WHZ Westsächsische
Hochschule Zwickau
Hochschule für Mobilität

Konzeption und prototypische Realisierung einer sicheren Deployment-Architektur für selbst gehostete GitHub-Runner an der WHZ

Fakultät für Physikalische Technik und Informatik
Fachgruppe Informatik

Bachelorarbeit

vorgelegt von

Luke Heydel

Matrikelnummer: 47052

betreut von

Nico Herbig, M.Sc.

und

Prof. Dr. Frank Grimm

im Wintersemester 2025/2026

Kurzfassung

Selbst gehostete GitHub-Runner ermöglichen die Ausführung von CI/CD-Workflows innerhalb organisationsspezifischer Infrastrukturen, sind jedoch mit Herausforderungen hinsichtlich Isolation, Persistenz und Wartbarkeit verbunden. Diese Arbeit untersucht die Konzeption und prototypische Umsetzung einer Architektur für ephemere, VM-basierte GitHub-Runner in einer bestehenden VMware-Infrastruktur.

Grundlage der Untersuchung ist die Analyse der bestehenden Runner-Umgebung an der Westsächsischen Hochschule Zwickau, in der Persistenz zwischen Job-Ausführungen und eingeschränkte Isolation als zentrale Schwachstellen identifiziert werden. Darauf aufbauend werden Anforderungen an eine Zielarchitektur abgeleitet und verschiedene Mechanismen zur Bereitstellung und Rücksetzung virtueller Maschinen im Hinblick auf ihre Eignung für den Betrieb ephemerer Runner bewertet.

Als Ergebnis wird eine Zielarchitektur entwickelt, die GitHub-Webhooks, einen Orchestrator, Ansible-basierte Automatisierung und VMware-basierte Instanziierung kombiniert. Für die Bereitstellung und Rückführung der Runner-Instanzen wird Instant Clone als geeigneter Mechanismus ausgewählt. Auf dieser Grundlage wird ein Prototyp implementiert, der sowohl den Lifecycle einzelner Runner-Instanzen als auch den Deployment- und Update-Prozess der Basisinstanz automatisiert.

Die Evaluation zeigt, dass die wesentlichen Anforderungen im Rahmen des Prototyps erfüllt werden. Insbesondere werden eine reproduzierbare Ausgangsumgebung, die automatisierte Rückführung von Runner-Instanzen sowie die Nicht-Persistenz datei- und containerbezogener Zustände zwischen einzelnen Job-Ausführungen nachgewiesen. Damit leistet die Arbeit einen Beitrag zur sicheren und wartungsarmen Bereitstellung selbst gehosteter GitHub-Runner in virtualisierten Hochschul- und Organisationsumgebungen.

Schlagwörter: GitHub Actions, Self-Hosted Runner, Ephemeral Runner, VMware, Instant Clone, Ansible, CI/CD

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

ARC Action Runner Controller

CI/CD Continuous Integration and Continuous Deployment

VM Virtuelle Maschine

VMM Virtual Machine Monitor

WHZ Westsächsischen Hochschule Zwickau

ZKI Zentrum für Kommunikationstechnik und Informationsverarbeitung

1. Einleitung

Die Entwicklung von Software hat sich in den vergangenen Jahren grundlegend gewandelt. Während frühere Entwicklungsprozesse durch lange Releasezyklen und manuelle Integrationsschritte gekennzeichnet waren, werden heutige Systeme durch kontinuierliche Weiterentwicklung, schnelle Fehlerbehebung und unmittelbare Bereitstellung neuer Funktionalitäten charakterisiert. In diesem Kontext sind automatisierte Build-, Test- und Deploymentprozesse nicht länger optional, sondern stellen die infrastrukturelle Grundlage moderner Softwareentwicklung dar.

Continuous Integration and Continuous Deployment (CI/CD) adressieren exakt diese Anforderungen. Durch die automatisierte Integration von Codeänderungen, reproduzierbare Testausführungen und standardisierte Bereitstellungsmechanismen können Integrationsrisiken reduziert, die Transparenz im Entwicklungsprozess erhöht und kurze Feedbackzyklen ermöglicht werden.

Eine empirische Untersuchungen im DevOps-Kontext zeigte, dass ein hoher Grad an Automatisierung mit verbesserter Softwarequalität, höherer Deployment-Frequenz und verkürzter Wiederherstellungszeit nach Fehlern korreliert [1]. CI/CD stellen demnach zentrale Bausteine leistungsfähiger Entwicklungsteams dar.

1.1. Motivation und Problemstellung

Während cloudbasierte Ausführungsumgebungen für CI/CD eine sofort nutzbare, standardisierte Infrastruktur bereitstellen, liegt bei selbst betriebenen Ausführungsumgebungen die Verantwortung beim Betreiber. Die Bereitstellung, Konfiguration, Isolation und Wartung der Ausführungsumgebungen müssen oftmals eigenständig konzipiert und umgesetzt werden. Insbesondere in institutionellen Infrastrukturen mit heterogenen Nutzergruppen resultieren daraus spezifische sicherheits- und betriebsrelevante Anforderungen.

An der Westsächsischen Hochschule Zwickau (WHZ) wird ein GitHub Enterprise Server[1] betrieben und durch das Zentrum für Kommunikationstechnik und Informationsverarbeitung (ZKI) verwaltet. Das ZKI ist die IT-Abteilung der Hochschule.

[1] GitHub Enterprise Server ist eine selbstgehostete Version der GitHub-Plattform.[2]

GitHub Actions wird als CI/CD-Plattform genutzt, da dieses im GitHub Enterprise Server integriert ist. Hierfür kommen selbst gehostete Runner als Ausführungsinstanzen innerhalb einer bestehenden VMware-basierten Virtualisierungsumgebung zum Einsatz. Die vorliegende Infrastruktur erlaubt sowohl den Mitarbeitenden des ZKI als auch den Studierenden die Ausführung von CI/CD auf hochschuleigenen Ressourcen.

Die aktuelle Architektur basiert auf persistent betriebenen virtuellen Maschinen. Im Rahmen einer CI/CD-Ausführung erzeugte Änderungen, Artefakte oder temporäre Dateien verbleiben potenziell auf dem System. Dies kann zu Seiteneffekten zwischen unterschiedlichen Job-Ausführungen führen.

In Anbetracht dessen ergibt sich die Notwendigkeit, eine Architektur zu entwickeln, die jede Job-Ausführung in einer definierten, reproduzierbaren Umgebung ermöglicht, Persistenz systematisch vermeidet und gleichzeitig den operativen Wartungsaufwand reduziert.

1.2. Zielsetzung

Die Zielsetzung dieser Arbeit besteht in der Konzeption und prototypischen Realisierung einer sicheren Deployment-Architektur für selbst gehostete Runner an der WHZ. Der Fokus liegt auf der Entwicklung einer Architektur zur ephemeren Bereitstellung selbst gehosteter Runner innerhalb der bestehenden VMware-Infrastruktur mit dem Ziel, Persistenz zwischen einzelnen Job-Ausführungen systematisch zu vermeiden.

1.3. Aufbau der Arbeit

Die vorliegende Arbeit folgt einem empirisch-technischen Vorgehen. In einem Grundlagenkapitel werden zunächst zentrale technische Konzepte und Begriffe erläutert, um ein gemeinsames Verständnis der zugrunde liegenden Systeme zu schaffen. Darauf aufbauend wird die bestehende Infrastruktur analysiert und es werden strukturierte Anforderungen an eine zukünftige Zielarchitektur abgeleitet. Anschließend wird der architektonische Lösungsraum unter Berücksichtigung der definierten Rahmenbedingungen untersucht und eingegrenzt. Auf dieser Basis erfolgt die Konzeption und prototypische Implementierung einer konkreten Zielarchitektur. Abschließend wird die entwickelte Lösung anhand der zuvor formulierten Anforderungen evaluiert und die Ergebnisse reflektiert.

1.4. Abgrenzung

Die vorliegende Arbeit fokussiert sich auf die Konzeption und prototypische Realisierung einer Deployment-Architektur für selbst gehostete Runner. Eine detaillierte Analyse produktspezifischer Implementierungsdetails der eingesetzten Virtualisierungsplattform VMware sowie eine vollständige Betrachtung der zugehörigen API-Spezifikationen sind nicht Gegenstand dieser Untersuchung.

Des Weiteren wurde keine Analyse der internen Implementierung von GitHub Actions vorgenommen, insbesondere nicht hinsichtlich der Workflow-Engine oder der vollständigen GitHub-API-Spezifikation. Für weiterführende technische Details wird auf die jeweilige Hersteller- und Produktdokumentation verwiesen [3, 4].

Während der Erstellung dieser Arbeit wurde am 5. Februar 2026 der GitHub Actions Runner Scale Set Client veröffentlicht. Dieser abstrahiert die Orchestrierung von Runnern, sodass für eine eigene Implementierung im Wesentlichen nur noch die Bereitstellungslogik der Ausführungsinstanz zu implementieren ist. Die Integration und Nutzung des Scale Set Client ist nicht Bestandteil dieser Arbeit. [5]

2. Grundlagen

Dieses Kapitel führt die für die Arbeit notwendigen terminologischen und technischen Grundlagen ein. Es behandelt die verwendete CI/CD-Plattform sowie zentrale Aspekte der Virtualisierung mit Schwerpunkt auf VMware.

2.1. GitHub Actions und Runner-Architektur

In diesem Abschnitt werden die grundlegenden Konzepte von GitHub Actions dargelegt und die Rolle von Runnern als Ausführungsinstanzen erörtert. Das Ziel besteht darin, ein technisches Verständnis der Workflow-Ausführung zu schaffen, das als Grundlage für die spätere Analyse und Architekturkonzeption dient.

2.1.1. GitHub Actions

GitHub Actions ist ein von GitHub bereitgestellter Dienst zur Automatisierung wiederkehrender Prozesse im Kontext von Software-Repositories. Dieser ermöglicht die Umsetzung von Abläufen zur kontinuierlichen Integration und Bereitstellung von Softwareänderungen.

Der Workflow (Arbeitsablauf) stellt die zentrale Komponente der Automatisierung dar. Es handelt sich um einen konfigurierbaren, automatisierten Prozess, der auf spezifische Trigger (Auslöser) reagiert. Ein Repository kann mehrere Workflows umfassen, um unterschiedliche Aufgaben getrennt voneinander abzubilden. Workflows werden als YAML-Dateien im Repository definiert. Typische Trigger sind Push- und Pull-Request-Ereignisse, manuelle Ausführungen oder zeitgesteuerte Läufe. [6]

Ein Workflow setzt sich aus einem oder mehreren Jobs (Aufträgen) zusammen. Jeder Job wird durch eine Abfolge von Steps (Schritten) definiert, die sequenziell ausgeführt werden. Steps können Shell-Kommandos ausführen oder vordefinierte Actions (Aktionen) nutzen. Actions sind wiederverwendbare Bausteine zur Umsetzung typischer Aufgaben, beispielsweise zum Pullen eines Repositorys. Abhängig vom Workflow-Design können Jobs parallel ausgeführt oder über Abhängigkeiten in eine definierte Ausführungsreihenfolge gebracht werden. [7]

2.1.2. Runner-Architektur

Runner sind Ausführungsinstanzen, auf denen die Jobs eines Workflows verarbeitet werden. GitHub verwendet den Begriff dabei häufig im Sinne der Runner-Maschine. Die technische Realisierung erfolgt durch eine Runner-Anwendung, die auf einem Hostsystem betrieben wird und die Laufzeitumgebung für die Abarbeitung der Steps bereitstellt. Als Hostsysteme kommen insbesondere physische Server oder virtuelle Maschinen in Betracht. [8, 9]

Damit ein Runner Jobs annehmen kann, wird dieser bei GitHub registriert. Nach erfolgreicher Registrierung wartet die Runner-Anwendung auf zugewiesene Jobs, die durch GitHub geplant und an den Runner übergeben werden. Die Ausführung erfolgt in der lokalen Arbeitsumgebung des Runners. Das weitere Verhalten nach Abschluss eines Jobs hängt vom Betriebsmodell der Runner-Instanz ab. [9]

Persistente Runner bleiben registriert und können nacheinander weitere Jobs ausführen. Dabei kann der lokal erzeugte Zustand in Form von temporären Dateien, Artefakten oder Caches ohne zusätzliche Bereinigungs- oder Rücksetzmechanismen zwischen den Job-Ausführungen bestehen bleiben. Ephemere Runner hingegen sind so konfiguriert, dass sie genau einen Job verarbeiten und anschließend automatisch deregistriert werden. [9, 10]

2.1.3. GitHub-gehostete und selbst gehostete Runner

Für die Ausführung von GitHub-Workflows werden Runner als Ausführungsinstanz benötigt. Dabei differenziert GitHub zwischen von GitHub-gehosteten Runnern und von Anwendern selbst gehosteten Runnern.

GitHub-gehostete Runner werden in der GitHub-Infrastruktur bereitgestellt und zentral verwaltet. Die Ausführungsinstanz beinhalten gängige Werkzeuge und Laufzeitkomponenten und werden durch GitHub gewartet. Diese Runner werden ephemer auf virtuellen Maschinen oder Containern betrieben. Es ist festzustellen, dass container-betriebene Runner nicht in der Lage sind, selbst container-spezifische Anwendungen wie Docker auszuführen. [8]

Demgegenüber werden selbst gehostete Runner auf eigener Infrastruktur betrieben und bei GitHub registriert, sodass sie anschließend Jobs empfangen und ausführen können. Die Kontrolle und Verantwortung für das Betriebssystem, die installierte Software, den Netzwerkzugriff sowie Wartung und Verfügbarkeit obliegen in diesem Fall vollständig dem Anwender. [10]

In Konstellationen mit einem GitHub Enterprise Server werden GitHub-gehostete Runner nicht zur Verfügung gestellt. Die Ausführung von Jobs erfolgt in diesem Fall über selbst

gehostete Runner, die gegen die jeweilige Enterprise-Instanz registriert werden. [11]

2.2. Terminologische Grundlagen

Für den Rahmen dieser Arbeit werden im Folgenden Isolation, Persistenz, Ephemerität, Reproduzierbarkeit sowie das zugrunde liegende Sicherheitsverständnis definiert. Die Definitionen dienen als gemeinsame Grundlage für die Analyse, die Konzeption und die Evaluation.

2.2.1. Artefakt und Persistenz

Unter Artefakt wird im Kontext dieser Arbeit ein Erzeugnis verstanden, das während einer Job-Ausführung auf dem Runner-Host entsteht, etwa Dateien im Arbeitsverzeichnis, Cache-Inhalte oder Konfigurations- und Installationszustände. Diese Verwendung ist von „Artifacts“ als GitHub-Funktion zum expliziten Hochladen von Ergebnissen abzugrenzen.

Persistenz beschreibt die Eigenschaft, dass Artefakte über das Ende einer Job-Ausführung hinaus erhalten bleiben und in nachfolgenden Ausführungen vorhanden sein können.

2.2.2. Isolation

Unter Isolation wird im Kontext dieser Arbeit die Trennung von Job-Ausführungen verstanden, sodass sich Jobs nicht durch Artefakte in der Ausführungsinstanz gegenseitig beeinflussen.

2.2.3. Ephemerität

Unter Ephemerität wird im Kontext dieser Arbeit ein Betriebsmodell verstanden, bei dem Ausführungsinstanzen nur temporär existieren. Es trifft zunächst keine Aussage darüber, ob das zugrunde liegende Hostsystem nach der Ausführung zurückgesetzt oder neu bereitgestellt wird.

2.2.4. Reproduzierbarkeit

Unter Reproduzierbarkeit wird im Kontext dieser Arbeit die Eigenschaft verstanden, dass Job-Ausführungen bei gleichen Eingaben und Randbedingungen in einem definierten Ausgangszustand zu konsistenten Ergebnissen führen.

2.2.5. Sicherheitsverständnis

Unter Sicherheitsverständnis wird im Kontext dieser Arbeit die Isolation zwischen Job-Ausführungen sowie die Vermeidung unerwünschter Persistenz verstanden. Im Mittelpunkt stehen damit saubere Startbedingungen und die Verhinderung von Artefaktübertragungen zwischen unterschiedlichen Ausführungen.

Nicht Gegenstand der Arbeit ist eine umfassende Betrachtung komplexer Angriffsmodelle, die Analyse von Workflow-Härtungsmaßnahmen, Penetrationstests oder die sicherheitstechnische Bewertung der GitHub-Plattform selbst.

2.3. Virtualisierung als Ausführungsinstanz

Virtualisierung bildet die Basis für isolierte Ausführungsinstanz. Zunächst werden die Grundprinzipien der Virtualisierung dargestellt. Anschließend werden die für diese Arbeit relevanten Konzepte von VMware erläutert, da diese Arbeit auf eine VMware-basierten Infrastruktur aufsetzt.

Vergleichbare Konzepte existieren auch in anderen Virtualisierungsumgebungen, teilweise unter abweichender Terminologie [12, 13].

2.3.1. Grundprinzip der Virtualisierung

Virtualisierung bezeichnet die Abstraktion physischer Hardware, sodass mehrere voneinander getrennte Systeminstanzen auf einem gemeinsamen physischen Host betrieben werden können. Die Bereitstellung dieser Instanzen erfolgt durch einen Virtual Machine Monitor (VMM), der die Ausführung Virtuelle Maschine (VM) überwacht und den Zugriff auf Hardware-Ressourcen vermittelt. [14]

In der Praxis wird ein VMM häufig auch als „Hypervisor“ bezeichnet. [15]

Virtuelle Maschine werden durch einen VMM so bereitgestellt, dass die Interaktionen der jeweiligen VMs mit der zugrunde liegenden Hardware vermittelt und kontrolliert werden. Dadurch kann eine starke Isolation zwischen VMs erreicht werden, sodass Fehlerzustände oder sicherheitsrelevante Vorfälle innerhalb einer VM auf diese Instanz begrenzt bleiben. [16]

Im Rahmen dieser Arbeit wird Isolation als Trennung von Job-Ausführungen verstanden, sodass sich Jobs nicht druch Artefakten in der Ausführungsinstanz gegenseitig beeinflussen (vgl. Abschnitt 2.2.2). Die Ausführung von Jobs auf getrennten VMs entspricht dabei der Isolation über separate VMs und nutzt die durch den VMM bereitgestellte Systemisolation als technische Grundlage.

Virtuelle Maschinen besitzen einen klaren Lebenszyklus und können erzeugt, gestartet, gestoppt und wieder entfernt werden. Diese Eigenschaften ermöglichen es, Ausführungsinstanz kontrolliert bereitzustellen und diese bei Bedarf auszutauschen oder neu aufzusetzen, ohne dass die zugrunde liegende physische Hardware verändert werden muss. [17, 18]

2.3.2. Template

Ein *Template* (auch als „Golden Image“ bezeichnet) beschreibt einen definierten Referenzzustand einer virtuellen Maschine. Es umfasst typischerweise das installierte Betriebssystem, grundlegende Konfigurationen sowie vorinstallierte Software und dient als Grundlage, um wiederholt konsistente VM-Instanzen bereitzustellen. [19]

In VMware entsteht ein Template durch die Konvertierung einer bestehenden virtuellen Maschine. Es entspricht damit einem VM-Objekt, das nicht mehr direkt gestartet werden kann, sondern ausschließlich zur Instanziierung neuer virtueller Maschinen verwendet wird. [20]

2.3.3. Snapshot

Ein *Snapshot* speichert den Zustand einer virtuellen Maschine zu einem bestimmten Zeitpunkt. Dabei wird der Zustand der virtuellen Festplatte, der Konfiguration und des Arbeitsspeicher erfasst. Snapshots werden genutzt, um definierte Referenzpunkte zu schaffen, auf die eine VM bei Bedarf zurückgeführt werden kann. [21]

2.3.4. Full Clone

Auf Basis eines Templates oder eines Snapshots können neue virtuelle Maschinen durch Klonen erzeugt werden. Ein *Full Clone* stellt dabei eine vollständige, unabhängige Kopie einer virtuellen Maschine dar. Die geklonte VM besitzt eine eigene Festplattenkopie und ist damit nicht von der Ursprungsinstanz abhängig, wodurch eine isolierte Weiterverwendung ermöglicht wird. [22]

2.3.5. Linked Clone

Linked Clones werden als Ableitungen aus einem Snapshot erstellt. Dabei wird die Festplatte des Snapshots als Grundlage verwendet, während Schreiboperationen in separaten Festplatten (Delta Disks) gespeichert werden. Dies reduziert den Speicherbedarf gegenüber vollständigen Klonen, führt jedoch zu einer dauerhaften Abhängigkeit von der

zugrunde liegenden Festplatte des referenzierten Snapshot. Wird der Referenzzustand entfernt oder unzugänglich, kann der Linked Clone nicht mehr betrieben werden. [22]

2.3.6. Instant Clone

Ein *Instant Clone* wird auf Basis einer laufenden oder eingefrorenen VM (Golden VM) erstellt. Dabei wird wie bei Linked Clones die Festplatte des Referenzzustand als Grundlage benutzt und Änderungen in einer Delta Disk gespeichert. Zusätzlich wird auch der Arbeitsspeicher des Referenzzustandes als Grundlage genutzt und Änderungen daran als Delta gespeichert. Dadurch, dass der *Instant Clone* auf Basis einer laufenden VM erstellt wird und dessen Arbeitsspeicher nutzt, benötigt der *Instant Clone* keinen Bootvorgang. [22]

Es ist zu beachten, dass nicht nur der jeweilige *Instant Clone* eine eigene Delta Disk erhält, sondern auch die Referenz-VM beim Klonvorgang mit einer Delta Disk versehen wird. Bei jedem weiteren Erstellen von *Instant Clones* werden zusätzliche Delta Disks an diese bereits vorhandenen angehängt. Dadurch kann eine zunehmend tiefe, verkettete Struktur aus Delta Disks entstehen, was sich bei vielen Klonen negativ auf Verwaltung und Performance auswirken kann. [23]

Dieses Problem lässt sich vermeiden, indem die Referenz-VM eingefroren wird. In diesem Zustand bleibt die initial erzeugte Delta Disk der Referenz-VM unverändert, sodass bei weiteren Klonvorgängen keine neuen Delta Disks für die Referenz-VM mehr angelegt werden müssen. Zum Aufheben des eingefrorenen Zustands ist ein Neustart der VM erforderlich. [23]

2.4. Ansible

Ansible ist ein Automatisierungswerkzeug zur Konfiguration, Bereitstellung und Orchestrierung von IT-Systemen [24]. Die Ausführung erfolgt dabei von einem *Control Node* aus, auf dem Ansible installiert ist. Von diesem System aus werden die zu verwaltenden Zielsysteme, die sogenannten *Managed Nodes*, adressiert und automatisiert gesteuert [25].

Die Menge der durch Ansible verwalteten Zielsysteme wird in einem `Inventory` beschrieben. Ein Inventory umfasst `Hosts`, also konkrete Server oder virtuelle Maschinen, die von Ansible angesprochen werden, und kann diese zusätzlich in Gruppen zusammenfassen. Darüber hinaus besteht die Möglichkeit, im Inventory sowie in zugehörigen Variablendateien host- oder gruppenspezifische Parameter zu hinterlegen. [26]

Die Installation von Ansible auf den Managed Nodes ist nicht erforderlich. Für eine Vielzahl von Standard-Modulen wird jedoch eine Python-Laufzeitumgebung vorausgesetzt, da Ansible eigens generierten Python-Code auf dem Zielsystem ausführt. [27]

Die eigentlichen Automatisierungsabläufe werden in *Playbooks* beschrieben. Playbooks sind YAML-basierte Dateien, die einen oder mehrere *Plays* enthalten. Ein Play ordnet eine Folge von Aufgaben bestimmten Hosts oder Host-Gruppen aus dem Inventory zu. Die kleinste Ausführungseinheit ist dabei eine *Task*, die jeweils genau eine Operation beschreibt, etwa das Ausführen eines Befehls, das Kopieren einer Datei oder einen API-Aufruf. [28]

3. Analyse der bestehenden Infrastruktur

In diesem Kapitel wird die aktuell eingesetzte Infrastruktur zum Betrieb selbst gehosteter GitHub-Runner an der WHZ untersucht. Das Ziel besteht in der Darstellung des technischen und organisatorischen Ist-Zustands. Auf dieser Grundlage erfolgt die Identifizierung von sicherheits- und betriebsrelevanten Risiken, insbesondere hinsichtlich Persistenz und Isolation. Die gewonnenen Erkenntnisse dienen anschließend als Basis für die Ableitung von Anforderungen an eine zukünftige Zielarchitektur.

3.1. Aktuelle Runner-Architektur an der WHZ

Der Betrieb der selbst gehosteten Runner an der WHZ erfolgt auf einer virtualisierten Infrastruktur auf Basis von VMware. Die Runner werden als virtuelle Maschinen bereitgestellt, die jeweils mehrere Runner-Instanzen parallel hosten.

Die Infrastruktur besteht aus acht virtuellen Maschinen, auf denen jeweils acht Runner-Instanzen konfiguriert sind. Demzufolge können bis zu 64 Runner gleichzeitig aktiv sein. Innerhalb dieser 64 Runner sind neben den regulären Ausführungsinstanzen auch spezialisierte Runner für automatisierte Abhängigkeits- und Sicherheitsprüfungen vorgesehen, die im GitHub-Kontext als sogenannte „Dependabot“ bezeichnet werden [29].

Zur organisatorischen und technischen Strukturierung der Nutzung existieren vier logisch getrennte Runner-Gruppen: *Dependabot Public*, *Dependabot Private*, *General Public* und *General Private*. Die Gruppen *Dependabot Public* und *General Public* sind für alle nutzbar. Die Gruppen *Dependabot Private* und *General Private* sind ausschließlich für das ZKI vorgesehen. Die Trennung erfolgt sowohl auf Ebene der virtuellen Maschinen als auch auf GitHub-Seite durch eine getrennte Zuweisung der Runner-Gruppen zu den jeweiligen Repositories und Organisationen.

Innerhalb der virtuellen Maschinen wird jede Runner-Instanz unter einem eigenen Systembenutzer betrieben. Diese Benutzer verfügen über keine Root-Rechte und besitzen jeweils ein separates Arbeitsverzeichnis. Zudem hat jeder Systembenutzer eine eigene Docker-Rootless-Instanz. Docker findet dabei unter anderem Anwendung bei der Ausführung containerisierter Build- oder Testumgebungen.

Die Runner werden persistent betrieben und verbleiben dauerhaft im System. Eine automatisierte vollständige Rücksetzung der virtuellen Maschinen oder der Runner-Instanzen nach Abschluss eines Jobs ist nicht implementiert.

Die virtuellen Maschinen werden in wöchentlichen Abständen im Rahmen geplanter Wartungsarbeiten aktualisiert.

3.2. Analyse des Ist-Zustandes

Auf Basis des in Abschnitt 3.1 dargestellten Ist-Zustandes werden im Folgenden sicherheits- und betriebsrelevante Risiken der aktuell selbst betriebenen Infrastruktur analysiert.

3.2.1. Persistenz zwischen Jobs

Die Ausführung der Runner-Instanzen erfolgt auf persistent betriebenen virtuellen Maschinen. Eine automatisierte Rücksetzung der virtuellen Maschinen oder der Runner-Umgebung nach Abschluss eines Jobs ist nicht implementiert. Infolgedessen können Artefakte, die im Rahmen einer Workflow-Ausführung entstehen, über mehrere Ausführungen hinweg bestehen bleiben. Diese persistenten Artefakte können eine Reproduzierbarkeit von Build- und Testergebnissen beeinträchtigen, sofern nachfolgende Jobs unbeabsichtigt von Artefakten vorheriger Jobs beeinflusst werden.

Darüber hinaus entsteht dadurch ein sicherheitsrelevantes Risiko, da verbleibende Artefakte von nachfolgenden Jobs eingesehen und missbraucht werden könnten. Dies betrifft insbesondere sicherheitskritische Informationen wie Zugangsdaten, Tokens oder Passwörter, die beispielsweise in Dateien, Umgebungsvariablen oder Log-Ausgaben zwischengespeichert werden und somit potenziell ausgespäht werden können.

3.2.2. Eingeschränkte Isolation zwischen Job-Ausführungen

Die aktuelle Infrastruktur zeichnet sich durch den Einsatz mehrerer Runner-Instanzen in paralleler Ausführung auf einer einzigen virtuellen Maschine aus. Obwohl jede Runner-Instanz unter einem eigenen Systembenutzer ohne Root-Rechte ausgeführt wird und über ein separates Arbeitsverzeichnis verfügt, wird die gemeinsame Systemumgebung von den Runner-Instanzen geteilt. Insbesondere gemeinsam genutzte Ressourcen wie CPU, Arbeitsspeicher, Festplattenspeicher und Netzwerk können zu Wechselwirkungen zwischen parallelen Job-Ausführungen führen.

Unter der Prämisse, dass innerhalb der virtuellen Maschine Docker eingesetzt wird, kann Persistenz auch durch containerbezogene Artefakte entstehen. Jeder Runner nutzt

zwar eine eigene rootless Docker-Instanz und ist damit gegenüber anderen Runnern isoliert, innerhalb derselben Instanz können jedoch Images, Layer und Build-Caches zwischen Jobs erhalten bleiben. Dies führt zu einer zunehmenden Abweichung vom definierten Ausgangszustand der Ausführungsinstanz.

Die Aufteilung der Runner-Infrastruktur in separate Runner-Gruppen für die allgemeine Nutzung und Mitarbeitenden des ZKI stellt eine technische Trennung auf Gruppenebene dar. Persistente Zustände werden nicht zwischen den Runner-Gruppen übertragen, sodass eine infrastrukturelle Abgrenzung der unterschiedlichen Nutzungskontexte gegeben ist. Innerhalb der jeweiligen Runner-Grupppe bleibt jedoch das grundlegende Problem der Persitenz bestehen. Die bestehende Isolation manifestiert sich demnach als gruppenbezogen, jedoch nicht ausführungsspezifisch. Eine technisch durchgesetzte Trennung einzelner Job-Kontexte wird dadurch nicht erreicht.

3.2.3. Ressourcen- und Skalierungsgrenzen des aktuellen Modells

Die aktuelle Infrastruktur betreibt mehrere Runner-Instanzen parallel auf einer virtuellen Maschine. Damit wird eine Reduktion der Anzahl der benötigten virtuellen Maschinen erreicht und somit auch eine Reduktion des unmittelbaren Ressourcenbedarfs pro Runner-Instanz. Gleichzeitig resultiert die Bündelung mehrerer Runner auf einem Host in einer festen Kopplung zwischen Isolationsgrad und Ressourcenstruktur der Umgebung.

Eine vollständige Entkopplung einzelner Runner durch dedizierte virtuelle Maschinen würde die Anzahl gleichzeitig betriebener VMs signifikant erhöhen. Dies hätte unmittelbare Auswirkungen auf den Speicherbedarf[1], die Verwaltung von Systemabbildern sowie den operativen Wartungsaufwand. Der aktuelle Aufbau verdeutlicht somit einen strukturellen Zielkonflikt zwischen infrastruktureller Ressourceneffizienz und Isolation.

3.3. Abgeleitete Anforderungen

Die nachfolgenden Anforderungen beschreiben den angestrebten Soll-Zustand einer zukünftigen Architektur für den Betrieb selbst gehosteter Runner. Sie werden aus den in Abschnitt 3.2 identifizierten sicherheits- und betriebsrelevanten Risiken sowie aus den organisatorischen, technischen und infrastrukturellen Rahmenbedingungen abgeleitet.

Ziel ist es, die im Ist-Zustand erkannten strukturellen Schwächen in konkrete und überprüfbare Kriterien zu überführen. Dadurch soll eine reproduzierbare, isolierte und administrativ beherrschbare Ausführungsinstanz ermöglicht werden. Die Anforderungen bilden

[1] Beispielrechnung: In der aktuellen Konfiguration werden 64 VMs mit jeweils 20 GB Speicherplatz betrieben, was einem Gesamtbedarf von 64 $\times$ 20 GB = 1280 GB ergibt.

die normative Grundlage für die Konzeption der Zielarchitektur, deren prototypische Implementierung sowie die anschließende Evaluation. Zur Nachvollziehbarkeit werden sie thematisch kategorisiert und fortlaufend nummeriert.

3.3.1. Ableitung aus der Ist-Analyse

Die Anforderungen adressieren die folgenden Problemfelder des Ist-Zustandes:

- **Persistenz zwischen Jobs:** Da keine Rücksetzung derzeit erfolgt, werden Anforderungen zur jobbezogenen Rückführung in einen definierten Ausgangszustand abgeleitet.
- **Eingeschränkte Isolation durch gemeinsame VM-Nutzung:** Aus der parallelen Ausführung mehrerer Runner auf einer VM ergeben sich Anforderungen zur technischen Durchsetzung von Isolation und zur Vermeidung wechselseitiger Beeinflussung.
- **Containerbezogene Persistenz:** Aus dem Einsatz von Docker und der Möglichkeit verbleibender Images, Layer und Caches werden Anforderungen zur Begrenzung containerbezogener Zustände abgeleitet.

3.3.2. Funktionale Anforderungen

F1: Jeder CI/CD-Job muss in einer definierten und reproduzierbaren Ausgangsumgebung starten.

F2: Nach Abschluss eines Jobs muss die Runner-Instanz automatisiert in den definierten Ausgangszustand zurückgeführt oder vollständig neu bereitgestellt werden.

F3: Die Bereitstellung und Rückführung von Runner-Instanzen muss automatisiert erfolgen.

F4: Der Runner-Lifecycle (Bereitstellung, Registrierung, Ausführung, Bereinigung sowie Deregistrierung und Rückbau) muss ohne manuellen Eingriff ausführbar sein.

F5: Die Runner-Architektur muss containerbasierte CI/CD-Jobs unterstützen (Docker-basierte Workflows).

3.3.3. Anforderungen an Isolation und Trennung

I1: Job-Ausführungen unterschiedlicher Repositories oder Nutzerkontexte dürfen sich nicht gegenseitig beeinflussen.

I2: Zwischen zwei Job-Ausführungen dürfen keine Artefakte oder Zustände aus vorherigen Ausführungen nutzbar oder einsehbar sein.

I3: Container-bezogene Zustände (z. B. Images, Layer, Build-Caches, Container und Volumes) dürfen zwischen zwei Job-Ausführungen nicht persistent erhalten bleiben.

3.3.4. Sicherheits- und Systemanforderungen

S1: Runner-Instanzen dürfen keinen privilegierten Zugriff auf das zugrunde liegende Host-System besitzen.

S2: Der Zustand der Runner-Instanz muss vor und nach jeder Job-Ausführung eindeutig definiert sein.

S3: Systemweite Änderungen innerhalb der Runner-Instanz, die durch Jobs verursacht werden, dürfen nach der Rückführung nicht bestehen bleiben.

3.3.5. Anforderungen an Wartbarkeit und Betrieb

W1: Die Runner-Umgebung muss zentral wartbar sein, sodass Aktualisierungen reproduzierbar ausgerollt werden können.

W2: Bereitstellung und Neuinitialisierung der Runner-Instanzen müssen ohne manuelle VM-Konfiguration erfolgen und deklarativ über Ansible abbildbar sein.

W3: Lösungs muss sowohl für Linux aus auch Windows möglich sein.

3.3.6. Rahmenbedingungen und Constraints

R1: Die Lösung muss innerhalb der bestehenden VMware-basierten Virtualisierungsinfrastruktur umgesetzt werden.

R2: Die Orchestrierung und Steuerung der virtuellen Maschinen muss mit Ansible umgesetzt werden.

C1: Pro virtueller Maschine darf maximal eine Runner-Instanz betrieben werden.

4. Architektonische Lösungsansätze

In diesem Kapitel wird der Lösungsraum für den Betrieb ephemerer, VM-basierter Runner unter den in Abschnitt 3.3 abgeleiteten Anforderungen und Rahmenbedingungen dargestellt. Das Ziel besteht darin, relevante Architekturansätze zu systematisieren und ihre grundlegenden Eigenschaften hinsichtlich Zustandsrücksetzung, Bereitstellung und Betriebscharakteristik gegenüberzustellen.

4.1. Referenzansatz: GitHub Action Runner Controller

GitHub stellt mit dem *Action Runner Controller (ARC)* eine Lösung zur Orchestrierung selbst gehosteter Runner in Kubernetes-Umgebungen bereit. Das Ziel des ARC besteht darin, die Bereitstellung, Skalierung und Bereinigung von Runner-Instanzen zu automatisieren und somit einen weitgehend wartungsarmen Betrieb ephemeren Ausführungsinstanzen zu ermöglichen. [30]

Die konzeptionelle Grundlage von ARC bildet das Controller-Prinzip, bei dem ein gewünschter Zielzustand (beispielsweise eine bestimmte Anzahl verfügbarer Runner) beschrieben wird und der Controller kontinuierlich dafür Sorge trägt, dass dieser Zustand durch Erzeugen, Konfigurieren und Entfernen von Runner-Instanzen erreicht wird. [30]

In der vorliegenden Arbeit fungiert ARC als konzeptionelles Referenzmuster für ein automatisiertes Runner-Lifecycle-Management, wobei insbesondere die Trennung von Orchestrierungslogik (Controller) und Ausführungsinstanz des Runners von Relevanz ist. Eine direkte Übernahme des ARC ist jedoch nicht möglich, da der ARC Kubernetes als Ausführungs- und Orchestrierungsplattform voraussetzt [30]. In der Folge wird ein VMware-basierter Lösungsraum untersucht, wobei das Controller-Prinzip als architektonische Leitidee beibehalten wird.

4.2. Persistente Runner mit Cleanup-Mechanismen

Ein naheliegender Ansatz zum Betrieb selbst gehosteter Runner besteht darin, Runner-Instanzen nach Abschluss eines Jobs durch gezielte Bereinigungsmechanismen in einen

definierten Zustand zurückzuführen.

Die als *Cleanup* bezeichneten Maßnahmen umfassen die Bereinigung des Arbeitsverzeichnisses des Runners, das Entfernen temporärer Dateien sowie die Rücksetzung oder Löschung von Cache-Inhalten. Im Falle der Nutzung von Workflows, die auf containerbasierten Build- oder Testumgebungen basieren, umfasst dies zudem die Bereinigung von Docker-Ressourcen. Dies beinhaltet die Entfernung von Containern, Images, Volumes oder Netzwerken. Zusätzlich besteht die Möglichkeit, restriktive Berechtigungskonzepte zu implementieren, um Schreibzugriffe auf definierte Verzeichnisse zu limitieren und somit die Anzahl potenzieller Persistenzpfade zu minimieren.

Die vollständige und zuverlässige Umsetzung eines Cleanup-Ansatzes ist jedoch in heterogenen CI/CD-Szenarien nur eingeschränkt gewährleistbar. Die konkrete Lokalisation sowie die Form jobbezogener Artefakte sind in hohem Maße abhängig von den eingesetzten Werkzeugen, Programmiersprachen und Build-Prozessen. Darüber hinaus können Artefakte außerhalb der unmittelbaren Arbeitsverzeichnisse entstehen, beispielsweise durch global installierte Abhängigkeiten, systemweite Konfigurationen oder von Laufzeitumgebungen angelegte Caches.

Der Cleanup-basierte Betrieb stellt demnach einen Ansatz dar, der Persistenzrisiken reduzieren kann, jedoch keine inhärente Garantie für eine vollständig reproduzierbare Ausgangsumgebung pro Job bietet.

4.3. Rücksetzung bestehender virtueller Maschinen

Eine zweite Ansatzklasse besteht in der Rücksetzung bestehender virtueller Maschinen auf einen definierten Ausgangszustand. Das Ziel besteht darin, die Persistenz zwischen Job-Ausführungen systematisch zu vermeiden, ohne für jede Ausführung eine vollständig neue VM instanziieren zu müssen.

4.3.1. Snapshot-basierte Rücksetzung

Im Rahmen einer snapshot-basierten Rücksetzung wird ein definierter Systemzustand einer VM als Snapshot erfasst und nach Abschluss einer Job-Ausführung wiederhergestellt. Der Snapshot fungiert als Referenzpunkt, auf den die VM mittels der Revert-Operation zurückgeführt wird. Hierdurch wird der Dateisystem- und Konfigurationszustand der VM auf einen reproduzierbaren Ausgangszustand zurückgesetzt, sodass nachfolgende Job-Ausführungen unter den gleichen Startbedingungen beginnen können.

Für den Betrieb bedeutet dies, dass eine Runner-VM dauerhaft existieren kann, jedoch

nach jeder Nutzung in einen definierten Referenzzustand zurückgesetzt wird und anschließend wieder bei GitHub angemeldet wird. Das Verfahren reduziert somit das Risiko, dass Artefakte über mehrere Job-Ausführungen hinweg bestehen bleiben. Gleichzeitig erfordert der Ansatz ein konsistentes Snapshot-Management, da der Referenzsnapshot als Ausgangsbasis dauerhaft verfügbar sein muss.

Die Realisierung von Änderungen am gewünschten Basiszustand für System- und Sicherheitsupdates erfolgt mittels des Einspielens der Updates in der VM und dem anschließenden Erstellen eines neuen Referenzsnapshots. Die snapshot-basierte Rücksetzung fungiert folglich als Reset-Mechanismus, der die Ephemerität auf Zustandsebene abbildet, ohne dass eine Ersetzung der VM-Instanz erforderlich ist.

4.3.2. Independent Non-Persistent Disk

Eine weitere Variante zur Rücksetzung bestehender VMs kann durch den Einsatz von VMwares Festplattenmodus *Independent Non-Persistent Disk* realisiert werden. Dabei werden Schreibzugriffe der VM zwar während des Betriebs zugelassen, die resultierenden Änderungen werden jedoch nicht dauerhaft in den zugrunde liegenden virtuellen Datenträger übernommen. Stattdessen werden Modifikationen nur temporär vorgehalten und beim Neustarten der VM verworfen. [31]

Die Aktivierung des Festplattenmodus ist ausschließlich unter der Voraussetzung möglich, dass keine vorhandenen Snapshots für die virtuelle Maschine vorliegen. Nach erfolgter Einstellung des Festplattenmodus besteht keine Möglichkeit mehr zur Erstellung von Snapshots. Die vorliegende Mechanik findet ausschließlich Anwendung auf vollwertige virtuelle Maschinen (Full Clones). Linked oder Instant Clones sind von dieser Mechanik ausgenommen. [31]

Für den Einsatz als ephemere Ausführungsinstanz bedeutet dies, dass der Zustand der VM auf Dateisystemebene nach Abschluss einer Job-Ausführung durch einen kontrollierten Neustart in einen definierten Ausgangszustand zurückgeführt werden kann. Der Reset-Mechanismus, der durch den Independent Non-Persistent Disk bereitgestellt wird, modelliert die Ephemerität durch Zustandsverwerfung beim Neustart.

Die Realisierung von Änderungen am gewünschten Basiszustand für Updates erfolgt durch Abschalten der VM und Umstellung des Festplattenmodus auf den regulären *Dependent Disk*-Modus. Nach dem Starten der VM besteht nun wieder die Möglichkeit, persistente Änderungen vorzunehmen. Abschließend ist die VM wieder abzuschalten und der Festplattenmodus wieder auf *Independent Non-Persistent Disk* zu setzen.

Da dieser Festplattenmodus nur für vollwertige VMs möglich ist, besteht die Notwendigkeit, entweder jede Maschine einzeln zu aktualisieren oder ein aktualisiertes Template zu verwenden und alle VMs zu löschen und neu bereitzustellen.

4.4. Neuinstanziierung virtueller Maschinen

Als dritte Ansatzklasse wird die Neuinstanziierung virtueller Maschinen betrachtet. Im Gegensatz zur Rücksetzung einer bestehenden VM wird hierbei nach Abschluss einer Job-Ausführung die verwendete Instanz verworfen und für nachfolgende Jobs eine neue VM aus einem definierten Referenzzustand bereitgestellt. Der Referenzzustand wird durch ein Template oder einen Snapshot beschrieben und gewährleistet, dass jede neue Instanz unter vergleichbaren Startbedingungen erzeugt wird.

Die Neuinstanziierung ermöglicht eine klare Trennung zwischen Ausführungsinstanzen, da sich einzelne Jobs nicht denselben VM-Lebenszyklus teilen. Die Vermeidung von Persistenz kann durch die vollständige Entfernung der verwendeten Instanz nach Abschluss des Jobs erfolgen. Die praktische Anwendbarkeit dieses Ansatzes ist dabei in hohem Maße von der Bereitstellungszeit, dem Ressourcenbedarf sowie den Eigenschaften des jeweiligen Klonmechanismus abhängig. Im Folgenden werden Full Clones, Linked Clones und Instant Clones als zentrale Varianten der Neuinstanziierung eingeordnet.

4.4.1. Full Clone

Im Rahmen der Neuinstanziierung eines *Full Clones* erfolgt die Erstellung einer Kopie des vollständigen virtuellen Datenträgers. Dies resultiert in einer hohen I/O-Anforderung und einem signifikanten Verschleiß der physischen Festplatte. Die Bereitstellungszeit ist folglich abhängig vom Kopierumfang des Referenzzustandes.

Ein weiterer Aspekt, der in diesem Zusammenhang zu berücksichtigen ist, ist die signifikant hohe Speicheranforderung während des Betriebs paralleler VM-Instanzen (vgl. 3.2.3). Der Grund hierfür ist, dass jede VM eine separate Kopie der Festplatte erfordert.

Die Realisierung von Änderungen am gewünschten Basiszustand für Updates erfolgt mittels der Erstellung eines aktualisierten Templates oder aktualisierten Snapshots.

4.4.2. Linked Clone

Der Referenzzustand eines *Linked Clones* wird auf Basis eines Snapshots definiert, wobei ausschließlich Änderungen auf eine Deltadisk gespeichert werden. Infolgedessen ist

bei der Neuinstanziierung keine Kopie der Basisdisk erforderlich, was eine hohe I/O-Anforderung, wie sie bei Full Clones auftritt, überflüssig macht.

Die Realisierung von Änderungen am gewünschten Basiszustand für Updates erfolgt mittels eines aktualisierten Snapshots.

4.4.3. Instant Clone

Die Bereitstellung von *Instant Clones* erfolgt auf Basis einer dauerhaft verfügbaren laufenden VM, was eine schnelle Instanziierung ermöglicht, da kein Bootvorgang benötigt wird.

Für ephemere Ausführungsinstanzen kann dies vorteilhaft sein, da Instanzen kurzfristig erzeugt und nach Abschluss einer Job-Ausführung verworfen werden können. Gleichzeitig ist dieser Ansatz an die Verfügbarkeit einer laufenden VM gebunden.

Die Realisierung von Änderungen am gewünschten Basiszustand für Updates erfolgt mittels der Aktualisierung der laufenden Golden VM.

Um die Bildung einer Delta-Chain durch das Erstellen von Instant Clones zu unterbinden, ist es erforderlich, dass die Golden VM eingefroren ist. In der Konsequenz dessen ist es erforderlich, dass vor dem Update die VM neu startet und anschließend erneut eingefroren wird.

5. Konzeption der Zielarchitektur

In diesem Kapitel erfolgt auf Basis des in Kapitel 4 dargestellten Lösungsraums die Selektion des Bereitstellungs- und Rücksetzansatzes. Darauf aufbauend wird die daraus resultierende Zielarchitektur beschrieben, einschließlich der Deployment- und Update-Abläufe sowie des Lifecycles einer einzelnen Runner-Instanz.

5.1. Auswahl des Bereitstellungs-Ansatzes

In diesen Abschnitt werden geeignete Kandidaten für eine prototypische Umsetzung eingegrenzt. Im Anschluss erfolgt eine empirische Evaluation anhand definierter Messpunkte, mit dem Ziel, eine fundierte Auswahlentscheidung für den Bereitstellungs-Ansatz zu treffen.

5.1.1. Vorauswahl

Der Cleanup-Mechanismus wird ausgeschlossen, da eine vollständige und verlässliche Entfernung von Artefakten in der Praxis nur eingeschränkt gewährleistet werden kann (vgl. Abschnitt 4.2). Snapshot-basierte Rücksetzungen werden verworfen, da für den betrachteten Betrieb pro Runner eine dedizierte VM erforderlich ist und zusätzlich Snapshot-Kopien Speicher- und Verwaltungsaufwand verursachen. Die Neuinitialisierung über Full Clones kann ebenfalls ausgeschlossen werden, da das wiederholte Kopieren vollständiger VM-Vorlagen hohe I/O-Anforderungen erzeugt und die Bereitstellungsdauer signifikant erhöht (vgl. Abschnitt 4.4.1).

Für die weitere Betrachtung verbleiben Independent Non-Persistent Disks als resetorientierter Ansatz, sowie Linked und Instant Clones aufgrund ihres vergleichsweise geringen Ressourcenbedarfs und der erwarteten kurzen Bereitstellungszeiten.

5.1.2. Empirische Entscheidungsgrundlage

Im Zielmodell wird jede Job-Ausführung auf einer ephemeren Instanz ausgeführt. Der entscheidende operative Overhead besteht demnach in der wiederholten Rückführung

in einen definierten Ausgangszustand. Die Selektion der VM-Mechanik erfolgt demnach anhand der *Rücksetzzeit*.

Die Zeitspanne der Rücksetzzeit umfasst den Zeitraum vom Auslösen der Rücksetzoperation durch den Orchestrator bis zur Neuinitialisierung der VM und der Verbindung zu dieser. Das Konfigurieren und Starten der Runner-Anwendung wurde bewusst nicht betrachtet, da diese vom GitHub Enterprise Server abhängig ist.

Zur Verbesserung der Vergleichbarkeit erfolgt eine Zerlegung der Rücksetzzeit in ihre wesentlichen Anteile. Da die analysierten Mechaniken unterschiedliche Teilschritte erfordern, werden nicht anwendbare Schritte in der Auswertung entsprechend gekennzeichnet.

Es ist darauf hinzuweisen, dass sämtliche Messungen unter identischen Ausgangsbedingungen $n = 5$-mal wiederholt wurden. Die Mittelwerte werden im nächsten Abschnitt der Arbeit präsentiert, während die Rohdaten im Anhang G dokumentiert sind.

5.1.3. Ergebnis und Interpretation

Tabelle 5.1.: Gemessene Rücksetzzeiten (Mittelwert aus $n = 5$ Läufen)

Mechanik	**Rücksetzzeit (s)**	**StdAbw (s)**
Independent Non-Persistent Disk	12,67	0,36
Linked Clone	88,04	1,54
Instant Clone	11,52	0,84

Tabelle 5.2.: Zerlegung der Rücksetzzeit in Teilschritte (Mittelwert aus $n = 5$ Läufen)

Teilschritt	**Indep. Disk (s)**	**Linked (s)**	**Instant (s)**
VM löschen	–	3,18	3,92
Clone erzeugen	–	5,05	3,43
Power Cycle (Reset)	4,40	–	–
Netzwerkadapter setzten	–	(inkl.)	2,67
Verbindungsaufbau (SSH)	8,27	79,81	1,49
Gesamt	12,67	88,04	11,52

Die in Tabelle 5.1 dargestellten Ergebnisse zeigen signifikante Unterschiede in den gemessenen Rücksetzzeiten der untersuchten Mechaniken.

Der Vergleich ergab, dass Instant Clone mit einer durchschnittlichen Zeit von 11,52 Sekunden die geringste Rücksetzzeit aufweist. An zweiter Stelle befindet sich Independent Non-Persistent Disk mit einer durchschnittlichen Zeit von 12,67 Sekunden. Als letztes weist Linked Clone mit einer durchschnittlichen Zeit von 88,04 Sekunden die signifikant längste Rücksetzzeit auf.

Die Zerlegung des Prozesses in Teilschritte, wie in Tabelle 5.2 dargestellt, verdeutlicht, dass die Gesamtdauer primär durch den Schritt Verbindungsaufbau bestimmt wird. In der vorliegenden Methodik wird der Messpunkt nicht nur auf den eigentlichen Aufbau der SSH-Verbindung beschränkt, sondern umfasst auch das Hochfahren des Gastbetriebssystems bis zur Erreichbarkeit über das Netzwerk. Die vorliegende Datenanalyse zeigt, dass der pro-Job Overhead zu einem signifikanten Anteil durch diese Komponente repräsentiert wird.

Der Großteil der Zeit, welcher beim Linked Clone mit 79,81 Sekunden veranschlagt wird, fällt auf dessen Verbindungsaufbau- bzw. Bootstrapping-Anteil. Dies ist darauf zurückzuführen, dass Linked Clones zwar die Erstellung des VM-Zustands beschleunigen, jedoch weiterhin den vollständigen Bootvorgang der Instanz erfordern. Demgegenüber ist der infrastrukturelle Anteil aus VM-Löschung und Clone-Erzeugung vergleichsweise gering und beläuft sich auf etwa 8,23 Sekunden.

Darüber hinaus wurde festgestellt, dass das integrierte Setzen des Netzwerkadapters während des Erzeugungsprozesses des Linked Clones zu einer Verlangsamung dieses Prozesses im Vergleich zu Instant Clones führt. Im Allgemeinen wurde festgestellt, dass das integrierte Setzen des Netzwerkadapters eine höhere Geschwindigkeit aufweist als das separate Erstellen und Setzen des Netzwerkadapters, wie es bei der Instant-Clone-Methode der Fall ist.

Der Instant Clone reduziert die Rücksetzzeit durch den kurzen Erreichbarkeits-Anteil von 1,49 Sekunden. Dieser Vorgang entspricht den Erwartungen, da Instant Clones aus einem eingefrorenen Ausgangszustand instanziiert werden und somit ein beschleunigtes Bereitstellen der virtuellen Maschine ermöglichen. Der zusätzliche Schritt zum Setzen des Netzwerkadapters, welcher etwa 2,67 Sekunden in Anspruch nimmt, hat nur einen marginalen Einfluss im Vergleich zum Bootstrapping bei den Linked Clone.

Im Falle von Independent Non-Persistent Disk erfolgt der Reset selbst relativ zügig mit einen Power Cycle von etwa 4,4 Sekunden. Die Gesamtdauer wird jedoch in erster Linie durch die Erreichbarkeit der VM bestimmt, was etwa 8,27 Sekunden in Anspruch nimmt. Im Vergleich zu Instant Clone fällt dieser Anteil höher aus, da die virtuelle Maschine regulär gestartet und erst danach wieder über das Netzwerk erreichbar wird.

5.1.4. Auswahlentscheidung

Auf Grundlage der Messergebnisse wird Instant Clone als Bereitstellungs- und Rücksetzansatz für den Prototypen ausgewählt. Dieser Ansatz zeichnet sich durch die schnellste Rücksetzzeit von etwa 11,52 Sekunden aus. Dies trägt signifikant zur Reduzierung des pro-Job Overheads im ephemeren Betrieb bei. Instant Clone zeichnet sich zudem durch seine Ressourcenschonung aus, da die Instanzen aus einem eingefrorenen Ausgangszustand abgeleitet werden und kein wiederholtes Kopieren vollständiger VM-Images erfordern.

Des Weiteren wird durch diesen Lösungsansatz die Aktualisierung der Basisreferenz vereinfacht, da die Durchführung von Updates zentral an der Golden VM erfolgt (vgl. 4.4.3).

5.2. Zielarchitektur

Die Zielarchitektur basiert auf ephemeren Runnern, bei dem jede Job-Ausführung in einer neu instanziierten virtuellen Maschine erfolgt. Hierdurch wird ein definierter Ausgangszustand pro Job sichergestellt und die Persistenz zwischen den Ausführungen vermieden.

Als Kernkomponenten der Architektur lassen sich GitHub als Auslöser und Steuerinstanz für Workflows, ein zentraler Orchestrierungsdienst (Controller) sowie die VMware-Infrastruktur zur VM-basierten Bereitstellung der Runner identifizieren. Der Controller empfängt GitHub-Webhooks und initiiert über VMware die Erzeugung bzw. Entfernung von Runner-VMs. Die Konfiguration der Runner-VMs erfolgt deklarativ mittels Ansible, sodass eine reproduzierbare Installation, der Konfigurationsstand sowie die Registrierung gewährleistet werden können.

Für die Registrierung des Runners ist seitens des Controllers eine Just-In-Time-Config (JIT-Config) über die API von GitHub zu erstellen. Im Zuge dessen wird dem Controller ein Einmal-Token zurückgegeben, welcher für die Registrierung des Runners erforderlich ist.

Die Grundlage für die Instanziierung bildet eine gepflegte Golden VM, aus der Runner-Instanzen mittels Instant Clone erzeugt werden. Eine konfigurierte Anzahl an Runners steht dauerhaft zur Verfügung und wird durch den Deployment- und Update-Ablauf gestartet. Im nachfolgenden Abschnitt wird die konkrete Abfolge der Bereitstellung des Startzustandes dargelegt.

5.3. Deployment- und Update-Ablauf

Für den Betrieb ephemerer Runner ist ein Deployment- und Update-Ablauf erforderlich, der sowohl die initiale Bereitstellung als auch die Erneuerung der Runner nach Aktualisierungen der Golden VM beschreibt. Aufgrund der statischen Konfiguration der Anzahl der Runner im Prototyp wird der Runner-Pool als definierte Menge von VM-Instanzen betrachtet, die über das Ansible-Inventar eindeutig adressiert wird.

Der Deployment- und Update-Ablauf wird in Abbildung A.1 visuell dargestellt.

Ein Scheduler (Cron) löst standardmäßig den Deployment- und Update-Ablauf aus. Alternativ kann dieser manuell gestartet werden. Im ersten Schritt erfolgt die Deregistrierung aller Runner von GitHub sowie die Löschung der Runner-VMs. Dieser Schritt ist eine notwendige Maßnahme, um zu verhindern, dass Runner geklont werden, während sich die Golden VM in einem unbestimmten Zustand befindet, der durch die laufende Aktualisierung entsteht. Im nächsten Schritt ist ein Neustart der Golden VM erforderlich, um den eingefrorenen Zustand zu beenden (vgl. Abschnitt 2.3.6). Im darauf folgenden Schritt wird die Golden VM aktualisiert und in einen konsistenten Ausgangszustand überführt, der als Basis für die Clone-Erzeugung dient. Im Anschluss wird die Golden VM erneut eingefroren. Abschließend wird der im Ansible-Inventar statisch definierte Runner-Pool neu bereitgestellt, indem für jede Runner-Instanz ein Instant Clone erzeugt wird und über eine JIT-Config beim GitHub Runner Service registriert wird. Nach Abschluss befindet sich der Runner-Pool im Betriebszustand und stellt wartende Runner für eingehende Jobs bereit.

5.4. Runner-Lifecycle

Aufbauend auf dem in Abschnitt 5.3 dargestellten Betriebszustand des bereitgestellten Runner-Pools wird im Folgenden der Lifecycle einer einzelnen Runner-VM betrachtet. Der Fokus liegt auf dem Ablauf von der Jobzuweisung über die Ausführung bis zur automatisierten Rückführung.

Der Runner-Lifecycle wird in Abbildung A.2 visuell dargestellt.

Ausgangspunkt ist eine registrierte Runner-VM im Idle-Zustand, der von GitHub einen Job zugewiesen bekommt. Nach Abschluss der Ausführung wird das Ergebnis an GitHub übermittelt. Das Ende der Ausführung des Jobs löst eine Webhook aus, welche durch den Orchestrator verarbeitet wird. In der Folge initialisiert der Orchestrator ein Ansible-Playbook, welches dazu dient, die bestehende Runner-VM zu löschen und eine neue Instanz mittels Instant Clone aus der Golden VM zu erzeugen. Für die Registrierung des Runners wird eine JIT-Config angefordert. Mit dieser wird auf der neu instanziierten virtuellen Maschine der Runner gestartet. Abschließend registriert sich der neue Runner und

befindet sich wieder im Idle-Zustand, sodass der Zyklus für die nächste Job-Ausführung erneut beginnt.

5.5. Windows-Runner

Die konzipierte Zielarchitektur lässt sich prinzipiell auch auf Windows-basierte Runner übertragen. Die Bereitstellung auf der Seite von VMware bleibt dabei unverändert, da Mechanismen wie Instant Clone, VM-Löschung und die Verwaltung der Basis-VM unabhängig vom eingesetzten Gastbetriebssystem funktionieren. Für das Betriebssystem Windows ist jedoch eine separate Golden VM erforderlich, die den gewünschten Ausgangszustand der Runner-Instanz abbildet.

Der wesentliche Unterschied zur Linux-basierten Umsetzung liegt in der Art der Konfiguration und Fernverwaltung. Anstelle einer SSH-basierten Anbindung wird für das Betriebssystem Windows das Windows Remote Management (WinRM) in Ansible zur Kommunikation mit der Runner-VM verwendet. In der Konsequenz bedürfen die Ansible-Playbooks und Tasks einer Anpassung, um die spezifischen Schritte für das jeweilige Betriebssystem zu berücksichtigen. Dies betrifft beispielsweise die Bereitstellung der Runner-Software, die Übergabe der JIT-Konfiguration sowie den Start des Runner-Dienstes.

6. Prototypische Implementierung

In diesem Kapitel wird die prototypische Umsetzung der in Kapitel 5 konzipierten Zielarchitektur beschrieben. Der Fokus liegt auf der technischen Realisierung der Orchestrierung, der Ansible-basierten Provisionierung sowie der Anbindung an GitHub-Webhooks und die VMware-Infrastruktur. Im Rahmen dessen erfolgt die Darstellung der wesentlichen Implementierungsbausteine, ihrer Schnittstellen und zentraler Abläufe.

6.1. Implementierungsübersicht und Werkzeuge

Der Orchestrator wurde als schlanker Webservice in Python implementiert und verwendet FastAPI[1] mit Uvicorn[2] als ASGI-Server[3]. Der Webservice stellt einen Endpunkt zur Entgegennahme von GitHub-Webhooks bereit. Der Orchestrator nimmt die Validierung eingehender Webhooks vor, filtert relevante Events und leitet anschließend die Ausführung der entsprechenden Ansible-Playbooks mit dem extrahierten Kontext der Webhooks ein.

Ansible übernimmt sämtliche Bereitstellungs- und Rücksetzoperationen, einschließlich der Interaktion mit VMware, der Konfiguration der Runner-VMs sowie der API-Anforderung und Verwendung der JIT-Config zur Runner-Registrierung. Die fachliche Logik zur VM-Steuerung wurde bewusst in Ansible ausgelagert, um die Austauschbarkeit des Python-basierten Webservices zu gewährleisten. Die Anzahl der Runner ist im Prototyp statisch konfiguriert und wird über ein Ansible-Inventar abgebildet, das die Runner-Instanzen eindeutig adressiert.

Für zeitgesteuerte Updates und Neu-Bereitstellungen wird ein Scheduler (Cron) verwendet, der die Update- und Deploy-Ansible-Playbooks direkt auslöst und nicht vom Orchestrator abhängig ist.

[1] FastAPI ist ein Python-Webframework zur Entwicklung von APIs auf Basis des ASGI-Standards [32].

[2] Uvicorn ist ein ASGI-Server für Python [33]. Uvicorn wird standardmäßig für FastAPI Anwendungen genutzt [32].

[3] ASGI (*Asynchronous Server Gateway Interface*) ist eine Python-Schnittstellenspezifikation zwischen Webserver und Anwendung, die asynchrone Requests und WebSockets unterstützt [34].

Der vollständige Quellcode der Implementierung, einschließlich des Orchestrator-Services, der Ansible-Playbooks, des Ansible-Inventars und der Hilfsskripte, ist in einem GitHub-Repository abgelegt [35].

6.2. Orchestrator

Der Orchestrator stellt dabei die zentrale Komponente der prototypischen Implementierung dar. Dieser stellt den Webhook-Endpunkt für GitHub bereit, übernimmt die Sicherheitsprüfung und Vorverarbeitung eingehender Events und initiiert darauf aufbauend die Ausführung der Ansible-basierten Orchestrierung. Die nachfolgenden Abschnitte befassen sich zunächst mit dem Deployment und der Konfiguration der Komponente und erörtern im weiteren Verlauf die Webhook-Verarbeitung unter Einbezug der Signaturprüfung, der Event-Filterung sowie der Übergabe des Ausführungskontexts an Ansible.

6.2.1. Deployment und Konfiguration

Der Orchestrator wird als Docker-Container auf einer dedizierten virtuellen Maschine innerhalb der VMware-Infrastruktur betrieben. Im Entwicklungsbetrieb wird für die Erreichbarkeit des Webservices ein vorgeschalteter Reverse-Proxy auf Basis von Nginx[4] eingesetzt. Nginx wird ebenfalls in einer containerisierten Ausführung betrieben, ist auf dem externen Port 443 aktiv und leitet Requests über ein internes Docker-Netzwerk an den Orchestrator-Container weiter.

Die Laufzeitkonfiguration des Webservices wird über eine Environment-Variable bereitgestellt und beim Containerstart eingelesen. Dies umfasst ausschließlich den GitHub-Webhook-Secret, weitere Informationen dazu finden sich im Abschnitt 6.2.2.

Die statische Applikationskonfiguration des Webservices über die Ansible-Informationen (z. B. Ansiblepfad, Inventarpfad und Artefaktverzeichnis) ist im Programmcode von Python integriert. Zur strukturierten Verwaltung dieser Parameter wird ein Settings-Modell verwendet, das die Konfiguration typisiert und zentral zusammenführt.

Die hostspezifische Konfiguration der Runner-Instanzen wird durch das Ansible-Inventar repräsentiert. Die Konfiguration der Parameter über VMware (API-Endpunkt, Anmeldeinformationen, Template-Name etc.) sowie über GitHub (API-Endpunkte, Token) wird ebenfalls über das Ansible-Inventar abgebildet. Im Zuge dessen erfolgt eine Verschlüsselung sensibler Daten, welche erst zur Laufzeit zur Verfügung gestellt werden. Dafür wird Ansi-

[4] Nginx ist ein Webserver und Reverse-Proxy, der u. a. zur Weiterleitung eingehender HTTP(S)-Anfragen an interne Dienste eingesetzt wird [36].

ble Vault[5] eingesetzt.

Im Deployment werden die erforderlichen Schlüsselmaterialien (z. B. Vault-Key und SSH-Key) als Volumes in den Container eingebunden, um eine Ablage in Container-Layern zu vermeiden. Die Containerdefinitions- und Konfigurationsdateien sind im Repository dokumentiert [35].

6.2.2. Webhook-Verarbeitung und Sicherheitsprüfung

Die Verarbeitung der eingehenden GitHub-Webhooks erfolgt vollständig innerhalb des Python-Webservices. Der Dienst stellt einen HTTPS-Endpunkt bereit, nimmt Webhook-Requests entgegen und führt eine Signaturprüfung durch. Zu diesem Zweck wird der von GitHub übermittelte Header `x-hub-signature-256` ausgewertet und mittels HMAC-SHA-256 gegen das lokal konfiguriertes GitHub-Webhook-Secret verifiziert. Im Falle fehlender Signaturen oder Abweichungen erfolgt eine Abweisung der Anfrage mittels HTTP 403, sodass ausschließlich authentische Webhook-Requests verarbeitet werden.

Die Signaturprüfung wird durch FastAPIs Dependency Injection realisiert und somit zentral für alle relevanten Routen erzwungen. Hierdurch erfolgt eine Entkopplung der eigentlichen Handler-Funktionen von der Sicherheitslogik, wobei die Prüfung für jede Anfrage dennoch garantiert ist.

Nach erfolgreicher Verifikation erfolgt die Typisierung und Validierung des Payloads anhand eines Pydantic-Datenmodells[6]. Der Webservice verarbeitet ausschließlich Events des Typs `workflow_job` und filtert diese auf den Status `completed`. Events, welche diese Kriterien nicht erfüllen, werden in der Verarbeitung ignoriert und mittels des Status `ingnored` quittiert. Für relevante Events wird der `runner_name` aus dem Payload extrahiert und als Environment-Variable an die nachgelagerte Automatisierung übergeben.

Die Orchestrierungslogik wird durch das Starten eines Ansible-Playbooks über die Ansible Runner-API ausgeführt. Der Webservice nimmt dabei die Funktion eines Triggers ein. Die Asynchronität des Playbooks ist essenziell, um eine Blockade des Python-Prozesses zu verhindern und die Timeout-Regel von GitHub zu erfüllen. Gemäß dieser Regelung wird eine Webhook-Nachricht nach zehn Sekunden ohne Antwort als Timeout betrachtet [39].

Die wesentlichen Implementierungsdetails sind im Listing 6.1 dargestellt. Die vollständige Implementierung, bei der unter anderem die Python-Komponenten in mehreren Dateien unterteilt sind, sind im referenzierten Repository enthalten [35].

[5] Ansible Vault ermöglicht die verschlüsselte Ablage sensibler Variablen und Dateien und deren Entschlüsselung zur Laufzeit [37].

[6] Pydantic ist eine Python-Bibliothek zur Validierung und Typisierung strukturierter Daten auf Basis von Typannotationen [38].

Listing 6.1: Webhook-Validierung, Event-Filter und Start der Ansible-Orchestrierung

```
# Dependency: HMAC-SHA256 Signaturpruefung (x-hub-signature-256)
securityAgent = GitHubWebhookSecurity(secret=settings.github_secret)
app = FastAPI(dependencies=[Depends(securityAgent)])

@router.post("/webhook")
async def webhook(payload: WebhookPayload):
    # nur workflow_job.completed verarbeiten
    if payload.action != "completed" or not payload.workflow_job:
        return {"status": "ignored"}

    # runner_name als Environment-Varibale an Ansible geben
    os.environ["runner_name"] = payload.workflow_job.runner_name

    # Orchestrierung an Ansible delegieren (asynchron)
    run_async(
        private_data_dir=settings.ansible_private_dir,
        playbook=settings.ansible_webhook_playbook,
        artifact_dir=settings.ansible_artifact_dir,
    )
    return {"status": "processed"}
```

6.3. Golden VM: Erstellung und Basisumgebung

Die Bereitstellung ephemerer Runner-VMs erfolgt auf Basis einer Golden VM, welche den definierten Softwarestand der Runner-Umgebung repräsentiert. Für den Prototyp wurde hierfür eine Ubuntu-basierte VM-Vorlage verwendet und manuell zu einer lauffähigen Runner-Basis ausgebaut.

Die Basiskonfiguration umfasst die Anlage eines dedizierten Runner-Benutzers ohne administrative Rechte sowie die Einrichtung eines SSH-Zugangs für die automatisierte Konfiguration durch Ansible. Zu diesem Zweck wurde ein SSH-Schlüssel auf der Golden VM hinterlegt. Darüber hinaus erfolgte die Installation von Docker. Es erfolgte keine Installation zusätzlicher Programmiersprachen, Tools oder Frameworks. Gemäß den spezifischen Erfordernissen des produktiven Betriebs ist eine Installation dieser Komponenten erforderlich. Darüber hinaus wurde die Runner-Anwendung auf der Golden VM installiert.

Im Rahmen des Update- und Bereitstellungsprozesses werden Gastbetriebssystem-Operationen benötigt, zu deren Realisierung die VMware Tools installiert werden. Die konkrete Aktualisierung der Golden VM sowie die Erzeugung des eingefrorenen Parent-Zustands werden im Rahmen der Deployment- und Update-Abläufe in Abschnitt 6.5 beschrieben.

6.4. Ansible

Die Orchestrierungslogik wird im vorliegenden Prototyp unter Verwendung von Ansible implementiert. Im Folgenden werden das Inventar- und das Variablenmodell, die Zuordnung von Runnern zu VM-Instanzen sowie der Abruf der JIT-Config mit anschließendem Runner-Start beschrieben.

6.4.1. Inventar und Variablenmodell

Für die Orchestrierung wird ein statisches Ansible-Inventar verwendet, das die im Prototyp vorgesehenen Runner-VMs als Hosts abbildet. Die Hostnamen im Inventar entsprechen dabei den VM-Namen bzw. FQDN (Fully Qualified Domain Name) innerhalb der VMware-Umgebung und werden für die technische Adressierung (z. B. SSH und VMware-Operationen) genutzt. Des Weiteren ist im Inventar auch das Golden Image als Host aufgeführt, damit auf diese für das Updaten zugegriffen werden kann. Die YAML-basierte Inventardefinition ist im Anhang B.1.

Den einzelnen Hosts werden über separate `host_vars`-Dateien Parameter zugeordnet. Dies umfasst eine Beschreibung der Registrierung des Runners sowie eine MAC-Adresse für das indirekte Setzen des FQDN. Eine Beispieldatei ist im Anhang B.2 zu finden.

Die Strukturierung der Runner erfolgt zusätzlich über Ansible-Gruppen, die inhaltlich den in GitHub verwendeten Runner-Gruppen entsprechen. Für jede Ansible-Gruppe wird eine separate `group_vars`-Datei verwendet. Gruppenspezifische Einstellungen beinhalten die `runner_group_id` sowie Labels, welche für die Registrierung des Runners benötigt werden. Eine Beispieldatei ist im Anhang B.3 zu finden.

6.4.2. Zuordnung zwischen Runner und VM

Der Webhook beinhaltet den `runner_name` des Runners, welcher die Aufgabe abgeschlossen hat. Der Webservice extrahiert diesen Wert, übergibt ihn an Ansible und initialisiert die Ausführung eines Playbooks. Damit ergibt sich die Frage, auf welchem Inventar-Host das Playbook ausgeführt werden soll, da Ansible nicht automatisch aus dem `runner_name` die zugehörige VM ableiten kann. Der Grund dafür ist, dass die VMs den FQDN als Hostnamen nutzen.

Für die Lösung wird Zu Beginn des Playbooks ein initialer Task ausgeführt, der die Zuordnung vom `runner_name` zu einem Inventar-Host ermittelt. Zu diesem Zweck wird über die im Inventar konfigurierten Runner-Hosts iteriert und derjenige Host ausgewählt, dessen hinterlegter `registration.name` mit dem aus dem Webhook übergebenen `runner_name` übereinstimmt.

Wird kein geeigneter Host ermittelt, bricht das Playbook mit einer entsprechenden Fehlermeldung ab. Im Erfolgsfall wird der gefundene Host zur Laufzeit der dynamischen Gruppe `selected_runner` hinzugefügt. Dies hat zur Folge, dass alle nachfolgenden Tasks gezielt mit `hosts: selected_runner` ausgeführt werden können. Die konkrete Umsetzung der zuvor beschriebenen Zuordnung ist in Listing 6.2 dargestellt.

Listing 6.2: Zuordnung von `runner_name` zu Inventar-Host

```
- name: Select target runner from webhook payload
  hosts: localhost
  gather_facts: false
  vars:
    runner_name: "{{ lookup('env','runner_name') }}"
  tasks:
    - name: Search through runner group
      set_fact:
        target_runner: "{{ item }}"
      loop: "{{ groups['runners'] }}"
      when: hostvars[item].registration.name | default('') == runner_name
      run_once: true

    - name: Fail if no matching runner was found
      ansible.builtin.fail:
        msg: "No runner found with name = {{ runner_name }}"
      when: target_runner is not defined

    - name: Add found host
      ansible.builtin.add_host:
        name: "{{ target_runner }}"
        groups: selected_runner

- name: Execute steps on selected runner
  hosts: selected_runner
  tasks:
     ...
```

6.4.3. JIT-Konfiguration und Runner-Start

Die Registrierung der Runner-Instanzen erfolgt über eine JIT-Config, die für jede Neuinstanziierung dynamisch über die GitHub-API erzeugt wird (vgl. 5.2). Im Rahmen des Ansible-Ablaufs erfolgt die Ausführung eines HTTP-POST gegen den JIT-Endpunkt sowie die Extraktion des von GitHub zurückgegebenen Parameters `encoded_jit_config`. Listing 6.3 zeigt den entsprechenden Ansible-Task.

Listing 6.3: GitHub-API-Aufruf zur Erzeugung der JIT-Konfiguration

```
- name: API Call to generate JIT-Config
  ansible.builtin.uri:
    url: "{{ github_jit_endpoint }}"
    method: POST
    headers:
      Accept: "application/vnd.github+json"
      Authorization: "Bearer {{ github_auth_token }}"
      X-GitHub-Api-Version: "2022-11-28"
      Content-Type: "application/json"
    body_format: json
    body: "{{ registration | to_json }}"
    return_content: true
    status_code: 201
    timeout: 10
  register: jit_resp
  delegate_to: localhost

- name: Set JIT-Config as fact
  ansible.builtin.set_fact:
    encoded_jit_config: "{{ jit_resp.json.encoded_jit_config }}"
```

Im Anschluss an den Erzeugungsprozess wird die JIT-Config für den Start des Runner-Prozesses an die Runner-VM übergeben. Der Prototyp sieht eine zweistufige Vorgehensweise vor. Zunächst wird eine temporäre Environment-Datei auf der VM erzeugt, die den JIT-Parameter enthält. Anschließend wird ein systemd-Dienst gestartet. Der Dienst führt das Runner-Startskript unter dem Benutzerkonto `runner` aus, welches über keine administrativen Rechte verfügt.

Die Erstellung der Environment-Datei sowie der Start des Dienstes erfolgt mittels Ansible, wie in der Listing 6.4 dargestellt. Der systemd-Dienst ist bereits auf der Golden VM vorinstalliert und wird im Anhang D.1 dargestellt.

Listing 6.4: Übergabe der JIT-Config und Start des Runner-Dienstes

```
- name: Create runner env file
  ansible.builtin.copy:
    dest: /tmp/runner.env
    owner: runner
    group: runner
    mode: "0600"
    content: |
      START_OPTION=--jitconfig {{ encoded_jit_config }}

- name: Start Runner (systemd)
  ansible.builtin.command:
    cmd: "systemctl start action-runner.service"
```

6.5. Deployment- und Update-Ablauf

Der Deployment- und Update-Ablauf setzt die in Abbildung A.1 dargestellten Phasen technisch um und wird im Prototyp über einen zeitgesteuerten Cron-Job angestoßen. Der Cron-Job initialisiert die zugehörigen Ansible-Playbooks in definierter Reihenfolge und bildet damit sowohl das Wartungsfenster (Update der Golden Image) als auch die anschließende Neu-Bereitstellung der Runners ab. Das Skript des Cronjobs ist dem Anhang D.2 zu entnehmen.

Die Phase *Update der Basisumgebung* wird durch ein Playbook umgesetzt, dass in Listing 6.5 dargestellt wird. Die verwendeten Ansible-Tasks sind im Anhang C zu finden.

Im Vorfeld des Updates der Golden Image erfolgt eine Deregistrierung sämtlicher Runner von GitHub sowie die Löschung der zugehörigen Runner-VMs. Die Deregistrierung erfolgt dabei nicht über Runner-Namen, sondern über die von GitHub vergebenen Runner-IDs. Zu diesem Zweck wird zunächst die Liste der registrierten Runner mittels eines API-Aufrufs abgerufen und anschließend für jede enthaltene Runner-ID ein DELETE-Request an den entsprechenden Runner-Remove-Endpoint ausgeführt.

Im Anschluss erfolgt die eigentliche Aktualisierung des Golden Image. Zu diesem Zweck wird das Golden Image neu gestartet, um den eingefrorenem Zustand zu beenden. Im darauffolgenden Schritt wird ein Netzwerkadapter hinzugefügt, um eine Verbindung per SSH herzustellen, die es Ansible ermöglicht, die Aktualisierung auszuführen.

Im Prototyp beschränkt sich die Aktualisierung auf Systemupdates mittels `apt update` und `apt upgrade`. Im Anschluss wird das Golden Image eingefroren. Abschließend wird der Netzwerkadapter entfernt, da dieser andernfalls von Instant Clones mitkopiert wird, was zu Netzwerkkonflikten führt.

Listing 6.5: Playbook des Deployment- und Update-Ablauf

```
1  - name: Remove runners from GitHub
2    hosts: localhost
3    tasks:
4      - include_tasks: tasks/github-remove-runner.yml
5
6  - name: Delete runner-vms
7    hosts: runners
8    tasks:
9      - include_tasks: tasks/delete-vm.yml
10
11 - name: Update template
12   hosts: template
13   tasks:
14     - include_tasks: tasks/power-restart.yml
15     - include_tasks: tasks/set-mac-address.yml
16     - include_tasks: tasks/update-template.yml
17     - include_tasks: tasks/freeze-guest-host.yml
18     - include_tasks: tasks/remove-nic.yml
```

Für die Phase *Pool Deployment* wird ein separates Playbook ausgeführt, dass in Listing 6.6 dargestellt wird.

Im Rahmen dieses Prozesses wird das Inventar der statisch definierten Runner-Instanzen iteriert und für jede Instanz ein Instant Clone erzeugt sowie eine MAC-Adresse gesetzt. Im Anschluss an den Prozess der Clone-Erzeugung werden pro Runner eine JIT-Config über die GitHub-API angefordert und die Runner-Anwendung gestartet. Dieser Prozess führt dazu, dass sich die Instanzen erneut registrieren und in den Betriebszustand übergehen (vgl. 6.4.3).

Listing 6.6: Playbook des Pool Deployment

```
1 - name: Deploy runners
2   hosts: runners
3   tasks:
4     - include_tasks: tasks/create-instant-clone.yml
5     - include_tasks: tasks/set-mac-address.yml
6     - include_tasks: tasks/github-generate-jit.yml
7     - include_tasks: tasks/runner-start.yml
```

6.6. Runner-Lifecycle

Der Betrieb einer Runner-Instanz wird durch den in Abbildung A.2 dargestellten Ablauf realisiert. Die Phasen 0 und 3, welche die Jobzuweisung sowie den erneuten Registrierungs- bzw. Idle-Zustand umfassen, werden durch GitHub bzw. den Runner selbst abgebildet. In der ersten Phase (Trigger) erfolgt die Verarbeitung des Webhooks im Orchestrator, wie bereits in Abschnitt 6.2.2 beschrieben. Die Zuordnung des im Webhook-Payload enthaltenen `runner_name` zu einem konkreten Inventar-Host erfolgt in Ansible gemäß der in Abschnitt 6.4.2 dargestellten Vorgehensweise. In der Folge wird daher Phase 2 in den Mittelpunkt gestellt, welche die technische Rückführung und Bereitstellung der Runner-VM implementiert.

Die Rückführung und Bereitstellung wird durch ein Ansible-Playbook realisiert, welches in Listing 6.7 dargestellt wird. Die verwendeten Ansible-Tasks sind im Anhang C zu finden.

Das vorliegende Playbook weist im Wesentlichen drei Schritte auf. Zunächst wird die vorhandene Runner-VM verworfen und anschließend ein neuer Instant Clone aus der Golden VM erzeugt. Im darauffolgenden Schritt wird die erforderliche MAC-Adresse gesetzt. Danach wird eine neue JIT-Config über die GitHub-API angefordert und schließlich wird die Runner-Anwendung auf der neu instanziierten VM gestartet.

Listing 6.7: Playbook für Rückführung und Bereitstellung einer Runner-VM

```
1 - name: Setup VM again
2   hosts: selected_runner
3   tasks:
4     - include_tasks: tasks/delete-vm.yml
5     - include_tasks: tasks/create-instant-clone.yml
6     - include_tasks: tasks/set-mac-address.yml
7     - include_tasks: tasks/github-generate-jit.yml
8     - include_tasks: tasks/runner-start.yml
```

6.7. Windows-Runner

Im Rahmen der vorliegenden Arbeit wurde die Übertragbarkeit der Zielarchitektur auf Windows-basierte Runner grundsätzlich untersucht und manuell geprüft. Die Untersuchung ergab, dass die Bereitstellung von VMware auch für Windows-basierte VMs prinzipiell auf die gleiche Weise umgesetzt werden kann und sich der grundlegende Ablauf der Neuinstanziierung konzeptionell nicht ändert.

Eine vollständige Automatisierung dieser Abläufe in Form von Ansible-Playbooks und Tasks wurde hingegen nicht realisiert. Die Schritte zur Konfiguration, Fernverwaltung und Runner-Registrierung hätten dabei gesondert implementiert werden müssen. Die prototypische Implementierung ist demnach auf die automatisierte Umsetzung für Linux-basierte Runner-Instanzen beschränkt.

7. Evaluation der Zielarchitektur

In diesem Kapitel erfolgt eine Evaluierung der konzipierten und prototypisch implementierten Zielarchitektur. Im Folgenden werden die Methodik, die durchgeführten Tests, die Bewertung der Anforderungserfüllung sowie die Grenzen der Evaluation dargelegt.

7.1. Evaluationsmethodik

Da es sich um einen Prototypen handelt, erfolgt die Evaluation als Kombination aus funktionalen Tests sowie konzeptioneller Bewertung von Eigenschaften, die im gegebenen Rahmen nicht vollständig empirisch überprüft werden können. Funktionale Tests dienen dem Nachweis des automatisierten Lifecycles sowie der erwarteten Zustandsrückführung pro Job. Der Einsatz konzeptioneller Bewertungen ermöglicht die Einordnung architektureller Eigenschaften, wie etwa die Wartbarkeit des Prototyps.

Abschließend erfolgt die Dokumentation der Ergebnisse in Form eines Anforderungsabgleichs. Im Zuge dessen erfolgt eine Zuordnung jeder Anforderung zu einer Test- bzw. Nachweismethode sowie eine Bewertung der Erfüllung der Anforderung als *erfüllt*, *teilweise erfüllt* oder *nicht erfüllt*. Die zugehörigen Evidenzen werden referenziert.

7.2. Tests

In diesem Abschnitt erfolgt die Dokumentation der durchgeführten Tests zur Überprüfung der zentralen Anforderungen. Die Testfälle werden jeweils durch Ablauf, Erwartung und Evidenz (Workflow-Ergebnis) nachvollziehbar belegt.

7.2.1. Test 1: Persistenzfreiheit zwischen zwei Job-Ausführungen

Das Ziel des Tests liegt auf der Erbringung des Nachweises, dass zwischen zwei Job-Ausführungen keine Dateisystem-Artefakte aus der vorherigen Ausführung erhalten bleiben. Zu diesem Zweck wird ein zweistufiger Workflow verwendet, welcher sich aus zwei aufeinanderfolgenden Jobs zusammensetzt. Der Workflow ist dem Anhang E.1 zu entnehmen.

Im ersten Job wird eine Datei in der Runner-VM erzeugt. Im Rahmen des zweiten Jobs erfolgt eine Prüfung, ob diese Datei nach wie vor existiert. Es wird davon ausgegangen, dass die Datei im zweiten Job nicht vorhanden ist.

Um sicherzustellen, dass beide Jobs dem gleichen Runner-Kontext zugeordnet werden, ist während der Testdurchführung nur eine Runner-Instanz in GitHub registriert. Diese Vorgehensweise dient dazu, die Ausführung des zweiten Jobs auf einer anderen Runner-Instanz zu unterbinden und somit eine Verfälschung des Testergebnisses zu verhindern.

Die Ausführung des Workflows erfolgte erfolgreich, ohne dass es zu Fehlern kam, siehe Anhang F.1.

7.2.2. Test 2: Persistenzfreiheit containerbezogener Zustände

Das Ziel des Tests liegt auf der Erbringung des Nachweises, dass zwischen zwei Job-Ausführungen Container-bezogene Artefakte aus der vorherigen Ausführung erhalten bleiben. Zu diesem Zweck wird ein zweistufiger Workflow verwendet, welcher sich aus zwei aufeinanderfolgenden Jobs zusammensetzt. Wie bereits im Test 1 beschrieben, ist lediglich eine Runner-Instanz registriert. Der Workflow ist dem Anhang E.2 zu entnehmen.

Im ersten Job werden ein Docker-Image sowie ein Docker-Volume erzeugt. Im zweiten Job wird geprüft, ob das zuvor erzeugte Image bzw. Volume weiterhin vorhanden ist. Es wird davon ausgegangen, dass Container-bezogene Artefakte im zweiten Job nicht vorhanden sind.

Die Ausführung des Workflows erfolgte erfolgreich, ohne dass es zu Fehlern kam, siehe Anhang F.2.

7.2.3. Test 3: Ausführung ohne Root-Rechte

Das Ziel des Tests liegt auf der Erbringung des Nachweises, dass Job-Schritte auf der Runner-Instanz nicht mit Root-Rechten ausgeführt werden. Hierzu wird in einem Workflow die effektive Benutzer-ID geprüft. Der Workflow ist dem Anhang E.3 zu entnehmen. Die Erwartung ist, dass `id -u` ungleich `0` ist und damit keine Root-Ausführung vorliegt.

Die Ausführung des Workflows erfolgte erfolgreich, ohne dass es zu Fehlern kam, siehe Anhang F.3.

7.3. Bewertung

Tabelle 7.1.: Anforderungsabgleich: Zuordnung von Anforderungen zu Nachweisen und Bewertung

ID	Kriterium / Nachweis	Bewertung	Evidenz
F1	Start in definierter Ausgangsumgebung	erfüllt	7.2.1
F2	Keine Persistenz zwischen Jobs	erfüllt	7.2.1
F3	Automatisierte Bereitstellung der Runner-Instanzen	erfüllt	7.2.1
F4	Runner-Lifecycle ohne manuellen Eingriff	erfüllt	7.2.1
F5	Unterstützung containerbasierter Jobs	erfüllt	7.2.2
I1	Keine Beeinflussung zwischen Jobs	erfüllt	7.2.1
I2	Artefakte vorheriger Jobs nicht nutzbar	erfüllt	7.2.1
I3	Keine containerbasierten Artefakte zwischen Jobs	erfüllt	7.2.2
S1	Kein privilegierter Host-Zugriff der Runner-Instanzen	erfüllt	7.2.3
S2	Zustand vor/nach Job eindeutig definiert	erfüllt	7.2.1
S3	Keine systemweiten Änderungen durch Jobs	erfüllt	7.2.1
W1	Golden Image automatisierbar wartbar	erfüllt	F.4
W2	VM-Konfiguration deklarativ über Ansible	erfüllt	B
W3	Unterstützung für Windows	teilweise erfüllt	6.7
R1	Umsetzung innerhalb VMware-Infrastruktur	erfüllt	6, C
R2	Orchestrierung/Steuerung mittels Ansible	erfüllt	6, C
C1	Es wird nur ein Runner pro VM betrieben	erfüllt	6.4

7.4. Grenzen

Die Evaluation erfolgt im Rahmen eines Prototyps und ist entsprechend in Umfang und Aussagekraft begrenzt. Es wurde keine Langzeitbeobachtung durchgeführt, sodass Aussagen zur Stabilität über längere Betriebszeiträume (z. B. Wochen/Monate) nicht möglich sind.

Des Weiteren beinhaltet die Evaluation keine Prüfung der Hochverfügbarkeit oder des Failovers. Fehlerszenarien wie Ausfälle des Orchestrators, von Ansible-Control oder VMware sowie Wiederanlaufstrategien wurden weder konzipiert noch getestet.

Die sicherheitstechnische Bewertung beschränkt sich auf strukturelle Eigenschaften der Architektur und einfache Negativtests wie z. B. Ausführung ohne Root-Rechte (vgl. 7.2.3). Ein Security-Audit oder Exploit-basierte Tests sind nicht Bestandteil (vgl. 2.2.5).

Die Evaluation basiert auf einer begrenzten Testumgebung mit einer geringen Anzahl statisch konfigurierter Runner und ohne Lasttests unter realistischen Parallelitätsbedingungen. Die Ergebnisse sind demnach primär als Nachweis der grundsätzlichen Funktionsfähigkeit und Zielerreichung im Prototypbetrieb zu interpretieren.

8. Fazit und Ausblick

In diesem Kapitel werden die wesentlichen Ergebnisse der Arbeit zusammengefasst und in Bezug auf die Zielsetzung der Arbeit eingeordnet. Abschließend werden potenzielle Weiterentwicklungen und offene Fragestellungen skizziert, die sich aus dem prototypischen Charakter der Umsetzung ergeben.

8.1. Zusammenfassung der Ergebnisse

Ziel dieser Arbeit war die Konzeption und prototypische Realisierung einer sicheren Deployment-Architektur für selbst gehostete GitHub-Runner der WHZ. Ausgangspunkt der Arbeit war die Analyse der bestehenden Runner-Umgebung, in der insbesondere die Persistenz zwischen Job-Ausführungen sowie die eingeschränkte Isolation als zentrale Schwachstellen identifiziert wurden.

Auf dieser Grundlage wurden Anforderungen an eine Zielarchitektur abgeleitet und verschiedene Bereitstellungs- und Rücksetzmechanismen für virtuelle Maschinen in VMware untersucht. Die konzeptionelle und empirische Bewertung ergab, dass Instant Clone für den betrachteten Anwendungsfall den geeignetsten Mechanismus darstellt. Die ausschlaggebenden Faktoren sind dabei insbesondere die geringe Rücksetzzeit, der vergleichsweise geringe Ressourcenbedarf sowie die zentrale Pflege der Basisumgebung über eine Golden VM.

Aufbauend auf diesen Erkenntnissen wurde eine Zielarchitektur entwickelt, die GitHub-Webhooks, einen Orchestrator, Ansible-basierte Automatisierung und die VMware-Infrastruktur zu einem durchgängig automatisierten Runner-Lifecycle verbindet. Die prototypische Implementierung demonstriert, dass Runner-Instanzen nach Abschluss eines Jobs automatisch neu bereitgestellt werden. Dadurch wird die Persistenz zwischen Ausführungen wirksam unterbunden.

Ein weiterer zentraler Bestandteil der Arbeit war die Konzeption und prototypische Umsetzung eines automatisierten Deployment- und Update-Ablaufs für die Runner-Umgebung. Die Trennung zwischen Golden VM als gepflegter Ausgangsbasis und den daraus erzeugten Runner-Instanzen ermöglicht eine zentrale Durchführung von Aktualisierungen und eine konsistente Übertragung auf neu bereitgestellte Instanzen. Der implementierte

Ablauf demonstriert, dass sowohl die Neu-Bereitstellung des Runner-Pools als auch die Rückführung einzelner Instanzen ohne manuelle Wartung umsetzbar ist. Die vorliegende Arbeit adressiert demnach nicht nur die Ausführungsisolation einzelner Jobs, sondern auch die betriebliche Wartbarkeit der zugrunde liegenden Runner-Infrastruktur.

Die Evaluation bestätigt, dass die wesentlichen Anforderungen im Rahmen des Prototyps erfüllt werden. Insbesondere konnten die automatisierte Bereitstellung und Rückführung der Runner-Instanzen, die Reproduzierbarkeit der Ausgangsumgebung sowie die Nicht-Persistenz datei- und containerbezogener Zustände zwischen Job-Ausführungen nachgewiesen werden. Die vorliegende Arbeit leistet einen praxisnahen Beitrag zur strukturierten Bereitstellung ephemerer GitHub-Runner in einer bestehenden Virtualisierungsumgebung.

8.2. Ausblick

Für den weiteren Ausbau der Lösung bieten sich mehrere Entwicklungsschritte an. Ein naheliegender nächster Schritt ist die betriebliche Härtung des Prototyps, etwa durch erweiterte Fehlerbehandlung, Monitoring, Logging und eine robustere Behandlung von Ausnahmefällen im Orchestrierungsablauf.

Darüber hinaus wäre eine Weiterentwicklung in Richtung dynamischer Skalierung sinnvoll. Im vorliegenden Prototyp ist die Anzahl der Runner statisch konfiguriert. Perspektivisch könnte die Bereitstellung bedarfsgesteuert an die aktuelle Joblast angepasst werden. Darüber hinaus wäre eine vertiefte Analyse des Ressourcenverbrauchs unter realistischen Parallelitätsbedingungen von großem Interesse.

Ein weiterer Aspekt des Ausbaus betrifft die Unterstützung zusätzlicher Gastbetriebssysteme, insbesondere Windows-basierter Runner. Im Rahmen der Arbeit wurde die grundsätzliche architektonische Übertragbarkeit betrachtet, jedoch nicht automatisiert umgesetzt.

Über den konkret betrachteten Einsatz von VMware hinaus ist auch eine Übertragbarkeit des Konzepts auf andere Virtualisierungsplattformen denkbar. Die Zielarchitektur basiert auf generischen Mechanismen wie Templates, Snapshots und differenziellen Klonverfahren. Dies setzt jedoch voraus, dass die jeweilige Plattform eine API-basierte Steuerung sowie die klonbasierte Instanziierung virtueller Maschinen unterstützt. In diesem Fall bleibt die grundlegende Struktur der Architektur unverändert. Im Wesentlichen ist lediglich eine Anpassung der plattformspezifischen Anbindung der Orchestrierung erforderlich.

Ein weiterer Aspekt betrifft das derzeit notwendige Wartungsfenster beim Update der Golden VM. Im Prototyp werden bestehende Runner vor der Aktualisierung deregistriert und entfernt, um einen konsistenten Ausgangszustand der Golden VM sicherzustellen.

Im Rahmen einer weiterführenden Untersuchung wäre zu analysieren, inwiefern sich das Wartungsfenster verkürzen lässt. Als mögliche Ansätze wären das Parallelvorhalten von Basisinstanzen oder die Ableitung einer neuen Golden VM aus einer separaten, aktualisierten Kopie zu erwägen. In der Konsequenz könnte die Aktualisierung der Basisumgebung stärker vom laufenden Betrieb entkoppelt werden.

Schließlich bieten sich weiterführende Untersuchungen zur praktischen Eignung und zur langfristigen Einsetzbarkeit der Architektur an. Dies betrifft insbesondere die Bewertung unter erweiterten Betriebsbedingungen sowie die weitere Validierung in einem umfassenderen Anwendungskontext. Die vorliegende Arbeit kann somit als Grundlage für eine Weiterentwicklung über den prototypischen Einsatz hinaus dienen.

Literaturverzeichnis

[1] Nicole Forsgren, Jez Humble und Gene Kim. *Accelerate: Building and Scaling High-Performing Technology Organizations*. Portland, OR: IT Revolution Press, 2018.

[2] GitHub. *About GitHub Enterprise Server*. Zugriff am 20. Feburar 2026. 2026. url: https://docs.github.com/en/enterprise-server@3.17/admin/overview/about-github-enterprise-server.

[3] Broadcom. *VMware vSphere*. Zugrif am 24. Februar 2026. 2026. url: https://techdocs.broadcom.com/us/en/vmware-cis/vsphere.html.

[4] GitHub. *REST API endpoints for self-hosted runners*. Zugrif am 24. Februar 2026. 2026. url: https://docs.github.com/en/rest/actions/self-hosted-runners.

[5] GitHub. *GitHub Actions: Early February 2026 updates*. Zugrif am 20. Februar 2026. 2026. url: https://github.blog/changelog/2026-02-05-github-actions-early-february-2026-updates/.

[6] GitHub. *Workflows*. Zugriff am 20. Feburar 2026. 2026. url: https://docs.github.com/en/actions/concepts/workflows-and-actions/workflows.

[7] GitHub. *Understanding GitHub Actions*. Zugriff am 20. Feburar 2026. 2026. url: https://docs.github.com/en/actions/get-started/understand-github-actions.

[8] GitHub. *GitHub-hosted runners*. Zugriff am 20. Feburar 2026. 2026. url: https://docs.github.com/en/actions/concepts/runners/github-hosted-runners.

[9] GitHub. *Self-hosted runners reference*. Zugriff am 20. Feburar 2026. 2026. url: https://docs.github.com/en/actions/reference/runners/self-hosted-runners.

[10] GitHub. *Self-hosted runners*. Zugriff am 20. Feburar 2026. 2026. url: https://docs.github.com/en/actions/concepts/runners/self-hosted-runners.

[11] GitHub. *Von GitHub gehostete Runner*. Zugriff am 20. Feburar 2026. 2026. url: https://docs.github.com/en/enterprise-server@3.19/actions/concepts/runners/github-hosted-runners.

[12] Proxmox. *VM Templates and Clones*. Zugrif am 24. Februar 2026. 2026. url: https://pve.proxmox.com/wiki/VM_Templates_and_Clones.

[13] Microsoft. *Hyper-V documentation*. Zugrif am 24. Februar 2026. 2026. url: https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/.

[14] Gerald J. Popek und Robert P. Goldberg. *Formal requirements for virtualizable third generation architectures*. Zugrif am 20. Februar 2026. 1974. url: https://doi.org/10.1145/361011.361073.

[15] VMWare. *What is a hypervisor*. Zugrif am 20. Februar 2026. 1974. url: https://www.vmware.com/topics/hypervisor.

[16] Mendel Rosenblum und Tal Garfinkel. *Virtual Machine Monitors: Current Technology and Future Trends*. Zugrif am 20. Februar 2026. 2005. url: https://web.stanford.edu/group/comparch/papers/Computer_RosenblumGarfinkel.pdf.

[17] libvirt. *Virtual Machine Lifecycle*. Zugrif am 20. Februar 2026. 2026. url: https://wiki.libvirt.org/VM_lifecycle.html.

[18] Broadcom. *Virtual Machine Lifecycle*. Zugrif am 20. Februar 2026. 2026. url: https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/8-0/vsphere-virtual-machine-administration/introduction-to-vmware-vsphere-virtual-machinesvsphere-vm-admin/virtual-machine-lifecyclevsphere-vm-admin.html.

[19] Broadcom. *Deploy a Virtual Machine from a Template*. Zugrif am 20. Februar 2026. 2026. url: https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/7-0/vsphere-virtual-machine-administration/deploying-virtual-machinesvm-admin/deploy-a-virtual-machine-from-a-template-h5vm-admin.html.

[20] Broadcom. *Clone a Virtual Machine to a Template*. Zugrif am 20. Februar 2026. 2026. url: https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/7-0/vsphere-virtual-machine-administration/deploying-virtual-machinesvm-admin/clone-a-virtual-machine-to-a-template-h5vm-admin.html.

[21] Broadcom. *Using Snapshots To Manage Virtual Machines*. Zugrif am 20. Februar 2026. 2026. url: https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/7-0/vsphere-virtual-machine-administration/managing-virtual-machinesvm-admin/using-snapshots-to-manage-virtual-machinesvm-admin.html.

[22] Sreekanth Setty. *Understanding Clones in VMware vSphere 7*. Zugrif am 20. Februar 2026. 2026. url: https://www.vmware.com/docs/cloning-vsphere7-perf.

[23] Broadcom. *Advantages of Cloning from a Frozen Virtual Machine*. Zugrif am 20. Februar 2026. 2025. url: https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere-sdks-tools/7-0/web-services-sdk-programming-guide/virtual-machine-management/linked-virtual-machines/instant-clone-virtual-machines/run-state-of-the-instant-clone-source/advantages-of-cloning-from-a-frozen-virtual-machine.html.

[24] Ansible. *Introduction to Ansible*. Zugrif am 25. Februar 2026. 2026. url: https://docs.ansible.com/projects/ansible/latest/getting_started/introduction.html.

[25] Ansible. *Getting started with Ansible*. Zugrif am 25. Februar 2026. 2026. url: https://docs.ansible.com/projects/ansible/latest/getting_started/index.html.

[26] Ansible. *Building an inventory*. Zugrif am 25. Februar 2026. 2026. url: https://docs.ansible.com/projects/ansible/latest/getting_started/get_started_inventory.html.

[27] Ansible. *Installing Ansible*. Zugrif am 25. Februar 2026. 2026. url: https://docs.ansible.com/projects/ansible/latest/installation_guide/intro_installation.html.

[28] Ansible. *Ansible playbooks*. Zugrif am 25. Februar 2026. 2026. url: https://docs.ansible.com/projects/ansible/latest/playbook_guide/playbooks_intro.html.

[29] GitHub. *Dependabot quickstart guide*. Zugrif am 24. Februar 2026. 2026. url: https://docs.github.com/en/enterprise-server@3.17/code-security/tutorials/secure-your-dependencies/dependabot-quickstart-guide.

[30] GitHub. *Repository: Action Runner Controller (ARC)*. Zugrif am 20. Februar 2026. 2026. url: https://github.com/actions/actions-runner-controller.

[31] Broadcom Lam. *Changing Virtual Hard Disk Node and Mode Settings*. Zugrif am 20. Februar 2026. 2025. url: https://techdocs.broadcom.com/us/en/vmware-cis/desktop-hypervisors/workstation-pro/17-0/using-vmware-workstation-pro/configuring-virtual-machine-hardware-settings/configuring-and-maintaining-virtual-hard-disks/changing-virtual-hard-disk-node-and-mode-settings.html.

[32] Sebastián Ramírez. *FastAPI*. Zugrif am 24. Februar 2026. 2026. url: https://fastapi.tiangolo.com/.

[33] Marcelo Trylesinski. *Uvicorn*. Zugrif am 24. Februar 2026. 2026. url: https://uvicorn.dev/.

[34] ASGI Team. *ASGI Documentation*. Zugrif am 24. Februar 2026. 2018. url: https://asgi.readthedocs.io/en/latest/introduction.html.

[35] Luke Heydel. *Action Runner Controller*. Zugrif am 24. Februar 2026. 2026. url: https://github.fh-zwickau.de/luh22mbe/action-runner-controller.

[36] NGINX. *nginx*. Zugrif am 24. Februar 2026. url: https://nginx.org/en/.

[37] Ansible. *Protecting sensitive data with Ansible vault*. Zugrif am 24. Februar 2026. 2026. url: https://docs.ansible.com/projects/ansible/latest/vault_guide/index.html.

[38] Pydantic. *Welcome to Pydantic*. Zugrif am 24. Februar 2026. 2026. url: https://docs.pydantic.dev/latest/.

[39] GitHub. *Troubleshooting the REST API*. Zugrif am 24. Februar 2026. 2026. url: https://docs.github.com/en/enterprise-server@3.17/rest/using-the-rest-api/troubleshooting-the-rest-api#timeouts.

Übersicht verwendeter Hilfsmitte

ChatGPT (OpenAI) wurde in den folgenden Anwendungsfällen unterstützend verwendet:

- zur Unterstützung der Literaturrecherche,
- zur Umwandlung stichpunktartiger Entwürfe in Fließtext,
- zur Unterstützung bei der Formulierung und sprachlichen Straffung,
- zur Überführung von Tabellen in LaTeX-Code,
- zur Erstellung einer Ausgangsbasis für ein Cron-Job-Skript (vgl. D.2),
- zur Unterstützung beim Debugging von Ansible- und Python-Code
- zur Erstellung der Kurzfassung.

DeepL Write (DeepL SE) wurde in den folgenden Anwendungsfällen unterstützend verwendet:

- zur sprachlich-stilistischen Überarbeitung eigener Textentwürfe,
- zur Glättung von Formulierungen,
- zur Anpassung an einen sachlich-wissenschaftlichen Schreibstil.

Anhang

A. Diagramme

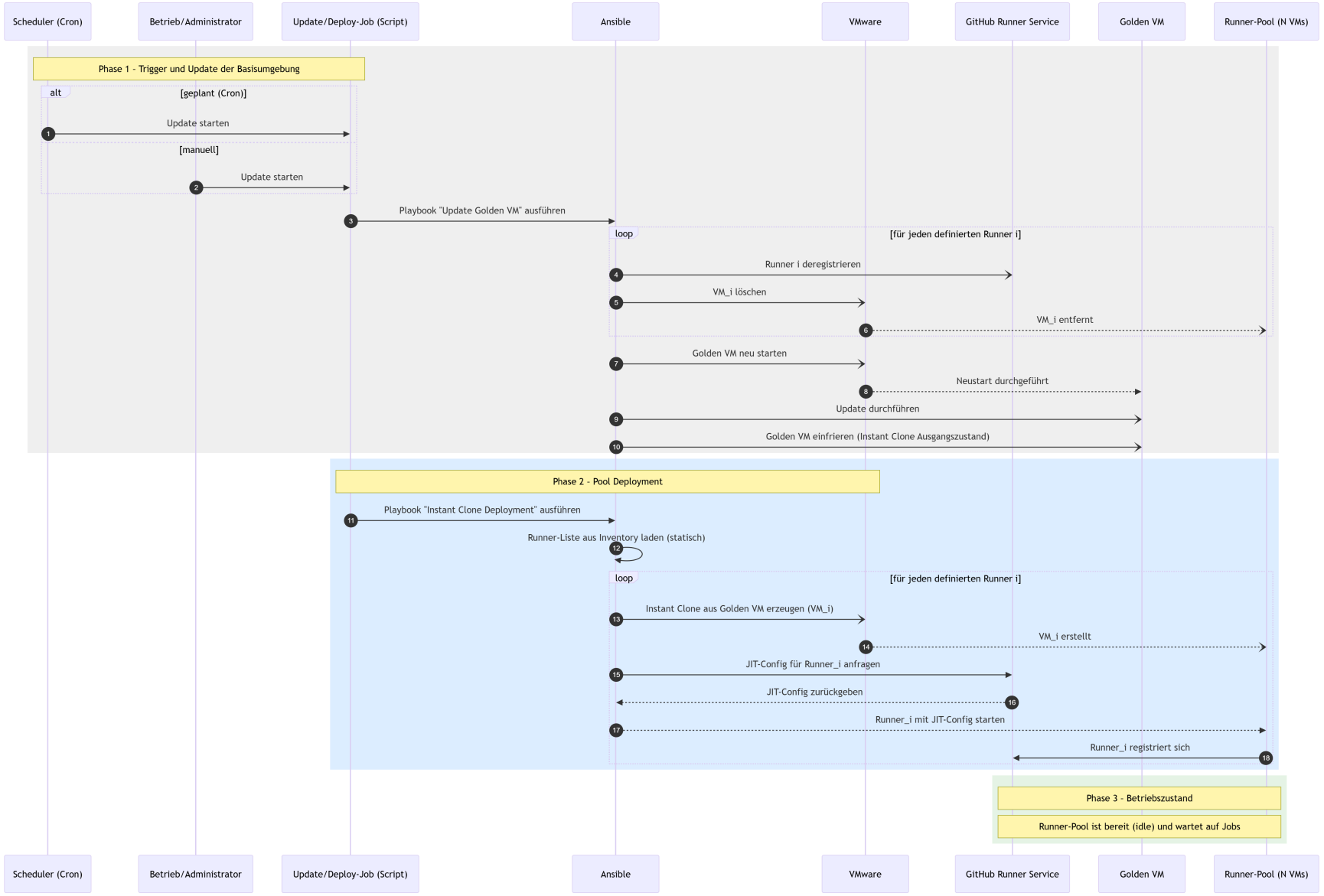

Abbildung A.1.: Sequenzdiagramm des Deployment- und Update-Ablauf

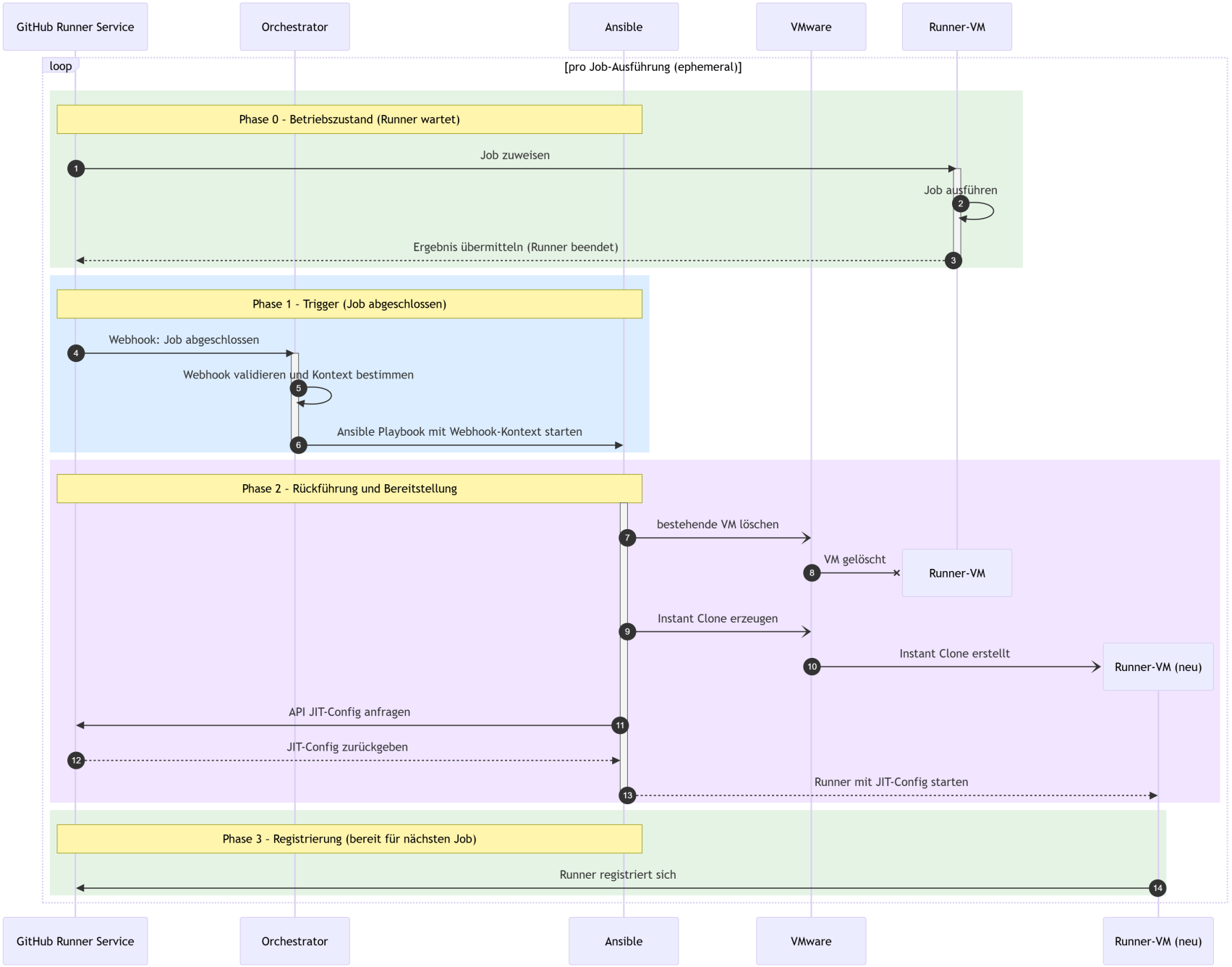

Abbildung A.2.: Sequenzdiagramm des Runner-Lifecycles

B. Ansible Inventory

Listing B.1: Ansible Inventory

```
all:
  children:
    runners:
      children:
        runners_group_1:
          hosts:
            runner-11.example.com:
            runner-12.example.com:
            runner-13.example.com:
        runners_group_2:
            runner-21.example.com
            runner-22.example.com
            runner-33.example.com
    templates:
      hosts:
        golden-vm.example.com:
```

Listing B.2: host_vars für eine Runner-VM

```
registration:
  name: test-runner-01
  runner_group_id: '{{ runner_group }}'
  labels: '{{ labels }}'
hardware_mac_address: 00:50:56:AA:BB:CC
```

Listing B.3: group_vars für eine Runner-Gruppe

```
runner_group: 1
labels:
  - ubuntu
  - single-run
```

C. Ansible Tasks

Listing C.1: GitHub-Remove-Runner

```
- name: API Call to list runners
  ansible.builtin.uri:
    url: "{{ github_runner_endpoint }}?per_page=100"
    method: GET
    headers:
      Accept: "application/vnd.github+json"
      Authorization: "Bearer {{ github_auth_token }}"
      X-GitHub-Api-Version: "2022-11-28"
      Content-Type: "application/json"
    return_content: true
    status_code: 200
    timeout: 10
  register: list_runner_resp

- name: API Call remove runners
  ansible.builtin.uri:
    url: "{{ github_runner_endpoint }}/{{item.id}}"
    method: DELETE
    headers:
      Accept: "application/vnd.github+json"
      Authorization: "Bearer {{ github_auth_token }}"
      X-GitHub-Api-Version: "2022-11-28"
    status_code: 204
    timeout: 10
  loop: "{{ list_runner_resp.json.runners }}"
```

Listing C.2: VM löschen

```
- name: Delete VM
  community.vmware.vmware_guest:
    validate_certs: false
    hostname: "{{ vcenter_hostname }}"
    username: "{{ vcenter_username }}"
    password: "{{ vcenter_password }}"
    datacenter: "{{ vcenter_datacenter }}"
    name: "{{ inventory_hostname }}"

    state: absent # delete VM
    force: true   # delete Powered-On VMs
  delegate_to: localhost
```

Listing C.3: VM neustarten

```
- name: Restart VM
  vmware.vmware.vm_powerstate:
    hostname: "{{ vcenter_hostname }}"
    username: "{{ vcenter_username }}"
    password: "{{ vcenter_password }}"
    datacenter: "{{ vcenter_datacenter }}"
    validate_certs: false
    name: "{{ inventory_hostname }}"

```

```
10      state: restarted
11    delegate_to: localhost
```

Listing C.4: MAC-Adresse setzen

```
1  - name: Add network adapter to VM
2    community.vmware.vmware_guest_network:
3      validate_certs: false
4      hostname: "{{ vcenter_hostname }}"
5      username: "{{ vcenter_username }}"
6      password: "{{ vcenter_password }}"
7      datacenter: "{{ vcenter_datacenter }}"
8      name: "{{ inventory_hostname }}"
9
10     state: present
11     network_name: "{{ hardware_network_vlan }}"
12     mac_address: "{{ hardware_mac_address }}"
13     connected: true
14     start_connected: true
15   delegate_to: localhost
```

Listing C.5: Update Golden VM

```
1  - name: Update
2    ansible.builtin.command:
3      cmd: "apt update"
4
5  - name: Upgrade
6    ansible.builtin.command:
7      cmd: "apt upgrade -y"
```

Listing C.6: Freeze VM

```
1  - name: Freeze
2    ansible.builtin.command:
3      cmd: vmware-rpctool "instantclone.freeze"
```

Listing C.7: Netzwerkadapter entfernen

```
1  - name: Remove network adapter to VM
2    community.vmware.vmware_guest_network:
3      validate_certs: false
4      hostname: "{{ vcenter_hostname }}"
5      username: "{{ vcenter_username }}"
6      password: "{{ vcenter_password }}"
7      datacenter: "{{ vcenter_datacenter }}"
8      name: "{{ inventory_hostname }}"
9
10     state: absent
11     mac_address: "{{ hardware_mac_address }}"
12   delegate_to: localhost
```

Listing C.8: Instant Clone erzeugen

```
1  - name: Create instant clone
2    community.vmware.vmware_guest_instant_clone:
```

```
    validate_certs: false
    hostname: "{{ vcenter_hostname }}"
    username: "{{ vcenter_username }}"
    password: "{{ vcenter_password }}"
    datacenter: "{{ vcenter_datacenter }}"
    datastore: "{{ vcenter_datastore }}"
    name: "{{ inventory_hostname }}"

    folder: ""
    parent_vm: "{{ vcenter_instant_clone_parent }}"
    host: "{{ vcenter_computing_resource }}"
  delegate_to: localhost
```

D. Skripte

Listing D.1: systemd-Dienst

```
[Unit]
Description=Start GitHub Action Runner
[Service]
Type=simple
User=runner
Group=runner
WorkingDirectory=/home/runner/actions-runner
EnvironmentFile=/tmp/runner.env
ExecStart=/bin/bash -lc '/home/runner/actions-runner/run.sh "$START_OPTION"'
Restart=off
[Install]
WantedBy=multi-user.target
```

Listing D.2: Cron-Job

```
#!/usr/bin/env bash
set -euo pipefail

export PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
export ANSIBLE_HOST_KEY_CHECKING="${ANSIBLE_HOST_KEY_CHECKING:-False}"

INV="/app/ansible/inventory"
VAULT_PASS="/app/ansible/.vault_pass"

PB1="/app/ansible/playbooks/vmware-update.yml"
PB2="/app/ansible/playbooks/deploy-instant-clones-runner.yml"

log() { echo "[$(date -Is)] $*"; }

log "Starte Ansible Job"
log "Inventory: ${INV}"

[[ -e "$INV" ]] || { log "FEHLER: Inventory nicht gefunden: $INV"; exit 2; }
[[ -f "$VAULT_PASS" ]] || { log "FEHLER: Vault-Passwortdatei nicht gefunden: $VAULT_PASS"; exit 2; }
[[ -f "$PB1" ]] || { log "FEHLER: Playbook nicht gefunden: $PB1"; exit 2; }
[[ -f "$PB2" ]] || { log "FEHLER: Playbook nicht gefunden: $PB2"; exit 2; }

log "Playbook 1: $PB1"
ansible-playbook -i "$INV" --vault-password-file "$VAULT_PASS" "$PB1"

log "Playbook 2: $PB2"
ansible-playbook -i "$INV" --vault-password-file "$VAULT_PASS" "$PB2"

log "Fertig"
```

E. Tests

Listing E.1: Test 1: Persistenzfreiheit zwischen zwei Job-Ausführungen

```
name: "Test␣1␣-␣Persistenzfreiheit"
on:
  workflow_dispatch:

jobs:
  create_marker:
    name: "Job␣1:␣Marker-Datei␣erstellen"
    runs-on: [self-hosted, ubuntu]
    steps:
      - name: Create marker file
        shell: bash
        run: |
          echo "marker␣created␣at␣$(date␣-Is)" > /tmp/persist_marker.txt
          ls -l /tmp/persist_marker.txt

  verify_absent:
    name: "Job␣2:␣Marker-Datei␣darf␣nicht␣existieren"
    runs-on: [self-hosted, ubuntu]
    needs: [create_marker]
    steps:
      - name: Verify marker file is absent
        shell: bash
        run: |
          if [ -e /tmp/persist_marker.txt ]; then
            echo "ERROR:␣Persistenz␣gefunden:␣/tmp/persist_marker.txt␣existiert␣noch."
            exit 1
          fi
          echo "OK:␣Keine␣Persistenz␣-␣Marker-Datei␣ist␣nicht␣vorhanden."
```

Listing E.2: Test 1: Persistenzfreiheit zwischen zwei Job-Ausführungen

```
name: "Test␣2␣-␣Docker-Artefakte␣dürfen␣nicht␣persistieren"
on:
  workflow_dispatch:

jobs:
  run1_create_docker_state:
    name: "Job␣1:␣Docker-Image/Volume␣erzeugen"
    runs-on: [self-hosted, ubuntu]
    steps:
      - name: Preconditions
        shell: bash
        run: |
          docker --version
          docker info >/dev/null

      - name: Create docker image + volume (must NOT persist)
        shell: bash
        run: |
          # Create a tiny image to detect persistence reliably
          cat > Dockerfile <<'EOF'
          FROM alpine:3.20
```

```
          RUN echo "persist-test" > /hello.txt
          EOF

          docker build -t persist-test:run1 .
          docker volume create persistvol

          # Touch a file inside the volume
          docker run --rm -v persistvol:/data alpine:3.20 sh -c 'echo hi > /data/x'

          echo "Created image and volume:"
          docker images --format '{{.Repository}}:{{.Tag}}' | grep -E '^persist-test:run1$' || true
          docker volume ls --format '{{.Name}}' | grep -E '^persistvol$' || true

  run2_verify_no_docker_persistence:
    name: "Job 2: Docker-Image/Volume darf nicht existieren"
    runs-on: [self-hosted, ubuntu]
    needs: [run1_create_docker_state]
    steps:
      - name: Preconditions
        shell: bash
        run: |
          docker --version
          docker info >/dev/null

      - name: Verify image is absent
        shell: bash
        run: |
          if docker images --format '{{.Repository}}:{{.Tag}}' | grep -q '^persist-test:run1$'; then
            echo "ERROR: Docker image persisted: persist-test:run1"
            docker images | head -n 50
            exit 1
          fi
          echo "OK: Docker image did not persist"

      - name: Verify volume is absent
        shell: bash
        run: |
          if docker volume ls --format '{{.Name}}' | grep -q '^persistvol$'; then
            echo "ERROR: Docker volume persisted: persistvol"
            docker volume ls | head -n 200
            exit 1
          fi
          echo "OK: Docker volume did not persist"
```

Listing E.3: Test 3: Ausführung ohne Root-Rechte

```
name: "Test 3 - Job läuft nicht als Root"
on:
  workflow_dispatch:

jobs:
  non_root_check:
    runs-on: [self-hosted, ubuntu]
    steps:
      - name: Assert non-root
        shell: bash
        run: |
```

```
12          if [ "$(id␣-u)" -eq 0 ]; then
13            echo "ERROR:␣Job␣is␣running␣as␣root."
14            exit 1
15          fi
16          echo "OK:␣Job␣is␣not␣running␣as␣root."
```

F. Ergebnise und Evidenz

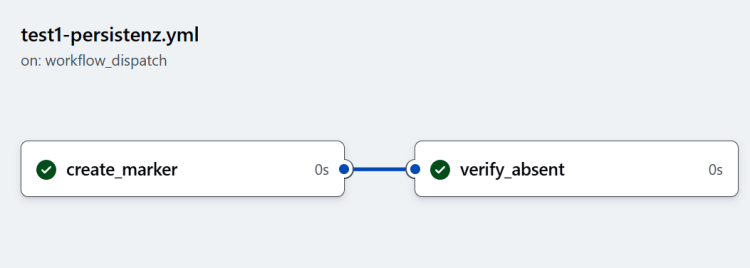

Abbildung F.1.: Workflow-Ergebnis für Test 1

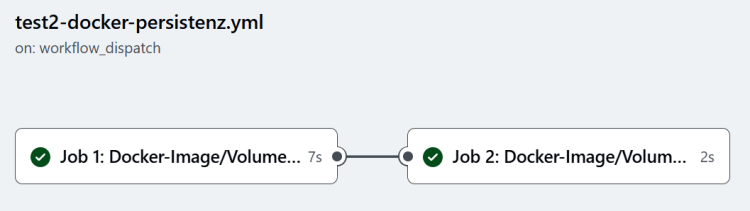

Abbildung F.2.: Workflow-Ergebnis für Test 2

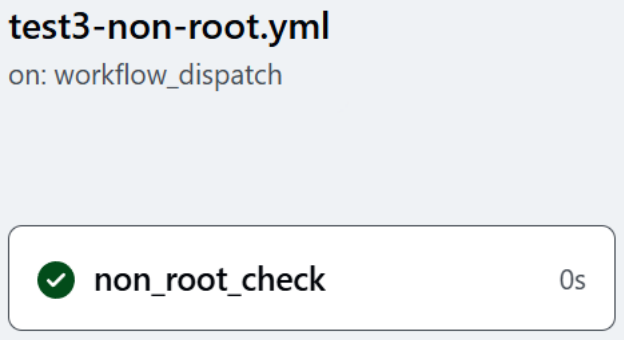

Abbildung F.3.: Workflow-Ergebnis für Test 3

```
PLAY RECAP *********************************************************************
localhost                  : ok=3    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
whz-github-runner-30b.zw.fh-zwickau.de : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
whz-github-runner-31b.zw.fh-zwickau.de : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
whz-github-runner-32b.zw.fh-zwickau.de : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
whz-github-runner-33b.zw.fh-zwickau.de : ok=12   changed=6    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

TASKS RECAP ********************************************************************
Mittwoch 25 Februar 2026  19:58:58 +0100 (0:00:01.100)       0:01:10.786 ******
===============================================================================
Wait for SSH connection ------------------------------------------------ 36.59s
Remove network adapter to VM -------------------------------------------- 9.55s
Update ------------------------------------------------------------------ 9.28s
Delete VM --------------------------------------------------------------- 5.65s
Add network adapter to VM ----------------------------------------------- 2.74s
Restart VM -------------------------------------------------------------- 2.56s
Upgrade ----------------------------------------------------------------- 1.13s
Freeze via VMware Tools ------------------------------------------------- 1.10s
API Call to list runners ------------------------------------------------ 1.02s
API Call remove runners ------------------------------------------------- 0.70s
include_tasks ----------------------------------------------------------- 0.09s
include_tasks ----------------------------------------------------------- 0.06s
include_tasks ----------------------------------------------------------- 0.04s
include_tasks ----------------------------------------------------------- 0.04s
include_tasks ----------------------------------------------------------- 0.03s
include_tasks ----------------------------------------------------------- 0.03s
include_tasks ----------------------------------------------------------- 0.03s
```

Abbildung F.4.: Ansible-Log für das Updaten des Golden Image

G. Messdaten der Rücksetzzeiten

Tabelle G.1.: Rohdaten der Rücksetzzeit für Independent Non-Persistent Disk

Versuch	Power Cycle (s)	SSH (s)	Gesamt (s)
1	4.15	8.41	12.56
2	4.53	8.57	13.10
3	4.32	7.99	12.31
4	4.35	8.02	12.37
5	4.66	8.34	13.00
Mittelwert	4.40	8.27	12.67

Tabelle G.2.: Rohdaten der Rücksetzzeit für Instant Clone

Versuch	VM löschen (s)	Clone erzeugen (s)	Netzwerkadapter setzen (s)	SSH (s)	Gesamt (s)
1	4.24	2.90	2.78	1.48	11.40
2	3.48	3.26	2.41	1.44	10.59
3	3.73	5.24	2.35	1.50	12.82
4	3.96	2.74	2.86	1.51	11.07
5	4.19	3.03	2.95	1.53	11.70
Mittelwert	3.92	3.43	2.67	1.49	11.52

Tabelle G.3.: Rohdaten der Rücksetzzeit für Linked Clone

Versuch	VM löschen (s)	Clone erzeugen (s)	SSH (s)	Gesamt (s)
1	0.71	5.08	79.74	85.53
2	3.41	5.64	78.53	87.58
3	3.78	4.85	80.27	88.90
4	4.22	4.98	80.06	89.26
5	3.76	4.71	80.46	88.93
Mittelwert	3.18	5.05	79.81	88.04

WHZ Westsächsische Hochschule Zwickau
Hochschule für Mobilität

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit in allen Teilen selbstständig angefertigt und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt habe, und dass die Arbeit in gleicher oder ähnlicher Form in noch keiner anderen Prüfung vorgelegen hat. Mir ist bewusst, dass ich Autor/in der vorliegenden Arbeit bin und volle Verantwortung für den Text trage. ☑

Ich erkläre, dass ich wörtlich oder sinngemäß aus anderen Werken – dazu gehören auch Internetquellen – übernommene Inhalte als solche kenntlich gemacht und die entsprechenden Quellen angegeben habe. ☑

Mir ist bewusst, dass meine Arbeit auf Plagiate überprüft werden kann. Mir ist bekannt, dass es sich bei der Abgabe eines Plagiats um ein schweres akademisches Fehlverhalten handelt und dass Täuschungen nach der für mich gültigen Prüfungsordnung geahndet werden. ☑

Zusätzlich versichere ich, dass ich auf künstlicher Intelligenz (KI) basierende Werkzeuge nur in Absprache mit den Prüfern verwendet habe. Dabei stand meine eigene geistige Leistung im Vordergrund, und ich habe jederzeit den Prozess steuernd bearbeitet. ☑

Diese Werkzeuge habe ich im Quellenverzeichnis in der Rubrik „Übersicht verwendeter Hilfsmittel" mit ihrem Produktnamen und einer Übersicht des im Rahmen dieser Prüfungs-/Studienarbeit genutzten Funktionsumfangs unter Angabe der Textstelle in der Arbeit vollständig aufgeführt. ☑

Ich versichere, dass ich keine KI-basierten Tools verwendet habe, deren Nutzung die Prüfer explizit schriftlich ausgeschlossen haben. Ich bin mir bewusst, dass die Verwendung von Texten oder anderen Inhalten und Produkten, die durch KI-basierte Tools generiert wurden, keine Garantie für deren Qualität darstellt. ☑

Ich verantworte die Übernahme jeglicher von mir verwendeter maschinell generierter Passagen vollumfänglich selbst und trage die Verantwortung für eventuell durch die KI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate. ☑

Bockau, 26.02.2026 Luke Heydel

Ort, Datum Unterschrift