler entspricht dem Prozess-Schritt „CRM Anfragen“ aus Abbildung 4.2.
214 Register	Mit diesem Handler wird das Erstellen eines Accounts im CRM und den nachgelagerten Systemen angestoßen.
611 UpdateAccounts	Über diese Schnittstelle kann der Snapshot im App Backend aktualisiert werden.
1011 Auth	Dieser Handler wird vom Benutzer angefragt, um sich im System zu authentifizieren.
1012 Pay	Über diesen Handler stellt ein Benutzer die Anfrage, sein Konto auszugleichen.
1013 Register	Dieser Handler muss von den Benutzern verwendet werden, um einen Account im System zu erstellen.
1014 CreditRequest	Ein Benutzer stellt eine Anfrage an diesen Handler, um einen Kreditantrag im System zu erstellen.
1015 CreditState	Will ein Benutzer den aktuellen Status seines Kreditantrags erfahren, muss dieser Handler angefragt werden.

4.2.2 Jobs

Für die Bearbeitung von wiederkehrenden Aufgaben wurden die Job-Funktionen des Application Frameworks verwendet. Ein Job wird dabei durch das Verstreichen einer

bestimmten Zeitspanne ausgelöst und ist nicht an einen externen Trigger gebunden. Tabelle 4.3 beschreibt die Zugehörigkeiten und die Aufgaben der in den Services eingesetzten Jobs. Die Einbettung der Jobs in die Services wird in Abbildung 4.4 gezeigt. Hierbei wurde jedem Job auf Basis der in Abschnitt 4.2 beschriebenen Vorlage eine Nummer zugeteilt. Zu jedem Job muss ein Zeitintervall angegeben werden, nach dessen Ablauf der Job ausgeführt werden soll. Diese Zeitintervall können durch die Konfiguration der Anwendungslandschaft schnell geändert werden. In dieser Arbeit wurde jedoch pro Job immer das gleiche Zeitintervall verwendet.

Tabelle 4.3: Beschreibung der Jobs

Funktions Nr. Name	**Beschreibung**	**Intervall**
101 Banking	In diesem Job werden alle neu in Service „1“ eingegangenen Anträge verarbeitet.	1 s
201 SendRequests	Mit diesem Job werden alle noch nicht weitergeleiteten Kreditanträge im CRM ausgelesen und über den Handler „113“ an Service „1“ gesendet. Ein Kreditantrag zählt für das CRM als noch nicht weitergeleitet, solange nicht bestätigt wurde, dass der Antrag in Service „1“ gespeichert wurde.	1 s
201 UpdateRequests	Über diesen Job werden die Status aller weitergeleiteten, aber noch nicht abgeschlossenen Kreditanträge über Handler „115“ bei Service „1“ angefragt. Ist in der Antwort ein abschließender Status vorhanden, gilt der entsprechende Antrag für Service „2“ als beendet.	1 s
501 BankingUpdate	Dies ist der einzige im App-Backend angesiedelte Job. Durch ihn werden regelmäßig alle neuen Accountänderungen über den Handler „111“ aus Service „1“ ausgelesen und mithilfe des Handlers „611“ in Service „6“ gespeichert.	1 s

4.2.3 Handler Szenario 1

Das für Szenario 1 implementierte App Backend basiert auf Events, die mithilfe von RabbitMQ als Eventbroker verteilt werden. Die sonstigen auf Unterabschnitt 4.2.1 basierenden

Services der Applikationslandschaft verwenden weiterhin REST. Abbildung 4.5 zeigt eine Übersicht des Backends und die in dieser Implementierung vorhandenen Eventflüsse. Dabei sind MQ-Handler in dunkelrot dargestellt und REST-Handler in rot. Darüber hinaus entspricht jede Pfeilfarbe einem Event-Key, sodass bildlich nachverfolgt werden kann, wie die Events zwischen den Services verschickt werden. Die farbigen Pfeile innerhalb eines Services wurden gewählt, um zu zeigen welches Event welche Client-Funktion auslöst. Tabelle 4.5 beschreibt die Zugehörigkeiten und die Aufgaben der MQ-Handler. Hierbei wurde jedem Handler auf Basis der in Abschnitt 4.2 beschriebenen Vorlage eine Nummer zugeteilt. Um den Eventfluss besser darzustellen, wurde in Tabelle 4.4 aufgezeigt, welcher Service welche Events empfängt und versendet.

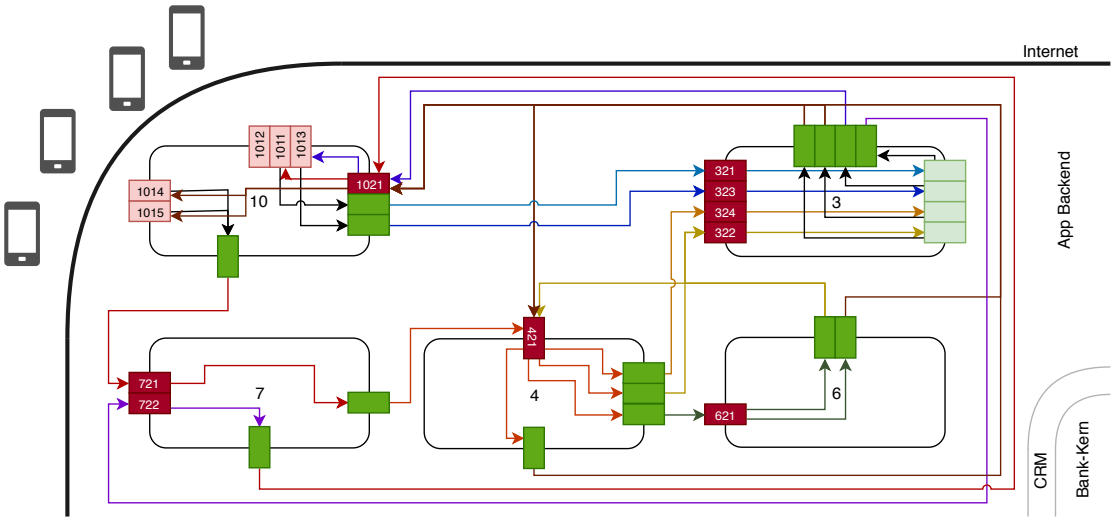


Abbildung 4.5: Überblick über Szenario 1. Gleiche Pfeilfarben repräsentieren Events mit den gleichen Event-Keys. In rot werden REST-Handler und in dunkelrot werden MQ-Handler dargestellt. Grün repräsentiert REST-Client Funktionen und dunkelgrün repräsentiert Event-Sender Funktionen.

Tabelle 4.4: Mapping der Event-Keys auf die sendenden und empfangenden Services

Event-Key	**Senden**					**Empfangen**				
	10	7	6	4	3	10	7	6	4	3
Authenticate	x	-	-	-	-	-	x	-	-	-
Authenticate.Failed	-	x	-	-	-	x	-	-	-	-
Authenticate.Successful	-	x	-	-	-	-	-	-	x	-
Authentication.CRM.Finished	-	-	-	-	x	-	x	-	-	-
Authentication.Finished	-	x	-	-	x	x	-	-	-	-
Authentication.Request	x	-	-	-	-	-	-	-	-	x
CreditRequest.BankingCheck	-	-	-	x	-	-	-	x	-	-
CreditRequest.BankingCheck.Successful	-	-	x	-	-	-	-	-	x	x
CreditRequest.Finished	-	-	x	x	x	x	-	-	x	-
CreditRequestState.Check	-	-	-	x	-	-	-	-	-	x
CreditRequestState.Finished	-	-	-	x	x	x	-	-	x	-
Registration	x	-	-	-	-	-	-	-	-	x
Registration.Finished	-	-	-	-	x	x	-	-	-	-

Der RabbitMQ-Server wurde, wie in Unterabschnitt 3.3.2 beschrieben, in der ICC betrieben. Alle Services konnten über einen Kubernetes-Service auf ihn zugreifen. In Abbildung 4.5 wurde er nicht aufgeführt, da nur der Nachrichtenfluss zwischen den Services des Backends dargestellt wurde. So muss in Abbildung 4.5 immer berücksichtigt werden, dass jedes Event über den Eventbroker vom Sender an den Empfänger weitergeleitet wird. Für die Implementierung des Backends wurde auf das Choreography-Pattern zurückgegriffen. Bei diesem Pattern wird der Geschäftsprozess durch das Zusammenspiel der Services ohne eine zentrale, kontrollierende Instanz realisiert. So wurden die Services „8“ und „9“ bei dieser Implementierung nicht benötigt.

Um den Geschäftsprozess abzubilden, musste der Fassaden-Service „10“ um den Handler „1021“ erweitert werden. So kann der Service Antwort-Events empfangen, um damit die synchronen REST-Anfragen zu beantworten. Aus diesem Grund wurden die Fassaden-Handler so umgestaltet, dass mit dem Empfang einer REST-Anfrage ein Event an das Backend gesendet wird und die Verarbeitung der REST-Anfrage so lange unterbrochen wird, bis ein Antwort-Event über Handler „1021“ empfangen wurde. Nachdem das entsprechende Event empfangen wurde, wird durch den korrespondierenden REST-Handler eine Antwort versendet.

Tabelle 4.5: Definitionen Szenario 1 Handler

Funktions Nr. Name	Definition
321 ECheckLogin	Dieser Handler leitet Loginanfragen an Service „2“ weiter.
322 ECreditRequest	Dieser Handler leitet einen Kreditantrag an Service „2“ weiter. Dies entspricht „CRM informieren“ im Geschäftsprozess.
323 ERegister	Dieser Handler leitet Registrierungsanfragen an Service „2“ weiter.
324 ECreditState	Dieser Handler leitet eine Statusanfrage an Service „2“ weiter. Dies entspricht „CRM anfragen“ im Geschäftsprozess.
421 EUpdateProcess	Über diesen Handler werden Systemprozesse verwaltet. Es können neue erstellt, schon bestehende aktualisiert oder der aktuelle Status angefragt werden. Bei einem Update können Prozess-Parameter und -Status bearbeitet werden. Im Event, das den Handler auslöst, kann ein Paremeter definiert werden, der bestimmt ob, ein Event auf Basis des Systemprozess-Status nach der Bearbeitung des Handlers versendet werden soll. Ist der Parameter nicht definiert, wird der Prozess nur aktualisiert. Dieser Handler entspricht im Prozess den Schritten „Prozess prüfen“ und „Prozess aktualisieren“.
621 ECheckCredit	Der Handler prüft, ob ein Kreditantrag auf Basis des Bank-Snapshots im App-Backend angenommen werden kann. Der korrespondierende Prozessschritt ist „Betrag prüfen“.
721 ECheckAuth	Dieser Handler prüft das Authorisierungstoken einer Anfrage. Bei erfolgreicher Prüfung wird ein Event mit der eigentlichen Anfrage versendet. Der Handler entspricht dem Geschäftsprozessschritt „Authentifizieren“.
722 ELoginUpdate	Über diesen Handler wird ein neues Authorisierungstoken für einen Benutzer gespeichert und in einem Event an die Fassade gesendet.
1021 EResponse	Dieser Handler verarbeitet alle für die Fassade bestimmten Events. In dem Handler wird das empfangende Event angenommen und an den wartenden REST-Handler weitergeleitet.

Im RabbitMQ-Server existiert für jeden Service ein Exchange, über den definiert wird, welche Nachrichten ein Service empfängt und jeweils ein Exchange, an den die Services

Nachrichten senden. Dieser Ansatz wurde gewählt, da so sichergestellt ist, dass die Konfigurationen der einzelnen Services entkoppelt voneinander sind. Damit konnte die interne Nachrichtenverteilung beliebig konfiguriert werden, ohne Änderungen an Queues und Services vornehmen zu müssen. Für das interne Routing wurde pro Event-Art jeweils ein Exchange verwendet. Jede Service-Art verfügte über eine Queue, die ihre Nachrichten von dem jeweiligen Empfänger-Exchange erhielt. So kamen bei dem Einsatz von mehreren Service-Instanzen alle Nachrichten aus dem gleichen Pool. Service „10“ ist eine Ausnahme davon. Bei diesem Service wurde eine Queue pro Service-Instanz erstellt, da dieser Service mit dem Request-Response Pattern arbeitet. So wurden die Nachrichten an die richtige Service-Instanz geleitet.

4.2.4 Handler Szenario 2

In Szenario 2 wurden nur REST-Handler für die Implementierung des Geschäftsprozesses verwendet. Abbildung 4.6 zeigt die exklusiv in diesem Szenario verwendeten REST-Handler. Diese sind alle in rot dargestellt. Zwischen den Services wurden nur einfache Pfeile verwendet, da eine Client-Funktion immer nur einen einzigen REST-Handler ansprechen kann. Dazu wird in der Abbildung eine Übersicht der Kommunikationswege von Anfragen durch das App Backend gezeigt. Für ein komplettes Bild muss Abbildung 4.4 aus Unterabschnitt 4.2.1 zusätzlich herangezogen werden. Tabelle 4.2 beschreibt die exklusiven Handler des Szenarios genauer. Hierbei wurde jedem Handler auf Basis der in Abschnitt 4.2 beschriebenen Vorlage eine Nummer zugeteilt.

Die Implementierung dieses Szenarios geschah auf Basis des Orchestration-Patterns. Dies ist der Grund, weshalb Service „8“ und „9“ als zentrale Koordinatoren verwendet wurden. Sie sprechen die einzelnen Backend-Services an, um den Geschäftsprozess zu erbringen. Service „8“ steht für den Geschäftsprozessteil „a“ und Service „9“ für den Geschäftsprozessteil „b“in Abbildung 4.2.

Die in Unterabschnitt 4.2.1 beschriebenen Fassaden-Handler wurden so implementiert, dass sie die Anfragen synchron an die korrespondierenden Handler weiterleiten. Dies ist in Abbildung 4.6 durch Verbindungen zwischen den Services mit Hinweisen auf die Handler dargestellt.

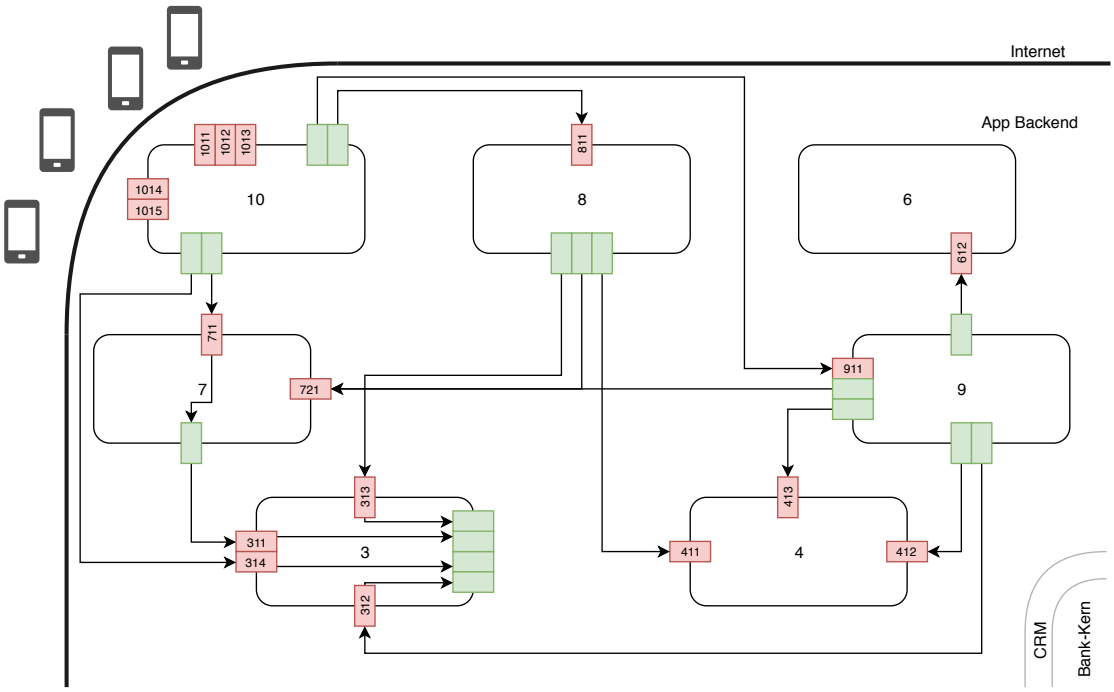


Abbildung 4.6: Überblick über Szenario 2. In rot werden REST-Handler und in blau werden Jobs dargestellt. Grün repräsentiert REST-Client Funktionen.

Tabelle 4.6: Definitionen der Handler aus Szenario 2

Funktions Nr. Name	**Definition**
311 CheckLogin	Dieser Handler leitet Loginanfragen an Service „2“ weiter.
312 CreditRequest	Dieser Handler leitet einen Kreditantrag an Service „2“ weiter. Dies entspricht „CRM informieren“ im Geschäftsprozess.
313 CreditState	Dieser Handler leitet eine Statusanfrage an Service „2“ weiter. Dies entspricht „CRM anfragen“ im Geschäftsprozess.
314 Register	Dieser Handler leitet Registrierungsanfragen an Service „2“ weiter.
411 GetProcess	Mit diesem Handler kann der aktuelle Status eines Systemprozesses abgefragt werden. Dies entspricht dem Schritt „Prozess prüfen“ aus Geschäftsprozessteil „b“.

Fortsetzung nächste Seite

Tabelle 4.6 – Fortsetzung vorherige Seite

412 SaveProcess	Über diesen Handler kann ein neuer Systemprozess gestartet werden. Bei einem erfolgreichen Start wird eine für den Systemprozess eindeutige ID zurückgegeben. Dieser Handler entspricht Schritt „Prozess prüfen“ in Geschäftsprozessteil „a“.
413 UpdateProcess	Dieser Handler bietet die Möglichkeit, einen durch eine ID definierten Systemprozess mit neuen Informationen zu versorgen. Dies kann ein neuer Status sein, es können aber auch weitere für den Systemprozess benötigte Informationen sein. Dies entspricht im Geschäftsprozess dem Schritt „Prozess aktualisieren“.
612 CheckCredit	Hierüber wird geprüft, ob ein Kunde laut den Daten im Zwischenspeicher des App-Backends einen Kredit bekommt oder nicht.
711 Auth	Dieser Handler koordiniert den Login eines Nutzers. Mit erfolgreichem Login wird ein Authorisierungstoken bereit gestellt.
712 CheckAuth	Der Handler prüft das Authorisierungstoken einer Anfrage.
811 RequestCredit	Dies ist der zentrale Handler, um in diesem Szenario einen Kreditantrag zu stellen. Er führt die für die Bearbeitung nötigen Schritte nacheinander aus. Für jeden Schritt werden die dafür benötigten Services angesprochen. Nach der Bearbeitung der Anfrage antwortet er mit einer aufbereiteten Antwort, die den aktuellen Status des Antrags enthält.
911 CreditState	Dies ist der zentrale Handler, um den Stand eines Kreditantrags zu prüfen. Der Handler orchestriert die für den Erhalt des Status nötigen Anfragen. Als Antwort sendet er den aktuellen Status des Antrags.

4.3 Simulations Modell

Die Untersuchung basierte auf MBT. Hierfür musste ein Modell für die Generierung von Anfragen an das System entwickelt werden. Das benötigte Modell wurde mit ABM definiert. Der Geschäftsprozess aus Abschnitt 4.1 diente dafür als Grundlage. Für die Implementierung des Modells wurde das in Unterabschnitt 2.5.2 vorgestellte Framework verwendet. So war sichergestellt, dass die Agenten in der ICC simuliert werden können und ohne Probleme auf die ebenfalls in der ICC ausgeführte Anwendungslandschaft zugreifen

können. Des Weiteren kann so die Anzahl der Agenten skaliert werden, indem mehrere Simulationsumgebungen in der ICC gestartet werden.

4.3.1 Definition

Der in Abschnitt 4.1 vorgestellte Geschäftsprozess wurde für die Definition des Modells als Basis verwendet. In diesem Prozess bedient ein Nutzer die App der Bank auf einem Smartphone, um die Prozessteile zu starten. Smartphones wurden im Modell nicht berücksichtigt, stattdessen griffen Agenten direkt auf das System zu. Für den Zugriff wurde REST verwendet. Ein Agent repräsentiert demnach einen Nutzer mitsamt seinem Smartphone. Weitere Agenten wie beispielsweise Mitarbeiter der Bank, die einen Antrag prüfen, wurden bei dieser Untersuchung nicht berücksichtigt und deshalb nicht umgesetzt. Die verwendeten Agenten sollten nur den Prozess ausführen, weshalb sie während der Simulation nicht mit anderen Agenten interagieren.

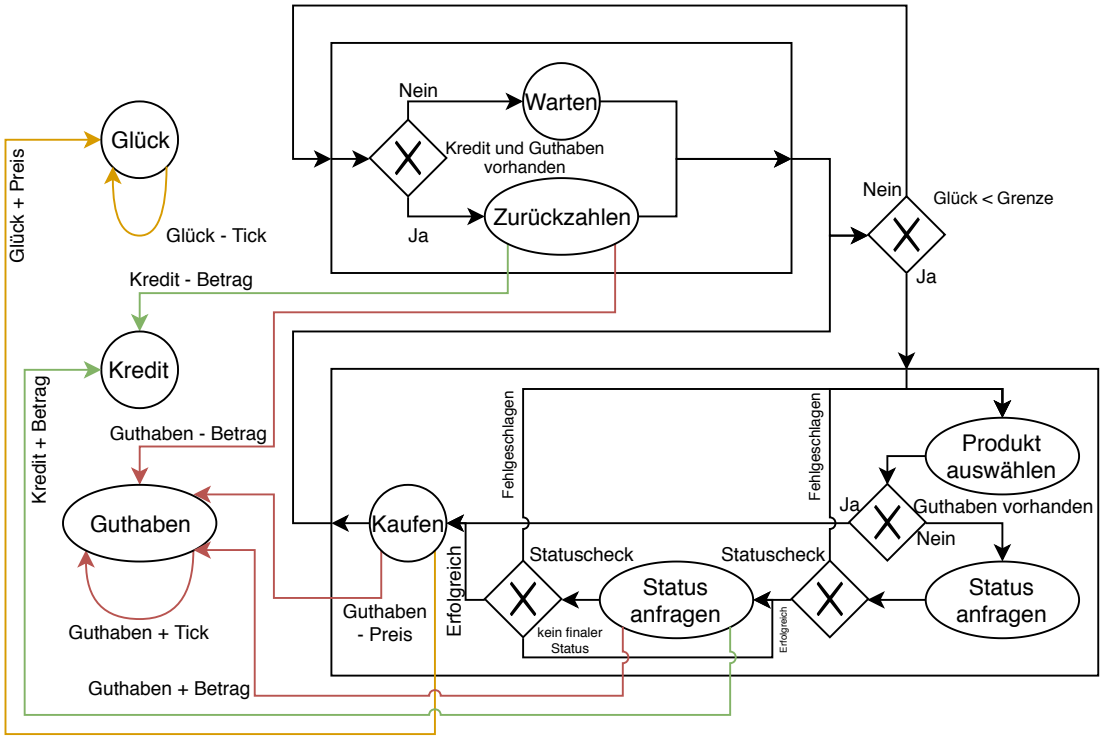


Abbildung 4.7: Abstrahiertes Agentendefinition. Änderungen an den Ressourcen eines Agenten werden in Farbe dargestellt.

Aus dem zugrundeliegenden Geschäftsprozess ergaben sich für die Agenten zwei Aktionen. Sie mussten erst eine Anfrage an das System stellen, um einen Kreditantrag zu erstellen und dann über eine zweite Anfrage den Status des Antrags ermitteln. Da es das Ziel dieser Arbeit ist, zu zeigen, wie solche Untersuchungen mit umfangreicheren Agenten durchgeführt werden können, verfügen die Agenten über ein rudimentäres Bedürfnis, über das gesteuert wird, wann ein Agent einen Kreditantrag stellt. Das Bedürfnis ist das Ziel „glücklich“ zu sein. Wie „glücklich“ ein Agent ist, wird durch eine Variable mit einem Wert von 0 bis 100 repräsentiert. Der Wert der Variable sinkt mit dem Verstreichen der Zeit. Ein Agent gilt bis zu einer bestimmten Grenze als „glücklich“. Fällt der Wert unter diese Grenze, versucht der Agent den Wert wieder über die Grenze zu heben. Dies ist möglich, indem ein Agent etwas kauft. Der Zuwachs an „Glück“ ist dabei direkt proportional zum ausgegebenen Betrag. Ein Agent verfügt über ein bestimmtes Guthaben, das für den Kauf von Dingen verwendet werden kann. Um ein regelmäßiges Einkommen zu simulieren, erhält jeder Agent während der Simulation regelmäßig neues Guthaben. Reicht das aktuelle Guthaben eines Agenten, dessen Wert unter die Grenze gefallen ist, nicht aus, um einen Kauf zu tätigen, muss der Agent einen Kredit aufnehmen. So kann er das benötigete Guthaben von der Bank erhalten. Hierfür stellt er einen Kreditantrag an das System. Eine abstrahierte Übersicht der Zusammenhänge eines Agenten in diesem Modell ist in Abbildung 4.7 dargestellt.

4.3.2 Umsetzung

Das Modell wurde mit dem vorgestellten Agent-Framework implementiert. Der implementierte Agent verfügt dabei über die beiden in der Definition beschriebenen Variablen und über einige Konfigurationsparameter. Dadurch ist es möglich, das gleiche Agentenmodell für mehrere Test-Szenarien zu verwenden. Für das automatische Generieren einer Agentenkonfiguration wurde ein Service in der ICC bereitgestellt. Wie der Service in der ICC betrieben wurde ist in Abschnitt 3.4 beschrieben.

Für die Implementierung des Agenten wurde eine neues Interaction-Layer über das Framework erstellt. In diesem wurden die im Modell benötigten Interaktionen implementiert. Daneben musste der vorhandene REST-Communication-Layer des Frameworks angepasst werden, da die bisherige Umsetzung Performance-Probleme bei der Simulation von mehr als 100 Agenten aufwies. Die Probleme entstand durch das massenhafte Versenden und Empfangen von REST-Anfragen. So stieg die Bearbeitungsdauer eines Ticks auf ein Vielfaches der normalen Bearbeitungsdauer an. Mit der optimierten Implementierung

konnte die Zahl von parallelen Anfragen, die ohne zu große Performance-Probleme verarbeitet werden konnten verfünffacht werden. Werden in der neuen Version mehr als die zulässige Anzahl an Anfragen durch die Agenten gestellt, werden sie in einer Queue zwischengespeichert und nacheinander abgearbeitet. Dadurch werden nie mehr als die vorgegebene Anzahl an Anfragen parallel bearbeitet.

Ziele

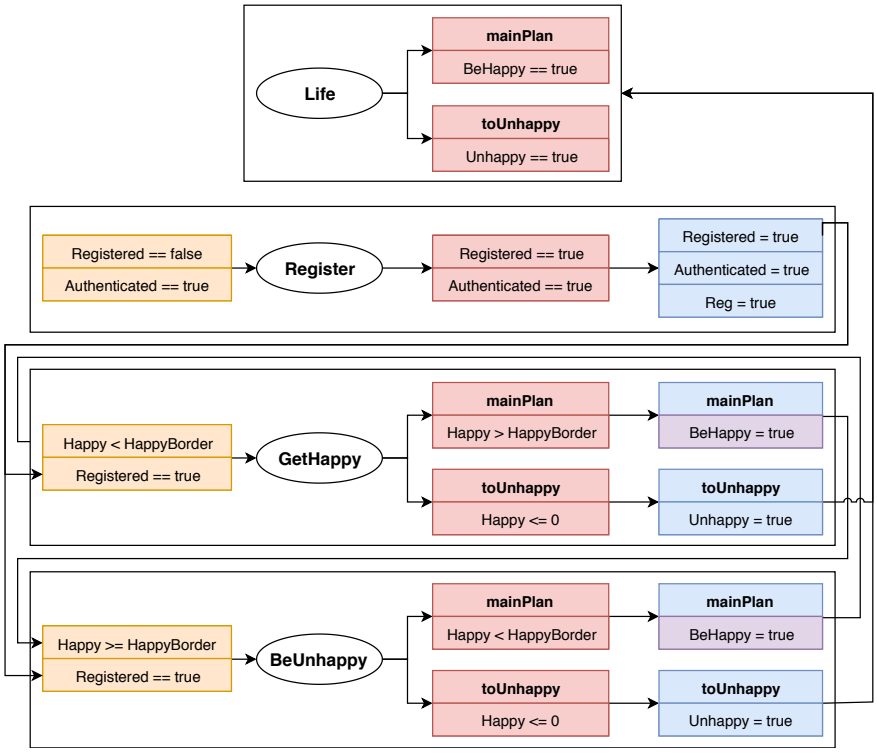


Abbildung 4.8: Die Ziele im Agentenmodell. Die Ellipsen beschreiben die Zielnamen. In rot werden Bedingungen für den Abschluss der Ziele dargestellt. Blau und lila stellen die Zustandsänderungen durch das Erreichen der Ziele dar und Orange die Zustände die vorhanden sein müssen, um ein Ziel verfolgen zu können. Lila bedeutet, dass die Änderung nur für die Planung berücksichtigt wird.

Abbildung 4.8 stellt die aus dem Modell abgeleiteten Agenten-Ziele dar. An oberster Stelle steht das „Lebens“-Ziel der Agenten. Dies ist der Zustand, den die Agenten versuchen zu erreichen. So ist das „Lebens“-Ziel eines Agenten „glücklich“ zu sein. Dies kann er über die Ziele „BeHappy“ und „GetHappy“ erreichen. Für eine fortlaufende Simulation, können

die Agenten das „Lebens“-Ziel nie erreichen. Dies ist sichergestellt, indem das Erreichen der Ziele „BeHappy“ und „GetHappy“ nur die Variable „BeHappy = true“ in Aussicht stellt. Nach dem Erreichen der beiden Ziele wird die Variable nicht geändert. Das erste Ziel wird durch einen Agenten verfolgt, wenn der „Glück“-Wert des Agenten über der beschriebenen Grenze liegt und das zweite, wenn er darunter ist. Um die Ziele verfolgen zu können müssen die Agenten zuvor das Ziel „Register“ erreicht haben. Dieses Ziel war technisch nötig, da die Daten eines Agenten für das Ausführen des Prozesses im System vorhanden sein mussten. Über das Ziel „Register“ wurde dies zum Start des Agenten sichergestellt.

Aktionen

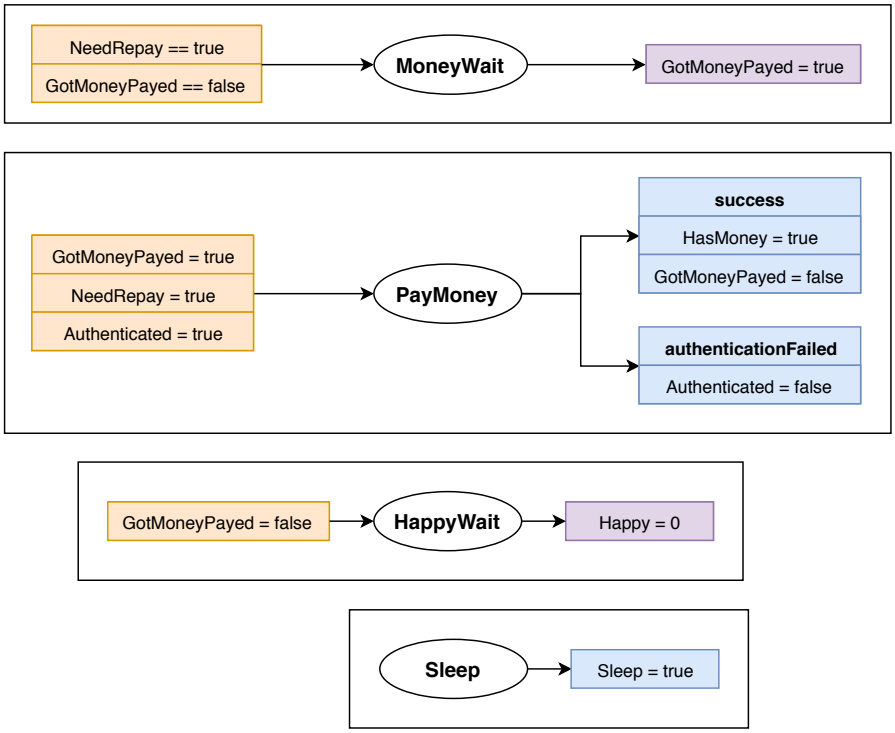


Abbildung 4.9: Darstellung eines Teils der Aktionen im Agentenmodell. Die Ellipsen beschreiben die Aktionsnamen. Blau und lila stellen die Zustandsänderungen durch das Erreichen der Aktion dar und Orange die Zustände die vorhanden sein müssen, um eine Aktion ausführen zu können. Lila bedeutet, dass die Änderung nur für die Planung berücksichtigt wird.

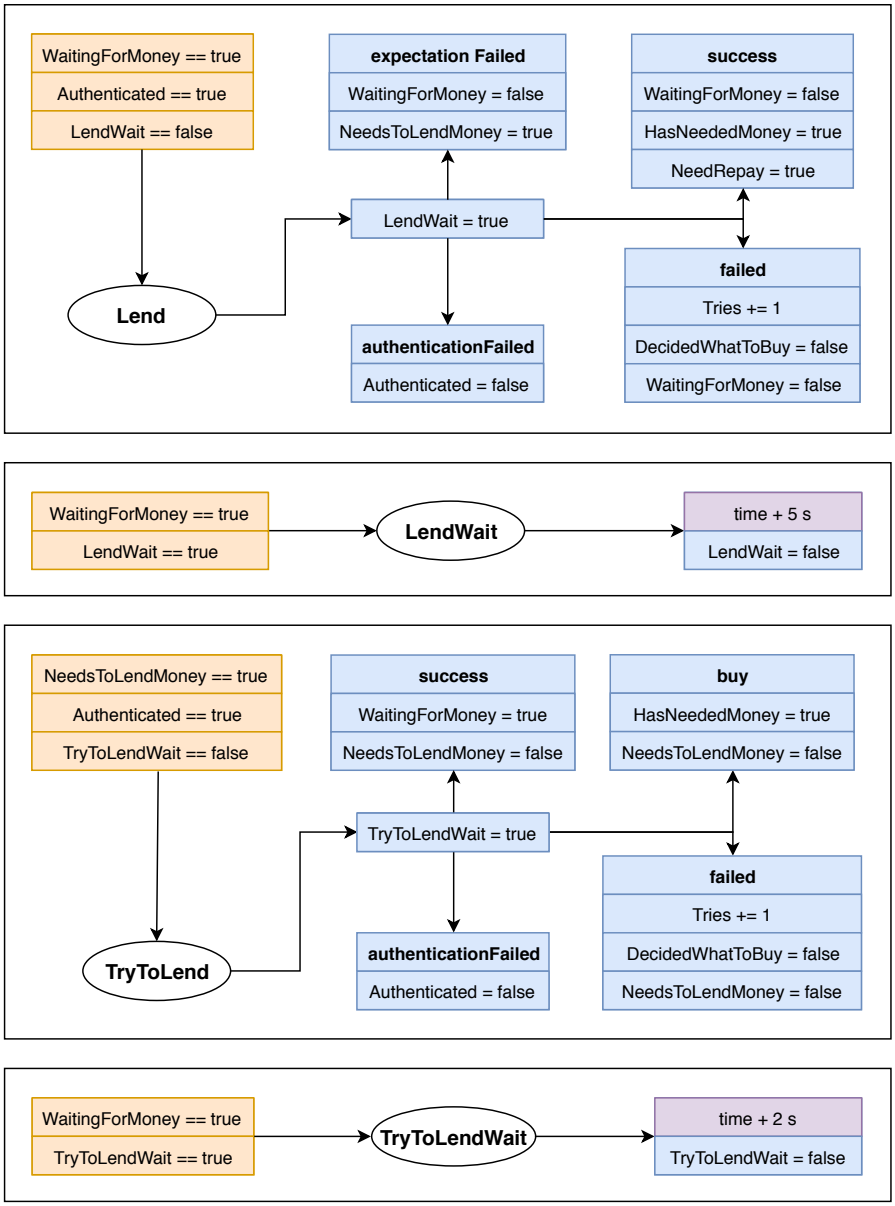


Abbildung 4.10: Darstellung eines Teils der Aktionen im Agentenmodell. Die Ellipsen beschreiben die Aktionsnamen. Blau und lila stellen die Zustandsänderungen durch das Erreichen der Aktion dar und Orange die Zustände die vorhanden sein müssen, um eine Aktion ausführen zu können. Lila bedeutet, dass die Änderung nur für die Planung berücksichtigt wird.

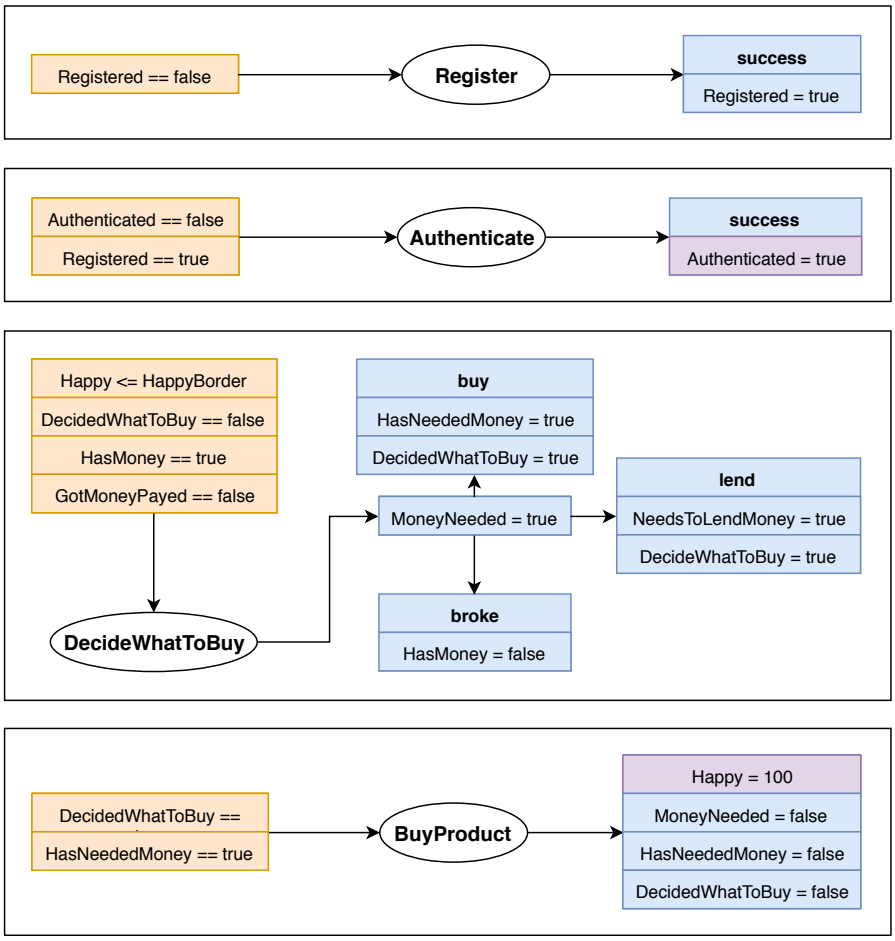


Abbildung 4.11: Darstellung eines Teils der Aktionen im Agentenmodell. Die Ellipsen beschreiben die Aktionsnamen. Blau und lila stellen die Zustandsänderungen durch das Erreichen der Aktion dar und Orange die Zustände die vorhanden sein müssen, um eine Aktion ausführen zu können. Lila bedeutet, dass die Änderung nur für die Planung berücksichtigt wird.

In Anlehnung an die Vorstellung der Ziele sind die implementierten Aktionen und deren Abhängigkeiten in Abbildung 4.11, 4.9 und 4.10 dargestellt. Aktionen die für ihre Erfüllung eine REST-Anfrage an das System stellen, sind in den Abbildungen rot hinterlegt. In den drei Abbildungen ist das beschriebene Modell abgebildet. Ist der „Glück“-Wert eines Agenten über der Grenze, ist er „glücklich“ genug und kann deshalb all sein vorhandenes Guthaben dazu verwenden, über die Aktion „RePay“ Kredite abzubezahlen. Dies ist nötig, da jeder Agent ein festgelegtes Kreditlimit einhalten muss. Ansonsten macht ein Agent nichts, wenn er „glücklich“ ist. Fällt der „Glück“-Wert eins Agenten unter die Grenze,

sucht er sich ein Produkt aus, das ihn wieder über die Grenze hebt. Benötigt er hierfür einen Kredit, stellt er einen entsprechenden Antrag über die Aktion „TryToLend“ und erfährt über die Aktion „Lend“, ob der Kredit genehmigt wurde. Besitzt der Agent genug Guthaben für das ausgewählte Produkt, kauft er es. Bekommt ein Agent den Kredit nicht, versucht er ein billigeres Produkt zu kaufen. Schlägt auch das fehl, wartet der Agent bis er wieder Guthaben erhält. Wenn der Agent vom System den Fehler erhält, dass er nicht autorisiert werden kann, versucht er sich mit der Aktion „Authenticated“ erneut zu autorisieren. Die REST-Anfragen von den Aktionen „Register“, „Authenticated“ und „RePay“ gehören nicht zum definierten Geschäftsprozess. Im System mussten jedoch Nutzer angelegt werden und es musste mitgeteilt werden, ob ein Nutzer seinen Kredit zurückgezahlt hat, weshalb diese Aktionen eingeführt wurden.

Konfiguration

Für den Einsatz des Agenten-Modells in verschiedenen Szenarien können Konfigurationsparameter geändert werden, um das Verhalten der Agenten anzupassen. So können beispielsweise sehr lang laufende Tests ausgeführt werden, in denen Agenten sich möglichst real verhalten. Es können aber auch Tests ausgeführt werden, die darauf abzielen, in möglichst kurzer Zeit möglichst viele Anfragen zu generieren. Hierbei leidet allerdings der Realitätsbezug der Simulation. Was in beiden Fällen vorhanden ist, sind logische und nachvollziehbare Zusammenhänge zwischen den von den Agenten ausgeführten Aktionen. In Tabelle 4.7 werden die in dieser Arbeit verwendeten Parameter vorgestellt.

Tabelle 4.7: Beschreibung der möglichen Konfigurationsparameter des Agentenmodells

Parameter	**Beschreibung**
HappyTickTime	Über diesen Parameter wird festgelegt in welchem Intervall der „Glück“-Wert angepasst wird.
HappyTickValueTime	Dieser Parameter gibt an, für welche Zeitspanne der Parameter „HappyTickValue“ definiert wird.
HappyTickValue	Mit diesem Parameter wird angegeben, um welchen Wert sich der „Glück“-Wert pro der im „HappyTickValueTime“ Parameter definierten Zeitspanne ändert.
MoneyTickTime	Dieser Parameter gibt an, in welchem Intervall die Agenten Guthaben durch die Simulationsumgebung erhalten.

Fortsetzung nächste Seite

Tabelle 4.7 – Fortsetzung vorherige Seite

MoneyTickValue	Der Parameter legt fest, welchen Guthaben-Betrag ein Agent pro Guthaben-Spanne erhält.
MoneyConversionRate	Mit diesem Parameter wird festgelegt wie viel Glück einem Agenten eine Geldeinheit wert ist.
MoneyStart	Mit diesem Parameter kann gesteuert werden, wie niedrig das Vermögen eines Agenten sein muss, um Guthaben durch die Simulationsumgebung erhält. Als Vermögen wird hierfür die Summe seiner Kredite von seinem Guthaben abgezogen.
MaxCredit	Dieser Parameter gibt die maximal möglichen Kreditsumme an, die ein Agent in Anspruch nehmen kann.
HappyBorder	Dieser Parameter entspricht der beschriebenen Grenze. Fällt der „Glück“-Wert unter den in diesem Parameter hinterlegten Wert gilt der Agent als „unglücklich“.
OverBorderMultiplier	Um die Anzahl an Anfragen besser steuern zu können, kann mit diesem Faktor der „HappyTickValue“ Wert über der Grenze angepasst werden.
UnderBorderMultiplier	Dieser Faktor passt den „HappyTickValue“ Wert unter der Grenze an.

4.4 Test-Szenarien

Die beiden in Abschnitt 4.2 vorgestellten Szenarien wurden jeweils mit zwei Test-Szenarien untersucht. Diese bauten auf den in Abschnitt 2.2 vorgestellten Performance-Test-Szenarien auf. Es wurde ein Stability-Test auf der Basis eines Volume-Tests und einen Stress-Test ausgewählt. Für die Generierung der entsprechenden Lastprofile wurde das vorgestellte Agentenmodell verwendet. Auf diesem Weg konnten beliebig viele unterschiedliche Lastprofile generiert werden. Dabei konnte durch die zentrale Agentenverwaltung sichergestellt werden, dass Lastprofile wiederholt verwendet werden können.

Jedes Test-Szenario wurde für beide Szenario mehrfach ausgeführt, um ein allgemeines Ergebnis zu erhalten. Damit die Tests dabei immer auf der gleichen Basis ausgeführt wurden, wurde das Testen mit dem in Abbildung 4.12 dargestellten Prozess automatisiert. So wurde vor dem Start eines Tests sichergestellt, dass zu dem Zeitpunkt kein anderer Test ausgeführt wurde. Danach wurden die Datenbank-StatfulSets in der ICC gelöscht und,

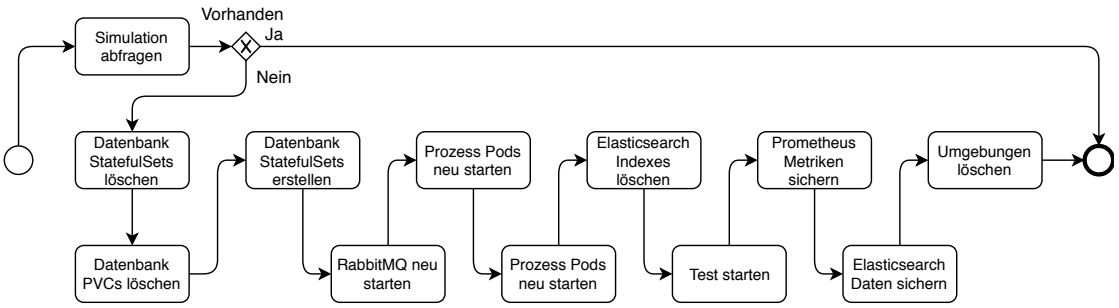


Abbildung 4.12: Der Prozess nach dem die Tests automatisiert ausgeführt werden

wenn nötig, der RabbitMQ Pod neu gestartet. Nachdem die Datenbanken gelöscht waren, wurden alle Persistent Volume Claims (PVCs) der Datenbanken entfernt. Daraufhin wurden die Datenbanken-StatefulSets wieder erstellt. So konnte sichergestellt werden, dass die Datenbanken beim Teststart immer im gleichen Zustand waren. War all dies geschehen, wurden die Geschäftsprozess-Pods neu gestartet und der Elasticsearch Index für die Logs entleert. Dies war nötig, da nicht genug Festplattenspeicher für Elasticsearch vorhanden war, um die Logs von mehreren Testläufen zu speichern. Nach diesen Schritten wurde der Test gestartet. Nach Ende des Tests wurden erst alle Prometheus-Metriken lokal gespeichert und danach alle Elasticsearch-Daten heruntergeladen. Zum Schluss wurde die Simulationsumgebung gelöscht.

Jedem Test wurde einer Lauf-ID zugeordnet. Eine Lauf-ID wurde pro Test-Szenario zwei Mal verwendet, da hierüber definiert wurde, welche Agentenkonfigurationen verwendet werden sollen. So konnte sichergestellt werden, dass für beide Prozess-Szenarien immer jeweils ein Lauf mit der genau gleichen Konfiguration ausgeführt wurde. Während eines Test-Laufs konnten Metriken in Prometheus und Elasticsearch genutzt werden, um den Verlauf zu überwachen. So konnte bei dem Auftreten eines Fehlers frühzeitig reagiert werden.

Die Last beider Test-Szenarien wurde auf Basis der gleichen ABM-Implementierung generiert. Um die Eigenheiten der Test-Szenarien auszuarbeiten, konnte das implementierte ABM konfiguriert werden. Für eine Variation in den einzelnen Tests wurden alle Agentenkonfigurationen zufällig auf Basis von Normalverteilungen erstellt. Dafür können Unter- und Obergrenzen, sowie die Grundwerte für jeden Parameter angegeben werden. Alle für die Szenarien verwendeten Werte sind in Tabelle 4.8 aufgeführt. Durch die für den Stability-Test gewählten Parameter war gegeben, dass die Agenten ihr Vermögen gleichmäßig ausgeben und eine gewisse Zeit während des Wartens auf neues Guthaben

keine Schulden haben. Der Stress-Test wurde so konfiguriert, dass die Agenten immer Schulden haben und sie, auch wenn sie Guthaben bekommen haben, sofort wieder einen Kreditantrag stellen müssen. Um das zu erreichen, wurde unter anderem der Parameter „MoneyStart“ verwendet. Im Stress-Test wurde so bestimmt, dass Agenten nur neues Guthaben erhalten, wenn ihr Vermögen kleiner als -1000 ist. So wurden sichergestellt, dass parallele Anfragen gestellt werden.

Tabelle 4.8: Aufführung der für die Konfigurationsparameter verwendeten Werte in beiden Test-Szenarien

Parameter	**Stability-Test**			**Stress-Test**		
	Min	**Median**	**Max**	**Min**	**Median**	**Max**
Duration		20 s			120 s	
Environments		6			10	
AgentsPerEnvironment		800			800	
CicleSpan		5 s			5 s	
ParallelRequests		250			250	
HappyTickTime		1 s			1 s	
HappyTickValueTime		1 s			1 s	
HappyTickValue	-1,5	-1	-0,5	-2,1	-1,4	-0,7
MoneyTickTime		300 s			300 s	
MoneyTickValue	450	3000	5400	450	1920	7700
MoneyConversionRate	0,1	0,2	1	0,1	0,2	1
MoneyStart		1000			-1000	
MaxCredit		3000			3000	
HappyBorder	40	60	90	40	60	90
OverBorderMultiplier		1		3,5	4	4,5
BorderMultiplier	0,4	0,5	0,6667	0,118	0,125	0,133

Die Konfiguration und weiteren Einstellungen der Pods und Services, mit denen sie für die Tests gestartet werden, sind in Tabelle 4.9 für Szenario 1 und in Tabelle 4.10 für Szenario 2 aufgeführt. Service „10“ wurde in beiden Szenarien mit mehr Speicher konfiguriert, da er den Status aller Anfragen halten muss während sie im Backend bearbeitet werden. Ansonsten verfügen in Szenario 2 alle Services über die gleichen Ressourcen. In Szenario 1 wurden die Ressourcen von Service „4“ angepasst, da dieser einen Teil der Aufgaben

von Handler „8“ und „9“ aus Szenario 2 übernahm. Für die Services in Szenario 1, die MQ-Handler verwendeten, wurde die Größe der jeweiligen Pools angegeben. Service „10“ benötigte sehr wenige Worker, da er empfangene Nachrichten sehr schnell verarbeiten konnte. Service „3“ und „4“ benötigten einen großen Pool, da durch sie die meisten Nachrichten verarbeitet werden.

Tabelle 4.9: Einstellungen der Service Pods von Szenario 1

Parameter	**10**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
CPU in mCPU	500	500	500	500	1000	500	500	500
Speicher in MiB	600	300	300	300	600	300	300	300
Anzahl Worker in MQ Pool	21	101	101	-	301	201	-	-

Tabelle 4.10: Einstellungen der Service Pods von Szenario 2

Parameter	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
CPU in mCPU	500	500	500	500	500	500	500	500	500	500
Speicher in MiB	600	300	300	300	300	300	300	300	300	300

4.5 KPIs

Für den Vergleich der beiden Prozess-Szenarien wurden KPIs verwendet. Um sie vergleichen zu können, mussten sie nach dem Erheben gespeichert werden. Für das Erheben wurden die vorgestellten Messsysteme verwendet. Um sie später zur Verfügung zu haben, wurden die KPIs lokal gespeichert. So konnten sie im weiteren Verlauf der Untersuchung ausgewertet werden. Ein Teil der KPIs wurde für die Untersuchung mit beiden Messsystemen erhoben, um einen Vergleich zwischen den verwendeten Systemen zu ermöglichen. Diese KPIs wurden über „Prometheus“ und den „ELK Stack“ erhoben. Obwohl ein Geschäftsprozess untersucht wird, wurden die verwendeten KPIs auf dem Applikations-Level erhoben und nicht auf dem Prozess-Level. Die folgenden Abschnitte enthalten die Definitionen der verwendeten KPIs. Sie basieren auf den drei Metrik-Arten, die von Jiang und Hassan beschrieben wurden [28]. Es wird auch dargestellt, wie sie berechnet und auf welchem Wege sie erhoben wurden.

4.5.1 Antwortzeit

Ein verwendeter KPI ist die Antwortzeit einer Funktion. Dafür wurde die Zeit vom Start bis zum Ende einer Funktion, unabhängig vom Ausgang der Funktion gemessen. In der Untersuchung wurde erwartet, dass pro Sekunde mehrere Hundert bis Tausend neue Werte entstehen. Um diese Werte-Menge verarbeiten und darstellen zu können, wurde für jede Funktion ein sekündlicher Wert berechnet, über den nachverfolgt werden konnte, wie sich die Antwortzeit im Verlauf eines Tests ändert. Die Berechnung der Metrik wurde mithilfe der beiden verwendeten Messsysteme durchgeführt. Obwohl beide Systeme auf sehr unterschiedlicher Basis arbeiteten, war die Art der Berechnung sehr ähnlich.

$$\begin{aligned} &histogram\_quantile(0.5, \\ &sum(rate(morph\_handler\_duration\_ms\_hist\_bucket[1m]))by(handler, le) \\ &) >= 0 \end{aligned} \tag{4.1}$$

Für die Berechnung des KPI mithilfe von Prometheus sammelten alle untersuchten Services die Antwortzeiten einer Funktion in einem Histogramm, in dem die Zeit in Millisekunden repräsentiert wurden. Dieses Histogramm konnte unabhängig vom Szenario verwendet werden, wodurch Vergleiche zwischen beiden Prozess-Szenarien vereinfacht wurden. Damit konnten außerdem im Nachhinein verschiedene Quantile berechnet werden. Nach dem Sammeln der Histogramm-Daten durch Prometheus, wurde aus ihnen mithilfe der Gleichung 4.1 der Verlauf für einen Test berechnet und für die weitere Verarbeitung gespeichert.

Für die Berechnung des KPI durch den ELK Stack, wurde in den Anwendungen geloggt wie lange es dauerte bis eine Funktion antwortete. Die so entstandenen Logs enthielten für die spätere Auswertung eine eindeutige Kennung. Sie verfügten außerdem über das Feld „duration“, in dem die Dauer gespeichert wurde und ein Feld „handler“, das den Namen des Handlers enthielt. Um aus diesen Logs Median-Werte zu berechnen, wurden sie erst mit einer „Filter“-Aggregation aus allen Logs gefiltert. Danach wurden die einzelnen Handler über eine „Term“-Aggregation getrennt und die so entstandenen Buckets über eine „Date Histogramm“-Aggregation in Buckets pro Sekunde geteilt. Aus diesen konnten mit einer „Percentiles“-Aggregation die Median-Werte abgeleitet werden. Der Aufbau der verwendeten Querys ist in Anhang 1 und 2 aufgeführt.

4.5.2 Durchsatz

Als Durchsatz-KPI wird im Allgemeinen die bearbeitete Menge an Anfragen einer Applikation in einem definierten Zeitraum verstanden [28]. In dieser Arbeit wurde der Durchsatz für einen Zeitraum von einer Sekunde betrachtet. Der KPI ist wichtig, da damit die Verarbeitungszeit relativiert werden kann. So könnten zwei Systeme, die die gleiche Verarbeitungszeit besitzen, eines dieser beiden Systeme aber doppelt so viele Anfragen erfolgreich bearbeiten kann, als unterschiedlich leistungsfähig eingestuft werden. Die Berechnung des KPI baute für beide Messsysteme auf den in Unterabschnitt 4.5.1 definierten Grundlagen auf. In beiden Fällen mussten keine neuen Daten erhoben werden.

$$rate(morph\_handler\_duration\_ms\_hist\_count[1m]) \tag{4.2}$$

In Prometheus wurden die Daten des Histogramms verwendet, mit dem auch die Antwortzeit berechnet wurden. Dies war Möglich, da jedes Histogramm über einen Zähler verfügt, der die Anzahl aller Einträge darstellt. So konnte diese Metrik über die Gleichung 4.2 von Prometheus berechnet werden.

Bei der Berechnung der Metrik über den ELK Stack wurde auf die schon bekannten Logs zurückgegriffen. Für die Berechnung des KPI wurden annähernd die gleichen Aggregationen verwendet. Statt dem Median der Antwortzeit wurde die Anzahl der im letzten Histogramm vorhandenen Einträge als Metrik verwendet. Der Aufbau der verwendeten Querys ist in Anhang 1 und 3 aufgeführt.

4.5.3 Ressourcenverbrauch

Diese KPIs repräsentierten die von einem Service benötigte CPU-Zeit und den verbrauchten Speicher. Der Weg auf dem die KPIs erhoben wurden, entspricht nicht dem optimalen Weg. Der optimale Weg für die Erhebung konnte nicht gewählt werden, da die dafür benötigten Rechte in der ICC nicht vorhanden waren.

$$rate(process\_cpu\_seconds\_total[1m]) \tag{4.3}$$

Die KPIs wurde über die von den meisten Prometheus-Frameworks oder -Exportern zur Verfügung gestellten default Metriken berechnet. So verfügten alle verwendeten Systeme über den Counter „process_cpu_seconds_total“ und den Gauge „process_resident_memory_bytes“. Mithilfe des Zählers und der Gleichung 4.3 konnte der sekündliche CPU-Verbrauch einer Anwendung ermittelt werden. Der Gauge entsprach dem von einer Anwendung benötigten Speicher, ohne dass weitere Berechnungen nötig waren. Die so erhaltenen Daten beschränkten sich auf die einzelne Services und nicht auf eventuell benötigte weitere Anwendungen in einem Pod und stellten auch nicht die von einem gesamten Pod benötigten Ressourcen in einem Kubernetes-Cluster dar.

4.5.4 Queue Länge

Um Szenario 1 besser untersuchen zu können, wurde für dieses Szenario auch die Länge der Queue als KPI erhoben. Dabei wurden die Queues betrachtet, die durch die Services verwendet wurden, um neue Nachrichten von dem RabbitMQ-Server zu empfangen. Die Länge beschreibt, wie viele Nachrichten in einer Queue warten, um von einem Service abgeholt und bearbeitet zu werden. So steigt die Länge einer Queue an, wenn die zu einer Queue gehörenden Services Nachrichten nicht schnell genug verarbeiten können. Für diesen KPI mussten neue Daten gesammelt werden, was nur mithilfe von Prometheus geschehen ist. Die Daten für den KPI wurden über den RabbitMQ-Exporter gesammelt. Dieser bot den Gauge „rabbitmq_queue_messages“, der genau den gewünschten Wert lieferte. So reichte es aus, den Gauge nach dem Test ohne weitere Berechnungen abzufragen und für die spätere Verarbeitung zu speichern.

5 Ergebnisse

Dieses Kapitel widmet sich den Ergebnissen aus dem mehrmaligen Ausführen der beiden Test-Szenarien. Der Sress Test wurde 45 Mal und der Stability-Test wurde 10 Mal für das Event-driven Szenario (Szenario 1) und REST Szenario (Szenario 2) ausgeführt. Die aus den Tests resultierenden Daten wurden anhand der KPIs gegenübergestellt. Hierfür wurde über den Verlauf der Tests der Median aus den gesammelten Metriken gebildet. Aus den so berechneten Verläufen wurden Diagramme erstellt anhand derer die Szenarien verglichen werden konnten. Bei KPIs, die auf Handler bezogen sind, wurden die Daten in einem Diagramm pro Service und Szenario zusammengefasst. Beziehen sie sich auf Services, wurde ein Diagramm pro Szenario erstellt. Sind in beiden Szenarien Daten für den Vergleich vorhanden, werden in einer Darstellung immer beide Szenarien übereinander gezeigt. Das obere Diagramm repräsentiert dabei immer Szenario 1 und das untere Szenario 2. Ist nur ein Diagramm vorhanden, wird an passender Stelle beschrieben, auf welches Szenario sich das Diagramm bezieht. Wenn vier Diagramme dargestellt sind, repräsentieren die beiden oberen Diagramme Szenario 1 und die beiden unteren Szenario 2, wobei jeweils das erste und dritte Diagramm mithilfe von Prometheus erstellt wurde, während das zweite und vierte mittels Elasticsearch angefertigt wurde.

5.1 Stress-Test

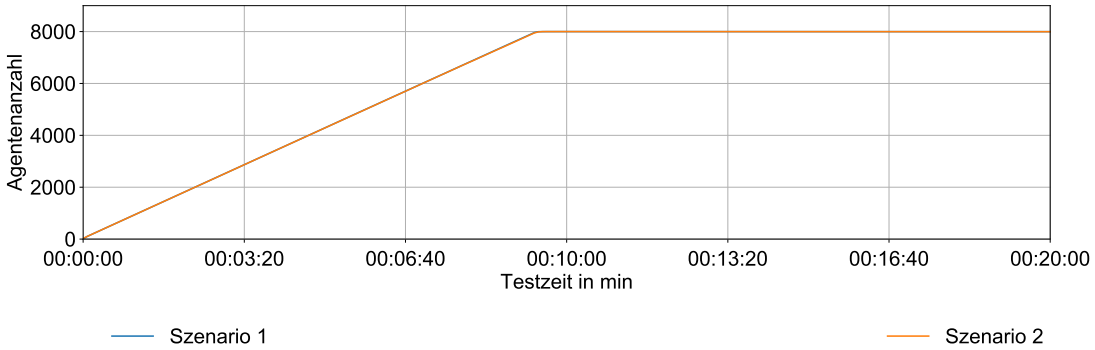
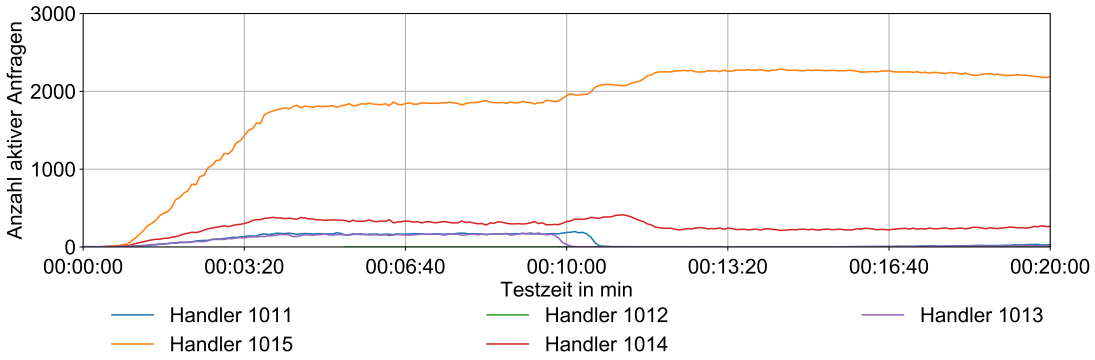


Abbildung 5.1: Agentenanzahl während der Stress-Tests

Für einen Vergleich der Entwicklung der Agentenpopulation während eines Tests wurde der Verlauf beider Szenarien in einem Diagramm dargestellt (vgl. Abbildung 5.1). Es zeigt sich, dass die maximale Agentenanzahl in beiden Szenarien nach annähernd 8,5 min erreicht wurde. Dieser Zeitraum wird als Initialisierungszeitraum bezeichnet. Danach blieb die Menge der simulierten Agenten konstant bis zum Ende des Tests bei 8000.

(a) Szenario 1



(b) Szenario 2

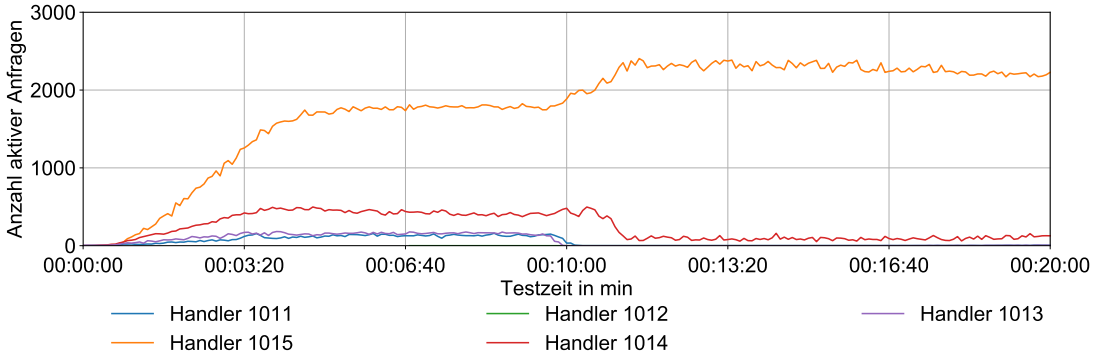
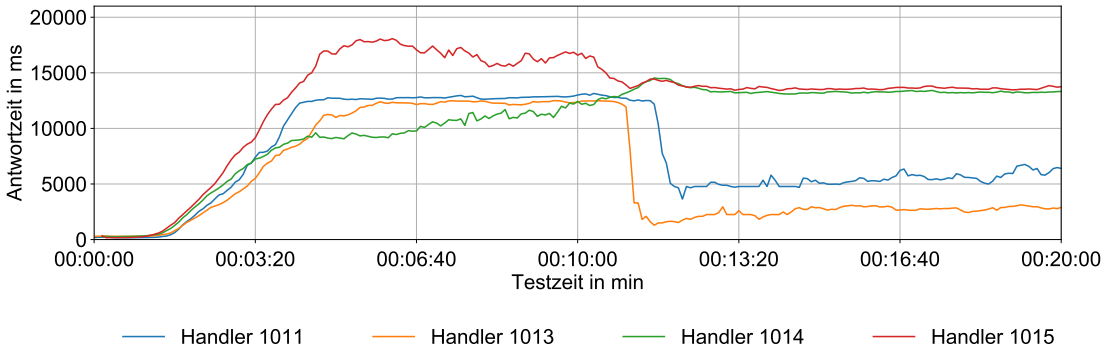


Abbildung 5.2: Aktive Anfragen während der Stress-Tests

Bei der Betrachtung der aktiven Anfragen (vgl. Abbildung 5.2) kann festgestellt werden, dass in beiden Szenarien annähernd die gleiche Menge an aktiven Anfragen ermittelt wurde. Einzig der Handler „1014“ weist Unterschiede auf. In Szenario 1 war er mit einer kleinen Ausnahme über den gesamten Test auf dem gleichen Level bei annähernd 350 Anfragen. In Szenario 2 wurden während des Initialisierungszeitraums 450 bis 500 Anfragen gemessen. Nach der Initialisierung fiel der Wert für den restlichen Testzeitraum auf 50 Anfragen.

5.1.1 Antwortzeit

(a) Szenario 1



(b) Szenario 2

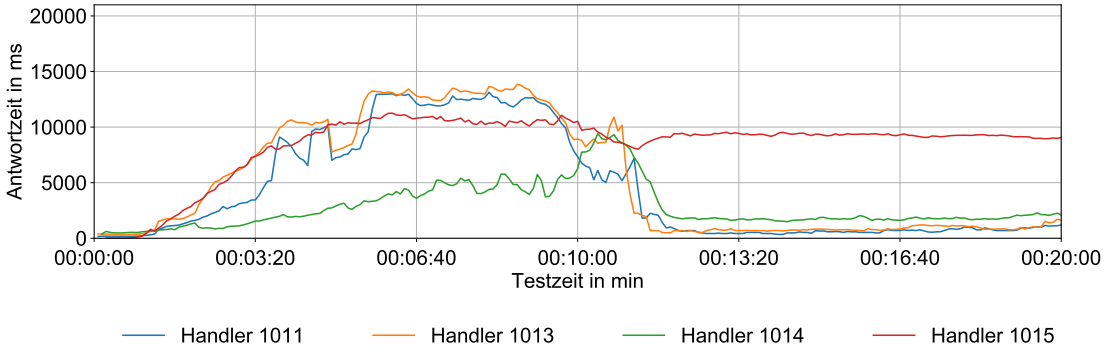
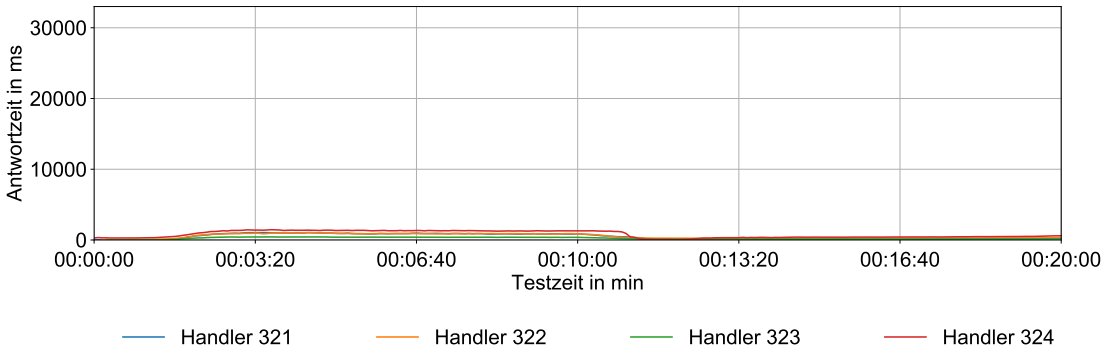


Abbildung 5.3: Antwortzeiten von Service „10“ (Prometheus)

Für den Vergleich der Antwortzeiten wurde Service „10“ herangezogen (vgl. Abbildung 5.3). Die Antwortzeiten beider Szenarien ähnelten sich in ihrem Verlauf. Anfangs stiegen sie an und blieben bis zum Initialisierungsende auf einem sehr hohen Niveau, meist sogar über 10 s. Dabei wurden in Szenario 1 deutlich höhere Antwortzeiten gemessen als in Szenario 2. Nach der Initialisierung blieb Handler “1015“ in beiden Szenarien auf diesem Niveau. In Szenario 2 fielen die restlichen Handler auf eine Zeit von unter 3 s ab. In Szenario 1 hielt Handler „1014“ das gleiche Niveau wie “1015“. Die Antwortzeiten der restlichen Handler fielen auch in diesem Szenario, jedoch nur auf 5 s und 3 s.

(a) Szenario 1



(b) Szenario 2

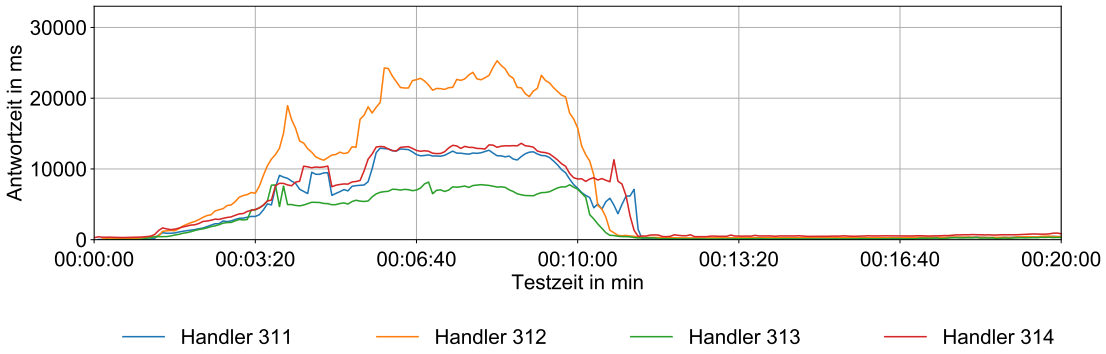
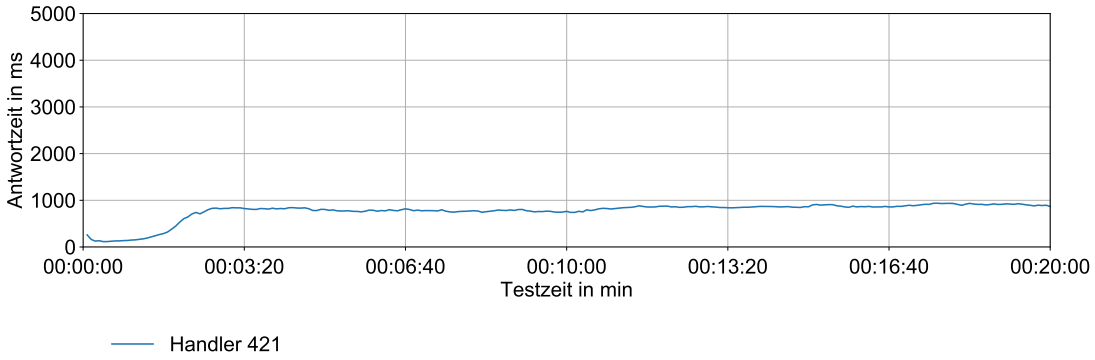


Abbildung 5.4: Antwortzeiten von Service „3“ (Prometheus)

Auch die Antwortzeiten von Service „3 “ wurden verglichen (vlg. Abbildung 5.4). Der Initialisierungszeitraum war in beiden Szenarien wieder erkennbar. In Szenario 1 stiegen die Antwortzeiten während dieser Zeit auf maximal 2 s. In Szenario 2 war der Initialisierungszeitraum sehr viel deutlicher sichtbar. So benötigte ein Handler über 20 s bis er antwortete und selbst der schnellste Handler benötigte immer noch 6 s bis 8 s. Nach der Initialisierung fielen die Antwortzeiten beider Szenarien auf unter 1 s.

(a) Szenario 1



(b) Szenario 2

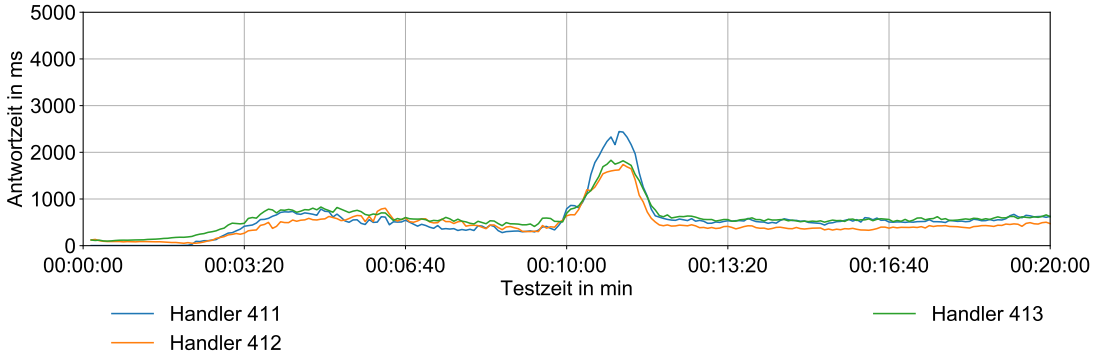


Abbildung 5.5: Antwortzeiten von Service „4“ (Prometheus)

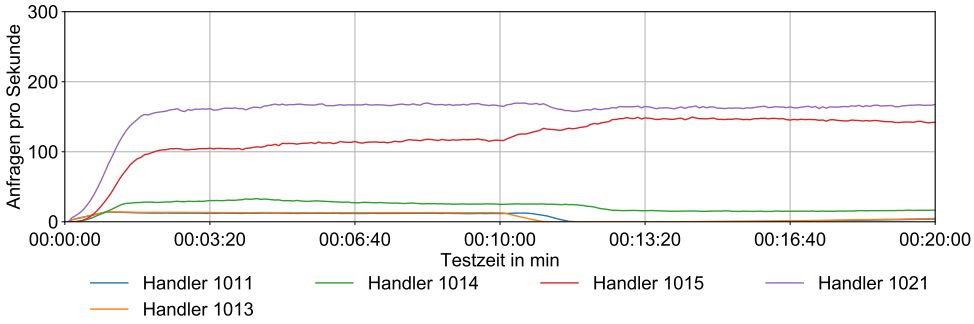
Im Vergleich der Antwortzeiten von Service „4“ konnte ein entscheidender Unterschiede beider Szenarien festgestellt werden (vgl. Abbildung 5.5). Szenario 1 besitzt für Service „2“ nur den Handler „421“ während Szenario 2 drei Handler für diesen Service verwendet. Die Antwortzeit von „421“ war mit 900 ms bis 1 s im Großteil der Tests länger als die Zeiten der drei Handler von Szenario 2, welche zwischen 500 ms und 900 ms lagen. Am Ende der Initialisierung wuchsen die Antwortzeiten aller Handler aus Szenario 2 kurzzeitig auf ca. 2 s an.

5.1.2 Durchsatz

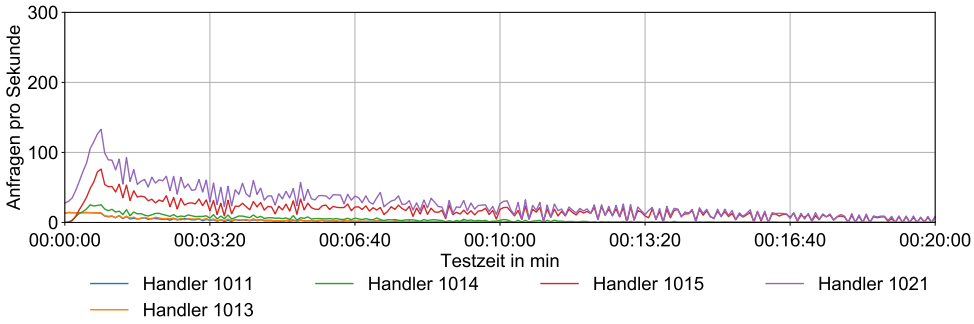
Der Durchsatz wurde anhand von Service „10“ verglichen (vgl. Abbildung 5.6). Er war in beiden Szenarien auf Basis von Prometheus in der Initialisierungsphase noch ähnlich. Danach zeigte sich jedoch, dass Handler „1015“ aus Szenario 2 annähernd 100 Anfragen

pro Sekunde mehr verarbeiten konnte. Darüber hinaus konnte auch Handler „1014“ in Szenario 2 erkennbar mehr Anfragen als dieser Handler aus Szenario 1 bearbeiten. In Szenario 1 war ein weiterer Handler „1021“ vorhanden. Dieser verarbeitete pro Sekunde immer die Summe aller Anfragen der anderen Handler dieses Szenarios. Beim Vergleich der beiden Szenario 1 Diagramme ist ein großer Unterschied feststellbar. Im Elasticsearch-Diagramm für Szenario 1 entstand anfangs eine Anfragespitze. Hiernach fielen die Werte langsam bis zum Ende des Tests auf null. Der Verlauf von Szenario 2 in Elasticsearch entsprach dem durch Prometheus ermittelten Verlauf.

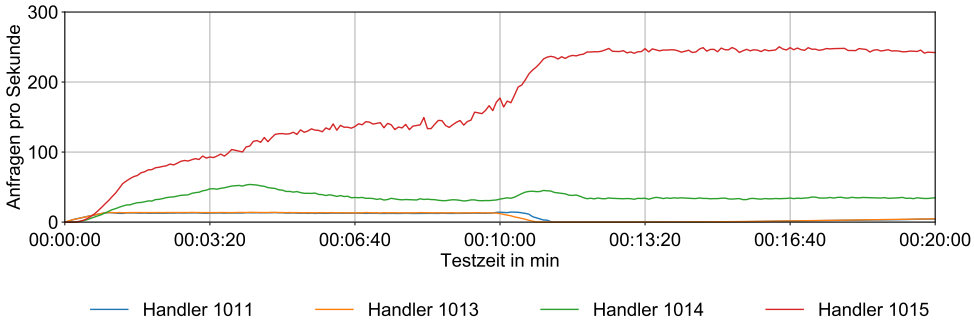
(a) Szenario 1 (Prometheus)



(b) Szenario 1 (Elasticsearch)



(c) Szenario 2 (Prometheus)



(d) Szenario 2 (Elasticsearch)

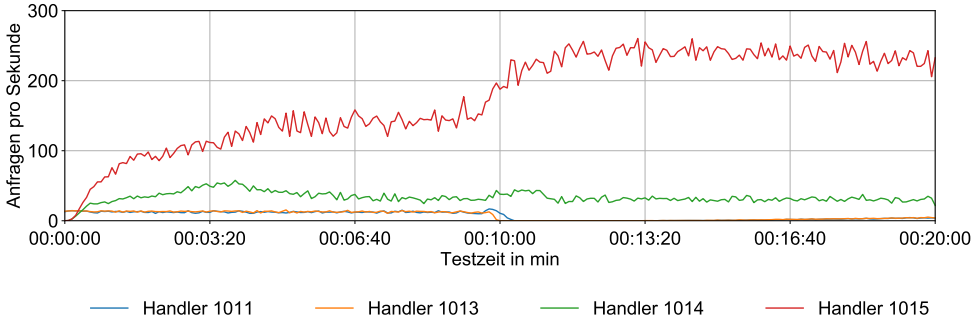
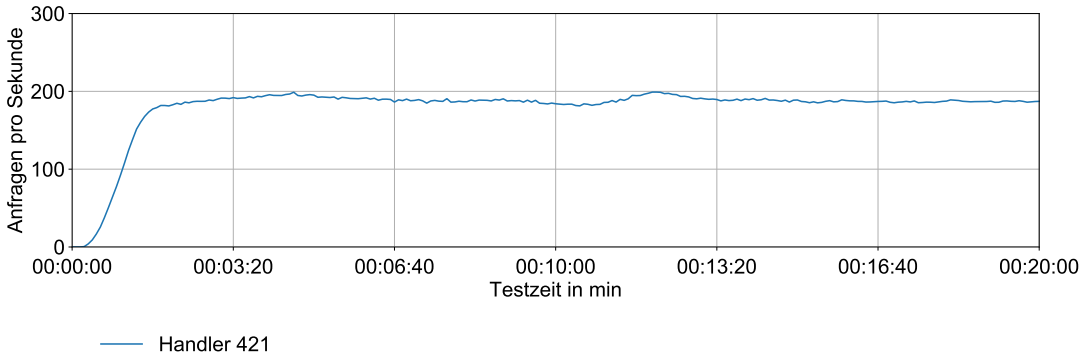


Abbildung 5.6: Durchsatz von Service „10“

(a) Szenario 1



(b) Szenario 2

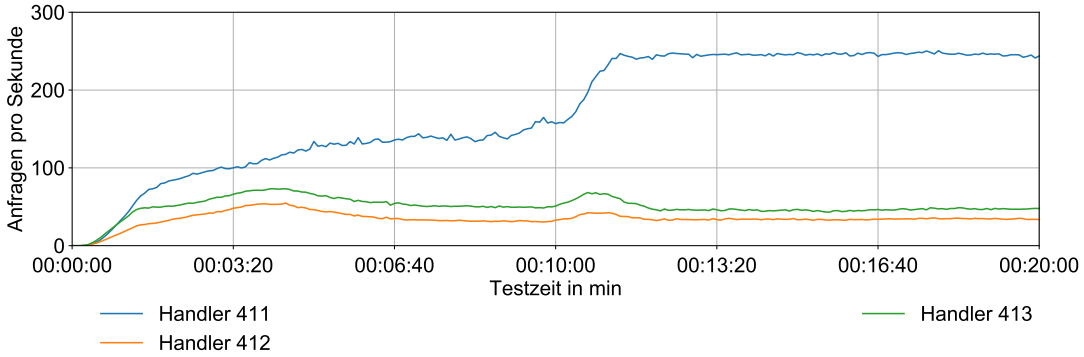
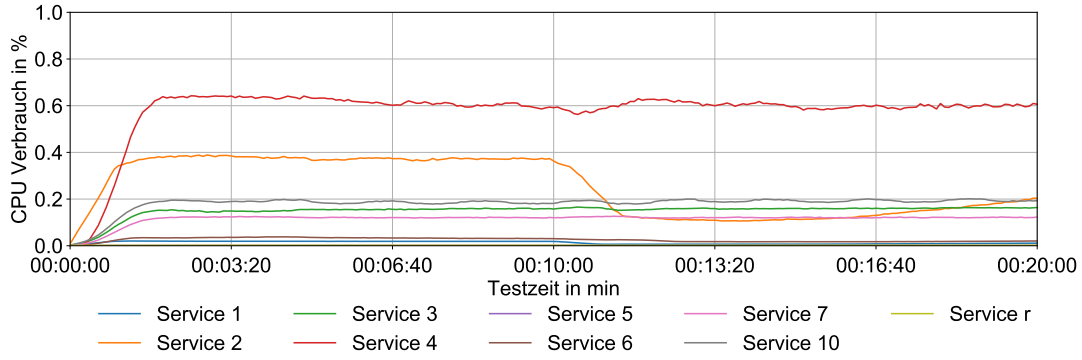


Abbildung 5.7: Durchsatz von Service „4“ (Prometheus)

Bei dem Vergleich des Services „4 “ in Bezug auf den Durchsatz (vgl. Abbildung 5.7) zeigte sich, dass in Szenario 2 mehr Anfragen bearbeitet werden konnten als in Szenario 1. In Szenario 1 konnte Handler „421“ konstant annähernd 200 Anfragen bearbeiten. Während der Initialisierung in Szenario 2 stieg die Summe der bearbeiteten Anfragen schnell auf 150 Anfragen pro Sekunde an. Danach stieg die Summe bis zum Ende der Initialisierung aufgrund von Handler „411“ leicht bis zu annähernd 250 Anfragen pro Sekunde an. Die beiden anderen Handler in Szenario 2 blieben nach dem anfänglichen Anstieg die restliche Simulation auf einem gleichen Level. Nach der Initialisierung stieg der Durchsatz von Handler „411“ noch einmal sprunghaft um 100 Anfragen pro Sekunde auf 250 Anfragen pro Sekunde an.

5.1.3 Kapazität

(a) Szenario 1



(b) Szenario 2

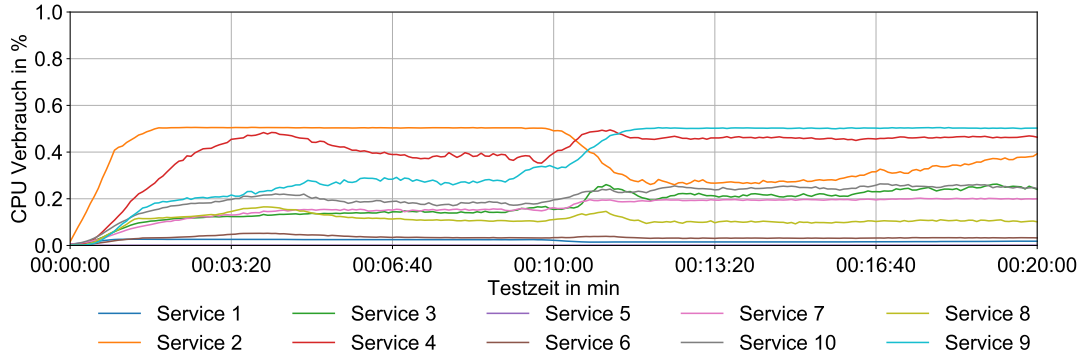
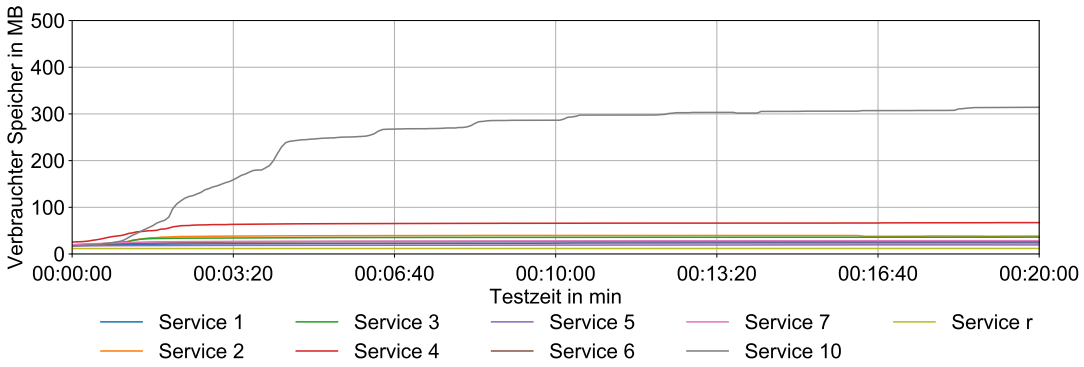


Abbildung 5.8: CPU Verbrauch der Services (Prometheus)

Um beide Szenarien bezüglich der Kapazität zu vergleichen, wurde der prozentuale CPU-Verbrauch herangezogen (vgl. Abbildung 5.8). Bemerkenswert ist der hohe Wert für Service „4“ aus Szenario 1. Der Median dieses Services lag während des Tests bei 0,6 %. Während der Initialisierungsphase verbrauchte der Service „2“ in Szenario 1 fast 0,4 %. Danach fiel der Wert auf unter 0,2 %. Auf diesem Wert befanden sich alle sonstigen in diesem Szenario verwendeten Services über den kompletten Testzeitraum. Auch in Szenario 2 benötigte Service „4“ viel CPU. Die Auslastung schwankte dabei zwischen 0,4 % und 0,5 %. Service „2“ verhielt sich ähnlich wie in Szenario 1, er verbrauchte jedoch im gesamten Test 0,1 % mehr als in Szenario 1. Service „9“ war nur in Szenario 2 vorhanden und verbrauchte während der Initialisierung 0,3 % und danach 0,2 %. Die sonstigen Services befanden sich während der Initialisierungsphase unter 0,2 %. Danach stiegen einige auf 0,2 % bis 0,3 % an.

(a) Szenario 1



(b) Szenario 2

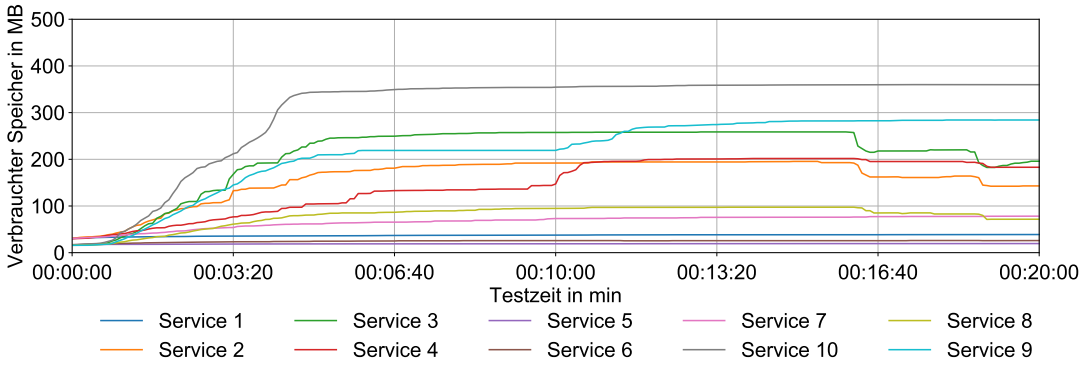


Abbildung 5.9: Speicherverbrauch der Services (Prometheus)

Neben der CPU-Auslastung wurde der durch die Services verbrauchte Speicher untersucht (vgl. Abbildung 5.9). In Szenario 1 benötigte nur Service „10“ mehr als 100 MB Speicher. Dieser wuchs während des kompletten Tests auf am Ende 300 MB verbrauchten Speicher an. Auch in Szenario 2 wurde durch Service „10“ der meiste Speicher verbraucht. Dabei wuchs der Speicher jedoch nur in den ersten Minuten des Tests und stagnierte danach bei 350 MB. Des Weiteren waren in Szenario 2 vier Services vorhanden, die zwischen 100 MB und 300 MB an Speicher benötigten. Die restlichen Services in diesem Szenario verbrauchten unter 100 MB.

5.1.4 Message Queue

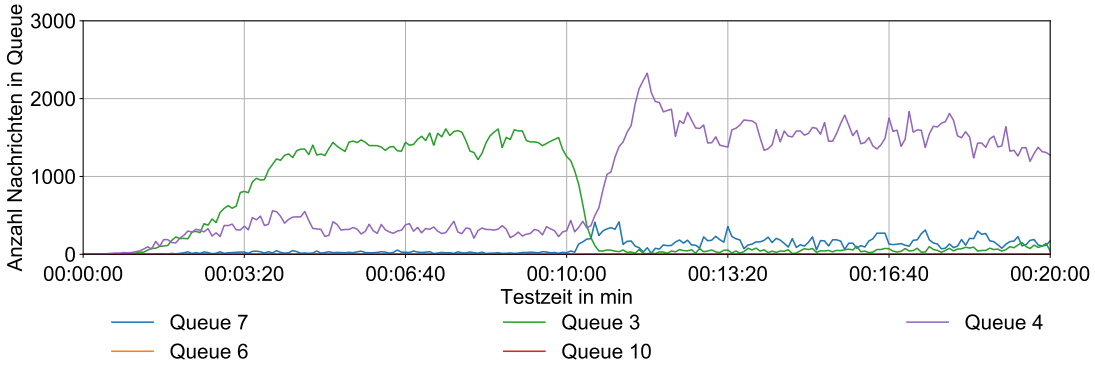


Abbildung 5.10: Länge der RabbitMQ Queues (Prometheus)

Während des Tests von Szenario 1 wurde zusätzlich zu den schon vorgestellten KPIs die Länge der Service-Queues gemessen (vgl. Abbildung 5.10). Es konnte festgestellt werden, dass während der Initialisierungsphase die Queue von Service „3“ bis zu 1500 Nachrichten enthielt. Danach sank die Anzahl der in dieser Queue enthaltenen Nachrichten auf annähernd null. Dafür stieg die Anzahl der Nachrichten in der Queue für Service „4“ nach der Initialisierung auf ca. 1500 Nachrichten, wobei Spitzen mit bis zu 2400 Nachrichten in der Queue gemessen wurden.

5.2 Stability-Test

Bei der Betrachtung der Agentenentwicklung der Stability-Tests (vgl. Abbildung 5.11) konnte festgestellt werden, dass die maximale Agentenanzahl für beide Szenarien nach 8,5 min erreicht wurde. Die Zeitspanne bis zum Erreichen der maximalen Agentenanzahl wird auch hier als Initialisierungsphase bezeichnet. Die maximale Anzahl belief sich bei diesen Tests auf 4800 Agenten. Wurde die Anzahl erreicht, konnte sie in beiden Szenarien konstant gehalten werden.

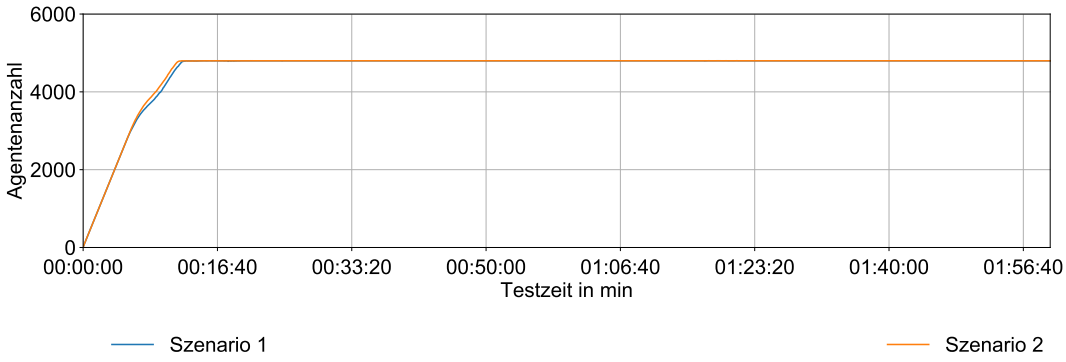
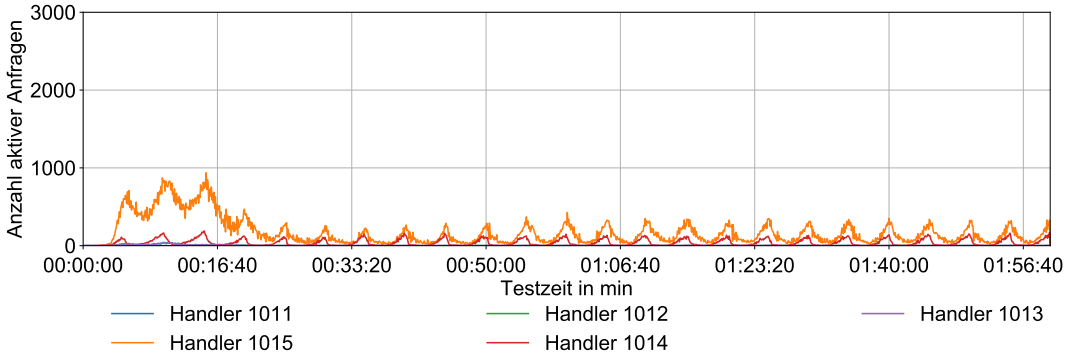


Abbildung 5.11: Agentenanzahl während der Stability-Tests

(a) Szenario 1



(b) Szenario 2

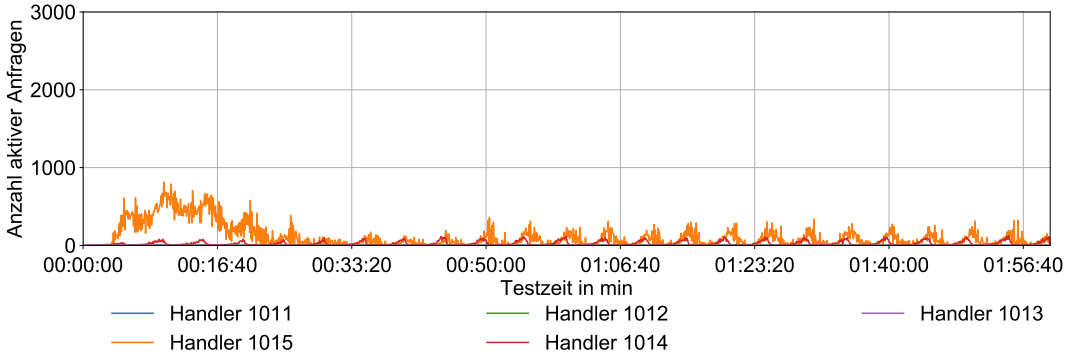


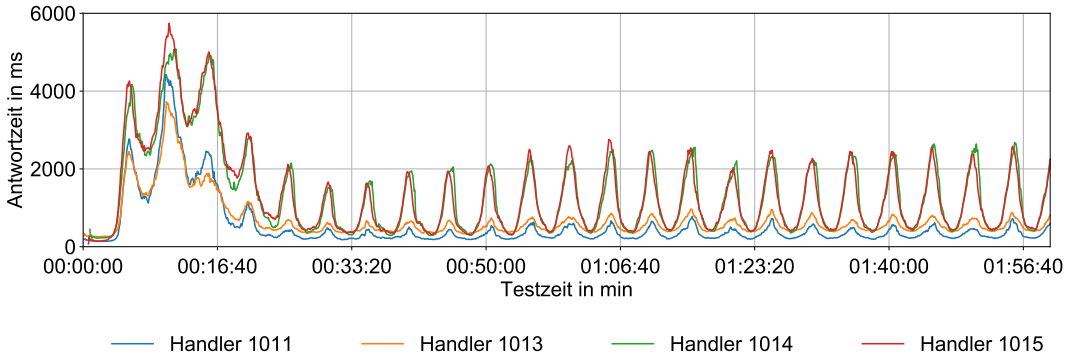
Abbildung 5.12: Aktive Anfragen während der Stability-Tests

Nach der Agentenentwicklung ist bei der Auswertung der sekündlichen Anfragen durch die Simulationsumgebung zu sehen, dass bei diesen Tests deutlich weniger aktive Anfragen

vorhanden waren als bei den Stress-Tests. Die Anzahl der Anfragen für beide Szenarien unterschied sich dabei nur geringfügig bezüglich der Lastspitzen. Die meisten Anfragen wurden an die Handler „1015“ und „1014“ gestellt. Dabei zeigte sich, dass sich nach den ersten Lastspitzen des Tests ein wiederkehrender Zyklus einstellte, der ebenfalls in beiden Szenarien zu sehen war.

5.2.1 Antwortzeit

(a) Szenario 1



(b) Szenario 2

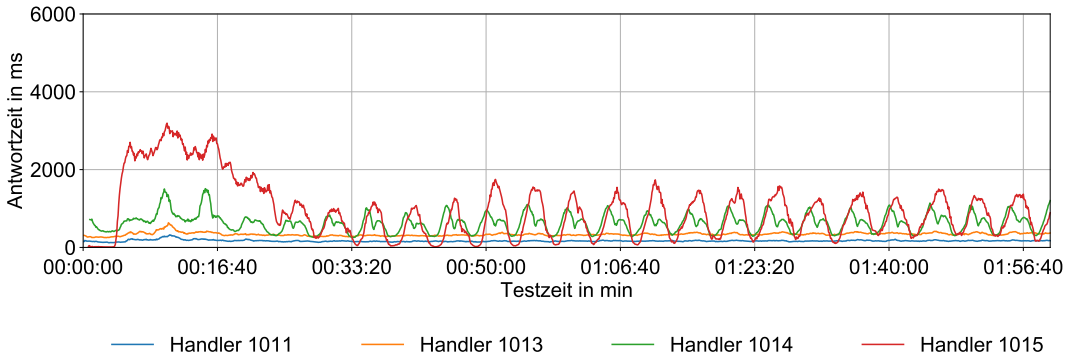
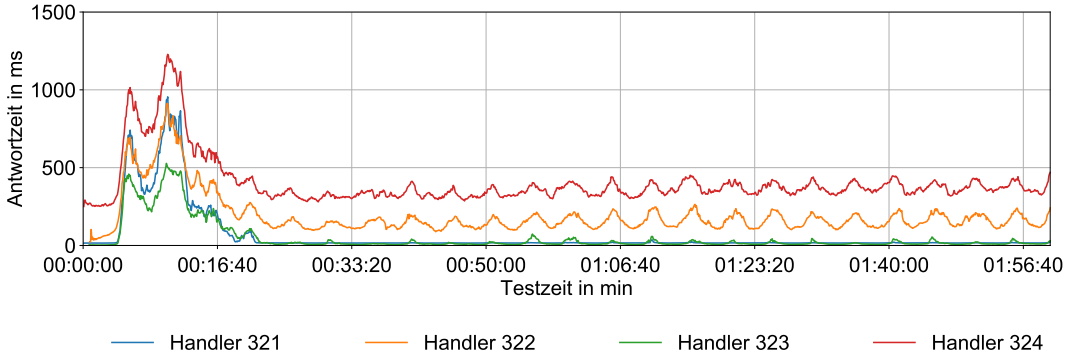


Abbildung 5.13: Antwortzeiten von Service „10“ (Prometheus)

Beim Vergleich der Antwortzeiten von Service „10“ fielen auch dort Zyklen auf (vgl. Abbildung 5.13). Die Zyklen wiederholten sich alle 5 min. Anfangs gab es eine große Spitze, über die Dauer von bis zu 20 min. Danach setzte der konstante Zyklus ein. Dabei pendelten die Handler „1014“ und “1015“ in Szenario 1 zwischen 500 ms und 3 s. Die Handler „1011“ und „1013“ pendelten sich zwischen 200 ms und 900 ms ein. Alle Handler

in Szenario 2 waren schneller als die Handler in Szenario 1. So pendelten die Handler „1014“ und “1015“ zwischen 100 ms und 1 s und die Handler „1011“ und „1013“ waren konstant bei 200 ms und 300 ms.

(a) Szenario 1



(b) Szenario 2

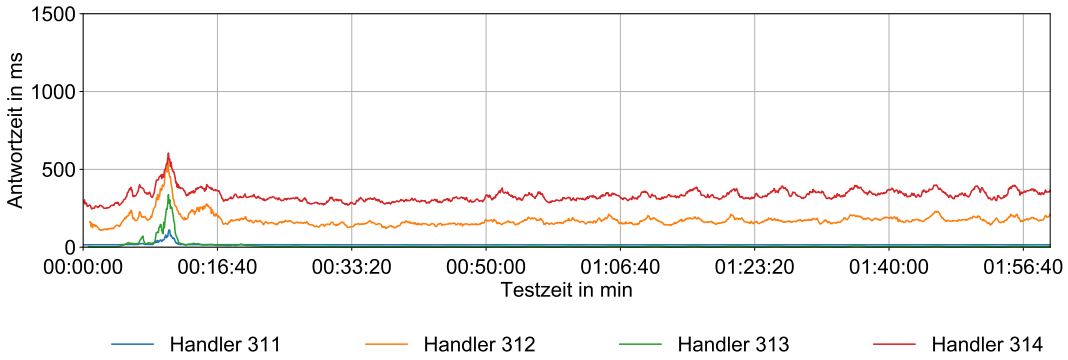
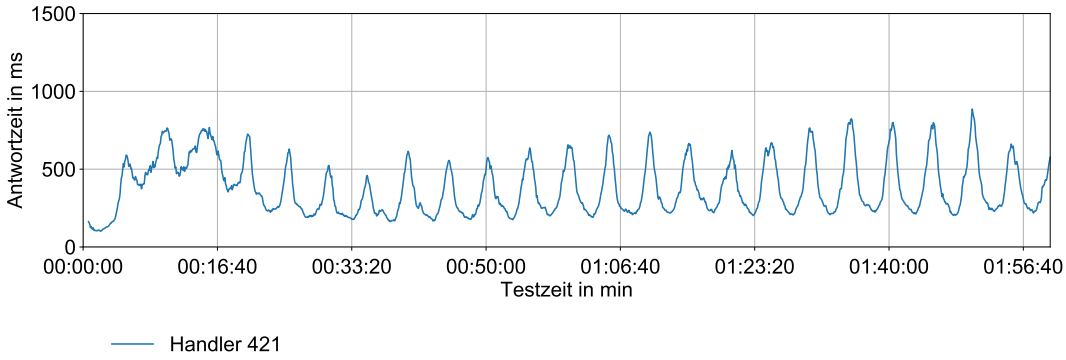


Abbildung 5.14: Antwortzeiten von Service „3“ (Prometheus)

Service 3 war im Stability-Test in beiden Szenarien im Verlauf der Tests annähernd gleich schnell (vgl. Abbildung 5.14). Einzig während der Initialisierung war Szenario 1 deutlich langsamer als Szenario 2. In Szenario 1 war der Zyklus deutlich erkennbar, in Szenario 2 hingegen konnte der Zyklus kaum ausgemacht werden.

(a) Szenario 1



(b) Szenario 2

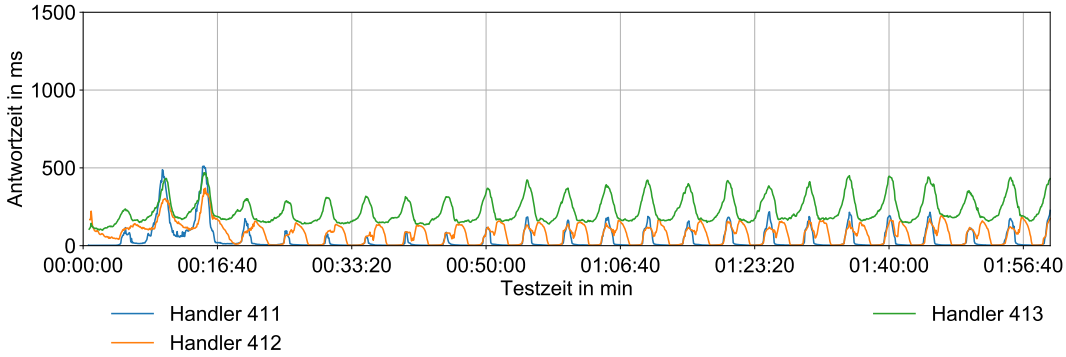


Abbildung 5.15: Antwortzeiten von Service „4“ (Prometheus)

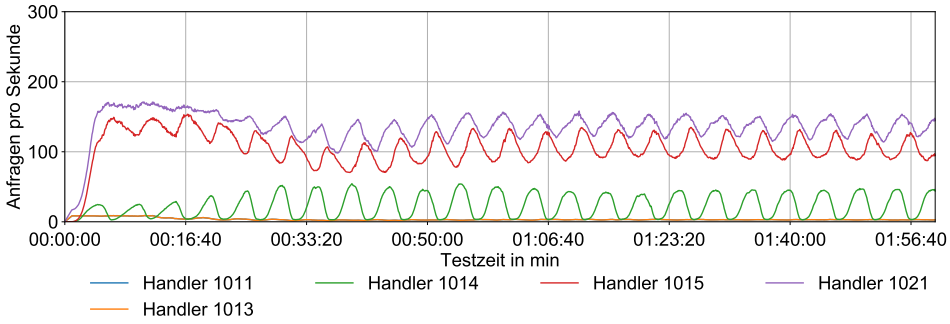
Bei der Untersuchung von Service 4 (vgl. Abbildung 5.15) fiel wieder die zyklische Veränderung des Antwortverhaltens auf. Dabei war der eine Handler von Szenario 1 jedoch langsamer als jeder Handler von Szenario 2. So benötigte Handler „421“ in Szenario 1 abhängig vom Zeitpunkt im Zyklus 250 ms bis 750 ms für die Verarbeitung einer Anfrage. Der langsamste Szenario 2 Handler („314“) schwankte zwischen 150 ms und 400 ms, während die beiden anderen Handler („311“ und „313“) außer während der Spitzen am Anfang maximal 100 ms benötigten.

5.2.2 Durchsatz

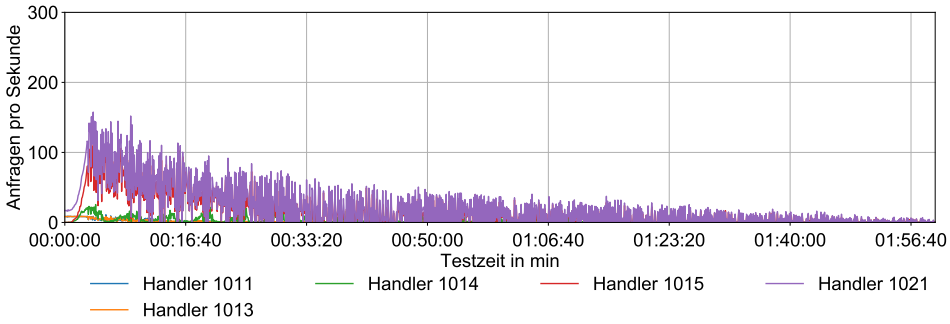
Auch im Durchsatz von Service „10“ konnte ein Zyklus beobachtet werden (vgl. Abbildung 5.16). Die Szenarien ähnelten sich bei der Prometheus-Datengrundlage nur minimal. Szenario 1 besaß wieder Handler „1021“, der nicht in Szenario 2 vorkam. Er schwankte bei

diesen Tests zwischen 100 und 150 Anfragen pro Sekunde. Das Verhalten von Handler „1015“ ähnelte sich bei beiden Szenarien auf Basis von Prometheus sehr, jedoch schwankte er in Szenario 1 zwischen 80 und 130 Anfragen pro Sekunde und in Szenario 2 zwischen 90 und 150 Anfragen pro Sekunde. Auch Handler „1014“ verhielt sich in beiden Szenarien sehr ähnlich. Dabei war Szenario 1 mit bis zu 50 Anfragen pro Sekunde jedoch meist langsamer als Szenario 2 mit bis zu 70 Anfragen pro Sekunde. Die anderen Handler waren in beiden Szenarien bei unter 10 Anfragen pro Sekunde. Die Diagramme von Szenario 2 auf beiden Datengrundlagen ähnelten sich sehr stark. Prometheus wies jedoch weniger starke Schwankungen auf. Szenario 1 auf Basis von ELK Stack fiel nach einem kurzen Anstieg auf annähernd 0 Anfragen pro Sekunde über den gesamten Test ab.

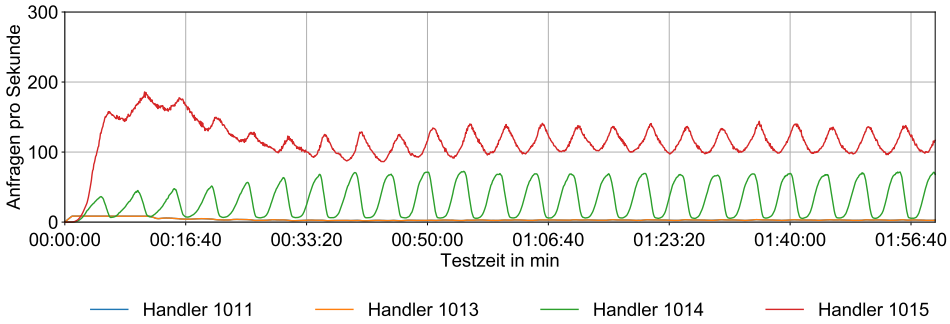
(a) Szenario 1



(b) Szenario 2 Elasticsearch



(c) Szenario 2



(d) Szenario 2 Elasticsearch

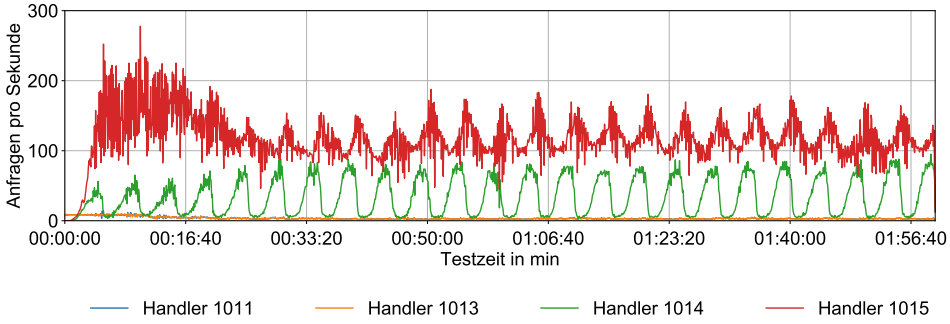
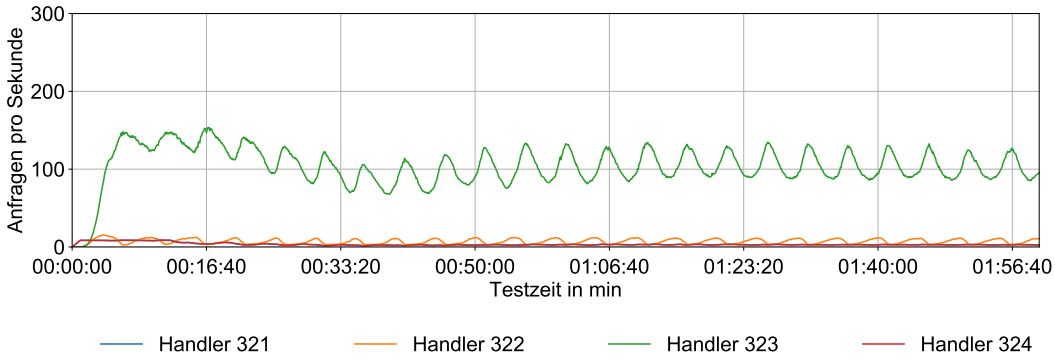


Abbildung 5.16: Durchsatz von Service „10“

(a) Szenario 1



(b) Szenario 2

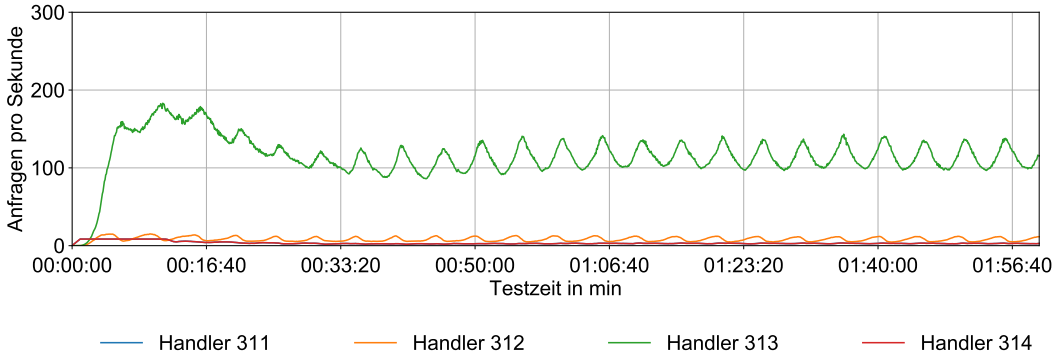
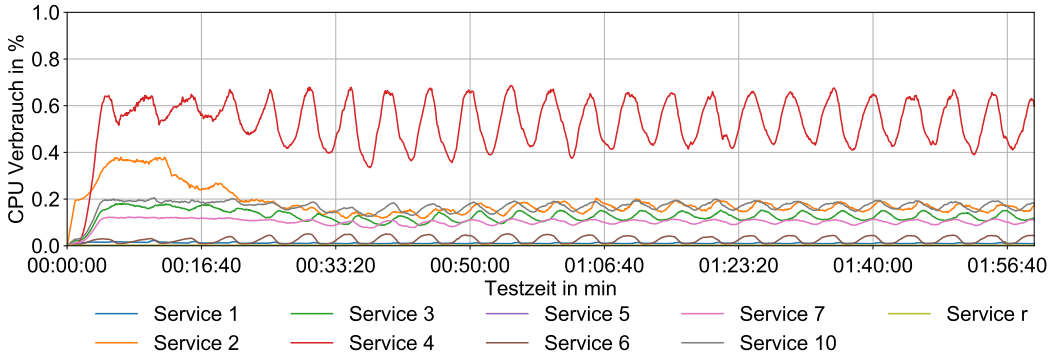


Abbildung 5.17: Durchsatz von Service „3“ (Prometheus)

Service 3 unterschied sich zwischen beiden Szenarien nicht (vgl. Abbildung 5.17). Die Handler „322“ und „323“ verliefen beide zyklisch. So schwankte Handler „323“ zwischen 80 und 140 Anfragen pro Sekunde. Alle anderen Handler lagen bei unter 10 Anfragen pro Sekunde.

5.2.3 Kapazität

(a) Szenario 1



(b) Szenario 2

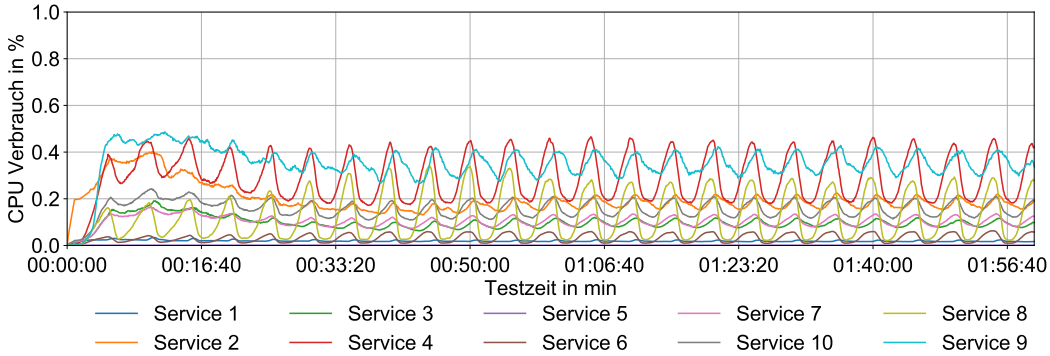
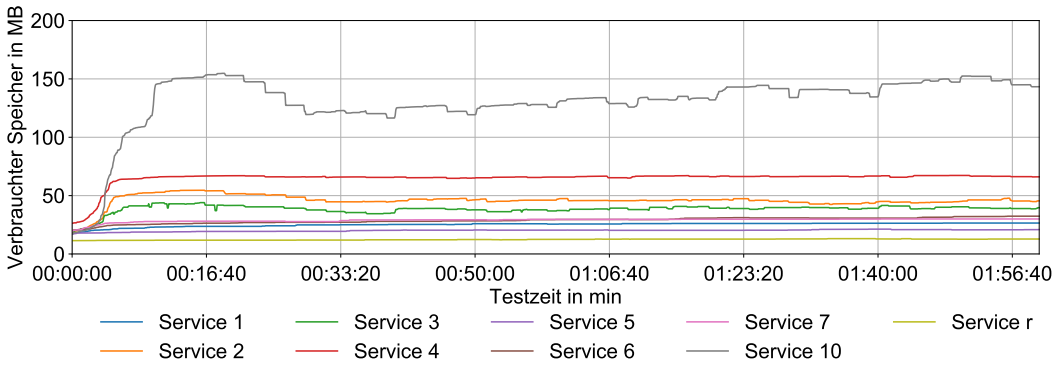


Abbildung 5.18: CPU-Verbrauch der Services (Prometheus)

Für den Vergleich des Ressourcenverbauch der beiden Szenarien wurde der prozentuale CPU-Verbrauch herangezogen (vgl. Abbildung 5.18). Alle Services in beiden Szenarien unterlagen einem Zyklus. Service „4“ in Szenario 1 schwankte dabei zwischen 0,4 % und 0,7 %. Die restlichen Services aus Szenario 1 benötigten nie mehr als 0,2 % CPU. In Szenario 2 beschränkte sich der maximale Verbrauch der Services auf 0,4 %. Die Services „4“, „8“ und „9“ benötigten mehr als 0,2 % CPU. Service „4“ wies dabei Schwankungen zwischen 0,2 % und 0,4 % auf. Service 8 schwankte zwischen annähernd 0 % und 0,35 % und Service 9 zwischen 0,35 % und 0,4 %.

(a) Szenario 1



(b) Szenario 2

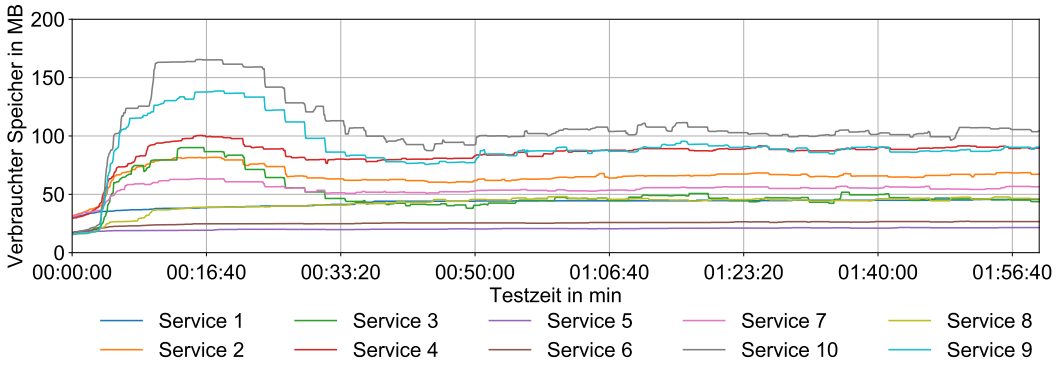


Abbildung 5.19: Speicherverbrauch der Services (Prometheus)

Auch der Speicherverbrauch wurde für den Vergleich der Szenarien im Stability-Test herangezogen (vgl. Abbildung 5.19). Service „10“ verbrauchte in beiden Szenarien den meisten Speicher. In Szenario 1 waren es nach der Initialisierungsphase 120 MB und stieg im Verlauf der Zeit auf 140 MB an. In Szenario 2 lag der Speicherverbrauch von Service „10“ nach der Initialisierung bei ungefähr 100 MB. Ansonsten verbrauchte in Szenario 1 kein Service mehr als 70 MB über den gesamten Testverlauf. In Szenario 2 verbrauchten Service 4 und 9 ca. 850 MB. Der benötigte Speicher nach der Initialisierungsphase bewegte sich auf dem gleichen Level und nährte sich einem Wert von 80 MB 100 MB an.

5.2.4 Message Queue

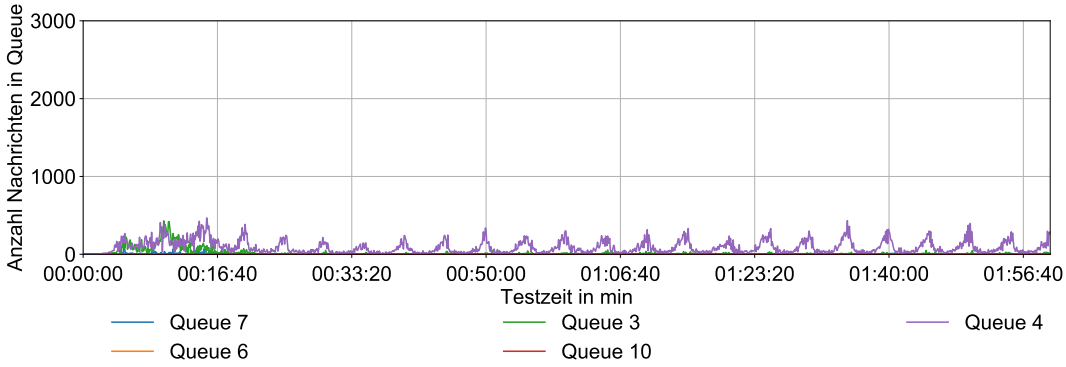


Abbildung 5.20: Länge der RabbitMQ Queues (Prometheus)

In den Stability-Tests wuchsen die Queues in Szenario 1 nie über 500 Nachrichten an (vgl. Abbildung 5.20). Über den gesamten Verlauf des Tests enthielten nur zwei Queues erkennbar Nachrichten. Während der Initialisierung enthielt die Queue für Service „3“ bis zu 500 Nachrichten. Im weiteren Verlauf war die Länge dieser Queue kleiner. Die Queue für Service „4“ schwankte jedoch über die gesamte Simulation hinweg zwischen 0 und 500 Nachrichten. Die Schwankungen traten ebenfalls zyklisch auf.

6 Diskussion

6.1 Ergebnisse

6.1.1 Simulation

Die Simulation der Agenten während der Tests ist entsprechend der Konfiguration problemlos verlaufen. In beiden Test-Szenarien wurden 8,5 min benötigt, bis die maximale Agentenanzahl erreicht wurde. Nach dem Anstieg begann in dieser Arbeit der eigentliche Testzeitraum. Ahmad et al. haben auch zwei verschiedene Test-Szenarien verwendet und dabei einen ähnlichen Verlauf der Anzahl der simulierten Nutzer wie in den Test-Szenarien dieser Untersuchung [1]. Auch Hao et al. haben für ihren agentenbasierten Test auf ein zyklisches Ansteigen der Agentenanzahl gesetzt. In ihrer Untersuchung war die Zeit zwischen dem Erstellen von Agenten lange, sodass ein stufenförmiger Verlauf entstanden ist [21].

Während des Stability-Tests konnte in beiden Szenarien nach der Initialisierung ein durch die Simulations-Konfiguration bestimmter Zyklus ausgemacht werden. Der Zyklus wiederholte sich in einem Rhythmus von 5 min. Er beeinflusste vor allem Handler, die für die Kreditaufnahme verwendet wurden. Dies sind die Handler, denen der vorgegebene Geschäftsprozess zugrunde liegt. Der Zyklus kam durch die zyklische Guthabenverteilung während der Test-Simulation zustande. So bekam jeder Agent alle 5 min neues Guthaben. Nachdem die Agenten das Guthaben bekommen hatten, mussten sie in der Regel keine neuen Kredite aufnehmen, weshalb keine Anträge mehr an das System gestellt wurden. Innerhalb von 5 min verbrauchten sie jedoch das gesamte Guthaben und mussten daraufhin wieder Kredite aufnehmen. Hierfür mussten sie Anfragen an das System stellen, wodurch der Zyklus zustande kam. So könnte das Entstehen dieses Zyklus innerhalb der Anwendungslandschaft als emergents Verhalten der Simulation für die Generierung von

Last verstanden werden, da nur definiert wurde, dass alle Agenten in einem bestimmten Zyklus Guthaben erhalten. Darüber hinaus wurden jedoch keine weiteren Vorgaben gegeben. Der Zyklus konnte nur in den Stability-Tests beobachtet werden.

Der Stress-Test weist keinen Zyklus auf, da über die Einstellung „MoneyStart“ sichergestellt wurde, dass die Agenten nur sehr selten ausreichend Guthaben besaßen und deshalb immer Kredite aufnehmen müssen. Agenten im Stress-Test verwendeten das erhaltene Guthaben immer dazu, um alte Kredite zu begleichen und danach neue Kredite aufnehmen zu können. Doch selbst ohne diese Einstellung wäre der Zyklus nur sehr schwach ausgeprägt, weil über die gesamte Simulationsdauer die maximale Anzahl an Anfragen durch die Agenten aktiv gehalten wurde. So sind in der Summe immer 2500 Anfragen in beiden Szenarien aktiv. Es kann davon ausgegangen werden, dass sogar noch mehr Agenten Anfragen stellen wollten, diese jedoch in einer Queue zwischengespeichert werden mussten, um die Performance der Test-Simulation nicht zu stark zu beeinflussen. Damit wird der Zyklus verschleppt, da die Anträge der Agenten nicht mehr zu genau dem Zeitpunkt versendet werden, zu dem sie eigentlich gestellt wurden. Hinzu kommt, dass durch diese Begrenzung im Stress-Test weniger Anfragen bearbeitet werden können, da zu viele Anfragen auf Bearbeitung warten.

6.1.2 Leistungsfähigkeit

Szenario 2 schneidet in beiden Test-Szenarien besser ab als Szenario 1. Auch Hong et al. haben ein System auf Basis von RabbitMQ mit einem REST-basierten System verglichen. Sie haben in ihrer Untersuchung mehrere unterschiedliche Mengen von simulierten Nutzern für die Tests verwendet. Dabei erhielten sie ein sehr ähnliches Ergebnis. In dieser Untersuchung wurden mit der steigenden Menge an simulierten Nutzern immer mehr Anfragen an die beiden untersuchten Systeme gestellt. Beide Systeme konnten mit sehr wenigen Nutzern gleich viele Anfragen bearbeiten. Mit steigender Nutzerzahl konnte das REST-System immer mehr Anfragen bearbeiten, während das RabbitMQ-System stagnierte. Nach dem Überschreiten einer kritischen Nutzeranzahl wurde das REST System zu instabil, um den Service in der nötigen Qualität erbringen zu können. Ab diesem Punkt ist auch die Fehlerrate des REST Systems stark angestiegen. Laut Hong et al. ist der Grund für dieses Verhalten das vollständige Verbrauchen der Ressourcen durch das REST System. [23]

Die Ergebnisse der Tests bestätigen das von Hong et al. beschriebene Verhalten. Bei

niedriger Last im Stability-Test konnten beide Szenarien die gleiche Menge an Anfragen bewältigen. Allerdings konnten im Stress-Test beide Szenarien nicht mehr die nötigen QoS erbringen, da die Antwortzeiten in beiden Szenarien sehr hoch waren. Szenario 2 konnte aber noch deutlich mehr Anfragen als Szenario 1 verarbeiten. Es wurde zu keiner Zeit eine kritische Nutzeranzahl überschritten, die zu einem instabilen Verhalten von Szenario 2 führte. Da sich die Fehlerquoten beider Szenarien nicht unterschieden, wurden sie in dieser Arbeit nicht aufgeführt.

Es konnte auch festgestellt werden, dass Szenario 2 mehr Ressourcen verbraucht als Szenario 1. Naik hat in einem Survey verschiedene Kommunikationstechnologien verglichen und dabei festgestellt, dass HTTP Kommunikation mehr Ressourcen als MAQP benötigt [36]. Dies deckt sich mit den Testergebnissen, da die Services in Szenario 1 deutlich mehr Speicher benötigen als in Szenario 2. Hierbei muss jedoch berücksichtigt werden, dass durch den Engpass bei Service „4“ und der damit verbundenen geringeren Anzahl von Anfragen pro Sekunde durch die Test-Simulation, während des Stress-Tests in Szenario 1 weniger Anfragen verarbeitet werden mussten. Dies kann auch ein Grund für den geringeren Ressourcenverbrauch in Szenario 1 sein, wodurch für dieses Test-Szenario nur schwer Vergleiche in diesem Bereich angestellt werden können. Die Unterschiede vor allem bezüglich des Speicherbedarf sind jedoch groß, da in Szenario 2 fünf Handler über 100 MB sind während in Szenario 1 nur ein Service „10“ darüber ist.

Auch während des Stability-Tests konnte ein erhöhter Ressourcenverbrauch durch Szenario 2 ausgemacht werden. Der Service, der den meisten Speicher benötigt, ist in beiden Szenarien Service „10“. Das kann dadurch erklärt werden, dass er in beiden Szenarien auf REST basiert und alle Anfragen an das System entgegen nimmt und weiterleitet. Ansonsten verbrauchen besonders die Services, die Anfragen an andere Services weiterleiten, in Szenario 2 mehr Speicher als in Szenario 1. Auch der CPU-Verbrauch während des Stability-Tests wird besonders durch die weiterleitenden Services bestimmt. Service „4“ in Szenario 1 verbraucht zwar die meiste CPU-Zeit von allen Services in beiden Szenarien, er übernimmt jedoch mit einem Teil des RabbitMQ Servers die Aufgaben von Service „4“, „8“ und „9“ aus Szenario 2, die zusammen mehr CPU benötigten. Es wird vermutet, dass der Mehrverbrauch durch Szenario 2 damit zusammenhängt, dass ein Service in einem Event-driven System nur auf Events reagiert und eventuell daraus resultierende Events verschickt ohne danach in einem Prozess auf eine Antwort zu warten. So kann das Warten auf die Antwort einer synchronen Anfrage und das damit verbundene Blockieren des Verarbeitungs-Threads vermieden werden. Darüber hinaus können RabbitMQ Consumer, wie von Fernandes et al. beschrieben, nur so viele Nachrichten aus der Queue holen, wie

sie parallel verarbeiten können [19]. Dies ist bei REST basierten Lösungen der Fall, die solange wie möglich alle Anfragen eines Services annehmen und für die Bearbeitung einen Thread verwenden.

6.1.3 Flaschenhälse

Die beschriebenen Antwortzeitprobleme in den Szenarien und darüber hinaus die Probleme beim Verarbeiten von Anfragen durch Szenario 1 im Stress-Test weisen auf Flaschenhälse in beiden Szenarien hin. Um diese ausfindig zu machen, wurden verschiedene Definitionen von schlechten Pattern in KPIs herangezogen. Die Pattern wurden dann verwendet, um die gesammelten Daten auf Merkmale der Pattern hin zu untersuchen. Einer der problematischen Handler ist „1015“, der bei 2500 Anfragen pro Sekunde nach der Initialisierungsphase für die Antwort in Szenario 2 9 s und in Szenario 1 sogar 13 s benötigte. Während der Initialisierung benötigte Handler „1015“ sogar noch länger. In dieser Zeit benötigten jedoch auch die anderen in beiden Szenarien vertretenen Handler während des Stress-Tests teilweise bis zu 14 s um zu antworten. Aus diesem Grund wurden beide Szenarien bezüglich der Existenz von Flaschenhälsen untersucht.

Smith und Williams konnten in ihren Untersuchungen zu Flaschenhälsen in Systemen verschiedene Pattern ausmachen. Zwei Pattern, die auch in dieser Arbeit vorkommen sind One Lane Bridge (OLB) und Empty Semi Trucks (EST). Sie haben OLBs als Punkt während der Ausführung einer Applikation definiert, an dem nur sehr bedingt oder überhaupt nicht parallel prozessiert werden kann. EST bedeutet, dass sehr viele Anfragen benötigt werden, um eine Aufgabe zu erfüllen [44, 43]. Wert et al. haben in ihren Untersuchungen die Pattern wieder aufgegriffen und erweitert [50, 51]. Laut ihrer Definition treten OLBs auf, wenn eine passive Ressource die Parallelität einer Applikation einschränkt. Als passive Ressourcen verstehen sie Mutexes, Connection Pools oder Datenbank Locks. Ein OLB liegt nach ihrer Definition vor, wenn hohe Antwortzeiten mit geringen CPU-Verbrauch zusammen fallen. Sie erweiterten den EST dahingehend, dass dieser vorliegt, wenn viele Einzel-Nachrichten anstatt einer gebündelten Nachricht versendet werden. Dadurch können Probleme auftreten, denn durch jede Nachricht wird ein gewisser Overhead verursacht. Darüber hinaus haben Wert et al. die Bedeutung von Bottleneck Resource (BR) definiert [50]. BR tritt auf, wenn das System nicht die benötigten Ressourcen zur Verfügung hat, beispielsweise wenn der CPU Verbrauch sehr hoch ist.

Bei der Analyse der Ergebnisse können Symptome aller vorgestellten Ursachen entdeckt werden. Für eine genaue Analyse der vorhandenen Symptome müssten die Szenarien jedoch weiter vor diesem Hintergrund untersucht werden. Hierfür sind weitere Test-Szenarien nötig. Die Ergebnisse des Stress-Tests lassen mit den teilweise sehr hohen Antwortzeiten darauf schließen, dass Flaschenhälse in den Services vorhanden sind. Während des Stability-Tests konnten jedoch keine Flaschenhälse ausgemacht werden.

Verantwortlich für die langen Antwortzeiten der Handler während der Initialisierungsphase in beiden Szenarien war Service „2“. Die Ergebnisse lassen in beiden Szenarien jedoch unterschiedliche Ursachen vermuten. So benötigte Service „2“ in Szenario 2 während der Initialisierungsphase 0,5 % CPU, was dem konfigurierten maximalen CPU-Verbrauch des Services entspricht. Hinzu kommen die sehr hohen Antwortzeiten des Services, die durch die Messungen von Service 3 identifiziert werden konnten. Die Antwortzeiten von Service „3“ können mit denen von Service „2“ gleichgesetzt werden, da Service „3“ nur die Aufgabe hat, die Anfragen an Service „2“ weiterzuleiten. Daraus kann geschlossen werden, dass bei Service „2“ in Szenario 2 bei der durch den Stress-Test gegebenen Anzahl an Anfragen ein BR vorlag. Demgegenüber waren die Antwortzeiten von Service „2“ in Szenario 1 mit weniger als 2 s sehr kurz und Service „2“ verbrauchte während der Initialisierung konstant 0,4 % CPU. Die Antwortzeiten der Handler waren in Szenario 1 trotzdem sehr hoch. Dies lässt auf eine OLB in Service „3“ schließen. Wird auch die Länge der RabbitMQ Queues in Szenario 1 betrachtet, kann davon ausgegangen werden, dass Service „3“ nicht über genügend Worker-Threads für das Verarbeiten von Nachrichten verfügte und deshalb nicht genug parallele Anfragen an Service „2“ stellte. Zusätzlich schöpfte er die verfügbaren Ressourcen nicht voll aus. Ein Grund dafür könnte sein, dass der für die Nachrichtenverarbeitung verwendete Worker-Pool nicht dynamisch skalieren kann. Stattdessen muss er, wie in Unterabschnitt 2.5.1 beschrieben, vor dem start des Service in der Konfiguration festgelegt werden. Daraus kann geschlossen werden, dass ein OLB vorliegt, dem der Worker-Pool zugrunde liegt.

Nach der Initialisierung können in beiden Szenarien an unterschiedlichen Stellen weitere Flaschenhälse ausgemacht werden. In dieser Phase hatten die Handler „1014“ und „1015“ in Szenario 1 lange Antwortzeiten. Hier kann davon ausgegangen werden, dass ein OLB in Service „4“ im Nachrichten-Worker-Pool vorlag, da dieser Pool eine feste Größe besaß. Dies zeigte sich in der Länge der RabbitMQ Queue für Service „4“ und dem damit verbundenen gringer Ressourcenverbrauch durch Service „4“. Die Queue enthielt nach der Initialisierung durchgehend über 1000 Nachrichten. So war der CPU- und Speicher-Verbrauch von Service „4“ weit vom Maximum entfernt. In Szenario 2 kann der Flaschenhals wiederum durch BR

erklärt werden. So benötigte Service „9“ nach der Initialisierung 0,5 % CPU. Das ist das für diesen Service definierte Maximum, weshalb die Anfragen nicht schneller bearbeitet werden konnten.

6.2 Messsysteme

Prometheus ließ sich bei der Vorbereitung der Systeme für die Untersuchung ohne Probleme und schnell aufsetzen. Das war möglich, da Prometheus nur aus einem einzigen Server besteht, der seine eigene Datenbank besitzt. Bei dem Sammeln der Daten in der untersuchten Größenordnung kam der Server mit den eingestellten 200 m CPU und 400 Mi Speicher aus. Für alle in dieser Arbeit verwendeten Infrastruktur-Komponenten sind Prometheus-Exporter vorhanden, sodass diese Systeme überwacht werden können. Dabei ist es jedoch nötig, mindestens eine weitere Applikation, die dem jeweiligen Exporter entspricht, pro Systemart zusätzlich in der Lösung zu installieren. Ein Integrationsproblem von Prometheus könnte auftreten, wenn eine Applikation ohne Exporter überwacht werden soll. Bei solchen Systemen könnten es sich um weniger verbreitete Infrastruktur-Komponenten oder Eigenentwicklungen handeln. Besteht für diese Systeme eine schon vorhandene Schnittstellen um Metriken zu exportieren, kann ein Prometheus Exporter entwickelt werden. Ist dies nicht möglich, ist der einzige Weg, um Prometheus verwenden zu können, eine Integration des Prometheus Codes in die Applikation. Durch die Verwendung des Kubernetes-Operators für Prometheus in dieser Arbeit, wurde die Bereitstellung einer Prometheus-Instanz noch einmal deutlich vereinfacht.

Der Einsatz des ELK Stacks erfordert mehr Aufwand als die Verwendung von Prometheus und bedarf einer ständigen Wartung. So hat das Sammeln und Speichern von Logs sowohl Vorteile als auch Nachteile. Der Vorteil des Sammelns von Logs ist, dass keine Änderungen an bestehenden Systemen vorgenommen werden müssen, sofern diese schon sinnvolle Logs generieren. Der Nachteil ist jedoch, dass für das Verarbeiten, Speichern und Analysieren der Logs sehr viele Ressourcen benötigt werden [29]. Für die Installation von Elasticsearch selbst werden, wie beschrieben, mehrere Server benötigt, die mit genügend Ressourcen ausgestattet sein müssen. Daneben werden die schon vorgestellten Applikationen für die Bearbeitung und Sammlung der Logs verwendet. Vor allem das Verarbeiten von Logs kann sehr viele Ressourcen benötigen. Die Einführung des ELK Stacks hat somit einen großen Einfluss auf das bereits vorhandene System und benötigt speziell dafür vorgesehene Server.

Das Speichern von Zeitreihen in Prometheus erfordert nur wenige Ressourcen. In Zeitreihen werden neben Tags nur Zahlen gespeichert, die allerdings jeder Zeit auf unterschiedlichste Weise analysiert werden können. Diese Einschränkungen der Datenstruktur machen die Auswertung der Daten einfach. Es können jedoch nur allgemeine Metriken für eine Applikation berechnet werden, da keine individuellen Nachrichten oder Parameter gespeichert werden. Wie diese Untersuchung gezeigt hat, kann bei der Überwachung eines Systems mithilfe dieser Daten bereits eine Menge Informationen erfasst werden.

Durch die Speicherung von Texten in Elasticsearch können die Analysen weit über die Analysen mithilfe von Prometheus hinausgehen. Dies ist möglich, da die kompletten Logs nach beliebigen Themen analysiert und nicht nur durch die überwachten Systeme vorgefertigte Metriken eingesehen werden können. So können jederzeit neue Metriken aus den Logs erhoben werden. Solange die benötigten Metriken aus den schon erhobenen Logs berechnet werden können, müssen die überwachten System dafür nicht geändert werden. Wie die Ergebnisse dieser Untersuchung zeigen, ist es aber möglich, die gleichen Metriken wie über Prometheus zu erhalten. Durch die Untersuchung können die Probleme bei dem Sammeln von Logs verdeutlicht werden. Das zeigt sich in der Darstellung der durch den ELK Stack bereitgestellten KPIs für Szenario 1. Der Verlauf entstand durch eine falsch konfigurierte Komponente des Applikations-Frameworks, wodurch sehr viele Logs für das Empfangen und Versenden von Nachrichten über RabbitMQ erzeugt wurden. Durch die nicht vorhergesehene Menge an Logs wurde der ELK Stack überlastet, sodass nicht alle Logs indexiert werden konnten und Daten verloren gingen. Hinzu kam, dass die Metriken direkt nach der Simulation erhoben wurden und deshalb noch viele Logs auf Indexierung warteten. Diese Probleme können schnell auftreten, da Logs aus Text bestehen und deshalb viel Speicher verbrauchen können. Das zeigte sich auch im Verbrauch des Festplattenspeichers der Elasticsearch-Knoten. Dieser war so hoch, das nach einigen Testläufen alle in Elasticsearch gehaltenen Daten gelöscht werden mussten.

7 Zusammenfassung und Fazit

Diese Arbeit befasste sich zunächst mit einem allgemeinen Überblick der Thematik, um die wichtigsten Begrifflichkeiten festzulegen. Dabei wurden die in vorangegangenen Arbeiten entwickelten Frameworks vorgestellt, die im weiteren Verlauf dieser Arbeit benötigt wurden, um verschiedene Performance-Tests auf einer beispielhaften Anwendungslandschaft ausführen zu können. Ein Framework bot die Möglichkeit, auf der Basis von GoLang, schnell neue Services für eine Anwendungslandschaft zu erstellen. Mit dem zweiten Framework konnten Agenten auf Basis von MARS implementiert werden. Darüber hinaus wurden die für die Auswertung der Tests verwendeten Systeme vorgestellt und genauer beschrieben.

Um Flaschenhälse in einer auf einem Geschäftsprozess basierenden Anwendungslandschaft untersuchen zu können, wurde der Geschäftsprozess einer beispielhaften Bank herangezogen. Hierfür wurden im weiteren Verlauf der Arbeit einzelne Services erstellt. Für die Generierung der Last wurde ein Bedürfnis-getriebenes ABM mit dem Framework implementiert, das den vorgestellten Geschäftsprozess ausführen kann. Für das Ausführen von Tests wurden noch weitere Systeme wie Prometheus oder der ELK Stack eingeführt und an die mit den Frameworks erstellten Systeme angebunden.

Nach dem Aufsetzen aller benötigten Systeme in der ICC, wurden zwei verschiedene Arten von Performance-Tests ausgeführt. Hierfür wurde das implementierte ABM mithilfe einer sehr leichtgewichtigen Simulationsumgebung in der ICC simuliert. Während der Testläufe wurden Daten in Prometheus und Elasticsearch erfasst. Nach den Tests und der Aufbereitung der gesammelten Daten konnten diese dargestellt und analysiert werden. Dabei konnten auf Basis von bekannten Pattern Flaschenhälse in beiden Systemen identifiziert werden.

7.1 Fazit

Die Untersuchung zeigte, dass eine agentenbasierte Simulation mit einem Modell auf Basis von MARS für das Generieren von Last, bei einem Performance-Test eingesetzt werden kann. Es müssen jedoch verschiedene Punkte berücksichtigt werden. So wurde die Last in dieser Untersuchung mit einem leichtgewichtigen Simulations-System generiert, dass das mit MARS implementierte Modelle mit nur geringen Änderungen ausführen konnte, anstatt das MARS-System direkt zu verwenden. Damit konnte die volle Kontrolle über den Ablauf der einzelnen Ticks gewonnen werden, was für die Durchführung der Tests wichtig war, da das SUT dem realen Zeitverlauf unterliegt und dies bei der Generierung der Last berücksichtigt werden muss. Darüber hinaus konnten durch die Verwendung des leichtgewichtigen Systems viele weitere Einschränkungen des komplexen MARS-Systems vermieden werden, da in dieser Untersuchung nur einfache Simulationseigenschaften benötigt wurden. Ein weiterer wichtiger Aspekt bei ABS-basierten Performance-Testes ist die Kommunikation zwischen der Simulation und dem SUT. Hier hat sich gezeigt, dass der gewählte Weg nicht optimal war, da die Verarbeitung der REST-Anfragen viele Ressourcen benötigte und so der Ablauf der Simulation negativ beeinflusst wurde. Für die Entscheidungsfindung der Agenten wurde GOAP verwendet. Dies war für den prototypischen Test dieser Arbeit ausreichend, für komplexere Szenarien sollte jedoch ein in der Agentenmodellierung weiter verbreitetes Verfahren oder eine optimiertere Planung mit GOAP verwendet werden. So könnte die Berechnungsdauer pro Tick während der Simulation deutlich gesenkt werden. Der hierfür nötige Austausch der Planungskomponente im Framework ist problemlos möglich.

Die Analyse der durch die Tests generierten Daten ergab, dass das auf REST basierende Szenario auf Anfragen meist schneller antwortete und mehr Anfragen verarbeiten konnte als das Event-basierte Szenario. Demgegenüber verbrauchte das Event-Szenario bei gleicher Last weniger Ressourcen. Außerdem hat sich für beide Szenarien gezeigt, dass es mit den KPIs, die alle über Prometheus erhoben werden konnten, möglich war, Services ausfindig zu machen, die Flaschenhälse enthielten. Um die Services genauer analysieren zu können und die Flaschenhälse auf einzelne Komponenten eines Services einzuschränken, wären jedoch weitere Informationen benötigt worden. Diese können entweder in Form von Wissen über die Implementierungsdetails, durch weitere Metriken oder durch eine tiefere Analyse der Logs erhalten werden.

Es zeigte sich auch, dass beide verwendeten Messsysteme ihre Vor- und Nachteile haben. Prometheus ist ein sehr leichtgewichtiges System, das bei Vorhandensein der nötigen

Schnittstellen auf Seiten des zu überwachenden Systems schnell eingerichtet und ohne große Probleme betrieben werden kann. Trotz des einfachen Aufbaus können mit Prometheus sehr wichtige und auch komplexe KPIs erhoben werden. Im Vergleich dazu ist der ELK Stack ein sehr komplexes System, das aktive Pflege und sehr viele Ressourcen benötigt. Dafür bietet es jedoch für die Analyse der erfassten Daten ein größeres Repertoire an Möglichkeiten.

7.2 Ausblick

Mit dieser Arbeit wurde gezeigt, dass es gut möglich ist, mit einer verteilten ABS Performance-Tests für eine Anwendungslandschaft durchzuführen. Aufbauend auf dem implementierten ABM und den beiden vorhandenen Szenarien könnte eine noch umfassendere Untersuchung mit weiteren Last-Szenarien gestartet werden. So könnten die Grenzen der Szenarien genauer untersucht werden. Für weiterführende Projekte sollte jedoch überprüft werden, ob anstatt der ICC eine Umgebung verwendet werden kann, in der mehr Rechte für die Untersuchung zur Verfügung stehen, da die ICC ein sehr striktes Rechtemanagement umsetzt. Ein weiteres interessantes Thema wäre, die von Syer et al. untersuchte Korrelation von Metriken und Logs weiterzuführen, sodass für eine Analyse auf Basis der Logs nur noch die wichtigen Logs eingesehen werden müssen. [47]

Zudem könnten für weitere Projekte aufbauend auf einer verteilten ABS ein adaptives und autonomes System für die Analyse von Anwendungslandschaften erstellt werden. Auf diese Weise könnte ein sich an den realen Gebrauch anpassendes nutzerbasiertes Test-System in einer Build-Pipeline realisiert werden. Die größte Herausforderung wäre dabei die automatische Anpassung des Verhaltensmodells an die realen Gegebenheiten. Das Verhaltensmodell könnte jedoch, wie Vögele et al. beschrieben haben, auf der Basis von erhobenen Logs generiert werden [48]. Erste Ansätze eines adaptiven Test-Systems haben Schulz et al. vorgestellt [42].

Literaturverzeichnis

[1] Ahmad, Tanwir ; Truscan, Dragos ; Porres, Ivan: MBPeT: A model-based performance testing tool / 2012 Fourth International Conference on Advances in System Testing and Validation Lifecycle. 2012. – Forschungsbericht

[2] Arora, Pardeep K. ; Bhatia, Rajesh: A Systematic Review of Agent-Based Test Case Generation for Regression Testing. In: *Arabian Journal for Science and Engineering* 43 (2018), feb, Nr. 2, S. 447–470. – ISSN 21914281

[3] Bagnasco, S ; Berzano, D ; Guarise, A ; Lusso, S ; Masera, M ; Vallero, S: Monitoring of IaaS and scientific applications on the Cloud using the Elasticsearch ecosystem. In: *Journal of Physics: Conference Series* 608 (2015), may, Nr. 1. – ISSN 17426596

[4] Bai, Jun: Feasibility analysis of big log data real time search based on Hbase and ElasticSearch. In: *Proceedings - International Conference on Natural Computation*, IEEE, jul 2013, S. 1166–1170. – ISBN 9781467347143

[5] Baiquan, Xu: Design of platform for performance testing based on JADE. In: *Proceedings - 2014 6th International Conference on Measuring Technology and Mechatronics Automation, ICMTMA 2014*, IEEE, jan 2014, S. 251–254. – ISBN 978-1-4799-3435-5

[6] Beck, Fabian: *Modell eines Kunden auf einem Energievergleichsportal*, HAW Hamburg, Hausarbeit, 2017

[7] Beck, Fabian: *Erfassung von Logs eines in Kubernetes verteilten Geschäftsprozesses*, HAW Hamburg, Hausarbeit, 2018

[8] Bellifemine, Fabio ; Caire, Giovanni ; Greenwood, Dominic: *Developing Multi-Agent Systems with JADE.* Chichester, UK : John Wiley & Sons, Ltd, mar 2007 (Wiley Series in Agent Technology). – 1–286 S. – ISBN 9780470057476

[9] Bernardino, Maicon ; Zorzo, Avelino F. ; Rodrigues, Elder M.: Canopus: A Domain-Specific Language for Modeling Performance Testing. In: *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, IEEE, apr 2016, S. 157–167. – ISBN 9781509018260

[10] Bonabeau, Eric: Agent-based modeling: methods and techniques for simulating human systems. In: *Proceedings of the National Academy of Sciences* 99 (2002), may, Nr. suppl. 3, S. 7280–7287. – ISBN 0027-8424 (Print)\r0027-8424 (Linking)

[11] Brailsford ; C., Sally ; Brailsford ; C., Sally: *Modeling human behavior: an (ID)entity crisis?* 2014

[12] Brattstrom, Morgan ; Morreale, Patricia: Scalable Agentless Cloud Network Monitoring. In: *Proceedings - 4th IEEE International Conference on Cyber Security and Cloud Computing, CSCloud 2017 and 3rd IEEE International Conference of Scalable and Smart Cloud, SSC 2017*, IEEE, jun 2017, S. 171–176. – ISBN 9781509066438

[13] Casti, John L.: *Would-Be Worlds: How Simulation is Changing the Frontiers of Science*. J. Wiley, 1997. – xii + 242 pages S. – ISBN 0-471-12308-0

[14] Chece, Sebastian: *Lasttest Definition, Ziele, Best Practices und Fehlervermeidung.* – URL https://www.testing-board.com/lasttest-und-performancetest/. – Zugriffsdatum: 2018-10-07

[15] Chen, Tse-Hsun ; Syer, Mark D. ; Shang, Weiyi ; Jiang, Zhen M. ; Hassan, Ahmed E. ; Nasser, Mohamed ; Flora, Parminder: Analytics-Driven Load Testing: An Industrial Experience Report on Load Testing of Large-Scale Systems. In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, IEEE, may 2017, S. 243–252. – ISBN 978-1-5386-2717-4

[16] Daniel Bryant: *Scaling Microservices at Gilt with Scala, Docker and AWS.* – URL https://www.infoq.com/news/2015/04/scaling-microservices-gilt. – Zugriffsdatum: 2019-02-24

[17] Dugan, Robert F. ; Glinert, Ephraim P. ; Shokoufandeh, Ali: The Sisyphus database retrieval software performance antipattern. In: *Proceedings of the third international workshop on Software and performance - WOSP '02*. New York, New York, USA : ACM Press, 2004, S. 10. – ISBN 1581135637

[18] Elastic: *Elastic Stack and product documentation*. 2018. – URL https://www.elastic.co/guide/index.html

[19] Fernandes, Joel L. ; Lopes, Ivo C. ; Rodrigues, Joel J. ; Ullah, Sana: Performance evaluation of RESTful web services and AMQP protocol. In: *International Conference on Ubiquitous and Future Networks, ICUFN*, IEEE, jul 2013, S. 810–815. – ISBN 9781467359900

[20] Hamilton, James ; Schofield, Brad ; Gonzalez Berges, Manuel ; Tournier CERN, Jean-Charles: SCADA Statistics Monitoring Using the Elastic Stack (Elasticsearch, Logstash, Kibana); SCADA Statistics Monitoring Using the Elastic Stack (Elasticsearch, Logstash, Kibana). (2017)

[21] Hao, Dan ; Chen, Yinghui ; Tang, Fan ; Qi, Feng ; Hao D., Chen Y Tang F Qi F.: Distributed agent-based performance testing framework on Web Services. In: *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on* (2010), jul, S. 90–94. ISBN 9781424460526

[22] HAW Hamburg: *Dokumentations-Wiki des Departments Informatik der HAW Hamburg*. – URL https://userdoc.informatik.haw-hamburg.de/doku.php. – Zugriffsdatum: 2018-08-03

[23] Hong, Xian J. ; Yang, Hyun S. ; Kim, Young H.: Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application. In: *2018 International Conference on Information and Communication Technology Convergence (ICTC)* (2018), oct, S. 257–259. ISBN 9781538650417

[24] Hoorn, Andre van ; Vögele, Christian ; Schulz, Eike ; Hasselbring, Wilhelm ; Krcmar, Helmut: Automatic Extraction of Probabilistic Workload Specifications for Load Testing Session-Based Application Systems. In: *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, ICST, 2015, S. 139–146. – ISBN 978-1-63190-057-0

[25] Huaji, Zhu ; Huarui, Wu: Research on web application load testing model. In: *2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)* (2017), S. 1175–1178. ISBN 978-1-5090-6414-4

[26] Hüning, Christian ; Adebahr, Mitja ; Thiel-Clemen, Thomas ; Dalski, Jan ; Lenfers, Ulfia ; Grundmann, Lukas ; Dybulla, Janus ; Kiker, Gregory A.:

Modeling & Simulation as a Service with the Massive Multi-Agent System MARS. In: *Spring Simulation Multiconference* (2016), S. to appear. – ISBN 978-1-5108-2315-0

[27] Hüning, Christian ; Ramcke, Eike-Christian ; Sarstedt, Stefan ; Steffens, Ulrike: Business Process simulation with the cloud-based Massive Multi-Agent System MARS. In: *AKWI Tagungsband*. 2015, S. 40–47

[28] Jiang, Zhen M. ; Hassan, Ahmed E.: A Survey on Load Testing of Large-Scale Software Systems. In: *IEEE Transactions on Software Engineering* 41 (2015), nov, Nr. 11, S. 1091–1118. – ISSN 00985589

[29] Jiang, Zhen M. ; Hassan, Ahmed E. ; Hamann, Gilbert ; Flora, Parminder: Automated performance analysis of load tests. In: *IEEE International Conference on Software Maintenance, ICSM*, IEEE, sep 2009, S. 125–134. – ISBN 9781424448289

[30] Ma, Bo ; Chen, Bin ; Bai, Xiaoying ; Huang, Junfei: Design of BDI agent for adaptive performance testing of Web services. In: *Proceedings - International Conference on Quality Software*, IEEE, jul 2010, S. 435–440. – ISBN 9780769541310

[31] Macal, Charles ; North, Michael: Introductory tutorial: Agent-based modeling and simulation. In: *Proceedings - Winter Simulation Conference* Bd. 2015-Janua, IEEE, dec 2014, S. 6–20. – ISBN 9781479974863

[32] Macal, Charles M. ; North, Michael J.: Tutorial on agent-based modeling and simulation. In: *Proceedings of the 37th conference on Winter simulation* (2005), S. 2–15. – ISBN 0-7803-9519-0

[33] Macal, Charles M. ; North, Michael J.: Agent-based modeling and simulation. In: *Proceedings of the 2009 Winter Simulation Conference (WSC)* (2009), S. 86–98. – ISBN 978-1-4244-5770-0

[34] Meier, J.D. ; Farre, Carlos ; Bansode, Prashant ; Barber, Scott ; Rea, Dennis: *Performance Testing Guidance for Web Applications*. 2007. – URL https://docs.microsoft.com/en-us/previous-versions/msp-n-p/bb924375(v{%}3Dpandp.10). – Zugriffsdatum: 2018-10-07

[35] Molyneaux, Ian: *The Art of Application Performance Testing*. 2. O'Reilly Media, Inc., 2009. – 159 S. – ISBN 9780596551056

[36] Naik, Nitin: Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In: *2017 IEEE International Symposium on Systems Engineering, ISSE 2017 - Proceedings*, IEEE, oct 2017, S. 1–7. – ISBN 9781538634035

[37] Prakash, Tarun ; Kakkar, Misha ; Patel, Kritika: Geo-identification of web users through logs using ELK stack. In: *Proceedings of the 2016 6th International Conference - Cloud System and Big Data Engineering, Confluence 2016*, IEEE, jan 2016, S. 606–610. – ISBN 9781467382021

[38] Prometheus: *Prometheus - Monitoring system & time series database*. 2018. – URL https://prometheus.io/. – Zugriffsdatum: 2018-08-03

[39] Ramakrishnan, Raghu: Setting Realistic Think Times in Performance Testing - A Practitioner ' s Approach. In: *Proceedings of the 10th Innovations in Software Engineering Conference on - ISEC '17* (2017), S. 157–164. ISBN 9781450348560

[40] Rohmann, Christian: Elasticsearch, Logstash & Kibana. In: *linux-magazin* (2016), feb

[41] Ruffo, G. ; Schifanella, R. ; Sereno, M. ; Politi, R.: WALTy: A user behavior tailored tool for evaluating web application performance. In: *Proceedings - Third IEEE International Symposium on Network Computing and Applications, NCA 2004*, IEEE, 2004, S. 77–86. – ISBN 0769522424

[42] Schulz, Henning ; Angerstein, Tobias ; Hoorn, André van: Towards Automating Representative Load Testing in Continuous Software Engineering. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. New York, New York, USA : ACM Press, 2018, S. 123–126. – ISBN 978-1-4503-5629-9

[43] Smith, Connie ; G. Williams, Lloyd: More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In: *More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot*, 2003, S. 717–725

[44] Smith, Connie ; Williams, Lloyd: *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002. – 510 S. – ISBN 0201722291

[45] Smith, Connie U. ; Williams, Lloyd G.: New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In: *Computer Measurement Group Conference* (2002), S. 667–674

[46] Stankovic, Nenad: Patterns and tools for performance testing. In: *2006 IEEE International Conference on Electro Information Technology*, IEEE, may 2006, S. 152–157. – URL http://ieeexplore.ieee.org/document/4017687/. – ISBN 078039593X

[47] Syer, Mark D. ; Jiang, Zhen M. ; Nagappan, Meiyappan ; Hassan, Ahmed E. ; Nasser, Mohamed ; Flora, Parminder: Leveraging performance counters and execution logs to diagnose memory-related performance issues. In: *IEEE International Conference on Software Maintenance, ICSM*, IEEE, sep 2013, S. 110–119. – ISBN 1063-6773 VO -

[48] Vögele, Christian ; Hoorn, André van ; Schulz, Eike ; Hasselbring, Wilhelm ; Krcmar, Helmut: WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems. In: *Software and Systems Modeling* 17 (2018), may, Nr. 2, S. 443–477. – ISSN 16191374

[49] Wainer, Gabriel A. ; Vakilzadian, Hamid. ; Society for Modeling and Simulation International, Christopher D.: *Proceedings of the Summer Computer Simulation Conference : SCSC '07 : the Mission Valley Marriott, San Diego, California, July 15-18, 2007*. Society For Computer Simulation, 2007. – 1363 S. – ISBN 1565553160

[50] Wert, Alexander ; Happe, Jens ; Happe, Lucia: Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In: *Proceedings - International Conference on Software Engineering*, IEEE, may 2013, S. 552–561. – ISBN 9781467330763

[51] Wert, Alexander ; Oehler, Marius ; Heger, Christoph ; Farahbod, Roozbeh: Automatic detection of performance anti-patterns in inter-component communications. In: *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures - QoSA '14*, 2014, S. 3–12. – ISBN 9781450325769

[52] Zhu K., Fu J Li Y. ; Zhu, Kunhua ; Fu, Junhui ; Li, Yancui: Research the performance testing and performance improvement strategy in web application. In: *ICETC 2010 - 2010 2nd International Conference on Education Technology and Computer* Bd. 2, IEEE, jun 2010, S. V2–328 –V2–332. – ISBN 9781424463688

Anhang

1 Elasticsearch-Anfrage für den Erhalt aller Handler

```
{
  "query": {
    "range" : {
      "time": {
        "gte": "now-" + options.time,
        "lt":  "now"
       }
    }
  },
  "aggs": {
    "base_handler_filter": {
      "filter": {
        "term": {
          "msg.keyword": "BaseHandlerDuration"
        }
      },
      "aggs": {
        "tags": {
          "terms": {
            "field": "handler.keyword",
            "size": 100
} } } } } }
```

2 Elasticsearch-Anfrage für die Berechnung der Antwortzeiten pro Handler

```
{
  "query": {
    "range": {
      "time": {
        "gte": "now-" + options.time,
        "lt": "now"
```

```
7         }
8       }
9     },
10    "aggs": {
11      "base_handler_filter": {
12        "filter": {
13          "bool": {
14            "must": [
15              {"term": {"handler.keyword": handlerKey}},
16              {"range": {"statusCode": {"lt": 300}}}
17            ]
18          }
19        },
20        "aggs": {
21          "per_time": {
22            "date_histogram": {
23              "field": "time",
24              "interval": options.interval
25            },
26            "aggs": {
27              "percentiles": {
28                "percentiles": {
29                  "field": "duration",
30                  "percents": [50]
31  } } } } } } } }
```

3 Elasticsearch-Anfrage für die Berechnung des Durchsatzes pro Handler

```
1   {
2     "query": {
3       "bool": {
4         "filter": [
5           {
```

```
6             "term": {
7               "handler.keyword": handlerKey
8             }
9           },
10          {
11            "range": {
12              "statusCode": {
13                "lt": "300"
14              }
15            }
16          },
17          {
18            "range": {
19              "time": {
20                "gte": "now-" + options.time,
21                "lt": "now"
22              }
23            }
24          }
25        ]
26      }
27    },
28    "aggs": {
29      "per_time": {
30        "date_histogram": {
31          "field": "time",
32          "interval": options.interval
33  } } } }
```

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Mastertarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Simulation und KPI-basierte Analyse von Geschäftsprozessen in Anwendungslandschaften

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

_____	_____	_____
Ort	Datum	Unterschrift im Original