

Bachelorarbeit

Jannik Sturhann

Konzeptionierung und Umsetzung eines reproduzierbaren
Continuous Delivery Systems

Jannik Sturhann

Konzeptionierung und Umsetzung eines reproduzierbaren Continuous Delivery Systems

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

Betreuender Prüfer: Prof. Dr. Martin Hübner
Zweitgutachter: Prof. Dr.-Ing. Lars Hamann

Eingereicht am: 14. April 2023

Jannik Sturhann

Thema der Arbeit

Konzeptionierung und Umsetzung eines reproduzierbaren Continuous Delivery Systems

Stichworte

Continuous Delivery, DevOps, Automatisierung, Deployment-Pipeline, Containerisierung, Infrastruktur als Code, Jenkins, Ansible, Docker

Kurzzusammenfassung

Die vorliegende Bachelorarbeit beschäftigt sich mit der Konzeptionierung und Umsetzung eines reproduzierbar provisionierbaren Continuous Delivery (CD) Systems. Am Beispiel eines Demonstrations-Systems wird dabei gezeigt, wie der Provisionierungsprozess eines einfachen verteilten Systems durch das CD System abgebildet werden kann. Das erarbeitete CD System kann effektiv zur Provisionierung von kleineren verteilten Systemen eingesetzt werden und trägt dabei zur Umsetzung des Grundsatzes der effektiven Zusammenarbeit der DevOps-Kultur bei.

Jannik Sturhann

Title of Thesis

Design and implementation of a reproducible continuous delivery system

Keywords

Continuous Delivery, DevOps, Automation, Deployment-Pipeline, Containerization, Infrastructure as Code, Jenkins, Ansible, Docker

Abstract

The present bachelor thesis deals with the conceptual design and implementation of a Continuous Delivery (CD) system that can be provisioned in a reproducible way. Based on a demonstration system, it is shown how the provisioning of a simple distributed system can be integrated and executed through the CD system. The developed CD system can be used effectively for provisioning smaller distributed systems and thus contributes to the implementation of the effective cooperation principle in the DevOps culture.

Inhaltsverzeichnis

Abbildungsverzeichnis	vi
Tabellenverzeichnis	vii
Abkürzungen	viii
1 Einleitung	1
1.1 Zielsetzung	4
1.2 Struktur der Arbeit	4
2 Grundlagen	6
2.1 DevOps und CiCd	6
2.2 Provisionierung	6
2.3 Configuration Management	7
2.4 Infrastructure as Code	9
3 Demonstrations System	10
3.1 Systembeschreibung	10
3.1.1 Anwendungsfall	10
3.1.2 Systemverteilung	11
3.2 Infrastruktur und Provisionierung	13
3.2.1 Provisionierung	14
4 Anforderungen	17
4.1 R1: Abstraktion / Komplexität des CM-Tools verbergen	17
4.2 R2: Reproduzierbarkeit	18
4.3 R3 Change Control Workflow	18
4.4 R4: Verwendung von Open-Source Tools	18
4.5 R5: Versionierung der Konfigurationsschritte des CM-Tools	19
4.6 R6: Möglichkeit zur Integration weiterer Tools	19

5	Technologieauswahl	20
5.1	Auswahl CD-System-Lösung	20
5.1.1	GitLab CI/CD	23
5.1.2	Jenkins	25
5.1.3	Auswahl	26
5.2	Auswahl CM-Tool	27
6	Konzeption und Implementierung	28
6.1	Provisionierung	29
6.1.1	Provisionierung der Ausführungsumgebung	30
6.1.2	Start des Docker-Containers	32
6.1.3	Provisionierung des Jenkins Systems	33
6.2	Integration der Provisionierungsprozesse des Demonstrations Systems . . .	40
6.2.1	Struktur der Jenkins Pipelines	40
6.2.2	Integration von Ansible	43
6.2.3	Integration von Maven und Flyway	45
6.3	Change Control Workflows	45
6.3.1	Änderung der Jenkins Konfiguration	46
6.3.2	Änderungen an einer Pipeline	46
6.3.3	Änderung des jenkins-provisioning-index	47
6.3.4	Hinzufügen oder Entfernen von Pipelines	47
7	Test	48
7.1	Testablauf	48
7.2	Testauswertung	50
8	Fazit und Ausblick	52
8.1	Ausblick	53
	Literaturverzeichnis	54
A	Anhang	57
A.1	cas.yaml	57
A.2	Applikations Provisionierung Jenkinsfile	58
	Selbstständigkeitserklärung	61

Abbildungsverzeichnis

3.1	Ablauf einer Hypertext Transfer Protocol (HTTP) GET Anfrage an /api/rsa-key	10
3.2	Deployment-Diagramm des Demonstrations-Systems	11
6.1	Aufbau der Jenkins Ausführungsumgebung	31
6.2	Start- und Provisionierungs-Prozess des Jenkins-Docker-Containers	34
6.3	Jenkins Web Oberfläche während der Ausführung des “Bootstrap“ Jobs	38
6.4	Eingabemaske bei der Provisionierung des Load-Balancers	42
6.5	Eingabemaske bei der Provisionierung des Applikations-Servers	43
7.1	Eingabemaske der Datenbank-Provisionierung-Pipeline	49
7.2	Eingabemaske der Applikations-Provisionierung-Pipeline	50

Tabellenverzeichnis

3.1	Hostnamen und IP-Adressen der Nodes im Demonstrations-System	14
5.1	Vergleich von CD-Systemen nach Open-Source, Möglichkeit zur Datenein- gabe und CasC Unterstützung	22
6.1	Für das CD-System relevante Git Repositories	29

Abkürzungen

API Application Programming Interface.

CasC Configuration as Code.

CD Continuous Delivery.

CI Continuous Integration.

CI/CD Continuous Integration/Continuous Delivery.

CLI Command Line Interface.

CM Configuration Management.

DSL Domain-specific language.

GUI Graphical User Interface.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

IaC Infrastructure as Code.

JAR Java Archive.

JSON JavaScript Object Notation.

RSA Rivest–Shamir–Adleman.

SaaS Software as a Service.

Abkürzungen

SCP Secure copy protocol.

SQL Structured Query Language.

SSH Secure Shell.

SSO Single Sign-on.

TLS Transport Layer Security.

URL Uniform Resource Locator.

YAML YAML Ain't Markup Language.

1 Einleitung

Continuous Integration/Continuous Delivery (CI/CD) findet mit der zunehmenden Adaption der DevOps Kultur in immer mehr Unternehmen Verwendung [Jokinen, 2020]. Dabei beschreibt CI/CD eine Praktik aus der Software-Entwicklung, bei der Änderungen an einer Software automatisiert gebaut, getestet, und bereitgestellt werden [Halstenberg u. a., 2020, S. 16], [El Khalyly u. a., 2020, S. 1]. Der Lebenszyklus einer Software wird dabei in verschiedene Phasen unterteilt, die angelehnt an die industrielle Fließfertigung aufeinander aufbauen [Halstenberg u. a., 2020, S. 17]. Einige dieser Phasen lassen sich durch Tools automatisieren, wie z.B. die Continuous Integration (CI) Phase in der die Software zusammengebaut und getestet wird oder die Continuous Delivery (CD) Phase, in der ein testbarer Release Kandidat für die Software erstellt wird, welcher schließlich durch eine manuelle Interaktion auf einem Produktivsystem bereitgestellt werden kann [Halstenberg u. a., 2020, S. 19].

Um CI/CD in der Praxis umzusetzen, werden Systeme verwendet, welche die beschriebenen Prozesse ausführen. Diese bieten in der Regel eine Weboberfläche an, um die verschiedenen Prozesse zu starten und deren Ausführung zu überwachen. Dabei werden die Prozesse in Form von Pipelines abgebildet. Pipelines sind mehrstufige Prozesse, die aufeinander aufbauen können. Die einzelnen Stufen können entweder sequenziell oder parallel ausgeführt werden. Das Beschreiben der Prozesse innerhalb der Pipelines erfolgt oftmals durch Scriptsprachen wie beispielsweise Bash.

Damit aber eine Software, wie beispielsweise ein Web Service, mit einer über das Internet erreichbaren Application Programming Interface (API), auf einem Server bereitgestellt werden kann, muss dieser Server zuvor entsprechend konfiguriert werden [Sacks, 2014, S. 61]. Bei dem Beispiel des Web Service wäre es nicht untypisch, dass vor der Inbetriebnahme auf dem Server eine Firewall eingerichtet, Ports freigegeben und Abhängigkeiten installiert werden müssen, die zum ordnungsgemäßen Betrieb der Software notwendig sind. Die Ausführung dieser Konfigurationsschritte, mit der Ziel der Bereitstellung der Software auf einem Server, wird auch Provisionierung genannt [Sacks, 2014, S. 62].

Die manuelle Ausführung dieser Konfigurationsschritte ist ein klassisches Aufgabengebiet der Systemadministration. Hierbei ist es wichtig, dass alle Konfigurationsschritte ausführlich dokumentiert sind und immer in der gleichen Reihenfolge ausgeführt werden. Wenn mehrere Systeme provisioniert werden müssen, kann die manuelle Provisionierung gegebenenfalls viel Zeit in Anspruch nehmen und ist darüber hinaus wegen eventueller Unachtsamkeit fehleranfällig [Morris und Safari, 2020, S. 6]. Es kann durch die manuelle Provisionierung also nicht sichergestellt werden, dass die Konfigurationsschritte immer fehlerfrei und in der selben Reihenfolge ausgeführt werden und der Zustand des Systems nach der Provisionierung reproduzierbar ist.

Um den Problemen der mangelnden Reproduzierbarkeit und des zeitlichen Aufwandes bei der manuellen Provisionierung zu begegnen, hat sich der Infrastructure as Code (IaC) Ansatz entwickelt. Dieser besagt, dass durch automatisierte und reproduzierbare Routinen das Risiko und der notwendige Aufwand bei der Durchführung von Änderungen an einer IT-Infrastruktur reduziert wird [Morris und Safari, 2020, S. 4]. Dabei wird die Infrastruktur (IaC), wie auch die Provisionierung der Infrastruktur (Configuration as Code (CasC)), als maschinenlesbarer “Code“ beschrieben, der von entsprechenden Tools gelesen und ausgeführt werden kann. Der “Code“ dient gleichzeitig auch als Dokumentation, welche stets aktuell ist und durch gängige Versionskontrollsysteme verwaltet werden kann. Dies bietet Einsicht in den Entwicklungsgang der Konfigurationsschritte und eröffnet Möglichkeiten zum Widerruf von fehlerhaften Änderungen. Das ist insbesondere bei dem Betrieb von komplexen Systemen von Bedeutung [Morris und Safari, 2020, S. 37]. Da sich diese Arbeit hauptsächlich mit der Provisionierung von Servern und den Bereitstellungsprozessen von Software auf diesen befasst, ist hier vor allem der CasC Ansatz relevant. Prominente Beispiele für Tools, die CasC umsetzen sind Chef, Ansible oder Puppet [Morris und Safari, 2020, S. 38], [Kostromin, 2020]. Derartige Produkte werden im folgenden als Configuration Management (CM)-Tools bezeichnet.

Viele CM Tools werden über ein Command Line Interface (CLI) bedient. Dabei wird das Tool entweder direkt auf dem lokalen Computer ausgeführt, wie bei Ansible oder auf einem Provisioning Server, wie es z.B. bei Puppet der Fall ist. Soll das Tool lokal ausgeführt werden, muss es zuvor neben den von dem Tool benötigten Abhängigkeiten installiert werden. Die Person, die das CM-Tool verwenden möchte, muss außerdem wissen, welche Parameter an das Tool übergeben werden müssen, um einen bestimmten Provisionierungsprozess auszuführen. Zudem muss die Person über die Verbindungsinformationen zu dem System verfügen, das provisioniert werden soll. Oftmals müssen auch

Passwörter oder andere Geheimnisse bekannt sein, welche zur Authentifizierung auf dem System notwendig sind.

Die genannten Voraussetzungen bei der Ausführung von CM-Tools stellen eine Hürde dar, welche verhindert, dass alle Personen in einem Entwicklungsteam oder einer Abteilung ohne großen Aufwand einen Provisionierungsprozess auf einem System durchführen können. Dies widerspricht dem Grundsatz der effektiven Zusammenarbeit der DevOps Kultur [Halstenberg u. a., 2020, S. 5 ff.], welche von immer mehr Unternehmen adaptiert wird [Díaz u. a., 2021]. Besonders kritisch ist es, wenn nur eine Person in einem Team über das genannte Wissen verfügt. Fällt diese Person unerwartet aus, ist es dem Rest des Teams, mangels Wissens, nicht oder nur mit Umwegen möglich, Systeme zu provisionieren und Software bereitzustellen.

Um es zu ermöglichen, dass alle an einem Projekt beteiligten Personen die Provisionierungsprozesse möglichst einfach ausführen können, bietet es sich an, die Prozesse von einer zentralen Stelle aus zu starten und zu überwachen. CI/CD Systeme mit einer Weboberfläche und der Möglichkeit zur Ausführung von Pipelines, die durch Scriptsprachen implementierte Prozesse abbilden, erfüllen die zuvor genannten Kriterien und wurden bereits erfolgreich in Verbindung mit CM Tools verwendet [Seck u. a., 2022], [Paloposki, 2018]. Über die Pipelines kann dabei das CLI der CM Tools angesprochen werden und über die Web Oberfläche des CI/CD Systems kann die Ausführung der Prozesse überwacht werden. Da CI/CD Systeme mit der fortschreitenden Adaption der DevOps Kultur bereits in vielen Firmen Verwendung finden [Jokinen, 2020] und auch als Werkzeug in der Software-Entwicklung verbreitet sind [Soni, 2015], ist davon auszugehen, dass viele Personen bereits mit derartigen Systemen in Kontakt gekommen sind, was die Einführung erleichtern kann. Im Folgenden wird anstatt von CI/CD Systemen über CD Systeme gesprochen, da sich diese Arbeit primär mit der Bereitstellung von Software und weniger mit der Integration der selbigen befasst.

In dieser Arbeit wird davon ausgegangen, dass das CD-System selbst betrieben und keine Cloud-Lösung verwendet wird. Daraus ergibt sich, dass auch der Server, auf dem das CD-System betrieben wird, zuvor provisioniert werden muss. Dabei muss unter anderem das CD-System selbst und das CM-Tool installiert werden und die Verbindungsinformationen zu den Systemen, die provisioniert werden sollen, müssen zur Verfügung gestellt werden. Darüber hinaus gibt es viele weitere Einstellungen in der Konfiguration eines CD-Systems zu beachten. So müssen z.B. Benutzerkonten verwaltet und die Pipelines, die von dem System ausgeführt werden sollen, angelegt werden.

Es ist daher wünschenswert auch den Provisionierungsprozess des CD Systems zu automatisieren, um diesen reproduzierbar ausführen zu können. In diesem Prozess sollte neben der Installation von Abhängigkeiten auch die gesamte Konfiguration des CD Systems und der damit verbundenen Pipelines stattfinden. Durch einen derartigen Prozess ließen sich Test-Umgebungen für das CD-System ohne großen Aufwand bereitstellen und sollte es beispielsweise einmal zu einem Festplattenausfall bei der Maschine, auf der das CD System betrieben wird, kommen, könnte dieses ohne großen Aufwand wieder hergestellt werden.

1.1 Zielsetzung

Ziel dieser Arbeit ist, ein CD System zu konzeptionieren und umzusetzen, durch welches die Provisionierung von anderen Systemen durchgeführt werden kann. Ein Kernaspekt der Arbeit liegt dabei darauf, das CD-System derart aufzubauen, dass es selbst automatisiert und damit reproduzierbar provisioniert werden kann. Um anschaulich zu zeigen, wie die Provisionierung von anderen Systemen mittels eines CM-Tools in das CD-System integriert werden kann, wird im Rahmen der Arbeit ein einfaches verteiltes System aufgebaut, welches von dem CD System provisioniert wird. Dieses besteht aus einem Applikations-Server, einem Load-Balancer und einer Datenbank.

1.2 Struktur der Arbeit

Zunächst werden Grundlagen vermittelt, welche zum Verständnis der nachfolgenden Kapitel relevant sind. Dabei werden die Themenbereiche DevOps, CI/CD, Configuration Management und IaC bzw. CasC behandelt. Darüber hinaus werden populäre Tools vorgestellt, die für das jeweilige Themengebiet und für die vorliegende Arbeit relevant sind.

Anschließend erfolgt eine Beschreibung des verteilten Systems, das zur Demonstration des CD Systems eingesetzt werden soll. Dieses wird im folgenden "Demonstrations-System" genannt. Es wird auf die Architektur des Systems und die einzelnen Komponenten eingegangen. Außerdem werden die Schritte die bei der Provisionierung des System ausgeführt werden, beschrieben, da diese später durch das CD-System ausgeführt werden sollen.

Nach der Beschreibung des Demonstrations-Systems erfolgt eine kurze Analyse der Anforderungen an das zu entwickelnde CD System. Die Anforderungen werden unter anderem aus dem Provisionierungsprozess des Demonstrations-Systems abgeleitet. Der entstandene Anforderungskatalog wird im weiteren Verlauf der Arbeit verwendet, um Tools auszuwählen, die zur Umsetzung des CD Systems verwendet werden und um die erarbeitete Lösung zu bewerten.

Anhand des Anforderungskataloges wird anschließend eine CD-System-Lösung und ein CM-Tool ausgewählt. Dabei werden verschiedene Möglichkeiten betrachtet und mit Bezug auf verwandte Arbeiten, eine Auswahl getroffen.

Anschließend wird beschrieben, wie das CD System konzeptioniert wird. Dabei wird auf die Umsetzung der Anforderungen eingegangen und es wird im Besonderen beschrieben, wie die Eigenschaft der Reproduzierbarkeit erreicht und die Integration des CM Tools umgesetzt wird.

Zum Abschluss werden die Kernaspekte des erarbeiteten CD Systems noch einmal zusammengefasst und deren Bedeutung in der Praxis beleuchtet. Außerdem wird diskutiert, welche Verbesserungsmöglichkeiten es gibt.

2 Grundlagen

2.1 DevOps und CiCd

DevOps ist eine Methode aus der Softwareentwicklung, in der die enge Zusammenarbeit zwischen dem Development- und dem Operations-Team im Zentrum steht [Halstenberg u. a., 2020, S. 1]. Das Ziel ist, die schnelle und zuverlässige Bereitstellung von Software zu ermöglichen. Eine wichtige Erweiterung von DevOps ist CI/CD. Dabei geht es darum, dass Änderungen an einer Software durch den Einsatz von Pipelines kontinuierlich integriert und gegebenenfalls sogar automatisch bereitgestellt werden. Der Begriff der Pipeline wird verwendet, da in der Regel bis zur Bereitstellung der Software mehrere Prozessschritte nacheinander durchlaufen werden. Durch den Einsatz von Automatisierungstools im Rahmen von CI/CD ist es möglich, Software sehr schnell zu integrieren und bereitzustellen [Halstenberg u. a., 2020, S. 23].

Ein weiterer Punkt der DevOps Kultur ist, dass Prozesse und Workflows standardisiert und zwischen den Teams geteilt werden. Durch die Automatisierung der Prozesse wird es außerdem leichter diese zu teilen, da viele Implementierungsdetails durch ein Automatisierungstool abstrahiert werden können.

2.2 Provisionierung

Das Ziel der Provisionierung ist die Bereitstellung von Ressourcen, wie z.B. Hardware oder Software, um eine bestimmte Umgebung oder einen bestimmten Dienst betreiben zu können [Sacks, 2014, S. 62]. Da sich die vorliegende Arbeit vornehmlich mit der Verteilung von Software befasst, wird in diesem Abschnitt auch nur das entsprechende Teilgebiet der Provisionierung beleuchtet. Speziell ist der Prozess der Bereitstellung und Konfiguration von Software im Kontext einer IT-Infrastruktur, in der Nodes mit Betriebssystemen betrieben werden, relevant.

Die Ausführung von Provisionierungsprozessen kann sowohl manuell als auch automatisiert erfolgen. Im Rahmen dieser Arbeit wird jedoch ausschließlich die automatisierte Provisionierung betrachtet, bei der eine Kommunikation zwischen zwei Computern oder Nodes stattfindet. Die Beziehung zwischen den beiden Nodes lässt sich durch eine Primary/Secondary-Topologie beschreiben. Dabei stellt die Secondary-Node die Partei dar, die provisioniert werden soll, während die Primary-Node die Partei mit dem Wissen darüber darstellt, welche Schritte auf der Secondary-Node ausgeführt werden sollen [Hill, 2021, S. 16-17].

Es existieren zwei grundlegende Ansätze zur Bereitstellung von Software auf den Secondary-Nodes. Bei dem erste Ansatz, der Push-Provisionierung, kennt die Primary-Node alle Secondary-Nodes und weiß für jede Secondary-Node, welche Software auf dieser bereitgestellt werden soll. Im Falle einer Konfigurationsänderung wird diese von der Primary-Node auf die Secondary-Nodes "geschoben". Ansible ist ein Beispiel für ein Tool, das die Push-Provisionierung umsetzt. Im Gegensatz dazu gibt es bei der Pull-Provisionierung eine andere Vorgehensweise. Hier wissen alle Secondary-Nodes, welche Aufgabe sie später erfüllen sollen und stellen eine Anfrage an die Primary-Node, um die Provisionierungsschritte zu erfragen. In bestimmten Zeitintervallen wird immer wieder abgefragt, ob es Konfigurationsänderungen gibt, und sofern dies der Fall, werden diese angewendet [Hill, 2021, S. 17-18]. Puppet und Chef sind Beispiele für Tools, bei denen der Pull-Ansatz empfohlen wird. Es ist zu beachten, dass bei der Pull-Provisionierung oft ein Stück Software, ein sogenannter Agent, auf den Secondary-Nodes installiert werden muss. Diese Software ist für die Kommunikation mit der Primary-Node zuständig und stellt sicher, dass neue Updates geholt und ausgeführt werden. Da beim Push-Ansatz kein Agent erforderlich ist, wird dieser oftmals als unkomplizierter betrachtet. Dies könnte auch ein Grund dafür sein, warum Ansible das am meisten in der Industrie verwendete Automatisierungstool ist [Guerriero u. a., 2019, S. 585].

2.3 Configuration Management

Configuration Management befasst sich mit der Verwaltung von Artefakten und deren Beziehungen innerhalb eines Projekts. Dies umfasst die Speicherung, den Abruf, die eindeutige Identifizierung und die Modifikation von Artefakten, um eine effektive Kontrolle über die Konfiguration zu gewährleisten [Humble und Farley, 2010, S. 31]. Ein wichtiger Grundsatz beim Configuration Management ist, dass jegliche Konfiguration in Form

von Dateien verwaltet werden sollte, die in einem Versionskontrollsystem verwaltet werden [Humble und Farley, 2010, S. 33]. Dies bietet Einsicht in den Entwicklungsgang der Artefakte, bzw. der Konfiguration, was insbesondere bei dem Betrieb von komplexen Systemen von Bedeutung ist [Morris und Safari, 2020, S. 37].

Configuration Management lässt sich grob in Software Configuration Management und Environment Configuration Management unterteilen. Während Software Configuration Management sich auf die Verwaltung von Änderungen an Software-Artefakten während des Entwicklungsprozesses konzentriert, geht es beim Environment Configuration Management um die Verwaltung von Änderungen der Systemumgebungen, in denen die Software ausgeführt wird. In Kontext dieser Arbeit ist vor allem das Environment Configuration Management relevant.

Ein wichtiger Aspekt des Environment Configuration Management ist, die Provisionierung der Systemumgebungen vollständig zu automatisieren, um die Wiederherstellung bekannter, funktionierender Zustände zu ermöglichen und die Kosten und Risiken der manuellen Verwaltung zu reduzieren [Humble und Farley, 2010, S. 50]. Um dieses Ziel zu erreichen, werden oftmals Tools eingesetzt, mit denen die Provisionierung der Systemumgebungen verwaltet wird. Diese werden im folgenden als CM-Tools bezeichnet. Diese Tools ermöglichen die Automatisierung der Provisionierung und stellen Funktionen für die Bereitstellung von Software und die Durchführung von Änderungen bereit. Dabei wird oftmals eine eigene Domain-specific language (DSL) verwendet, in welcher der Zustand der Systemumgebungen in Text-Dateien beschrieben wird.

Ansible ist ein Beispiel für ein solches CM-Tool. Mithilfe von Ansible können Secondary-Nodes, die im Kontext von Ansible “Managed Nodes“ genannt werden, provisioniert, und Software auf diesen bereitgestellt werden [Moser, 2022, S. 11]. Dazu werden Scripts, sogenannte “Playbooks“ erstellt, in denen die Schritte, die bei der Provisionierung ausgeführt werden sollen, in einer DSL beschrieben werden. Das Ausführen der einzelnen Schritte, der sogenannten “Tasks“ wird durch “Modules“ realisiert. Ein “Module“ ist eine wiederverwendbare, in Python geschriebene Funktion, die von einem Playbook aufgerufen wird, um eine Aktion auf einer “Managed Node“ auszuführen.

Die Ausführung eines Playbooks wird “Play“ genannt. Dabei wird Ansible auf einer Primary-Node, im Kontext von Ansible auch “Control Node“ genannt, ausgeführt. Von dort wird eine Secure Shell (SSH)-Verbindung zu einer oder mehreren “Managed Nodes“ hergestellt, die provisioniert werden sollen. Ansible folgt daher dem Grundsatz der Push-Provisionierung. Anschließend werden die einzelnen, im “Playbook“ beschriebenen

Aktionen, auf den “Managed Nodes“ ausgeführt. Alle zu verwaltenden “Managed Nodes“ werden aus einem sogenannten Inventar bezogen. Ein Inventar enthält die Liste aller Nodes, teilt diese ggf. in Gruppen ein und beinhaltet die notwendigen Verbindungsinformationen, etwa in Form von IP-Adressen. Das Inventar kann entweder statisch, z.B. in Form einer YAML Ain’t Markup Language (YAML) Datei oder dynamisch bereitgestellt werden.

2.4 Infrastructure as Code

IaC ist ein Ansatz zur Automatisierung von Infrastruktur auf Basis von Praktiken aus der Softwareentwicklung. Im Kern geht es darum, einen konsistenten und wiederholbaren Prozess für die Bereitstellung und Konfiguration von Systemen zu etablieren, bei dem Änderungen durch “Code“ beschrieben und automatisiert ausgeführt werden [Morris und Safari, 2020, S. 4].

IaC wird verwendet, um die Effizienz und Konsistenz bei der Bereitstellung von IT-Infrastruktur zu verbessern, indem die manuelle Provisionierung von Servern durch automatisierte Prozesse ersetzt wird [Morris und Safari, 2020, S. 4]. Dabei wird die gewünschte Infrastruktur, als maschinenlesbarer “Code“ beschrieben, der von entsprechenden Tools, wie z.B. Terraform gelesen und ausgeführt werden kann. Bei der Ausführung des “Codes“ wird die Infrastruktur automatisch erstellt. Dies bietet den Vorteil, dass die Beschreibung der Infrastruktur in Versionskontrollsystemen verwaltet werden kann, was eine bessere Wartbarkeit der Konfiguration, wie im Abschnitt 2.3 beschrieben, ermöglicht. Außerdem kann so eine IT-Infrastruktur mit sehr geringem Aufwand und in kurzer Zeit, in einer reproduzierbaren Weise erstellt werden.

CasC ist aus dem IaC Konzept entstanden. Während es bei IaC primär um die Bereitstellung von IT-Infrastrukturen geht, beschäftigt sich CasC mit der Automatisierung von Konfigurationsänderungen auf vorhandener Infrastruktur bzw. deren Provisionierung. Das ebenfalls im Abschnitt 2.3 vorgestellte CM-Tool Ansible implementiert CasC, da die durch Ansible verwalteten Konfigurationen der Systemumgebungen ausschließlich durch maschinenlesbaren “Code“ beschrieben werden.

3 Demonstrations System

Im späteren Verlauf der Arbeit wird erläutert, wie das CD-System konzeptioniert, implementiert und getestet wird. Da durch das CD-System Provisionierungsprozesse ausgeführt werden sollen, wird in diesem Kapitel ein einfaches verteiltes System beschrieben, welches durch das zu entwickelnde CD-System provisioniert werden soll. Dieses System dient zur Demonstration des CD-Systems und wird daher als “Demonstrations-System“ bezeichnet. Das Demonstrations-System bietet eine über HTTP erreichbare API an, über die es möglich ist, Rivest–Shamir–Adleman (RSA) Schlüsselpaare zu generieren. Das System besteht aus einem Load-Balancer, zwei Applikations-Servern und einem Datenbank-Server.

3.1 Systembeschreibung

In diesem Abschnitt wird kurz beschrieben, wie das Demonstrations-System funktioniert und aus welchen Komponenten es zusammengesetzt ist.

3.1.1 Anwendungsfall

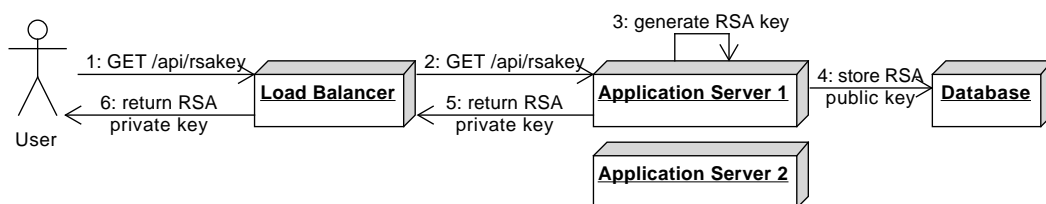


Abbildung 3.1: Ablauf einer HTTP GET Anfrage an `/api/rsakey`

Ein beispielhafter Ablauf einer Anfrage an das System wird in Abbildung 3.1 dargestellt. Zunächst wird eine HTTP GET Anfrage mit dem Pfad `/api/rsakey` an den Load Balancer gesendet (1). Von dort wird die Anfrage an einen der Applikations Server weitergeleitet (2). Die Auswahl des Applikations-Servers erfolgt dabei durch das “Round-robin“ Verfahren [Tanenbaum und van Steen, 2020, S. 144] Auf dem Applikations Server wird nun ein RSA-Schlüsselpaar generiert (3). Der öffentliche Teil des Schlüsselpaares wird anschließend in der Datenbank gespeichert (4), während der private Teil an den Anfragenden zurückgegeben wird (5, 6).

3.1.2 Systemverteilung

Wie in Abbildung 3.1 bereits erkennbar, besteht das Demonstrations-System aus mehreren Nodes: einem Load Balancer, zwei Instanzen eines Applikations Servers und einer Datenbank. Eine Node beschreibt in diesem Kontext ein Element eines verteilten Systems, welches Rechenoperationen ausführt [Tanenbaum und van Steen, 2020, S. 2]. Im diesem Fall sind die Nodes virtuelle Maschinen mit dem Betriebssystem Debian. Die Struktur des Demonstrations-Systems wird in Abbildung 3.2 dargestellt. Im folgenden werden die einzelnen Komponenten des Systems kurz erläutert.

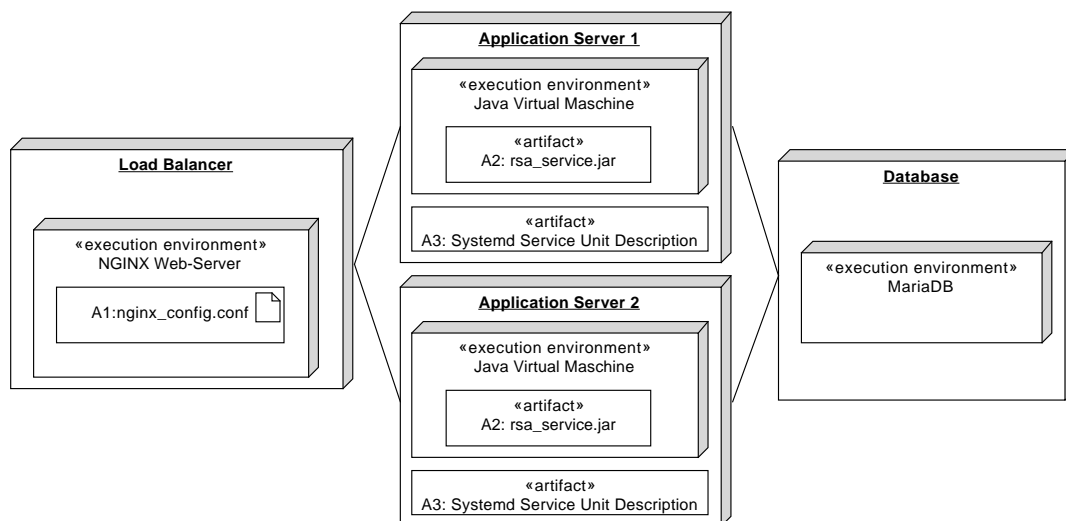


Abbildung 3.2: Deployment-Diagramm des Demonstrations-Systems

Load Balancer

Die Aufgabe des Load Balancers ist, Anfragen entgegenzunehmen und diese an jeweils einen Applikations-Server weiterzuleiten. Dazu wird ein NGINX Webserver verwendet, der als Load Balancer konfiguriert wird. Die Konfiguration erfolgt durch das Definieren sogenannter "Server Blocks"¹ innerhalb der Konfigurationsdatei.

Datenbank

Als Datenbank wurde MariaDB² verwendet. Dabei handelt es sich um ein relationales Datenbankmanagementsystem (RDBMS), das auf der MySQL³ Datenbank basiert und von der Open-Source-Community entwickelt wird.

Application Server

Die Applikation zur Erstellung des RSA-Schlüsselpaars wird mit der Programmiersprache Java implementiert und ist in eine HTTP API Schicht und in eine Datenzugriffsschicht unterteilt. Dabei nimmt die HTTP API Schicht Anfragen entgegen, stößt die notwendigen Operationen an und verwendet dabei auch die Datenzugriffsschicht.

Zur Automatisierung des Java-Build-Prozesses und zur Verwaltung der benötigten Abhängigkeiten wird das Build-Management-Tool Maven⁴ verwendet. Dieses gibt auch die Ordnerstruktur für die Quellcode Dateien im Projekt vor und wird verwendet, um ein ausführbares Java Archive (JAR) Artefakt zu erstellen, welches im Provisionierungsprozess auf die Nodes übertragen wird. Maven wird über ein mitgeliefertes CLI bedient.

Auf den einzelnen Nodes wird das JAR Artefakt der Applikation über eine Systemd Service Unit⁵ verwaltet. Dies bietet die Möglichkeit, die Applikation bei dem Start der Node, bzw. des Betriebssystems, automatisch mit hochzufahren. Darüber hinaus bietet Systemd mit journald⁶ eine Logging Komponente, welche Log-Rotation implementiert. Somit erlaubt Systemd eine einfache Nachverfolgung der Log Ausgaben, was vor allem im Fehlerfall relevant ist.

¹https://www.nginx.com/resources/wiki/start/topics/examples/server_blocks/

²<https://mariadb.org/>

³www.mysql.com

⁴<https://maven.apache.org/>

⁵<https://www.freedesktop.org/software/systemd/man/systemd.service.html>

⁶<https://man7.org/linux/man-pages/man5/journald.conf.5.html>

Die Datenzugriffsschicht stellt zur Erfüllung ihrer Aufgaben eine Verbindung zu der zuvor beschriebenen MariaDB Datenbank her. Die dazu benötigten Verbindungs- und Anmeldeinformationen werden der Applikation bei dem Start als CLI Parameter übergeben. Außerdem ist es erforderlich, dass das Datenschema auf der Datenbank aktuell ist, damit alle Operationen der Datenzugriffsschicht korrekt ausgeführt werden können. Zur Aktualisierung des Datenbank-Schemas wird das Tool Flyway verwendet, welches ermöglicht, Änderungen des Schemas in Structured Query Language (SQL) Skripten zu definieren, die dann durch den CLI Befehl `flyway migrate` auf die Datenbank angewendet werden können. Dies ermöglicht es, den Stand des Datenbank-Schemas zu versionieren, den entstandenen Versionsverlauf zu verfolgen und sicherzustellen, dass Änderungen nachvollziehbar und geordnet ausgeführt werden.

3.2 Infrastruktur und Provisionierung

Am Beispiel des Demonstrations-Systems soll veranschaulicht werden, wie Provisionierungsprozesse in das CD-System integriert und durch dieses ausgeführt werden können. Um diese Provisionierungsprozesse später durch das CD-System abbilden zu können, werden in diesem Abschnitt die einzelnen Schritte betrachtet, die zur Provisionierung des Demonstrations-Systems notwendig sind.

Die Nodes des Demonstrations-Systems werden als virtuelle Maschinen betrieben, um einer Ausführungsumgebung mit realen Computern, die über ein Netzwerk miteinander verbunden sind, möglichst nahe zu kommen. Dabei wird das Tool Vagrant⁷ eingesetzt. Vagrant ermöglicht es, über eine Konfigurationsdatei, virtuelle Maschinen zu konfigurieren. Das starten und stoppen der virtuellen Maschinen erfolgt über ein von Vagrant zur Verfügung gestelltes CLI. Zur Erstellung der in Abbildung 3.2 beschriebenen Nodes, wird eine Datei mit dem Namen `vagrantfile` angelegt, in der alle Nodes konfiguriert werden. Jede Node verwendet das Betriebssystem Debian 11, erhält einen Hostnamen und wird einem privaten Netzwerk hinzugefügt, über welches die Nodes miteinander kommunizieren. Die Zuordnung der Nodes auf Hostnamen und IP-Adressen kann der Tabelle 3.1 entnommen werden. Außerdem wird ein SSH Schlüssel für den Benutzer `vagrant` hinterlegt. Alle virtuellen Maschinen können über den Befehl `vagrant up` erstellt und gestartet werden. Sollten diese später einmal nicht mehr benötigt werden, können sie mit

⁷<https://www.vagrantup.com/>

dem Befehl `vagrant halt` gestoppt und mit dem Befehl `vagrant destroy` entfernt werden.

Name	Hostname	IP-Adresse
Load Balancer	load-balancer	192.168.62.10
Application Server 1	webserver-1	192.168.62.21
Application Server 2	webserver-2	192.168.62.22
Database	database	192.168.62.30

Tabelle 3.1: Hostnamen und IP-Adressen der Nodes im Demonstrations-System

3.2.1 Provisionierung

Im folgenden wird für jede Komponente des Demonstrations-Systems beschrieben, welche Schritte bei der Provisionierung ausgeführt werden müssen. Dies ist notwendig, um später die Integration der Provisionierung des Demonstrations-Systems in das CD-System nachvollziehbar darstellen zu können.

Load Balancer

Bei der Provisionierung des Load Balancers wird zunächst der NGINX Webserver über den APT⁸ Paketmanager installiert. Zur Konfiguration des Web-Servers als Load Balancer⁹ wird die Konfigurationsdatei `nginx_config.cfg` zunächst im Ordner `/etc/nginx/sites-available/` abgelegt und anschließend ein Symlink `/etc/nginx/sites-enabled/default` erstellt, welcher die zuvor erstellte Datei referenziert. Durch das Erstellen des Symlinks wird die Konfigurationsdatei bei dem Start des NGINX Web-Servers automatisch gelesen und als Konfiguration angewendet. Die Konfigurationsdatei enthält unter anderem die Hostnamen der zwei Applikations Server, damit Anfragen an diese weitergegeben werden können. Abschließend wird der NGINX Webserver mit dem Befehl `systemctl restart nginx.service` neu gestartet, womit die Änderungen an den Konfigurationsdateien wirksam werden.

⁸<https://www.debian.org/doc/manuals/debian-faq/pkgtools.de.html>

⁹<https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/>

Datenbank

Bei der Provisionierung der Datenbank wird zunächst MariaDB über APT installiert. In der Konfigurationsdatei von MariaDB wird anschließend festgelegt, dass die Datenbank auch über das Netzwerk erreichbar ist. Dazu wird in der Datei `/etc/mysql/mariadb.conf.d/50-server.cnf` der Wert der Variable `bind-address` auf `0.0.0.0` gesetzt. Innerhalb von MariaDB wird eine Datenbank mit dem Namen `rsa_key_database` angelegt und außerdem ein Konto mit dem Namen `rsa_key_service` angelegt, welches über ein Passwort zur Authentifizierung verfügt. Abschließend wird MariaDB mit dem Shell Befehl `systemctl restart mariadb.service` neu gestartet.

Applikations-Server

Damit die Java-Applikation auf eine Node übertragen werden kann, muss diese zunächst kompiliert werden. Dies geschieht durch das Tool Maven und wird durch den folgenden Befehl gestartet.

```
1 mvn clean compile assembly:single
```

Dabei ist `mvn` der Befehl für Maven. Durch `clean` werden alle zuvor erstellten Build-Dateien entfernt und durch `compile` wird der Quellcode des Projekts kompiliert. Durch die letzte Phase `assembly:single` wird ein ausführbares JAR Artifikat im `target` Ordner erstellt, das alle Java-Abhängigkeiten des Projekts enthält.

Bei der eigentlichen Provisionierung wird zunächst die Java-Laufzeitumgebung mit dem Paketmanager APT auf der Node installiert. Im nächsten Schritt wird die zuvor erstellte JAR Datei auf die Node übertragen.

Da die Ausführung der Applikation durch Systemd erfolgt, wird als nächstes eine Systemd Service Unit angelegt. Dazu wird eine Konfigurationsdatei im Ordner `/etc/systemd/system/` erstellt. Der Name der Datei ist dabei gleichzeitig auch der Name der System Service Unit. In der Konfigurationsdatei ist unter anderem der Startbefehl der Applikation enthalten. Außerdem werden die Verbindungsinformationen sowie Benutzername und Passwort der Datenbank der Applikation als CLI Parameter übergeben. Diese sollten dynamisch in die Konfigurationsdatei eingetragen werden. Nachdem das JAR Artifikat auf die Node übertragen und die Systemd Service Unit angelegt wurde, wird die Applikation mit dem Befehl `systemd restart rsa_service.service` gestartet bzw. neu gestartet.

Zuletzt wird die Datenbankmigration durch das Tool Flyway ausgeführt. Die dazu benötigten SQL Skripte werden zusammen mit dem Quellcode des Projektes verwaltet und versioniert. Der Flyway Migrationsprozess wird über folgenden Befehl gestartet.

```
1 flyway migrate -url=jdbc:mysql://192.168.62.30:3306/rsa_key_database -user=rsa_key_service -password=secret -locations=flyway
```

Durch den Befehl stellt Flyway eine Verbindung zur Datenbank her und führt die im Unterordner `flyway` abgelegten SQL Skripte in sequenzieller Reihenfolge aus. Dabei hat jedes Script eine eindeutige Versionsnummer und es werden nur Skripte ausgeführt, die noch nicht auf die Datenbank angewendet wurden. In der Tabelle `flyway_schema_history`, die von Flyway auf der Datenbank verwaltet wird, ist gespeichert, welche Versionen bereits auf die Datenbank angewendet wurden.

4 Anforderungen

Um bewerten zu können, welche Software-Lösung als Basis für die Umsetzung des geplanten CD-Systems am Besten geeignet ist, muss zunächst herausgearbeitet werden, welche Anforderungen von dem CD-System erfüllt werden müssen.

Zu diesem Zweck wird in diesem Abschnitt ein Anforderungskatalog erstellt, der als Bewertungsbasis für die Auswahl einer CD-Software-Lösung genutzt wird, die wiederum die Basis des zu entwickelnden CD-Systems darstellt. Der Anforderungskatalog dient außerdem als Referenzpunkt bei der Bewertung technischer Entscheidungen, die im Kontext der Konzeptionierung und Umsetzung des CD-Systems getroffen werden. Bei der Erstellung des Anforderungskataloges werden die Eigenschaften des im vorherigen Kapitel beschriebenen Demonstrations-Systems mit einbezogen.

4.1 R1: Abstraktion / Komplexität des CM-Tools verbergen

Gemäß der DevOps Kultur sollen Prozesse von allen an einem Projekt beteiligten Personen möglichst barrierefrei ausführbar sein [Halstenberg u. a., 2020, S. 5], [Amaradri und Nutalapati, 2016, S. 5]. Damit die Hürde, einen Provisionierungsprozess auszuführen, möglichst klein ist, ist es notwendig, CM-Tool spezifische Details zu verbergen. CM-Tool spezifische Details können dabei z.B. die Speicherorte der Dateien sein, in denen die Konfigurationsschritte als "Code" abgespeichert sind oder die Namen der CLI Parameter, die zur korrekten Ausführung eines Prozesses an das CM-Tool übergeben werden müssen oder auch die Kenntnis von zusätzlichen Abhängigkeiten, die auf dem System installiert sein müssen, damit der Provisionierungsprozess ausgeführt werden kann. Das Starten und Überwachen der Provisionierungsprozesse soll über ein Graphical User Interface (GUI) erfolgen, welches möglichst ohne Vorwissen verwendet werden kann und alle Details verbirgt, die zum Ausführen eines Provisionierungsprozesses nicht unbedingt

notwendig sind. Es soll ebenfalls möglich sein, über das genannte GUI, Parameter eingeben zu können, welche bei der Ausführung der Pipeline berücksichtigt werden und ggf. an das CM-Tool weitergegeben werden.

4.2 R2: Reproduzierbarkeit

CD-Systeme sind komplex und bieten eine Vielzahl von Konfigurationsparametern an. Daher sollen alle Schritte, die zur Konfiguration des CD-Systems notwendig sind, durch CasC abgebildet werden, damit der Zustand des CD-Systems zuverlässig reproduziert werden kann Morris und Safari [2020, S. 14]. Dies bietet außerdem den Vorteil, dass das CD-System automatisiert aufgesetzt wird und Änderungen an der Konfiguration des Systems durch ein Versionskontrollsystem erfasst werden können. Zuletzt kann auch Configuration Drift [Morris und Safari, 2020, S. 17] durch CasC verhindert werden, da Änderungen an der Konfiguration nicht manuell durchgeführt sondern automatisiert vorgenommen werden. Bei der Provisionierung sollen auch die Pipelines des CD-Systems erstellt werden.

4.3 R3 Change Control Workflow

Es ist davon auszugehen, dass sich sowohl die durch das CM-Tools verwalteten Konfigurationsschritte, als auch die Pipelines innerhalb des CD-Systems stetig verändern Zampetti u. a. [2021] Aiello und Sachs [2010]. Es ist daher wichtig, dass es einen klar definierten Prozess gibt, mit dem Änderungen verwaltet und auch nachvollzogen, bzw. versioniert werden können.

4.4 R4: Verwendung von Open-Source Tools

Die Umsetzung des CD-Systems soll ausschließlich unter Verwendung von Open-Source Produkten erfolgen, welche frei verwendet werden dürfen, um einen möglichen Vendor Lock-In zu vermeiden Gröschel [2012].

4.5 R5: Versionierung der Konfigurationsschritte des CM-Tools

Sollte einmal ein Fehler bei der Entwicklung der Konfigurationsschritte auftreten, ist es wünschenswert, dass ohne viel Aufwand ein vorheriger Stand dieser Konfigurationsschritte verwendet werden kann Morris und Safari [2020, S. 38]. Daher sollen die Konfigurationsschritte über ein Versionskontrollsystem verwaltet werden. Zudem soll es in der CD-Pipeline die Möglichkeit geben, eine bestimmte Version der Konfigurationsschritte auszuwählen, welche dann bei der Provisionierung verwendet werden.

4.6 R6: Möglichkeit zur Integration weiterer Tools

Das im vorherigen Kapitel beschriebene Demonstrations System verwendet das Build-Management-Tool Maven zur Erstellung von JAR Artefakten und das Tool Flyway zum Verwalten und Aktualisieren des Datenbankschemas. Das CD-System muss alle im vorherigen Kapitel beschriebenen Prozesse, die zur vollständigen Provisionierung der Nodes und zur Bereitstellung der Software auf diesen notwendig sind, abbilden. Dazu ist es nötig, dass auch Tools, wie z.B. Flyway oder Maven, in das CD-System integriert werden können.

5 Technologieauswahl

Wie in der Einführung beschrieben, ist das Ziel dieser Arbeit zu zeigen, wie ein reproduzierbar provisionierbares CD-System aufgebaut werden kann. Über die Pipelines des CD-Systems soll es dabei möglich sein, ein CM Tool auszuführen, welches verwendet wird, um die Provisionierung des in Kapitel 3 beschriebenen Demonstrations-System durchzuführen. Da das System auf der Basis von bereits bestehenden Software-Produkten aufgebaut wird, erfolgt in diesem Kapitel eine Auswahl der Basistechnologien.

Zunächst wird eine CD-System-Lösung gewählt, welche als Basistechnologie für den Aufbau des beschriebenen CD-Systems verwendet wird. Dazu werden zunächst einige populäre CD-System-Lösungen aufgezeigt. Anhand einer Auswahl der im Kapitel 4 definierten Anforderungen, wird die Menge gefiltert, um die Auswahlmöglichkeiten einzuschränken. Die verbleibenden CD-System-Lösungen werden anschließend kurz vorgestellt und es werden verwandte Arbeiten betrachtet, die eine ähnliche Zielsetzung wie die vorliegende Arbeit haben und dabei die jeweilige Technologie verwendet haben. Abschließend wird auf Grundlage der verwandten Arbeiten und der definierten Anforderungen eine der CD-System-Lösungen als Basistechnologie ausgewählt.

Im zweiten Teil des Kapitels wird ein CM-Tool ausgewählt, welches in das CD-Systems integriert wird. Dieser Abschnitt wird bewusst kurz gehalten, da der Fokus dieser Arbeit auf der Umsetzung eines reproduzierbar provisionierbaren CD-Systems liegt und die Integration eines CM-Tools nur eine untergeordnete Rolle spielt.

5.1 Auswahl CD-System-Lösung

Das CD-System soll auf der Basis einer bereits bestehenden CD-System-Lösung aufgebaut werden. Diese muss eine Ausführungsumgebung für die Pipelines und eine Weboberfläche, mit der die Ausführung der Pipelines überwacht werden kann, bereitstellen.

Da es nur wenige Systeme gibt, die sich ausschließlich auf den Themenbereich CD spezialisiert haben, wird im Folgenden eine Reihe von CI/CD-Systemen betrachtet, die neben der Abbildung von CD auch die Umsetzung von CI Prozessen ermöglichen.

Gemäß [Polkhovskiy, 2016, S. 35] sind Jenkins, Atlassian Bamboo, CircleCI, CodeShip, JetBrains TeamCity und TravisCI die am weitesten verbreitetsten CI/CD Lösungen. [Virtanen, 2021, S. 3] vergleicht die CI/CD Systeme Jenkins, Atlassian Bamboo, Azure Pipelines, Google Cloud Build, AWS CodeBuild and CodePipeline, GitHub Actions, GitLab CI/CD und Bitbucket Pipelines. [Mazrae u. a., 2023, S. 12] untersucht die Migrationsbewegungen zwischen den CI/CD-Systemen GitHub Actions, Jenkins, TravisCI, GitLab CI/CD, CircleCI, Azure DevOps, AppVeyor, Hudson, TeamCity, Cruise Control, Drone, Bitbucket Pipelines, Netlify und Atlassian Bamboo.

In jeder der genannten Arbeiten wurde die Menge der betrachteten CI/CD-Systeme bereits auf populäre bzw. viel verwendete Systeme eingeschränkt. Es wird daher davon ausgegangen, dass die Vereinigung der Mengen der jeweils untersuchten CI/CD-Systeme, die meisten populären Systeme beinhaltet. Dabei werden die CI/CD-Systeme Hudson und CruiseControl nicht beachtet, da diese Produkte nicht mehr aktiv weiterentwickelt werden [Mazrae u. a., 2023, S. 13]. Außerdem werden auch Systeme wie z.B. AWX , die auf ein bestimmtes CM-Tool zugeschnitten sind, nicht betrachtet, da die Möglichkeit gegeben werden soll, das CD-System mit unterschiedlichen CM-Tools verwenden zu können.

Aus den verbleibenden System wird nun eine Auswahl für das zu implementierende CD-System getroffen. Es liegt allerdings außerhalb des Umfangs dieser Arbeit, alle genannten CI/CD Systeme ausführlich miteinander zu vergleichen. Daher wird zunächst die Menge der CI/CD Systeme anhand des Ausschluss-Prinzips reduziert. Hierbei wird nach den folgenden Kriterien gefiltert:

Open Source Software Wie in Anforderung R4 (4.4) beschrieben, soll nur Open-Source-Software für die Umsetzung des CD-Systems verwendet werden.

Dateneingabe durch GUI In Anforderung R1 (4.1) wird gefordert, dass spezifische Details des CM-Tools verborgen werden sollen. Dabei ist jedoch nicht davon auszugehen, dass ein Provisionierungsprozess immer mit den genau gleichen Parametern abläuft. Es ist beispielsweise denkbar, dass Passwörter oder die Version der Konfigurationsschritte, die ausgeführt werden sollen, wie in R5 (4.5) beschrieben, durch einen Nutzer eingegeben werden müssen. Daher ist es notwendig, dass durch ein

GUI Eingaben gemacht werden können, die dann dem Provisionierungsprozess zur Verfügung gestellt werden.

Configuration as Code Gemäß R2 (4.2) soll die Reproduzierbarkeit des Systems durch den Einsatz von CasC erfolgen. Dazu ist es erforderlich, dass die CD-System-Lösung die Konfiguration durch maschinenlesbare Dateien oder Scripte ermöglicht. Bei cloud-basierten Lösungen ist die Anforderung erfüllt, sofern es möglich ist, dass die Pipelines des CI/CD Systems über CasC konfiguriert werden können, da die Konfiguration und Provisionierung des Systems von dem jeweiligen Anbieter erfolgt.

CI/CD System	Open Source	Dateneingabe in GUI	CasC
AppVeyor	Nein	Nein	Ja
Atlassian Bamboo	Nein	Ja	Ja
AWS CodeBuild	Nein	Nein	Ja
Azure DevOps	Nein	Ja	Ja
Bitbucket Pipelines	Nein	Ja	Ja
CircleCI	Nein	Ja	Ja
CodeShip	Nein	Nein	Ja
Drone	Ja	Nein	Ja
GitHub Actions	Nein	Ja	Ja
GitLab CI/CD	Ja	Ja	Ja
Google Cloud Build	Nein	Nein	Ja
Jenkins	Ja	Ja	Ja
JetBrains TeamCity	Nein	Nein	Ja
Netlify	Ja	Nein	Ja
TravisCI	Nein	Nein	Ja

Tabelle 5.1: Vergleich von CD-Systemen nach Open-Source, Möglichkeit zur Dateneingabe und CasC Unterstützung

In der Tabelle 5.1 sind alle eingangs genannten CI/CD-Systeme aufgeführt. Für jedes System ist angegeben, ob es sich um ein Open-Source-Projekt handelt, ob die Möglichkeit der Dateneingabe in einem GUI gegeben ist und ob das System durch maschinenlesbare Dateien (CasC) konfiguriert werden kann.

Es ist zu sehen, dass alle genannten Systeme CasC unterstützen. Dabei verwenden die meisten CI/CD-Systeme eine YAML¹ basierte Konfigurationsdatei zum beschreiben der Pipelines. Beispiele hierfür sind GitLab CI/CD², Bitbucket Pipelines³, GitHub Actions⁴ oder TravisCi⁵. YAML ist keine Programmiersprache sondern wird primär zur Serialisierung von Daten eingesetzt. Innerhalb der YAML-Konfigurationsdateien ist es allerdings i.d.R. möglich, Shell Befehle festzulegen, welche später von den Pipelines durchgeführt werden. Dies erlaubt es, auch ohne die Verwendung einer Programmiersprache eine Vielzahl von Anwendungsfällen abzudecken. Die Ausnahme bilden die Systeme Jenkins und Atlassian Bamboo, welche es erlauben, Pipelines in Groovy (Jenkins) oder Java (Bamboo) zu beschreiben. Beide dieser Sprachen basieren auf der Java-Laufzeitumgebung und können als Programmiersprachen betrachtet werden.

Die Auswahl der Systeme schränkt sich erstmals ein, nachdem geprüft wurde, ob das jeweilige CI/CD-System die Eingabe von Daten über das GUI unterstützt. Dabei bieten nur 7 der 15 betrachteten System die genannte Möglichkeit.

Berücksichtigt man zusätzlich, ob es sich bei dem jeweiligen CI/CD-System um ein Open-Source-Projekt handelt, wird die Auswahl weiter eingeschränkt. Insgesamt sind nur vier der untersuchten Systeme unter einer Open-Source-Lizenz verfügbar. Wird nun die Menge der Systeme, die unter einer Open-Source-Lizenz verfügbar sind, mit der Menge der Systeme, die Dateneingabe durch ein GUI ermöglichen, geschnitten, bleiben als einzige Systeme, die beide Kriterien erfüllen, Jenkins und GitLab CI/CD zurück. Im Folgenden soll jedes der beiden Systeme kurz vorgestellt werden.

5.1.1 GitLab CI/CD

Das Gitlab Projekt wurde im Jahr 2011 von dem ukrainischen Software-Entwickler Dmi-triy Zaporozhets als Open-Source Hobby-Projekt gestartet⁶. Das Projekt ist seitdem

¹<https://yaml.org/>

²https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html

³<https://support.atlassian.com/bitbucket-cloud/docs/bitbucket-pipelines-configuration-reference/>

⁴<https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions>

⁵<https://docs.travis-ci.com/user/customizing-the-build/>

⁶<https://about.gitlab.com/blog/2021/11/10/a-special-farewell-from-gitlab-dmitriy-zaporozhets/>

stark gewachsen und 2014 wurde die Firma GitLab inc. von Dmitriy Zaporozhets und Sid Sijbrandij gegründet ⁷.

Heute wird GitLab als “komplette DevOps-Plattform“ vermarktet. Es bietet neben der Kern-Funktionalität, einer Weboberfläche zur Verwaltung von Git Repositories, Möglichkeiten zum Projekt-Management und erlaubt durch GitLab CI/CD, Integrations- und Deployment-Prozesse innerhalb der Weboberfläche abzubilden. GitLab kann entweder auf eigenen Servern betrieben oder von dem Anbieter als Software as a Service (SaaS) bereitgestellt werden. Es gibt dabei die “GitLab Community Edition“, die unter der MIT Lizenz verfügbar ist und die “GitLab Enterprise Edition“, die mehr Features beinhaltet, jedoch restriktiver lizenziert wurde und kostenpflichtig ist.

Die Grundlage von GitLab CI/CD bilden sogenannte “Runner“. Runner sind Nodes, auf denen die “GitLab Runner“ Software installiert ist. Diese Nodes werden dann von der GitLab Instanz als Agenten verwendet, um Pipelines und damit die CI/CD Prozesse auszuführen. Da die Nodes, auf denen die “GitLab Runner“ Software ausgeführt wird, selbst aufgesetzt werden können, lässt sich Anforderung R6 (4.6) gut umsetzen.

Die Konfiguration von GitLab erfolgt durch eine Ruby Konfigurationsdatei. Daher ist es grundsätzlich möglich, GitLab durch CasC zu konfigurieren. In dieser Konfigurationsdatei können aber nur grundlegende Einstellungen getroffen werden und z.B. keine Nutzerkonten oder Projekte angelegt werden. Um auch derartige Aktionen automatisieren zu können, bietet Gitlab die “Rails Console“ an. Dies ist eine interaktive Ruby Shell, mit der es möglich ist, die Gitlab Instanz zu verwalten. Die Anforderung der reproduzierbaren Provisionierung (R2 4.2) ist somit umsetzbar. Allerdings ist es dabei notwendig, Einstellungen über die Ruby Shell zu machen, was komplizierter ist, als nur eine einzelne Konfigurationsdatei anzulegen.

Die Kernfunktionalität von GitLab ist das Verwalten von Git-Repositories. Innerhalb der Repositories kann eine YAML Datei mit dem Namen `gitlab-ci.yml` angelegt werden, welche die Beschreibung der Pipelines enthält, die auf den “Runner“ Nodes ausgeführt werden. Durch die Verwendung der maschinenlesbaren YAML Sprache, können die Pipelines in reproduzierbarer Weise ausgeführt werden (R2 4.2). Da diese Dateien außerdem innerhalb des Git-Repositories verwaltet werden, ist eine Versionierung der Konfigurationsschritte bereits gegeben (R5 4.5). Änderungen werden mit dem Git Workflow umgesetzt, welcher als Grundlage für die Umsetzung der Anforderung R3 (4.3) genutzt werden kann.

⁷<https://about.gitlab.com/company/history/>

```
1 build-job:
2   stage: build
3   script:
4     - echo "Hello, $GITLAB_USER_LOGIN!"
```

Listing 5.1: Beispiel einer GitLab CI/CD Pipeline Definition in YAML

Es wurde bereits gezeigt, dass es möglich ist, CM-Tools innerhalb von GitLab CI/CD auszuführen. [Seck u. a., 2022, S. 4] zeigt, wie GitLab CI/CD zusammen mit dem CM-Tool Ansible verwendet werden kann, um Provisionierungsprozesse zu vereinfachen und Komplexität zu verbergen. Auf dem offiziellen Blog von GitLab ist ebenfalls eine Anleitung zu finden, die beschreibt, wie das CM-Tool Ansible innerhalb von GitLab CI/CD ausgeführt werden kann⁸.

5.1.2 Jenkins

Jenkins ist ein Open-Source CI/CD System, welches in Java implementiert ist. Es ist aus dem Hudson Projekt hervorgegangen, welches von Koshuke Kawaguchi bei Sun Microsystems entwickelt wurde. Hudson sollte nach der Übernahme von Sun Microsystem durch Oracle im Jahr 2010 kommerzialisiert werden. In einer Abstimmung der Hudson Community im Jahr 2011 wurde beschlossen, Hudson zu Jenkins umzubenennen und als Open-Source Projekt weiterzuentwickeln [Richtarik, 2016, S. 14].

Ursprünglich konzipiert als eine CI-Plattform, zielt Jenkins primär darauf ab, die Build- und Test-Prozesse von Software-Projekten abzubilden [Armenise, 2015, S. 24]. Im Laufe der Zeit wurde das Projekt immer wieder erweitert und um neue Funktionalität ergänzt. Mit wachsender Verbreitung der “Extreme Programming“ Methoden wurden später auch Funktionen für die Umsetzung von CD-Prozesse implementiert [Armenise, 2015, S. 25].

Jenkins verfügt über ein Plugin-System, mit dem die Grundfunktionalität des Systems um neue Funktionen erweitert werden kann. Im Jenkins Plugin Index sind zum Zeitpunkt dieser Arbeit über 1800 verschiedene Plugins gelistet⁹. Über Plugins wird in Jenkins beispielsweise die Integration des Versionskontrollsystems Git oder die Berechtigungskontrolle von Benutzerkonten umgesetzt.

⁸<https://about.gitlab.com/blog/2019/07/01/using-ansible-and-gitlab-as-infrastructure-for-code/>

⁹<https://plugins.jenkins.io/>

Über das Plugin “Configuration as Code“¹⁰ ist es möglich, die gesamte Konfiguration des Systems über eine YAML Datei zu definieren. Zusammen mit dem “Job DSL“¹¹ Plugin, welches es ermöglicht, Projekte bzw. Pipelines innerhalb von Jenkins anzulegen, kann das gesamte System über CasC konfiguriert werden. Daher lässt sich mit Jenkins Anforderung R2 (4.2) umsetzen. Steinhauer untersucht, wie eine Jenkins-Umgebung mit CasC aufgesetzt werden kann. Die Ergebnisse seiner Arbeit zeigen, dass eine Jenkins-Instanz voll automatisiert aufgesetzt werden kann, auch wenn in der Arbeit kein Bezug auf das vorher beschriebene “Configuration as Code“ Jenkins-Plugin genommen wurde [Steinhauer, 2017].

Innerhalb der Pipelines, welche durch das “Pipeline“ Plugin bereitgestellt werden, können Prozesse durch Groovy Code, der auf der Java Virtual Maschine ausgeführt wird, modelliert werden. Es gibt eine Reihe von vordefinierten Funktionen, die “Steps“¹² genannt werden. Mithilfe einer solchen Funktion ist es auch möglich, Shell Befehle auszuführen, was für den CLI Aufruf des CM-Tools nötig ist (R6, 4.6). Es wurde bereits am Beispiel des CM-Tools Ansible gezeigt, dass die Integration von CM-Tools in Jenkins möglich ist [Paloposki, 2018].

Durch das “Git“ Plugin ist es möglich, Repositories zu klonen. Es ist außerdem möglich, auch die Pipelines in einem Git Repository zu verwalten und bei der Ausführung zu klonen. Daher ist es hier, ähnlich wie der GitLab CI/CD ebenfalls möglich, mit dem Git Workflow die Anforderungen R3 (4.3) und R5 (4.5) umzusetzen.

5.1.3 Auswahl

Nach einer kurzen Vorstellung beider Technologien und der Beschreibung ihrer wichtigsten Eigenschaften, wird im Rahmen dieser Arbeit Jenkins für die Umsetzung des CD-Systems verwendet. Die Entscheidung für Jenkins basiert zum einen darauf, dass es einfacher einzurichten und zu konfigurieren ist als GitLab. Während bei GitLab das Hauptsystem und die Agenten getrennt sind und mindestens zwei Systeme bereitgestellt werden müssen, um Pipelines auszuführen, genügt bei Jenkins eine einzige Instanz.

Zudem bietet Jenkins umfangreichere Möglichkeiten für die Eingabe von Nutzerdaten in die Pipeline. Durch das “pipeline-input-step“¹³ Plugin werden verschiedene GUI Elemente

¹⁰[ConfigurationasCode](#)

¹¹<https://plugins.jenkins.io/job-dsl/>

¹²<https://www.jenkins.io/doc/pipeline/steps/>

¹³<https://plugins.jenkins.io/pipeline-input-step/>

angeboten, über die Daten eingegeben werden können. Außerdem ist es möglich, während der Ausführung einer Pipeline Daten vom Nutzer abzufragen. Dadurch können Felder mit berechneten Default-Werten gefüllt oder Auswahlmöglichkeiten für ein Drop-Down-Feld dynamisch bestimmt werden. Im Gegensatz dazu ist bei GitLab CI/CD die Dateneingabe nur vor dem Start einer Pipeline möglich und es können nur Strings eingegeben werden.

Darüber hinaus wurde Jenkins von Anfang an als CI-System geplant und später um CD-Funktionalitäten erweitert. GitLab hingegen wurde als System zur Verwaltung von Git-Repositories konzipiert und liegt daher thematisch nicht so nah am zu implementierenden CD-System wie Jenkins. Außerdem ermöglicht die Verwendung von Jenkins die Nutzung verschiedener Git-Hosting-Möglichkeiten, während bei der Verwendung von GitLab CI/CD nur GitLab als Git-Hosting-Lösung in Frage kommt.

5.2 Auswahl CM-Tool

Für die Umsetzung des geplanten CD-Systems muss neben dem CI/CD-System auch ein CM-Tool ausgewählt werden. Die Wahl des CM-Tools ist allerdings weniger wichtig als die Wahl des CI/CD-Systems, da lediglich gezeigt werden soll, dass eine Integration eines CM-Tools in ein CD-System generell möglich ist. Außerdem liegt der Fokus der Arbeit auf der Umsetzung eines CD-Systems. Es ist jedoch wichtig, dass das CM-Tool ein CLI bereitstellt, über das Befehle an das Tool gegeben werden können.

Im praktischen Teil dieser Arbeit wird Ansible als CM-Tool verwendet. Ansible ist das am weitesten in der Industrie verbreitete CM-Tool und ist auf Push-Provisionierung ausgelegt [Guerriero u. a., 2019, S. 585]. Die Entscheidung für Ansible fiel unter anderem deshalb, weil die in dem vorherigen Abschnitt referenzierten Arbeiten, die gezeigt haben, wie ein CM-Tool in ein CI/CD-System integriert werden kann, beide Ansible als CM-Tool verwendet haben und daher davon auszugehen ist, dass eine Integration von Ansible in Jenkins möglich ist [Paloposki, 2018], [Seck u. a., 2022, S. 4].

6 Konzeption und Implementierung

In diesem Kapitel wird dargelegt, wie das CD-System im praktischen Teil der Arbeit realisiert wurde und wie dabei die einzelnen Anforderungen umgesetzt werden.

Dazu soll zunächst beleuchtet werden, wie das CD-System provisioniert und gestartet wird. Anschließend wird erläutert, wie die Provisionierung des Demonstrations-Systems unter Verwendung des CM-Tools Ansible in das CD-System integriert wird. Zum Abschluss werden außerdem die Prozesse erläutert, durch die Änderungen am CD-System und an dem Provisionierungsprozess des Demonstrations-Systems (R3, 4.3) eingepflegt werden können.

An dieser Stelle soll bereits drauf hingewiesen werden, dass Git als Versionskontrollsystem verwendet wird. Dabei werden nicht alle für das CD-System relevante Dateien, in einem einzigen Git Repository verwaltet. Tabelle 6.1 gibt einen Überblick zu den verwendeten Git Repositories. Die Aufteilung der Dateien nach Zuständigkeiten in mehrere Git Repositories ermöglicht es, die unterschiedlichen Bestandteile des CD-Systems unabhängig voneinander zu versionieren.

Git Repository Name	Beschreibung
jenkins-system	Enthält benötigte Dateien zur Provisionierung des Jenkins Systems
jenkins-provisioning-index	Enthält die Definition des Bootstrap Jobs, das Ansible Inventory und die Shared Library für wiederverwendbare Groovy Funktionen
jenkins-application-provisioning	Enthält das Ansible Playbook für die Provisionierung des Applikations Servers und die Beschreibung der dazugehörigen Jenkins Pipeline
rsa-key-service	Enthält den Java Code für die RSA Applikation sowie die Flyway Datenbankmigrationen
jenkins-load-balancer-provisioning	Enthält das Ansible Playbook für die Provisionierung des Load Balancers und die Beschreibung der dazugehörigen Jenkins Pipeline
jenkins-database-provisioning	Enthält das Ansible Playbook für die Provisionierung der Datenbank und die Beschreibung der dazugehörigen Jenkins Pipeline

Tabelle 6.1: Für das CD-System relevante Git Repositories

6.1 Provisionierung

Ein Hauptziel bei der Umsetzung des CD-Systems ist, dass es reproduzierbar provisioniert werden kann (R2, 4.2). Das bedeutet, dass das CD-System durch die automatisierte Anwendung der Konfigurationsschritte immer in den gleichen Konfigurations-Zustand gebracht werden kann. Ein Konfigurations-Zustand bezieht sich dabei auf die Einstellungen des Jenkins Systems, die erstellten Pipelines und die installierten Abhängigkeiten in der Ausführungsumgebung. Um einen Konfigurations-Zustand reproduzierbar erreichen zu können, ist es erforderlich, dass jede Form von Konfiguration an dem System durch das Anwenden von maschinenlesbaren Dateien oder Scripten erfolgt Morris und Safari [2020, S. 10].

Um sicherzustellen, dass kein Configuration Drift [Morris und Safari, 2020, S. 17 ff.] durch manuelle Änderungen im Laufe der Zeit auftritt, wird die Ausführungsumgebung des Jenkins-Servers so aufgesetzt, dass die Datei-basierte Datenbank des Jenkins-Systems

nur so lange besteht, wie die Ausführungsumgebung aktiv ist. Somit gehen alle manuellen Änderungen nach einem Neustart verloren. Folglich wird bei jedem Systemneustart die automatisierte Provisionierung ausgeführt. Dies bietet den Vorteil, dass keine Backups der Datei-basierten Datenbank angelegt werden müssen und ermöglicht außerdem, dass der Zustand des Systems, da er ja als Code definiert ist, durch übliche Versionskontrollsysteme verwaltet werden kann.

In den folgenden Unterabschnitten wird dargelegt, wie die automatisierte Provisionierung des CD-Systems, bzw. des Jenkins-Servers mittels CasC umgesetzt wird.

6.1.1 Provisionierung der Ausführungsumgebung

Die Ausführungsumgebung beschreibt die Umgebung in welcher das Jenkins System ausgeführt wird. Im Kontext des CD-Systems umfasst diese das Betriebssystem, die auf diesem installierte Software-Komponenten und die dazugehörige Konfiguration.

Als Tool, um die Ausführungsumgebung zu verwalten, wird Docker verwendet, da bereits gezeigt wurde, wie Docker zur effektiven Provisionierung von Jenkins eingesetzt werden kann [Steinhauer, 2017, S. 73]. Docker ermöglicht es, die Ausführungsumgebung für Programme mittels sogenannter Container vom Host-System zu isolieren. Container können durch die Verwendung von Dockerfiles in reproduzierbarer Weise erstellt und auf individuelle Bedürfnisse zugeschnitten werden. Die Verwendung des Tools bietet sich daher für die Umsetzung der Anforderung R2 (4.2) an.

Bei dem Start des Docker-Container kann zudem durch die Verwendung von Docker-Volumes gesteuert werden, welche Daten nach einem Neustart innerhalb des Docker-Containers erhalten bleiben und welche verloren gehen. Dies wird genutzt, um sicherzustellen, dass die Datei-basierte Datenbank des Jenkins-Systems bei jedem Neustart verworfen wird.

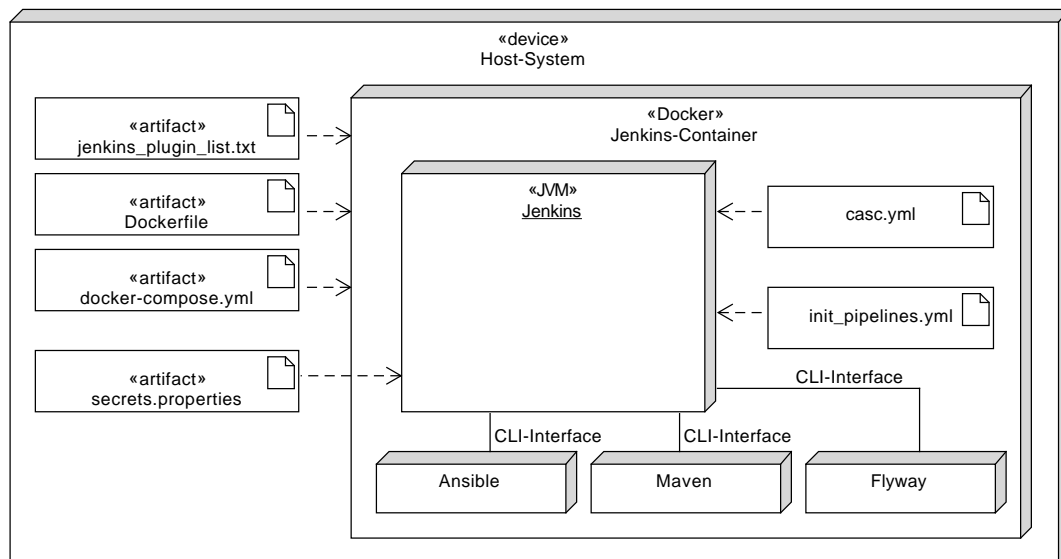


Abbildung 6.1: Aufbau der Jenkins Ausführungsumgebung

Abbildung 6.1 zeigt die Verteilungssicht des CD-Systems und bietet eine Übersicht, welche Konfigurationsdateien und Skripte im Build-Prozess des Docker-Containers und bei dem Starten des Jenkins-Systems benötigt werden. Pfeile, die von einem Artefakt auf eine Komponente zeigen, weisen darauf hin, dass das jeweilige Artefakt zur Provisionierung der jeweiligen Komponente benötigt wird.

Als Basis für den Docker-Container, der die Ausführungsumgebung des Jenkins-Systems darstellt, wird eines der offiziellen Jenkins-Docker-Images verwendet¹. Diese Docker-Images basieren wiederum auf den Docker-Images des Debian Betriebssystems, enthalten aber bereits die Java-Laufzeitumgebung und das Jenkins-System. Konkret wurde das Docker-Image `jenkins/jenkins:lts-jdk11` gewählt, da dieses Langzeitunterstützung bietet.

Die weitere Provisionierung des Docker-Containers findet im Docker-Build-Prozess statt. Alle hierfür benötigten Dateien werden im Git Repository “jenkins-system“ (siehe Tabelle 6.1) verwaltet. Es wird eine `Dockerfile` Datei verwendet, in der alle weiteren Provisionierungsschritte durch Code beschrieben werden². Bei dem Docker-Build-Prozess werden zunächst alle benötigten Pakete über den Paketmanager APT im Container installiert.

¹<https://hub.docker.com/r/jenkins/jenkins>

²<https://docs.docker.com/engine/reference/builder/>

Anschließend wird Ansible über den Python-Paketmanager pip und schließlich das Build-Management-Tool Maven und Flyway installiert. Die Installation der Tools Flyway und Maven ist notwendig für die Integration der Provisionierungsprozesse des Demonstrations Systems und auch Grundlage für die Umsetzung der Anforderung R6 (4.6).

Innerhalb des Docker-Build-Prozesses werden auch die vom Jenkins System benötigten Plugins installiert. Hierfür wird zunächst die Datei `jenkins_plugin_list.txt`, welche die Liste der benötigten Plugins enthält, in den Build-Container kopiert. In der Datei werden die Namen der Plugins und jeweils die zu installierende Versionsnummer angegeben. Anschließend wird das CLI von Jenkins aufgerufen, um alle in der Liste enthaltenen Plugins zu installieren (siehe Listing 6.1.1).

```
1 jenkins-plugin-cli --plugin-file jenkins_plugin_list.txt
```

Listing 6.1: Aufruf des Jenkins CLI, um benötigte Plugins zu installieren

Neben der Liste der Plugins werden außerdem die Dateien `init_pipelines.groovy` und `cas.c.yaml` in den Docker-Container kopiert. Es wird außerdem die Umgebungsvariable `CASC_JENKINS_CONFIG` gesetzt, welche den Pfad der `cas.c.yaml` Datei als Wert enthält. Die genannten Dateien und Variablen werden später bei der Provisionierung des Jenkins Systems benötigt.

6.1.2 Start des Docker-Containers

Die Ausführungsumgebung bzw. der Docker-Container wird mithilfe von Docker-Compose gestartet und gestoppt. Docker-Compose ist ein Tool, das es ermöglicht, Docker-Container über eine YAML Datei zu konfigurieren und über ein CLI zu starten. Ohne Docker-Compose müssten alle Parameter manuell bei jedem Start des Containers über das CLI übergeben werden. Es ist wichtig zu erwähnen, dass es neben Docker-Compose auch andere Lösungen zur Verwaltung von Docker-Containern gibt, wie zum Beispiel Kubernetes, Docker Swarm oder Cloud-Anbieter wie AWS. Aus Gründen der Einfachheit wurde in dieser Arbeit jedoch Docker-Compose gewählt.

```
1 version: "3.3"
2 services:
3   jenkins:
4     build:
5       context: .
6       dockerfile: Dockerfile
7     volumes:
```

```
8     - ./secrets.properties:/run/secrets/secrets.properties
9     environment:
10      - 'PIPELINE_REPO_URL=git@github.com:jannst/jenkins-provisioning-
        index.git'
11      - 'INVENTORY_REPO_URL=git@github.com:jannst/jenkins-ansible-
        inventory.git'
12     ports:
13      - "8080:8080"
```

Listing 6.2: Aufruf des Jenkins CLI, um benötigte Plugins zu installieren

Listing 6.1.2 zeigt den Inhalt der `docker-compose.yml` Datei, die von Docker-Compose verwendet wird. In dem `build` Abschnitt wird definiert, wie der Container gebaut wird. In diesem Fall wird das zuvor beschriebene `Dockerfile` referenziert, welches im selben Ordner wie die `docker-compose.yml` Datei abgelegt ist. Außerdem werden hier Umgebungsvariablen definiert, die später vom Jenkins-System verwendet werden und ein Docker-Volume angelegt, das verwendet wird, um die `secrets.properties` Datei innerhalb des Containers zur Verfügung zu stellen. Im letzten Abschnitt der Datei wird ein Port-Mapping definiert, welches den Port mit der Nummer 8080 nach außen öffnet.

Um den Docker-Container zu starten, bzw. diesen zu bauen, muss zunächst das Git Repository "jenkins-system" (siehe Tabelle 6.1) geklont werden. Anschließend wird innerhalb des Repositories der CLI Befehl `docker-compose up` ausgeführt. Um den Container wieder zu entfernen, kann der Befehl `docker-compose down` verwendet werden.

6.1.3 Provisionierung des Jenkins Systems

Wie bereits erwähnt, wird die dateibasierte Datenbank des Jenkins-Systems bei jedem Neustart der Ausführungsumgebung verworfen und nicht dauerhaft gespeichert. Dadurch gehen alle Daten des Systems bei einem Neustart verloren.

Um dieses Problem zu lösen, wird das Jenkins-System bei jedem Start der Ausführungsumgebung neu provisioniert, indem die maschinenlesbaren Konfigurationsdateien ausgeführt werden. Dadurch werden alle erforderlichen Einstellungen und Pipelines angewendet bzw. angelegt, um den Zustand des Systems wiederherzustellen.

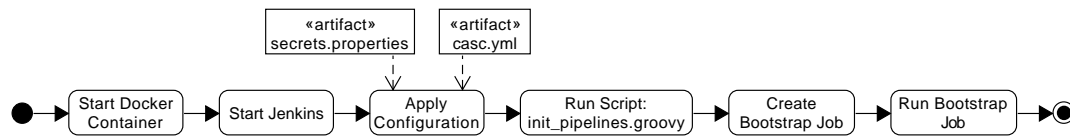


Abbildung 6.2: Start- und Provisionierungs-Prozess des Jenkins-Docker-Containers

Die Abbildung 6.2 zeigt den Ablauf des Provisionierungs-Prozesses, der bei jedem Start des Jenkins-Systems durchlaufen wird. Der Prozess lässt sich grob in zwei Abschnitte unterteilen.

Im ersten Abschnitt wird die Ausführungsumgebung gestartet und die Konfiguration des Jenkins-Systems mithilfe des “Configuration as Code“ Plugins durchgeführt. Im zweiten Abschnitt werden die Pipelines für die Provisionierung des Demonstrations-Systems erstellt und konfiguriert. Dieser Abschnitt beginnt mit der Ausführung des `init_pipelines.groovy` Skripts.

Konfiguration des Jenkins Systems

Nachdem der Docker-Containers und damit auch das Jenkins System gestartet wurde, wird innerhalb des Jenkins Systems das “Configuration as Code“ Plugin aufgerufen (siehe Abb. 6.2). Das Plugin wurde während der Provisionierung der Ausführungsumgebung bereits im Container installiert (siehe Abschnitt 6.1.1). Die ebenfalls in diesem Schritt gesetzte Umgebungsvariable `CASC_JENKINS_CONFIG` wird nun von dem Plugin gelesen. Der Wert der Variable ist der Pfad der `casc.yml` Datei, welche ebenfalls bei der Provisionierung der Ausführungsumgebung in den Container kopiert wurde. Anschließend wird die `casc.yml` Datei gelesen und die dort vorgenommenen Konfigurationen werden durch das “Configuration as Code“ Plugin angewendet. Der gesamte Inhalt der `casc.yml` Datei ist in Anhang A.1 zu finden. Die Dokumentation zu den Konfigurationsmöglichkeiten ist auf der Internetseite des Plugins³ zu finden. In der folgenden Auflistung werden die relevantesten Einstellungen, die in der `casc.yml` Datei getroffen werden, erläutert. Die Zeilenangaben beziehen sich dabei auf den zuvor genannten Anhang.

³<https://plugins.jenkins.io/configuration-as-code/>

Anlegen Admin Konto Es wird ein Benutzerkonto mit dem Namen “admin“ angelegt. Das Passwort für dieses Konto wird aus der Variable `JENKINS_ADMIN_PASSWORD` geladen (Zeile 1-7).

Anlegen eines Seed Jobs Die Datei `init_pipelines.groovy`, welche bei der Provisionierung der Ausführungsumgebung im Container abgelegt wurde, wird als Seed Job referenziert (Zeile 8-9).

Anlegen von Anmeldeinformationen Die SSH Schlüssel, die für die spätere Anmeldung auf dem Demonstrations System und für GitHub benötigt werden, werden als benannte “domainCredentials“ angelegt. Später können diese über den vergebenen Namen in den Pipelines verwendet werden (Zeile 10-27).

Anlegen einer Shared Library Um die CLI Integration von Ansible über wiederverwendbare Funktionen abbilden zu können, wird eine eigene Shared Library verwendet, deren Funktionen in einem dem Git Repository “jenkins-provisioning-index“ (siehe Tabelle 6.1) verwaltet werden. Damit die Funktionen der Library verwendet werden können, muss diese in der Konfigurationsdatei angegeben werden (Zeile 33-43).

Innerhalb der `caso.yml` Datei werden Variablen referenziert, die Geheimnisse enthalten, wie z.B. die Variable `JENKINS_ADMIN_PASSWORD`, welche das Passwort für das Admin Konto enthält. Diese Informationen werden in die Datei `secrets.properties` ausgelagert, damit sie nicht zusammen mit Inhalten der `caso.yml` Datei in ein Versionskontrollsystem gelangen. Über die Verwendung von Docker-Volumes wird die `secrets.properties` Datei zur Laufzeit in dem Container verfügbar gemacht (siehe Listing 6.1.2 Zeile 8). Das “Configuration as Code“ Plugin sucht automatisch an dem Pfad `/run/secrets/secrets.properties` nach einer “secrets.properties“ Datei⁴, weshalb diese an genau diesem Pfad durch das Docker-Volume zur Verfügung gestellt wird. In der `secrets.properties` Datei werden Schlüssel-Wert Tupel angegeben, die dann in der `caso.yml` Datei referenziert und aufgelöst werden können.

Als letzter Schritt wird das `init_pipelines.groovy` Script von dem “Configuration as Code“ Plugin angestoßen. Dieses, in der Programmiersprache Groovy implementierte Script, wird dazu verwendet, den “Bootstrap“ Job mittels der JobDSL Syntax anzulegen und diesen im Anschluss zu starten. Die Aufgabe des “Bootstrap“ Job ist alle anderen Pipelines zu erzeugen. Zum Starten eines Jobs bietet JobDSL die Funktion

⁴<https://github.com/jenkinsci/configuration-as-code-plugin/blob/master/docs/features/secrets.adoc>

`queue(jobName: String)`⁵, um einen Job der Warteschlange von auszuführenden Jobs hinzuzufügen. Leider ist es in der praktischen Umsetzung nicht gelungen, diese Funktion zielführend einzusetzen, da eine Race-Condition innerhalb des Jenkins-Systems vorliegt, deren Auswirkung ist, dass die Job-Warteschlange geleert wird, bevor der Seed Job ausgeführt werden kann. Daher wurde eine alternative Lösung erarbeitet, bei der die Jenkins Groovy API verwendet wird, um den Seed Job zu starten (siehe Listing 6.3).

```
1 ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor
   ()
2 service.schedule({
3     try {
4         def seedJobItem = Jenkins.getInstance().getItemMap().get(seedJobName)
5         if(seedJobItem) {
6             println 'Scheduling seed job'
7             seedJobItem.scheduleBuild()
8         }
9     } catch(Exception e) {
10        e.printStackTrace()
11    }
12 }, 5, TimeUnit.SECONDS)
```

Listing 6.3: Workaround zum Starten des "Bootstrap" Jobs nach 5 Sekunden

Anlegen der Jenkins Pipelines

Zur Provisionierung der Komponenten des Demonstrations Systems (Load-Balancer, Applikations-Server und Datenbank), werden drei Pipelines im Jenkins System angelegt. Anforderung R5 (4.5) fordert, dass die Konfigurationsschritte des CM-Tools, also in diesem Fall die Ansible Playbooks, versionierbar sind, und bei der Ausführung der Konfigurationsschritte eine bestimmte Version der Playbooks ausgewählt werden kann. Daher wird jedes Ansible Playbook in einem eigenen Git Repository verwaltet (siehe Tabelle 6.1). Obwohl diese Aufteilung auf den ersten Blick umständlich erscheinen mag, bietet dieses Vorgehen durch die Versionierbarkeit einen entschiedenen Vorteil und hat sich bereits in der Praxis bewährt [Bertrand u. a., 2020, S. 876].

Es erschien sinnvoll, neben den Playbooks auch die Definition der Pipelines innerhalb der entsprechenden Git Repositories zu verwalten. Alle Repositories, die einen Provisionierungsprozess des Demonstrations Systems beinhalten ("jenkins-application-provisioning",

⁵<https://github.com/jenkinsci/job-dsl-plugin/wiki/Job-DSL-Commands>

“jenkins-load-balancer-provisioning“, “jenkins-database-provisioning“) enthalten daher mindestens folgende Dateien.

Jenkinsfile Definition der Jenkins Pipeline.

seed.groovy JobDSL Script zum Anlegen der Pipeline.

playbook.yaml Ansible Playbook, das von der Pipeline ausgeführt wird.

Es ist zu erkennen, dass in jedem der Repositories eine Datei mit dem Namen `seed.groovy` enthalten ist. Diese Datei enthält JobDSL Syntax und wird dazu verwendet, die jeweilige Pipeline automatisiert innerhalb des Jenkins Systems anzulegen.

Um alle Pipelines zu erstellen, muss jedes Repository geklont und das jeweilige `seed.groovy` JobDSL Script ausgeführt werden. Dieser Prozess, der ebenfalls zur Provisionierung des CD-Systems gehört, wird im zuvor beschriebenen “Bootstrap“ Job ausgeführt, dessen einzige Aufgabe es ist, alle anderen Jobs bzw. Pipelines anzulegen. Zur Verdeutlichung ist in Abbildung 6.3 die Liste der verfügbaren Jobs bzw. Pipelines innerhalb der Jenkins Web Oberfläche zu sehen.

1: Während der Ausführung des Bootstrap Jobs

S	W	Name ↓	Last Success	Last Failure	Last Duration	
⋮	☀	Bootstrap	N/A	N/A	N/A	▶

2: Nach der Ausführung des Bootstrap Jobs

S	W	Name ↓	Last Success	Last Failure	Last Duration	
✓	☀	Bootstrap	35 sec #1	N/A	16 sec	▶
⋮	☀	Database Provisioning	N/A	N/A	N/A	▶
⋮	☀	Load Balancer Provisioning	N/A	N/A	N/A	▶
⋮	☀	Webserver Provisioning	N/A	N/A	N/A	▶

Abbildung 6.3: Jenkins Web Oberfläche während der Ausführung des “Bootstrap“ Jobs

Direkt das dem Start des Jenkins Systems ist nur der “Bootstrap“ Job verfügbar (1). Dieser wird automatisiert durch das `init_pipelines.yml` Script gestartet (siehe Listing 6.3). Nach der erfolgreichen Ausführung des “Bootstrap“ Jobs sind auch die Pipelines, die zur Provisionierung des Demonstrations Systems verwendet werden, verfügbar (2).

Der “Bootstrap“ Job wird in dem `init_pipelines.yml` Script durch das JobDSL Plugin angelegt und anschließend ausgeführt. Das Starten des `init_pipelines.yml` Script ist dabei die letzte Aktion des “Configuration as Code“ Plugins bei dem hochfahren des Systems (siehe Abbildung 6.2). Es ist zu beachten, dass der Code des “Bootstrap“ Jobs in dem Git Repository “jenkins-provisioning-index“ (siehe Tabelle 6.1) verwaltet wird und bei dem Anlegen des Jobs lediglich die URL dieses Repositories bekannt ist (siehe Listing 6.4, Zeile 9).

```
1 pipelineJob("Bootstrap") {
2   definition {
```

```
3     cpsScm {
4         scm {
5             git {
6                 branch("main")
7                 remote {
8                     credentials('git_access')
9                     url('git@github.com:jannst/jenkins-provisioning-index
10                        .git')
11                 }
12             }
13         }
14     }
15 }
16 }
```

Listing 6.4: Anlegen des “Bootstrap“ Jobs durch JobDSL Syntax (init_pipelines.groovy)

Bei dem Ausführen des “Bootstrap“ Jobs wird zunächst das “jenkins-provisioning-index“ Repository geklont und anschließend das `Bootstrap.Jenkinsfile` Groovy Script ausgeführt, welches Jenkins Pipeline Syntax⁶ beinhaltet. Anschließend wird die Datei `pipelines.json` gelesen, die ebenfalls im genannten Repository verwaltet wird und die URLs zu allen referenzierten Pipeline Repositories im JavaScript Object Notation (JSON) Format enthält (Siehe Listing 6.5).

```
1 [
2     {"url": "git@github.com:jannst/jenkins-load-balancer-provisioning.git"},
3     {"url": "git@github.com:jannst/jenkins-webserver-provisioning.git"},
4     {"url": "git@github.com:jannst/jenkins-database-provisioning.git"}
5 ]
```

Listing 6.5: pipelines.json

Danach wird jedes der referenzierten Repositories geklont und es wird innerhalb der Repositories nach einer `seed.groovy` Datei gesucht, welche, falls vorhanden, ausgeführt wird. Innerhalb der `seed.groovy` Dateien werden die eigentlichen Pipelines zur Provisionierung des Demonstrations Systems, wie sie in Abbildung 6.3 (2) zu sehen sind, durch JobDSL Syntax (analog Listing 6.4) beschrieben. Nachdem alle `seed.groovy` Scripts der referenzierten Repositories ausgeführt wurden und damit alle Pipelines erstellt sind, ist die Provisionierung bzw. der Start des Jenkins Systems abgeschlossen.

⁶<https://www.jenkins.io/doc/book/pipeline/syntax/>

6.2 Integration der Provisionierungsprozesse des Demonstrations Systems

In diesem Abschnitt erfolgt die Beschreibung der Integration der im Kapitel 3 beschriebenen Provisionierung der Demonstrations-Systems in das CD-System. Eine umfassende Behandlung jedes einzelnen Provisionierungsprozesses wird aufgrund ihrer hohen Ähnlichkeit vermieden, um Wiederholungen zu vermeiden. Stattdessen werden die wesentlichen Aspekte der Integration herausgearbeitet und anhand der Provisionierung einzelner Komponenten des Demonstrations-Systems erläutert.

6.2.1 Struktur der Jenkins Pipelines

Pro Provisionierungsprozess wird im Jenkins System eine Pipeline angelegt, die über die Web Oberfläche gestartet werden kann (siehe Abbildung 6.3). Der Ablauf der Pipeline wird dabei durch Groovy Code beschrieben. Wird eine Pipeline ausgeführt, so wird zunächst das zugehörige Repository (Siehe Tabelle 6.1) mit der Jenkinsfile Datei geklont. Anschließend wird die Jenkinsfile Datei mittels des "Workflow" Jenkins Plugins ausgeführt. Dieses Plugin stellt dabei die "Pipeline Syntax"⁷ zur Verfügung. In Listing 6.6 ist als Anschauungsbeispiel der Inhalt des Jenkinsfile dargestellt, welches zur Provisionierung des Load Balancers verwendet wird.

```
1 node {
2     @Library('jenkins-cli-adapter')_
3     stage("execute Ansible") {
4         checkout scm
5         def ansiblePlaybookTags = ansibleGetPlaybookTags(playbook: "playbook.
6             yaml")
7         def playbookGitTags = gitGetTagMap()
8         def ansibleInventoryPath = ansibleFetchInventory(inventoryRepoUrl:
9             env.PROVISIONING_INDEX_REPO, credentialsId: 'git_access')
10        def userInput = ansibleUserInput(playbookVersions: playbookGitTags,
11            playbookTags: ansiblePlaybookTags)
12        checkout scm: [$class: 'GitSCM', branches: [[name: userInput.
13            playbookVersion ]], poll: false
14        runPlaybook(
15            credentialsId: 'provisioning_key',
16            inventory: ansibleInventoryPath,
17            playbook: "playbook.yaml",
```

⁷<https://www.jenkins.io/doc/book/pipeline/syntax/>

```
14     tags: userInput.ansibleTagArgument
15     )
16 }
17 }
```

Listing 6.6: Jenkinsfile zur Provisionierung des Load Balancers

Am Anfang der Pipeline (Zeile 2) wird die Shared Library eingebunden, welche in Abschnitt 6.1.3 kurz beschrieben wurde. Diese stellt die folgenden Funktion zur Verfügung: `ansibleFetchInventory`, `ansibleGetPlaybookTags`, `ansibleUserInput`, `fetchProvisioningIndex`, `gitGetTagMap`, `runPlaybook`, welche ebenfalls im Rahmen dieser Arbeit implementiert wurden. Der Quellcode der Funktionen ist auf der CD zu finden, welche dieser Arbeit beigelegt ist.

Bei der Ausführung der Pipeline wird zunächst das Ansible Playbook analysiert, um eine Liste der darin verwendeten Tags zu erstellen (Zeile 5). Daraufhin wird eine Übersicht aller Git-Tags und -Branches des zugehörigen Repositorys "jenkins-load-balancer-provisioning" erzeugt (Zeile 6). Zudem wird das Repository "jenkins-provisioning-index" geklont, welches das von Ansible benötigte Inventar der Nodes enthält (Zeile 7). Anschließend wird eine Eingabemaske auf der Web-Oberfläche dargestellt, in der ein Git-Tag bzw. -Branch des "jenkins-load-balancer-provisioning" Repositorys ausgewählt und die auszuführenden Tags des Ansible Playbooks spezifiziert werden können (Zeile 8). Schließlich wird der gewünschte Stand des Repositorys ausgecheckt (Zeile 9) und das Playbook ausgeführt (Zeile 10-15).

Load-Balancer Provisioning

Playbook_Version

origin/main (2023-03-11) ▾

Execute_All_Tags
If not checked, select the tags/roles that should be executed from the list below

install_nginx

configure_nginx

restart_nginx

Proceed **Abort**

Abbildung 6.4: Eingabemaske bei der Provisionierung des Load-Balancers

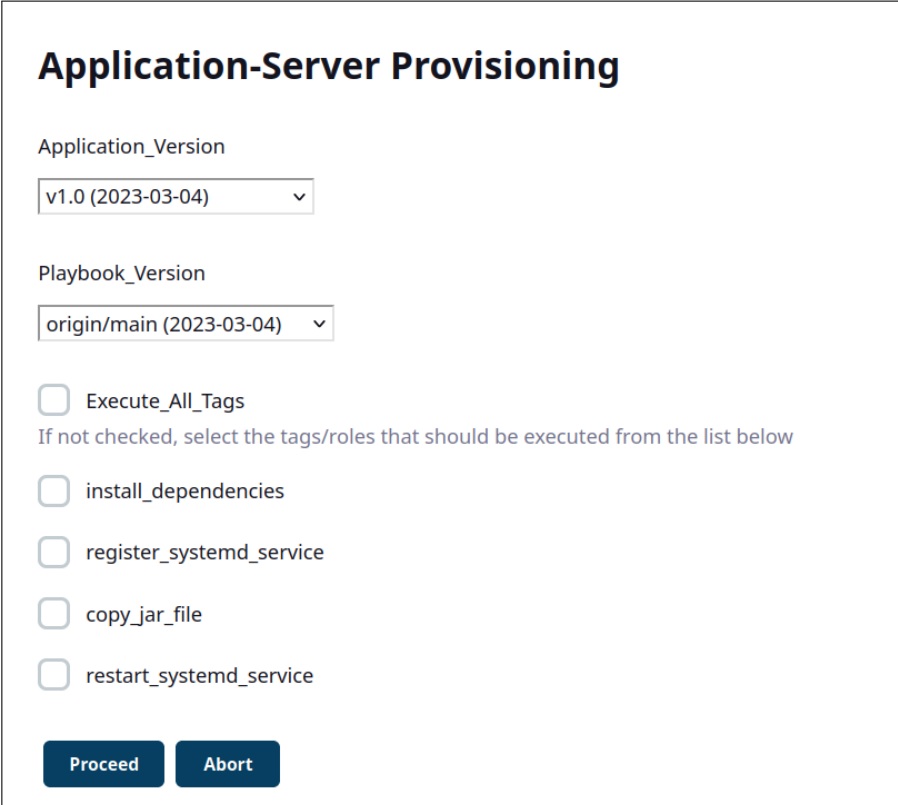
In Abbildung 6.4 ist die zuvor beschriebene Eingabemaske zu sehen, die von der Pipeline auf der Web Oberfläche angezeigt wird. Es ist zu erkennen, wie hier die auszuführenden Tags und die Version des Ansible Playbooks ausgewählt werden kann. Dabei ist es, wie in Anforderung R1 (4.1) gefordert, nicht notwendig, tieferes Verständnis von Ansible, bzw. dem zugehörigen CLI zu besitzen, da alle Eingaben über das GUI gemacht werden.

Die Verwendung von Dropdown-Elementen in dem GUI, die mit einer Liste von Auswahlmöglichkeiten befüllt werden, vereinfacht die Dateneinabe abermals. Die Liste der Auswahlmöglichkeiten wird dabei dynamisch in der Pipeline erstellt. Diese praktische Möglichkeit bietet neben Jenkins keines der in der Technologieauswahl betrachteten CI/CD-Systeme. In Abbildung 6.4 wird diese Option verwendet, um eine Version, also einen Git-Tag oder -Branch des Git Repositories “jenkins-load-balancer-provisioning“ auszuwählen, in dem das Ansible Playbook gespeichert wird. Durch die Möglichkeit zur Auswahl einer Version des Repositories und damit des Ansible Playbooks wird Anforderung R5 (4.5) umgesetzt.

Die Auswahl von einzelnen Tags des Ansible Playbooks in dem GUI, ermöglicht es, nur eine Teilmenge der Konfigurationsschritte des Playbooks auszuführen. Diese Möglichkeit

ist in allen umgesetzten Pipelines gegeben. Dies ist besonders praktisch, wenn beispielsweise nach einem Update der Applikation nur die Applikation auf den entsprechenden Nodes installiert, nicht aber die Java-Laufzeitumgebung erneut installiert werden soll.

Weiterhin ist es möglich, die genannte Eingabemaske individuell zu erweitern. So wurde z.B. bei der Pipeline zur Provisionierung der Applikations Servers (siehe Anhang A.2) ein weiteres Feld eingebaut, durch welches es möglich ist, die Version des “rsa-key-service“ Git Repositories auszuwählen (siehe Abbildung 6.5).



Application-Server Provisioning

Application_Version
v1.0 (2023-03-04) ▾

Playbook_Version
origin/main (2023-03-04) ▾

Execute_All_Tags
If not checked, select the tags/roles that should be executed from the list below

install_dependencies

register_systemd_service

copy_jar_file

restart_systemd_service

Proceed **Abort**

Abbildung 6.5: Eingabemaske bei der Provisionierung des Applikations-Servers

6.2.2 Integration von Ansible

Bei allen Provisionierungsprozessen werden die Konfigurationsschritte, letztlich über das CM-Tool Ansible ausgeführt. Ein Beispiel für einen Konfigurationsschritt ist etwa die Installation von Software-Paketen über APT oder das Kopieren von Artefakten auf die

Nodes des Demonstrations Systems. Die Konfigurationsschritte eines Provisionierungsprozesses werden in Ansible Playbooks verwaltet. Listing 6.7 zeigt, wie der Provisionierungsprozess des Applikations Servers (siehe Abschnitt 3.2.1) durch ein Ansible Playbook abgebildet wird. An dieser Stelle sind auch Tags der einzelnen Konfigurationsschritte definiert, die in der Eingabemaske (siehe Abb. 6.5) ausgewählt werden können (Zeilen 12,15,22,25). Für die Provisionierung der anderen Komponenten des Demonstrations Systems gibt es ebenfalls je ein Ansible Playbook. Diese sind dem Listing 6.7 aber sehr ähnlich und werden daher hier nicht näher erläutert.

```
1 - name: Application-Server Provisioning
2   hosts: application-server
3   become: true
4   tasks:
5     - name: Installing dependencies
6       package:
7         name: "{{item}}"
8         state: present
9         update_cache: yes
10      with_items:
11        - openjdk-17-jre
12      tags: 'install_dependencies'
13     - name: setup systemd unit file
14       template: src=rsa_service.j2 dest=/etc/systemd/system/rsa_service.
15         service
16       tags: 'register_systemd_service'
17     - name: Copy file with owner and permissions
18       ansible.builtin.copy:
19         src: "{{local_application_jar}}"
20         dest: "/opt/rsa-key-generator-service.jar"
21         owner: vagrant
22         group: vagrant
23       tags: 'copy_jar_file'
24     - name: start rsa-key-generator-service
25       systemd: state=restarted name=rsa_service daemon_reload=yes
26       tags: 'restart_systemd_service'
```

Listing 6.7: Ansible Playbook zur Provisionierung der Applikations Server

Innerhalb der Pipelines für die Provisionierungsprozesse wird die Ausführung der Ansible Playbooks durch die Wrapperfunktion `runPlaybook` angestoßen (siehe z.B. Listing 6.6 Zeile 10-14). Diese wird innerhalb der selbst geschriebenen Shared Library, die im Git Repository "jenkins-provisioning-index" verwaltet wird, definiert. Die gesamte Implemen-

tierung der Funktion ist auf der beigefügten CD zu finden. Die Aufgabe der Funktion besteht darin, die Parameter, die von der Ansible CLI benötigt werden, anzunehmen und diese in das von Ansible benötigten Format umzuwandeln. Nachdem das Ansible CLI von der Wrapperfunktion aufgerufen wurden, können die Kommandozeilenausgaben über die Jenkins Web Oberfläche verfolgt werden.

Zuletzt wird bei der Ausführung eines Ansible Playbooks ein Inventar der Nodes benötigt, die provisioniert werden sollen. Diese Information werden in der YAML Datei `inventory.yaml` abgelegt, welches in dem Git Repository "jenkins-provisioning-index" verwaltet wird. Bei jeder Ausführung einer Pipeline wird somit auch immer das genannte Repository ausgecheckt, damit auf das Inventar zugegriffen werden kann.

6.2.3 Integration von Maven und Flyway

Für die vollständige Provisionierung und Bereitstellung des Applikations Servers ist es notwendig, dass die Applikation durch das Build-Management-Tool Maven gebaut und das Datebankschema durch das Tool Flyway migriert wird. Daher wird in Anforderung R6 (4.6) die Möglichkeit zur Integration weiterer Tools gefordert. Im Abschnitt zur Provisionierung der Ausführungsumgebung (6.1.1) wurde bereits beschrieben, wie diese Tools im Docker-Container installiert werden.

Die Integration dieser Tools in die Pipelines erfolgt über die `sh` Funktion⁸ in der Jenkins Pipeline, welche einen Shell Befehl aufruft. Listing 6.2.3 zeigt, wie das Build-Management-Tool Maven innerhalb der Provisionierungs-Pipeline des Application Servers aufgerufen wird.

```
1 sh(script: "mvn clean compile jar:jar")
```

6.3 Change Control Workflows

Wie in Anforderung R3 (4.3) beschrieben, ist davon auszugehen, dass sich sowohl die durch Ansible verwalteten Playbooks sowie die Pipelines, welche die Playbooks ausführen, stetig verändern Zampetti u. a. [2021] Aiello und Sachs [2010]. Es ist ebenfalls zu

⁸<https://www.jenkins.io/doc/pipeline/steps/workflow-durable-task-step/#sh-shell-script>

erwarten, dass die Konfiguration des Jenkins Servers zu einem Zeitpunkt geändert, Pipelines gelöscht bzw. hinzugefügt oder das von Ansible verwendete Inventar der Geräte geändert werden muss. In diesem Abschnitt werden daher die Prozesse, welche die beschriebenen Szenarien abdecken, kurz umrissen.

6.3.1 Änderung der Jenkins Konfiguration

Wie in Abschnitt 6.1 beschrieben, sind die Dateien, über die der Jenkins-Server und die Ausführungsumgebung provisioniert werden, im Git Repository "jenkins-system" gespeichert. Soll etwas an der Konfiguration geändert werden, muss zunächst dieses Repository ausgecheckt werden. Anschließend können die Dateien bearbeitet werden. Nach der Bearbeitung empfiehlt es sich, den Docker-Container neu zu bauen, damit alle Artefakte wie z.B. die `cas.c.yml` Datei in aktueller Version im Container vorliegen. Wird das Docker-Compose Tool verwendet, kann der Container durch folgenden Befehl gestartet und dabei neu gebaut werden.

```
1 docker-compose up --build
```

Da das Jenkins System bei jedem Start neu provisioniert wird, ist der Prozess mit dem Start des Containers, und dem commit der Änderungen in Git, abgeschlossen.

6.3.2 Änderungen an einer Pipeline

Jede Pipeline des CD-Systems wird in einem eigenen Git Repository verwaltet (siehe Abschnitt 6.1.3). Dabei wird bei jeder Ausführung der Pipeline das jeweilige Git Repository vom Jenkins System geklont. Durch dieses Vorgehen ist es sehr einfach, Änderungen an den Pipelines auf dem Jenkins System bereitzustellen. Es muss lediglich das Git Repository lokal ausgecheckt, die Änderungen vorgenommen und die Änderungen durch einen Git "commit" verteilt werden. Wird danach die entsprechende Pipeline auf dem Jenkins System gestartet, sind die Änderung direkt verfügbar. Ein Neustart des Systems ist nicht erforderlich.

6.3.3 Änderung des jenkins-provisioning-index

In dem “jenkins-provisioning-index“ Git Repository wird das von Ansible verwendete Inventar der Nodes, der “Bootstrap“ Job und die wiederverwendbaren Funktionen verwaltet. Der Prozess, um Änderungen an den jeweiligen Dateien auf dem Jenkins bereitzustellen, ist analog zu dem zuvor beschriebenen Prozess zur Änderung der Pipelines. Es müssen lediglich die Dateien verändert und der neue Stand des Repositories durch einen Git “commit“ verfügbar gemacht werden.

6.3.4 Hinzufügen oder Entfernen von Pipelines

Soll eine neue Pipeline hinzugefügt werden, muss zunächst ein neues Repository angelegt werden. In dem neuen Repository wird eine `seed.groovy` Datei mit JobDSL Syntax angelegt, die beschreibt, wie der neue Job erstellt werden soll (analog Listing 6.4). Außerdem muss eine `Jenkinsfile` Datei angelegt werden, die beschreibt, was genau bei der Ausführung der Pipeline passiert. Gegebenfalls wird außerdem ein Ansible Playbook erstellt, das von der Pipeline aufgerufen wird.

Nachdem die Dateien angelegt und durch einen Git “commit“ auf dem Repository verfügbar gemacht wurden, muss dem “Bootstrap“ Job mitgeteilt werden, dass bei der Provisionierung des Jenkins Systems ein weiteres Git Repository ausgecheckt und das enthaltene `seed.groovy` Script ausgeführt werden muss. Dazu wird die Uniform Resource Locator (URL) des Git Repositories in der Datei `pipelines.json` (siehe Listing 6.5), welche im Git Repository “jenkins-provisioning-index“ verwaltet wird, eingetragen. Wird nun das Jenkins System neu gestartet oder der “Bootstrap“ Job manuell ausgeführt, wird die neue Pipeline automatisch angelegt.

Soll eine Pipeline entfernt werden, genügt es, den entsprechenden Eintrag aus der `pipelines.json` Datei zu entfernen. Hierbei ist es allerdings notwendig, dass das Jenkins System neu gestartet wird, damit die Pipeline auch von dem GUI verschwindet.

7 Test

Um zu verifizieren, dass das CD-System wie erwartet funktioniert, wurde es gemäß des im vorherigen Kapitel beschriebenen Konzeptes aufgebaut und es wurde ein Integrations-Test mit dem im Kapitel 3 beschriebenen Demonstrations-System durchgeführt. Dabei wurden alle Komponenten des Demonstrations-Systems durch das CD-System provisioniert und es wurde sichergestellt, dass das Demonstrations-System nach der Provisionierung verwendet werden konnte.

Dazu wurden zunächst die im Abschnitt 3.2.1 beschriebenen Schritte, die bei der Provisionierung ausgeführt werden müssen, in Ansible Playbooks abgebildet. Anschließend wurden diese Ansible Playbooks gemäß des im Abschnitt 6.2 beschriebenen Vorgehens, in das CD-System integriert.

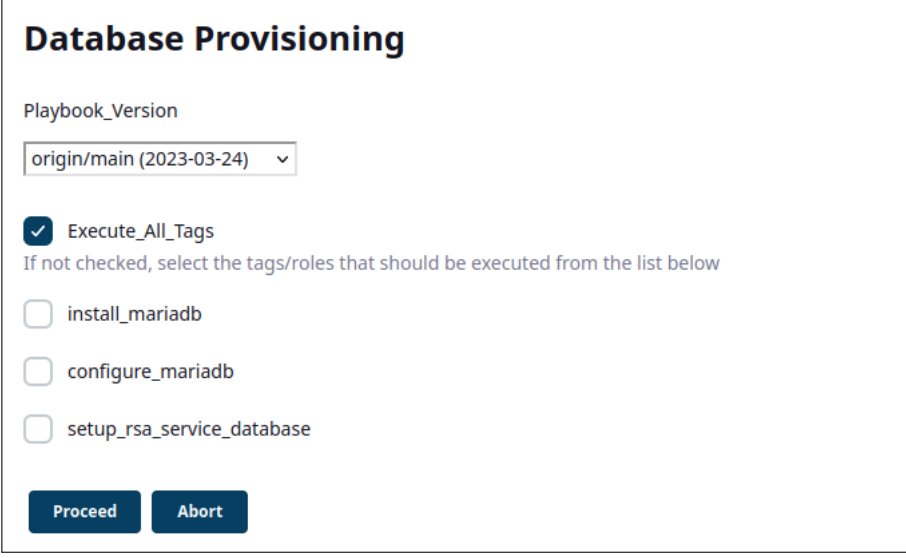
7.1 Testablauf

Zunächst wurden die Nodes des Demonstrations-System mit dem Befehl `vagrant up` hochgefahren. Nach dem Start der Nodes war auf diesen nichts weiter als das Debian 11 Betriebssystem installiert.

Anschließend wurde das CD-System gestartet bzw. provisioniert. Dazu wurde das Git Repository "jenkins-system" ausgecheckt und die Datei `secrets.properties` erstellt, in welche die SSH Schlüssel für den Zugriff auf die Nodes und den Zugriff auf die Git Repositories eingetragen wurden. Anschließend wurde die Provisionierung des CD-Systems mit dem Befehl `docker-compose up` gestartet, wie im Abschnitt 6.1.2 beschrieben. Durch diesen Befehl wurde auch die automatische Provisionierung des CD-Systems angestoßen. Die gesamte Provisionierung des Systems, inklusive der Erstellung der Jenkins Pipelines, dauerte ungefähr 4 Minuten.

Anschließend wurden die Komponenten des Demonstrations-Systems provisioniert. Dazu wurden die Pipelines "Database Provisioning", "Load Balancer Provisioning" und

“Application-Server Provisioning“ nacheinander über die Weboberfläche des Jenkins-Systems gestartet. Bei der Auswahl der auszuführenden Ansible Tags wurde bei allen Pipelines die Option “Execute All Tags“ gewählt. Außerdem wurde bei allen Playbooks der Stand des “main“ Branches des jeweiligen Git Repositories gewählt. Als Beispiel ist in Abbildung 7.1 die ausgefüllte Eingabemaske der Datenbank-Provisionierung-Pipeline zu sehen.



Database Provisioning

Playbook_Version

origin/main (2023-03-24) ▾

Execute_All_Tags
If not checked, select the tags/roles that should be executed from the list below

install_mariadb

configure_mariadb

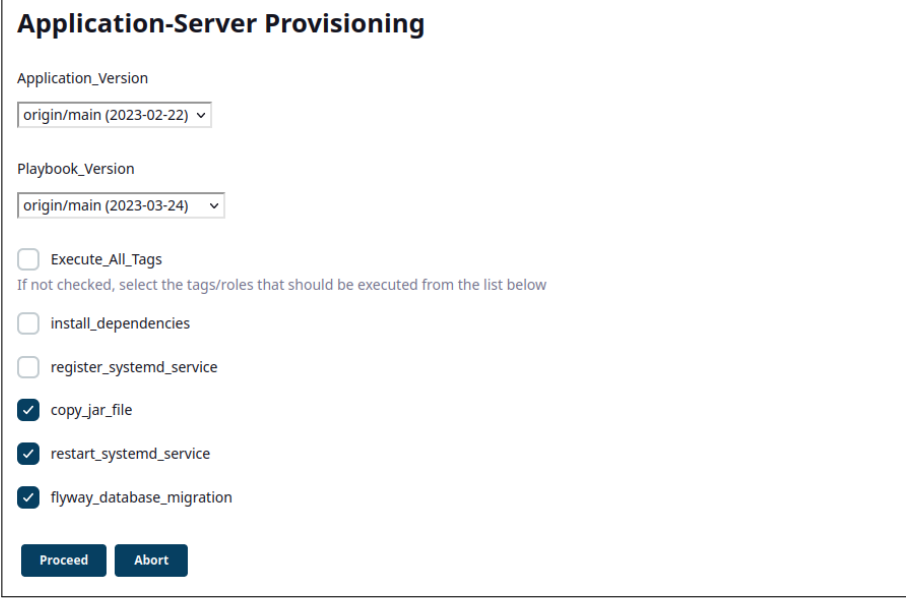
setup_rsa_service_database

Proceed Abort

Abbildung 7.1: Eingabemaske der Datenbank-Provisionierung-Pipeline

Die für die Ausführung aller Pipelines inklusive der für das Ausfüllen der Eingabemasken benötigte Zeit betrug etwa 6 Minuten. Anschließend wurde mit dem Befehl `curl 192.168.62.10/api/rsakey` ein neues RSA-Schlüsselpaar über das Demonstrations-System erstellt und damit validiert, dass die Provisionierung des selbigen erfolgreich war.

Außerdem wurde die Ausführung einzelner, ausgewählter Provisionierungsschritte getestet. Im produktiven Einsatz ist ein solcher Anwendungsfall nicht untypisch, da oft nur einzelne Software-Komponenten aktualisiert oder die Konfiguration dieser geändert werden soll. Im Test sollte daher nur die RSA Applikation aktualisiert und die Datenbankmigration ausgeführt werden. Dazu wurde die “Application-Server Provisioning“ Pipeline gestartet und in der Eingabemaske wurden nur die Tags “copy_jar_file“, “restart_systemd_service“ und “flyway_database_migration“ ausgewählt, wie in Abbildung 7.2 zu sehen ist.



Application-Server Provisioning

Application_Version
origin/main (2023-02-22) ▾

Playbook_Version
origin/main (2023-03-24) ▾

Execute_All_Tags
If not checked, select the tags/roles that should be executed from the list below

install_dependencies

register_systemd_service

copy_jar_file

restart_systemd_service

flyway_database_migration

Abbildung 7.2: Eingabemaske der Applikations-Provisionierung-Pipeline

Auch hier wurde die Pipeline erfolgreich durchlaufen. Da nur eine Teilmenge der Schritte des Provisionierungsprozesses des Applikations-Servers durchlaufen wurden, benötigte die Ausführung der Pipeline in dieser Konstellation nur etwa eine Minute.

7.2 Testauswertung

Die durchgeführten Tests haben gezeigt, dass die Provisionierungsprozesse des Demonstrations-Systems gut in das CD-System integriert werden konnten. Bei der Provisionierung des Demonstrations-Systems war kein besonderes Wissen über Ansible erforderlich. Außerdem konnte die Provisionierung über ein GUI ausgeführt werden, weshalb eine gute Benutzerfreundlichkeit gegeben ist.

Die Zeit, die für die Provisionierung benötigt wird, wurde gemessen und das gesamte System konnte in etwa zehn Minuten provisioniert werden. Bei der Provisionierung ist ebenfalls ein gewisses Maß an Skalierbarkeit gegeben, weil Ansible standardmäßig fünf parallel laufende Prozesse einsetzt, um gleiche Konfigurationsschritte auf mehreren Nodes gleichzeitig auszuführen¹.

¹https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_strategies.html

Ein weiterer wichtiger Aspekt ist der Umgang mit Fehlern bei der Ausführung der Pipelines. Obwohl bei den durchgeführten Tests keine Fehler aufgetreten sind, können alle Konsolenausgaben, die bei der Ausführung einer Pipeline entstehen, über die Weboberfläche eingesehen werden. Dies ermöglicht eine einfache Nachverfolgung im Fehlerfall.

Neben den bereits erläuterten Vorzügen durch die Verwendung von CasC, wird die Wartbarkeit des Systems durch die Verwendung eigener Git-Repositories für jede Pipeline bzw. jedes Ansible Playbook gesteigert, da so neue Pipelines einfach in das System eingehängt oder wieder entfernt werden können. Zudem ist das System durch den standardisierten Aufbau der Pipelines, wie in Abschnitt 6.1.3 und 6.2.1 beschrieben, leicht erweiterbar.

Schließlich sollte auch die Sicherheit des Systems betrachtet werden. Das CD-System benötigt zur Durchführung der Provisionierung des Demonstrations-Systems Zugriff auf den SSH-Schlüssel, der zur Authentifizierung auf den Nodes verwendet wird. Für den Zugriff auf die Git-Repositories wird ebenfalls ein SSH-Schlüssel benötigt. Die Verwaltung der genannten Informationen erfolgt in der Datei `secrets.properties`, welche von der Versionskontrolle ausgenommen ist, um ungewolltes Speichern im Git-Repository zu verhindern. Es ist außerdem möglich, mehrere Nutzer auf dem Jenkins-System anzulegen und mit Hilfe entsprechender Plugins Nutzerrechte zu vergeben. Es ist also ein gewisses Maß an Sicherheit bei dem beschriebenen CD-System gegeben. Sollte das System zur Provisionierung in einer Produktivumgebung verwendet werden, empfiehlt es sich allerdings, den Dienst in einem privaten Netzwerk zu betreiben, da sensible SSH-Schlüssel durch den Dienst verwaltet werden und es in der Vergangenheit bereits Sicherheitslücken im Jenkins-System oder bei einzelnen Plugins gegeben hat.

8 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde ein reproduzierbar provisionierbares CD-System konzipiert und umgesetzt. Anhand des Demonstrations-Systems wurde gezeigt, wie die Provisionierung eines einfachen verteilten Systems in das CD-System integriert und ausgeführt werden kann. Dabei wurden auch die anderen im Kapitel 4 definierten Anforderungen erfolgreich umgesetzt.

Das CD-System wurde auf Basis der Technologien Jenkins, Ansible und Docker aufgebaut. Beim Start des CD-Systems wird dieses automatisch provisioniert, was möglich ist, weil der gesamte Zustand des Systems als "Code" verwaltet wird. Eine hohe Flexibilität und Portabilität des Systems wird unter anderem durch die Verwendung von Docker-Containern erreicht.

Für die Integration der Provisionierungsprozesse des Demonstrations-Systems wurden drei Pipelines implementiert, durch welche die benötigte Software auf jeder Komponente des Demonstrations-Systems automatisch bereitgestellt werden kann. Die Pipelines verwenden dazu unter anderem das CM-Tool Ansible. Es wurde außerdem darauf geachtet, dass die Pipelines einfach an individuelle Anforderungen angepasst werden können, was durch die Verwendung der Script-Sprache Groovy bei der Definition der Pipelines möglich ist. Über konfigurierbare grafische Eingabemasken in der Weboberfläche können verschiedene Versionen der Ansible Playbooks und der auszuführenden Konfigurationsschritte ausgewählt werden.

Die Testergebnisse zeigen, dass das CD-System mit geringem Aufwand automatisch provisioniert und die Provisionierung des Demonstrations-Systems erfolgreich durch die Pipelines abgebildet werden konnte. Die gesamte benötigte Zeit zur Provisionierung des CD-Systems und zur Ausführung aller Pipelines betrug dabei in etwa 10 Minuten. Daher ist davon auszugehen, dass das beschriebene CD-System effektiv zur Provisionierung von Systemen eingesetzt werden und dabei zur Umsetzung des Grundsatzes der effektiven Zusammenarbeit der DevOps-Kultur beitragen kann [Halstenberg u. a., 2020, S. 5 ff.].

Insgesamt lässt sich sagen, dass die Arbeit erfolgreich das Hauptziel der Konzeptionierung und Umsetzung eines reproduzierbar provisionierbaren CD-Systems erreicht und außerdem gezeigt hat, wie die Integration der Provisionierung eines anderen Systems in das genannte CD-System realisiert werden kann. Die implementierte Lösung erfüllt die gestellten Anforderungen und bietet eine zentralisierte, flexible, wartbare und einfach zu bedienende Lösung für die Bereitstellung von kleineren verteilten Systemen.

8.1 Ausblick

Obwohl die vorliegende Arbeit erfolgreich abgeschlossen wurde, sind weiterhin Optimierungen und Erweiterungen möglich. Beispielsweise könnte die Sicherheit des Systems durch den Einsatz von Hypertext Transfer Protocol Secure (HTTPS) oder die Verwaltung der SSH-Schlüssel mittels Technologien wie Hashicorp Vault¹ weiter gesteigert werden.

Eine weitere Verbesserung wäre die dauerhafte Speicherung der Ausführungsverläufe der Pipelines, was derzeit nicht möglich ist, da die Datei-basierte Datenbank des Jenkins-Systems nur so lange besteht, wie die Ausführungsumgebung aktiv ist.

Weiterhin wäre es möglich, das System für die öffentliche Verfügbarkeit im Internet weiterzuentwickeln. Insbesondere sollte hierbei Wert auf die Verwendung von HTTPS bzw. Transport Layer Security (TLS) gelegt werden.

Für die Nutzung des CD-Systems in einem Unternehmensumfeld könnten weitere Authentifizierungsmethoden wie Single Sign-on (SSO) hinzugefügt werden, um die Verwendung des Systems durch mehrere Nutzer zu verbessern. Außerdem kann durch die Erstellung automatisierter Tests für die Provisionierung des CD-Systems und der Pipelines dazu beigetragen werden, die Qualität und Stabilität des Systems weiter zu erhöhen. Darüber hinaus könnte durch Lasttests die Leistung und Skalierbarkeit des Systems genauer überprüft werden.

¹<https://www.vaultproject.io/>

Literaturverzeichnis

- [Aiello und Sachs 2010] AIELLO, Robert ; SACHS, Leslie: *Configuration Management Best Practices: Practical Methods That Work in the Real World*. 1st. Addison-Wesley Professional, 2010. – ISBN 0321685865
- [Amaradri und Nutalapati 2016] AMARADRI, Anand S. ; NUTALAPATI, Swetha B.: *Continuous Integration, Deployment and Testing in DevOps Environment*, , Department of Software Engineering, Diplomarbeit, 2016
- [Armenise 2015] ARMENISE, Valentina: Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery. In: *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, 2015, S. 24–27
- [Bertrand u. a. 2020] BERTRAND, Benjamin ; ARMANET, Stephane ; CHRISTENSSON, Johan ; CURRI, Alessio ; HARRISSON, Anders ; MUDINGAY, Remy: ICS Infrastructure Deployment Overview at ESS. In: *17th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2020, S. WEAPP04
- [Díaz u. a. 2021] DÍAZ, Jessica ; LÓPEZ-FERNÁNDEZ, Daniel ; PÉREZ, Jorge ; GONZÁLEZ-PRIETO, Ángel: Why are many businesses instilling a DevOps culture into their organization? In: *Empirical Software Engineering* 26 (2021), Mar, Nr. 2, S. 25. – URL <https://doi.org/10.1007/s10664-020-09919-3>. – ISSN 1573-7616
- [El Khalyly u. a. 2020] EL KHALYLY, Badr ; BELANGOUR, Abdessamad ; BANANE, Mouad ; ERRAISSI, Allae: A new metamodel approach of CI/CD applied to Internet of Things Ecosystem. In: *2020 IEEE 2nd International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS)*, 2020, S. 1–6
- [Gröschel 2012] GRÖSCHEL, Michael: *Entscheidungsfaktoren zum Einsatz von Open-Source-Software an Hochschulen*. S. 79–88, 01 2012. – ISBN 978-3-86488-013-1

- [Guerriero u. a. 2019] GUERRIERO, Michele ; GARRIGA, Martin ; TAMBURRI, Damian A. ; PALOMBA, Fabio: Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, S. 580–589
- [Halstenberg u. a. 2020] HALSTENBERG, Jürgen ; PFITZINGER, Bernd ; JESTÄDT, Thomas: DevOps, Ein Überblick. (2020), 01. ISBN 978-3-658-31404-0
- [Hill 2021] HILL, Dave: *A Comparison of Configuration Management Tools in Respect of Performance and Complexity*, Technological University Dublin, Dublin, Ireland, Diplomarbeit, 2021
- [Humble und Farley 2010] HUMBLE, Jez ; FARLEY, David: *Continuous delivery: reliable software releases through build, test, and deployment automation*. Tenth printing. Addison-Wesley Professional, 2010 (The Addison-Wesley Signature Series; A Martin Fowler Signature Book)
- [Jokinen 2020] JOKINEN, Oskari: *Software development using DevOps tools and CD pipelines : a case study*. 2020. – URL [URN:NBN:fi:hulib-202003241630](https://nbn-resolving.org/urn:nbn:fi:hulib-202003241630); <http://hdl.handle.net/10138/313590>
- [Kostromin 2020] KOSTROMIN, Roman: Survey of software configuration management tools of nodes in heterogeneous distributed computing environment, 07 2020
- [Mazrae u. a. 2023] MAZRAE, Pooya R. ; MENS, Tom ; GOLZADEH, Mehdi ; DECAN, Alexandre: On the usage, co-usage and migration of CI/CD tools: a qualitative analysis. (2023). – URL <https://decan.lexpage.net/files/EMSE-2023a.pdf>
- [Morris und Safari 2020] MORRIS, K. ; SAFARI, an O'Reilly Media C.: *Infrastructure as Code, 2nd Edition*. O'Reilly Media, Incorporated, 2020. – URL <https://books.google.de/books?id=VYtAzQECAAJ>. – ISBN 9781098114664
- [Moser 2022] MOSER, Bas Meijer; Lorin Hochstein; R.: *Ansible: Up and Running, 3rd Edition*. 3. O'Reilly Media, Inc., 2022. – ISBN 9781098109158; 1098109155
- [Paloposki 2018] PALOPOSKI, Antti: *Enabling Continuous Integration through deployment automation Case Study: Property transaction system of Finnish National Land Survey*, Aalto University. School of Electrical Engineering, Master's thesis, 2018. – 42 S. – URL <http://urn.fi/URN:NBN:fi:aalto-201804031976>
- [Polkhovskiy 2016] POLKHOVSKIY, Denis: *Comparison between continuous integration tools*, Faculty of Computing and Electrical Engineering, Diplomarbeit, 2016

- [Richtarik 2016] RICHTARIK, Denis: *Cloud Integration of Continuous Integration Declarative Pipelines [online]*, Masarykova univerzita, Fakulta informatiky, Brno, Master's thesis, 2016
- [Sacks 2014] SACKS, M.: *Pro website development and operations: Streamlining devops for large-scale websites*. 07 2014
- [Seck u. a. 2022] SECK, Ayoub ; BASSENE, Constantin S. ; ZABOLO, Siré E. ; OUYA, Samuel: Building an interactive Software Defined Network from the MPSI for MPLS Service provisioning with Gitlab and Ansible. In: *2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, 2022, S. 1–6
- [Soni 2015] SONI, Mitesh: End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery. In: *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2015, S. 85–89
- [Steinhauer 2017] STEINHAUER, Patrick: *Realisierung einer Infrastructure-as-Code-Anwendung zum automatisierten Aufsetzen eines CI-Service*. 2017
- [Tanenbaum und van Steen 2020] TANENBAUM, Andrew S. ; STEEN, Maarten van: *Distributed Systems*. 3.03. Pearson Education, Inc., 2020. – ISBN 978-90-815406-2-9
- [Virtanen 2021] VIRTANEN, Joni: *Comparing Different CI/CD Pipelines*, Diplomarbeit, 2021
- [Zampetti u. a. 2021] ZAMPETTI, Fiorella ; GEREMIA, Salvatore ; BAVOTA, Gabriele ; DI PENTA, Massimiliano: CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, S. 471–482

A Anhang

A.1 casc.yml

```
1 jenkins:
2   securityRealm:
3     local:
4       allowsSignup: false
5       users:
6         - id: admin
7           password: ${JENKINS_ADMIN_PASSWORD}
8 jobs:
9   - file: /usr/share/jenkins/ref/init_pipelines.groovy
10 credentials:
11   system:
12     domainCredentials:
13     - credentials:
14       - basicSSHUserPrivateKey:
15         id: "git_access"
16         privateKeySource:
17           directEntry:
18             privateKey: "${decodeBase64:${GIT_SSH_PRIVATE_KEY}}"
19             scope: GLOBAL
20             username: "jenkins"
21       - basicSSHUserPrivateKey:
22         id: "provisioning_key"
23         privateKeySource:
24           directEntry:
25             privateKey: "${decodeBase64:${ANSIBLE_MGMT_KEY}}"
26             scope: GLOBAL
27             username: "vagrant"
28 security:
29   gitHostKeyVerificationConfiguration:
30     sshHostKeyVerificationStrategy: "noHostKeyVerificationStrategy"
31   globalJobDslSecurityConfiguration:
32     useScriptSecurity: false
```

```
33 unclassified:
34   globalLibraries:
35     libraries:
36     - name: "jenkins-cli-adapter"
37       defaultVersion: "main"
38       retriever:
39         modernSCM:
40           scm:
41             git:
42               credentialsId: "git_access"
43               remote: "git@github.com:jannst/jenkins-cli-adapter.git"
```

A.2 Applikations Provisionierung Jenkinsfile

```
1 node {
2   @Library('jenkins-cli-adapter')_
3   stage("execute Ansible") {
4     checkout scm
5
6     def applicationGitTags = [];
7     dir("webservice") {
8       checkout([
9         $class: 'GitSCM',
10        branches: [[name: "main"]],
11        userRemoteConfigs: [[credentialsId: "git_access", url: "
12          git@github.com:jannst/rsa_key_service.git"]]
13        ])
14        applicationGitTags = gitGetTagMap()
15      }
16
17      def playbookFile = "playbook.yaml"
18      def playbookTags = ansibleGetPlaybookTags(playbook: playbookFile)
19      def ansibleInventoryPath = ansibleFetchInventory(inventoryRepoUrl:
20        env.PROVISIONING_INDEX_REPO, credentialsId: 'git_access')
21      def playbookGitTags = gitGetTagMap()
22
23      def tagSelectOptions = playbookTags.collect{[$class: '
24        BooleanParameterDefinition', defaultValue: false, name: it]}
25      def userInput = input(id: 'userInput', message: 'Application-Server
26        Provisioning',
27        parameters: [
```

```
25         [$class: 'ChoiceParameterDefinition', choices: new ArrayList
           <>(applicationGitTags.keySet()), name: '
           Application_Version'],
26         [$class: 'ChoiceParameterDefinition', choices: new ArrayList
           <>(playbookGitTags.keySet()), name: 'Playbook_Version'],
27         [$class: 'BooleanParameterDefinition', defaultValue: false,
           description: "If not checked, select the tags/roles that
           should be executed from the list below", name: '
           Execute_All_Tags']
28     ] + tagSelectOptions
29 )
30
31 if(!userInput['Execute_All_Tags'] && playbookTags.findAll{userInput[
   it]}.size() == 0) {
32     throw new IllegalArgumentException("No tags selected")
33 }
34
35 def ansibleTagArgument = userInput['Execute_All_Tags'] ? "all" :
   playbookTags.findAll{userInput[it]}.join(',')
36 def playbookVersion = playbookGitTags[userInput['Playbook_Version']]
37 def applicationVersion = applicationGitTags[userInput['
   Application_Version']]
38
39 checkout scm: [$class: 'GitSCM', branches: [[name: playbookVersion
   ]]], poll: false
40
41 def webserverJarFile = ""
42 dir("webservice") {
43     checkout([
44         $class: 'GitSCM',
45         branches: [[name: applicationVersion]],
46         userRemoteConfigs: [[credentialsId: "git_access", url: "
           git@github.com:jannst/rsa_key_service.git"]]
47     ])
48     sh(script: "mvn clean compile assembly:single")
49     def jarFiles = findFiles(glob: 'target/*.jar')
50     if(jarFiles.size() != 1) {
51         throw new RuntimeException("There are more than one or no jar
           file in target directory: " + jarFiles)
52     }
53     webserverJarFile = "webservice/" + jarFiles[0]
54 }
55
56
```

```
57     runPlaybook(  
58         credentialsId: 'provisioning_key',  
59         inventory: ansibleInventoryPath,  
60         playbook: playbookFile,  
61         tags: ansibleTagArgument,  
62         extraVars: [  
63             db_password:"secret",  
64             local_application_jar: webserverJarFile  
65         ]  
66     )  
67 }  
68 }
```

Erklärung zur selbstständigen Bearbeitung

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

Ort

Datum

Unterschrift im Original