Einführung von Chaos-Engineering für Business Anwendungen im Cloud-Native Bereich am Beispiel der SAP AI Services

Bachelorarbeit

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Niklas Loeser

Abgabedatum:	29. August 2022
Bearbeitungszeitraum:	06.06.2022 - 28.08.2022
Matrikelnummer, Kurs:	2972280, TINF19B1
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Betreuer der Ausbildungsfirma:	Sergey Smirnov
Gutachter der Dualen Hochschule:	Willhelm Stoll

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema:

> *Einführung von Chaos-Engineering für Business Anwendungen im Cloud-Native Bereich am Beispiel der SAP AI Services*

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 28. August 2022

Loeser, Niklas

Sperrvermerk

Die nachfolgende Arbeit enthält vertrauliche Daten der:

SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf, Deutschland

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung vom Dualen Partner vorliegt.

Abstract

- English -

Chaos Engineering builds confidence in the stability of a system by causing targeted system faults. To reduce the system faults induced risks, Chaos Engineering is implemented on a non-production system. Furthermore the Chaos Engineering should build confidence in the stability of the production system. The chaos experiments running on the non-production system need to provide helpful information on the stability of the production system. Additionally a cost minimization must be undertaken.

This thesis suggests a concept that enables chaos engineering on non-production systems that helps improving production systems. To build confidence in the production system, the concept first creates a comparability between the production and non-production system. It first identifies that a lot less user interactions occur on the non-production system. The missing user interactions are created by simulating user behavior by reusing existing tests.
The concept achieves a cost minimization by automating Chaos Engineering and reusing existing components. Thus personnel and system costs are saved. The concept enables fully automated experiments, which are automatically triggered, simulate users during the experiment, introduce chaos into a system, collect the results, and lastly bring the system to a steady state.

Abstract

- Deutsch -

Chaos-Engineering baut Vertrauen in die Stabilität eines Systems auf, indem gezielt Störungen in dem System verursacht werden. Um die durch die Störungen entstandenen Risiken zu senken, wird das Chaos-Engineering auf einem nicht-Produktivsystem eingeführt. Gleichzeitig soll durch das Chaos-Engineering Vertrauen in das Produktivsystem aufgebaut werden. Es müssen Rückschlüsse von den Chaos-Experimenten des nicht-Produktivsystems auf das Produktivsystem geschlossen werden. Zuletzt soll eine Kostenminimierung vorgenommen werden.

Die Arbeit führt ein neues Konzept ein, das Chaos-Engineering auf einem nicht-Produktivsystem ermöglicht, welches das Produktivsystem verbessert. Um Vertrauen in das Produktivsystem aufbauen zu können, schafft das Konzept eine Vergleichbarkeit zwischen dem nicht-Produktivsystem und dem Produktivsystem. Hierfür wird identifiziert, dass auf dem nicht-Produktivsystem deutlich weniger Nutzerinteraktionen stattfinden. Die fehlenden Nutzerinteraktionen werden auf dem nicht-Produktivsystem über vorhandene Tests simuliert.
Die Kostenminimierung wird erzielt, indem alle Schritte des Chaos-Engineerings automatisiert werden und vorhandene Komponenten wiederverwendet werden. Dadurch werden Personalkosten und Systemkosten eingespart. Es werden vollkommen automatisierte Chaos-Experimente ermöglicht, die automatisch gestartet werden, während des Experiments Nutzer auf dem nicht-Produktivsystem simulieren, das System gezielt stören, zuletzt alle Ergebnisse sammeln und das System in einen Normalzustand versetzen.

Inhaltsverzeichnis

Abkürzungsverzeichnis **VII**

Abbildungsverzeichnis **IX**

Tabellenverzeichnis **X**

Quellcodeverzeichnis **XI**

1 Einleitung **1**
1.1 Einführung 1
1.2 Motivation 2
1.3 Struktur 3

2 Stand der Technik **4**
2.1 Cloud 4
2.2 Systemtheorie 11
2.3 Hochverfügbarkeit 14
2.4 Tests 18

3 Konzeptionierung **28**
3.1 Anforderungen 29
3.2 Bewertungskriterien 29
3.3 Konzept 30
3.4 Varianten 37

4 Implementierung **39**
4.1 SAP Artificial Intelligence (AI) Dienste 39
4.2 Identifizierung der Komponenten 41
4.3 Architektur 46
4.4 Funktionale Tests 49
4.5 Jenkins 58
4.6 Experimentstatusbenachrichtigungen 61
4.7 Screenshots 64
4.8 Lasttests 68

5 Ergebnisse **74**

6 Schluss **76**
6.1 Zusammenfassung 76

6.2 Ausblick 77

Literaturverzeichnis **XII**

A Cloud **XVII**
A.1 On-Premises XVII
A.2 Monolithische Architektur XVIII

B Systemtheorie **XX**
B.1 Verteiltes System XX
B.2 Komplexes System XXII

C Abbildungen **XXVI**

Abkürzungsverzeichnis

AI	Artificial Intelligence
BER	Business Entity Recognition
BTP	Business Technology Platform
CaaS	Chaos as a Server
CF	Cloud Foundry
CPU	Central Processing Unit
CRUD	Create Read Update Delete
DNS	Domain Name Server
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
ID	Identifier
JSON	JavaScript Object Notation
K8s	Kubernetes
MTTD	Mean Time To Detect
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
PaaS	Platform as a Service
PDF	Portable Document Format
PR	Personalized Recommendation
SLA	Service Level Agreement
SLO	Service Level Objective
UI	User Interface

UID Unique Identifier

URL Uniform Resource Locator

VPN Virtual Private Network

Abbildungsverzeichnis

2.1 Skizze des Cloud-Anwendungsmodells 5
2.2 Skizze des Infrastructure as a Service (IaaS)-Modells 6
2.3 Skizze Containerisierung 7
2.4 Test-ähnliches Chaos-Engineering 24

3.1 Komponenten des Chaos-Engineering 33
3.2 Komponenten des vollständigen Konzepts 36

4.1 Komponenten des vollständigen Konzepts 45
4.2 Komponenten gruppiert anhand ihres Systems 48
4.3 Automatisierung von Jenkins 55
4.4 Slack-Nachricht eines erfolgreichen Experiments 63
4.5 Informationsfluss der Experimentergebnisse 69

A.1 Skizze des On-Premises-Modells XVIII
A.2 Skizze eines Monoliths XIX

C.1 Slack-Nachricht eines erfolglosen Experiments XXVI
C.2 Slack-Nachricht eines abgebrochenen Experiments XXVI

Tabellenverzeichnis

2.1 Aspekte der Transparenz 12
2.2 Eigenschaften eines allgemeinen komplexen Systems 13
2.3 Verfügbarkeitsklassen[32] 15

4.1 Zuweisung der praktischen Komponenten zu den konzeptuellen Komponenten 41

5.1 Ergebnis des Bewertungskriteriums Zeiteffizienz 74

Quellcodeverzeichnis

4.1 Ausführen einer Aktion im Hintergrund 52
4.2 ChaosToolkit Aktion trigger_test 54
4.3 Definition eines Pipelinestarts in einem Experiment 57
4.4 Kapselung der `sh`-Methode 61
4.5 Zusammenbau einer Slack-Benachrichtigung 63
4.6 Definition einer Slack-Benachrichtigung in einem Experiment 64

1 Einleitung

1.1 Einführung

Seit 2014 hat ein Wandel von einer monolithischen Architektur zu einer Microservice-Architektur stattgefunden[1]. Anstelle eines herkömmlichen Monoliths werden in einer Microservice-Architektur Anwendungen einer funktionalen Dekomposition unterzogen und anhand ihrer Funktionalität in viele einzelne Microservices unterteilt. Durch die funktionale Dekomposition entsteht eine Schar von Microservices, welche voneinander anhand ihrer Business-Logik abhängen. Zudem hängen die Microservices von bereits existierenden Microservices ab, da sich Microservices wiederverwenden lassen[2]. Aufgrund der vielen voneinander abhängenden Microservices entsteht ein komplexes System, in welchem scheinbar kleine Änderungen große Auswirkungen haben können[3].

Der Wandel zum Cloud-Computing-Modell brachte Verantwortung für die Software-Hersteller. In dem On-Premises Modell verwalten Kunden die ihnen verkaufte Software. Im Gegensatz dazu wird den Kunden im Cloud-Computing-Modell die Software über die Cloud bereitgestellt, jedoch von dem Software-Hersteller selbst verwaltet. Die Software läuft dabei oftmals auf fremden Cloud-Plattformen. Dennoch muss sichergestellt werden, dass die Software über ihren Lebenszyklus hinweg fehlerfrei bleibt und für die Kunden verfügbar ist[1].

Die SAP ist eines der Unternehmen, in welchen heutzutage Anwendungen nach dem Cloud-Computing-Modell entwickelt werden. Zu den Produkten zählen die SAP AI Services. Die SAP AI Services stellen Dienste zur Verfügung, welche auf künstlicher Intelligenz basieren. Ein Dienst erkennt z. B. über künstliche Intelligenz Text in Bildern. Für die Dienste muss sichergestellt werden, dass sie fehlerfrei bleiben und für die Kunden verfügbar sind. Die Verfügbarkeit und Korrektheit der Dienste ist aus einer unternehmerischen Sicht wichtig, da sie vertraglich über Service Level Agreements (SLAs) vereinbart sind und die in den SLAs definierten Service Level Objectives (SLOs) eingehalten werden müssen. Die Verfügbarkeit und Korrektheit haben zudem einen direkten Einfluss auf die Befriedigung

der Kunden. Wenn das Produkt nicht ausreichend verfügbar ist, dann wechseln Kunden zur Konkurrenz.

1.2 Motivation

Produkte, die dem Cloud-Computing-Modell folgen, müssen für den Kunden verfügbar sein. Ist die Verfügbarkeit nicht gegeben, dann folgt ein Verlust für das Unternehmen. Problem sind die unberechenbaren Folgen von kleinen Änderungen in einem komplexen System, da eine scheinbar harmlose Aktion einen Ausfall des kompletten Systems hervorrufen kann. Es kommt hinzu, dass es sich zudem um ein verteiltes System handelt. Die Dienste laufen auf Systemen, die aus mehreren Rechnern bestehen, die an beliebigen Orten stehen können. Das verteilte System ist eine weitere Fehlerquelle, welches Probleme wie eine erhöhte Latenz einführt. Die Probleme des verteilten Systems gepaart mit der Unberechenbarkeit des komplexen Systems können schlagartig zu einer Reihe an Fehlern führen, die das komplette System und dadurch den Kunden beeinträchtigen[3, 4].

Eine helfende Methodik ist hier Chaos-Engineering. Chaos-Engineering verursacht über Experimente in einem System kontrolliert Systemstörungen, wie z. B. eine erhöhte Latenz der Dienste. Sollten die Dienste robust gegen Störungen sein und korrekt im System installiert sein, dann sind die Dienste weiterhin verfügbar und das Experiment ist gelungen. Sollten jedoch die gestörten Dienste ausfallen und z. B. weitere unberechenbare Fehler verursachen, dann ist das Experiment gescheitert. Der Nutzer soll die Auswirkungen des Experiments nicht spüren, da das Chaos gezielt in einem kleinen Teil des Systems verursacht wird. Da die verursachten Störungen natürlich eintretende Störungen präventiv modellieren, wie z. B. den Ausfall eines Rechenzentrums, werden Systemfehler frühzeitig erkannt und verbannt[5].

Nun soll Chaos-Engineering in der SAP für die AI Dienste eingeführt werden, um die SLOs zu erfüllen und dadurch die SLAs zu wahren. Für die Umsetzung sollen folgende drei Ziele berücksichtigt werden. Das gezielte Stören eines Systems über Chaos-Engineering bringt hohe Risiken mit sich. Zuerst sollen die Risiken minimiert werden. Als Zweites sollen die Kosten des Chaos-Engineerings minimiert werden. Zuletzt soll durch das Chaos-Engineering die Qualität des Produktivsystems sichergestellt werden.

Um die Risiken des Chaos-Engineerings zu minimieren, wird ein Kompromiss eingegangen. Das Chaos-Engineering wird auf einem nicht-Produktivsystem umgesetzt. Nachteil des nicht-Produktivsystems ist die bedingte Vergleichbarkeit zum Produktivsystem[6]. Da die Vergleichbarkeit nur bedingt gegeben ist, kann die Qualität des Produktivsystems über Chaos-Experimente auf einem nicht-Produktivsystem auch nur bedingt sichergestellt werden. Es gibt somit einen Konflikt zwischen dem ersten und dritten Ziel, der Risikominimierung und Qualitätssicherung des Produktivsystems.

Um die Kosten des Chaos-Engineerings zu minimieren, werden zwei Aktionen vorgenommen. Es werden alle Schritte des Chaos-Engineerings automatisiert, sodass der manuelle Aufwand reduziert wird und dadurch Personalkosten eingespart werden. Als Zweites werden durch eine Wiederverwendung vorhandener Komponenten und Systeme Systemkosten eingespart.

Sodass die auf dem nicht-Produktivsystem laufenden Chaos-Experimente die Qualität des Produktivsystems sicherstellen, wird eine Vergleichbarkeit zwischen dem nicht-Produktivsystem und Produktivsystem hergestellt. Das nicht-Produktivsystem wird dem Produktivsystem angenähert, indem Lasttests und funktionale Tests wiederverwendet werden, um Nutzerinteraktionen zu simulieren.

Die umzusetzende Methodik soll somit die drei Ziele erfüllen und dabei bei Konflikten zwischen den Zielen Abwägungen treffen.

1.3 Struktur

Die Arbeit hat eine folgende Struktur: Nach der Einleitung werden die für die Arbeit benötigten Grundlagen genannt. In den Grundlagen wird ein Fokus auf die Verfügbarkeitsproblematik und Chaos-Engineering geworfen. Nachfolgend wird die aus den Anforderungen folgende Methodik einführt, inklusive Diskussionen zu Varianten und Parametern. In der folgenden Implementierung wird das Konzept praktisch an den SAP AI Services angewandt. Die daraus resultierenden Ergebnisse werden in dem nächsten Kapitel dargelegt und bewertet. Im Schluss wird zuletzt ein Blick auf die verrichtete Arbeit geworfen und ein Ausblick mit zukünftigen und aus der Arbeit folgenden Projekten gegeben.

2 Stand der Technik

In diesem Kapitel sollen die für die Arbeit nötigen Grundlagen eingeführt werden, sowie die darauf aufbauenden State-of-the-Art Methodiken.

2.1 Cloud

2.1.1 Cloud-Anwendungsmodell

Heutzutage wird für viele Produkte das Cloud-Anwendungsmodell verwendet[7]:

> Das Cloud-Anwendungsmodell verschiebt die Verantwortung der Software-Verwaltung an den Software-Hersteller und die Verwaltung der Hardware an die Betreiber eines Infrastructure as a Service (IaaS)-Dienstes[8]. Die Software wird von dem Software-Hersteller auf der Hardware eines IaaS-Betreibers installiert und verwaltet. Software-Updates werden im Rahmen der Verwaltung auch auf die Cloud aufgespielt, sodass der Kunde automatisch eine neuere Version nutzen kann.
>
> Im Besonderen muss der Software-Hersteller die Software warten. Der Software-Hersteller muss auftretende Probleme beheben.
>
> Der Software-Hersteller standardisiert die Software und verkauft an alle Kunden dieselbe standardisierte Version[8]. Die Software wird vom Kunden im Nachhinein angepasst.

Hierdurch entstehen folgende Vor- und Nachteile:

> Sowohl der Kunde, als auch der Software-Hersteller müssen die Software nicht verwalten. Da die Hardware über einen IaaS-Betreiber erlangt wird, müssen nur die verbrauchten Ressourcen bezahlt werden. Bei effizienten Anwendungen werden dadurch insgesamt Kosten gespart, als wenn die Anwendungen auf eigener Hardware laufen[9].

Durch den IaaS-Betreiber wird eine Skalierbarkeit erzielt[10]. Zudem kann die bereitgestellte Hardware genau dem Bedarf angepasst werden[10].

Vorteil für den Kunden ist die geringe Verantwortung, da der Software-Hersteller die Verantwortung der Software-Verwaltung hat. Gleichzeitig ist das ein Nachteil für den Software-Hersteller. Wenn der Software-Hersteller seiner Verantwortung nicht gerecht werden kann, dann folgen vertragliche Konsequenzen.

Indem der Software-Hersteller die Software standardisiert ausliefert, kann die Software sehr einfach an viele Kunden ausgeliefert werden.

In Abbildung 2.1 wird das Cloud-Anwendungsmodell skizzenhaft veranschaulicht.

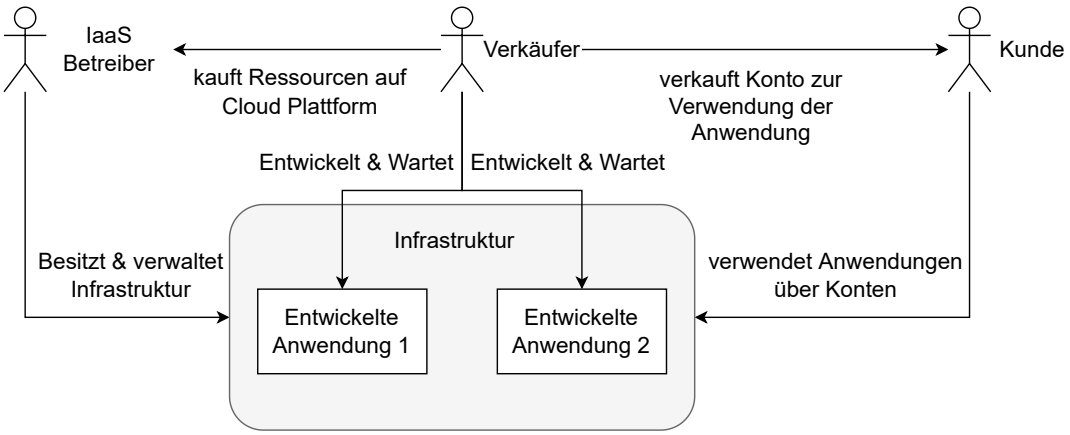

Abbildung 2.1: Skizze des Cloud-Anwendungsmodells

Oftmals wird heutzutage das Cloud-Anwendungsmodell gewählt, weil das Modell für die Beteiligten Flexibilität schafft und eine einfache und hohe Skalierung der Anwendung ermöglicht.

2.1.2 Infrastructure as a Service (IaaS)

Für das Cloud-Anwendungsmodell wird Hardware benötigt, auf welcher die Anwendungen laufen. Die Hardware wird von einem IaaS-Betreiber als Dienst bereitgestellt. Die komplette Verantwortung der Verwaltung der Hardware liegt bei dem Betreiber[11].

Um die bereitgestellte Hardware zu verwenden, werden von den IaaS-Betreibern erstellte Abstraktionsschichten verwendet. Die Abstraktionsschichten vereinfachen und

vereinheitlichen den Umgang mit der Hardware. Der IaaS-Betreiber kann also viele heterogene Hardware in Betrieb haben und die Ressourcen dann homogen über die Abstraktionsschicht bereitstellen[12].

Dieselbe Abstraktionsschicht kann auch instrumentalisiert werden, um die abstrahierten Komponenten zu einem gewissen Grad zu steuern. Es könnte z. B. der Ausfall eines Rechners simuliert werden, indem die Netzwerkzugriffsliste zeitweise angepasst wird. Je nach Funktionalität der Abstraktionsschicht könnte auch Last auf einem Rechner simuliert werden. Die Abstraktionsschicht kann somit für diverse Tests verwendet werden[1]. In Abbildung 2.2 wird das IaaS-Modell skizzenhaft veranschaulicht.

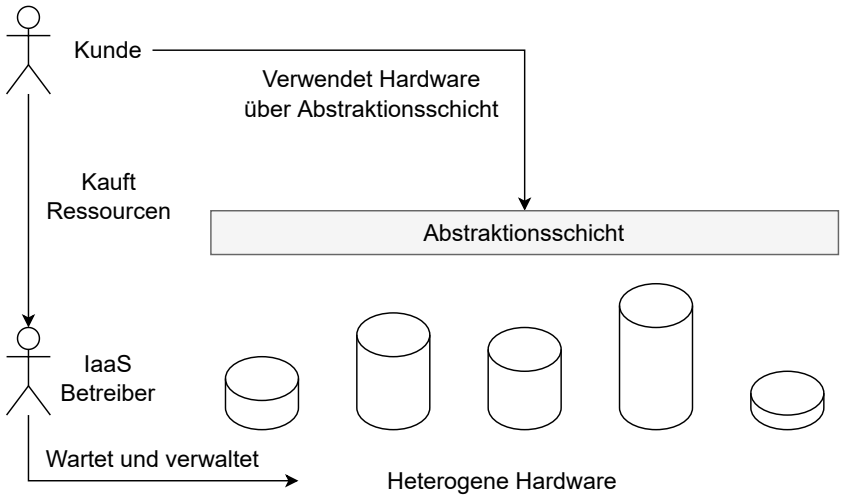

Abbildung 2.2: Skizze des IaaS-Modells

Über IaaS gelangt ein Unternehmen an die für das Cloud-Anwendungsmodell benötigten Ressourcen, es reicht jedoch noch nicht, um die Anwendungen auf der Hardware auszuführen. Hierfür werden weitere Abstraktionsschichten benötigt, die auf IaaS aufbauen. Für das Ausführen diverser Dienste wird auf Containerisierung gesetzt. Über Containerisierung wird jeder Dienst in einen eigenen Container gekapselt, welcher alle benötigten Bibliotheken enthält. Die Container laufen nebeneinander in einer Containerlaufzeit, welche auf der Hardware aufsetzt. Die Containerlaufzeit erlaubt das Ausführen beliebiger zur Laufzeit kompatiblen Container[13]. Da die Container in beliebigen Systemen ausgeführt werden, in welchen die Containerlaufzeit installiert ist, sind die Container portabel und

[1] Amazons Cloud-Produkte können über den AWS SSM instrumentalisiert werden

einfach zu verteilen[13]. Die Containerlaufzeit stellt für jeden Container ein Kernel bereit. Jeder Container enthält nur ein Dateisystem, in welchem die Abhängigkeiten liegen.

Containerd ist eine heutzutage weit verbreitete Containerlaufzeit[14] und dient als Basis für weitere Werkzeuge, die diese Abstraktionsschicht benötigen. Eins der Werkzeuge wird in Kubernetes (K8s) genannt. In Abbildung 2.3 wird das Prinzip der Containerisierung skizzenhaft veranschaulicht.

Auf einem Produktivsystem laufen oftmals viele Container nebeneinander, die Funktionen wie Authentifizierung, Routing, Load Balancing, Datenspeicherung und noch viele weitere Funktionen abnehmen. Wenn eines der Container abstürzen sollte, dann wird der Container nicht automatisch neu gestartet, da die Container nicht verwaltet werden. Zur Verwaltung der Container wird ein Containerverwaltungswerkzeug wie K8s auf der Hardware aufgesetzt, welches diese Rolle erfüllt.

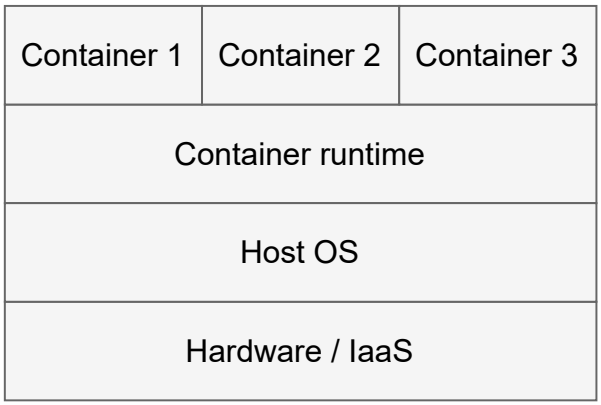

Abbildung 2.3: Skizze Containerisierung

2.1.3 Kubernetes (K8s)

K8s ist eine weit verbreitete[15] und portable Open-Source-Containerverwaltungssoftware[16]. K8s baut somit auf einer unterstützten Containerlaufzeit auf und stellt weitere Funktionalität bereit, welche im letzten Paragrafen erwähnt wurde. Eine der Funktionalitäten ist die Selbstreparatur-Fähigkeit. In K8s können mehrere Instanzen einer containerisierten Anwendung gestartet werden. Sollte eine der Instanzen abstürzen, dann startet K8s den Container automatisch neu, beziehungsweise ersetzt den Container durch einen neuen[16]. Das Werkzeug beinhaltet zudem Load-Balancing-Funkionalität, wodurch eine

hohe Skalierbarkeit erreicht wird, da der eingehende Nutzerverkehr auf mehrere Container derselben Anwendung verteilt wird[16]. Die Skalierbarkeit wird durch die deklarative Definition des Systemzustands ermöglicht, da das System nicht manuell in einen erwünschten Systemzustand gebracht werden muss, sondern K8s das System in den erwünschten Zustand bringt. Im Falle eines Fehlers befindet sich das System nicht mehr im erwünschten Zustand, wonach K8s das System wieder in den erwünschten Zustand befördert. Die Methode hat den Vorteil, dass der erwünschte Zustand dynamisch verändert werden kann, um z. B. die Anzahl der laufenden Instanzen einer Anwendung zu erhöhen.

K8s ist für ein verteiltes System konzipiert und stellt folgende Abstraktionen bereit:

Die erste Abstraktion ist die *Node*. Die *Node* stellt einen virtuellen oder physischen Rechner da[17]. Ein Beispiel hierfür wäre ein virtueller Rechner eines IaaS-Betreibers. Die *Nodes* stehen an einem beliebigen Ort. K8s unterscheidet zwischen kontrollierenden und arbeitenden *Nodes*. Die kontrollierenden *Nodes* verwalten das System und enthalten die Dienste, welche das System in den gewünschten Zustand bringen. Die arbeitenden *Nodes* führen die regulären Container aus.

Die Container werden nicht direkt auf die *Nodes* verteilt. Die Container werden in *Pods* gekapselt. Ein *Pod* ist das kleinste verwaltbare Objekt in einem K8s-System und stellt eine Gruppierung von Containern dar, welche gemeinsame Speicher- und Netzwerkressourcen haben[18]. Die *Pods* werden als volatil betrachtet und können bei Bedarf automatisch gelöscht und erstellt werden. Durch den *Pod* werden Containerlaufzeiten abstrahiert, da in einem *Pod* Container einer beliebigen unterstützten Containerlaufzeit laufen. Die hierfür vorhergesehene Laufzeit ist jedoch Containerd[18]. Die *Pods* werden von K8s arbeitenden *Nodes* zugewiesen und auf den *Nodes* ausgeführt.

Eine weitere Abstraktion sind *Namespaces*. Namensräume erlauben das Trennen von *Pods* unterschiedlicher Anwendungen[19]. Das Abtrennen der Anwendungen ist besonders auf Systemen wichtig, auf welchen viele unterschiedliche Anwendungen laufen, von welchen sichergestellt werden muss, dass die Anwendungen nicht untereinander interferieren. K8s erlaubt das Definieren von Rollen, sowie Berechtigungen für die Rollen, wodurch der Zugriff von *Pods* eines Namensraumes zu anderen Namensräumen kontrolliert wird.

Die drei Abstraktionen sind nur manche der vielen unterschiedlichen Abstraktionen und Funktionalitäten, die K8s bereitstellt. Für die Arbeit sind die drei Abstraktionen für die Implementierung und Kapselung der Chaos-Experimente von Relevanz, die anderen Funktionalitäten werden bei Bedarf erwähnt.

Bei Entwicklung von Anwendungen nach dem Cloud-Anwendungsmodell wird oftmals K8s zum Ausführen und Verwalten der Anwendungen gewählt[15]. K8s baut auf dem IaaS-Modell auf und ermöglicht eine feine Kontrolle über die laufenden Anwendungen. Dabei werden viele Funktionalitäten bereitstellt und das einfache Konfigurieren der Anwendungen ermöglicht.

2.1.4 Platform as a Service (PaaS)

Container-basierte Technologien wie K8s erlauben eine genaue Kontrolle der Anwendungen und der Infrastruktur, auf welcher die Anwendungen laufen. Jedoch muss für das Ausführen einer Anwendung in einer Container-basierten Umgebung aus der Anwendung zuerst ein Container erstellt werden. Der Container besteht aus einem minimalen Betriebssystem, den benötigten Abhängigkeiten und der Anwendung selbst[13].

PaaS bietet in Kontrast zu Containerisierung eine höhere Abstraktionsschicht. PaaS stellt zusätzlich eine Reihe von Laufzeiten und für Anwendungen benötigte Abhängigkeiten zur Verfügung. Dadurch wird die Anwendung direkt auf der Plattform ausgeführt, ohne dass zuvor eine dafür notwendige Umgebung installiert werden muss. Als Folge sinkt der Verwaltungsaufwand, jedoch wird gleichzeitig aber auch die Kontrolle über das System verringert[20].

Eine heutzutage weit verbreitete PaaS-Software ist Cloud Foundry[21]. Cloud Foundry ist Open-Source und unterstützt viele weitverbreitete Programmiersprachen und weitverbreitete Abhängigkeiten. Ein weiterer Vorteil von Cloud Foundry ist seine Erweiterbarkeit. Es kann Unterstützung für weitere Programmiersprachen und Abhängigkeiten hinzugefügt werden[22]. Da Cloud Foundry eine hochfunktionale und gut unterstützte PaaS-Software bietet, wird Cloud Foundry von Unternehmen verwendet, um ihren Kunden eine Plattform zu bieten, die in ihre eigenen Cloud-Produkte integriert ist. Ein Beispiel ist die SAP Business Technology Platform (BTP), welche als Teil der Cloud-Produkte Cloud Foundry verwendet und den Kunden bereitstellt[23].

Da PaaS-Software eine Abstraktionsschicht höher als K8s ist, kann diese auf K8s aufbauen. Andernfalls kann auf der über IaaS bereitgestellten Infrastruktur aufgebaut werden. Ein Beispiel für ein auf K8s aufbauendes Cloud Foundry ist Korifi[24].

2.1.5 Microservices

In dem heutzutage weit verbreiteten Cloud-Anwendungsmodell wird oftmals eine Microservice-Architektur für den Aufbau der Anwendungen gewählt[25]. Die Gründe für die Architektur und die Folgen daraus sollen hier besprochen werden.

Microservices entstehen, indem eine Anwendung einer funktionalen Dekomposition unterzogen wird und dadurch in einzelne eigenständige Dienste aufgeteilt wird[2].

Da die Dienste voneinander unabhängig sind, ist eine Microservice-Architektur sehr flexibel[26]. Die Dienste können von komplett unterschiedlichen Teams entwickelt werden. Da jeder Microservice in einem eigenen Prozess läuft und eine eigene Codebasis hat, können die Microservices in unterschiedlichen Programmiersprachen geschrieben sein. Hierdurch wird die Flexibilität erhöhrt.

Die Skalierbarkeit der auf die Microservices verteilten Anwendung ist sehr hoch, da sich die Microservices unabhängig voneinander skalieren lassen[27]. Um dem Nutzer ein reibungsloses Erlebnis bei Verwendung der Anwendung zu bieten, können stark beanspruchte Dienste automatisch hochskaliert werden, sodass einzelne Dienste niemals unter einer zu hohen Last stehen. Verwenden wenige Nutzer das System, so könnten wenig beanspruchte Dienste automatisch herunterskaliert werden[28]. Die Microservice-Architektur erlaubt es, nicht viel mehr Ressourcen zu verbrauchen, als für die Nutzer benötigt werden. Dies ist in Bezug auf die Cloud-Plattform praktisch, da dadurch nur soviel Kosten für den Betrieb der Dienste entstehen, wie die Nutzer verbrauchen. Es werden keine Ressourcen verschwendet.

Ein weiterer Vorteil ist die gute Wiederverwendbarkeit der Dienste. Viele der Dienste brauchen eine Form der Authentifizierung, sodass nur Kunden Zugang zum System haben. Hierfür könnte ein Authentifizierungsdienst geschrieben werden. Alle Dienste, die eine Authentifizierung benötigen, verwenden dann den Authentifizierungsdienst wieder[27].

Die hohe Flexibilität, Skalierbarkeit und Wiederverwendbarkeit sind eindeutige Vorteile, weshalb eine Microservice-Architektur gewählt wird, jedoch führen die Vorteile zu einem Problem. Durch die vielen Instanzen der vielen heterogenen entstandenen Microservices, besteht das System aus einer fast unüberschaubaren Schar an Microservices[29]. Es folgt eine komplexe und aufwendige Verwaltung des Systems. Da die Dienste in ihren eigenen Prozessen laufen, müssen die Dienste ihre Funktionalität über das Netzwerk bereitstellen und über das Netzwerk kommunizieren. Die Kommunikation über das Netzwerk, gepaart mit der Schar aus Diensten führt zu einem komplexen System und zu allen daraus folgenden Problematiken, die für Chaos-Engineering relevant sind.

Um heutzutage flexible und skalierbare Software zu schreiben, wird auf PaaS-Plattformen oder Containerisierung zurückgegriffen. Hierdurch kann Konfigurationsaufwand gespart werden und die Arbeit auf die Implementierung der Anwendung fokussiert werden. Falls etwas mehr Kontrolle gebraucht wird, dann eignen sich Werkzeuge wie K8s mehr, andernfalls werden Werkzeuge wie Cloud Foundry verwendet. Dazu eignet sich die Microservice-Architektur. Die Kombination aus der Microservice-Architektur und PaaS oder Containerisierung führt zu einem komplexen und verteilten System. Das komplexe und verteilte System und alle daraus entstehenden Folgen werden in 2.2 genauer erklärt.

2.2 Systemtheorie

In Sektion 2.1 wurde die Architektur heutiger Cloud-nativer Anwendungen erläutert. In dieser Sektion soll auf die dort identifizierten Systemtypen eingegangen werden. Für jeden Systemtyp sollen die Eigenschaften und die daraus entstehenden Problematiken erläutert werden. Zuerst wird das verteilte System betrachtet, danach folgt das komplexe System.

2.2.1 Verteiltes System

Ein verteiltes System lässt sich nach Tanenbaum folgenderweise definieren:

> A distributed system is a collection of independant computers that appears to its users as a single coherent system.[30]

Transparenz
Zugriff
Ort
Migration
Umzug
Vervielfältigung
Nebenläufigkeit
Versagen

Tabelle 2.1: Aspekte der Transparenz

Ein verteiltes System zeichnet sich nach Tanenbaum durch den Verbund mehrerer unabhängiger Rechner aus, welche für einen Nutzer wie ein zusammenhängendes System wirken[30]. Seine Definition lässt mehrere Punkte offen. Zuerst ist zu bemerken, dass die Anzahl der Rechner nicht definiert wurde. Ein verteiltes System kann somit aus sehr vielen Rechnern bestehen, z. B. 1000 Rechnern. Je mehr Rechner Bestandteil des verteilten Systems sind, desto komplexer ist die Verwaltung des Systems für einen Menschen, aber auch für ein Programm[30]. Hinzu kommt, dass die Rechner unabhängig sind[30]. Die Rechner müssen nicht alle baugleich sein, das System kann aus beliebigen unterschiedlichen Rechnern bestehen. Dadurch wird der Verwaltungsaufwand eines verteilten Systems erhöht. Die Art der Nutzer wurde zudem nicht definiert, es kann sich somit um Menschen, als auch um Programme handeln[30].

Ein verteiltes System, welches für den Nutzer als ein einheitliches System erscheint, gilt als transparent[30]. Die Eigenschaft der Transparenz wird in mehrere Aspekte aufgeteilt. Die Aspekte können Tabelle 2.1 entnommen werden.

Ein verteiltes System versteckt viel Komplexität vor dem Nutzer[30]. Trotz der vielen Eigenschaften soll das System robust gegenüber Fehlern sein. Doch was für Fehler können in einem verteilten System auftreten?

Grundsätzlich gibt es Latenz basierte Fehler, sowie fehlerhafte Serverantworten[4]. Die erste Art an Fehlern, die Latenz basierte Fehler[4], sind besonders auf die Ortstransparenz zurückzuführen. Die Rechner eines verteilten Systems können an anderen Enden der Erde physisch stehen. Dadurch hat das System eine durch die Distanz bedingte Latenz. Latenz kann auch durch eine hohe Last eines Rechners auftreten, wenn dieser z. B. viele Nutzeranfragen verarbeiten muss. Wenn ein Rechner stark belastet ist, dann verarbeitet er Nutzeranfragen langsam und antwortet auf die Anfragen verzögert. Das Ausfallen eines

Eigenschaften
Zahlreiche Interaktionen
Unordnung und Vielfalt
Rückmeldung
Extrinsisch
Spontane Ordnung
Nicht-Linearität
Robustheit
Modularität
Zeitlicher Verlauf
Anpassungsfähigkeit

Tabelle 2.2: Eigenschaften eines allgemeinen komplexen Systems

Rechners ist ein weiterer Fehler[30], welcher als Fall einer unendlichen Latenz verstanden wird, da die Nachricht den Rechner nie erreicht.

Die zweite Fehlerart besteht darin, dass der Server fehlerhafte Antworten versendet[4]. Ein Fehler ist das zufällige Versenden von zufälligen Nachrichten. Weitere Fehler beinhalten das Senden von Nachrichten mit falschem Rückgabewert. Dies kann z. B. ausgelöst werden, wenn der Rechner durch fehlerhaften Code in einen invaliden Zustand befördert wird[30].

In einem verteilten System treten unterschiedliche Fehler auf, welche durch unterschiedliche Ansätze erkannt werden. Der erste Ansatz betrachtet das verteilte System aus Sicht eines Nutzers. Es werden Nutzer-ähnliche Anfragen an das System gestellt und die Antworten beobachtet. Der Bruch der Fehlertransparenz wird anhand der Antworten erkannt. Der zweite Ansatz betrachtet das System von innen. Dafür müssen entsprechende Berechtigungen vorliegen, sodass das Innere betrachtet werden kann. Durch das Auswerten diverser Systemmetriken wird ein genaueres Bild des Systems erlangt, wodurch Fehler erkannt werden, selbst wenn die Fehlertransparenz nicht gebrochen wird.

2.2.2 Komplexes System

Ein komplexes System ist ein System, in welchem kleine Handlungen große und unberechenbare Auswirkungen haben können[3]. Das komplexe System lässt sich über seine Eigenschaften charakterisieren. Die von Ladyman und Wiesner [3] definierten Eigenschaften sind in Tabelle 2.2 zu sehen.

Nicht nur ein komplexes System hat die in der Tabelle genannten Eigenschaften. Auch ein heutiges verteiltes und containerisiertes System, welches nach der Microservices-Architektur entworfen wurde, hat die genannten Eigenschaften. Das verteilte und containerisierte System, welches nach der Microservices-Architektur entworfen wurde, wird fortan als Cloud-natives System bezeichnet.

Ein Cloud-natives System hat fast alle Eigenschaften eines allgemeinen komplexen Systems. Dem Cloud-nativen System mangelt es lediglich an einer völlig dezentralen Steuerung, anstelle der bisher vorhandenen verteilten und zentralen Steuerung. Dennoch ist das Cloud-native System als ein komplexes System zu bezeichnen.

Die viele komplexe Kommunikation zwischen Komponenten sowie Rückmeldung zwischen den zustandsorientierten Komponenten führt zu unvorhersehbaren Folgen[3]. Trotz der Folgen wird die Struktur und das Verhalten des Systems beibehalten. Aber nicht alle Folgen sind von den Verwaltern des Systems erwünscht. Ein Beispiel wäre der Ausfall mehrerer Komponenten, wodurch ein Teil einer Anwendung nicht mehr für Kunden verfügbar ist. Diese Folge ist ausdrücklich unerwünscht, ist jedoch im Rahmen der natürlichen Folgen eines komplexen Systems.

Für die bleibende Verfügbarkeit Cloud-nativer Anwendungen müssen unerwünschte und unvorhersehbare Folgen eines Cloud-nativen Systems entfernt werden. Auslöser für die Folgen können über Tests entdeckt werden. Die Eignung diverser Tests wird in Sektion 2.4 genauer betrachtet. Zuerst wird das unternehmerische Ziel betrachtet, die Verfügbarkeit Cloud-nativer Anwendungen.

2.3 Hochverfügbarkeit

In dieser Sektion soll die Hochverfügbarkeit im Kontext Cloud-nativer Systeme definiert werden. Zudem soll betrachtet werden, ob eine Hochverfügbarkeit überhaupt notwendig ist. Um die Verfügbarkeit zu messen, werden mehrere Metriken eingeführt. Folgend wird auf Verfügbarkeitszonen eingegangen. Zuletzt wird auf verwandte Eigenschaften, wie die Robustheit eingegangen, sodass diese von einer Verfügbarkeit abgegrenzt wird.

Die Verfügbarkeit beschreibt die anteilsmäßige Dauer, die ein Objekt voll funktionsfähig ist[31]. Ist das Objekt nur teils funktionsfähig, wird von einer Teilverfügbarkeit gesprochen.

Systemtyp	Nicht verfügbar (Minuten pro Jahr)	Verfügbarkeit (in Prozent)	Klasse
Nicht verwaltet	50000	90	1
Verwaltet	5000	99	2
Gut verwaltet	500	99.9	3
Fehlertolerant	50	99.99	4
Hoch verfügbar	5	99.999	5
Sehr hoch verfügbar	0.5	99.9999	6
Ultra hoch verfügbar	0.05	99.99999	7

Tabelle 2.3: Verfügbarkeitsklassen[32]

Die Verfügbarkeit wird als

$$A = \frac{E_{\text{verfügbar}}}{E_{\text{verfügbar}} + E_{\text{nicht verfügbar}}}$$

definiert, wo A die Verfügbarkeit und E eine erwartete Dauer ist[31]. Über die in dieser Sektion definierten Metriken kann die Verfügbarkeit auch über

$$A = \frac{MTTF}{MTTF + MTTR}$$

bestimmt werden[31, 32].

Bei der Bestimmung der Verfügbarkeit muss zwischen der Verfügbarkeit einzelner Dienste und der Verfügbarkeit des gesamten Systems differenziert werden. Ein einzelner Dienst in einem Cloud-nativen System kann eine sehr niedrige Verfügbarkeit haben. Da aber z. B. mehrere Instanzen des Dienstes laufen, wäre die Verfügbarkeit des kompletten Systems höher. Aus Sicht eines zahlenden Nutzers ist die Verfügbarkeit aller Dienste relevant, für die er einen Vertrag geschlossen hat. Da an einem System mehrere Anwendungen für mehrere Kunden laufen, wird fortan die Verfügbarkeit des kompletten Systems betrachtet.

Die Verfügbarkeit ist in unterschiedliche numerische Klassen unterteilt. Die Klassen beginnen bei eins und folgen aufsteigend. Anhand der Klasse wird die Verfügbarkeit über $A(n) = 1 - 10^{-n}$ bestimmt, wobei n die numerische Klasse ist. Für die erste Klasse entspräche dies einer Verfügbarkeit von $A(1) = 90\%$. Eine Hochverfügbarkeit wird als Klasse fünf definiert, also einer Verfügbarkeit von 99.999%[32]. Weitere Verfügbarkeitsklassen sind Tabelle 2.3 zu entnehmen.

Je höher die Verfügbarkeit ist, desto höher sind die Systemkosten, da dafür z. B. Systeme dupliziert werden und der Ressourcenverbrauch steigt[33]. Zudem wird für eine hohe Verfügbarkeit ein Wartungsteam benötigt, welches die Kosten weiter ansteigen lässt[33]. Um Kosten zu sparen, kann eine für den Kunden angemessen niedrige Verfügbarkeit sichergestellt werden. Beziehungsweise können die Kosten der Verfügbarkeit auf den Kunden geschoben werden. Hierfür werden für Produkte mehrere Servicepläne erstellt. Der beste Serviceplan sichert eine Hochverfügbarkeit zu, kostet aber dementsprechend mehr. Der normale Serviceplan, welche von den meisten Kunden verwendet wird, sichert die Hochverfügbarkeit nicht zu, sondern z. B. nur eine Fehlertoleranz. Dafür kostet der normale Serviceplan weniger. Der Kunde trägt somit die Kosten für eine höhere Verfügbarkeit. Da Hardwareversagen in den letzten Jahren selten geworden sind, kann durch minimale Konfiguration des Systems eine Verfügbarkeitsklasse von drei oder vier erreicht werden[32].

Da bei SAP die Kunden vertraglich keine Hochverfügbarkeit vereinbart haben, müssen die Kosten für ein hochverfügbares System nicht aufgebracht werden. Es soll für SAP ein fehlertolerantes Cloud-natives System zugesichert werden. Folglich fokussiert sich die Arbeit auf Fehlertoleranz in Cloud-nativen Systemen.

2.3.1 Mean Time To Failure (MTTF)

Die Laufzeit Cloud-nativer Anwendungen hängt von mehreren Faktoren ab, welche sich über Metriken quantifizieren lassen. Der erste Faktor ist die Fehleranfälligkeit einer Anwendung. Diese wird über die Metrik MTTF gemessen. MTTF ist die durchschnittliche Dauer (in Minuten), bis in einem gesunden System ein Fehler auftritt[32, 34].

2.3.2 Mean Time To Detect (MTTD)

Die Fehleranfälligkeit wird bereits über MTTF quantifiziert. Sobald Fehler entstehen, müssen diese erst erkannt werden, sodass die Fehler danach beseitigt werden. Der zweite Faktor, welcher die Laufzeit Cloud-nativer Anwendungen beeinflusst, ist die Fehlererkennung. Die Fehlererkennung wird über die Metrik MTTD quantifiziert. MTTD gibt die durchschnittliche Dauer an, die benötigt wird, um einen entstandenen Fehler zu erkennen[34].

2.3.3 Mean Time To Repair (MTTR)

Der dritte Faktor ist die Fehlerreparatur, welcher über die Metrik MTTR angegeben wird. Je schneller erkannte Fehler repariert werden, desto verfügbarer sind die Anwendungen. MTTR gibt nicht nur die durchschnittliche Dauer an, die benötigt wird, um einen erkannten Fehler zu reparieren. MTTR berücksichtigt die komplette Dauer ab Entstehung des Fehlers und beinhaltet somit MTTD[32, 34].

Um die Verfügbarkeit der Services zu erhöhen, kann nicht nur die Stabilität der Anwendungen verbessert werden. Faktoren wie die Fehlererkennung spielen auch eine bedeutende Rolle. Da die Methodik Chaos-Engineering in Systemen Störungen auslöst, sind diese Faktoren besonders im Chaos-Engineering nicht zu vernachlässigen.

2.3.4 Verfügbarkeitszonen

Es gibt mehrere Ansätze, um die Verfügbarkeit vorhandener Dienste sicherzustellen. Eine allgemeine Lösung ist eine Redundanz der Komponenten. Wenn eine Instanz einer Komponente ausfällt, dann ist mindestens eine weitere Instanz verfügbar.

Um die Verfügbarkeit eines Cloud-nativen Systems sicherzustellen, wird redundante Hardware eingesetzt, auf welcher das System läuft. Die Hardware wird physisch auf Zonen aufgeteilt. Wenn Erdbeben oder andere Umstände die Hardware innerhalb einer Zone unbrauchbar machen, dann soll die Hardware innerhalb der anderen Zonen unbeeinflusst weiterhin funktionieren. Die Zonen haben hierfür zueinander einen Mindestabstand[35].

Der Nachteil einer einfachen Redundanz ist der hohe Ressourcenverbrauch. Zudem kann eine Redundanz weitere Probleme verursachen, die die Verfügbarkeit beeinträchtigen. Dennoch ist eine Redundanz ein einfacher Schritt in Richtung einer höheren Verfügbarkeit eines Cloud-nativen Systems.

2.3.5 Robustheit

Eine wichtige Eigenschaft Cloud-nativer Systeme, welche die Verfügbarkeit beeinflusst, ist die Robustheit. Die Robustheit besteht aus der Stabilität und der Widerstandsfähigkeit[3].

Die Stabilität bezeichnet die Fähigkeit unter Störung vorhandene Systemstrukturen beizubehalten und nicht zu brechen. Im Gegensatz dazu steht die Widerstandsfähigkeit. Hierbei geht es nicht um das Beibehalten von Strukturen. Es wird davon ausgegangen, dass vorhandene Strukturen bereits gebrochen wurden. Widerstandsfähigkeit beschreibt die Fähigkeit zerbrochene Strukturen wiederherzustellen[3].

2.4 Tests

In bisherigen Sektionen wurden vorhandene Strukturen aufgezeigt, wie z. B. ein Cloud-natives System und seine Eigenschaften. Aus den vorhandenen Strukturen wurden vorhandene Probleme aufgezeigt. In der Sektion Hochverfügbarkeit wurden erwünschte Eigenschaften vorhandener Systeme genannt. Nun soll in dieser Sektion auf die Sicherstellung der Verfügbarkeit eingegangen werden und erste bereits vorhandene Lösungen genannt werden.

Zuerst werden Tests eingeführt und klassifiziert. Anhand der Gegebenheiten soll ein passender Test bestimmt werden, sowie ein kurzer Blick auf verwandte Tests geworfen werden, welche an dem Prozess des Testens beteiligt sind.

Tests sind eine qualitätssichernde Maßnahme, welche Fehler in einem vorhandenen Produkt finden sollen. Fehler sind Abweichungen von einem Soll-Zustand. Der erwartete Zustand der Anwendungen richtet sich dabei nach den Nutzern, da die Nutzer die Erwartungen an die Anwendungen stellen. Tests werden somit aus Nutzererwartungen abgeleitet. Fehler können in einzelnen Komponenten der Anwendung auftreten. Selbst wenn sich einzelne Komponenten komplett fehlerfrei verhalten, kann das komplette System dennoch fehlerhaft sein. Es spielen nicht nur einzelne Komponenten eine Rolle, sondern auch die Interaktionen zwischen den Komponenten[36].

Für das Testen muss zuerst zwischen einem Fehler und dem Versagen unterschieden werden. Tests entdecken das Versagen einzelner Komponenten oder des Systems[36]. Versagen entstehen durch einen oder mehrere Fehler in einzelnen Komponenten oder in dem kompletten System[36]. Da Tests letztendlich Fehler finden sollen, müssen Rückschlüsse von dem Versagen auf den oder die Fehler gezogen werden[36]. Nachdem Fehler gefunden wurden, werden die Fehler behoben und eine neue Version der davor fehlerhaften Objekte ausgeliefert. Jede Änderung der Software führt zu potenziellen neuen Fehlern. Aus

diesem Grund reichen einzelne Tests nicht aus. Die Tests müssen nach jeder Änderung wiederholt werden, sodass die Korrektheit der momentan verwendeten Version des Objekts sichergestellt wird. Wiederholte Tests sind zu automatisieren, da dadurch Arbeitsaufwand gespart wird. Um den Soll-Zustand des Systems sicherzustellen, muss für jeden Teilaspekt des Soll-Zustands ein Test geschrieben werden, sodass der komplette Soll-Zustand durch Tests abgedeckt wird[36].

In dem Test wird der Soll-Zustand des Objektes spezifiziert und dann mit dem tatsächlichen Zustand verglichen. Wenn Abweichungen festzustellen sind, dann entspricht das Objekt nicht den Nutzererwartungen, ist fehlerhaft und muss abgeändert werden[36].

Je größer ein vorhandenes System ist, desto mehr fehlerhafte Komponenten können existieren. Ein Cloud-natives System ist deshalb sehr fehleranfällig, da das System aus vielen Komponenten besteht und auf mehreren Abstraktionsschichten aufbaut. Fehler sind aus einer ökonomischen Sicht kostspielig, da die daraus entstehenden Versagen zu Vertragsbrüchen führen[37]. Andererseits verursacht die Implementierung der Tests sowie das dafür zugehörige Team auch Kosten. Für das Testen der Komponenten eines Systems, sowie des Testens des Systems ist somit immer eine Kostenabwägung zu treffen. In der Regel lohnt es sich gründlich zu testen, da die Folgen der Versagen kostspieliger sind[37]. Dies gilt im Besonderen, wenn die zu testenden Komponenten von vielen Kunden im Einsatz sind.

2.4.1 Klassifikationen

Es gibt eine Vielzahl unterschiedlicher Testtypen, welche anhand ihrer Eigenschaften klassifiziert werden. Die Eigenschaften werden in dieser Sektion eingeführt.

Eine Eigenschaft eines Tests ist die verwendete Testmethodik. Testmethodiken sind das statische und dynamische Testen. Bei statischen Tests ist das zu testende Objekt der Quellcode einzelner Komponenten[36]. Die Struktur und Inhalt des Quellcodes werden analysiert und mit einer erwarteten Struktur und einem erwarteten Inhalt verglichen. Vorteil dieser Testmethodik ist, dass der Test ausgeführt wird, bevor die Komponente gebaut wird. Durch diese Testmethodik kann aber nicht das Verhalten der laufenden Anwendung getestet werden[36].

Um das Verhalten der laufenden Anwendung zu testen, wird die Testmethodik der dynamischen Tests verwendet. Dynamische Tests testen die gebaute Anwendung, indem der Anwendung Eingabedaten zugeführt werden. Das daraus entstandene Resultat wird mit dem erwarteten Resultat verglichen.

Dynamische Tests werden in weitere Kategorien unterteilt, je nach dem verwendetem Wissen über das zu testende Objekt. Es wird zwischen Black-Box-Tests und White-Box-Tests unterschieden. Bei White-Box-Tests wird tiefes Wissen über den inneren Aufbau und die genaue Funktionsweise des Objekts herangezogen. Durch das Wissen über das Innere des Objekts können die Tests gezielt Schwachstellen testen. Dieses Wissen ist jedoch nicht immer verfügbar[36].

Wenn das Wissen über das Innere des Objekts nicht vorhanden ist, dann wird Black-Box-Testen angewandt. Black-Box-Testen kann auch angewandt werden, wenn dieses Wissen nicht benötigt wird. Das Objekt wird als ein schwarzer Container visualisiert, in welchem die inneren Strukturen und Prozesse nicht ersichtlich sind. Es kann nur das Verhalten bei Eingabe von Daten anhand der Ausgabe beobachtet werden. Der Black-Box-Test wird aus dem erwarteten Verhalten abgeleitet[36].

Durch Chaos-Engineering soll das Verhalten des kompletten Systems unter Störung betrachtet werden. Dabei ist das Verhalten des Systems bei Nutzerinteraktion besonders wichtig, da das System korrekt auf Anfragen des Nutzers reagieren soll. Da das Verhalten des laufenden Systems wichtig ist, werden im Folgenden dynamische Tests betrachtet.

2.4.2 Modultest

Modultests sind White-Box-Tests[38]. Das zu testende Objekt ist eine kleine Einheit einer laufenden Anwendung. Die Einheit ist z. B. eine einzelne Funktion der Anwendung. In einem Cloud-nativen System ist ein Microservice als Modul zu betrachten. Durch das Testen einzelner kleiner Einheiten ist das Rückschließen von Versagen auf Fehler einfach, da sich die Fehler in der kleinen Einheit befinden[38]. Um Anwendungen vollständig zu testen, werden dementsprechend viele Modultests benötigt.

Um die einzelne kleine Einheit unabhängig anderer Einheiten zu testen, werden Abhängigkeiten durch Attrappen ausgetauscht. Der Modultest befindet sich zu Beginn des Entwicklungszyklus, da Modultests nach Änderung eines Moduls ausgeführt werden. Da

Attrappen anstelle von Abhängigkeiten verwendet werden, können Modultests auch lokal ausgeführt werden[38].

2.4.3 Integrationstest

Nachdem über Modultests die Korrektheit einzelner Module sichergestellt wurde, wird über ein Integrationstest das korrekte Zusammenspiel der Module sichergestellt. Der Integrationstest ist ein Black-Box-Test. Voraussetzung für Integrationstests sind Modultests, da in Integrationstests davon ausgegangen wird, dass die einzelnen Module korrekt sind[39].

Da Microservices als Module in einem Cloud-nativen System zu betrachten sind, testet der Integrationstest die Kommunikation zwischen Microservices.

2.4.4 Systemtest

Nachdem Integrationstests durchgeführt wurden, folgen die Systemtests. Systemtests brauchen kein Wissen über das Innere des Systems und sind somit Black-Box-Tests. Systemtests unterscheiden sich von Integrationstests, indem sie das System als Ganzes testen. Integrationstests betrachten nur Interaktionen zwischen einzelnen Modulen. Aus den Modulen baut sich das System auf[39].

2.4.5 Funktionaler Test

Der funktionale Test ist ein Black-Box-Test, welcher einzelne Module eines kompletten Systems testet[36]. Es wird getestet, ob die Module den Anforderungen der Nutzer gerecht werden.

In einem Cloud-nativen-System kann der funktionale Test Endpunkte eines Dienstes testen, um z. B. festzustellen, ob der Dienst lebendig ist.

Ein Beispiel für einen funktionalen Test ist der Rauchtest. In einem Rauchtest wird ein Modul gestartet. Das erwartete Ergebnis ist der Start des Moduls ohne Versagen. Durch Rauchtests wird erkannt, ob ein Modul bei Start anfängt zu „brennen“[40].

2.4.6 Leistungstest

Der Leistungstest geht einen Schritt über einen funktionalen Test hinaus. Der Leistungstest testet nicht nur die Nutzeranforderungen, sondern auch die Nutzererwartungen. Hierfür wird die Leistung der Module getestet. Es werden z. B. die Antwortdauer und Durchsatzrate der Module erfasst.

Ein spezifischer Leistungstest ist der Lasttest. Der Lasttest simuliert eine hohe Nutzlast der Nutzer.

2.4.7 Chaos-Engineering

Über Tests wie z. B. den Modultest wird die Korrektheit einzelner Module sichergestellt, bevor die Anwendung ausgeliefert wird. Über weitere Tests wie den funktionalen Test, Integrationstest und Systemtest wird sichergestellt, dass die Module und das System den Erwartungen der Kunden entsprechen. Hierdurch wird die Verfügbarkeit der Anwendungen bedingt sichergestellt, da die Verfügbarkeit ein Teil des erwarteten Verhaltens ist und somit durch die Tests abgedeckt wird.

Die genannten Tests sind eine gute Grundlage für fehlertolerante und korrekte Anwendungen, reichen jedoch nicht aus, um eine Verfügbarkeit der Anwendungen zu garantieren. Die Tests garantieren eine korrekte Kommunikation zwischen den Modulen in einem stabilen System. Doch kommunizieren die Module korrekt, wenn das System von außen gestört wird? Durch ein Hardwareversagen der Hardware des IaaS-Betreibers könnte spontan ein Rechner des Systems ausfallen. Durch einen Stromausfall könnte eine ganze Region des Systems ausfallen. Das Netzwerk könnte gestört sein, wodurch die Kommunikation gestört wird.

Die davor genannten Tests eignen sich somit nicht, um die Verfügbarkeit in allen Situationen sicherzustellen. Um die Verfügbarkeit in außergewöhnlichen Situationen zu garantieren, wurde die Methodik Chaos-Engineering entworfen. In dieser Sektion soll zuerst ein Blick auf die Prinzipien des Chaos-Engineering geworfen werden. In der Lektüre wird im Zusammenhang mit Chaos-Engineering der Begriff Chaos-Test verwendet. Es soll zudem diskutiert werden, was ein Chaos-Test ist und ob ein Chaos-Test und Chaos-

Engineering als Synonyme verwendet werden. Zuletzt wird ein Blick auf vorhandene Methodiken zur Umsetzung von Chaos-Engineering geworfen.

2.4.7.1 Prinzipien

Chaos-Engineering ist dafür entworfen, Vertrauen in die Robustheit eines System aufzubauen. Um das Vertrauen aufzubauen, wird auf dem System experimentiert. Es werden zuerst realistische Versagen in dem System verursacht. Danach wird das Verhalten des Systems beobachtet[5].

Im ersten Schritt wird ein stabiler Zustand des Systems definiert. Der stabile Zustand ist ein Soll-Zustand, in welchem sich das System befinden soll. Der stabile Zustand ist somit der funktionierende Normalzustand eines Systems. Dieser kann z. B. über die Endpunkte eines Dienstes überprüft werden. Wenn der Endpunkt eines Dienstes korrekt auf Nutzereingabe reagiert, dann ist zumindest der Dienst in einem stabilen Zustand. Es wird somit z. B. die Verfügbarkeit einer Anwendung getestet. Als Test dient hier ein Black-Box-Test, welcher das Verhalten des Dienstes überprüft. Der Test muss zumindest überprüfen, ob Endpunkte korrekt auf Eingaben reagieren, beziehungsweise die Nutzeranforderungen erfüllt sind[5].

Im zweiten Schritt wird die Hypothese aufgestellt, dass das System robust ist. Da das System robust ist, sollte das System trotz eintretenden Versagen in einem stabilen Zustand bleiben. Eintretende Versagen werden als Chaos bezeichnet[5].

Daraufhin wird im dritten Schritt realistisches Chaos in dem System verursacht. Hierfür können unterschiedliche Methodiken verwendet werden. Es können entweder gezielt häufig auftretende Versagen verursacht werden, oder es können Versagen beliebig verursacht werden[5]. Um bestimmte Versagen zu simulieren, wird auf vorhandene Abstraktionsschichten zurückgegriffen. Zur Simulation von Hardwarefehlern wird z. B. die Abstraktionsschicht des IaaS-Betreibers verwendet.

Zuletzt wird das System beobachtet und der aktuelle Zustand mit dem stabilen Zustand verglichen. Sollte das System während des Experiments nicht in einem stabilen Zustand gewesen sein, dann ist das System nicht robust und muss verbessert werden. Für die Beobachtung dienen z. B. Black-Box-Tests[5]. Die Black-Box-Tests liefern nur einen Blick auf das System aus der Sicht eines Nutzers. Um das Verhalten des Systems während

des Chaos näher zu beobachten, müssen tiefgreifendere Informationen verwendet werden. Hierfür dienen z. B. Werkzeuge, welche die Metriken der laufenden Anwendungen aggregieren.

2.4.7.2 Experiment vs. Test

In der Literatur sind die Begriffe Chaos-Engineering, Chaos-Test und Chaos-Experiment zu finden[6]. Da die Begriffe fast synonymisch verwendet werden, sollen die Begriffe abgegrenzt werden. In dieser Sektion wird diskutiert, ob das Chaos-Engineering ein Test oder Experiment ist.

Chaos-Engineering erfüllt die Eigenschaften eines Tests. Chaos-Engineering kann als dynamischer Tests angesehen werden, welcher die Reaktion des Systems auf Versagen testet. Dabei wären die Eingaben des Tests Nutzdaten wie z. B. ein JavaScript Object Notation (JSON)-Dokument und ein Versagen wie z. B. der Ausfall eines *Pods*. Wenn das System robust ist, dann werden die Nutzdaten von dem System korrekt empfangen, bearbeitet und es wird eine korrekte Antwort erhalten. Andernfalls scheitert der Test. Eine Skizze kann Abbildung 2.4 entnommen werden. Chaos-Engineering ist auch eine qualitätssichernde Maßnahme.

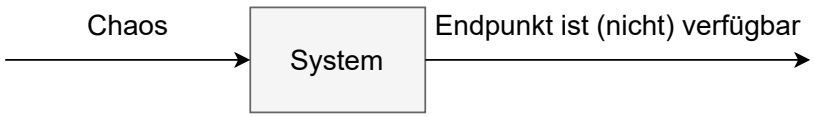

Abbildung 2.4: Test-ähnliches Chaos-Engineering

Obwohl Chaos-Engineering die Eigenschaften eines Tests augenscheinlich erfüllt, kann es nicht als Test bezeichnet werden. Das im System verursachte Chaos kann nicht als Eingabe des Tests betrachtet werden. Es ist losgelöst von den Nutzdaten. Nach den Prinzipien des Chaos-Engineering muss das System z. B. durch einen Test beobachtet werden. Dabei ist der Test aber nur ein Bestandteil des Chaos-Engineering. Der stabile Zustand des Systems kann an mehreren Stellen während des Experiments getestet werden. Das Chaos wird dabei nur einmalig verursacht. Wie der Wortlaut hier verrät und wie aus den Quellen herauszulesen ist, handelt es sich um ein Experiment.

Das Chaos-Engineering ist letztendlich die Methodik, über welche experimentell die Robustheit eines Systems sichergestellt werden kann. Ein Chaos-Experiment ist dabei

eine Instanz des Chaos-Engineering. Innerhalb eines Experiments werden Tests verwendet, um z. B. die Verfügbarkeit eines Dienstes zu Testen. Der Chaos-Test ist somit nur eine Komponente des Experiments.

2.4.7.3 Circuit-Breaker

Es gibt mehrere Methodiken, um Chaos-Engineering umzusetzen. Eine grundlegende Maßnahme ist die Kontrolle des verursachten Chaos. Dabei ist die Rede von dem *Blast-Radius*, dem Ausmaß des verursachten Chaos. Wenn das Ausmaß des Chaos zu groß ist und das System nicht robust ist, dann versagt das komplette System. Indem das Ausmaß verkleinert wird, wird somit der mögliche verrichtete Schaden kontrolliert. Es muss beachtet werden, dass kein unrealistisches Chaos gewählt wird, nur um das Ausmaß zu reduzieren.

Eine Methodik ist der *Circuit-Breaker*. Durch einen *Circuit-Breaker* wird der *Blast-Radius* kontrolliert. Ein *Circuit-Breaker* kontrolliert das Ausmaß des Chaos, indem versagende Komponenten von dem restlichen System abgetrennt werden. Hierdurch breitet das Chaos sich nicht aus, sondern wird isoliert. Anhand bestimmter definierter Kriterien wird eintretendes Chaos erkannt. Kriterien sind eine außergewöhnlich hohe Latenz, oder viele fehlerhafte Hypertext Transfer Protocol (HTTP)-Statuscodes. Nach einer definierten Dauer kann eine abgetrennte Komponente probeweise wieder an das System angeschlossen werden. Falls in der Komponente keine weiteren Versagen auftreten, dann bleibt diese an das System verbunden[41].

Vorteil eines *Circuit-Breakers* ist eindeutig die Isolation des Chaos. Wenn Chaos auf einem Produktivsystem verursacht wird, dann schützt der *Circuit-Breaker* die Kunden davor, mit versagenden Komponenten zu interagieren. Zudem kann diese Methodik nicht nur auf Produktivsystemen eingesetzt werden, sondern auch auf jeglichen anderen Systemen.

Nachteil ist, dass die Erkennung des Chaos gründlich implementiert sein muss. Ist die Erkennung des Chaos zu aggressiv, dann werden fehlerfreie Komponenten vom System abgetrennt. Ist die Erkennung zu passiv, dann werden fehlerhafte Komponenten gar nicht, oder zu spät abgetrennt. Eine korrekte Implementierung eines *Circuit-Breakers* ist folglich herausfordernd.

2.4.7.4 Iterative Methode

Anstelle in einem System von Anfang an das volle Chaos zu verursachen und dabei auf *Circuit-Breaker* zu vertrauen, kann auch eine iterative Herangehensweise gewählt werden. Chaos-Experimente werden im Produktivsystem ausgeführt. Da zu Beginn das Vertrauen in das System und die Werkzeuge fehlt, kann das Vertrauen iterativ aufgebaut werden. Zuerst wird in einem Chaos-Experiment einfach zu kontrollierendes Chaos verursacht, welches kleine Auswirkungen hat. Das Chaos könnte den Absturz eine *Pods* verursachen. Nachdem Vertrauen in die Werkzeuge und das System aufgebaut wurde, wird komplexeres Chaos im System verursacht, welches größere Auswirkungen mit sich zieht. Hierdurch wird Stück für Stück Vertrauen aufgebaut[6].

Vorteil der Methode ist, dass nicht nur Vertrauen in das System aufgebaut wird, sondern auch in das verwendete Werkzeug zur Umsetzung des Chaos-Experiments. Zudem wird direkt auf dem Produktivsystem getestet. Da zu Beginn das Ausmaß der Experimente reduziert wird, wird der mögliche Schaden reduziert.

Das Iterieren und Vergrößern des *Blast Radius* ist gleichzeitig ein Vorteil und ein Nachteil. Da das Ausmaß schrittweise vergrößert wird, wird eine gewisse Dauer benötigt, bis realistisch großes Chaos erreicht wird. In Fällen, in welchen die Verfügbarkeit der Anwendung schnell garantiert werden muss, eignet sich die Methodik somit nicht.

2.4.7.5 System-Pipeline-Methodik

Die *System-Pipeline-Methodik* verursacht Chaos auf zwei oder mehreren Systemen nacheinander. Es wird ähnlich zur iterativen Methode das Vertrauen langsam aufgebaut. Das Vertrauen wird jedoch nicht aufgebaut, indem der *Blast Radius* vergrößert wird, sondern indem auf unterschiedlichen Systemen experimentiert wird. Es wird auf einem nicht-Produktivsystem gestartet. Auf dem System wird ein Chaos-Experiment ausgeführt. Gelingt das Experiment, dann wird es mit einer gewissen Wahrscheinlichkeit auch auf dem nächsten System gelingen, da beide Systeme Ähnlichkeiten haben. Also wird das Experiment nach einem Gelingen dann z. B. auf dem Produktivsystem ausgeführt. Wenn das Experiment bereits auf dem nicht-Produktivsystem fehlschlägt, dann wird das Experiment nicht auf dem Produktivsystem ausgeführt[6].

Vorteil der *System-Pipeline-Methodik* ist die Verwendung eines oder mehrerer Systeme, um auftretende Versagen bereits vor dem Produktivsystem abzufangen. Dadurch wird die Fehlerfrequenz auf dem Produktivsystem gesenkt, das Fehlerausmaß jedoch nicht verringert.

Nachteil ist der hohe Aufwand zum Aufsetzen der Chaos-Experimente, da diese in mehreren Systemen aufgesetzt werden müssen. Hinzu kommt der hohe Ressourcenverbrauch, da mehrere laufende Systeme benötigt werden.

3 Konzeptionierung

In dem Kapitel Stand der Technik wurde gezeigt, dass heutige flexible Anwendungen als Cloud-native Anwendungen entwickelt werden. Die Anwendungen folgen dabei einer Microservices-Architektur. Cloud-native Systeme sind komplex und Versagen können in vielen Bereichen auftreten. Es kann die Hardware eines IaaS-Betreibers versagen. Einzelne Module können Fehler enthalten, oder auch das komplette System verhält sich fehlerhaft. Die durch die Fehler entstandenen Versagen haben einen Einfluss auf die Verfügbarkeit der Anwendung und dadurch auf das Erlebnis eines zahlenden Kunden, welcher die Produkte verwendet. Die Verfügbarkeit hat somit einen direkten Einfluss auf die Kundenzufriedenheit und dadurch auf die Einnahmen eines Unternehmens. Eine zu hohe Verfügbarkeit wird nicht benötigt, erhöht nur die benötigten Ressourcen und sollte deshalb nicht angestrebt werden.

Um die Verfügbarkeit zu garantieren, werden qualitätssichernde Maßnahmen eingesetzt. Diese reichen von einfachen Modultests, über Integrationstests und Systemtests bis zu funktionalen Tests, wie dem Rauchtest. Dennoch wird durch die qualitätssichernden Maßnahmen die Verfügbarkeit in außergewöhnlichen Situationen nicht sichergestellt, es wird die Methodik Chaos-Engineering benötigt. Chaos-Engineering baut Vertrauen in die Verfügbarkeit eines Systems in außergewöhnlichen Situationen auf. Um das Vertrauen in das produktive System aufzubauen, müssen Chaos-Experimente im Produktivsystem ausgeführt werden. Dabei verursachen die Experimente gezielt Störungen. Wenn das Ausmaß der Störungen nicht kontrolliert wird, dann wird die Verfügbarkeit des Produktivsystems negativ beeinflusst.

Um die Risiken des Chaos-Engineerings zu senken, wird das Chaos-Engineering auf einem nicht-Produktivsystem umgesetzt. Es bedarf hierfür eines Konzepts, welches dabei die drei in der Motivation genannten Ziele einhält. Hierfür werden zuerst genaue Anforderungen aus den Zielen abgeleitet. In diesem Kapitel soll das Konzept eingeführt werden.

3.1 Anforderungen

Die konkrete Aufgabe, die genannt wurde, ist das effiziente Umsetzen von Chaos-Engineering auf einem nicht-Produktivsystem. Für die Aufgabe wurden die drei Ziele der Risikominimierung, Kostenminimierung und Qualitätssicherung des Produktivsystems genannt.

Für das Ziel der Risikominimierung wird gefordert, dass das Chaos-Engineering auf einem nicht-Produktivsystem umgesetzt wird. Dadurch beschränken sich die negativen Auswirkungen der Experimente auf das nicht-Produktivsystem.

Um eine Kostenminimierung zu erzielen, wird eine Kosteneffizienz gefordert. Die Kosteneffizienz setzt sich aus zwei Faktoren zusammen. Der erste Faktor ist die Zeiteffizienz. Eine Zeiteffizienz wird erreicht, indem die manuelle Arbeit durch Automatisierung gesenkt wird. Indem manuelle Arbeit gesenkt wird, werden Personalkosten gespart. Der zweite Faktor ist die Minimierung der laufenden Systemkosten des Chaos-Engineerings. Die Minimierung der laufenden Systemkosten soll durch eine Wiederverwendung vorhandener Systeme und Komponenten erzielt werden.

Um das Ziel der Qualitätssicherung des Produktivsystems zu erreichen, wird eine Vergleichbarkeit gefordert. Das nicht-Produktivsystem soll dem Produktivsystem angepasst werden, sodass beide System sich ähneln. Über auf dem nicht-Produktivsystem laufende Chaos-Experimente sollen für die Verfügbarkeit des Produktivsystems wichtige Informationen gesammelt werden.

3.2 Bewertungskriterien

Das entwickelte Konzept und die dazugehörige Implementation werden anhand der Anforderungen umgesetzt. Um die Qualität der Ergebnisse bewerten zu können, werden anhand der Anforderungen in dieser Sektion Bewertungskriterien abgeleitet. Es wird zuerst zwischen harten und weichen Bewertungskriterien unterschieden.

Die harten Bewertungskriterien können einfachen gemessen werden und sind quantitativ vergleichbar. Im Gegensatz dazu ist bei den weichen Bewertungskriterien eine Messung schwer durchführbar, oder die quantitative Vergleichbarkeit fehlt.

3.2.1 Harte Bewertungskriterien

Eine Anforderung an das Konzept und die Implementierung ist die Effizienz. Als hartes Bewertungskriterium für die Effizienz wird die Zeiteffizienz herangezogen. Es wird jeweils gemessen, wie viel Zeit ein Mitarbeiter in ein manuell ablaufendes Experiment investieren muss, im Gegensatz zu der Zeitinvestition in ein automatisiertes Experiment. Das Kriterium wird jeweils über $\frac{t_{manuell}}{t_{automatisch}}$ und $t_{manuell} - t_{automatisch}$ gemessen, wobei eine größere Zahl besser ist.

3.2.2 Weiche Bewertungskriterien

Aus der Effizienz wird zudem die Kosteneffizienz abgeleitet. Die Kosteneffizienz ist ein weiches Bewertungskriterium, da sie nur schwer gemessen werden kann, weil die Kosteneffizienz sich aus mehreren Bestandteilen zusammensetzt. Die Kosteneffizienz hängt z. B. von der Zeiteffizienz ab. Je mehr manuelle Arbeit automatisiert wird, desto mehr Zeit kann ein Mitarbeiter in andere Projekte investieren.

Das Konzept hat als weitere Anforderung das Schaffen einer Vergleichbarkeit zwischen einem nicht-Produktivsystem und Produktivsystem identifiziert. Die Vergleichbarkeit ist ein fundamentaler Aspekt der Arbeit und wird bewertet. Da die Vergleichbarkeit nur schwer gemessen werden kann, ist sie ein weiches Bewertungskriterium.

3.3 Konzept

Als Basis für das Konzept dienen die Komponenten des Chaos-Engineerings. Es werden zuerst die Komponenten des Chaos-Engineerings identifiziert. Bereits vorhandene Komponenten sollen wiederverwendet werden, um den Zeitaufwand einer Implementation zu senken. Im Anschluss werden die vorhandenen Komponenten um weitere erweitert, sodass die Anforderungen erfüllt werden. Anhand der Anforderungen werden zudem die vorhandenen Komponenten angepasst. Indem Chaos-Engineering als Basis für das Konzept gewählt wird, soll eine Wiederverwendung vorhandener Komponenten ermöglicht werden und somit das zweite Ziel erfüllt werden.

3.3.1 Identifizierung der Komponenten

Chaos-Engineering dient als Basis für das Konzept. Über Chaos-Engineering werden für ein oder mehrere Systeme mehrere Chaos-Experimente entworfen, welche den Prinzipien des Chaos-Engineering folgen.

Um alle Chaos-Experimente zu verwalten, wird die Komponente *Chaos-Verwalter* benötigt. Unter Verwaltung werden typische Create Read Update Delete (CRUD)-Aktionen, sowie ein Ausführen und Suchen verstanden. Ausgeführte Chaos-Experimente haben ein Ergebnis. Chaos-Experimente können entweder scheitern oder erfolgreich sein. Die Ergebnisse der Chaos-Experimente werden persistent in dem *Chaos-Verwalter* abgelegt, um nach Vollendung eines Experiments verfügbar zu sein.

Zur Ausführung der Chaos-Experimente wird die Komponente *Chaos-Laufzeit* benötigt. Die Laufzeit ist ein Interpreter für Chaos-Experiment-Definitionen und dient als Umgebung, welche alle für die Ausführung benötigten Komponenten enthält. Ein Prinzip des Chaos-Engineering ist der *Steady State*, der definierte stabile Zustand des Systems. Um den stabilen Zustand eines Systems zu überprüfen, wird mindestens ein Black-Box-Test benötigt. Der Test wird von der Laufzeit gestartet und wird als *Probe* bezeichnet. Die definierten *Proben* werden von der Laufzeit mehrfach ausgeführt. Zuerst werden die *Proben* bei Beginn des Experiments ausgeführt, sodass das Experiment abgebrochen wird, wenn das System nicht in einem stabilen Zustand ist. Störungen werden dadurch nur in einem gesunden System verursacht. Nachdem das Chaos im System verursacht wurde, werden die *Proben* erneut ausgeführt. Nun wird getestet, ob der stabile Zustand beibehalten wurde. Die Ergebnisse der *Proben* beeinflussen direkt das Ergebnis des Experiments.

Ein Prinzip des Chaos-Engineering besagt, dass als Teil eines Experiments realistisches Chaos in einem System verursacht werden soll. Das Stören des Systems wird als Komponente *Aktion* bezeichnet. Die *Aktionen* definieren imperativ die Schritte, die benötigt werden, um ein System zu stören. Die *Chaos-Laufzeit* führt alle definierten Aktionen nach dem Ausführen der ersten *Proben* aus. Das gestörte System ist auch eine Komponente, da das System an dem Prozess beteiligt ist.

Die Ergebnisse der Experimente werden über eine weitere Komponente, der *Experimentauswertung* visualisiert. Dadurch können bei Bedarf Statistiken zu den Experimenten erlangt werden.

3.3.2 Automatisierung

Die bis jetzt identifizierten Komponenten reichen für manuelles Chaos-Engineering aus. Um eine Zeiteffizienz zu erzielen, soll das Chaos-Engineering automatisiert werden. Hierbei soll nicht nur das Chaos-Experiment automatisiert werden, sondern auch alle weiteren Schritte der Vorbereitung und Nachbereitung. Die Automatisierung reduziert die manuelle Arbeit und erhöht dadurch die automatisierte Arbeit. Um automatisiertes Chaos-Engineering zu ermöglichen, wird ein *Chaos-Auslöser* benötigt. Die Komponente *Chaos-Auslöser* sendet periodisch oder in unregelmäßigen Intervallen Anfragen an die *Chaos-Verwaltung*. Über die Anfragen werden Experimente automatisiert ausgelöst.

Dennoch wird das letzte Prinzip nicht automatisiert, die Beobachtung des Systems. Für eine automatisierte Beobachtung müssen die Ergebnisse der Experimente an die Mitarbeiter gelangen, wenn die Mitarbeiter die Ergebnisse benötigen. Die Mitarbeiter benötigen die Ergebnisse genau dann, wenn die Experimente scheitern. Es fehlt somit die Komponente *Mitarbeiterbenachrichtigung*.

Nun wurden alle Komponenten genannt, die für ein automatisiertes generelles Chaos-Engineering benötigt werden. Die Komponenten sind Abbildung 3.1 zu entnehmen. In der nächsten Sektion wird betrachtet, wie nun eine Vergleichbarkeit geschaffen wird.

3.3.3 Vergleichbarkeit

Um eine Vergleichbarkeit zwischen dem Produktivsystem und einem nicht-Produktivsystem zu schaffen, müssen zuerst die Unterschiede der Systeme betrachtet werden. Im Anschluss werden die Unterschiede priorisiert. Zuletzt wird anhand der priorisierten Unterschiede eine Lösung für die Vergleichbarkeit aufgestellt.

Unterschiede Da Cloud-native Systeme sehr komplex sind, können sich mehrere Cloud-native Systeme in vielen Aspekten unterscheiden.

Der erste Aspekt, in welchem sich zwei Systeme unterscheiden können, ist die unterliegende Hardware, auf welcher die Dienste laufen. Die Hardware wird durch einen IaaS-Betreiber einheitlich bereitgestellt. Wenn für beide Systeme ein unterschiedlicher IaaS-Betreiber gewählt wird, dann verhält sich die Hardware unterschiedlich.

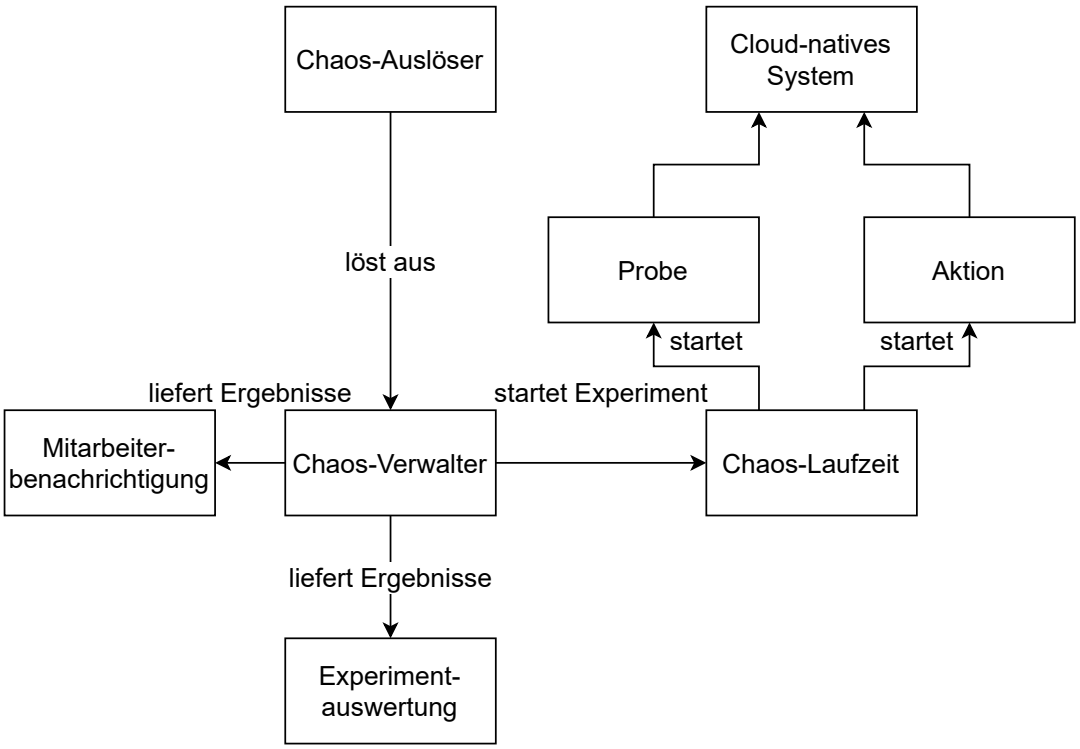

Abbildung 3.1: Komponenten des Chaos-Engineering

Es wird angenommen, dass auf einem nicht-Produktivsystem dieselben Dienste laufen, die auch auf einem Produktivsystem laufen. Es wird zudem angenommen, dass die Dienste auf dieselbe Art und Weise gebaut wurden. Folglich verwenden beide Systeme entweder eine PaaS-Lösung, oder eine Containerverwaltung. Beide Systeme verwenden dieselbe Lösung, sodass Dienste des nicht-Produktivsystems schlussendlich über dieselbe Methodik auf dem Produktivsystem installiert werden. Wenn für beide Systeme eine Containerverwaltung wie K8s gewählt wurde, dann wäre ein weiterer Unterscheidungspunkt die Containerlaufzeit. Dies ist auch auszuschließen, da die Dienste für eine spezielle Containerlaufzeit gebaut wurden, wie z. B. Containerd. Deshalb muss auf beiden Systemen dieselbe Containerlaufzeit verwendet werden.

Wie genannt, laufen auf beiden Systemen dieselben Dienste. Jedoch die Skalierung der Dienste kann sich unterscheiden. Da sich auf dem nicht-Produktivsystem weniger Kunden befinden, werden weniger Ressourcen benötigt. Zudem wird die Anzahl der laufenden Instanzen eines Dienstes klein gehalten. Im Gegensatz dazu steht das Produktivsystem.

Da auf dem Produktivsystem die meisten Kunden arbeiten, müssen entsprechend viele Instanzen der Dienste vorhanden sein. Es wird durch die Kunden mehr Hardware benötigt.

Die Gegebenheit, dass die Kunden sich hauptsächlich auf dem Produktivsystem befinden, beeinflusst die Interaktionen zwischen den Diensten. In Systemen, auf welchen Kunden arbeiten, geschehen mehr Interaktionen.

Priorisierung In diesem Abschnitt werden die genannten Unterschiede priorisiert. Da es sich um komplexe Systeme handelt, ist es schwer zu sagen, was für Auswirkungen die Unterschiede in den Systemen haben. Dennoch sind Interaktionen ein wichtiger Bestandteil komplexer Systeme. Das Ausmaß der Unterschiede in den Interaktionen zwischen beiden Systemen kann somit die Schwere eines Unterschiedes indizieren.

Eine andere Hardware hat keinen direkten Einfluss auf die Interaktionen zwischen den Diensten eines Cloud-nativen Systems. Somit hat dieser Unterschied eine geringe Priorität.

Der nächste Unterschied sind die Kunden, die hauptsächlich auf dem Produktivsystem sind. Die Kunden interagieren mit dem System und lösen dadurch Interaktionen zwischen den Diensten aus. Auf dem nicht-Produktivsystem sind dadurch weniger Interaktionen. Hiernach ist der Unterschied mit einer hohen Priorität zu versehen.

Augenscheinlich unterscheiden sich das Produktivsystem und ein nicht-Produktivsystem höchstens nur durch den IaaS-Betreiber und die Kunden, die das System verwenden. Dennoch kann es weitere kleine Unterschiede geben, welche einen großen Einfluss auf das Verhalten der Systeme haben können. Da aber im Besonderen die Kunden einen großen Einfluss auf die Interaktionen zwischen den Diensten eines Systems haben, fokussiert sich eine Lösung auf eine Generierung Kunden ähnlicher Interaktionen. Kunden ähnliche Interaktionen werden im Folgenden als Nutzeranfragen bezeichnet.

Generierung von Nutzeranfragen Zur Generierung von Nutzeranfragen können unterschiedliche Methoden angewandt werden. Für das Konzept wird das dynamische Generieren von Nutzeranfragen ausgewählt, da dafür vorhandene Komponenten wiederverwendet werden können.

Das dynamische Generieren von Nutzeranfragen generiert Nutzeranfragen dynamisch, also nur für einen begrenzten Zeitraum. Der Zeitraum ist der Zeitraum eines Chaos-

Experiments, da die Nutzeranfragen während eines Experiments generiert werden sollen. Da die Generierung nur während des Experiments läuft, werden nur die benötigten Systemressourcen für die Generierung verwendet. Indem die Lebensdauer der Nutzeranfragengenerierung an das Experiment geknüpft wird, wird sichergestellt, dass minimale Ressourcen verbraucht werden. Zudem fügt sich die Nutzeranfragengenerierung dadurch konzeptuell in das Containermodell von K8s. Da die Nutzeranfragengenerierung an das Experiment geknüpft wird, wird für jedes laufende Experiment eine dazugehörige Nutzeranfragengenerierung erstellt, sodass mehrere Experimente in parallel laufen können. Nach Abarbeitung des Experiments wird die Generierung gestoppt und die Ergebnisse der Generierung ausgelesen.

Für die Nutzeranfragengenerierung wird das Konzept um eine weitere Komponente erweitert, der *Nutzergenerierung*. Die *Nutzergenerierung* bietet eine Schicht der Indirektion, sodass die Nutzeranfragengenerierung mittels unterschiedlicher Werkzeuge vereinheitlicht wird. Um eine Zeiteffizienz zu erzielen, wird ein Fokus auf Wiederverwendbarkeit von vorhandenem Code gelegt. Anstelle die Nutzeranfragengenerierung mit vorhandenen Werkzeugen neu zu implementieren, wird auf vorhandene Black-Box-Tests zurückgegriffen, wenn die Tests bereits vorhanden sind. Im Besonderen bieten sich funktionale Tests sowie Leistungstests an, da beide Tests in Nutzeranforderungen und Nutzererwartungen verankert sind. Für die Nutzeranfragengenerierung reichen funktionale Tests nicht vollkommen aus. Funktionale Tests fokussieren sich nur auf Nutzeranforderungen und berücksichtigen somit nicht einfache Nutzererwartungen, wie z. B. eine niedrige Latenz der Dienste. Leistungstests gehen zu weit, da Leistungstests sich zwar auf Nutzererwartungen fokussieren, jedoch das System zu intensiv testen.

Für eine Wiederverwendbarkeit von vorhandenem Code werden vorhandene Leistungstests modifiziert. Es wird die Dauer und Intensität der Tests parametrisiert. Die Dauer der modifizierten Leistungstests wird auf die Dauer des jeweiligen Experiments festgelegt. Um reale Nutzer zu simulieren, werden mehrere Methoden angewandt:

> Das Verhalten der Nutzer wird über das Verhalten der Leistungstests, beziehungsweise der funktionalen Tests simuliert. Beide Tests bilden Beispielverhalten der Nutzer ab, sodass diese alle Anforderungen der Nutzer aus ihrer Perspektive testen können.

Die parametrisierte Intensität beeinflusst die Menge der simulierten Nutzer. Die Anzahl der Nutzer wird anhand des Produktivsystems festgelegt. Es werden auf dem Produktivsystem die Anzahl der Nutzer gemessen. Da nur die Anzahl der Nutzer während des Betriebs gemessen werden sollen, werden alle Datenpunkte aussortiert, bei denen eine Nutzerzahl von null vorliegt. Von den resultierenden Datenpunkten wird der Median genommen, sodass eine mittlere Nutzerzahl bestimmt wird. Anstelle des Mittelwerts wird hier der Median verwendet, sodass die mittlere Nutzerzahl gegen Ausreißer stabil ist. Der über den Median bestimmte Wert dient fortan als Richtwert für die Intensität der Nutzeranfragengenerierung. Je nach Typ des Experiments kann eine höhere Intensität gewählt werden. Jedoch führt eine überhohe Intensität in regelmäßigen und automatisierten Experimenten zu einem höheren Ressourcenverbrauch.

Die Komponenten des vollständigen Konzepts sind Abbildung 3.2 zu entnehmen.

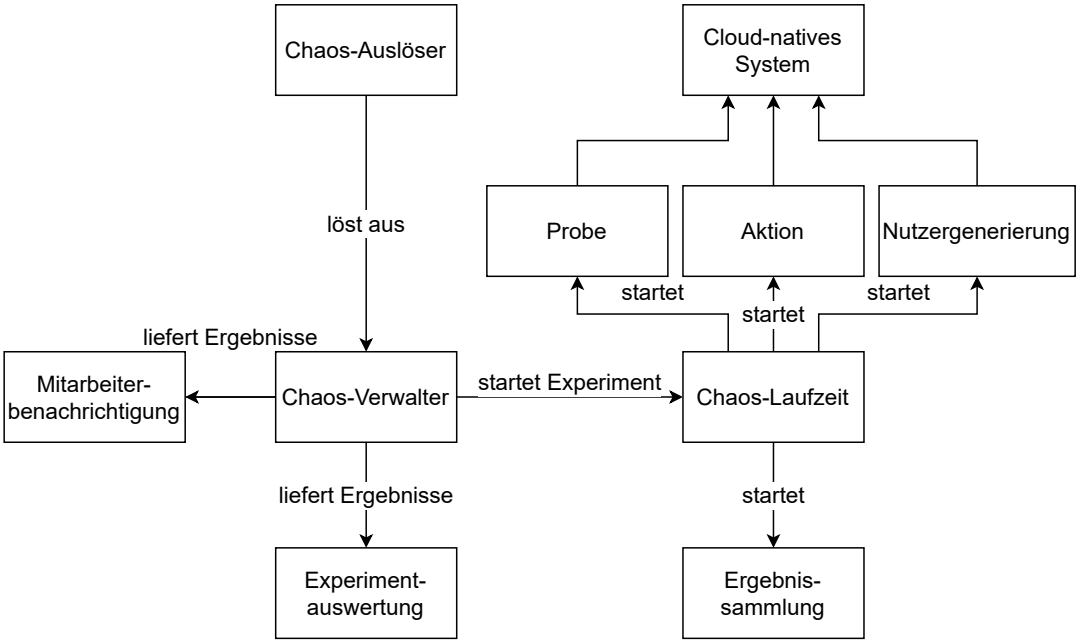

Abbildung 3.2: Komponenten des vollständigen Konzepts

Durch das Schaffen einer Vergleichbarkeit wird über ein nicht-Produktivsystem die Verfügbarkeit des Produktivsystems sichergestellt. Die Vergleichbarkeit wird über eine Nutzeranfragengenerierung geschaffen, welche nur für die Dauer des Experiments läuft, sodass eine Ressourceneffizienz erreicht wird. Zusätzlich zu der Nutzeranfragengenerierung

wird auf bisheriges automatisiertes Chaos-Engineering zurückgegriffen. Hierdurch wird eine Automatisierung erzielt, wodurch eine Zeiteffizienz erzielt wird.

Das Konzept erfüllt somit die Anforderungen der Kostenminimierung und Vergleichbarkeit und Qualitätssicherung des Produktivsystems. In der nächsten Sektion werden Varianten des entstandenen Konzepts genannt.

3.4 Varianten

Das davor aufgestellte Konzept ist nur eine von vielen Lösungen. Die an das Konzept gestellten Anforderungen lassen einen Spielraum für Variation. In dieser Sektion wird auf Variationen eingegangen.

Das davor eingeführte Konzept hat den Nachteil, dass keine realen Nutzer die Nutzeranfragen generieren, sondern dass diese programmatisch erzeugt werden. Hierdurch wird kein optimaler Wirkungsgrad in Bezug auf eine Vergleichbarkeit geschaffen, das es sich bei den generierten Anfragen um Simulationen von Nutzern handelt. Die Variationen lösen die Problematik, indem für die Nutzeranfragengenerierung reale Nutzer verwendet werden.

3.4.1 Nutzerteiler

Der Nutzerteiler fügt eine weitere Komponente in das Produktivsystem hinzu, welche die Nutzer aufteilt:

> Ein kleiner Anteil der Nutzer wird von dem Produktivsystem auf das nicht-Produktivsystem umgeleitet, sodass alle Anfragen der Nutzer auf dem nicht-Produktivsystem eintreffen. Der Anteil der Nutzer wird gesteuert, um die Belastung des nicht-Produktivsystems zu kontrollieren.
>
> Die Umleitung erfolgt programmatisch und nur für die Dauer eines Experiments.

Hierdurch würden mehrere Problematiken auftreten:

> Die Nutzer dürfen den Wechsel auf das nicht-Produktivsystem nicht bemerken. Hierfür wird ein komplexer Aufbau des (nicht-)Produktivsystems benötigt, da der

Zustand der arbeitenden Nutzer zwischen den Systemen synchronisiert werden muss, indem die Datenbanken z. B. zwischen den Systemen geteilt werden.

Theoretisch ist dieser Ansatz möglich, jedoch ist dies praktisch aus unternehmerischer Sicht nicht möglich. Die Software-Verträge, die ein Software-Hersteller mit einem Kunden abgeschlossen hat, erlauben es dem Software-Hersteller nicht, den Kunden spontan auf ein nicht-Produktivsystem umzuleiten.

3.4.2 Probekontovertrag

Die Problematik, dass der Software-Hersteller an den mit dem Kunden abgeschlossenen Vertrag gebunden ist, lässt sich lösen, indem ein neuer Vertrag abgeschlossen wird.

Der Kunde hat über den Vertrag nur Zugang zu dem nicht-Produktivsystem.

Über die Anzahl der verkauften Verträge wird die Belastung des nicht-Produktivsystems kontrolliert.

Die Variation hat folgende Nachteile:

Die Belastung des nicht-Produktivsystems kann nicht variabel angepasst werden, da nicht entsprechend viele Verträge spontan geschlossen werden.

Die Variation ist sowohl theoretisch als auch praktisch umsetzbar. Jedoch ist das Aufsetzen eines zweiten Vertrags aufwendig und schafft eine neue Vertragsbindung.

Beide genannten Variationen verwenden reale Nutzer, um Nutzeranfragen zur generieren. Hierdurch geht Flexibilität verloren, da eine Bindung zu den unvorhersehbaren Nutzern entsteht. Gleichzeitig wird ein erhöhter Realismus der Anfragen erzielt.

4 Implementierung

Nachdem nun ein Konzept für das effiziente Umsetzen von Chaos-Engineering auf einem nicht-Produktivsystem gegeben ist, gilt es das Konzept in die Tat umzusetzen. In der Praxis müssen weitere Punkte beachtet werden. Bei der Implementierung des Konzepts müssen Team-Grenzen berücksichtigt werden. Für die Erweiterung oder Veränderung von Team externen Komponenten muss mit weiteren Teams kommuniziert werden. Zudem muss beachtet werden, dass Ressourcen geteilt sein könnten. Bei (abnormaler) Nutzung der Ressourcen muss zuerst mit den weiteren Teilhabern abgestimmt werden.

In diesem Kapitel werden zuerst vorhandene Strukturen genannt und identifiziert. Danach folgt eine schrittweise Implementierung der für das Konzept benötigten Komponenten.

4.1 SAP AI Dienste

Das Konzept wird für die SAP AI Dienste umgesetzt. Die SAP bietet Software für die Steuerung von Geschäftsprozessen an[42]. Die AI Dienste sind ein Teil der angebotenen Cloud-Produkte und umfassen alle Produkte, die künstliche Intelligenz für die Steuerung und Optimierung von Geschäftsprozessen verwenden. Dabei folgen die Dienste einer bestimmten Struktur. Es gibt eine grundlegende Basis, welche künstliche Intelligenz mittels Machine Learning ermöglicht. Die Basis ist die ML Foundation. Alle weiteren AI Dienste bauen auf der ML Foundation auf und implementieren Service spezifische Businesslogik. Im Folgenden werden zwei grundlegende AI Dienste genannt, an welchen effizientes Chaos-Engineering umgesetzt wird.

Der erste Dienst, der auf der ML Foundation aufbaut, ist Business Entity Recognition (BER). BER erkennt Wirtschaftsobjekte in unstrukturierten Texten. Über BER können z. B. relevante Informationen aus E-Mails automatisiert extrahiert werden[43].

Der zweite Dienst ist Personalized Recommendation (PR). PR bietet Nutzern persönliche Empfehlungen basierend auf der Historie des Nutzers[44].

Sowohl BER als auch PR werden zwar als Dienste bezeichnet, bestehen jedoch selbst aus mehreren Diensten. Beide sind somit Anwendungen. Die beiden Anwendungen haben mehrere Abhängigkeiten. Einerseits haben sie eine Abhängigkeit zu der ML Foundation. Andererseits haben sie eine weitere Abhängigkeit zu grundlegenden Diensten, die z. B. eine Authentifizierung bereitstellen. Die Anwendungen sind zudem selbst Abhängigkeiten. Sie werden als Bausteine für weitere Anwendungen verwendet, die die bereitgestellte Funktionalität integrieren, um ihre eigene Businesslogik zu implementieren.

Beide AI Dienste werden von unterschiedlichen Teams verwaltet. Sowohl die Abhängigkeiten der AI Dienste, als auch die Anwendungen die von den AI Diensten abhängen, liegen außerhalb des eigenen Teams und werden von diesem nicht verwaltet.

Für beide AI Dienste wurde bereits ein rudimentäres Chaos-Engineering umgesetzt. Das bisher umgesetzte Chaos-Engineering zeichnet sich dadurch aus, dass an den genannten Anwendungen manuell experimentiert wird. Die Experimente erfüllen nur die notwendigen Aspekte des Chaos-Engineering und gehen darüber nicht hinaus. Der stabile Zustand der Systeme wird über Proben getestet und es werden Störungen an einem Ziel verursacht. Die Ergebnisse der Experimente werden gesammelt.

Als konkretes Ziel der Chaos-Experimente dienen die Instanzen der beiden Dienste, die in der Systemlandschaft `internal production` laufen. Die Systemlandschaft `internal production` ist ein nicht-Produktivsystem.

Die Experimente des bisher umgesetzten Chaos-Engineerings werden manuell gestartet. Auf Knopfdruck wird ein Experiment gestartet. Manche der Experimente werden über einen funktionalen Test oder Lasttest erweitert. Der ausgesuchte funktionale Test oder Lasttest wird manuell unmittelbar vor dem Experiment gestartet und unmittelbar nach dem Experiment gestoppt. Dies geschieht auch auf Knopfdruck. Die gesammelten Ergebnisse des beendeten Experiments werden im Anschluss manuell betrachtet. Da der Experimentierer nicht automatisch von Statusänderungen des Experiments benachrichtigt wird, muss der Experimentierer während des Experiments präsent bleiben, um eine Statusüberprüfung händisch und regelmäßig durchzuführen.

Zusätzlich sollen in der Praxis zu den Ergebnissen eines Experiments Screenshots von der Experimentauswertung genommen werden. Die Screenshots dienen für Kollegen als Beweis dafür, dass die Experimente ein System gestört haben, es aber dennoch verfügbar

Konzeptuelle Komponenten	Praktische Komponenten
Chaos-Auslöser	Jenkins
Chaos-Laufzeit	ChaosToolkit
Chaos-Verwalter	CaaS
Ergebnissammlung	Screenshots
Experimentauswertung	CaaS Weboberfläche
Mitarbeiterbenachrichtigung	Slack
Nutzergenerierung	funktionale Tests / Lasttests

Tabelle 4.1: Zuweisung der praktischen Komponenten zu den konzeptuellen Komponenten

blieb. Die Screenshots werden momentan manuell genommen und sollen automatisiert werden.

Das bisher umgesetzte Chaos-Engineering war ein erster Versuch Chaos-Engineering umzusetzen und hat somit viele Aspekte nicht berücksichtigt, die in dem Konzept berücksichtigt werden. Das Konzept wird dadurch nicht erfüllt. Es wird zwar auf einem nicht-Produktivsystem experimentiert, jedoch ist keine Effizienz gegeben, da jegliche Schritte manuell durchgeführt werden. Zudem ist eine Vergleichbarkeit nur bedingt gegeben. Es werden zwar funktionale Tests und Lasttests verwendet, diese werden aber nicht für jedes Experiment verwendet. Zudem werden die Tests unverändert und nicht parametrisiert verwendet. Es kann somit z. B. nicht die Intensität und Anzahl der simulierten Nutzer festgelegt werden.

Da dennoch Chaos-Engineering umgesetzt wurde, können Komponenten des bisherigen Chaos-Engineerings wiederverwendet werden, um Einarbeitungszeit und Implementierungszeit zu sparen. Bisher verwendete Komponenten werden in der nächsten Sektion identifiziert.

4.2 Identifizierung der Komponenten

In dieser Sektion werden die bisher verwendeten Komponenten identifiziert, die zu einem rudimentären Chaos-Engineering beigetragen haben. Die bereits verwendeten Komponenten werden wiederverwendet, um eine Kosteneffizienz zu erzielen. In Tabelle 4.1 ist eine Zuweisung der bereits verwendeten Komponenten zu den konzeptuellen Komponenten zu sehen.

4.2.1 Chaos as a Server (CaaS)

CaaS ist ein Werkzeug zur Verwaltung von Experimenten. Die Experimente fokussieren sich auf Dienste, die auf Cloud Foundry (CF)- oder K8s-Systemen laufen. In CaaS können Experimente angelegt werden. Dabei werden Vorlagen und Experimentschnipsel bereitgestellt, sodass das Anlegen vereinfacht wird. Die Experimente können in einem GitHub Repository liegen, sodass eine Versionierung möglich ist. Die Ergebnisse und Logs der Experimente werden in CaaS gesammelt und können dort visualisiert werden. Einzelne Experimente können von Außen gestartet werden.

CaaS ist ein internes Werkzeug und wird für die Verwaltung der Experimente verwendet. Da CaaS intern ist, kann bei Problemen direkt mit den zuständigen Teams kommuniziert werden, um schnellstmöglich eine Lösung zu finden. Bei CaaS handelt es sich um den *Chaos-Verwalter*. Der *Chaos-Verwalter* muss Befehle zur Ausführung von Experimenten entgegennehmen und die Ergebnisse zurückliefern können. CaaS erfüllt beide Aspekte. Für eine Automatisierung der Experimente muss die bereitgestellte Schnittstelle verwendet werden, um Experimente zu starten und die Ergebnisse auszulesen.

Die in CaaS gesammelten Ergebnisse können über eine minimale Oberfläche eingesehen werden. Hierdurch kann eine oberflächliche Auswertung vorgenommen werden. Die in CaaS integrierte Oberfläche dient somit als primitive *Experimentauswertung*.

CaaS stellt keine automatische Benachrichtigung bei Vollendung eines Experiments bereit. Diese Funktionalität kann jedoch innerhalb eines Experiments umgesetzt werden. Für das Ausführen eines einzelnen Experiments wird innerhalb von CaaS ChaosToolkit verwendet.

4.2.2 ChaosToolkit

ChaosToolkit ist ein quelloffenes Programm zur Ausführung von Chaos-Experimenten[45]. Die Experimente werden im JSON-Format spezifiziert und folgen einer definierten Struktur[46]:

> Jedes Experiment muss Methoden definieren, die linear von ChaosToolkit abgearbeitet werden. Die Methoden sind entweder Proben oder Aktionen. Über die definierten Methoden soll primär Chaos in einem System verrichtet werden. Die

Proben und Aktionen werden in dem Experiment unter dem Schlüssel `method` als Liste angegeben.

Um den stabilen Zustand eines Systems zu definieren, kann ein Experiment eine Liste aus Proben unter dem Schlüssel `steady-state-hypothesis` enthalten. Die Proben werden sowohl vor, als auch nach den Methoden ausgeführt. Scheitert mindestens eine Probe vor Ausführung der Methoden, wird das Experiment abgebrochen. Hierdurch wird Chaos nur auf einem (noch) „gesunden“ System verrichtet.

Das über die Methoden verrichtete Chaos kann das System so weit zerstören, sodass es sich nicht selbst reparieren kann. Um das System nach den Methoden zu reparieren, werden über den Schlüssel `rollbacks` Aktionen definiert, die das System wieder in den ursprünglichen Zustand befördern.

Die Aktionen und Proben sind parametrisiert. Um vermehrt auftretende Parameter, wie z. B. die ID einer Anwendung zentral zu definieren, können die Parameter in der Sektion `configuration` definiert werden. Die dort definierten Variablen können anstelle von Parametern über folgende Notation verwendet werden: `${name_der_variable}`.

Die Konfigurationssektion ist jedoch nicht für Geheimnisse geeignet, da die Werte als Klartext im Experiment gespeichert werden. Zugangsdaten und andere Geheimnisse werden über die Sektion `secrets` angegeben. Die Geheimnisse werden in anderen Anwendungen sicher gespeichert, werden in der Sektion referenziert und können fortan als Variable verwendet werden.

Zuletzt können in einem Experiment weitere besondere Steueraktionen definiert werden, die keinen Einfluss auf das Zielsystem haben. Die Steueraktionen werden zu bestimmten Zeitpunkten im Experiment aufgerufen. Über die Steueraktionen fügt CaaS Loggingfunktionalität zu ChaosToolkit hinzu, indem nach dem Experiment die erstellte Log zu CaaS hochgeladen wird.

Die in CaaS definierten Experimente entsprechen einem ähnlichen Format. Die Schlüssel der Experimentsektionen werden in der Notation „Camel Case“ angegeben und enden auf `Spec`. Beim Ausführen eines Experiments transformiert CaaS das eigene Experimentformat in das ChaosToolkit Experimentformat. Zusätzlich wird eine Steueraktion

hinzugefügt, um die Loggingfunktionalität zu ermöglichen. Da jegliche Experimente in dem Format von CaaS definiert werden, werden alle Experimente fortan in dem Format auch angegeben.

Da ChaosToolkit Proben und Aktionen startet, entspricht ChaosToolkit der Komponente *Chaos-Laufzeit*.

4.2.3 Jenkins

In Jenkins werden Pipelines zur Automatisierung von Aktionen definiert[47]. Die Pipelines können automatisch in einem Intervall ausgeführt werden. Zudem stellt Jenkins eine Schnittstelle bereit, über welche Pipelines gestartet werden können, sowie die Ergebnisse eingeholt werden können.

In Jenkins sind Pipelines für funktionale Tests für die Anwendungen BER und PR definiert, die in einem Intervall automatisch ausgeführt werden. Hierdurch wird sichergestellt, dass die Anwendungen ihre geforderte Funktionalität einhalten. Die in Jenkins definierten Pipelines, die funktionale Tests ausführen, entsprechen somit der Komponente *Nutzergenerierung*.

Da Jenkins Pipelines automatisch ausführen kann, kann Jenkins zudem als *Chaos-Auslöser* dienen, indem die in CaaS definierten Experimente über eine Schnittstelle ausgelöst werden.

4.2.4 Lasttests

Als qualitätssichernde Maßnahmen implementieren die Teams der AI Dienste Lasttests für ihre eigenen Dienste. Die Lasttests verwenden hauptsächlich die Werkzeuge Locust und JMeter. Die Lasttests dienen auch als *Nutzergenerierung*. Zur Bereitstellung der Lasttests stellen die Teams Docker-Images bereit, die in einem beliebigen System installiert werden können, welches eine Containerd-Laufzeit hat.

Die Lasttests sind aufgrund ihrer Architektur schwerer zu integrieren als die über Jenkins laufenden funktionalen Tests. Dafür bieten die Lasttests eine bessere *Nutzergenerierung*. Da die *Nutzergenerierung* der funktionalen Tests dennoch gut ist, werden zuerst die funktionalen Tests integriert, danach folgen die Lasttests.

4.2.5 Slack

Slack ist eine von mehreren verwendeten Nachrichtenplattformen[48]. Im Gegensatz zu z. B. E-Mails eignet sich Slack für Kurznachrichten, weshalb es sich für Statusnachrichten eignet[49]. Slack wird bereits in den über Jenkins ausgeführten funktionalen Tests verwendet, um Mitarbeiter zu benachrichtigen, wenn ein Test fehlschlägt. Slack kann somit als Komponente *Mitarbeiterbenachrichtigung* verwendet werden.

Wie den vorherigen Sektionen zu entnehmen ist, sind bereits viele diverse Werkzeuge in Gebrauch, die als Komponenten des Konzepts verwendet werden könnten. Um Zeit für die Suche und Einarbeitung nach neuen Werkzeugen zu sparen, werden die oben genannten Werkzeuge wiederverwendet. Die Komponenten des praktisch umgesetzten Konzepts werden in Abbildung 4.1 dargestellt. Für einen Vergleich kann Abbildung 3.2 betrachtet werden.

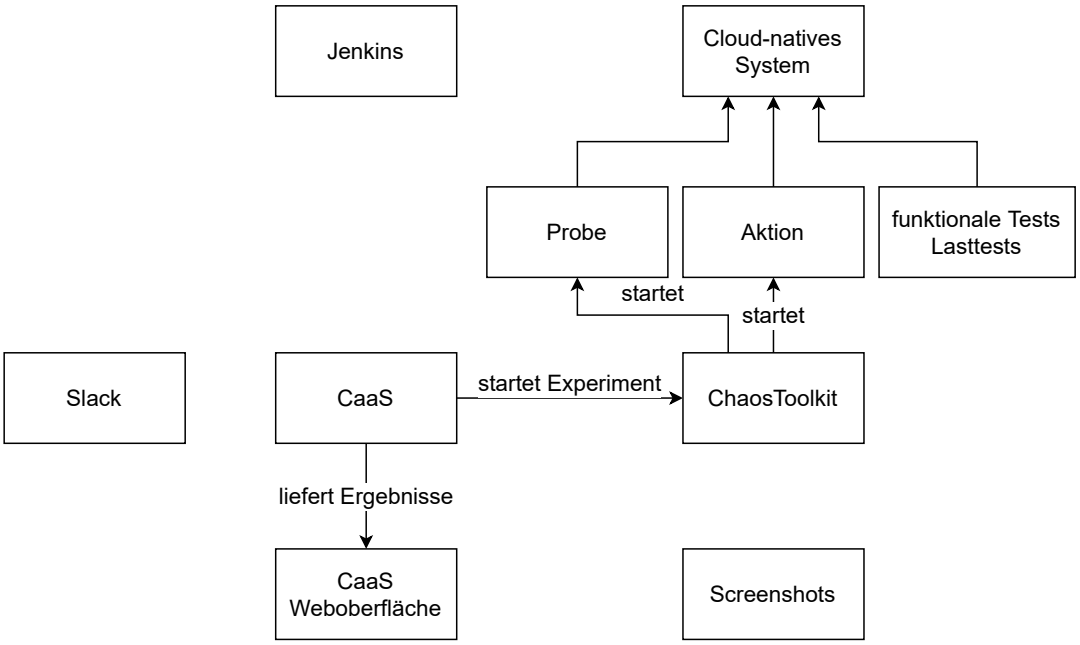

Abbildung 4.1: Komponenten des vollständigen Konzepts

In dem Diagramm wurden die theoretischen Komponenten durch die in der Praxis bereits verwendeten Komponenten ausgetauscht. Zu bemerken sind die fehlenden Beziehungen zwischen den meisten Komponenten. Die Beziehungen geben bereits automatisierte Prozesse an. Da CaaS als *Chaos-Verwalter* ein Experiment über ChaosToolkit startet, welches die *Chaos-Laufzeit* ist, ist die Beziehung `startet Experiment` hier anzufinden. Da

ChaosToolkit selbstständig Proben und Aktionen ausführt, sind diese in dem Diagramm auch anzufinden. Andere Beziehungen wurden noch nicht implementiert.

Es ist zudem zu bemerken, dass aufgrund von Regulationen Screenshots der Experimentauswertung gemacht werden müssen. Daher wurde die praxisbedingte Komponente *Screenshots* hinzugefügt. Das Sammeln und Nehmen der Screenshots fällt unter die Komponente *Ergebnissammlung*. Hierfür kann keine Komponente wiederverwendet werden, da keine Komponente im Einsatz ist, die automatisiert Screenshots nehmen kann.

4.3 Architektur

Aufgrund der Zeiteffizienz werden vorhandene Werkzeuge und Komponenten wiederverwendet. Die Komponenten wurden in der letzten Sektion identifiziert. Wie der Grafik 4.1 zu entnehmen ist, fehlen Beziehungen zwischen den Komponenten. Um die Beziehungen in den kommenden Sektionen zu implementieren, müssen architekturelle Details berücksichtigt werden.

4.3.1 Authentifizierung

Jegliche genannten Komponenten sind nicht für die Öffentlichkeit zugänglich und dementsprechend abgesichert. Um die diversen Komponenten zu verwenden, wird ein gültiger Nutzer inklusive seines Passworts benötigt. Der Nutzer kann ein regulärer Nutzer, oder auch ein technischer Nutzer sein.

Um Komponenten automatisiert zu verwenden, muss eine Authentifizierung an der Komponente stattfinden, sodass Aktionen zugelassen werden. Da die Aktionen über die HTTP-Schnittstellen geschehen, muss eine Authentifizierung über HTTP stattfinden. Ansonsten wird eine Antwort `401 Unauthorized` oder `403 Forbidden` zurückgesendet[50].

Um eine einfache Automatisierung zu erlauben, ermöglichen die Komponenten die einfache HTTP-Authentifizierung. Jegliche notwendigen Anmeldedaten werden über den HTTP-Header `Authorization` übermittelt[51]. Bei einer Anmeldung über Nutzername und Passwort wird ein Wert zusammengesetzt aus `Basic ` und dem Base64 kodierten Text

`Nutzer:Passwort` übertragen[52]. Die Dienste stellen teilweise eine Anmeldung über nur einen Token bereit. Dieser kann in dem Dienst selbst generiert werden und wird nur für die HTTP-Schnittstelle verwendet. Der Token kann nicht für die Anmeldung über die Weboberfläche der Dienste verwendet werden. Bei einer Authentifizierung über einen Token wird als Wert des HTTP-Headers `Authorization` der Wert zusammengesetzt aus `Bearer` und dem Token übertragen[53].

Für einen ersten Test der Schnittstellen kann der eigene Nutzername und das eigene Passwort verwendet werden. Für die weitere Entwicklung müssen aber technische Nutzer oder Tokens verwendet werden. Bei Verwendung echter Nutzer entsteht das Problem, dass die Anwendung dann von einigen wenigen Nutzern abhängt. Wenn die Nutzer z. B. das Unternehmen verlassen und die Nutzer gelöscht werden, dann scheitert die Authentifizierung. Da aber das Anfordern neuer technischer Nutzer, oder das Verwenden bisheriger technischer Nutzer zeitaufwendig sein kann, werden vorerst die eigenen Nutzerdaten verwendet, da dadurch kein weiterer direkter Zeitaufwand entsteht.

4.3.2 Erreichbarkeit

Eine Authentifizierung an allen notwendigen Komponenten ist nur möglich, wenn zuerst überhaupt eine Verbindung zu den Komponenten aufgebaut werden kann. Es muss zwischen zwei Netzwerken unterschieden werden. Es gibt das öffentliche Netzwerk und das interne unternehmenseigene Netzwerk. Das unternehmenseigene Netzwerk kann als Mitarbeiter nur über eine Virtual Private Network (VPN) erreicht werden. Aus dem internen Netzwerk können alle Dienste des öffentlichen Netzwerks ohne weiteres erreicht werden.

Für eine automatisierte Lösung ist die Verwendung einer VPN nicht möglich. Um Dienste innerhalb des internen Netzwerks zu erreichen, muss das anfragende Programm selbst in dem internen Netzwerk liegen. Einzelne Programme sind jedoch nur Teil eines größeren Systems, welches entweder im internen Netzwerk liegt, oder von der Öffentlichkeit aus erreichbar ist. Falls eine Komponente nun Teil des internen Netzwerks werden muss, dann muss die Komponente Teil eines neuen Systems werden, welches innerhalb des internen Netzwerks liegt.

Um zu betrachten, in welchem Netzwerk die Komponenten liegen, werden die Komponenten anhand ihres Systems gruppiert. Eine Systemübersicht ist in Abbildung 4.2 zu sehen. Weiße Rechtecke stellen Systeme dar.

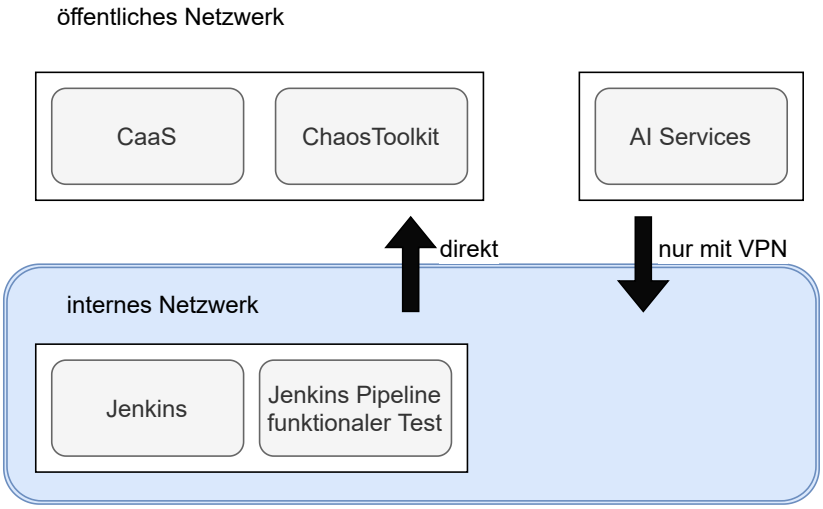

Abbildung 4.2: Komponenten gruppiert anhand ihres Systems

Nur das System, in welchem sich Jenkins befindet, ist Teil des internen Netzwerks. Alle weiteren Systeme sind öffentlich erreichbar. Es ist zudem zu bemerken, dass es sich bei den AI Diensten nicht um ein einzelnes System handelt, sondern um mehrere Systeme, auf welche die unterschiedlichen Dienste verstreut sind.

Wie aus Abbildung 3.2 bekannt ist, wird auf Jenkins zugegriffen, um funktionale Tests zu starten. Da die zugreifende Komponente ChaosToolkit Teil des öffentlichen Netzwerks ist, ist eine Verbindung nicht möglich. Es könnte entweder Jenkins in das öffentliche Netzwerk verschoben werden, oder ChaosToolkit in das interne Netzwerk. Es kommt hinzu, dass ChaosToolkit voll von CaaS abhängig ist. Zu Beginn gibt es keine ChaosToolkit-Instanz. Für jedes Experiment, welches in CaaS gestartet wird, wird von CaaS eine neue ChaosToolkit dynamisch erstellt.

Wenn Jenkins in das öffentliche Netzwerk verschoben werden würde, dann müssten alle Abhängigkeiten, die selbst im internen Netzwerk sind, rekursiv in das öffentliche Netzwerk verschoben werden. Andererseits müssen alle davon abhängenden Komponenten auch rekursiv in das interne Netzwerk verschoben werden, wenn ChaosToolkit in das interne Netzwerk verschoben wird.

Allgemein ist das Verschieben von Komponenten von Systemen auf andere Systeme kurzzeitig nicht umzusetzen, da die Komponenten mehrere Anteilhaber haben. Eine Verschiebung ist somit mit einem hohen Aufwand verbunden. Anstelle eine Komponente zu verschieben, wird deshalb die Komponente dupliziert. Alle Anteilhaber der Komponente arbeiten wie gewohnt, während die zweite Instanz der Komponente frei verwendet werden kann.

Um die Erreichbarkeitsproblematik zu lösen, wird ein weiteres System verwendet, welches sich im internen Netzwerk befindet. CaaS wird in dem neuen System installiert und hat dadurch Zugang zu dem Jenkins-System. Da CaaS selbst nur von Jenkins aus erreichbar sein muss, gibt es hier keine weiteren Probleme. Wenn nun über die neue CaaS-Instanz ein Experiment gestartet wird, dann wird eine neue ChaosToolkit-Instanz für die Dauer des Experiments gestartet, welche Zugang zu Jenkins hat. Die ursprüngliche CaaS-Instanz bleibt unberührt. Das alte CaaS-System wird als `MultiAZ` bezeichnet und das neue System als `StuffSS`.

Nachdem die architekturellen Probleme gelöst wurden, werden im Folgenden die fehlenden Beziehungen zwischen den Komponenten implementiert.

4.4 Funktionale Tests

Die Implementierung beginnt mit der Automatisierung der funktionalen Tests. Es wird bei den funktionalen Tests begonnen, da die funktionalen Tests gemeinsam mit den Lasttests durch das Schaffen einer Vergleichbarkeit im Zentrum der Arbeit liegen. Es wird in dieser Sektion zuerst betrachtet, an welcher Stelle für eine Implementierung angesetzt werden muss. Im Anschluss folgt das Aufsetzen einer geeigneten Entwicklungsumgebung. Zuletzt wird an der identifizierten Stelle angesetzt, um funktionale Tests in den Experimentablauf zu integrieren.

4.4.1 Ansatz

Die funktionalen Tests sollen vor Verrichtung des Chaos gestartet werden. Nach Start der Tests wird das Chaos verursacht, sodass die Tests kontinuierlich im Hintergrund laufen. Die Tests haben einen indirekten Einfluss auf das Zielobjekt des Experiments,

da die Tests Anfragen an das Zielobjekt senden. Nach Verrichtung des Chaos sollen die funktionalen Tests stoppen. Das Ergebnis der Tests soll das Ergebnis des kompletten Experiments beeinflussen. Wenn ein funktionaler Test scheitert, dann scheitert das komplette Experiment.

Da die funktionalen Tests das Experiment direkt beeinflussen und Teil des Experiments werden sollen, werden die Tests in die Experimentdefinition integriert. ChaosToolkit liest die von CaaS transformierte Experimentdefinition ein und arbeitet die definierten Bausteine ab. ChaosToolkit soll als Teil der normalen Abarbeitung den erstellten Code aufrufen, sodass die gewünschte Logik umgesetzt wird.

Es werden zwei Bausteine benötigt. Der erste Baustein startet kontinuierlich Tests, blockiert aber den Experimentablauf nicht. Der zweite Baustein holt zuletzt die Ergebnisse ein und blockiert, sodass bei Abschluss des Experiments alle Tests beendet wurden. Der erste Baustein wird als ChaosToolkit-Aktion umgesetzt, da Aktionen einen Einfluss auf das Zielobjekt haben. Der zweite Baustein wird als Probe umgesetzt, da die Ergebnisse das Ergebnis des Experiments beeinflussen sollen.

ChaosToolkit ist in Python programmiert und ist durch Python erweiterbar. In der Experimentdefinition können als Probe, Aktion oder Steueraktion Python-Module, beziehungsweise Python-Funktionen angegeben werden. Die aufgerufenen Module müssen in der Python-Laufzeit des Experiments vorhanden sein und werden dann von ChaosToolkit parametrisiert aufgerufen.

Es wird ein neues GitHub-Repository angelegt, sodass Versionierung verwendet wird. In dem neuen Repository wird ein Python-Modul angelegt, welches alle benötigten Proben, Aktionen und Steueraktionen enthält. Das Python-Modul muss in die Experimentlaufzeit integriert werden, sodass die entwickelte Funktionalität von ChaosToolkit aufgerufen werden kann.

4.4.2 Entwicklungsumgebung

In dieser Sektion wird genau erläutert, an welchem Ort der Code sich befindet und wie der Code anhand eines Experiments getestet wird.

Für die Entwicklung des Codes werden zwei GitHub-Repositories verwendet. Das erste Repository wird produktiv verwendet und ist Teil der GitHub-Organisation *Chaos-Drivers*. Die Organisation *Chaos-Drivers* enthält weitere Projekte, die ChaosToolkit anhand von Python-Modulen erweitern und ist somit der offizielle Ablageort für ein solches Projekt. In der Organisation lässt sich zudem ein Repository inklusive einer fertigen Dateistruktur automatisch generieren. Es wird das Repository `module_background_test-_extensions` generiert, in welchem alle ChaosToolkit-Erweiterungen entwickelt werden.

Das erste Repository ist nur für Produktivcode geeignet, da durch die Generierung des Repositories automatisch Konfigurationen angelegt wurden, sodass das Python-Modul automatisch bei Änderungen gebaut wird. Das gebaute Python-Modul landet in einer internen Registry, in welcher die gebauten Python-Module abgelegt werden, sodass die Module in internen Anwendungen installiert werden können.

Um kleine Änderungen schnell zu testen, wird ein *Fork* erstellt. Sobald signifikante Funktionalität auf dem *Fork* implementiert und getestet wurde, werden die Änderungen auf das Repository in der Organisation *Chaos-Drivers* gebracht.

`StuffSS` ist ein K8s-System. Die CaaS-Dienste laufen in K8s in einem Namespace als mehrere `Pods`. Wenn in CaaS ein Experiment gestartet wird, dann wird in einem Namespace in `StuffSS` ein weiterer `Pod` angelegt, in welchem sich das transformierte Experiment befindet. In dem `Pod` wird ChaosToolkit ausgeführt, um das Experiment abzuarbeiten.

Um die Änderungen zu testen, wird ein Testexperiment in einem besonderen Modus manuell gestartet. Eine Instanz von ChaosToolkit wird erstellt, das Experiment wird jedoch nicht abgearbeitet. Hierdurch kann das Python-Modul manuell auf der neuen Instanz installiert werden. Nach Installation wird das Experiment manuell fortgeführt. Hierfür wird Zugang zum CaaS-System `StuffSS` benötigt. Da `StuffSS` ein K8s-System ist, kann das Kommandozeilenprogramm `kubectl` verwendet werden, um mit dem System zu interagieren.

Zuerst wird der Code des Python-Moduls als Zip-Datei heruntergeladen. Über `kubectl cp modul.zip KUBERNETES_POD_ID:/workdir/ -n caas` wird die Datei auf die laufende ChaosToolkit-Instanz kopiert. Die Argumente `-n caas` werden benötigt, da sich die Instanz im K8s-Namespace `caas` befindet. Im Anschluss wird auf der Instanz eine interaktive Shell über `kubectl exec -it KUBERNETES_POD_ID`

`-n caas - /bin/sh` gestartet. Die Shell arbeitet direkt im Ordner `/workdir`. In dem Ordner befindet sich neben der kopierten `module.zip` das Experiment am Pfad `./experiment/experiment.json`.

Das Modul wird folgenderweise installiert. Über `unzip module.zip` wird zuerst die Zip entpackt. In dem nun vorhandenen Ordner wird das Modul über `python setup.py sdist bdist_wheel` gebaut. Im Unterordner `dist` des entpackten Projekts ist im Anschluss eine auf `.whl` endende Datei zu finden, welche über `pip install ./dist/DATEINAME.whl` installiert wird. Die Entwicklungsversion des Moduls ist schlussendlich auf der ChaosToolkit-Instanz installiert und kann von ChaosToolkit verwendet werden. Der tatsächliche Experimentdurchlauf wird im Anschluss über `chaos run ./experiment/experiment.json` gestartet. Die daraus resultierende Log wird in dem Ordner `/workdir` gespeichert. Über die Log wird zuletzt analysiert, ob der experimentelle Code erfolgreich war.

4.4.3 Allgemeine Bibliothek

Um die zu implementierende Logik wiederverwenden zu können, wird der Code, der die Logik umsetzt, in ein geteiltes Untermodul abgelegt. Das Untermodul `utilities` enthält in jeder Python-Datei eine geteilte Funktionalität. In der Datei `background_test.py` wird das Ausführen von funktionalen Tests abgelegt. Das Ausführen von funktionalen Tests ist jedoch nicht allgemein genug, um dafür eine geteilte Bibliothek anzulegen. Was jedoch wiederverwendbar ist, ist das Ausführen einer beliebigen Aktion kontinuierlich im Hintergrund. Dabei soll zudem angegeben werden können, ob die Aktion nur einmalig oder kontinuierlich ausgeführt wird. Die beliebige Aktion im Falle der funktionalen Tests ist das Ausführen eines Tests über Jenkins.

Die in der Datei befindliche Klasse `BackgroundTest` repräsentiert eine beliebige Aktion, die im Hintergrund ausgeführt wird. Nach Instanziierung wird die Aktion über die Methode `start` gestartet. Dabei werden die Aktion und für die Aktion benötigte Parameter und Zugangsdaten übergeben. Der Parameter `once` bestimmt, ob die Aktion nur einmalig läuft. Die Methode `start` startet dann einen neuen Thread, welcher die statische Methode `run_tests` ausführt und `run_tests` alle Parameter übergibt.

`run_tests` ist in 4.1 zu sehen.

```
1     @staticmethod
2     def run_tests(in_queue: Queue, out_queue: Queue, ↙
          ↳ configuration: Dict[str, Any], secrets: Secrets, ↙
          ↳ test_runner, once: bool):
3         results: List[Any] = list()
4         if once:
5             result = test_runner(configuration, secrets)
6             results.append(result)
7             in_queue.get()
8         else:
9             while in_queue.empty():
10                result = test_runner(configuration, secrets)
11                results.append(result)
12
13        out_queue.put(results)
```

Listing 4.1: Ausführen einer Aktion im Hintergrund

Je nach Parameter `once` wird nun die Aktion `test_runner` einmalig oder wiederholt aufgerufen. Das Ergebnis wird zu der Liste `results` hinzugefügt. Sodass der Thread mit dem Haupt-Thread kommunizieren kann, werden zwei synchrone Schlangen verwendet. Über `in_queue` empfängt der neue Thread Nachrichten, über `out_queue` versendet der neue Thread Nachrichten. Die Ergebnisse der Aktionen werden dementsprechend zuletzt über `out_queue` an den Haupt-Thread übermittelt. Der neue Thread bekommt über `in_queue` mitgeteilt, ob dieser das Ausführen der Aktionen beenden soll.

Das Beenden der kontinuierlichen Aktionen wird über die Methode `finish` ausgelöst, welche zum Beenden den Wert `True` in die besagte Schlange `in_queue` legt. Da `finish` im Anschluss auf die Ergebnisse wartet, blockiert die Methode, bis die aktuell ausgeführte Aktion beendet ist und ihr Ergebnis liefert.

4.4.4 ChaosToolkit Aktionen und Proben

Um funktionale Tests vom Experiment aus starten zu können, wird ein Untermodul `common` angelegt. In dem Untermodul werden alle für ChaosToolkit verfügbaren Objekte angelegt. Nach ChaosToolkit-Konvention werden die Dateien `probes.py` und

`actions.py` angelegt, welche die Proben und Aktionen enthalten. In `actions.py` wird die in 4.2 einsehbare Funktion angelegt.

```
1 def trigger_test(module: str, func: str, name: str, once: bool ↙
      ↳ = False, arguments: Dict[str, Any] = {}, secrets: Secrets ↙
      ↳ = {}):
2     module = import_module(module)
3     test_runner = getattr(module, func)
4     amount = "once" if once else "continuously"
5     logger.debug(f"Triggering test {module}.{func} {amount}")
6     start_background_test(name, arguments, secrets, ↙
          ↳ test_runner, once)
```

Listing 4.2: ChaosToolkit Aktion trigger_test

Die Funktion nimmt nur Parameter entgegen, die über das JSON-Format übergeben werden können. Es kann also keine Python-Funktion übergeben werden, um die für eine kontinuierliche Hintergrundaktion benötigte Aktion zu definieren. Es wird somit der Name und das Modul der Funktion angegeben, sodass das Modul über `import_module` geladen wird. Die Funktion wird über `getattr(module, func)` erhalten. Letztendlich werden alle verarbeiteten Parameter an `start_background_test` übergeben. `start_background_test` ist Teil der erstellten Bibliothek und erstellt eine neue Instanz von `BackgroundTest` und gibt alle Parameter an die Methode `start` weiter.

Genauso wie in `actions.py` ist in `probes.py` auch eine Funktion vorhanden, die eine Funktionalität des `BackgroundTest` umschließt. Während `start_background_test` eine Aktion im Hintergrund startet, stoppt die in `probes.py` definierte Funktion `get_test_results` die kontinuierliche Aktion und liefert die Ergebnisse zurück. Sodass genau die kontinuierliche Aktion gestoppt werden kann, die über die Aktion `start_background_test` gestartet wurde, werden die kontinuierlichen Aktionen benannt. Anhand des Namens werden die kontinuierlichen Aktionen in einer globalen Zuweisungstabelle eingepflegt.

4.4.5 Automatisierung von Jenkins

Über den davor gezeigten Code können bereits beliebige Aktionen einmalig oder kontinuierlich ausgeführt werden. Zusätzlich können die Aktionen in Chaos-Experimenten

eingebunden werden, sodass die Aktionen bei Start des Experiments gestartet werden und zuletzt gestoppt werden. In diesem Abschnitt wird erläutert, inwiefern das Starten einer Jenkins-Pipeline implementiert wird, sodass die funktionalen Tests gestartet werden.

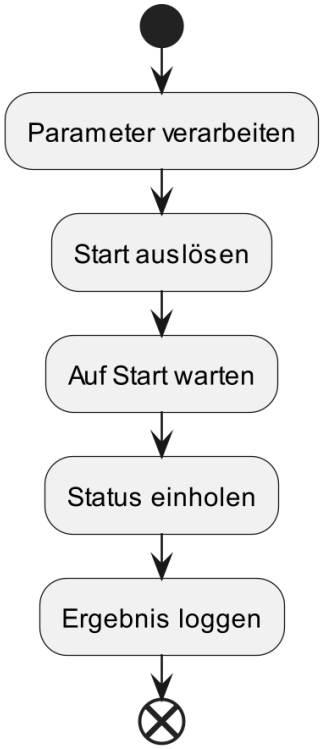

Abbildung 4.3: Automatisierung von Jenkins

Um das Projekt übersichtlich zu halten, wird für jede implementierte kontinuierliche Aktion ein weiteres Untermodul angelegt. In dem Untermodul `jenkins` liegt in der Funktion `jenkins_test` die neue kontinuierliche Aktion. Die neue Funktion folgt dem in Abbildung 4.3 dargestellten Vorgehen.

Zuerst werden die von der Bibliothek überreichten Parameter verarbeitet. Die Parameter `configuration` und `secrets` sind Zuweisungstabellen von Attributnamen zu Attributwerten. Für das Starten einer Jenkins-Pipeline wird die Uniform Resource Locator (URL) der Pipeline benötigt. Für eine Authentifizierung wird ein Nutzer und ein Token ausgelesen. Die Jenkins-Pipeline ist parametrisiert. Die Parameter werden über die Attribute `arguments` ausgelesen.

Für eine genauere Steuerung der Anfragen an die Jenkins-Schnittstelle werden weitere Attribute ausgelesen. Für die optionalen Attribute sind Standardwerte vorhanden, sodass der Standardwert gewählt wird, wenn das jeweilige Argument in `configuration` oder `secrets` nicht gefunden werden konnte.

Die in 4.3 genannten Aktionen, die mit der Jenkins-Schnittstelle kommunizieren, werden in die Bibliothek `jenkins.py` ausgelagert, die sich in dem Untermodul `utilities` befindet. Hierdurch könnte das Starten einer Jenkins-Pipeline von weiterem Code wiederverwendet werden. Zum Starten einer Pipeline wird eine POST-Anfrage an `PIPELINE_URL/buildWithParameters` gesendet. Im Körper der Anfrage werden alle davor spezifizierten Parameter für die Pipeline übergeben. Als Ergebnis wird eine URL zu einem Jenkins-Job erhalten.

Ein Jenkins-Job repräsentiert eine zukünftig auszuführende Pipeline und wird vorerst in eine Schlange eingereiht. Sobald der Job am Anfang der Schlange angekommen ist, wird der Job von Jenkins abgearbeitet und es wird die Pipeline ausgeführt. Anhand des Jobs wird im nächsten Schritt eine GET-Anfrage an `JOB_URL/api/json` geschickt, um eine JSON-Repräsentation des Jobs zu erhalten. Aus der Repräsentation wird die URL der erstellten Pipeline-Instanz ausgelesen.

Zuletzt wird nach demselben Verfahren eine GET-Anfrage an `INSTANZ_URL/api/json` gesendet, um den Status der laufenden Instanz zu erfassen, sobald der Status verfügbar ist. Bis aus einem Job eine laufende Instanz erstellt wird und die laufende Instanz terminiert, wird jeweils Zeit benötigt. Die zwei GET-Anfragen werden deshalb in einer Schleife ausgeführt, bis entweder ein Ergebnis vorhanden ist, oder z. B. der Job abgebrochen wurde.

Nach Durchlauf der Pipeline wird im Anschluss eine Log über die Funktion `jenkins_test` erstellt. Hierfür wird aus der aktuellen Uhrzeit, dem Datum und der URL der Pipeline eine Unique Identifier (UID) erstellt. In der Datei `/workdir/UID.log` wird die Konsolenausgabe der Pipeline gespeichert. Die Konsolenausgabe wird über eine GET-Anfrage an `INSTANZ_URL/consoleText` erhalten. Wie auch alle vorherigen Anfragen wird ein Nutzer und Token benötigt.

Bei Transformation des definierten CaaS-Experiments in das von ChaosToolkit verwendete Format wird automatisch eine Steueraktion hinzugefügt, welche alle Logdateien

hochlädt, die sich im Ordner `/workdir/` befinden. Über den Mechanismus wird die Konsolenausgabe der Pipeline den Experimentergebnissen angehängt.

Die erstellte Probe liefert ChaosToolkit nach Ausführung eine Liste aus den Ergebnissen der kontinuierliche ausgeführten Aktionen. Damit die Ergebnisse den Verlauf des Experiments beeinflussen, muss nun entschieden werden, ob die Werte innerhalb des stabilen Zustands liegen. Hierfür wird innerhalb der Probe in der Experimentdefinition eine Toleranz definiert. Die Toleranz kann selbst eine Python-Funktion sein, die anhand einer Eingabe entscheidet, ob der stabile Zustand eingehalten wurde. Um dem Experimentdefinierer die Wahl zu geben, das Verhalten der Probe zu festzulegen, wurden mehrere Funktionen implementiert, die auf Listen arbeiten. Zuerst wurde das Untermodul `tolerances` erstellt. Das Untermodul enthält die Funktionen `any_is`, `all_are` und `none_is`. Die Funktionen nehmen eine Liste entgegen und gleichen die Werte der Liste mit einem überreichten Wert ab. Dabei wird die Gleichheit der Elemente jeweils mit den Listenoperationen $\exists$, $\forall$ und $\nexists$ durchgeführt.

4.4.6 Modifikation eines Experiments

Da nun alle für die funktionalen Tests benötigte Funktionalität umgesetzt wurde, müssen die bereitgestellten Funktionen des Python-Moduls in ein vorhandenes Experiment integriert werden. Zuerst wird die entwickelte Aktion an den Beginn der definierten Methoden hinzugefügt. Ein Ausschnitt der Aktionsdefinition ist in 4.3 zu sehen.

```
1  "name": "trigger jenkins pipeline",
2  "type": "action",
3  "secrets": [
4      "jenkins"
5  ],
6  "background": false,
7  "provider": {
8      "type": "python",
9      "module": "background_test_extensions.common.actions",
10     "func": "trigger_test",
11     "arguments": {
12         "name": "${jenkins_test_name}",
13         "func": "jenkins_test",
```

```
14          "module": "background_test_extensions.jenkins",
15          "arguments": {
16              "job": "${jenkins_url}",
17              "max_queue_retry_count": 0,
18              "max_build_retry_count": 0,
19              "retry_interval": 1000,
20              "verify": false,
21              "user": "${username}",
22              "token": "${token}",
23              "arguments": {
```

Listing 4.3: Definition eines Pipelinestarts in einem Experiment

In dem Ausschnitt kann gesehen werden, wie die Python-Funktion `trigger_test` des Moduls `background_test_extensions.common.actions` angegeben wird. Alle Parameter der Funktion sind in dem äußeren Objekt `arguments` definiert. Es fällt dabei auf, dass in Zeile 12 der Wert nicht direkt übergeben wird. Vielmehr wird eine Variable angegeben, die in der Konfiguration oder in den Secrets angegeben wurde. Über die Variablen kann auf alle Attribute der über die in `secrets` angegeben Secrets zugegriffen werden. Die in der Liste angegebenen Secrets müssen dabei in der Sektion `secretsSpec` angegeben sein. Die weiteren Parameter für die Jenkins-Pipeline sind in dem innersten Objekt `arguments` angegeben, welches hier aufgrund der Länge nicht weiter angegeben wird.

Über eine ähnliche Syntax wird die implementierte Probe der Sektion `steadyStateSpec` angehängt. Als Attribut `tolerance` wird dabei eine weitere Probe definiert, die auf die Funktion `all_are` des Untermoduls `tolerances` verweist. Es wird als erwarteter Wert `SUCCESS` definiert. Hierdurch wird erzielt, dass während des Chaos eine Jenkins-Pipeline kontinuierlich ausgeführt wird und die Ergebnisse als Liste gesammelt werden. Wenn alle Pipeline-Durchläufe erfolgreich waren, dann ist das Experiment auch erfolgreich.

4.5 Jenkins

Nachdem in der letzten Sektion eine Jenkins-Integration für ChaosToolkit geschaffen wurde, wird in dieser Sektion eine CaaS-Integration für Jenkins geschaffen. Über die

Integration sollen Chaos-Experimente automatisiert aus Jenkins heraus gestartet werden. Dafür wird das folgende Konzept verwendet.

4.5.1 Konzept

CaaS stellt eine Schnittstelle bereit, über welche Experiment gestartet werden können. Zudem lassen sich die Ergebnisse der Experimente über die Schnittstelle einholen.

In Jenkins-Pipelines kann in einzelnen Schritten Code der Skriptsprache Groovy aufgerufen werden. Sodass Experimente über die Schnittstelle gestartet werden können, wird eine Bibliothek in Groovy geschrieben. Die Bibliothek kommuniziert mit der CaaS-Schnittstelle und verarbeitet die Ergebnisse.

Die Pipeline ruft in den einzelnen Schritten die entworfene Bibliothek auf und löst dadurch das Experiment aus. Da für die Kommunikation mit der Schnittstelle eine Bibliothek entworfen wird, können beliebig viele Experimente in einer Pipeline gestartet werden.

4.5.2 Groovy-Bibliothek

Um mit der CaaS-Schnittstelle kommunizieren zu können, muss zuerst eine Authentifizierung und Identifizierung eingerichtet werden. Sodass die Groovy-Bibliothek sich an CaaS authentifizieren kann, wird ein Token von CaaS benötigt. Der Token wird in der Weboberfläche aus einem Untermenü mit Details zu der Instanz kopiert. Die Informationen sind nur verfügbar, wenn der Nutzer an der Weboberfläche angemeldet ist.

Um eine Identifizierung einzurichten, wird anhand eines bereitgestellten Python-Skripts aus der URL der Jenkins-Instanz ein weiterer Token generiert. Der Token wird über die Weboberfläche von CaaS registriert.

Im nächsten Schritt werden beide Tokens über die Weboberfläche von Jenkins als Geheimnisse abgespeichert. Der Token der von CaaS wird unter dem Namen `CAAS_ID` abgelegt, während der Token von Jenkins unter dem Namen `JENKINS_ID` abgelegt wird.

Im Anschluss wird in Jenkins eine neue Pipeline angelegt. Die Pipeline besteht aus zwei Stufen. Zuerst wird ein Experiment gestartet. Zusätzlich wird das Ergebnis des Experiments eingeholt. In der zweiten Stufe wird das eingeholte Ergebnis ausgegeben.

Für das Starten eines Experiments wird von der CaaS-Dokumentation bereits eine Beispielpipeline bereitgestellt. Das Beispiel verwendet jedoch globale Variablen für die Übertragung von Parametern und eignet sich dadurch nicht als Bibliothek für den Start einer beliebigen Anzahl von Experimenten. Dennoch kann die Logik für die Kommunikation mit der Schnittstelle von CaaS übernommen werden.

Es werden drei Klassen modelliert. Die erste Klasse `CaaSExperimentDefinition` stellt ein Experiment in einer entfernten CaaS-Instanz dar. Es kapselt somit die URL von CaaS, den Namen des Experiments, den Namespace in welchem das Experiment gestartet werden soll und die beiden Tokens.

Die zweite Klasse `CaaSExperimentResultFuture` stellt ein Experiment dar, welches gerade in Ausführung ist. Das Ergebnis ist somit noch nicht vorhanden und es muss auf ein Ergebnis gewartet werden.

Das aus der Ausführung entstehende Ergebnis wird über die Klasse `CaaSExperiment-Result` modelliert. Aufgrund der Auftrennung der Logik in die drei Klassen wird das beliebige Ausführen einer `CaaSExperimentDefinition` ermöglicht.

`CaaSExperimentDefinition` erhält die Methode `trigger`, welche das Experiment ausführt. Hierfür wird über die Shell und das Werkzeug `curl` eine POST-Anfrage an `CAAS_URL/requestHandler` gesendet, wobei im Körper der Anfrage die Aktion `helmRemoteInstall` angegeben wird, was das Ausführen eines Experiments verursacht. Die weiteren Experimentsdetails werden auch im Körper der Anfrage übertragen. Das Ergebnis ist eine Nachricht im JSON-Format. Das Ergebnis der Anfrage wird über die Methode `processTriggerResult` verarbeitet, worauf der Name des erstellten Jobs erhalten wird. Aus dem Namen des Jobs und der Experimentdefinition wird eines Instanz der Klasse `CaaSExperimentResultFuture` erstellt und zurückgegeben.

Die Klasse `CaaSExperimentResultFuture` enthält die Methode `awaitResult`, welche die weitere Ausführung der Pipeline blockiert und auf ein Ergebnis des Experiments wartet. Hierfür wird in einer Schleife die Methode `tryGetResult` aufgerufen. `tryGetResult` ruft über eine POST-Anfrage an `CAAS_URL/requestHandler` den Status des Experiments ab. Als Aktion wird im Körper der Anfrage `journalRemote-Provider` übergeben. Sobald ein Ergebnis vorhanden ist, wird das Ergebnis über die Methode `parseResult` verarbeitet.

Hierdurch entsteht zuletzt eine Instanz der Klasse `CaaSExperimentResult`, welche eine fast reine Datenklasse ist und alle Ergebnisse beinhaltet. Die Klasse enthält nur die Methode `print`, welche alle Ergebnisse in die Konsolenausgabe der Pipeline schreibt.

Sodass die einzelnen Methoden der drei Klassen in die Konsolenausgabe schreiben können, sowie über die Shell Befehle ausführen können, werden von der aktuellen Stufe der Pipeline bereitgestellte Methoden benötigt. Die bereitgestellten Methoden sind im Kontext der Klasse nicht verfügbar. Um dennoch wie gewohnt die Methoden aufrufen zu können, erben die drei Klassen von der erstellten Klasse `ScriptEnv`. Die Klasse `ScriptEnv` speichert den aktuellen Schritt der Pipeline und leitet alle Aufrufe an die bekannten Methoden an den aktuellen Schritt der Pipeline weiter. Eine gekapselte Methode ist in 4.4 zu sehen.

```
1 Script script
2 void println(String message){
3     this.script.echo(message)
4 }
```

Listing 4.4: Kapselung der `sh`-Methode

Die hier entworfene Pipeline kann in Jenkins in einem Intervall ausgeführt werden, sodass Experimente automatisch periodisch ausgeführt werden.

4.6 Experimentstatusbenachrichtigungen

Nachdem eine Simulation von Nutzerinteraktionen in den Experimentablauf integriert wurde, folgt im Anschluss die automatische *Mitarbeiterbenachrichtigung*, sodass der Experimentierer nach Ablauf des Experiments benachrichtigt wird und nicht auf die Terminierung des Experiments warten muss. Hierfür wird ein Slack-Bot erstellt, über welchen die Nachrichten automatisch versandt werden.

4.6.1 Einrichten eines Slack-Bots

Zuerst wird ein neuer Textkanal in Slack erstellt, in welchen die automatisierten Nachrichten gesendet werden. Der Kanal wird auf privat gestellt, sodass nur die notwendigen

Mitarbeiter benachrichtigt werden, die zuvor eingeladen wurden. Der Kanal wird in einem speziellen internen Slack-Arbeitsraum erstellt, dem Arbeitsraum `App Testing`. In diesem Arbeitsraum haben alle Nutzer die Berechtigung, einen Slack-Bot sowie Textkanäle zu erstellen. Dadurch eignet sich der Arbeitsraum zum Testen der Integration, da ohne weitere Wartezeit die Integration direkt umgesetzt werden kann.

Im Anschluss wird über Slack's Weboberfläche in dem Arbeitsraum der neue Bot erstellt. Zunächst wird der Bot in den privaten Textkanal eingeladen, sodass der Kanal für den Bot sichtbar ist. Im Anschluss werden dem Bot granulare Berechtigungen gegeben. Der Bot bekommt nur die Berechtigung, in Textkanäle Nachrichten zu senden.

4.6.2 Erweiterung des Python-Moduls

Nachdem der Bot eingerichtet wurde, muss der Bot über das Python-Modul zum Nachrichtenversenden gesteuert werden, sodass eine Nachricht zum Ende des Experiments generiert wird. Das Senden einer Statusnachricht beeinflusst das Zielobjekt des Experiments nicht. Es kann sich somit um keine Probe oder Aktion handeln. Es wird somit eine Steueraktion entwickelt. Um die davor verwendete Konvention einzuhalten, wird das Untermodul `controls` in dem Untermodul `common` erstellt. Für jede entwickelte Steueraktion wird in dem Untermodul eine Datei angelegt. Jede Datei enthält eine Menge aus Funktionen, die zu bestimmten Zeitpunkten des Experiments aufgerufen werden. In der dort erstellten Datei `slack_notification.py` wird die Funktion `after_experiment_control` implementiert. Die Funktion wird am Ende des Experiments einmalig aufgerufen. Sie hat Zugriff auf die Konfigurationswerte und Geheimnisse des Experiments.

Die Funktion liest zuerst die Konfiguration aus und erhält die Identifier (ID) des Nachrichtenkanals, sodass Nachrichten in den spezifizierten Kanal gesendet werden. Der zweite anpassbare Parameter, der ausgelesen wird, ist `slack_result_predicates`, welcher die Bedingung spezifiziert, wann eine Nachricht gesendet wird. Der Parameter ist eine Liste aus Experimentstatus. Wenn die Liste z. B. den Wert „`aborted`" enthält, dann wird eine Nachricht gesendet, wenn das Experiment abgebrochen wurde. Aus den Geheimnissen wird ein Token ausgelesen, welcher den Bot identifiziert und gleichzeitig es erlaubt Nachrichten über den Bot zu versenden.

Nachdem alle notwendigen Parameter verarbeitet wurden, wird aus dem Experiment eine Nachricht zusammengebaut. Slack erlaubt das Versenden angereicherter Nachrichten, die über einen einfachen Text hinausgehen und z. B. Knöpfe, Tabellen, Bilder, Anhänge, Abtrenner und weitere Komponenten enthalten. Hierfür wird über Hilfsmethoden ein Python-Objekt zusammengebaut, welches eine Nachricht repräsentiert. Das Objekt ist in 4.5 zu sehen.

```
1    blocks = [
2        header(f"{status_icon} {title}"),
3        divider(),
4        section([
5            field(f"*from*: {start}"),
6            field(f"*to*: {end}")
7        ]),
8        section([field(f"*status*: {status} {status_icon}")])
9    ]
```

Listing 4.5: Zusammenbau einer Slack-Benachrichtigung

Die Hilfsmethoden werden in die Bibliothek `slack.py` gruppiert, welche sich in dem Untermodul `utilities` befindet. Die Hilfsmethoden geben entsprechende Zuweisungstabellen zurück, die jeweils ein Objekt einer Slack-Nachricht beschreiben. Die Nachricht enthält im Kopf ein Icon, welches den Zustand darstellt, sowie den Namen des Experiments. Nach einem horizontalen Abtrenner wird die Startzeit, Endzeit, sowie der ausgeschriebene Zustand angezeigt. In Slack sieht die finale Nachricht bei Erfolg[1] wie in Abbildung 4.4 aus.

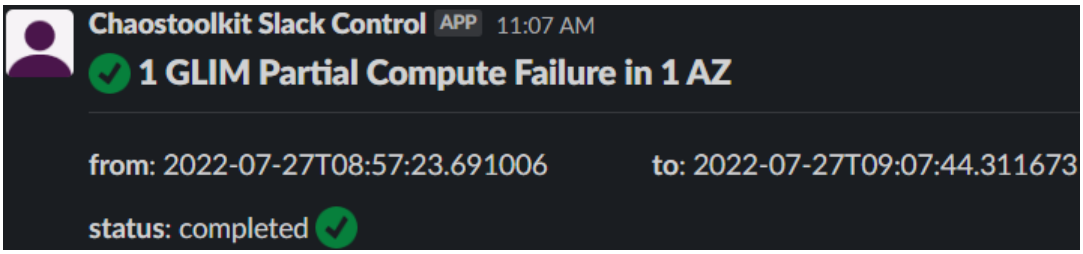

Abbildung 4.4: Slack-Nachricht eines erfolgreichen Experiments

Die zusammengebaute Nachricht wird über die Funktion `send_blocks` versendet, die sich auch in der entwickelten Bibliothek `slack.py` befindet. Die Funktion nimmt den

[1]Bei Scheitern sieht die Nachricht wie in Abbildung C.1 aus

Token des Bots, die ID des Textkanals und die gebaute Nachricht. Die Funktion versendet eine POST-Anfrage an `https://slack.com/api/chat.postMessage`, gibt dabei den Token in dem Header `Authorization` an und überträgt die gebaute Nachricht im JSON-Format. Dabei wird ein Objekt übertragen, welches die Attribute `channel` und `blocks` enthält. Der `channel` wird auf die Kanal-ID gesetzt und der Inhalt von `blocks` auf die übergebene zusammengebaute Nachricht.

4.6.3 Modifikation eines Experiments

Sodass die Funktionalität in den Ablaufprozess eines Experiments integriert wird, muss zuletzt eine Experimentdefinition angepasst werden. Zuerst wird die Steueraktion zu der Liste der definierten Steueraktionen hinzugefügt. Die Steueraktion wird wie in 4.6 definiert.

```
1 "name": "Report experiment result in Slack",
2 "provider": {
3     "type": "python",
4     "module": ↙
          ↪ "background_test_extensions.common.controls.slack_notification"
5 }
```

Listing 4.6: Definition einer Slack-Benachrichtigung in einem Experiment

Damit die Steueraktion die Konfiguration und Geheimnisse ausliest, müssen die entsprechenden Konfigurationswerte und Geheimnisse definiert werden. In der Konfiguration muss jeweils ein Attribut `slack_result_channel_id` und `slack_result_predicates` definiert werden. In der Sektion `secretsSpec` muss ein Secret `slack_result_bot_bearer` definiert werden, welches das Attribut `token` enthält.

4.7 Screenshots

Da das Hinzufügen einer Screenshot-Funktionalität weniger komplex als das Hinzufügen von Lasttests ist, wird zuerst die Screenshot-Funktionalität hinzugefügt. Es sollen Bilder von beliebigen Seiten automatisch genommen werden können und im Anschluss dem Experiment angehängt werden.

4.7.1 Ansatz

Um Bilder von Webseiten nehmen zu können, muss die Webseite zuerst angezeigt werden. Hierfür wird ein Browser benötigt. Der Browser zeigt die Webseite an und über den Browser wird ein Screenshot genommen. Da die Screenshots automatisch genommen werden sollen, muss die User Interface (UI) des Browsers deaktiviert werden, sodass alle Aktionen im Hintergrund geschehen. Um eine hohe Webseitenkompatibilität zu erzielen, wird ein Chromium basierter Browser verwendet. Zur Steuerung des Browsers wird eine Python-Bibliothek verwendet, da das restliche Projekt bereits in Python entwickelt wurde. Da Selenium eine weit verbreitete Bibliothek ist, wird Selenium verwendet. Selenium erlaubt jedoch nicht das Abfangen und Setzen der HTTP-Anfrage-Header. Da die Funktionalität benötigt wird, wird die Selenium-Erweiterung Selenium-Wire verwendet.

4.7.2 Erweiterung des Python-Moduls

Da die entwickelte Funktionalität als Bibliothek von anderen Code-Schnippseln verwendet wird, ist der Code in der neuen Datei `screenshot.py` in dem Untermodul `utilities` anzufinden.

Die neue Datei stellt die Funktion `make_screenshot` bereit, welche einen Screenshot einer Webseite nimmt. Als Parameter wird die URL der Webseite und der Dateiname übergeben, in welcher der Screenshot gespeichert wird. Um eine Authentifizierung zu der angegebenen Webseite zu ermöglichen, kann ein Benutzername sowie Passwort übergeben werden. Um eine Authentifizierung über z. B. einen Token zu ermöglichen, können HTTP-Header übergeben werden, die bei jeder Anfrage gesetzt werden.

Bei dem Erstellen von Screenshots sind mehrere Punkte zu beachten. Die Größe des Screenshots hängt von der Größe des Browser-Fensters ab. Wenn die anzuzeigende Webseite größer als der Browser ist, dann enthält der Screenshot nur einen Ausschnitt der Webseite. Alternativ könnte die Größe der Webseite nach dem Laden bestimmt werden und die Größe des Browsers auf die Größe der Webseite gesetzt werden. Jedoch Laden manche Webseiten ihren Inhalt dynamisch nach. Andere Webseiten sind kleiner oder gleich groß, wie der Browser, jedoch ist der verschachtelte Inhalt der Webseite größer als die Webseite selbst, wieso auf der Webseite dennoch gescrollt werden müsste. Letztendlich ist eine einfache Lösung, die Größe des Browsers zu parametrisieren und standardmäßig

auf einen großen Wert zu setzen. Standardmäßig wird die Breite auf 1920 Pixel und die Höhe auf 3840 Pixel gesetzt. Kaum Webseiten verwenden einen horizontalen Scroll, weshalb die Breite auf die normale Auflösung eines FullHD-Monitors gesetzt wurde. Sodass die komplette Webseiten-Höhe erfasst wird, wird die Höhe auf die doppelte Breite gesetzt.

Ein weiterer Punkt, der zu beachten ist, ist das Seitenladen. Nur weil eine Seite geladen ist, ist ihr Inhalt nicht unbedingt geladen, da der Inhalt dynamisch nachgeladen werden kann. Ein Beispiel sind die Diagramme, die innerhalb eines Grafana-Dashboards nachgeladen werden. Die erstellte Bibliothek kann nicht wissen, wann der Inhalt der Seite geladen ist. Deshalb wird hierfür dem Anwender die Möglichkeit gelassen, den Zeitpunkt selbst zu bestimmen, beziehungsweise auf das Ende des Ladens des Inhalts der Seite zu warten. Es wird als ein weiterer Parameter eine Funktion entgegengenommen. Die Funktion wird nach dem Laden der Webseite aufgerufen. Der Nutzer implementiert die Funktion und spezifiziert seine Wartelogik, welche auf das Laden des Inhalts wartet. Im einfachsten Fall wäre die Wartelogik die Funktion `sleep`, die n Sekunden wartet.

Die Funktion `make_screenshot` erstellt zuerst über die Funktion `create_driver` eine Instanz des Browsers. Der Funktion `create_driver` muss somit die Größe des Browsers übergeben werden. Sodass in der Funktion der Browser gestartet werden kann, muss zuerst die ausführbare Datei des Browsers gefunden werden. Es wird nicht nur der Browser benötigt, sondern auch der ausführbare Treiber, über welchen der Browser gesteuert wird. Nach beiden Dateien wird in den Ordnern der Umgebungsvariable `PATH` gesucht. Für den `chromium-browser` wird der Treiber `chromedriver` benötigt. Beide müssen im System installiert sein. Dann wird anhand des Treibers und des Browsers eine steuerbare Browser-Instanz gestartet. Der Instanz werden über den Treiber Startoptionen übergeben. Über die Startoption `-headless` wird die UI deaktiviert. Die Größe des Browsers wird über `window-size=BREITE,HÖHE` gesetzt. Da Chromium innerhalb eines Containers gestartet wird und dem Browser sehr wenig geteilter Arbeitsspeicher zugewiesen wird, kann dieser bei Laden einer Webseite abstürzen. Um das zu verhindern wird der geteilte Arbeitsspeicher über die Startoption `-disable-dev-shm-usage` deaktiviert.

Die Funktion `create_driver` erhält die neue Browser-Instanz. Sollten Authentifizierungsdetails angegeben worden sein, dann wird die URL der Webseite abgeändert, sodass die URL das Format `https://NUTZER:PASSWORD@host.domain/path` hat. Für

den Treiber wird zudem eine Callback-Funktion spezifiziert, die Anfragen des Browsers abfängt. In der Funktion werden die spezifizierten Anfrage-Header gesetzt.

Die Webseite wird über `driver.get(URL)` geladen. Im Anschluss wird der Parameter `page_load_cb` aufgerufen, um auf das Laden des Seiteninhalts zu warten, was standardmäßig eine Nulloperation ist. Der Screenshot wird nach Laden über `driver.save_screenshot(DATEINAME)` gespeichert. Zuletzt wird die Browser-Instanz beendet.

4.7.3 Einbindung in ChaosToolkit

Ziel der Bibliothek ist das automatisierte Nehmen von Screenshots beliebiger Webseiten. Um die Bibliothek in ChaosToolkit einzubinden, wird eine einfache Funktion in `actions.py` erstellt. Die dort erstellte Funktion `screenshot` nimmt alle für einen Screenshot benötigten Parameter entgegen und gibt die Parameter an die Screenshot-Bibliothek weiter. Über die einfache Aktion wird die Screenshot-Funktionalität in ein Experiment integriert und kann getestet werden.

Über die einfache Aktion können Screenshots von im Experiment definierten Screenshots genommen werden. Ein weiteres Ziel ist die Integration in weitere Python-Module. Sobald die in den anderen Python-Modulen definierten Aktionen Chaos im System verrichtet haben, sollen automatisiert Screenshots von Statusseiten genommen werden. Die Statusseiten werden von den anderen Python-Modulen dynamisch für den Durchlauf der Aktion generiert.

Um die Screenshot-Bibliothek in weitere Python-Module zu integrieren würden folgende weiteren Schritte benötigt werden.

Über die Bibliothek können Screenshots von beliebigen Webseiten genommen werden und in einer angegebenen Datei gespeichert werden. Um die Funktionalität in ChaosToolkit einzubinden, müssten vorhandene Aktionen und Proben aus bestehenden Python-Modulen angepasst werden, sodass die Module die neue Funktion aufrufen, um Screenshots von relevanten Webseiten zu nehmen, die temporär Ergebnisse einzelner Störungen anzeigen.

Ein zweiter Schritt ist das automatische Anhängen der Screenshots an die Experimentergebnisse. Die Bilder können den Experimentergebnissen nicht angehängt werden, da

eine solche Funktionalität derzeit nicht implementiert ist. Der Code, der sich mit den Experimentergebnissen beschäftigt und bereits die gesammelten Logs hochlädt, müsste verändert werden. Anstelle dass nur die Logs hochgeladen werden, müssten zusätzlich alle weiteren Anhänge hochgeladen werden.

Sobald die beiden Schritte von den jeweiligen Abteilungen vorgenommen wurden, muss sichergestellt werden, dass Chromium und die Treiber in dem Container des Experiments vorhanden sind. Hierfür können in der Definition des Experiments in der Konfiguration eine Liste aus Paketnamen angegeben werden, die im Container des Experiments installiert werden. Die Funktionalität wird bereits durch ein weiteres Python-Modul bereitgestellt, welche das Verhalten über eine Steueraktion implementiert hat, die vor dem Experiment ausgeführt wird.

4.8 Lasttests

Zuletzt werden die Lasttests implementiert. Die Implementierung der Lasttests ist ähnlich zu der Implementierung der funktionalen Tests, jedoch komplexer.

4.8.1 Ansatz

Die Lasttests sollen die funktionalen Tests zur Simulierung von Nutzerinteraktionen ersetzen. Die funktionalen Tests waren ein erster einfacher Schritt, um Interaktionen zu simulieren, jedoch erlauben die Lasttests eine feinere Steuerung der Interaktionen, aber sind dafür auch komplexer umzusetzen. Die Lasttests folgen einem ähnlichen Prozess wie die funktionalen Tests. Sie sollen auch während des Experiments im Hintergrund ausgeführt werden. Im Gegensatz zu den funktionalen Tests werden die Lasttests nicht kontinuierlich ausgeführt, sondern nur einmalig, da die Dauer eines Lasttests über einen Parameter auf die Länge des Experiments festgelegt wird.

Um Lasttests automatisiert zu starten, soll die davor implementierte Klasse `Background-Test` verwendet werden. Es muss dadurch nur eine Funktion implementiert werden, die einen Lasttest automatisch startet und zuletzt die Ergebnisse des Lasttests sammelt. Die Lasttests werden von den Teams bereitgestellt, die das Zielobjekt des Experiments verwalten.

Für das Ausführen eines Lasttests muss das Docker-Image des Lasttests in einem System mit einer Containerd-Laufzeit installiert werden. Da das K8s-System `StuffSS` Containerd unterstützt, wird der Lasttest in demselben System installiert. Das Docker-Image wird als K8s-Pod installiert. Sobald der Lasttests beendet ist, entsteht dadurch das Problem, dass K8s den Container in dem Pod automatisch entfernt und dadurch nicht mehr auf die Logs und Ergebnisse zugegriffen werden kann. Bevor der Lasttest terminiert, muss somit das Ergebnis an das Experiment übermittelt werden. Der Informationsfluss der Experimentergebnisse ist in Abbildung 4.5 zu sehen.

Abbildung 4.5: Informationsfluss der Experimentergebnisse

Der Informationsfluss kann auf zwei Arten umgesetzt werden. Entweder liest das Experiment kontinuierlich die letzten Ergebnisse des Lasttests ein, oder der Lasttests überträgt die Ergebnisse zuletzt einmalig. Wenn das Experiment die Ergebnisse kontinuierlich einliest, bis der Lasttest beendet, dann entsteht dadurch eine Ungenauigkeit, da das Experiment die Ergebnisse nicht genau zum Ende des Tests einliest, sondern abhängig vom Intervall eine kurze Zeit davor. Vorteil dieser Lösung ist, dass der Lasttest nicht verändert werden muss.

Wenn der Lasttest die Ergebnisse selbstständig überträgt, dann muss der Lasttest verändert werden. Vorteil ist jedoch, dass die genauen Ergebnisse übertragen werden. Da die genauen Ergebnisse benötigt werden, wird die zweite Methodik verwendet.

Bei den Ergebnissen der Lasttests, sowie den Lasttests ist zu beachten, dass unterschiedliche Teams unterschiedliche Bibliotheken zur Umsetzung der Tests verwenden. Die Ergebnisse sind somit in unterschiedlichen Formaten. Der zu implementierende Code muss somit die unterschiedlichen Formate einheitlich verarbeiten können.

4.8.2 Erweiterung des Python-Moduls

Um die neue einmalig auszuführende Aktion umzusetzen, wird analog zu dem Untermodul `jenkins` das Untermodul `load_test` angelegt. In dem Untermodul wird die gleichnamige Funktion `load_test` bereitgestellt, die das oben beschriebene Verhalten

umsetzt. Der Prozess ist analog zu dem Prozess des funktionalen Tests, weshalb der Prozess Abbildung 4.3 entnommen werden kann.

Parameterverarbeitung Zuerst verarbeitet die Funktion die übergebenen Parameter. Es liest die benötigten Attribute aus den Parametern `configuration` und `secrets` aus. Sodass der Lasttest in dem `StuffSS`-System gestartet werden kann, wird der Name des Docker-Images benötigt, sowie der K8s-Namespace, in welchem ein Pod mit dem Container gestartet werden soll. Sodass dem Test Parameter übergeben werden können, werden die Werte `duration`, `users`, `intensity` und `arguments` eingelesen. Die Dauer des Tests wird über `duration` festgelegt. Die nächsten beiden Parameter legen fest, wie viele parallele Nutzer simuliert werden sollen und wie viele Anfragen die Nutzer jeweils versenden. Der letzte Wert erlaubt es, beliebige weitere benötigte Werte an den Lasttest zu geben, sodass der Lasttest z. B. eine Authentifizierung vornehmen kann.

HTTP-Server Nachdem die Parameter verarbeitet wurden, wird ein einfacher HTTP-Server auf einem beliebigen offenen Port geöffnet. Der HTTP-Server nimmt die gesendeten Logs und Ergebnisse des Lasttests entgegen, kurz bevor der Lasttest terminiert. Der Server wird über die Methode `create_server` erstellt. Da der Server nach Erhalt der Logs und Ergebnisse nicht mehr benötigt wird, kann ein sehr einfach aufgebauter Server gewählt werden. Es wird der von Python bereitgestellte Server aus dem Modul `http.server` gewählt. Sodass ein zufälliger offener Port gewählt wird, wird der Port null genommen. Das Betriebssystem übernimmt dann die Aufgabe, einen zufälligen offenen Port zu wählen. Die Funktion `create_server` nimmt eine Funktion entgegen, die aufgerufen werden soll, sobald die Logs und Ergebnisse vorliegen, sodass beide weiterverarbeitet werden können.

Für den Server wird die neue dynamische Klasse `LoadTestResultHandler` in der Funktion `create_server` erstellt, die von `BaseHTTPRequestHandler` erbt. Sobald der Server Anfragen entgegennimmt, werden Methoden der neuen Klasse aufgerufen. Hierfür wird die Klasse der Instanz des Servers übergeben. Die Logs und Ergebnisse des Lasttests sind jeweils Textdateien. Beide sollen zusammengefügt in einer POST-Anfrage an den Server unter dem Pfad `/request` übergeben werden. In der neuen dynamischen Klasse `LoadTestResultHandler` wird die Methode `do_POST` definiert, welche bei einer POST-Anfrage aufgerufen wird.

In der `do_POST`-Methode wird zunächst überprüft, ob es sich um eine Übertragung der Logs und Ergebnisse handelt. Es wird der Pfad der Anfrage mit `/request` verglichen, sowie überprüft, ob eine Log und Ergebnisdatei vorhanden sind. Beide Dateien sollen im Körper der POST-Anfrage übertragen werden. Die Anfrage entspricht dem Format `multipart/form-data`. Die Log ist unter dem Schlüssel `log` zu finden und die Ergebnisse sind unter dem Schlüssel `result` zu finden. Inhalt ist jeweils eine Liste aus dem Namen der Datei und dem Inhalt der Datei. Die Daten liegen jeweils als rohe Bytes vor und müssen als `UTF-8`-Text dekodiert werden. Dafür wird die an die Funktion `create_server` übergebene Funktion aufgerufen, welche die Daten weiter verarbeitet. Nachdem der Server erstellt wurde, wird der Server von der Funktion `create_server` zurückgegeben.

Da das Betriebssystem dem Server einen zufälligen offenen Port zugewiesen hat, kann der Port in der Funktion `load_test` ausgelesen werden. Mit dem Port wird die aktuelle externe Adresse des eigenen Pods erlangt. K8s internes Domain Name Server (DNS) löst alle Anfragen an die Adresse auf, sodass die Anfragen an den ChaosToolkit-Pod geleitet werden. Hierfür wird aus dem Pod heraus mit der K8s-Schnittstelle kommuniziert, um die aktuelle IP des Containers herauszufinden. Die Adresse des ChaosToolkit-Pods hat folgenden Aufbau: `http://IP.NAMESPACE.pod.cluster.local:PORT`. Die IP-Segmente müssen dabei anstelle eines „.“ mit einem „-“ abgetrennt sein, um URL-konform zu sein.

Starten des Lasttests Sobald die Adresse des aktuellen Containers, sowie die weiteren Parameter vorhanden sind, wird ein K8s-Deployment erstellt, welches den Lasttest startet. Das Deployment ist eine Schablone für mehrere Pods, die aus der Schablone heraus erstellt werden und unter dem Deployment gruppiert werden. Das Deployment wird über die Funktion `generate_deployment` generiert, welches eine textuelle Definition generiert.

Das generierte Deployment wird im Anschluss direkt über die Funktion `create_deployment` erstellt. Für die Funktion `create_deployment` wird die K8s-Schnittstelle verwendet.

Zuletzt wird der erstellte Server gestartet, sodass auf die eintreffende Log und das Ergebnis gewartet wird. Sobald beide eintreffen, terminiert der Server und die nächste

Funktion wird aufgerufen, welche das erstellte Deployment wieder löscht, sodass das System aufgeräumt wird.

Verarbeitung der Daten Sobald über den Server das Ergebnis und die Log erhalten wurde, werden beide in der Funktion `load_test_result_callback` verarbeitet. Zuerst werden beide Dateien in dem Ordner `/workdir/` als Log-Datei abgelegt, sodass beide den Experimentergebnissen angehängt werden. Zur Generierung eines eindeutigen Namens wird dabei der übergebene Name der Datei verwendet.

Zuletzt muss das Ergebnis verarbeitet werden. Zuerst wird der Parameter `type` ausgelesen, welcher angibt, ob es sich um Ergebnisse eines Lasttests des Frameworks JUnit oder Locust handelt. Je nach Framework werden die Methoden `parse_junit` oder `parse_locust` aufgerufen, die die Ergebnisdatei übergeben bekommen.

Beide Methoden sind bisher leer, da für eine Implementierung realistische Daten zum Testen fehlen. Zuerst muss ein vorhandener Lasttest von dem verantwortlichen Team angepasst werden, sodass der angepasste Lasttest zum Testen der Integration verwendet werden kann. Für eine Anpassung werden folgende Schritte benötigt:

Die Lasttests werden als Docker-Container angegeben. Jeder Docker-Container hat eine `Dockerfile`, in welcher ein Startbefehl definiert wird. Der Startbefehl wird zuerst abgeändert, sodass anstelle des ursprünglichen Befehls ein neues Skript ausgeführt wird.

Im zweiten Schritt wird das Skript erstellt. Das Skript ruft zuerst den Lasttest auf, danach wird die POST-Anfrage an den temporären HTTP-Server des Experiments gesendet.

In den Umgebungsvariablen des Skripts liegen die dem Lasttest übergebenen Parameter. Um die Dauer des Tests auszulesen, wird die Umgebungsvariable `CHAOS_LOAD_TEST_DURATION` ausgelesen. Für die Intensität und Anzahl der Nutzer können die Variablen `CHAOS_LOAD_TEST_INTENSITY` und `CHAOS_LOAD_TEST_USERS` ausgelesen werden.

Die verarbeiteten Parameter werden dem Lasttest über den ursprünglich angegebenen Befehl übergeben.

Im letzten Schritt wird der Lasttest selbst angepasst. Die nun an den Lasttest übergebenen Parameter werden verwendet, um über das verwendet Framework die Dauer, Intensität und Anzahl der Nutzer festzulegen. Zudem muss eine Loggingfunktionalität, sowie die Ergebnisausgabe aktiviert werden.

4.8.3 Einbindung in ChaosToolkit

Wie bei den funktionalen Tests muss jeweils eine Probe und eine Aktion zu der Experimentdefinition hinzugefügt werden. Jedoch muss bei der auszuführenden Aktion nun anstelle des funktionalen Tests die Funktion des Lasttests angegeben werden. Zudem müssen die Dauer, Intensität und Anzahl der Nutzer angegeben werden.

5 Ergebnisse

Nachdem das entwickelte Konzept in der Praxis umgesetzt wurde, wird in diesem Kapitel auf die erzielten Ergebnisse eingegangen. Zuerst werden die Ergebnisse genannt. Zuletzt werden die Ergebnisse anhand der Anforderungen bewertet.

Ergebnis der Arbeit ist ein Konzept zur Umsetzung von effizientem Chaos-Engineering auf einem nicht-Produktivsystem sowie die Implementierung des Konzeptes. Das Konzept und die Umsetzung des Konzeptes sollen anhand der definierten Bewertungskriterien in dieser Sektion diskutiert und bewertet werden. Die Bewertungskriterien sind eine Zeiteffizienz, Kosteneffizienz und Vergleichbarkeit. Zuerst wird das Konzept diskutiert und bewertet.

Das Konzept legt neben einer Vergleichbarkeit einen Schwerpunkt auf die Automatisierung des kompletten Chaos-Experimentablaufs. Indem alle Schritte eines zuvor manuellen Chaos-Engineerings automatisiert werden, wird Arbeitszeit eingespart die währenddessen in andere Projekte investiert werden kann. Das Konzept automatisiert zusätzlich zu dem Experiment benötigte Vor- und Nacharbeit. Zu der Nacharbeit zählt beispielsweise das Nehmen von Screenshots zur Erfüllung von Vorgaben.

Die Dauer des manuellen Chaos-Engineerings wurde in der Praxis anhand durchgeführter Experimente abgeschätzt. Die Ausführung eines Experiments beträgt circa 10 Minuten. Hinzu kommen etwa fünf weitere Minuten für Nacharbeit. Für ein manuell ausgeführtes Experiment werden somit circa 15 Minuten benötigt. Bei automatischen Experimenten werden immer noch 10 Minuten zur Ausführung des Experiments benötigt, in diesen 10 Minuten muss der Experimentierer jedoch nicht anwesend sein, da nach Ablauf des Experiments eine automatisierte Benachrichtigung an ihn versandt wird. Die Nacharbeitung wird auch über Automatisierung übernommen. Es werden lediglich zwei Minuten zur Überprüfung der Ergebnisse benötigt.

Formel	$t_{manuell}$	$t_{automatisch}$	$\frac{t_{manuell}}{t_{automatisch}}$	$t_{manuell} - t_{automatisch}$
Wert	15min	2min	7.5	13min

Tabelle 5.1: Ergebnis des Bewertungskriteriums Zeiteffizienz

Die daraus resultierenden Werte sind 5.1 zu entnehmen. Wie der Tabelle zu entnehmen ist, wird ein Zeitersparnisfaktor von 7.5 erzielt. Absolut werden je Experiment circa 13 Minuten eingespart. Es wird somit eine sehr hohe Zeiteffizienz erzielt.

Neben der Automatisierung des Chaos-Engineerings setzt das Konzept auf eine hohe Wiederverwendung vorhandener Strukturen und Dienste. Durch die hohe Wiederverwendung soll eine Kosteneffizienz erzielt werden. Die Kosteneffizienz ist ein weiches Bewertungskriterium und schwer zu messen. Für die Bewertung der Kosteneffizienz werden somit die Bestandteile betrachtet, aus der sich die Kosteneffizienz zusammensetzt. Zuallererst ist die Zeiteffizienz ein großer Faktor. Durch die Automatisierung kann ein Mitarbeiter Zeit in ein anderes Projekt investieren, wodurch letztendlich Geld gespart wird.

Eine Folge der hohen Wiederverwendung vorhandener Strukturen und Dienste ist die Wiederverwendung von vorhandenen Systemen. Jedes laufende System hat enorme Grundkosten. Indem für das Konzept vorhandene Systeme wiederverwendet werden, werden weitere anfallende Infrastrukturkosten gesenkt. Aufgrund der hohen Zeiteffizienz und der niedrigen weiteren Infrastrukturkosten wird die Kosteneffizienz als hoch angesehen.

Das letzte Bewertungskriterium ist die Vergleichbarkeit. Die Vergleichbarkeit ist ein fundamentaler Aspekt der Arbeit aber dennoch schwer zu bewerten. Als Kernaspekt der fehlenden Vergleichbarkeit identifiziert das Konzept die niedrigen oder fehlenden Nutzerinteraktionen auf dem nicht-Produktivsystem. Um die Vergleichbarkeit herzustellen, wird auf vorhandene funktionale Tests sowie Lasttests gesetzt. Die Tests simulieren Nutzerverhalten und erzeugen dadurch Nutzerinteraktionen.

Das Konzept löst nur den Kernaspekt der fehlenden Vergleichbarkeit. Weitere kleinere Aspekte werden nicht berücksichtigt, beziehungsweise vernachlässigt. Da Nutzerinteraktionen nur simuliert werden, werden keine perfekt vergleichbaren Nutzerinteraktionen erzielt. Da die Tests Nutzerverhalten simulieren und die fehlenden Nutzerinteraktionen der Kernaspekt der Vergleichbarkeit ist, wird dennoch eine sehr gute Vergleichbarkeit geschaffen.

Letztendlich setzt das Konzept die Anforderungen der Effizienz und die implizierte Anforderung der Vergleichbarkeit gut um.

6 Schluss

Zuletzt wird ein Rückblick auf die gesamte Arbeit geworfen. Im Anschluss folgt ein Ausblick, in welchem zukünftige Schritte genannt werden.

6.1 Zusammenfassung

Zusammenfassend lässt sich bemerken, dass die Arbeit erfolgreich war. Als Basis für die Arbeit dient die Problematik, Chaos-Engineering effizient auf einem nicht-Produktivsystem umzusetzen. Aus der Problematik wurden die drei Ziele des Experimentierens auf einem nicht-Produktivsystem, eine Kostenminimierung und den Aufbau des Vertrauens in das Produktivsystem festgelegt. Zudem wurde durch Analyse der Ziele festgestellt, dass eine Vergleichbarkeit des nicht-Produktivsystems zu dem Produktivsystem fehlt und für ein Chaos-Engineering zuerst eine Vergleichbarkeit geschaffen werden muss. Dabei wurde festgestellt, dass der Grad der Vergleichbarkeit schwer zu bestimmen ist.

Aus den Zielen wurden konkrete Anforderungen abgeleitet. Besonders aus der Kostenminimierung ist die Kosteneffizienz und Zeiteffizienz entstanden.

Aus den granularen Anforderungen wurde ein allgemeines Konzept aufgestellt. Das Konzept setzt auf die Wiederverwendung von funktionalen Tests, sowie von Lasttests. Durch beide Tests sollte die Vergleichbarkeit geschaffen werden. Durch die Wiederverwendung von vorhandenen Komponenten sollte eine Zeiteffizienz und dadurch auch Kosteneffizienz erzielt werden. Die Zeiteffizienz sollte erhöht werden, indem alle Schritte des Chaos-Engineerings automatisiert werden.

Das Konzept wurde im Anschluss anhand der SAP AI-Dienste umgesetzt. Dabei wurde die Umsetzung des Projekts um einen weiteren in der Praxis notwendigen Schritt erweitert. Es wurde zusätzlich das automatisierte Nehmen von Screenshots von Webseiten implementiert.

Als Teil des umgesetzten Konzepts wurde das automatisierte Senden von Statusbenachrichtigungen über Slack ermöglicht. Zudem wurde eine Jenkins-Integration für CaaS

geschaffen, sowie eine CaaS-Integration für Jenkins. Sodass Lasttests für das Schaffen einer Vergleichbarkeit verwendet werden können, wurde eine Erweiterung für ChaosToolkit implementiert, die das automatische Starten und Verwalten containerisierter Lasttests ermöglicht.

Das in der Praxis umgesetzte Konzept konnte trotz fehlender Kleinigkeiten erfolgreich umgesetzt werden. Die Anforderungen konnten dabei eingehalten werden. Es wurde eine hohe Zeiteffizienz und dadurch eine Kosteneffizienz erzielt. Der Wirkungsgrad konnte durch die Testintegrationen gegeben werden. Indem alle Anforderungen erfüllt wurden, wurden alle Ziele erreicht.

6.2 Ausblick

Obwohl das Projekt erfolgreich abgeschlossen wurde, konnten einige Kleinigkeiten nicht vollkommen umgesetzt werden. Einige der für eine Umsetzung notwendigen Schritte sind von weiteren Teams abhängig. Dadurch steigt die Entwicklungszeit, da Aufwand für eine Kommunikation zwischen den Teams betrieben werden muss. Die in der Umsetzung fehlenden Schritte werden deshalb nach weiterer Kommunikation mit den zuständigen Teams zeitnah umgesetzt werden.

Nachdem die fehlenden Schritte umgesetzt wurden, ist die komplette Implementierung des Projekts vollständig. Dennoch ist das Ende der Implementierung erst der Anfang für eine ganze Reihe an automatisierten Experimenten auf einem nicht-Produktivsystem. Die bisherige Umsetzung des Konzepts stellt alle für das Konzept benötigten Bausteine bereit. Nach Ende des Projekts gilt es die bereitgestellten Bausteine in weitere vorhandene Experimente zu integrieren. Durch Integration in weitere Experimente soll die Stabilität der Implementation getestet werden.

Nachdem das Konzept und die Umsetzung des Konzepts weiter erprobt wurde, muss das Konzept innerhalb von SAP verbreitet werden, sodass weitere Teams das Konzept für ihre eigenen Produkte umsetzen können. Dadurch soll auch in weiteren Teams Vertrauen in die eigenen Produkte über automatisierte Chaos-Experimente aufgebaut werden.

Literaturverzeichnis

[1] Balalaie, A./ Heydarnoori, A./ Jamshidi, P. „Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture“. In: *IEEE Software* 33.3 (2016), S. 42–52.

[2] Martin Fowler, J. L. *Microservices*. Hrsg. von. https://martinfowler.com/articles/microservices.html. o.O., 2014. (Einsichtnahme: 15. 06. 2022).

[3] Ladyman, J./ Wiesner, K. *What Is a Complex System?* Yale University Press, 2020. URL: https://books.google.de/books?id=1Fz0DwAAQBAJ.

[4] Rosenthal, J. *Chaos-Engineering: System Resiliency in Practice*. 1st Edition. O'Reilly Media, 2020.

[5] community, C.-E. *Principles ofChaos-Engineering*. Hrsg. von Mathias Lafeldt, Gu Yu. https://principlesofchaos.org/. o.O., 2019. (Einsichtnahme: 14. 06. 2022).

[6] Miles, R. *Learning Chaos-Engineering: Discovering and Overcoming System Weaknesses Through Experimentation*. O'Reilly Media, 2019.

[7] Yang, S. „Move into the Cloud, shall we?“ In: 29 (2012).

[8] Matilla, T. „Comparing preconditions for cloud and on-premises development“. In: *Cloud-Based Software Engineering*. Department of Computer Science, University of Helsinki, 2013, S. 1–7.

[9] Linthicum, D. S. „Cloud-Native Applications and Cloud Migration: The Good, the Bad, and the Points Between“. In: *IEEE Cloud Computing* 4.5 (2017), S. 12–14.

[10] Kashyap, S./ Kaushik, N./ Chahal, D. „An Overview To Cloud Computing“. In: ().

[11] Patrick Cunningham CRM, F. „IT's Responsibility for Security, Compliance in the Cloud“. In: *Information Management* 44.5 (2010), HT6.

[12] Ananich, A. *What is IaaS?* Hrsg. von. https://web.archive.org/web/20160302153830/http://ananich.pro/2016/02/what-is-iaas/. o.O., 2016. (Einsichtnahme: 21. 06. 2022).

[13] Pahl, C. „Containerization and the PaaS Cloud“. In: *IEEE Cloud Computing* 2.3 (2015), S. 24–31.

[14] , Hrsg. *Total Containerd posts*. https://data.stackexchange.com/stackoverflow/revision/1618185/1973776/total-containerd-posts. o.O., 2022. (Einsichtnahme: 21. 06. 2022).

[15] , Hrsg. *Total Kubernetes Posts*. https://data.stackexchange.com/stackoverflow/revision/1608902/1963329/total-kubernetes-posts. o.O., 2022. (Einsichtnahme: 21. 06. 2022).

[16] , Hrsg. *What is Kubernetes?* https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. o.O., 2022. (Einsichtnahme: 22. 06. 2022).

[17] , Hrsg. *Nodes*. https://kubernetes.io/docs/concepts/architecture/nodes/. o.O., 2022. (Einsichtnahme: 22. 06. 2022).

[18] , Hrsg. *Pods*. https://kubernetes.io/docs/concepts/workloads/pods/. o.O., 2022. (Einsichtnahme: 22. 06. 2022).

[19] , Hrsg. *Namespaces*. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. o.O., 2022. (Einsichtnahme: 23. 06. 2022).

[20] Ananich, A. *What is PaaS?* Hrsg. von. https://web.archive.org/web/20161022140953/http://ananich.pro/2016/02/what-is-paas/. o.O., 2016. (Einsichtnahme: 21. 06. 2022).

[21] , Hrsg. *Total Cloud Foundry posts*. https://data.stackexchange.com/stackoverflow/revision/1608904/1963331/total-cloud-foundry-posts. o.O., 2022. (Einsichtnahme: 21. 06. 2022).

[22] , Hrsg. *Cloud Foundry Overview*. https://docs.cloudfoundry.org/concepts/overview.html. o.O., 2021. (Einsichtnahme: 22. 06. 2022).

[23] , Hrsg. *Cloud Foundry*. https://sapedia.one.int.sap/wiki/Cloud_Foundry. o.O., 2021. (Einsichtnahme: 23. 06. 2022).

[24] , Hrsg. *Cloud Foundry on Kubernetes*. https://github.com/cloudfoundry/korifi. o.O., 2022. (Einsichtnahme: 23. 06. 2022).

[25] Newman, S. *Building Microservices, 2nd Edition*. O'Reilly Media, 2021.

[26] Tyszberowicz, S. u. a. „Identifying microservices using functional decomposition“. In: *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer. 2018, S. 50–65.

[27] Dragoni, N. u. a. „Microservices: Yesterday, Today, and Tomorrow“. In: *Present and Ulterior Software Engineering*. Hrsg. von Mazzara, M./ Meyer, B. Cham: Springer International Publishing, 2017, S. 195–216. URL: https://doi.org/10.1007/978-3-319-67425-4_12.

[28] Dragoni, N. u. a. „Microservices: How to make your application scale“. In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer. 2017, S. 95–104.

[29] Fowler, M. *Microservice Trade-Offs*. Hrsg. von. https://martinfowler.com/articles/microservice-trade-offs.html. o.O., 2015. (Einsichtnahme: 24. 08. 2022).

[30] Andrew S. Tanenbaum, M. v. S. *Distributed Systems: Principles and Paradigms*. 2nd Edition. CreateSpace Independent Publishing Platform, 2016.

[31] Birolini, A. *Reliability engineering: theory and practice*. Springer Science & Business Media, 2013.

[32] Gray, J./ Siewiorek, D. „High-availability computer systems“. In: *Computer* 24.9 (1991), S. 39–48.

[33] Chumash, T. „Obtaining Five “ Nines ” of Availability for Internet Services“. In: 2006.

[34] Raj, P. *Site Reliability Engineering: An Introduction*. Hrsg. von. https://www.chaosnative.com/blog/site_reliability_engineering_an_introduction. o.O., 2021. (Einsichtnahme: 12. 08. 2022).

[35] Vohra, D. „Using Multiple Zones“. In: *Kubernetes Management Design Patterns: With Docker, CoreOS Linux, and Other Platforms*. Berkeley, CA: Apress, 2017, S. 91–116. URL: https://doi.org/10.1007/978-1-4842-2598-1_4.

[36] Spillner, A./ Linz, T./ Schaefer, H. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook, 2014. URL: https://books.google.de/books?id=gEG4BAAAQBAJ.

[37] Kapur, K. C./ Pecht, M. *Reliability engineering*. Bd. 86. John Wiley & Sons, 2014.

[38] Hamill, P. *Unit Test Frameworks*. O'Reilly Media, 2004.

[39] Mathur, S./ Malik, S. *Volume 1 – No. 12 Advancements in the V-Model*.

[40] Crosslake, O. *Smoke tests vs. BVTs*. https://crosslaketech.com/smoke-tests-vs-bvts. 2012. (Einsichtnahme: 12. 07. 2022).

[41] , Hrsg. *Circuit Breaking*. https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/. o.O., 2022. (Einsichtnahme: 06. 07. 2022).

[42] , Hrsg. *Was ist SAP?* https://www.sap.com/germany/about/company/what-is-sap.html. o.O., 2022. (Einsichtnahme: 26. 07. 2022).

[43] SE, S. *What Is Business Entity Recognition?* Hrsg. von. https://help.sap.com/docs/Business_Entity_Recognition/b43f8f61368d455793a241d2b10baeb2/894afc838ee54c0f8c7f7381a9dae27a.html. o.O., 2022. (Einsichtnahme: 26. 08. 2022).

[44] SE, S. *What Is Personalized Recommendation?* Hrsg. von. https://help.sap.com/docs/Personalized_Recommendation/2c2078b9efa84566ac19d44df9625c65/65078007342241bea72c2cdc9c61e6a7.html. o.O., 2022. (Einsichtnahme: 26. 08. 2022).

[45] Community, C. *The Chaos Engineering toolkit for Developers*. Hrsg. von. https://chaostoolkit.org/. o.O., 2022. (Einsichtnahme: 24. 08. 2022).

[46] Community, C. *Chaos Engineering Concepts in the Chaos Toolkit*. Hrsg. von. https://chaostoolkit.org/reference/concepts/. o.O., 2022. (Einsichtnahme: 24. 08. 2022).

[47] Community, J. *Jenkins*. Hrsg. von. https://www.jenkins.io/. o.O., 2022. (Einsichtnahme: 24. 08. 2022).

[48] Slack-Team. *Slack*. Hrsg. von. https://slack.com/. o.O., 2022. (Einsichtnahme: 24. 08. 2022).

[49] Slack-Team. *Is Slack better than email?* Hrsg. von. https://slack.com/why/slack-vs-email. o.O., 2022. (Einsichtnahme: 24. 08. 2022).

[50] R. Fielding M. Nottingham, J. R. *HTTP Semantics*. Hrsg. von. https://www.rfc-editor.org/rfc/rfc9110.html. o.O., 2022. (Einsichtnahme: 24. 08. 2022).

[51] IANA. *Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry*. Hrsg. von. https://www.iana.org/assignments/http-authschemes/http-authschemes.xhtml. o.O., 2022. (Einsichtnahme: 24. 08. 2022).

[52] Reschke, J. *The 'Basic' HTTP Authentication Scheme*. Hrsg. von. https://www.rfc-editor.org/rfc/rfc7617.html. o.O., 2015. (Einsichtnahme: 25. 08. 2022).

[53] M. Jones, D. H. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Hrsg. von. https://www.rfc-editor.org/rfc/rfc6750.html. o.O., 2012. (Einsichtnahme: 25. 08. 2022).

A Cloud

Wie in der Einführung erwähnt, laufen heutzutage viele Anwendungen in der Cloud nach dem Cloud-Computing-Modell. Zuerst soll betrachtet werden, wieso die Anwendungen in der Cloud laufen und wie die Cloud aufgebaut ist. Danach wird die Architektur der darauf laufenden Anwendungen erläutert und erste daraus folgende Problematiken aufgeworfen.

A.1 On-Premises

Bevor dem Wandel zu einem Cloud-Anwendungsmodell liefen die Anwendungen auf den Systemen des Kunden. Das On-Premises-Modell hat folgende Eigenschaften:

Die Software ist an den Kunden angepasst, da die Software auf den Rechnern des Kunden installiert und konfiguriert wird.

Da der Kunde zu Beginn kein Wissen über den Betrieb der Software hat, muss er dieses erst erlangen. Dafür können Schulungen gebucht werden, oder es wird der Software-Hersteller vertraglich verpflichtet, bei der Installation und Konfiguration zu unterstützen.

Sobald die Software auf den Rechnern des Kunden läuft, ist der Kunde für die Wartung der Software verantwortlich. Bei Problemen müssen diese von dem Kunden gelöst werden. Alternativ kann ein Support-Vertrag mit dem Software-Hersteller erstellt werden.

Hierdurch hat das On-Premises-Modell folgende Vor- und Nachteile:

Der Kunde hat eine hohe Kontrolle über die Software und Hardware, auf welcher die Software läuft, da die Software auf eigener Hardware installiert wird. Dies wird durch die initiale Konfiguration bestärkt. Der Kunde kann beispielsweise seine Hardware skalieren, wenn diese nicht ausreichen sollte, da dieser direkten Zugriff zu der Hardware hat.

Prominente Nachteile sind der hohe Verwaltungsaufwand, sowie die Verwaltungskosten die für die Software und Hardware benötigt werden. Die Nachteile gehen mit der hohen Kontrolle einher.

In Abbildung A.1 wird das On-Premises-Modell skizzenhaft veranschaulicht.

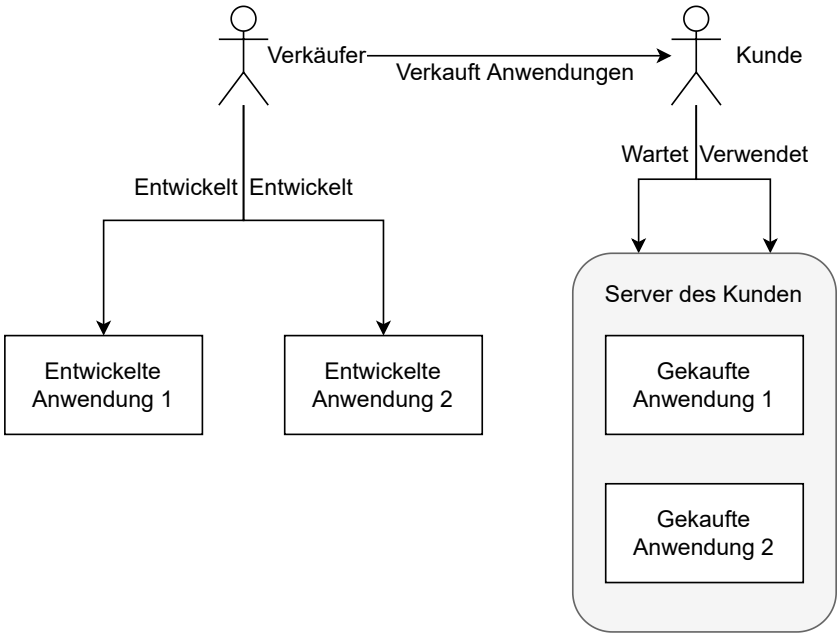

Abbildung A.1: Skizze des On-Premises-Modells

A.2 Monolithische Architektur

Um die Microservice-Architektur zu verstehen, muss zuerst die monolithische Architektur verstanden werden, aus welcher die Microservice-Architektur entstanden ist. Die monolithische Architektur zeichnet sich dadurch aus, dass die komplette Anwendung in ein Programm verpackt wird[2]. Die Anwendung stellt dem Nutzer beispielsweise eine Weboberfläche bereit, greift auf Nutzeranfrage auf eine Datenbank zu und gibt das Ergebnis zurück. Eine Authentifizierung des Nutzers wäre auch in der Anwendung enthalten. Das Programm besteht aus unterschiedlichen Modulen, wie z. B. dem Authentifizierungsmodul, wodurch eine gewisse Abgrenzung zwischen den Modulen besteht. Die Module erlauben es unterschiedlichen Teams, weitestgehend unabhängig voneinander zu programmieren, solange die Schnittstellen zwischen den Modulen nicht verändert werden.

Bei Änderung einzelner Module muss die komplette Anwendung neu an die Kunden ausgeliefert werden[2].

Der Monolith hat eine geringe Flexibilität, da die einzelnen Module der Anwendung aneinander gekoppelt sind. Die Module können nicht unabhängig voneinander ausgeliefert werden. Sollte ein Modul die Schnittstelle aktualisieren, so müssen andere darauf aufbauende Module auch aktualisiert werden. Es kann nicht einfach das aktualisierte Modul ausgeliefert werden und davon abhängige Module die bisherige Version des geänderten Moduls weiterhin verwenden. Zudem führt eine monolithische Architektur zu einer eingeschränkten Wiederverwendbarkeit von einzelnen Elementen. Innerhalb des Monoliths kann Code sehr gut wiederverwendet werden, da der Code in dem Monolith bereits vorhanden ist. Nach außen hin muss Code kopiert werden, es kann nicht die Funktionalität über eine laufende Instanz des ursprünglichen Monoliths wiederverwendet werden. Durch die hohe Kopplung der Module kann nur der Monolith als Ganzes skaliert werden. Einzelne Module können nicht nach Bedarf unterschiedlich stark skaliert werden.

Eindeutiger Vorteil des Monoliths ist sein simpler Aufbau. Durch die einheitliche Auslieferung der Anwendung wird der Auslieferungsprozess vereinfacht. In Abbildung A.2 wird der Aufbau eines Monolithen skizzenhaft veranschaulicht.

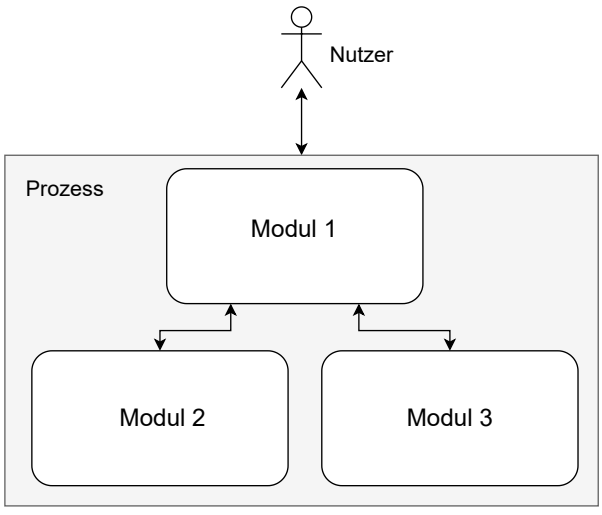

Abbildung A.2: Skizze eines Monoliths

B Systemtheorie

In Sektion 2.1 wurde die Architektur heutiger Cloud-nativer Anwendungen erläutert. In dieser Sektion soll auf die dort identifizierten Systemtypen eingegangen werden. Für jeden Systemtyp sollen die Eigenschaften und die daraus entstehenden Problematiken erläutert werden. Zuerst wird das verteilte System betrachtet, danach folgt das komplexe System.

B.1 Verteiltes System

Ein verteiltes System lässt sich nach Tanenbaum folgenderweise definieren:

> A distributed system is a collection of independant computers that appears to its users as a single coherent system.[30]

Ein verteiltes System zeichnet sich nach Tanenbaum durch den Verbund mehrerer unabhängiger Rechner aus, welche für einen Nutzer wie ein zusammenhängendes System wirken[30]. Seine Definition lässt mehrere Punkte offen. Zuerst ist zu bemerken, dass die Anzahl der Rechner nicht definiert wurde. Ein verteiltes System kann somit aus sehr vielen Rechnern bestehen, z. B. 1000 Rechnern. Je mehr Rechner Bestandteil des verteilten Systems sind, desto komplexer ist die Verwaltung des Systems für einen Menschen, aber auch für ein Programm. Hinzu kommt, dass die Rechner unabhängig sind. Die Rechner müssen nicht alle baugleich sein, das System kann aus beliebigen unterschiedlichen Rechnern bestehen. Dadurch wird der Verwaltungsaufwand eines verteilten Systems erhöht. Die Art der Nutzer wurde zudem nicht definiert, es kann sich somit um Menschen, als auch um Programme handeln.

Ein verteiltes System, welches für den Nutzer als ein einheitliches System erscheint, gilt als transparent[30]. Die Eigenschaft der Transparenz wird in mehrere Aspekte aufgeteilt. Die Aspekte können Tabelle 2.1 entnommen werden.

> Der erste Aspekt ist die Zugriffstransparenz. Die Zugriffstransparenz beschreibt den Architektur-unabhängigen Zugriff auf Daten und die einheitliche Repräsentation der Daten.

Der zweite Aspekt ist der einheitliche Zugriff auf Ressourcen, unabhängig des Orts der Ressourcen.

Der dritte Aspekt *Migrationstransparenz* beschreibt eine Eigenschaft der Ressourcen. Ressourcen können den Ort wechseln, ohne dass der Nutzer dies bemerkt.

Der vierte Aspekt *Umzugstransparenz* spezifiziert den Aspekt *Migration*. Ressourcen können den Ort wechseln, während sie in Betrieb sind, ohne dass der Nutzer dies bemerkt.

Um die Verfügbarkeit von Ressourcen in einem verteiltem System zu erhöhen, können diese vervielfältigt werden. Dabei muss das System nach außen hin wirken, als gäbe es nur eine Kopie der Ressource. Dieser Aspekt wird in der Sektion 2.3 genauer betrachtet.

Obwohl mehrere Nutzer gleichzeitig auf das System zugreifen können, sollen diese sich gegenseitig nicht beeinflussen. Ein Nutzer soll nicht bemerken, dass das System bereits durch andere Nutzer in Anspruch genommen wird. Dies gilt insbesondere für die Last, die durch einen Nutzer im System anfällt und wird in Sektion 2.3 genauer betrachtet.

Der letzte Aspekt *Fehlertransparenz* besagt, dass Fehler des Systems vor einem Nutzer verborgen sein sollen. Ein Ausfall einzelner Komponenten des Systems darf nicht zu einem Komplettausfall führen. Das System muss sich von Teilfehlern erholen, sodass kein Komplettausfall entsteht. Die Eigenschaft, sich von Fehler erholen zu können, wird *Widerstandsfähigkeit* genannt und in Sektion 2.3.5 genauer betrachtet.

Ein verteiltes System versteckt viel Komplexität vor dem Nutzer. Trotz der vielen Eigenschaften soll das System robust gegenüber Fehlern sein. Doch was für Fehler können in einem verteilten System auftreten?

Grundsätzlich gibt es zwei Fehlerarten[4]. Die erste Art an Fehlern, die auftreten kann, sind Latenz basierte Fehler[4], welche besonders auf die Ortstransparenz zurückzuführen sind. Die Rechner eines verteilten Systems können an anderen Enden der Erde physisch stehen. Dadurch hat das System eine durch die Distanz bedingte Latenz. Latenz kann auch durch eine hohe Last eines Rechners auftreten, wenn dieser z. B. viele Nutzeranfragen verarbeiten muss. Wenn ein Rechner stark belastet ist, dann verarbeitet er Nutzeranfragen

langsam und antwortet auf die Anfragen verzögert. Das Ausfallen eines Rechners ist ein weiterer Fehler[30], welcher als Fall einer unendlichen Latenz verstanden wird, da die Nachricht den Rechner nie erreicht.

Die zweite Fehlerart besteht darin, dass der Server fehlerhafte Antworten versendet[4]. Ein Fehler ist das zufällige Versenden von zufälligen Nachrichten. Weitere Fehler beinhalten das Senden von Nachrichten mit falschem Rückgabewert. Dies kann z. B. ausgelöst werden, wenn der Rechner durch fehlerhaften Code in einen invaliden Zustand befördert wird[30].

In einem verteilten System treten unterschiedliche Fehler auf, welche durch unterschiedliche Ansätze erkannt werden. Der erste Ansatz betrachtet das verteilte System aus Sicht eines Nutzers. Es werden Nutzer-ähnliche Anfragen an das System gestellt und die Antworten beobachtet. Sollte die Fehlertransparenz gebrochen werden, wird dies anhand der Antworten erkannt. Der zweite Ansatz betrachtet das System von innen. Dafür müssen entsprechende Berechtigungen vorliegen, sodass das Innere betrachtet werden kann. Durch das Auswerten diverser Systemmetriken wird ein genaueres Bild des Systems erlangt, wodurch Fehler erkannt werden, selbst wenn die Fehlertransparenz nicht gebrochen wird.

B.2 Komplexes System

Ein komplexes System ist ein System, in welchem kleine Handlungen große und unberechenbare Auswirkungen haben können. Das komplexe System lässt sich über seine Eigenschaften charakterisieren. Die von Ladyman und Wiesner [3] definierten Eigenschaften sind in Tabelle 2.2 zu sehen.

Die in der Tabelle genannten Eigenschaften sind Eigenschaften eines allgemeinen komplexen Systems und werden im Folgenden betrachtet. Zudem werden die Eigenschaften des allgemeinen komplexen Systems mit den Eigenschaften eines verteilten und containerisierten Systems verglichen, welches nach der Microservices-Architektur erstellt wurde. Als Beispiel für ein verteiltes und containerisiertes System dient hier K8s. Zur Abkürzung des verteilten und containerisierten Systems, welches nach der Microservices-Architektur erstellt, wird dieses im Folgenden als Cloud-natives System bezeichnet.

Ein komplexes System besteht aus vielen einzelnen Komponenten. Die Komponenten interagieren untereinander, wodurch zahlreiche Interaktionen entstehen. Auch in einem Cloud-nativen System gibt es viele Komponenten, z. B. *Pods*. Die *Pods* kommunizieren untereinander, wodurch zahlreiche Interaktionen entstehen.

Die vielen Komponenten eines komplexen Systems sind zudem unterschiedlich. Die Kommunikation zwischen diesen unterschiedlichen Komponenten wird nicht zentral verwaltet, sondern geschieht dezentral. In einem Cloud-nativen System wird die Kommunikation zwischen den Komponenten nicht dezentral verwaltet, außer die Funktionalität wird separat hinzugefügt. Die Kommunikation wird durch zentrale Komponenten verwaltet, wie z. B. einem Dienst, in welchem laufende Dienste registriert sind. Die verwaltenden Komponenten können aber dafür auf das System verteilt sein.

Unter *Rückmeldung* wird verstanden, dass die Kommunikation nicht nur einseitig stattfindet, sondern dass Komponenten auf Anfragen antworten können und Komponenten auf Antworten reagieren. In einem heutigen Cloud-nativen System ist dies auch der Fall. Die in einem verteilten System laufenden Container kommunizieren untereinander über das Anfrage-Antwort-basierte HTTP-Protokoll[1] und verarbeiten Antworten anderer Dienste.

Interaktionen finden nicht nur zwischen Komponenten eines komplexen Systems statt. Das komplexe System ist nach außen offen und extrinsisch. Interaktionen von außen führen zu vielen weiteren Interaktionen zwischen den Komponenten. Auch diese Eigenschaft ist in einem Cloud-nativen System wiederzufinden. Kunden sind nicht Teil des Systems, greifen aber dennoch von außen auf dieses zu. Durch die Interaktion der Kunden mit dem System wird eine Kette von Interaktionen innerhalb des Systems angestoßen. Der Kunde bekommt zudem meist eine Rückmeldung durch das System.

Ein allgemeines komplexes System ordnet sich spontan. Die Ursache für die spontane Ordnung kann nicht erkannt werden, ohne nähere Informationen über das System zu haben. Die Interaktionen, die zu einer spontanen Ordnung führen, entstehen letztendlich durch die dezentrale Verwaltung und auch externe Interaktionen. In einem Cloud-nativen System ist aus einer äußeren Sicht eine spontane Ordnung

[1] Für eine sichere Kommunikation dient das Hypertext Transfer Protocol Secure (HTTPS)-Protokoll

zu entdecken, wenn Komponenten spontan miteinander kommunizieren, um z. B. für den Kunden eine Aufgabe zu erledigen. Der Kunde weiß vor seiner Anfrage an das System nicht, welche Instanzen eines Dienstes für die Bearbeitung der Aufgabe verwendet werden. Dank Komponenten wie Lastverteilern wird die Kommunikation spontan auf bestimmte Komponenten geleitet.

In der Lektüre werden mehrere Arten der nicht-Linearität genannt. Die im Kontext eines Cloud-nativen Systems relevante Art ist die nicht-Linearität in Bezug auf die Interaktionen zwischen einer Kette an Komponenten. Die Interaktionen zwischen den Komponenten eines komplexen Systems sind nicht linear, da es Rückmeldungen zwischen Komponenten gibt. Die nicht-Linearität ist in einem Cloud-nativen System zu finden, da es in dem Cloud-nativen System Rückmeldung zwischen Komponenten in Form einer HTTP-Antwort gibt.

Eine weitere Eigenschaft eines komplexen Systems ist seine Robustheit. Die Robustheit spielt auf die Robustheit vorhandener Strukturen und vorhandenem Verhaltens gegen Störungen an. Auch Cloud-native Systeme haben eine gewisse Robustheit gegen Störungen. Unter Störungen bleiben Strukturen wie z. B. *Namespaces* erhalten und Dienste kommunizieren weiterhin mit benötigten Diensten. Die Art der Robustheit ist nicht mit einer Robustheit in Bezug auf die Verfügbarkeit der Dienste zu verwechseln. Anwendungen in einem gestörten Cloud-nativen System bleiben nicht verfügbar, außer es wurden Gegenmaßnahmen getroffen. Ein Cloud-natives System zeigt dennoch eine Robustheit der vorhandenen Strukturen und Interaktionen auf.

Die Modularität eines allgemeinen komplexen Systems ist eine weitere Eigenschaft, die ein Cloud-natives System im Ganzen erfüllt. Die Komponenten eines komplexen Systems formen Gruppen, wobei die Komponenten Arbeitsteilung begehen. Die Arbeitsteilung ist analog zu der Unterteilung einer Anwendung in Microservices. Das Bilden von Gruppen in einem Cloud-nativen System trifft zudem zu, da die Dienste unterschiedlicher Anwendungen in *Namespaces* gruppiert werden.

Was die Komplexität eines komplexen Systems erhöht, ist der zeitliche Verlauf. Die Komponenten eines komplexen Systems haben einen Zustand, welcher einen Einfluss auf die Interaktionen zwischen den Komponenten hat. Dies trifft auf Cloud-native Systeme zu, in welchen Komponenten (*Pods*) einen Zustand haben und je nach Zustand sich anders verhalten. Dabei ist die Besonderheit heutiger Dienste, dass

diese oftmals zustandslos entwickelt werden, sodass die Komplexität dieser reduziert wird. Zustandslose Komponenten können auf eine externe Datenbank zugreifen. Da die zustandslose Komponente die gespeicherten Daten zurückgibt, repräsentiert die Datenbank den Zustand der zustandslosen Komponente.

Um unter aktuellen Bedingungen effizient zu funktionieren, muss ein komplexes System sich den Umständen anpassen. Diese Eigenschaft wird als *Anpassungsfähigkeit* bezeichnet. Ein Cloud-natives System zeigt eine gewisse Anpassungsfähigkeit in Bezug auf die Lastverteilung und das Verwalten der *Pods*. Wenn ein *Pod* durch die Umstände ausgelastet ist, dann kann das System sich anpassen und die Last auf einen weiteren *Pod* verteilen. Das System kann sich auch bei Ausfall einer Komponente anpassen, da diese Komponente dann erneut erstellt wird.

Ein Cloud-natives System hat fast alle Eigenschaften eines allgemeinen komplexen Systems. Dem Cloud-nativen System mangelt es lediglich an einer völlig dezentralen Steuerung, anstelle der bisher vorhandenen verteilten und zentralen Steuerung. Dennoch ist das Cloud-native System als ein komplexes System zu bezeichnen.

Die viele komplexe Kommunikation zwischen Komponenten sowie Rückmeldung zwischen den Zustands-vollen Komponenten führt zu unvorhersehbaren Folgen. Trotz der Folgen wird die Struktur und das Verhalten des Systems beibehalten. Aber nicht alle Folgen sind von den Verwaltern des Systems erwünscht. Ein Beispiel wäre der Ausfall mehrerer Komponenten, wodurch ein Teil einer Anwendung nicht mehr für Kunden verfügbar ist. Diese Folge ist ausdrücklich unerwünscht, ist jedoch im Rahmen der natürlichen Folgen eines komplexen Systems.

Für die bleibende Verfügbarkeit Cloud-nativer Anwendungen müssen unerwünschte und unvorhersehbare Folgen eines Cloud-nativen Systems entfernt werden. Auslöser für die Folgen können über Tests entdeckt werden. Die Eignung diverser Tests wird in Sektion 2.4 genauer betrachtet. Zuerst wird das unternehmerische Ziel betrachtet, die Verfügbarkeit Cloud-nativer Anwendungen.

C Abbildungen

In diesem Kapitel werden weitere Abbildungen gegeben, die das Verständnis in die Thematik vertiefen, jedoch nicht dafür notwendig sind. Die Abbildungen sind in dem Hauptteil referenziert oder werden als Notizen genannt.

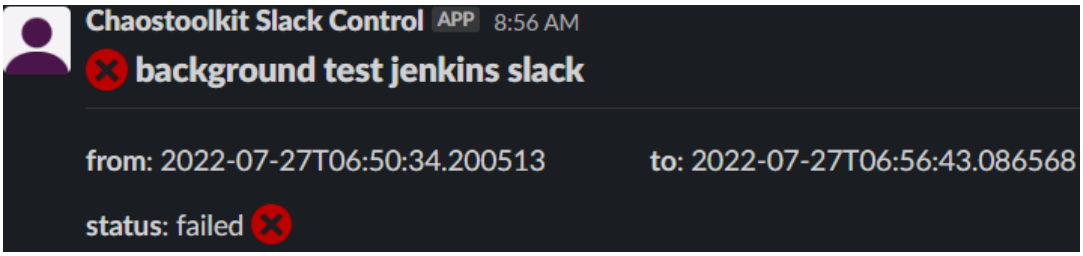

Abbildung C.1: Slack-Nachricht eines erfolglosen Experiments

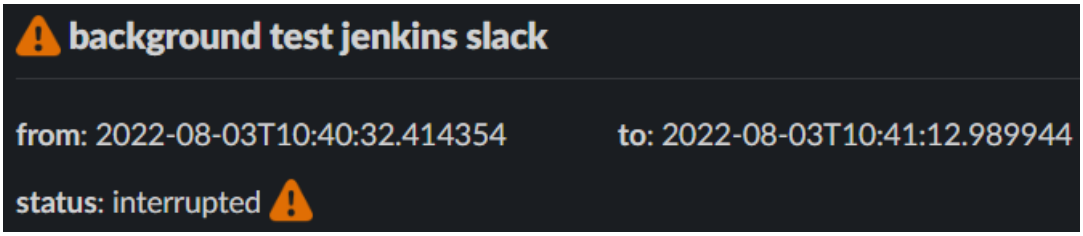

Abbildung C.2: Slack-Nachricht eines abgebrochenen Experiments