Technische Hochschule
Brandenburg
University of
Applied Sciences
Fachbereich
Informatik und Medien

BACHELORARBEIT

Optimierung von Deploymentprozessen mithilfe von Ansible

Vorgelegt von: Florian Stamer
am: 2025-09-22

zum
Erlangen des akademischen Grades

Bachelor of Science
(B. Sc.)

Erstbetreuer: Prof. Thomas Preuss
Zweitbetreuer: M. Sc. Christian Lange

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit zum Thema

Optimierung von Deploymentprozessen mithilfe von Ansible

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Brandenburg a. d. Havel, den 2025-09-22

Unterschrift *Florian Stamer*

Zusammenfassung

Die vorliegende Bachelorarbeit untersucht die Optimierung von Deploymentprozessen durch den Einsatz von Ansible in Kombination mit Containertechnologien.

Ausgangspunkt ist die Herausforderung, Software auf einer Vielzahl von Systemen effizient, konsistent und fehlerfrei bereitzustellen.

Ziel war die Entwicklung und Evaluierung eines automatisierten Deploymentprozesses, der die Nachteile manueller Verfahren überwindet.

Im theoretischen Teil werden Grundlagen moderner Deployment-Verfahren, Konfigurationsmanagement sowie die Containerisierung mit Podman erläutert. Darauf aufbauend wird ein Konzept vorgestellt, das Containerisierung, Container Registry und Ansible-Playbooks integriert.

Die Implementierung umfasst eine modulare Systemarchitektur mit MariaDB- und Mirthconnect Containern, deren Bereitstellung über Podman Compose und Ansible automatisiert erfolgt. Ergänzend wurde eine Weboberfläche entwickelt, um Playbooks komfortabel auszuführen und Statusmeldungen transparent darzustellen.

Die Evaluation zeigt, dass der Ansatz eine signifikante Zeitersparnis erzielt: Ein Deployment mit Ansible dauert durchschnittlich 30 Sekunden und erfordert minimalen manuellen Eingriff. Darüber hinaus konnten Fehlerquellen reduziert und die Wartbarkeit durch modulare Playbooks und reproduzierbare Containerstrukturen verbessert werden. In der Praxis bewährte sich das Konzept bei der Firma Vireq Software Solutions, wo es zu einer nachhaltigen Effizienzsteigerung in produktiven Prozessen beitrug.

Die Arbeit verdeutlicht, dass die Kombination von Automatisierung und Containerisierung eine robuste und zukunftsfähige Grundlage für effiziente Softwarebereitstellungen darstellt und Potenzial für weitere Entwicklungen wie CI/CD-Integration oder erweitertes Monitoring bietet.

Inhaltsverzeichnis

1 Einleitung

1.1 Wissenschaftliche Fragestellung

Wie kann der Deploymentprozess von Software mithilfe von Ansible und Container-Technologien automatisiert und effizienter gestaltet werden, sodass wiederholbare, wartbare und fehlerfreie Rollouts über viele Systeme hinweg möglich sind?

1.2 Motivation

Die vorliegende Arbeit resultiert aus einem Praktikumsprojekt bei der Firma Vireq Software Solutions, bei dem die Anforderung bestand, eigens entwickelte Software auf etwa 120 Systemen bereitzustellen und zu warten. Die manuelle Bereitstellung von Software in komplexen IT-Umgebungen ist jedoch mit erheblichem zeitlichem Aufwand verbunden, birgt eine erhöhte Fehleranfälligkeit und erschwert die konsistente Reproduzierbarkeit. Diese Problematik manifestiert sich insbesondere in Szenarien, in denen eine Vielzahl von Systemen und deren Abhängigkeiten zu berücksichtigen sind. Traditionelle manuelle Vorgehensweisen sind mitunter ineffizient und resultieren in einem erhöhten Wartungsaufwand.

Vor diesem Hintergrund entstand die Motivation, Verfahren zur Automatisierung des Deploymentprozesses zu untersuchen. Nach einer internen Analyse wurde der Ansatz eines automatisierten Bereitstellungsprozesses verfolgt, wobei sich Ansible in Kombination mit Containertechnologien als vielversprechende Lösung erwies. Die Automatisierung verspricht eine signifikante Steigerung der Effizienz, die Etablierung standardisierter Abläufe, die Reduktion von Fehlerquellen sowie eine verbesserte Wartbarkeit und Konsistenz über verschiedene Systeme hinweg.

So dient diese Arbeit der Ermittlung der Effizienz dieser Methode.

1.3 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung und Evaluierung eines automatisierten Deployment-prozesses, der die Schwächen manueller Verfahren überwindet und damit einen Beitrag zur Effizienzsteigerung und Qualitätsverbesserung in komplexen IT-Umgebungen leistet.

Der Fokus der Untersuchung liegt auf dem Vergleich zwischen einem herkömmlichen manuellen Vorgehen und einem automatisierten Ansatz, der auf Ansible und Containertechnologien basiert.

Der Einsatz von Ansible Playbooks zielt darauf ab, manuelle Eingriffe auf ein Minimum zu reduzieren sowie die Fehleranfälligkeit zu reduzieren und gleichzeitig eine erhöhte Konsistenz und Nachvollziehbarkeit der Abläufe zu gewährleisten. Dadurch wird eine deutliche Verkürzung der Deployment-Zeiten sowie eine Entlastung der personellen Ressourcen erwartet.

Des weiteren ist die Wartbarkeit von zentraler Relevanz. Modulare Playbooks und Containerlösungen ermöglichen eine flexible Anpassung, Erweiterung und Diagnose, wodurch die langfristige Betreuung der Systeme erleichtert wird. So liegt einer der Hauptmerkmale in der Sicherstellung einer hohen Reproduzierbarkeit, sodass Deployments auf allen Zielsystemen unter identischen Bedingungen konsistent ausgeführt werden können. Schließlich zielt der Einsatz von Containern wie Podman darauf ab, das Update-Management zu vereinfachen. Dies umfasst sowohl die saubere Versionierung von Software als auch die effiziente Bereitstellung von Aktualisierungen.

Die vorliegende Arbeit verfolgt das übergeordnete Ziel, zu demonstrieren, inwiefern moderne Automatisierungstechnologien einen nachhaltigen Mehrwert gegenüber traditionellen Deploymentprozessen bieten und wie sie die Stabilität, Effizienz und Qualität von Bereitstellungsprozessen verbessern können.

1.4 Aufbau der Arbeit

Kapitel 2 stellt die theoretischen Grundlagen von Deploymentprozessen, Konfigurationsmanagement mit Ansible und Containertechnologien vor. Kapitel 3 entwickelt auf dieser Basis ein Konzept zur Automatisierung mit Podman und Ansible. Kapitel 4 beschreibt die praktische Implementierung sowie die eingesetzte Systemarchitektur. Kapitel 5 diskutiert die Ergebnisse, bewertet die Effizienz des entwickelten Ansatzes und gibt einen Ausblick auf mögliche Weiterentwicklungen.

2 Theoretische Grundlagen

2.1 Deployment Prozesse

In modernen DevOps-Umgebungen bezeichnet Deployment den Prozess, durch den neue oder aktualisierte Anwendungen in eine Produktionsumgebung übertragen und damit für Endbenutzer verfügbar gemacht werden [**atlassianCICD**; **techtargetDeployment**] . Dieser Schritt ist essenziell, da er sicherstellt, dass neue Softwareversionen zuverlässig und effizient auf den Zielsystemen bereitgestellt werden. Ein sorgfältig gestalteter Deployment-Prozess kann die Markteinführungszeit von Funktionen und Updates signifikant reduzieren. Teams, die Deployments optimieren oder automatisieren, können schneller auf Kundenanforderungen reagieren und Updates mit höherer Frequenz ausrollen. Dies führt zu einer höheren Benutzerzufriedenheit und schafft Wettbewerbsvorteile. Deployment lässt sich demnach als der zentrale Mechanismus definieren, der die Übertragung von Codeänderungen von Entwicklern aus der Bau- und Testumgebung in den produktiven Betrieb ermöglicht [**humbleJez2010**].

2.1.1 Technische Schritte: Build, Testing, Release und Rollout

Im technischen Ablauf eines Deployment-Prozesses wird zunächst der Build ausgeführt: Im Rahmen des Prozesses der Softwareentwicklung wird der Quellcode kompiliert oder in Container-Images bzw. Pakete gepackt, um ein lauffähiges Artefakt zu erzeugen. Im Anschluss werden automatisierte Tests (Unit-, Integrations-Tests etc.) durchgeführt, um die korrekte Funktion des Builds und die Erfüllung der Qualitätskriterien zu prüfen. Nach dem erfolgreichen Abschluss der Tests erfolgt die Vorbereitung der Veröffentlichung. Das Artefakt wird versioniert und in ein Artefakt-Repository übertragen, [**techtargetDeployment**]. wobei verschiedene Optionen zur Auswahl stehen, wie beispielsweise ein Git-Repository, eine Container-Registry oder eine Paket-Registry.

Im finalen Schritt erfolgt die Implementierung der neuen Version, wobei die Software auf den Zielsystemen installiert oder gestartet wird. Es besteht die Möglichkeit, diesen Prozess in einem einzigen Schritt durchzuführen oder ihn gestaffelt auszuführen. Ein Beispiel für eine solche Methode ist das Rolling Deployment [**awsDeploymentStrategies**] , bei dem die neue Version schrittweise auf mehrere Server bzw. Instanzen verteilt wird, um Ausfallzeiten zu minimieren. Es ist von entscheidender Bedeutung, dass die Build- und die Deploy-Phase entkoppelt sind. Bereits erzeugte und getestete Artefakte können in

verschiedenen Umgebungen (Staging, Produktion) wiederverwendet werden. Eine typische Pipeline durchläuft bei jedem Merge in den Hauptzweig die Stufen Build, Test und Release in automatisierten Schritten [**atlassianCICD**].

2.1.2 Konfigurationsmanagement via Ansible

Ein leistungsfähiges Deployment erfordert konsequentes Konfigurationsmanagement. Die Verwaltung der Infrastruktur erfolgt in vielen Fällen nach dem Prinzip Infrastructure as Code (IaC). Im Rahmen dessen erfolgt die Definition von Servern, Netzwerken und Abhängigkeiten in Code, anstatt eine manuelle Konfiguration durchzuführen. Zu den prominenteren Tools zählen Terraform und Ansible [**turnbull2018infrastructure**].

In diesem Kontext ist Ansible ein besonders weit verbreitetes Werkzeug. Es handelt sich um ein agentenloses Automatisierungs- und Konfigurationsmanagement, das auf der Programmiersprache Python basiert und als Open-Source-Software frei verfügbar ist [**ansibleDocs**]. Die Kommuniktion mit Zielsystemem erfolgt mittels Secure Shell (SSH) oder Windows Remote Management (WinRM). Die Grundlage bilden sogenannte Playbooks, in denen Abläufe zur Installation von Software, zur Anpassung von Systemeinstellungen oder zur Orchestrierung komplexer Prozesse beschrieben werden. Durch seine deklarative Struktur ermöglicht Ansible die Umsetzung konsistenter und wiederholbarer Deployments über eine Vielzahl von Systemen hinweg. In der praktischen Anwendung wird es nicht nur für einzelne Konfigurationsaufgaben genutzt, sondern zunehmend auch zur Standardisierung und Automatisierung ganzer Deploymentprozesse, wodurch eine effiziente Verwaltung heterogener Systemlandschaften ermöglicht wird.

In der praktischen Anwendung findet Ansible nicht nur bei einzelnen Konfigurationsaufgaben Anwendung, sondern wird zunehmend auch zur Standardisierung und Automatisierung ganzer Deploymentprozesse genutzt. Der vorliegende Ansatz ermöglicht eine effiziente Verwaltung und konsistente Betreuung heterogener Systemlandschaften. Damit leistet Ansible einen wesentlichen Beitrag zur Umsetzung von IaC-Prinzipien und bildet die Grundlage für die in dieser Arbeit verfolgte Optimierung von Deploymentprozessen.

Neben der kostenlosen Community-Version existiert mit der Ansible Automation Platform von Red Hat eine erweiterte Variante, die zusätzliche Funktionen für den Unternehmenseinsatz bereitstellt. Unabhängig von der gewählten Version leistet Ansible einen wesentlichen Beitrag zur Umsetzung von IaC-Prinzipien und bildet die Grundlage für die in dieser Arbeit verfolgte Optimierung von Deploymentprozessen.

2.1.3 Monitoring und Logging nach dem Deployment

Nach Abschluss des Deployment-Prozesses ist eine Überwachung des Live-Betriebs erforderlich. Monitoring und Logging sind entscheidend, um den Zustand der neuen Version zu kontrollieren und Probleme früh zu erkennen [**atlassianCICD**]. Durch kontinuierliches

Monitoring der System- und Anwendungsmetriken (etwa über Prometheus/Grafana, CloudWatch oder ähnliche Tools) ist es möglich, CPU-, Speicher- und Antwortzeit-Verläufe im Auge zu behalten und Ausfälle unmittelbar zu detektieren.

Der Einsatz zentralisierter Log-Systeme (ELK-Stack, Fluentd, Loki) ermöglicht die Sammlung aller Anwendungs- und Systemlogs. Dies erlaubt die Analyse von Fehlerursachen und die Identifizierung von Trends. Hieraus ergibt sich die Möglichkeit, zu ermitteln, ob beispielsweise neue Fehler aufgetreten sind, die in direktem Zusammenhang mit dem jüngst implementierten Release stehen. Alerting-Mechanismen (E-Mail, Slack, PagerDuty) werden aktiviert, sobald definierte Schwellenwerte überschritten werden.

Es lässt sich folgern, dass ein sorgfältig konzipiertes Monitoring den Betrieb gewährleistet, indem es zeitnahe Rückkopplungsinformationen bereitstellt und es der Mannschaft ermöglicht, bei Bedarf unmittelbar in das Deployment einzugreifen. Die kontinuierliche Überwachung ermöglicht die Validierung der Auswirkungen einer Implementierung. Bei Bedarf können daraufhin umgehend Maßnahmen wie ein Rollback oder eine erneute Durchführung ergriffen werden.

2.2 Podman

Podman ist eine an Docker angelehnte Container-Engine, mit der sich Container erstellen und verwalten lassen. Podman stellt dabei eine CLI sowie eine Open Container Initiative konforme Engine und Runtime bereit, um Container zu verwalten und mit dem Betriebssystem zu kommunizieren und Ressourcen zuzuteilen. Ein großer Unterschied zu Docker ist, dass Podman standardmäßig nicht als Superuser, sondern als der Benutzer, der den Container startet, ausgeführt wird [**podmanDocs**]. So ermöglicht Podman eine sicherheitskonforme Kontrolle des gesamten Container- Ökosystems.

Da Podman stark an Docker angelehnt ist, sind die meisten Befehle der CLI ähnlich oder identisch. Beispielsweise entspricht `docker run` dem Befehl `podman run`. Sie können sogar mit einem Alias belegt werden, sodass `docker` zu `podman` wird. Der größte Unterschied zu Docker und anderen Container Engines besteht darin, dass Podman daemonless ist. Das bedeutet, dass Podman nicht auf einen Daemon angewiesen ist, der im Hintergrund des Systems verschiedene zugehörige Prozesse verwaltet. Dadurch ist es möglich, alle Container rootless auszuführen und diese ohne Superuser zu verwalten.

Für Nutzer, die GUIs bevorzugen, stellt Red Hat Podman Desktop bereit, welches alle Funktionen in einem Desktop Client mit grafischer Oberfläche vereint. [**podmanDocs**]

2.3 Podman Images

Um die Container Images zu speichern benötigt es eine Container Registry, dabei gibt es zwei Möglichkeiten, eine Öffentliche Container Registries (z. B. Docker Hub, Amazon Elastic

Container Registry (ECR), Azure Container Registry (ACR), Google Container Registry (GCR)) welche sich besonders für die Versionierung von Open Source Projekten eigenen. Oder Private Container Registries welche für sensible oder projektspezifische Images dienen.

Nachdem die theoretischen Grundlagen zu Deployment, Konfigurationsmanagement mit Ansible und Containerisierung erläutert wurden, werden im folgenden Kapitel die praktischen Grundlagen sowie ein darauf aufbauendes Konzept für einen automatisierten Deploymentprozess vorgestellt.

3 Konzept

Das in dieser Arbeit entwickelte Konzept dient als Brücke zwischen den theoretischen Grundlagen und der praktischen Implementierung des automatisierten Deploymentprozesses. Ziel ist es, auf Basis der zuvor beschriebenen Technologien ein strukturiertes Vorgehen zu entwerfen, das die identifizierten Schwächen manueller Verfahren überwindet und klare Anforderungen erfüllt. Dazu müssen die praktischen Möglichkeiten von Ansible sowie Podman geklärt und erprobt werden. So soll dadurch eine erhöhte Reproduzierbarkeit, Effizienz sowie Wartbarkeit erreicht werden.

Bei der Entwicklung des Konzepts wurden zudem einige Alternativen betrachtet. Neben Ansible existieren weitere Konfigurationsmanagement- und Automatisierungstools wie Puppet, Chef oder SaltStack, die ähnliche Ziele verfolgen. Ebenso stehen mit Docker oder Kubernetes verschiedene Containerlösungen zur Verfügung. Nach einer Analyse dieser Ansätze fiel die Entscheidung zugunsten von Ansible in Kombination mit Podman, da diese Werkzeuge einerseits den Anforderungen des Projekts entsprachen, andererseits eine einfache Integration, geringe Einstiegshürden und eine hohe Flexibilität bieten, außerdem besitzt Ansible die größte Community , und Podman bietet viele Sicherheitsfeatures im Vergleich zu Docker.

3.1 Vereinfachung durch Containerisierung

Die Containerisierung ermöglicht es, Software und ihre Abhängigkeiten in isolierten Umgebungen bereitzustellen. Dies führt zu einer höheren Portabilität zwischen verschiedenen Systemen und reduziert Konfigurationsprobleme, die durch unterschiedliche Umgebungen entstehen können. Besonders durch Werkzeuge wie Podman lassen sich Anwendungen konsistent auf Entwicklungs-, Test- und Produktionssystemen betreiben. Container bieten zudem eine leichtgewichtige Alternative zu virtuellen Maschinen, da sie auf den Kernel des Hostsystems zugreifen und somit ressourcenschonender arbeiten. [**dockerOverview**]

Ein weiterer Vorteil ist die vereinfachte Reproduzierbarkeit. Da ein Container stets auf einem definierten Image basiert, kann derselbe Zustand der Anwendung auf unterschiedlichen Systemen identisch nachgebildet werden. Dies verbessert sowohl die Nachvollziehbarkeit als auch die Fehlerdiagnose im Fehlerfall.

3.1.1 Container Images

Container Images stellen die Grundlage für Containerisierung dar, diese enthalten die Software, deren Abhängigkeiten, Konfigurationen und alle notwendigen Bibliotheken, um eine Anwendung lauffähig zu machen. Images sind unveränderlich und können über Versionen verwaltet werden, sodass jederzeit eine konsistente Umgebung reproduziert werden kann. Sie ermöglichen es, Anwendungen schnell zu starten, zu skalieren und zwischen verschiedenen Systemen zu portieren. Typischerweise werden Images aus einem Dockerfile oder einer ähnlichen Definition erzeugt, in der die Installation der Software und die Konfiguration der Laufzeitumgebung automatisiert beschrieben sind.

3.1.2 Podman Compose

Podman Compose ist ein Werkzeug zur Orchestrierung mehrerer Container, vergleichbar mit Docker Compose, jedoch ohne dass ein zentraler Daemon erforderlich ist. Mit Podman Compose lassen sich komplexe Multi-Container-Anwendungen lokal oder in produktiven Umgebungen starten, stoppen und verwalten. Die Konfiguration erfolgt über YAML-Dateien, in denen Abhängigkeiten, Netzwerke, Volumes und Umgebungsvariablen definiert werden. Dies ermöglicht eine einfache Nachbildung von Test- , Entwicklungs- und Produktionsumgebungen und erleichtert die Integration in automatisierte Deployment-Pipelines. Dabei kann kann entweder Docker Compose [**DockerCompose**] oder Podman Compose [**PodmanCompose**] verwendet werden. Podman Compose ist dabei eine Implementation des Compose Standards in Verbindung mit dem Podman Backend und ist in Python Implementiert.

3.1.3 Container Registry

Die Bereitstellung und wiederholte Nutzung von Container Images in einer produktiven Umgebung erfordert die Implementierung einer Container Registry. Der Speicherort ist zentralisiert und dient der Versionierung, Verwaltung und Bereitstellung der erstellten Abbilder (Images) von Containern für den Rollout. Container Registries ermöglichen eine standardisierte und reproduzierbare Bereitstellung von Software, da die Images unabhängig vom Zielsystem in identischer Form abgerufen werden können.

3.2 Automatisches Deployment durch Ansible

Ansible ist ein Agentenloses Konfigurationsmanagement- und Automatisierungswerkzeug, das auf YAML-basierten Playbooks basiert. Es ermöglicht die automatisierte Provisionierung, Konfiguration und das Deployment von Systemen und Anwendungen. Durch die deklarative Schreibweise lassen sich wiederholbare und nachvollziehbare Prozesse

definieren, die Fehleranfälligkeit manuell ausgeführter Installations- und Konfigurationsschritte erheblich reduzieren.[**ansibleDocs**] Im Kontext des Software Deployment erlaubt Ansible das zentrale Management mehrerer Systeme über SSH. Dadurch können komplexe Deployments, beispielsweise das Einspielen von Container Images, das Anpassen von Konfigurationsdateien oder das Neustarten von Diensten vollständig automatisiert werden. Zudem lässt sich Ansible leicht mit Container-Plattformen wie Docker oder Kubernetes integrieren, was die Automatisierung über den gesamten Lebenszyklus einer Anwendung hinweg unterstützt. [**automationWithAnsible**]

3.2.1 Ansible Ad Hoc Kommandos

Die Einarbeitung in Ansible erfolgt zunächst über die Analyse grundlegender Terminal-Befehle, so bietet /usr/bin/ansible Möglichkeiten für die Verwaltung von mehreren Nodes und Systemen, so können grundlegende Ansible Komponenten verwendet werden und ausprobiert werden, jedoch fehlt eine Wiederverwendbarkeit bei diesem Ansatz. Optimal sind Ad Hoc Commands für Aktionen die selten ausgeführt werden, beispielsweise um die Systeme für eine Urlaubsphase herunterzufahren. [**ansibleAdHoc**] [**AnsibleAdHocTom**]

```
root@ansible-master:~# ansible
usage: ansible [-h] [--version] [-v] [-b] [--become-method BECOME_METHOD] [--become-user BECOME_USER] [-K |
               --become-password-file BECOME_PASSWORD_FILE] [-i INVENTORY] [--list-hosts] [-l SUBSET]
               [-P POLL_INTERVAL] [-B SECONDS] [-o] [-t TREE] [--private-key PRIVATE_KEY_FILE] [-u REMOTE_USER]
               [-c CONNECTION] [-T TIMEOUT] [--ssh-common-args SSH_COMMON_ARGS] [--sftp-extra-args SFTP_EXTRA_ARGS]
               [--scp-extra-args SCP_EXTRA_ARGS] [--ssh-extra-args SSH_EXTRA_ARGS] [-k |
               --connection-password-file CONNECTION_PASSWORD_FILE] [-C] [-D] [-e EXTRA_VARS] [--vault-id VAULT_IDS]
               [-J | --vault-password-file VAULT_PASSWORD_FILES] [-f FORKS] [-M MODULE_PATH] [--playbook-dir BASEDIR]
               [--task-timeout TASK_TIMEOUT] [-a MODULE_ARGS] [-m MODULE_NAME]
```

Abbildung 1: Struktur von Ansible ad-hoc

3.2.2 Ansible Inventory

Um mit Ansible verschiedene Systeme via SSH zu erreichen und zu kontrollieren muss auf dem zu kontrollierenden System eine Authentifizierung via SSH Schlüssel oder Passwort möglich sein. Ist dies gegeben können auf dem Kontrollsystem diese Systeme in einer Hosts oder Inventory Datei abgebildet werden. Dabei kann diese entweder eine INI oder eine YAML, sowie statisch oder dynamisch sein, Ansible unterstützt beides. [**AnsibleInventoriesTom**] Ein Beispiel würde so aussehen:

Listing 3.1: Inventory Beispiel

```
1 [test]
2 10.20.0.152
3 10.20.0.25
4
5 [prod]
```

```
6  10.20.0.125
```

3.2.3 Ansible Groups und Patterns

Ansible Groups sind Kategorien im Inventar, die mehrere Hosts gemeinsam organisieren. Diese dienen dazu, Playbooks und Ad-hoc-Befehle gezielt auf mehrere Hosts anzuwenden, dabei existieren ohne eigene Definition zwei Standardgruppen: **all** welche alle Hosts im Inventar enthält und **ungrouped** welche Hosts, die in keiner Gruppe definiert sind beinhaltet.

Es besteht jedoch auch die Möglichkeit, eine beliebige Anzahl eigens erstellter Gruppen anzulegen sowie diese mittels „patterns“ in unterschiedliche Gruppen zusammenzufassen.

Pattern	Bedeutung
`test`	alle Hosts in der Gruppe `test`
`test:prod`	alle Hosts beider Gruppen
`test:&prod`	Schnittmenge: Hosts in beiden Gruppen

Dabei enthält das in Unterabschnitt 3.2.2 genannte Inventar 2 Gruppen, test und prod. Das obere Beispiel ist im INI Format, das äquivalente YAML würde wie folgt aussehen:

Listing 3.2: Inventory Beispiel YAML

```
1  ---
2  test:
3      hosts:
4          10.20.0.152:
5          10.20.0.25:
6  prod:
7      hosts:
8          10.20.0.125:
```

Über den ansible-inventory command ist es möglich mit einem Inventar zu interagieren, das folgende Beispiel ist ein Ausschnitt aus dem finalem Inventory dieser Arbeit.

Listing 3.3: Inventory Groups

```
root@ansible-master:~/ansible# ansible-inventory -i hosts --graph
@all:
  |--@ungrouped:
  |--@prod:
  |  |--server1
  |  |--server2
```

Mit der - - list Option gibt das Inventory eine JSON Repräsentation mit mehr Informationen über Hosts und Gruppen zurück, was sehr hilfreich zum debuggen ist. So kann man im Codebeispiel erkennen, dass die beiden enthaltenen Server Teil der Gruppe @all sowie @prod sind. So können in einem späterem Playbook verschiedene Aktionen für verschiedene Gruppen ausgeführt werden.

```
root@ansible-master:~/ansible# ansible-inventory -i hosts --list
{
    "_meta": {
        "hostvars": {
            "server1": {
                "ansible_host": "10.20.0.152",
                "ansible_ssh_private_key_file": "~/.ssh/ansible",
                "ansible_user": "root"
            },
            "server2": {
                "ansible_host": "10.20.0.25",
                "ansible_ssh_private_key_file": "~/.ssh/ansible",
                "ansible_user": "root"
            }
        }
    },
    "all": {
        "children": [
            "ungrouped",
            "prod"
        ]
    },
    "prod": {
        "hosts": [
            "server1",
            "server2"
        ]
    }
}
```

Abbildung 2: Ansible Inventory JSON List

3.2.4 Ansible Playbooks

Ansible-Playbooks beschreiben eine Reihe von von Automatisierungsaufgaben zur Konfiguration, Bereitstellung und Verwaltung von Remote Systemen auf wiederholbare und wiederverwendbare Weise. Playbooks konstituieren den Kern des Ansible Automatisierungsframeworks und bieten die Option, IT Prozesse über eine Vielzahl von Systemen hinweg mit minimalem manuellem Aufwand zu orchestrieren.[**AnsiblePlaybooks**] Dabei spezifizieren Playbooks immer den gewollten Zustand eines Systems und die nötigen Schritte um diesen zu erreichen. Einmal geschrieben bieten Playbooks eine extrem hohe Wiederverwendbarkeit sowie eine einfache Verwaltung mithilfe von Versionskontrolle wie Git.

3.2.4.1 Struktur / Syntax

Wie in Abschnitt 3.2 dargelegt, werden Playbooks in YAML verfasst und bestehen aus mehreren Komponenten, die in Kombination den Zielzustand sowie das angestrebte Verhalten beschreiben. [**AnsiblePlaybooksTom**] Ein Play richtet sich jeweils an eine definierte Gruppe von Hosts und fasst die darauf auszuführenden Aufgaben zusammen. Diese Aufgaben werden als Tasks bezeichnet und stellen konkrete Aktionen dar, etwa die Installation von Software oder das Neustarten von Diensten. Zur Umsetzung greifen Tasks auf Module zurück, die als kleine Programme spezielle Funktionen wie Paketinstallation oder Benutzerverwaltung übernehmen. Darüber hinaus existieren Handler, besondere Tasks, die nur dann ausgeführt werden, wenn sie explizit von anderen Aufgaben ausgelöst werden. Für mehr Flexibilität können Variablen eingesetzt werden, mit deren Hilfe dynamische Werte gespeichert und im gesamten Playbook genutzt werden. Templates ermöglichen es, Dateien mit Platzhaltern zu definieren, die anschließend durch Variablen dynamisch befüllt werden. Schließlich legt die separat definierte Inventory-Datei fest, auf welchen Hosts ein Playbook ausgeführt wird.

3.2.4.2 Ausführung

Listing 3.4: Beispiel Playbook Apache

```
1 - name: Installiere Apache auf Webservern
2   hosts: webservers
3   become: yes
4   tasks:
5     - name: Installiere Apache
6       apt:
7         name: apache24
8         state: present
```

Dieses Playbook installiert einen Apache-Webserver auf allen Hosts in der Gruppe „webservers“. Die Option „become: yes“ sorgt dafür, dass die Aufgaben mit erhöhten Rechten (z.B. root) ausgeführt werden. [**AnsiblePlaybooksTom**] Führt man das Playbook mit dem command

Listing 3.5: Ansible Playbook Ausführung

```
ansible-playbook -i inventory playbook.yaml
```

aus so wird der Output wie folgt aussehen:

Listing 3.6: Ansible Playbook Output

```
1 PLAY [Installiere Apache auf Webservern]
2 TASK [Gathering Facts]
3 10.20.0.152: ok=1 changed=1 unreachable=0 failed=0 skipped=0 rescued=0
    ignored=0
```

So kann festgestellt werden, dass Playbooks eine wesentliche Kernkomponente von Ansible darstellen. Es handelt sich um die Protokolle für verschiedene Operationen, die auf Hosts und Gruppen ausgeführt werden sollen.

Die Ausführung von Aufgaben erfolgt in Playbooks von oben nach unten mittels des Befehls „ansible-playbook“. Die eigentlichen Operationen werden von Ansible Modulen ausgeführt, die durch die Aufgaben geladen werden.

Das in diesem Kapitel entwickelte Konzept bildet die Grundlage für die praktische Umsetzung. Im nächsten Kapitel wird die daraus resultierende Implementierung des Deployments mit Podman und Ansible beschrieben.

4 Implementation

Die praktische Umsetzung erfolgt in drei wesentlichen Schritten. Zunächst wird die Software mithilfe von Podman Containern, wie in **Abschnitt 3.1** erläutert und verpackt. Im Anschluss werden Playbooks und die Abläufe in diesem erstellt. Schließlich erfolgt der Deploymentprozess auf verschiedene Testsysteme, anschließend das Verpacken und Verbinden von Software mithilfe von Containern sowie das Rollout auf die Zielsysteme via Ansible. Darüber hinaus werden zukünftige Prozesse wie Updates und Wartungen ebenfalls über Ansible realisiert.

4.1 Systemarchitektur

Die vorliegende Systemarchitektur des Deployments weist mehrere sicherheitsrelevante und infrastrukturelle Besonderheiten auf. Ein wesentliches Merkmal besteht in der vollständigen Abschottung der Zielumgebung von der öffentlichen Internet-Infrastruktur. Die Realisierung dieser Isolation erfolgte durch den Einsatz eines Site-to-Site Virtual Private Networks (S2S-VPN). Dieses stellt eine gesicherte Verbindung zwischen dem internen Firmennetzwerk und der Infrastruktur des Kunden her.

Im Firmennetzwerk befinden sich die Entwicklungs- und Verwaltungsserver, die kontinuierlich mit dem firmeninternen VPN verbunden sind. Über diese Verbindung können die Systeme auf das Zielsystem zugreifen, ohne eine direkte öffentliche Netzwerkanbindung nutzen zu müssen. Dadurch wird eine klare Trennung zwischen Entwicklungsumgebung und Kundensystemen gewährleistet. Der Verwaltungsserver beinhaltet zudem die vollständige Ansible Installation, einschließlich Playbook und Inventory.

Der Zugriff auf die Zielsysteme erfolgt über stationäre SSH-Schlüssel. Dadurch wird eine einheitliche und automatisierte Authentifizierung ermöglicht. Im Rahmen des Deploymentprozesses impliziert diese Architektur, dass der initiale Schritt des Hochladens von Container-Images in die private Container Registry innerhalb des privaten Netzwerks erfolgt. Das Rollout auf die Zielsysteme erfolgt anschließend über eine gesicherte VPN Verbindung. Die Kombination aus Containerisierung, Registry und gesicherter Netzwerkarchitektur ermöglicht somit eine kontrollierte, reproduzierbare und gleichzeitig sichere Bereitstellung der Software. So schafft die gewählte Architektur die notwendigen Rahmenbedingungen, um ein effizientes Deployment zu ermöglichen und umzusetzen.

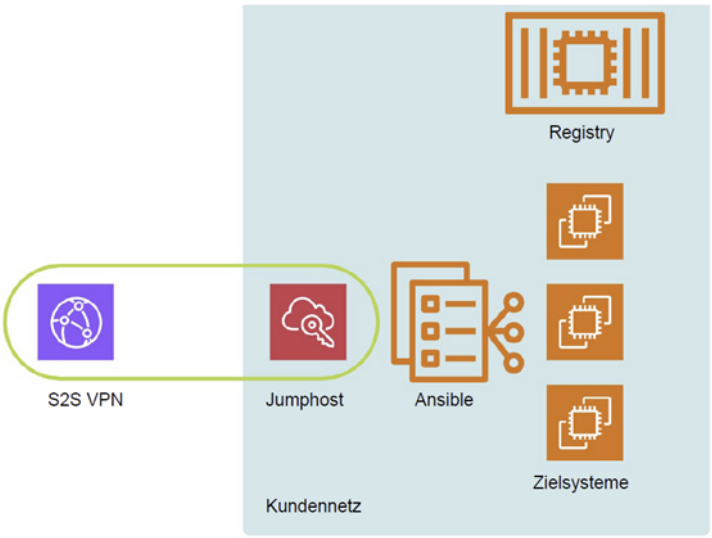

Abbildung 3: Vereinfachte Systemarchitektur

4.2 Container mit Podman

Wie in Abschnitt 2.2 erläutert, bietet Podman eine sichere und daemonlose Container-Engine, die in diesem Projekt als Grundlage für den Deploymentprozess diente. Im praktischen Einsatz lag der Schwerpunkt weniger auf den generellen Eigenschaften von Podman, sondern auf der Nutzung für die Containerisierung der projektspezifischen Softwarekomponenten. Das Deployment besteht aus zwei klar voneinander getrennten Containern:

- **Applikations Container**: Einem Applikations Container, in dem die Mirth® Connect-Instanz sowie projektspezifische Erweiterungen gebündelt sind.
- **Datenbank Container**: Einem Datenbank Container auf Basis eines MariaDB-Images, erweitert um einen SQL-Dump zur Initialisierung.

Die Container wurden so konzipiert, dass eine interne Kommunikation untereinander möglich ist, ohne dass dafür zusätzliche Ports am Hostsystem geöffnet werden müssen. Die Übertragung sensibler Daten, wie etwa Zugangsdaten, erfolgte mittels Environment-Variablen. Diese Vorgehensweise hat sowohl die Flexibilität als auch die Sicherheit des Verfahrens erhöht.

4.2.1 Image Erstellung

Im Rahmen des Containerisierungsprozesses der Datenbank wurde zunächst ein eigenes Image generiert. Die vorliegende Konfiguration basiert auf dem offiziellen MariaDB Image in der Version 11.3.2-jammy, welches aus dem Docker Hub bezogen wird. Die vorliegende Vorgehensweise gewährleistet, dass sicherheitsrelevante Patches sowie die offizielle Wartung durch die Community und das MariaDB-Projekt genutzt werden können. Da der Sinn des automatischen Deployments darin liegt Zeit zu sparen sowie das Deployment zu vereinfachen wird die Datenbank vorkonfiguriert installiert. Der Output ist in A.1 zu finden.

Listing 4.1: Dockerfile Datenbank

```
1  # Use the official MariaDB image from Docker Hub
2  FROM docker.io/library/mariadb:11.3.2-jammy
3
4  # Copy the SQL dump file into the Docker container
5  COPY dump.sql /docker-entrypoint-initdb.d/
6
7  # Expose the MySQL port
8  EXPOSE 3306
9
10 # Start MariaDB service
11 CMD ["mariadbd"]
```

Um dies zu erreichen wurde ein neues Container Image für die Datenbank erstellt, dazu wurde mit dem offiziellen Podman-Image, siehe Abschnitt 2.3 von MariaDB gestartet und ein Datenbankendump mit der Zielkonfiguration in das /docker-entrypoint-initdb.d/ verzeichnis kopiert. So wird sichergestellt, dass beim Deployment der Datenbank die Vorkonfiguration geladen wird. Nach einem erfolgreichen Build-Prozess wird das Image in die eigene Container Registry hochgeladen, damit es den Zielsystemen zur Verfügung steht. Durch dieses Vorgehen wird eine reproduzierbare und konsistente Bereitstellung der Datenbank ermöglicht. Änderungen an der Datenbankstruktur können durch die Anwendung von SQL-Dumps versioniert und somit dokumentiert werden. Dies gewährleistet eine hohe Wartbarkeit und Nachvollziehbarkeit des Datenbank-Deployments. Da der Mirthconnect Container welcher ausgerollt wird, seine Konfiguration in der Datenbank speichert, ist für diesen kein benutzerdefinierter Container nötig und das Standard Image aus der Docker Hub Container Registry welches von Nextgen bereitgestellt wird kann verwendet werden.

4.2.2 Podman Compose

Zur Verwaltung und Orchestrierung der einzelnen Container wurde Podman Compose eingesetzt. Podman Compose orientiert sich stark an Docker Compose und ermöglicht es, mehrere Container über eine gemeinsame Konfigurationsdatei (YAML-Format) zu definieren, zu verknüpfen und gemeinsam zu starten. Damit wird sichergestellt, dass die Anwendung nicht nur aus einzelnen isolierten Containern besteht, sondern als zusammenhängendes System betrieben werden kann. So definiert das Podman Compose zwei zentrale Container, sowie die für deren Zusammenarbeit benötigten Netzwerke und Volumes.

Listing 4.2: Podman Compose Services

```
services:
  mariadb:
  mirth-connect:

networks:
  mirth_network:
    driver: bridge
volumes:
 appdata_volume:
```

4.2.2.1 MariabDB

Der Container mariadb basiert auf dem offiziellen MariaDB Image und stellt die persistente Datenbankumgebung bereit. So werden über den Schlüssel Environment alle erforderlichen Datenbankparameter wie Passwörter, Datenbanknamen, sowie Benutzernamen festgelegt. Im vorliegenden Kontext werden sensible Daten mittels externer Variablen in Form einer sogenannten .ENV-Datei verwaltet, wodurch eine Hinterlegung von Passwörtern im Klartext im Compose-File verhindert wird.

Listing 4.3: Podman Compose MariaDB

```
mariadb:
    container_name: mariadb
    hostname: mariadb
    image: mariadb:latest
    restart: always
    environment:
      MARIADB_ROOT_PASSWORD: admin
      MARIADB_DATABASE : mirthdb
      MARIADB_USER: ${DATABASE_USERNAME}
      MARIADB_PASSWORD: ${DATABASE_PASSWORD}
    volumes:
      - ./data/db:/var/lib/mysql
    networks:
      mirth_network:
```

4.2.2.2 Mirthconnect

Der Mirthconnect Container enthält die eigentliche Applikation (Mirth® Connect) sowie projektspezifische Erweiterungen. Wie auch bei der Datenbank werden unter Environment die für den Betrieb notwendigen Datenbank- und Verbindungsparameter angegeben. Diese werden ebenfalls aus Umgebungsvariablen bezogen. Es sei jedoch angemerkt, dass es einige Besonderheiten gibt, so wird durch die Option „depends_on“ festgelegt, dass der Mirthconnect-Container erst dann gestartet werden darf, nachdem die Datenbank vollständig gestartet ist. Des Weiteren erfolgt die Freigabe des Ports 2443 auf dem Hostsystem über die Option „Ports“, um die Mirthconnect-Applikation zu erreichen.

Listing 4.4: Podman Compose Mirthconnect

```
mirth-connect:
    hostname: mirthconnect
    container_name: mirthconnect
    image: nextgenhealthcare/connect:latest
    restart: unless-stopped

    environment:
      DATABASE: ${DATABASE}
      DATABASE_URL: ${DATABASE_URL}
      DATABASE_MAX_CONNECTIONS: ${DATABASE_MAX_CONNECTIONS}
      DATABASE_USERNAME: ${DATABASE_USERNAME}
      DATABASE_PASSWORD: ${DATABASE_PASSWORD}
      DATABASE_MAX_RETRY: ${DATABASE_MAX_RETRY}
      DATABASE_RETRY_WAIT: ${DATABASE_RETRY_WAIT}
     # KEYSTORE_STOREPASS: ${KEYSTORE_STOREPASS}
     # KEYSTORE_KEYPASS: ${KEYSTORE_KEYPASS}
      VMOPTIONS: ${VMOPTIONS}
      _MP_HTTPS_PORT: ${_MP_HTTPS_PORT}

    ports:
      - 2443:2443
    networks:
      mirth_network:
    volumes:
      - appdata_volume:/opt/connect/appdata
    depends_on:
      - mariadb
```

4.2.2.3 Netzwerk & Volumes

Das interne Netzwerk „mirth_network“ wurde als Bridge-Netzwerk konfiguriert. Die Kommunikation zwischen den Containern erfolgt mittels interner Hostnamen, ohne dass der Datenverkehr das Hostsystem oder das öffentliche Netzwerk durchläuft. Dies resultiert in einer signifikanten Steigerung sowohl der Sicherheit als auch der Performance,

da der Zugriff von außen einer rigorosen Kontrolle unterliegt, ist nur der durch die in Unterunterabschnitt 4.2.2.2 definierte Port 2443 von außen erreichbar.

Listing 4.5: Podman Compose Volumes

```
networks:
  mirth_network:
    driver: bridge
volumes:
 appdata_volume:
```

4.3 Finales Podman Compose

Werden die einzeln erläuterten Komponenten zusammengefügt ergibt sich das folgende Podman Compose: siehe Abschnitt A.2. Der Zusammenschluss ergibt sich aus der Applikation und der Datenbank sowie der Netzwerkkonfiguration, welche später durch das Ansible Playbook ausgerollt werden kann.

Die Containerisierung der Komponenten reduziert den Konfigurationsaufwand und ermöglicht eine wiederholbare Bereitstellung, welche somit direkt zur Optimierung des Deploymentprozesses beiträgt.

4.4 Deployment durch Ansible

Das Ansible Playbook besteht aus mehreren Tasks die zusammen das gesamte Deployment darstellen, wie in Unterabschnitt 3.2.4 definiert. So beginnt das Playbook als erstes mit der Prüfung , ob die Container `mirthconnect` und `mariadb` bereits laufen.

4.4.1 Prüfung des Container-Status

Mithilfe von „podman_container_info“ wird geprüft, ob beide Container existieren und den Status „running“ besitzen:

Listing 4.6: Prüfung des Container-Status von Mirthconnect und MariaDB

```
- name: Gather facts on Mirth
  containers.podman.podman_container_info:
    name: mirthconnect

- name: Gather facts on DB
  containers.podman.podman_container_info:
    name: mariadb
```

Sind beide Container bereits aktiv, wird kein komplettes Deployment durchgeführt, sondern lediglich ein leichtgewichtiger Updateprozess gestartet.

4.4.2 Leichtgewichtiger Updateprozess

Falls die Container laufen, genügt es die Images zu aktualisieren und anschließend die Container erneut im Hintergrund zu starten:

Listing 4.7: Pull newest Images and Update

```
1 - name: Pull latest images using Podman Compose
2   command: podman-compose pull
3
4 - name: Start services
5   command: podman-compose up -d
```

Dieser Weg spart Zeit und minimiert unnötige Änderungen auf den Zielsystemen.

4.4.3 Normales Deployment

Wenn die Container nicht laufen, wird das vollständige Deployment wie zuvor beschrieben durchgeführt:

1. Rücksetzen des Podman Systems
2. Anlegen des Arbeitsverzeichnisses
3. Übertragen der Konfigurationsdateien (`.env`, `docker-compose.yml`)
4. Start der Container mit `podman-compose up -d`
5. Analyse der Exit Codes
6. Validierung der Containerzustände

Das Playbook beginnt bei einem normalen Rollout mit der Vorbereitung der Zielsysteme für das Container-Deployment. Zunächst wird das Podman System auf jedem Host zurückgesetzt, um sicherzustellen, dass keine alten oder fehlerhaften Containerreste vorhanden sind:

Listing 4.8: Reset Podman System for clean Workspace

```
1 - name: Reset Podman system
2   ansible.builtin.command: podman system reset -f
```

Anschließend wird ein dediziertes Verzeichnis für die Containerkonfiguration erstellt:

```
1 - name: Create podman directory in the home directory
2   ansible.builtin.file:
3     path: "{{ ansible_env.HOME }}/podman_test"
4     state: directory
```

Die notwendigen Konfigurationsdateien, wie `.env` und `docker-compose.yml`, werden auf das Zielsystem übertragen:

Listing 4.9: Copy Mirthconnect and MariabDB Compose Files

```
1 - name: Copy .env file to podman directory
2 - name: Copy docker-compose.yml file to podman directory
```

Diese Schritte sichern eine konsistente Umgebung auf allen Hosts und reduzieren menschliche Fehler beim manuellen Kopieren.

4.4.4 Container Start

Nach der Übertragung startet das Playbook die Container im Hintergrund mithilfe des command Moduls. Dies erfolgt mit dem Befehlt `podman-compose up -d`.

Listing 4.10: Start Containers

```
1 - name: Run podman-compose up -d
2   ansible.builtin.command: podman-compose up -d
```

Danach werden die Exit-Codes analysiert, um den Status des Deployments zu bestimmen:

Listing 4.11: Check Deployment with Return Codes

```
1 Exit-Code 0: Container erstellt
2 Exit-Code 125: Container bereits gestartet
3 Andere Codes: Fehler beim Start - name: Extract all exit codes
    using grep - name: Convert exit codes to list of integers
```

Basierend auf diesen Codes gibt das Playbook differenzierte Statusmeldungen aus und der betroffene Task schlägt fehl, falls der Start nicht erfolgreich war.

4.4.5 Validierung / Health Check

Nach dem Deployment werden die Container auf ihren Status geprüft, da es vorkommen kann, dass ein Container zwar ausgerollt wurde, aber nicht erfolgreich gestartet ist. Zuerst der Mirth-Connect-Container:

Listing 4.12: Check Container-Status of Mirthconnect

```
1 - name: Gather facts on the Mirth
2 - name: Check and display Mirth status or fail if not running
```

Analog wird der MariaDB-Container überprüft:

Listing 4.13: Check Container-Status of MariaDB

```
1 - name: Gather facts on the DB
2 - name: Check and display DB status or fail if not running
```

4.4.6 Rollback-Mechanismus

Schlägt das Deployment fehl, wird die zuvor gesicherte `docker-compose.yml` zurückgespielt und die Container erneut gestartet:

Listing 4.14: Rollback Task

```
1  - name: Rollback to previous config
2    copy:
3      src: "{{ ansible_env.HOME
         }}/podman_test/docker-compose.yml.bak"
4      dest: "{{ ansible_env.HOME
         }}/podman_test/docker-compose.yml"
5      remote_src: true
6
7  - name: Restart podman-compose with backup
8    command: podman-compose up -d
```

Dadurch bleibt das System auch bei einem fehlerhaften Update in einem funktionsfähigen Zustand. Das Playbook bricht dennoch mit einem `fail` ab, damit der Administrator über den Fehler informiert wird.

Das Playbook automatisiert alle notwendigen Schritte und reduziert manuelle Eingriffe auf ein Minimum, ein entscheidender Faktor für Effizienzsteigerung und Fehlerreduktion. Was somit wiederum zur Optimierung des Rollout Prozesses beiträgt.

4.5 Automatisierungsprozess

Das Playbook vereint mehrere Aspekte der Infrastrukturautomatisierung:

- **Fehlerprävention:** Durch Abfrage des Container Status wird nur dann ein volles Deployment ausgeführt, wenn es notwendig ist.
- **Fehlererkennung:** Die Exit Codes, der Container Status und Rückmeldungen von `podman-compose` werden dynamisch ausgewertet.
- **Rollback:** Bei fehlerhaften Deployments wird automatisch die letzte funktionierende Konfiguration wiederhergestellt.
- **Statusüberwachung:** Mirthconnect und MariaDB werden am Ende erneut geprüft, um sicherzustellen, dass die Dienste laufen.

Die Segmentierung in initiales Deployment und Statusüberprüfung verbessert die Wartbarkeit und erleichtert die Fehlersuche. Der Ansatz demonstriert, wie durch Ansible eine reproduzierbare, zuverlässige und automatisierte Containerbereitstellung umgesetzt werden kann.

Wird nun das vollständige Playbook siehe Abschnitt A.3 ausgeführt erzeugt es die folgende Konsolenausgabe siehe Abschnitt A.4.

4.6 Geschwindigkeitsmessung

Zur Bewertung der Effizienz eines automatisierten Deployments wurde die Ausführungszeit des Ansible Playbooks auf zwei Zielsystemen gemessen. Die Messung wurde unter Verwendung einer Zeitmessung durchgeführt, um die Dauer des Deployments exakt zu erfassen. Im Rahmen der Analyse wurde festgestellt, dass das Deployment mittels des Ansible Skripts einen durchschnittlichen Zeitbedarf von 30 Sekunden aufwies. Ein manuelles Deployment hingegen benötigte durch das Verbinden zu den Systemen, sowie eingeben der Befehle zum Installieren der verschiedenen Abhängigkeiten 5 Minuten. So konnte ein erheblicher Effizienzzuwachs festgestellt werden, welcher umso größer skaliert desto mehr Systeme im Deployment enthalten sind. Diese Messung fungiert als Referenzwert für den Vergleich mit einem manuellen Deployment und ermöglicht eine quantitative Beurteilung der Zeitersparnis, die durch die Automatisierung erzielt wird. Anzumerken ist, dass die Ausführung in einem Testaufbau unter Proxmox mit insgesamt drei Virtuellen Maschinen erfolgte: einer Host-VM und zwei Zielsystemen. Die virtuellen Maschinen waren über 10GBE angebunden und die darunter liegende Hardware wird durch einen Server mit einem 24-Kern AMD EPYC Prozessor bereitgestellt.

4.7 WebUI

Zur systematischen Ausführung von Ansible Playbooks sowie zur konsolidierten Erfassung der zugehörigen Rückgabewerte wurde eine webbasierte Benutzeroberfläche (Web User Interface, WebUI) entwickelt. Ziel dieser Implementierung war es, die Berührungen mit der Konsole für Nutzer zu reduzieren und gleichzeitig eine transparente Übersicht über den Ausführungsstatus sowie aufgetretene Fehlermeldungen bereitzustellen.

Die WebUI fungiert als Vermittlungsschicht zwischen den Benutzerinteraktionen und der Ansible-Engine und bietet die folgenden Funktionalitäten:

- **Selektive Playbook-Ausführung:** Anwender sind in der Lage, spezifische Playbooks aus einem definierten Verzeichnis direkt über die Oberfläche zu starten.
- **Zentralisierte Ausgabeerfassung:** Standardausgabe (stdout), Standardfehler (stderr) und die Rückgabewerte der Playbook-Ausführung werden erfasst, konsolidiert und strukturiert aufbereitet.
- **Strukturierte Ergebnisdarstellung:** Insbesondere die Zusammenfassung der einzelnen Hosts (PLAY RECAP) wird in tabellarischer Form präsentiert, sodass der Erfolg oder Misserfolg jeder Operation pro Host unmittelbar nachvollzogen werden kann.

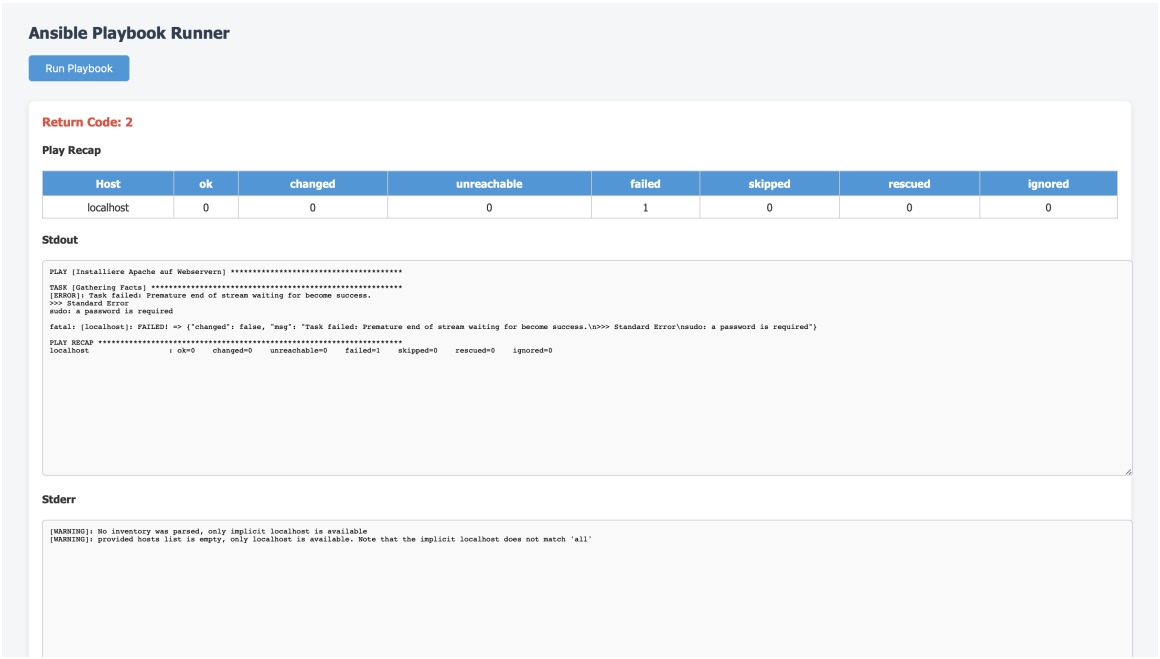

Abbildung 4: Simples WebUI in Python

Das WebUI stellt die Implementierung eines benutzerorientierten Monitoring-Systems für die Infrastrukturautomatisierung dar. Durch diese Lösung werden sowohl die Benutzerfreundlichkeit erhöht als auch die Effizienz der Fehlerdiagnose und die Präzision der Statusanalyse komplexer Automatisierungsabläufe signifikant verbessert.

So liefert die im diesem Kapitel dargestellte Implementierung die Basis für eine Bewertung des entwickelten Ansatzes. Im folgenden Kapitel werden die Ergebnisse diskutiert, die Effizienz des Verfahrens analysiert und mögliche Weiterentwicklungen aufgezeigt.

5 Diskussion und Evaluation

5.1 Fazit

Die zentrale Forschungsfrage dieser Arbeit lautete, wie sich Deploymentprozesse mithilfe von Ansible und Containertechnologien automatisieren und effizienter gestalten lassen, sodass wiederholbare, wartbare und fehlerfreie Rollouts über viele Systeme hinweg möglich sind.

Die Ergebnisse zeigen, dass dieser Anspruch durch die entwickelte Lösung erfüllt werden konnte. Ansible ermöglicht es, komplexe Installations- und Konfigurationsschritte strukturiert in Playbooks abzubilden und zentral zu steuern. Dadurch wird eine fehlerreduzierte und reproduzierbare Ausführung über beliebig viele Zielsysteme hinweg sichergestellt. Die Einbindung von Podman Containern sorgt zusätzlich für standardisierte, portierbare und konsistente Umgebungen, wodurch der Konfigurationsaufwand reduziert und die Wartbarkeit verbessert wird.

Darüber hinaus konnte nachgewiesen werden, dass sich der Deploymentprozess signifikant beschleunigen lässt. Wie in Abschnitt 4.6 dargestellt, reduziert die Automatisierung die Ausführungszeit auf durchschnittlich 30 Sekunden und senkt zugleich den personellen Aufwand. Damit trägt der Ansatz nicht nur zur Effizienzsteigerung, sondern auch zur Entlastung administrativer Ressourcen bei. Insgesamt beantwortet die Arbeit die Forschungsfrage positiv, die Kombination von Ansible und Containerisierung bietet einen zukunftsfähigen Ansatz für effiziente, sichere und konsistente Deploymentprozesse. Sie stellt eine deutliche Verbesserung gegenüber traditionellen und manuellen Verfahren dar und legt die Grundlage für eine nachhaltige Optimierung in komplexen IT-Umgebungen.

Kritische Reflexion

Trotz dieser positiven Ergebnisse bestehen gewisse Einschränkungen. Die Evaluation erfolgte in einer Testumgebung mit begrenzter Systemanzahl, sodass die Übertragbarkeit nur bedingt bewertet werden konnte. So findet das Deployment zwar aktuell bei der Firma Vireq Software Solutions Verwendung, konnte aber bisher noch nicht auf sehr großen Infrastrukturen oder hochdynamischen Multi-Cloud-Umgebungen getestet werden. Außerdem erfolgte das initiale Deployment nicht vollständig automatisiert, da Zielsysteme und Verbindungseigenschaften manuell hinterlegt werden mussten. Hier besteht noch

Optimierungspotenzial, insbesondere durch die Integration in CI/CD-Pipelines oder durch eine erweiterte Parametrisierung innerhalb der WebUI.

Darüber hinaus wurde der Vergleich mit alternativen Automatisierungs- und Containerlösungen nur auf theoretischer Ebene betrachtet. Ein empirischer Vergleich mit Tools wie Puppet, Chef oder Kubernetes könnte weitere Erkenntnisse über die jeweiligen Stärken und Schwächen liefern.

Diese Grenzen zeigen, dass die Arbeit einen wichtigen und ersten Schritt darstellt. Für zukünftige Untersuchungen wäre es sinnvoll, die Lösung in größeren, produktiven Infrastrukturen zu evaluieren, um Skalierbarkeit, Robustheit und Sicherheit noch umfassender beurteilen zu können.

5.2 Produktive Anwendung bei Vireq Software Solutions

Das in dieser Arbeit präsentierte Konzept findet produktive Anwendung bei der Firma Vireq Software Solutions. An dieser Stelle erfolgt der Einsatz zur effizienten und automatisierten Durchführung von Softwarebereitstellungen. Die Kombination von Ansible und Containertechnologien ermöglichte die Schaffung eines skalierbaren und wartbaren Deploymentprozesses, der eine zentrale Verwaltung der Zielsysteme sowie deren kontinuierliche Aktualisierung ermöglicht. Das Konzept leistet demnach einen wesentlichen Beitrag zur Optimierung der betrieblichen Abläufe und zur Sicherstellung einer hohen Zuverlässigkeit im Produktiveinsatz.

5.3 Ausblick

Um den in dieser Arbeit dargestellten Ansatz des Deployments effizienter zu gestalten, könnte das Logging der Automatisierung weiter verbessert werden, sodass bei Fehlschlägen des Update- oder Deploymentprozesses eine Fehlermeldung via E-Mail oder Push-Benachrichtigung gesendet wird. Aktuell wird das initiale Deployment noch nicht vollautomatisch ausgeführt, da zuerst immer noch die Zielsysteme sowie Verbindungseigenschaften definiert werden müssen. Diese könnten als angebbare Parameter im WEbUI eingebaut werden.

5.3.1 Mögliche Weiterentwicklungen

Für die Zukunft ergeben sich mehrere Ansatzpunkte zur Erweiterung und Optimierung des in dieser Arbeit dargestellten Deploymentsystems:

- **Automatisiertes Initial-Deployment:** Derzeit erfordert die Definition von Zielsystemen und Verbindungseigenschaften noch manuelle Eingaben. Eine Erweiterung der

WebUI um parametrisierbare Vorlagen und automatische Erkennung von Zielsystemen könnte diesen Prozess vollständig automatisieren.

- **Erweitertes Monitoring und Logging:** Das Logging könnte so erweitert werden, dass bei Fehlern während des Deployments automatisch Benachrichtigungen per E-Mail, Push-Mitteilung oder in ein zentrales Monitoring-System gesendet werden. Zusätzlich könnten detaillierte Metriken über Laufzeiten und Ressourcenauslastung der Playbooks erfasst werden.
- **Rollenbasierte Benutzerverwaltung:** Eine fein granulare Rechteverwaltung innerhalb der WebUI würde es ermöglichen, unterschiedliche Benutzergruppen mit spezifischen Berechtigungen für die Ausführung und Verwaltung von Playbooks zu versehen.
- **Integration von Continuous Integration / Continuous Deployment (CI/CD):** Die WebUI könnte an bestehende CI/CD-Pipelines angebunden werden, um Deployments automatisch nach erfolgreichem Build oder Test auszulösen.
- **Erweiterung auf hybride und Multi-Cloud-Umgebungen:** Der Ansatz könnte so skaliert werden, dass Deployments über unterschiedliche Cloud-Anbieter hinweg koordiniert werden und hybride On-Premise- und Cloud-Infrastrukturen unterstützt werden.
- **Optimierung der Performance und Skalierbarkeit:** Durch Parallelisierung der Playbook-Ausführung und Lastverteilung auf mehrere Worker-Instanzen könnte die Effizienz bei großflächigen Deployments weiter erhöht werden.
- **Erweiterte Visualisierung der Ergebnisse:** Die tabellarische Darstellung der Playbook-Ergebnisse könnte durch Dashboards ergänzt werden, die Trends, historische Fehlerhäufigkeiten und Metriken der Zielsysteme visualisieren.

Abbildungsverzeichnis

Listings

Anhang A

Anhang

A.1 Podman Image Build

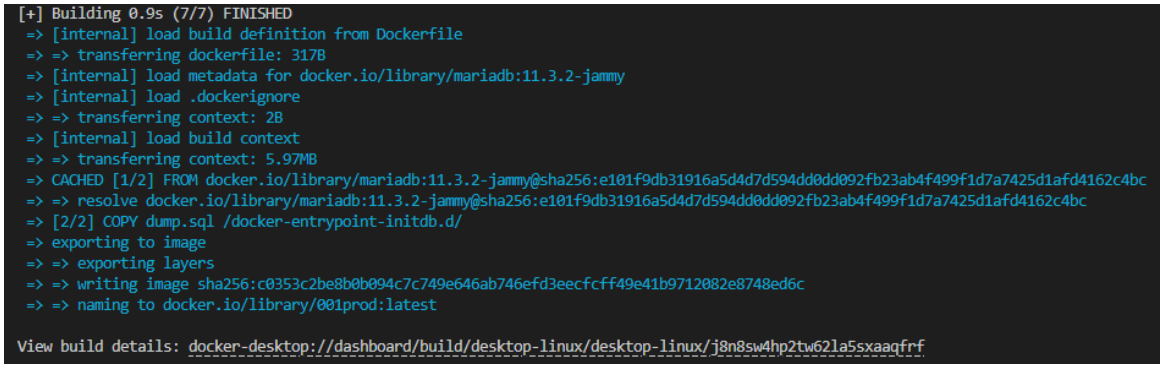

Abbildung 5: Build Log Image

A.2 Podman Compose File

Listing A.1: Podman Compose Datei

```
services:
  mariadb:
    container_name: mariadb
    hostname: mariadb
    image: mariadb:latest
    restart: always
    environment:
      MARIADB_ROOT_PASSWORD: admin
      MARIADB_DATABASE : mirthdb
      MARIADB_USER: ${DATABASE_USERNAME}
      MARIADB_PASSWORD: ${DATABASE_PASSWORD}
    volumes:
      - ./data/db:/var/lib/mysql
```

```
14      networks:
15        mirth_network:
16
17    mirth-connect:
18      hostname: mirthconnect
19      container_name: mirthconnect
20      image: nextgenhealthcare/connect:latest
21      restart: unless-stopped
22
23      environment:
24        DATABASE: ${DATABASE}
25        DATABASE_URL: ${DATABASE_URL}
26        DATABASE_MAX_CONNECTIONS: ${DATABASE_MAX_CONNECTIONS}
27        DATABASE_USERNAME: ${DATABASE_USERNAME}
28        DATABASE_PASSWORD: ${DATABASE_PASSWORD}
29        DATABASE_MAX_RETRY: ${DATABASE_MAX_RETRY}
30        DATABASE_RETRY_WAIT: ${DATABASE_RETRY_WAIT}
31       # KEYSTORE_STOREPASS: ${KEYSTORE_STOREPASS}
32       # KEYSTORE_KEYPASS: ${KEYSTORE_KEYPASS}
33        VMOPTIONS: ${VMOPTIONS}
34        _MP_HTTPS_PORT: ${_MP_HTTPS_PORT}
35
36      ports:
37        - 2443:2443
38      networks:
39        mirth_network:
40      volumes:
41        - appdata_volume:/opt/connect/appdata
42      depends_on:
43        - mariadb
44
45  networks:
46    mirth_network:
47      driver: bridge
48  volumes:
49   appdata_volume:
```

A.3 Gesamtes Ansible Playbook

Listing A.2: Gesamtes Ansible Playbook

```
1  ---
2  - name: Check running containers first
3    hosts: prod
4    remote_user: root
5    tasks:
6
7      - name: Gather facts on Mirth
8        containers.podman.podman_container_info:
9          name: mirthconnect
```

```
10        register: mirth_info
11        ignore_errors: true
12
13      - name: Gather facts on DB
14        containers.podman.podman_container_info:
15          name: mariadb
16        register: db_info
17        ignore_errors: true
18
19      - name: Set fact if both containers are running
20        ansible.builtin.set_fact:
21          containers_running: >-
22            {{ (mirth_info.containers | length > 0 and
                  mirth_info.containers[0].State.Status == "running")
23            and (db_info.containers | length > 0 and
                  db_info.containers[0].State.Status == "running") }}
24
25      - name: Run lightweight update if containers are already running
26        block:
27          - name: Pull latest images using Podman Compose
28            ansible.builtin.command: podman-compose pull
29            args:
30              chdir: "{{ ansible_env.HOME }}/podman_test"
31            register: pull_result
32            changed_when: "'Pull complete' in pull_result.stdout or
                  'Downloaded newer image' in pull_result.stdout"
33            failed_when: pull_result.rc != 0
34
35          - name: Start services
36            ansible.builtin.command: podman-compose up -d
37            args:
38              chdir: "{{ ansible_env.HOME }}/podman_test"
39            when: pull_result.changed
40            register: start_result
41            failed_when: start_result.rc != 0
42        when: containers_running
43
44      - name: Full deployment if containers are not running
45        block:
46
47          - name: Backup existing podman-compose config
48            ansible.builtin.copy:
49              src: "{{ ansible_env.HOME }}/podman_test/docker-compose.yml"
50              dest: "{{ ansible_env.HOME
                      }}/podman_test/docker-compose.yml.bak"
51              remote_src: true
52            ignore_errors: true
53
54          - name: Reset Podman system
55            ansible.builtin.command: podman system reset -f
56            args:
57              creates: /var/lib/containers/storage
```

```
58
59          - name: Create podman directory in the home directory
60            ansible.builtin.file:
61              path: "{{ ansible_env.HOME }}/podman_test"
62              state: directory
63              mode: '0755'
64
65          - name: Copy .env file to podman directory
66            ansible.builtin.copy:
67              src: "{{ playbook_dir }}/env"
68              dest: "{{ ansible_env.HOME }}/podman_test/.env"
69              mode: '0644'
70
71          - name: Copy docker-compose.yml file to podman directory
72            ansible.builtin.copy:
73              src: ./docker-compose.yml
74              dest: "{{ ansible_env.HOME
                    }}/podman_test/docker-compose.yml"
75              mode: '0644'
76
77          - name: Run podman-compose up -d
78            ansible.builtin.command: podman-compose up -d
79            args:
80              chdir: "{{ ansible_env.HOME }}/podman_test"
81            register: podman_compose_result
82            failed_when: podman_compose_result.rc != 0
83
84          - name: Extract exit codes
85            ansible.builtin.shell: |
86              set -o pipefail
87              echo "{{ podman_compose_result.stderr }}" | grep -o 'exit
                    code: [0-9]\+' | awk '{print $3}' || true
88            register: exit_codes_result
89            changed_when: false
90
91          - name: Convert exit codes to list of integers
92            ansible.builtin.set_fact:
93              exit_codes: "{{ exit_codes_result.stdout_lines | map('int')
                    | list }}"
94
95          - name: Fail if all exit codes are non-zero
96            ansible.builtin.fail:
97              msg: "podman-compose up -d failed with exit codes {{
                    exit_codes }}"
98            when: exit_codes | select('eq', 0) | list | length == 0
99
100         - name: Wait for containers
101           ansible.builtin.wait_for:
102             timeout: 10
103
104         - name: Verify Mirth container
105           block:
```

```
106             - name: Gather facts on the Mirth
107               containers.podman.podman_container_info:
108                 name: mirthconnect
109               register: mirth_final
110
111             - name: Check if Mirth is running
112               ansible.builtin.fail:
113                 msg: "Mirth is not running after deployment"
114               when: mirth_final.containers | length == 0 or
                      mirth_final.containers[0].State.Status != "running"
115
116         - name: Verify DB container
117           block:
118             - name: Gather facts on the DB
119               containers.podman.podman_container_info:
120                 name: mariadb
121               register: db_final
122
123             - name: Check if DB is running
124               ansible.builtin.fail:
125                 msg: "DB is not running after deployment"
126               when: db_final.containers | length == 0 or
                      db_final.containers[0].State.Status != "running"
127
128       rescue:
129         - name: Rollback to previous config
130           ansible.builtin.copy:
131             src: "{{ ansible_env.HOME
                    }}/podman_test/docker-compose.yml.bak"
132             dest: "{{ ansible_env.HOME
                    }}/podman_test/docker-compose.yml"
133             remote_src: true
134           ignore_errors: true
135
136         - name: Restart podman-compose with backup
137           ansible.builtin.command: podman-compose up -d
138           args:
139             chdir: "{{ ansible_env.HOME }}/podman_test"
140           ignore_errors: true
141
142         - name: Fail the play after rollback
143           ansible.builtin.fail:
144             msg: "Deployment failed, rollback executed with backup
                  docker-compose.yml"
145       when: not containers_running
```

A.4 Ansible Deployment Output

Listing A.3: Ansible Playbook Output

```
PLAY [Check running containers first]
**********

TASK [Gathering Facts]
ok: [server1]
ok: [server2]

TASK [Gather facts on Mirth]
ok: [server1]
ok: [server2]

TASK [Gather facts on DB]
ok: [server1]
ok: [server2]

TASK [Set fact if both containers are running]
ok: [server1]
ok: [server2]

TASK [Run lightweight update if containers are already running]
skipping: [server1]
skipping: [server2]

TASK [Backup existing podman-compose config]
ok: [server1]
ok: [server2]

TASK [Reset Podman system]
ok: [server1]
ok: [server2]

TASK [Create podman directory in the home directory]
ok: [server1]
ok: [server2]

TASK [Copy .env file to podman directory]
ok: [server1]
ok: [server2]

TASK [Copy docker-compose.yml file to podman directory]
ok: [server1]
ok: [server2]

TASK [Run podman-compose up -d]
ok: [server1]
ok: [server2]

TASK [Extract exit codes]
ok: [server1]
ok: [server2]

TASK [Convert exit codes to list of integers]
```

```
53 ok: [server1]
54 ok: [server2]
55
56 TASK [Fail if all exit codes are non-zero]
57 skipping: [server1]
58 skipping: [server2]
59
60 TASK [Wait for containers]
61 ok: [server1]
62 ok: [server2]
63
64 TASK [Verify Mirth container : Gather facts on the Mirth]
65 ok: [server1]
66 ok: [server2]
67
68 TASK [Verify Mirth container : Check if Mirth is running]
69 failed: [server2] => {
70     "msg": "Mirth is not running after deployment"
71 }
72 skipping: [server1]
73
74 TASK [Verify DB container : Gather facts on the DB]
75 ok: [server1]
76 ok: [server2]
77
78 TASK [Verify DB container : Check if DB is running]
79 ok: [server1]
80 ok: [server2]
81
82 TASK [Rollback to previous config]
83 ok: [server2]
84
85 TASK [Restart podman-compose with backup]
86 changed: [server2]
87
88 TASK [Fail the play after rollback]
89 fatal: [server2]: FAILED! => {
90     "msg": "Deployment failed, rollback executed with backup
          docker-compose.yml"
91 }
92
93 PLAY RECAP
94 server1                    : ok=15   changed=0    unreachable=0
     failed=0    skipped=3    rescued=0    ignored=0
95 server2                    : ok=18   changed=1    unreachable=0
     failed=1    skipped=1    rescued=0    ignored=0
```

A.5 WebUI Flask App Code

Listing A.4: WebUI Flask App

```
from flask import Flask, render_template_string
import subprocess
import re
import glob

app = Flask(__name__)

TEMPLATE = """
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Ansible Playbook Runner</title>
  <style>
    body { font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif; margin: 40px; background-color: #f4f6f8; color: #333; }
    h2 { color: #2c3e50; }
    form { margin-bottom: 30px; }
    button { background-color: #3498db; color: white; border: none; padding: 10px 25px; font-size: 16px; border-radius: 5px; cursor: pointer; transition: 0.2s; }
    button:hover { background-color: #2980b9; }
    textarea { width: 100%; height: 300px; font-family: monospace; padding: 10px; border-radius: 5px; border: 1px solid #ccc; background-color: #f9f9f9; resize: vertical; }
    table { width: 100%; border-collapse: collapse; margin-bottom: 20px; }
    table, th, td { border: 1px solid #ccc; }
    th { background-color: #3498db; color: white; padding: 8px; text-align: center; }
    td { text-align: center; padding: 6px; }
    .result-section { background-color: white; padding: 20px; border-radius: 8px; box-shadow: 0px 2px 6px rgba(0,0,0,0.1); margin-bottom: 20px; }
    .return-code { font-size: 18px; font-weight: bold; color: #e74c3c; }
  </style>
</head>
<body>
  <h2>Ansible Playbook Runner</h2>
  <form method="post" action="/run">
    <button type="submit">Run Playbook</button>
  </form>

  {% if result %}
  <div class="result-section">
    <div class="return-code">Return Code: {{ result.returncode }}</div>
```

```
38
39      <h4>Play Recap</h4>
40      {% if result.play_recap %}
41        <table>
42          <tr>
43            <th>Host</th><th>ok</th><th>changed</th><th>unreachable</th><
                th>failed</th>
44            <th>skipped</th><th>rescued</th><th>ignored</th>
45          </tr>
46          {% for host,row in result.play_recap.items() %}
47          <tr>
48            <td>{{ host }}</td>
49            {% for v in row %}<td>{{ v }}</td>{% endfor %}
50          </tr>
51          {% endfor %}
52        </table>
53      {% else %}
54        <p><em>No PLAY RECAP found in output.</em></p>
55      {% endif %}
56
57      <h4>Stdout</h4>
58      <textarea readonly>{{ result.stdout }}</textarea>
59
60      <h4>Stderr</h4>
61      <textarea readonly>{{ result.stderr }}</textarea>
62    </div>
63    {% endif %}
64  </body>
65  </html>
66  """
67
68  def find_playbook():
69      files = glob.glob("*.yml")
70      return files[0] if files else None
71
72  def parse_play_recap(output):
73      recap_idx = output.find('PLAY RECAP')
74      if recap_idx == -1:
75          return None
76      recap_text = output[recap_idx:]
77      pattern = re.compile(
78          r"^\s*(?P<host>\S+)\s*:\s*ok=(?P<ok>\d+)\s+changed=(?P<changed
              >\d+)\s+unreachable=(?P<unreachable>\d+)\s+failed=(?P<failed
              >\d+)\s+skipped=(?P<skipped>\d+)\s+rescued=(?P<rescued>\d+)\s
              +ignored=(?P<ignored>\d+)",
79          re.MULTILINE
80      )
81      result = {}
```

```
    for m in pattern.finditer(recap_text):
        host = m.group('host')
        row = [int(m.group(g)) for g in ('ok','changed','unreachable','
           failed','skipped','rescued','ignored')]
        result[host] = row
    return result if result else None

@app.route('/')
def index():
    return render_template_string(TEMPLATE, result=None)

@app.route('/run', methods=['POST'])
def run_playbook():
    playbook = find_playbook()
    if not playbook:
        result = type('R', (), {})()
        result.returncode = -1
        result.stdout = ''
        result.stderr = 'No .yml playbook found in current directory'
        result.play_recap = None
        return render_template_string(TEMPLATE, result=result)

    # Always run locally using --connection=local
    try:
        proc = subprocess.run(
            ['ansible-playbook', '--connection=local', playbook],
            capture_output=True, text=True, timeout=3600
        )
    except subprocess.TimeoutExpired as e:
        result = type('R', (), {})()
        result.returncode = -1
        result.stdout = e.stdout or ''
        result.stderr = 'Timeout expired'
        result.play_recap = None
        return render_template_string(TEMPLATE, result=result)

    stdout = proc.stdout
    stderr = proc.stderr
    returncode = proc.returncode
    play_recap = parse_play_recap(stdout + "\n" + stderr)

    result = type('R', (), {})()
    result.returncode = returncode
    result.stdout = stdout
    result.stderr = stderr
    result.play_recap = play_recap

    return render_template_string(TEMPLATE, result=result)
```

```
129
130 if __name__ == '__main__':
131     app.run(host='0.0.0.0', port=5002)
```