Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Entwicklung eines Multiagentensystems für das dezentrale Deployment von Softwarekomponenten

Dominik Wagner

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h.c. Frank Leymann
Betreuer/in:	Dr. Uwe Breitenbücher Miles Stötzner, M.Sc.
Beginn am:	8. März 2022
Beendet am:	30. September 2022

Kurzfassung

Moderne Anwendungen bestehen häufig aus verschiedenen Komponenten, welche auf unterschiedlichen Cloud Plattformen, Internet of Things oder Edge-Geräte und On-Premises Infrastrukturen ausgeführt werden. Weiterhin können Teile dieser Infrastruktur von verschiedenen Organisationen betrieben werden, die aufgrund von Compliance-Regeln und Sicherheitsbedenken oft keinen Zugriff auf Schnittstellen oder Anmeldedaten untereinander freigeben können. Weitere Einschränkungen entstehen durch Sicherheitsbeschränkungen von Private Clouds und der Netzwerkarchitektur, in der sich viele IoT-Geräte befinden, da diese oft keine eingehenden Verbindungen erlauben. Manuelle Softwaredeployments in diesen komplexen Umgebungen sind fehleranfällig, zeitaufwendig und erfordern viel Expertise. Aus diesem Grund wird in dieser Arbeit ein dezentralisiertes auf agentenbasiertes Konzept für automatisierte Deployments in verteilte und heterogene Umgebungen vorgestellt. In diesem Konzept wird die Anwendungstopologie mit TOSCA modelliert und anschließend aufgeteilt. Die einzelnen Teile werden jeweils einem Agenten zugeordnet, der das Deployment teilautonom ausführt. Es werden dabei mehrere Ansätze entwickelt und verglichen, um die Topologie so aufzuteilen, dass einzelne Agenten nur Zugriff auf den für sie relevanten Teil des Modells erhalten. Das beschriebene Konzept ermöglicht durch die Agenten ein organisationsübergreifendes Deployment ohne zentralen Zugang zu allen beteiligten Netzwerken und Infrastrukturen. Um die Koordination und den Informationsaustausch zwischen den Agenten zu ermöglichen, wird eine MQTT-basierte Kommunikationsarchitektur präsentiert, welche auch in Umgebungen mit Sicherheitsbeschränkungen, ohne eingehende Verbindungen funktioniert. Zusätzlich wurde in dieser Arbeit die Kommunikationsarchitektur um ein Sicherheitskonzept erweitert, welches den Schaden, durch eine Kompromittierung von einzelnen Agenten, einschränkt. Dabei lässt sich genau kontrollieren, welcher Agent auf welche Informationen eines anderen Agenten Zugriff hat. Der Lebenszyklus einer Anwendung umfasst nicht nur das Deployment, sondern auch die Entwicklung und Änderungen während der Lebenszeit der Anwendung. Deshalb wird in dieser Arbeit ein GitOps-basierten Workflow präsentiert, der die Zusammenarbeit von verschiedenen Organisationen an einer Anwendung erleichtert. Dieser Workflow erlaubt es, Änderungen einem Review zu unterziehen und diese automatisiert auf ein Deployment anzuwenden. Mit dieser Arbeit ermöglichen wir die Entwicklung und das Deployment von organisationsübergreifenden Multicloud- und IoT-Anwendungen in heterogenen Umgebungen mit verteilten Agenten.

Abstract

Modern applications often consist of multiple components running on various cloud platforms, Internet of things or edge devices, and on-premises infrastructures. Furthermore, parts of this infrastructure may be operated by different organizations, which often cannot grant access to interfaces or credentials among themselves due to compliance rules and security concerns. Further constraints arise from security restrictions of private clouds and the architecture of the networks in which many IoT devices reside, as these often do not allow inbound connections. Manual software deployments in these complex environments are error-prone, time-consuming, and require a great deal of technical expertise. This thesis therefore presents a decentralized agent-based concept for automated software deployments to distributed and heterogeneous environments. In this concept, the application topology is modeled using TOSCA and subsequently partitioned into subgraphs. The individual parts are each assigned to an agent that performs the deployment semi-autonomously. Several approaches are presented and compared to split the topology in such a way that individual agents only have access to the part of the model that is relevant to them. The described concept enables a cross-organizational deployment based on agents without central access to all involved networks and systems. To enable coordination and information exchange between the agents, an MQTT-based communication architecture is presented. This architecture works in environments that do not allow inbound connections. In addition, a security extension to the communication architecture is presented that limits the damage caused by the compromise of individual agents. With this extension, the flow of information between the agents can be controlled via a rule set. The lifecycle of an application includes not only the deployment but also the development and changes during the lifetime of the application. Therefore, this work presents a GitOps-based workflow that facilitates collaboration between different organizations on an application. This workflow allows reviewing changes and applying them automatically to a deployment. With this thesis, we enable the development and deployment of multi-cloud and IoT applications across organizations in heterogeneous environments with distributed agents.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACL Access Control List. 57

API Application Programming Interface. 66

CSAR Cloud Service Archive. 22

EDMM Essential Deployment Meta Model. 28

IoT Internet of Things. 15

MQTT Message Queuing Telemetry Transport. 19

NAT Network Address Translation. 16

PRS Proxy Replacement Subgraph. 43

RPS Reduced Proxy Subgraph. 44

SaaS Software-as-a-Service. 39

TOSCA Topology and Orchestration Specification for Cloud Applications. 15

TSS Total Separated Subgraph. 46

TSSN Total Separated Subgraph with Network. 47

VM Virtuelle Maschine. 21

1 Einleitung

Konzepte wie Internet of Things (IoT), Edge oder Cloud Computing verändern nicht nur wie Software entwickelt, verteilt und betrieben wird, sondern auch das tägliche Leben der Nutzer [AIM10]. Diese Technologien finden sich in vielen Bereichen des Lebens wieder und umfassen unter anderem Smart-Home-Geräte, Connected Cars, intelligente Strom- und Verkehrsnetze oder industrielle Prozesskontrolle [Cis20; SY16].
Um die genannten Beispiele umzusetzen müssen Anwendungen entwickelt und provisioniert werden, welche die Vorteile von IoT, Edge Computing und Cloud Computing miteinander kombinieren. Deshalb bestehen viele Anwendungssysteme aus verschiedenen Komponenten, die auf unterschiedlichsten Geräten und Plattformen betrieben werden, diese reichen von kleinen IoT-Geräten bis zu großen Cloud-Rechenzentren. Dabei können verschiedene Teile dieser Infrastruktur zu unterschiedlichen Organisationen gehören.
Diese Heterogenität in Kombination mit neuen Softwareentwicklungsparadigmen wie GitOps stellt große Herausforderungen an Softwaredeployment, Wartung und Betrieb. Erschwerend kommt hinzu, dass unterschiedliche Geräte und Plattformen diverse Protokolle, Schnittstellen und Standards zum Softwaredeployment und Management verwenden [MCRG15]. Die unterschiedlichen Komponenten des Anwendungssystems können weiterhin lokale und über Netzwerk erreichbare Abhängigkeiten aufweisen [BCS18]. Manche Softwarekomponenten benötigen zum Beispiel eine Ausführungsumgebung wie Java oder eine Datenbank auf einem anderen Gerät. Manuelle Softwaredeployments erfordern daher viel Expertise, sind zeitaufwendig und fehleranfällig [BBK+13; OGP03]. Deshalb kann eine Orchestrierung und Automatisierung des Deployments helfen, Fehler zu vermeiden und Kosten zu sparen [KBL+19].

Eine Orchestrierung bedeutet in diesem Kontext, das Koordinieren und Automatisieren von allen Schritten und Prozessen, die für ein Deployment einer Anwendung bestehend aus mehreren Komponenten in heterogene Umgebungen notwendig sind [OAS20a]. Um das zu ermöglichen, können solche komplexen Multicloud- und IoT-Anwendungen mithilfe der „Topology and Orchestration Specification for Cloud Applications“ (TOSCA) [OAS20b] in Form eines Graphen modelliert werden. Die Komponenten der Anwendung werden als Knoten und die Abhängigkeiten sowie die Verbindungen als Kanten modelliert. Der Graph wird dann einem deklarativen und portablen Modell gespeichert [BSW14; OAS20b]. Alle zum Deployment benötigten Skripte, Binärdateien und andere Ressourcen lassen sich zusammen mit dem deklarativen Modell in einem Archiv kombinieren.
Ein Orchestrator kann anhand des Modells bestimmen, welche Aufgaben für ein erfolgreiches Deployment ausgeführt werden müssen [BBK+14]. Um dieses Deployment auszuführen, gibt es verschiedene Konzepte. Ein zentraler Orchestrator verbindet sich mit allen beteiligten Ressourcen wie IoT-Geräte, virtuelle Server oder Cloud Plattformen und führt die notwendigen Aufgaben für das Deployment aus [KBL+19]. Dazu werden zum Beispiel Skripte ausgeführt oder Schnittstellen zur Kontrolle von Cloud-Ressourcen genutzt [KBL+19; WBK+20]. In einem dezentralen Deployment werden auf allen beteiligten Ressourcen Agenten installiert [KBL+19]. Die Agenten holen sich die Aufgaben, die für das Deployment notwendig sind, von einem zentralen Manager und führen

diese auf der Ressource aus, auf der sie installiert sind. Ein zentraler Orchestrator ist einfacher zu konfigurieren, weniger komplex und es müssen keine Agenten vorinstalliert werden. Eine Lösung basierend auf einem zentralen Orchestrator kann aber für ein heterogenes organisationsübergreifendes Multicloud- und IoT-Deployment aus mehreren Gründen nicht eingesetzt werden [KBL+19; WBK+20].
Einerseits wird eine Verbindung von einem zentralen Orchestrator zu allen Geräten und Ressourcen benötigt, was aber oft nicht möglich ist [KBL+19]. Ressourcen, welche sich in Private Clouds oder IoT Netzwerken befinden, können häufig aufgrund von Sicherheitsmaßnahmen wie Firewalls oder der Netzwerkarchitektur nicht von außerhalb angesprochen werden [KBL+19; LS18]. Insbesondere Smart-Home-Anwendungen, die über Heimnetzwerke kommunizieren, sind durch die Network Address Translation (NAT) in Routern nicht direkt erreichbar [FSK05; HP11]. Auch außerhalb von Heimnetzwerken gibt es Einschränkungen in der Konnektivität von IoT-Geräten. Mobile Geräte haben zum Beispiel oft eine unzuverlässige Verbindung und eine begrenzte Stromversorgung [VPM+21].
Andererseits benötigt ein zentraler Orchestrator Zugang zu der internen Infrastruktur und zu Anmeldeinformationen von Cloud Plattformen aller beteiligter Organisationen. Aufgrund von Sicherheitsbedenken oder Compliance-Regeln kann dieser Zugang oft nicht gewährt werden.

Um diese Problematiken zu lösen, wird in dieser Masterarbeit ein dezentralisiertes Deploymentkonzept bestehend aus kooperierenden Agenten präsentiert, welche eigenständig Teile des Deployments ausführen können. In diesem dezentralen Ansatz wird eine Agentensoftware auf jeder teilnehmenden Ressource installiert. Die einzelnen Instanzen der Agentensoftware kommunizieren über eine Kommunikationsarchitektur die auf einen eventbasierten Message Broker basiert. Ein Message Broker ist eine Komponente, die verteilten Anwendungen das Austauschen von Nachrichten ermöglicht [LPB+15]. Das hat den Vorteil, dass die lokal installierten Agenten nur eine Verbindung aus ihrem Netzwerk heraus zu dem Message Broker aufbauen müssen. Der Message Broker ermöglicht so Kommunikation zwischen den Agenten trotz Sicherheitsbeschränkungen, welche eingehende Verbindungen blockieren [KBL+19]. Auch erlaubt ein eventbasiertes asynchrones Kommunikationsmodell bei sporadisch vorhandener Internetverbindung zu kommunizieren, da Nachrichten während Unterbrechungen vom Broker gespeichert und später übertragen werden.

Die in dieser Arbeit konzipierten Agenten erlauben es weiterhin teilautonome und asynchrone Deployments auszuführen. Ein Agent kann alle Komponenten einer Software, die lokal provisioniert werden sollen, anhand des TOSCA Modells bestimmen und anschließend die notwendigen Operationen für das Deployment weitgehend autark ausführen. Kommunikation ist dabei nur für die Koordination der Agenten untereinander notwendig. Das reduziert die Netzwerknutzung im Vergleich mit dem zentralisierten Deploymentkonzept, da nicht jede einzelne Operation über das Netzwerk gesteuert werden muss. Durch das dezentrale Konzept können unabhängige Teile des TOSCA Modells lokal autonom provisioniert werden. Diese Autonomie erlaubt einen geringeren Stromverbrauch, da Datenübertragungen per Funk mehr Energie benötigen können als lokale autonome Datenverarbeitung [PK00].

Darüber hinaus ermöglicht das in dieser Arbeit beschriebene Konzept ein organisationsübergreifendes Deployment, ohne dass die beteiligten Organisationen Anmeldedaten offenlegen müssen. Es entfällt auch die Anforderung, einen externen Zugang zu internen Schnittstellen oder interner Infrastruktur für einen zentralen Orchestrator bereitzustellen. Geräte und Instanzen melden sich durch die Agenten nach dem Start bei dem Message Broker und werden nicht von einem zentralen Orchestrator instantiiert oder gestartet. Die beteiligten Organisationen behalten die Kontrolle über

ihre interne Infrastruktur und sind eigenständig für das Starten von Ressourcen wie virtuelle Maschinen oder das Bereitstellen von IoT-Geräten verantwortlich.
Cloud Computing ermöglicht durch Skalierung dynamisch auf geänderte Workloads zu reagieren [MG11]. Deshalb bietet das in dieser Arbeit entwickelte Konzept die Möglichkeit, dass Organisationen ein bestehendes Deployment um neue Instanzen erweitern oder nicht mehr benötigte stoppen können.

Um die organisationsübergreifende Zusammenarbeit für das gemeinsame Erstellen, Verändern und Provisionieren eines Anwendungsmodells zu verbessern, wird in dieser Arbeit ein GitOps-basierter Workflow erarbeitet. Hierbei wird das Modell und alle Artefakte in einer Versionsverwaltung an einem zentralen Ort gespeichert. Das reduziert die Chancen für einen menschlichen Fehler, erlaubt durch die Versionsverwaltung das Zurückkehren in einen älteren Zustand und hinterlässt automatisch einen Audit Trail [Wea22]. Der Workflow sieht die Verwendung einer Plattform vor, welche für Veränderungen einen Review-Prozess bietet, das Erstellen von Issues ermöglicht und automatisiert bei Änderungen ein Deployment ausführen kann.

Um das Deployment einer in TOSCA modellierten Anwendung über ein verteiltes dezentralisiertes Agentensystem zu ermöglichen, muss die TOSCA Spezifikation erweitert werden. Eine Instanz eines Agenten muss wissen, welcher Teil der Gesamtanwendung von dem Agenten lokal provisioniert werden soll. Deshalb wird im Verlauf dieser Arbeit ein Konzept entwickelt, um die Zuordnung eines Subgraph einer Anwendungstopologie zu einem Agenten in einem TOSCA Modell zu repräsentieren. Ein Subgraph ist ein Teil der Anwendungstopologie, die provisioniert werden soll.

In dieser Arbeit werden weiterhin mehrere Ansätze entwickelt und verglichen, um ein TOSCA Modell in mehrere dieser Subgraphen aufzuteilen. Dadurch muss nicht das gesamte TOSCA Modell auf jeden Agenten übermittelt werden, sondern nur der Subgraph, welcher dem Agenten zugewiesen ist. Das reduziert die notwendige Bandbreite zur Übertragung. Weiterhin enthält ein TOSCA Modell sensible Informationen über die gesamte Infrastruktur einer Anwendung, daher sollte ein einzelner Agent nur Zugang zu den Teilen des Gesamtmodells erhalten, welche er für das lokale Deployment benötigt. So werden bei der Kompromittierung eines Agenten nicht alle im TOSCA Modell enthaltenen Informationen preisgegeben. Insbesondere IoT-Geräte sind hier besonders bedroht, da sie sich häufig an exponierten, öffentlich zugänglichen Standorten befinden.
Kompromittierte Agenten könnten jedoch nicht nur auf den lokalen Subgraph des Agenten zugreifen, sondern auch die Kommunikationsarchitektur nutzen, um Informationen aus anderen Teilen der Topologie zu erhalten. Deshalb präsentiert diese Arbeit ein Sicherheitskonzept, das die Kommunikationsarchitektur erweitert. Das Sicherheitskonzept erlaubt eine genaue Kontrolle, welcher Agent Zugriff auf welche Informationen erhält. Es basiert dabei auf Regeln für den Nachrichtenaustausch in Verbindung mit einer TOSCA Erweiterung, welche die Sichtbarkeit von Informationen direkt im TOSCA Modell kontrolliert.

Um die Machbarkeit der präsentierten Konzepte zu validieren wird ein Prototyp mit einem Anwendungsszenario konzipiert und entwickelt.

Die Ziele dieser Arbeit werden in Abschnitt 1.1 nochmals zusammengefasst dargestellt.

1.1 Ziele

Um die Herausforderungen des Deployments von Multicloud- und IoT-Anwendungen in heterogenen und organisationsübergreifenden Umgebungen zu erfüllen, hat diese Arbeit folgenden Ziele:

1. Entwicklung eines agentenbasierten Deploymentkonzepts, bei dem Organisationen ihre Infrastruktur selbst kontrollieren und nur wenige Informationen über diese offenlegen müssen.
2. Entwicklung eines Konzepts, um lokale Agenten in TOSCA Modelle zu integriert.
3. Entwicklung eines Konzepts um ein TOSCA Modell in Subgraphen für unterschiedliche Agenten aufzuteilen, sodass einzelne Agenten nur Zugriff auf den für sie relevanten Teil des Modells erhalten und die Subgraphen möglichst autonom provisioniert werden können.
4. Konzeption und Entwicklung einer Agentensoftware um verteilte TOSCA-basierte Softwaredeployments zu ermöglichen.
5. Entwicklung eines eventbasierten Kommunikationsmodells, um Softwaredeployments auf Agenten mit sporadischer Konnektivität und Beschränkungen durch Sicherheitsmaßnahmen zu ermöglichen.
6. Entwicklung eines GitOps-basierten Workflows, um die Zusammenarbeit an einem zentralen TOSCA Modell für alle Teilnehmer zu erleichtern.

1.2 Gliederung

Kapitel 1 – Einleitung: Die Einleitung befasst sich mit der Motivation und den Zielen dieser Arbeit.

Kapitel 2 – Grundlagen: In diesem Kapitel werden die Grundlagen zu den verwendeten Konzepten und Technologien erklärt. Es beinhaltet unter anderem eine Einführung in die TOSCA Spezifikation und Related Work.

Kapitel 3 – Konzept für die Entwicklung eines Multiagentensystems: Dieses Kapitel umfasst unterschiedliche Konzepte, um die in Abschnitt 1.1 formulierten Ziele zu erreichen. Hierfür wird ein Ansatz, um Agenten in TOSCA zu repräsentieren, vorgestellt. Weiterhin werden mehrere Konzepte für die Aufteilung eines TOSCA Modells in mehrere agentenspezifische Subgraphen erarbeitet und verglichen. Weiterhin enthält dieser Abschnitt ein Konzept für die Kommunikation der Agenten untereinander.

Kapitel 4 – Implementierung des Prototyps: Dieser Teil der Arbeit setzt sich mit dem Konzept und der Implementierung des Prototyps auseinander. Er enthält eine Architekturbeschreibung des Prototyps und ein Anwendungsszenario, um die praktische Umsetzbarkeit der in Kapitel 3 erarbeiteten Konzepte zu validieren.

Kapitel 5 – Zusammenfassung und Ausblick: Hier werden die Ergebnisse der Arbeit zusammengefasst und Anknüpfungspunkte für zukünftige Arbeiten vorgestellt.

2 Grundlagen

In diesem Kapitel werden die Grundlagen dieser Arbeit und verwandte Arbeiten beschrieben. Abschnitt 2.1 beschreibt die Grundlagen von Cloud Computing und die unterschiedlichen Cloud Deploymentmodelle wie Private Cloud oder Public Cloud. In Abschnitt 2.2 wird TOSCA näher beschrieben und erklärt. Das zur Kommunikation der Agenten untereinander verwendete Kommunikationsprotokoll „Message Queuing Telemetry Transport“ (MQTT) wird in Abschnitt 2.3 näher beschreiben. Abschnitt 2.4 umfasst eine Beschreibung von GitOps. Am Ende dieses Kapitels in Abschnitt 2.5 werden verwandte Arbeiten vorgestellt.

2.1 Cloud Computing

Cloud Computing bezeichnet laut dem National Institute of Standard and Technologies (NIST)[1] ein Modell, um Kunden den Zugriff auf geteilte Computing-Ressourcen und Services On-Demand zu ermöglichen [AFG+10; MG11]. Die Ressourcen umfassen zum Beispiel Server, Speicherplatz, Netzwerke, Plattformen und Anwendungen, welche dem Kunden über ein Netzwerk zur Verfügung stehen [AFG+10; MG11]. Der Kunde kann die gewünschten Ressourcen schnell skalieren und an seine Anforderungen anpassen [HYA+15]. NIST hat hierbei die folgenden fünf essenziellen Eigenschaften von Cloud Computing identifiziert: „On-demand self-service“, „Broad network access“, „Resource pooling“, „Rapid elasticity“ und „Measured service“. „On-demand self-service“ beschreibt die Möglichkeit eines Kunden, die gewünschten Ressourcen zeitnah und ohne menschliche Interaktion anzupassen [MG11]. So kann ein Kunde Ressourcen kurzzeitig buchen und sie wieder freigeben, sobald diese nicht mehr benötigt werden [AFG+10]. Die freigegebenen Ressourcen gehen dabei in einen geteilten Ressourcenpool zurück, den sich viele Kunden durch „Resource pooling“ teilen [MG11]. Ein Cloud-Provider bietet seine Ressourcen in einem Pool an, aus dem viele Kunden Ressourcen buchen und wieder freigeben können [MG11]. Als Kunde ist dabei nicht ersichtlich an welchem Standort sich die Ressourcen befinden [MG11]. Manche Cloud-Provider bieten aber die Möglichkeit an Einschränkungen zum Beispiel auf eine geografische Region oder ein Rechenzentrum vorzunehmen [MG11]. Die Eigenschaft „Rapid elasticity“ beschreibt, dass Ressourcen schnell und dynamisch provisioniert und wieder freigegeben werden können [MG11]. In manchen Fällen werden die Ressourcen für Sekunden gebucht und anschließend wieder freigegeben. Diese Skalierung kann automatisch oder manuell erfolgen und ermöglicht es Kunden auf kurzzeitig auftretende Lastspitzen zu reagieren [AFG+10; MG11]. Kunden können zum Beispiel ihrem Webshop während einer erfolgreichen Werbeaktion dynamisch mehr Server, Bandbreite oder Speicherplatz aus einem endlos erscheinenden Ressourcenpool zur Verfügung stellen [MG11]. In einem eigenen Rechenzentrum müsste der Kunde für seinen Webshop die benötigten Ressourcen im Voraus kaufen, welche dann außerhalb der Werbeaktion ungenutzt bleiben. Die Eigenschaft

[1] https://www.nist.gov/ (zuletzt besucht 28.09.2022)

„Measured service“ bedeutet, dass die Ressourcennutzung innerhalb der Cloud genau überwacht wird und dem Kunden eine transparente Nutzungskontrolle erlaubt [MG11]. „Broad network access“ beschreibt, dass Kunden die Ressourcen und Dienste der Cloud über eine Netzwerkverbindung nutzen können [MG11].

Neben den Vorteilen und Chancen, die Cloud Computing bietet, gibt es für Kunden auch Nachteile. Beim Cloud Computing werden Daten, Anwendungen und möglicherweise sensiblen Informationen einem externen Anbieter anvertraut [AFG+10]. Das Bedrohungsmodell für in der Cloud umgesetzte Dienste umfasst auch anderen Kunden des Cloud-Providers, da beim Cloud Computing Ressourcen von verschiedenen Kunden mit unterschiedlichen Interessen geteilt werden. Dies bedeutet, dass Kunden auch vor andere Kunden geschützt werden müssen [AFG+10]. Neben Sicherheitsbedenken gibt es bei Cloud-Providern auch einen Vendor- oder Daten Lock-in [AFG+10]. Ein Vendor Lock-in bedeutet, dass ein Kunde von einem Anbieter abhängig ist und sich ein Wechsel zu einem anderen Anbieter wirtschaftlich nicht lohnt [OST16]. Ein Wechsel ist aufwendig, da unterschiedliche Cloud-Provider verschiedene Dienste und Ressourcen anbieten. Selbst Dienste mit ähnlicher Funktionalität haben unterschiedliche Schnittstellen zur Interaktion [AFG+10]. Deshalb kann es aufwendig sein, Daten und Anwendungen von einem Cloud-Provider zu einem anderen umzuziehen. Auch beim Softwaredeployment gibt es erhebliche Unterschiede zwischen den Anbietern [WBF+19]. TOSCA hat das Ziel den Vendor Lock-in zu reduzieren und ein Vendor neutrales Ökosystem zu erschaffen [OAS20b]. In Abschnitt 2.2 wird TOSCA näher beschrieben.

2.1.1 Cloud Deploymentmodelle

Es gibt unterschiedliche Cloud Deploymentmodelle, die sich zum Beispiel in Kosten, Anzahl der Nutzer, Betreiber der Cloud und Sicherheit unterscheiden. Im Folgenden werden die von Mell und Grance [MG11] beschrieben Modelle näher betrachtet. „Public Clouds“ sind Clouds, die der Öffentlichkeit oder einer großen Gruppe zugänglich sind [MG11]. Im Gegensatz dazu beschreibt „Private Cloud“ Infrastruktur oder Dienste, die nur einer einzigen Organisation zur Verfügung stehen [AFG+10]. Diese können auf eigener Infrastruktur oder von einem Anbieter betrieben werden [MG11]. Diese Clouds profitieren nicht so stark durch Skaleneffekte wie „Public Clouds“, wenn diese nicht eine ähnliche Größe erreichen [AFG+10]. „Private Clouds“ eignen sich primär bei konsistenter Ressourcenauslastung oder wenn aufgrund von Datenschutz oder Sicherheitsaspekten „Public Clouds“ nicht genutzt werden können [AFG+10; Goy14]. Eine „Community Cloud“ steht einer bestimmten Gruppe von Kunden zur Verfügung, die gemeinsame Interessen hat [MG11]. Solche Gruppen können unter anderem Behörden einer Regierung oder Unternehmen im Gesundheitssektor sein, die gemeinsame Anforderungen an den Datenschutz und die Sicherheit haben. Bei „Hybrid Cloud“ handelt es sich um die Kombination aus mehreren Cloud-Infrastrukturen [MG11]. Die Clouds sind über eine Schnittstelle miteinander verbunden [MG11]. Ein Anwendungsfall ist datenschutzkritische Teile einer Anwendung in einer „Private Cloud“ zu betreiben, während nicht kritische Aufgaben in einer „Public Cloud“ erledigt werden [Goy14].

2.2 Topology and Orchestration Specification for Cloud Applications (TOSCA)

Die „Topology and Orchestration Specification for Cloud Applications“ (TOSCA) ist ein OASIS[2] Standard, der erstmals 2013 in der Version 1.0 veröffentlicht wurde [PS13]. TOSCA hat das Ziel komplexe Multicloud-Anwendungen in einem portablen Format zu modellieren, ihr Deployment zu automatisieren und ihren Lebenszyklus zu verwalten [OAS20b]. Im Jahr 2020 wurde der Entwurf der Version 2.0 veröffentlicht [OAS20c]. Die Version ist YAML[3]-basiert und legt ihren Fokus nicht nur auf das Modellieren von Cloud-Anwendungen, sondern auch auf die Bereiche Function-as-a-Service, Virtualized Network Functions, IoT- und Edge-Computing [OAS20b].
Anwendungen werden in TOSCA als Service Templates modelliert. Ein Service Template enthält ein Topology Template, das die einzelnen Komponenten der zu modellierenden Anwendung in einem typisierten Graph abbildet. Ein Node Template ist ein Knoten in diesem Graph und repräsentiert jeweils eine Komponente der Anwendung. Die Beziehungen der Knoten werden über Relationship Templates repräsentiert. Node und Relationship Templates sind Instanzen von Node oder Relationship Types. Die Typen definieren die Semantik der Templates über zum Beispiel Properties, Attributes und Interfaces. Über Properties können die Templates konfiguriert werden. Sie repräsentieren den gewünschten Zustand eines Templates bei seiner Instantiierung. Attributes stellen den aktuellen Zustand einer Komponente dar und werden meist während des Deployments von Operationen gesetzt. Attributes und Properties können auch Funktionen enthalten, welche anderen Attributes und Properties von Node und Relationship Templates lesen können. Für Node Templates können Requirements und Capabilities definiert werden. Node Templates mit passenden Requirements und Capabilities können über Relationship Templates miteinander verbunden werden. Interfaces bestehen aus Operationen, welche von Implementation Artifacts implementiert werden. Implementation Artifacts sind zum Beispiel Skripte, Java Anwendungen oder Binärdateien. Jede Art von Implementierung kann verwendet werden, wenn diese vom eingesetzten Orchestrator unterstützt wird. [OAS20b]

Die Abbildung 2.1 zeigt den Aufbau einer Webanwendung. Im Beispiel sind unter anderem „Frontend“, „Backend“ oder „WebServer“ Node Templates. Beim Deployment werden aus diesen Templates die Instanzen der Anwendungskomponenten erzeugt. Die Node Types der Node Templates sind in der jeweiligen zweiten Zeile erkennbar. So ist zum Beispiel der WebServer vom Node Type „ApacheServer“[4]. Die gerichteten Verbindungen sind Relationship Templates und zeigen die Abhängigkeiten der einzelnen Komponenten. So sind unter anderem das Frontend und der WebServer über ein Relationship Template vom Type „hostedOn“ miteinander verbunden. Das bedeutet, dass das Frontend auf dem WebServer gehostet wird. Des Weiteren repräsentiert das Relationship Template vom Type „connectsTo“ zwischen Frontend und Backend, dass eine Verbindung zwischen diesen zwei Node Templates aufgebaut wird. Alle hier dargestellten Komponenten laufen auf einer virtuellen Maschine (VM), die im unteren Teil der Abbildung dargestellt ist. Jedes Node Template kann Properties, Attributes und Operationen in Interfaces besitzen. Diese werden in Abbildung 2.1 aus Gründen der Übersichtlichkeit nicht dargestellt. So kann in diesem Beispiel ein Property namens „Port“ für die Datenbank definiert sein, welches festlegt auf welchem Port die Datenbank erreichbar

[2] https://www.oasis-open.org/ (zuletzt besucht 28.09.2022)
[3] https://yaml.org/ (zuletzt besucht 28.09.2022)
[4] https://httpd.apache.org/ (zuletzt besucht 28.09.2022)

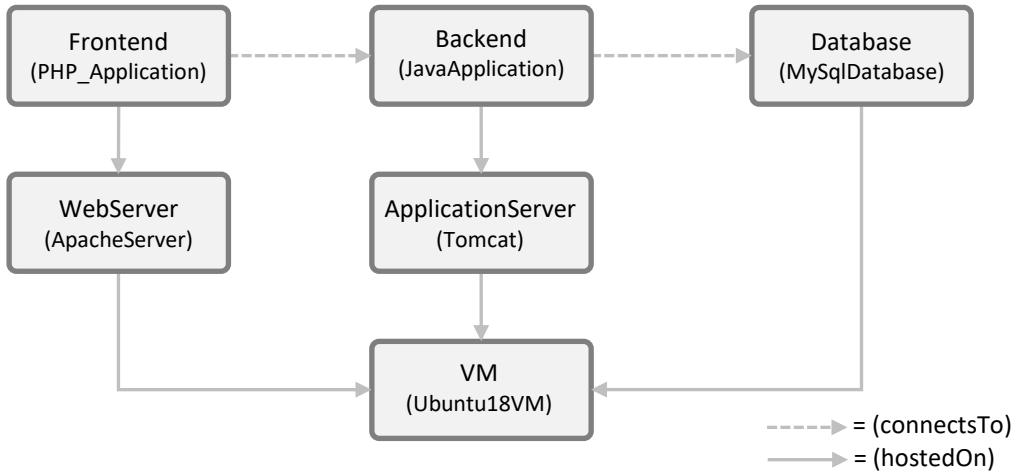

Abbildung 2.1: Vereinfachtes TOSCA Beispiel einer Webanwendung. [Notation basiert auf Vino4TOSCA BBK+12]

sein soll. Properties und Attributes können in Operationen als Inputs verwendet werden. In dem Beispiel könnte das connectsTo Relationship Template welches Backend und Database verbindet eine Operation definieren, welche das Port Property liest, um die Verbindung aufzubauen. Das Service Template und alle weiteren Artefakte können zu einem Cloud Service Archive (CSAR) zusammengefasst werden. Dieses Archiv kann von einem Orchestrator interpretiert und provisioniert werden. [OAS20b]

2.2.1 TOSCA Orchestrator

Ein TOSCA Orchestrator hat die Aufgabe ein CSAR zu interpretieren, die enthaltene Anwendung zu provisionieren und während ihres Lebenszyklus zu managen [OAS20b]. Der Orchestrator entpackt dafür das Archiv und interpretiert das enthaltene Service Template [OAS20b].
Um eine Anwendung zu provisionieren gibt es zwei unterschiedliche Vorgehensweisen [WBK+20]. Endres et al. [EBF+17] beschreiben den deklarativen und den imperativen Ansatz. Beim deklarativen Ansatz wird der gewünschte Zustand und die Struktur der Anwendung beschreiben. Der Orchestrator bestimmt dann die notwendigen Schritte, um diesen Zustand zu erreichen. Deklarative Modelle sind generell intuitiver zu erstellen und zu verstehen. Beim imperativen Ansatz werden alle Schritte und deren Reihenfolge, die während des Deployments ausgeführt werden müssen, zum Beispiel in einem Skript oder Workflow beschrieben. Imperative Modelle erlauben mehr Kontrolle über die Schritte, die ausgeführt werden und sind anpassbarer. Jedoch sind imperative Modelle typisch komplexer zu erstellen. Sie eignen sich zum Beispiel für komplexe Deployments oder wenn mehrere Deployment-Systeme kombiniert werden müssen. In TOSCA kann die Struktur der Anwendung deklarativ im Topology Template beschrieben werden. Der imperative Ansatz wird über Workflows realisiert. [EBF+17; WBK+20]

Da deklarative Modelle intuitiver und einfacher zu verstehen sind, wird der in dieser Arbeit entwickelte Orchestrator diesen Ansatz unterstützen [WBK+20]. Dabei werden die für das Deployment notwendigen Schritte anhand der beschriebenen Struktur der Anwendung bestimmt [OAS20b]. Weiterhin wird dieses Verfahren in der TOSCA Spezifikation als die häufiger verwendete Option

beschrieben [OAS20b]. Um die notwendigen Schritte für das Gesamt-Deployment zu berechnen, muss bekannt sein welche Operationen für das Deployment jedes einzelnen Node oder Relationship Templates ausgeführt werden müssen. Der aktuelle Entwurf der TOSCA 2.0 Spezifikation enthält kein normativ definiertes Interface, um den Lebenszyklus eines Node oder Relationship Templates zu verwalten [OAS20b]. In einigen Beispielen aus dem Entwurf wird das in den vorherigen TOSCA Versionen definierte Lifecycle Interface verwendet [OAS20a; OAS20b]. Deshalb wird in dieser Arbeit das in der TOSCA 1.3 Spezifikation [OAS20a] definierte Lifecycle Interface genutzt, um Node und Relationship Templates zu provisionieren. Das Interface besteht aus den Operationen *create*, *configure*, *start*, *stop* und *delete*. Um ein Node oder Relationship Template zu provisionieren werden nacheinander die Operationen *create*, *configure* und *start* aufgerufen [OAS20a]. Node Templates können aber nicht in beliebiger Reihenfolge provisioniert werden, da diese Abhängigkeiten untereinander haben können. In Abbildung 2.1 benötigt das Frontend den WebServer, um ausgeführt werden zu können. Hier gibt das hostedOn Relationship Template eine Deploymentreihenfolge vor [BBK+14]. Zunächst muss der Orchestrator den WebServer, dann das hostedOn Relationship Template und anschließend das Frontend provisionieren. Für die Verbindung zwischen Frontend und Backend funktioniert diese Reihenfolge nicht. Ein Relationship Template vom Type connectsTo repräsentiert zum Beispiel eine HTTP-Verbindung. Diese Verbindung kann nicht gestartet werden, bevor nicht Quelle und Ziel provisioniert wurden [BBK+14]. Breitenbücher et al. [BBK+14] beschreiben ein topologisches Sortierverfahren für Node und Relationship Templates, das die Beziehungen der Node Templates nutzt, um eine Deploymentreihenfolge anhand der Anwendungstopologie zu bestimmen. Ist die Deploymentreihenfolge bestimmt, führt der Orchestrator nacheinander für alle Node und Relationship Templates die Operationen des Lifecycle Interfaces aus.

2.3 Message Queuing Telemetry Transport (MQTT) Protokoll

Das „Message Queuing Telemetry Transport“ (MQTT) Protokoll ist ein von OASIS[2] entwickelter Standard zur Nachrichtenübermittlung [CBB+19]. Das Protokoll verwendet eine als Broker oder Server bezeichnete Komponente, zu der sich alle Clients verbinden. Der Broker empfängt alle gesendeten Nachrichten und verteilt diese an die Empfänger. MQTT arbeitet nach dem „publish/subscribe“ Message Pattern mit einem hierarchisch gegliederten Topic-Baum [CBB+19]. Bei diesem Verfahren können Empfänger ihr Interesse an bestimmten Nachrichten beim Broker registrieren [EFGK03]. Wird eine passende Nachricht versendet, leitet der Broker die Nachricht an alle interessierten Empfänger weiter [EFGK03]. Um Nachrichten interessierten Empfängern zuordnen zu können, werden Topics verwendet. Ein Topic ist ein String, der von Empfängern beim Registrieren angeben wird. Wird eine Nachricht versendet, enthält diese Nachricht ebenfalls diesen String und der Broker kann die Nachricht den interessierten Empfängern zuordnen [CBB+19]. Topics können beim Registrieren mit einem Schrägstrich in unterschiedliche Level aufgeteilt werden [CBB+19]. Darüber hinaus können verschiedene Wildcardzeichen genutzt werden, um sich für ein oder mehrere Level zu registrieren [CBB+19]. Es gibt das Single-Level Wildcardzeichen „+“ und das Multi-Level Wildcardzeichen „#“ [CBB+19]. Registriert sich ein Empfänger zum Beispiel auf den Topic „zuhause/stockwerk2/+“, funktioniert das „+“ Zeichen wie ein Platzhalter für ein Level an dieser Stelle. Der Empfänger erhält zum Beispiel alle Nachrichten auf dem Topic „zuhause/stockwerk2/badezimmer“, aber keine von „zuhause/stockwerk2/badezimmer/temperaturSensor“. Das Rautezeichen erlaubt das Ersetzen von beliebig vielen Topic Levels [CBB+19].
Für die Nachrichtenübertragung werden drei unterschiedliche Quality of Service Levels angeboten

[CBB+19]. „At most once“ sorgt dafür, dass Nachrichten maximal einmal übertragen werden [CBB+19]. Dabei können Nachrichten verloren gehen. „At least once“ ermöglicht, dass keine Nachrichten verloren gehen, aber Duplikate auftreten können [CBB+19]. Das „Exactly once“ Level ermöglicht, dass keine Nachrichten verloren gehen und Duplikate verhindert werden [CBB+19]. Das Protokoll erlaubt neben dem Nachrichteninhalt auch benutzerdefinierte Schlüsselwertpaare, um Metadaten in einer Nachricht zu übertragen [CBB+19].
Durch die Verwendung dieses Protokolls entsteht eine Entkoppelung der Sender und Empfänger [CBB+19]. Sender und Empfänger müssen sich nicht kennen und benötigen auch keine direkte Verbindung zueinander [EFGK03]. Es genügt die Verwendung eines gemeinsamen Topics, um Nachrichten austauschen zu können. Weiterhin besteht eine zeitliche Entkoppelung, da Sender und Empfänger nicht gleichzeitig kommunizieren müssen [EFGK03]. So kann zum Beispiel ein Empfänger offline sein, während eine Nachricht für ihn gesendet wird. Der Broker stellt dem Empfänger die Nachricht zu, wenn er wieder online kommt [EFGK03].

2.4 GitOps

GitOps bezeichnet ein von Weaveworks[5] beschriebenes Continuous-Deployment Modell, um mithilfe von Git[6] Software auf Kubernetes[7] Cluster zu betreiben [BH21; Ric17a; Wea22]. Der Name GitOps leitet sich von den Worten Git und Operations ab. Es hat das Ziel, Entwickler den Betrieb (Operations) und das Provisionieren von Software mit bekannten Werkzeugen zu ermöglichen [Ric17b]. Dabei wird Git als sogenannte einzige „Source of Truth“ [Ric17a] verwendet, um den gewünschten Zustand des Systems zu beschreiben [Wea18].
Durch Git erhält man Versionskontrolle, Versionshistorie und Code Review Möglichkeiten für ein Kubernetes Cluster [Ric17a]. Richardson [Ric17a] beschreibt, dass alle verwendeten Ressourcen deklarativ modelliert in einem einzigen Git-Repository gespeichert werden. Durch *Infrastructure as Code* (IaC) kann die gesamte Infrastruktur und Software in Git verwaltet werden. Dabei werden neben dem Quellcode auch zum Beispiel Kubernetes Manifeste in Git gespeichert.
Wird eine Änderung über Pull Requests zu Git hinzugefügt, sorgt eine Softwarekomponente automatisch dafür, dass die Änderungen von Build und Release Pipelines gebaut und auf das Kubernetes Cluster angewendet werden [Ric17a]. Entstehen Probleme durch die Änderungen, sind Rollbacks zu älteren Versionen durch Git jederzeit möglich. Treten Abweichungen zwischen dem in Git beschriebenen Istzustand und dem im Cluster provisionierten Sollzustand auf, benachrichtigt das System die Nutzer beispielsweise über E-Mail- oder Slackbenachrichtigungen [Ric17a]. Um Abweichungen zu erkennen stellt Richardson [Ric17a] mehrere Werkzeuge vor. Abweichungen können zum Beispiel entstehen, wenn manuell Änderungen an der Konfiguration im Kubernetes Cluster vorgenommen werden [Ric17a]. Solche manuellen Änderungen sorgen dafür, dass nicht mehr nachvollzogen werden kann, wie der aktuelle Zustand im Cluster erreicht wurde.
Abbildung 2.2 zeigt den Prozess anhand einer Codeänderung die provisioniert wird. Das Anwendungsrepository enthält den Quelltext der Anwendung und das Konfigurationsrepository enthält alle Konfiguration, die für das Deployment benötigt wird. Alle Anwendungen werden in GitOps als unveränderliche Container entwickelt [Ric17b; Wea]. Ein Container enthält eine Anwendung und

[5] https://www.weave.works/ (zuletzt besucht 28.09.2022)
[6] https://git-scm.com/ (zuletzt besucht 28.09.2022)
[7] https://kubernetes.io/ (zuletzt besucht 28.09.2022)

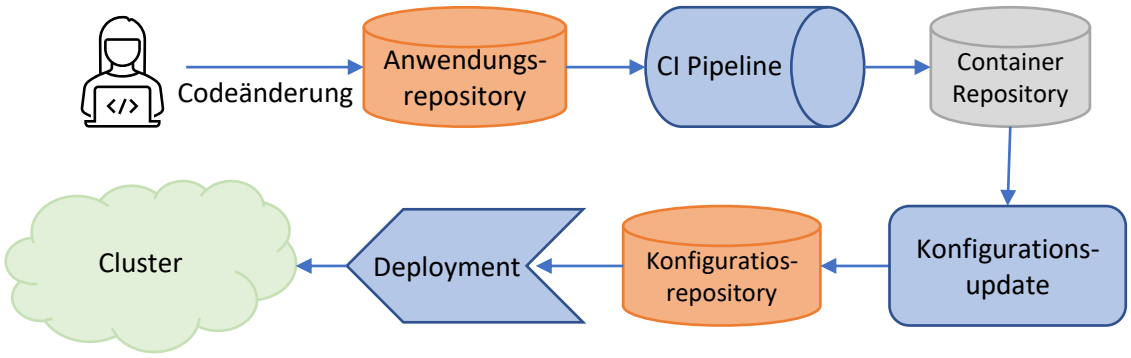

Abbildung 2.2: Vereinfachtes Beispiel einer Codeänderung mit anschließendem Deployment im GitOps Prozess. [Basiert auf Wea]

alle ihre Abhängigkeiten in einem ausführbaren Format [Mer14]. Das Konfigurationsrepository enthält zum Beispiel die Kubernetes Manifeste [Wea]. In der Konfiguration wird auf die Container in dem Container Repository verwiesen.
Die Änderung an einer Anwendung wird von einem Entwickler über einen Pull Request vorgeschlagen. Der Pull Request kann einem Review unterzogen oder direkt gemerged werden [Wea]. Sind die Änderungen in das Anwendungsrepository aufgenommen, baut eine Continuous Integration Pipeline die Anwendungen und checkt die neuen Container in das Container Repository ein [Wea]. Durch Bereitstellen einer neuen Version einer Anwendung im Container Repository wird automatisiert eine neue Konfiguration im Konfigurationsrepository erstellt [Wea]. Die neue Konfiguration beinhaltet eine Referenz auf die neue Version des aktualisierten Containers. Änderungen am Konfigurationsrepository lösen automatisiert ein neues Deployment aus [Wea]. Dafür wird die neue Konfiguration über ein Synchronisationswerkzeug auf das Kubernetes Cluster angewendet [Wea]. Werden neue Komponenten hinzugefügt oder die Struktur der Anwendung verändert, muss ein Entwickler die Konfiguration manuell anpassen.
Weaveworks [Wea] beschreibt vier Prinzipien, die für GitOps erforderlich sind. Diese Prinzipien werden zur Entwicklung des in Abschnitt 3.5 beschriebenen GitOps-basierten Workflows für TOSCA verwendet. Das erste Prinzip beschreibt, dass das komplette System deklarativ beschreiben sein muss [Wea]. Dazu zählt nicht nur die Kubernetes Konfiguration, sondern auch die verwendete virtuelle Infrastruktur [Ric17b]. Das zweite Prinzip nimmt die deklarative Systembeschreibung und speichert diese in einem Git-Repository [Ric17b]. Dadurch sind einfache Rollbacks zu älteren Zuständen möglich. Das dritte von Weaveworks [Wea] beschriebene Prinzip verlangt, dass Änderungen am Zustand in Git automatisiert angewendet werden. Direkt wenn eine neue Version einer Softwarekomponente entwickelt wurde, wird diese provisioniert. Das vierte Prinzip beschreibt, dass Agenten den provisionierten Systemzustand überprüfen und bei Abweichungen von der in Git gespeicherten Konfiguration automatisch den Benutzer benachrichtigen [Ric17b].

2.5 Related Work

Die Abschnitte 2.5.1 bis 2.5.5 beschreiben mehrere verwandte Arbeiten genauer. Die erste Arbeit beschäftigt sich mit verteilten Deployments in heterogene Umgebungen und der Problematik von Sicherheitsbeschränkungen, welche eingehende Verbindungen verhindern. Wild et al. [WBK+20] legen den Fokus auf organisationsübergreifendes Deployment. Wurster et al. [WBB+21] nutzen einen zentralisierten Ansatz und entwickeln in Ihrer Arbeit ein Konzept, um TOSCA Modelle in technologiespezifische Subgraphen aufzuteilen, um diese mit unterschiedlichen Deploymenttechnologien zu provisionieren. Saatkamp et al. [SBKL20] formalisieren mehrere Algorithmen, die Topologie Modelle verarbeiten und aufteilen. Die letzte Arbeit von Panarello et al. [PBL+17] beschreibt eine Methode, um Anwendungen in zusammenarbeitende Clouds zu provisionieren.
Abschnitt 2.5.6 gibt einen kurzen Überblick über eine Auswahl an weiteren Arbeiten, die entweder einen zentralisierten Orchestrator nutzen oder nicht TOSCA-basiert sind. In Abschnitt 2.5.7 werden weitere relevante TOSCA Orchestratoren vorgestellt.

2.5.1 Verteiltes Anwendungsdeployment in öffentliche und private Netzwerke

Képes et al. [KBL+19] beschreiben in Ihrer Arbeit ein Konzept für automatisierte Softwaredeployments in heterogenen Umgebungen bestehend aus öffentlichen und privaten Netzwerken. Diese Umgebungen bestehen aus Cloud-Instanzen, Edge- und IoT-Geräten, welche sich in unterschiedlichen Netzwerken befinden können. Die Autoren argumentieren, dass bestehende Lösungen basierend auf zentralen Orchestratoren nicht eingesetzt werden können, da diese eine direkte Verbindung zu jedem Geräte benötigen, um Schnittstellen wie SSH zu nutzen. IoT-Geräte für zum Beispiel Smart-Homes sind oft durch Firewalls, welche eingehende Verbindungen blockieren, geschützt und können daher nicht von zentralen Orchestratoren erreicht werden.
Aus diesem Grund wird in der Arbeit ein hybrider Ansatz zwischen einem zentralisierten Orchestrator und verteilten Agenten präsentiert. Dieser Ansatz bietet den Vorteil, Kommunikation trotz Firewalls zu ermöglichen und nicht auf jedem Gerät ein Agenten zu benötigen. In diesem hybriden Ansatz gibt es in jedem Netzwerk, in dem Ressourcen provisioniert werden, ein Deployment System das lokal Operationen ausführen kann. Die Deployment Systeme, in den einzelnen Netzwerken, können trotz Firewalls, Verbindungen von innerhalb der Netzwerke zu einem Message Broker aufbauen. Dieser Message Broker wird zur Kommunikation zwischen den Deployment Systemen eingesetzt.
Im Ansatz von Képes et al. [KBL+19] werden Anwendungen deklarativ mit TOSCA modelliert und anhand dieses Modells werden die für das Deployment auszuführenden Operationen bestimmt. Die Arbeit erweitert das bestehenden Deploymentkonzept von einem zentralen Orchestrator um die Deployment Systeme, die sich in jedem Netzwerk befinden und einem Verfahren zur Bestimmung der korrekten Ausführungsumgebung für jede Operation des Deployments. Das Verfahren ermittelt dazu das Netzwerk der zu provisionierenden Ressource und führt dann die Operation auf dem Deployment System im Netzwerk der Ressource aus.

Um dieses Konzept zu ermöglichen, müssen die verwendeten Geräte oder Instanzen in den einzelnen Netzwerken vor dem Deployment beim lokalen Deployment System mit Anmeldedaten und IP- oder MAC-Adresse registriert werden. Es können zum Beispiel Geräte oder virtuelle Maschinen registriert werden. Wird ein Deployment begonnen, übernimmt das startende Deployment System zeitweise die Rolle des koordinierenden Deployment Systems. Dieses bestimmt zuerst einen

Deploymentplan anhand des deklarativen Deploymentmodells und dem von Breitenbücher et al. [BBK+14] beschrieben Verfahrens. Dieser Plan enthält für jede Komponente alle Operationen, die für ein Deployment ausgeführt werden müssen. Um während des Deployments eine dieser Operationen ausführen zu können, muss die Ausführungsumgebung für die Operation bestimmt werden. Dazu wird für jede Komponente einer Anwendung die Infrastrukturkomponente identifiziert, auf der sie ausgeführt wird.
Die Infrastrukturkomponente ist die unterste Komponente in jedem Anwendungsstack. Képes et al. [KBL+19] beschreiben in Ihrer Arbeit ein Verfahren, um diese Infrastrukturkomponente für jede Komponente anhand der hostedOn Verbindungen zu identifizieren. In Abbildung 2.1 ist die VM die Infrastrukturkomponente für die anderen Komponenten. Anschließend führt jedes Deployment System anhand der registrierten Daten über die verwendeten Geräte ein Matching-Verfahren aus. Das Matching-Verfahren überprüft, ob die Infrastrukturkomponente zu einem lokalen Gerät passt. Hat ein Deployment System ein Match, führt es die Operation lokal aus. Die Operationen nutzen dann zum Beispiel Netzwerkschnittstellen, um die Komponente auf dem Gerät oder der Instanz zu installieren. Durch dieses Verfahren ist es möglich für jede Komponente alle für das Deployment notwendigen Operationen im lokalen Netzwerk auszuführen. Dadurch behindert keine Firewall die Kommunikation zwischen Deployment System und Zielgerät oder Zielinstanz, da sich das jeweilige Deployment System im lokalen Netzwerk des Zielgeräts befindet. Zur Validierung der Machbarkeit Ihrer Arbeit haben Képes et al. [KBL+19] ihr Konzept in einem auf dem OpenTOSCA-Container[8]-basierten Prototypen implementiert. Dieser Prototyp wurde dann für das Deployment eines Smart-Home-Szenarios verwendet.
Das in der Arbeit von Képes et al. [KBL+19] vorgestellte hybride Deploymentverfahren hat den Vorteil, Deployments zu ermöglichen, bei denen sich die Geräte und Instanzen in unterschiedlichen Netzwerken befinden. Durch die Deployment Systeme in jedem Netzwerk ist es möglich zu Geräten oder Instanzen eine Verbindung aufzubauen und Operationen auszuführen, obwohl jedes Netzwerk durch eine Firewall geschützt ist. Weiterhin können Geräte mit stark limitierten Hardwareressourcen eingesetzt werden, da nicht auf jedem Gerät, sondern nur auf einem Gerät im Netzwerk ein Deployment System benötigt wird.
Jedoch kann jedes Deployment System nur einzelne Operationen ausführen und ist für die Gesamtkoordination von einem zentralen koordinierenden Deployment System abhängig. Dadurch ist keine Autonomie der einzelnen Deployment Systeme im Deploymentprozess möglich. Weiterhin haben einzelne Geräte keine Autonomie, da jede Operation von einem Deployment System über eine Netzwerkverbindung auf dem Gerät ausgeführt und koordiniert wird. Das benötigt mehr Netzwerkkommunikation als lokale teilautonome Deployments auf jedem Gerät. Auch müssen beim hybriden Ansatz Zugangsdaten zu Clouds oder interner Infrastruktur im Deploymentmodell hinterlegt und damit geteilt werden. Das ist aufgrund von Compliance-Richtlinien und Sicherheitsbedenken in einem organisationsübergreifenden Deployment problematisch [WBK+20]. Auch ist es bei mobilen Geräten möglich, dass diese während der Ausführung einer Operation das Netzwerk wechseln und somit nicht mehr für das lokale Deployment System erreichbar sind.

[8] https://opentosca.github.io/container/ (zuletzt besucht 28.09.2022)

2.5.2 Dezentralisiertes organisationsübergreifendes Anwendungsdeployment

Wild et al. [WBK+20] präsentieren ein Konzept, um Anwendungen automatisiert über Organisationsgrenzen hinweg zu provisionieren, ohne dass die Teilnehmer Schnittstellen zu ihrer Infrastruktur von außerhalb zugänglich machen müssen. Sie argumentieren, dass die verbreiteten zentralen Orchestratoren nicht für organisationsübergreifende Deployments geeignet sind, da der zentrale Orchestrator Zugriff auf Zugangsdaten für Cloud-Provider und zu der internen Infrastruktur der beteiligten Organisationen benötigt. Das Teilen von Zugangsdaten oder ein Zugriff auf die interne Infrastruktur von außerhalb ist aufgrund von Sicherheitsbedenken und Compliance-Richtlinien oft nicht möglich. Der Ansatz von Wild et al. [WBK+20], um dieses Problem zu lösen, beginnt mit einem deklarativen Modell der *Application-Specific Components*.
Diese werden mit dem deklarativen „Essential Deployment Meta Model“ (EDMM), das in einer vorausgegangenen Arbeit entwickelt wurde, modelliert [WBF+19]. Das EDMM nutzt ähnlich wie das in Abschnitt 2.2 beschriebene TOSCA eine graphbasierte Struktur aus *Components* und *Relations* [WBB+21; WBF+19]. Die Gesamtanwendung kann mit EDMM modelliert werden, indem logische, funktionale oder physikalische Komponenten der Anwendung als Components modelliert werden [WBB+21; WBF+19]. Die Abhängigkeiten und die Verbindungen zwischen den Components werden mit Relations modelliert. Über Operationen können Components und Relations installiert oder gestoppt werde [WBF+19]. *Properties* repräsentieren den gewünschten Zustand, den eine Component erreichen soll und ermöglichen, Inputs für den Deploymentprozess zu definieren [WBF+19]. Eine ausführliche Beschreibung von EDMM befindet sich in der Arbeit von Wurster et al. [WBF+19].
Der von Wild et al. [WBK+20] beschriebene Ansatz umfasst fünf Schritte für ein organisationsübergreifendes Deployment. Zuerst wird ein deklaratives Modell der Anwendung ohne konkrete Hosting-Umgebung modelliert. Das Modell enthält alle *Application-Specific Components* einer Anwendung, die für die Realisierung des Use Cases notwendig sind, wie zum Beispiel Business-, Speicher- oder Kommunikations-Components und alle Informationen, welche zur Zusammenarbeit dieser benötigt werden. In Schritt zwei wird jedem Teilnehmer ein Subgraph dieses Modells über Annotationen zugewiesen. Weiterhin werden in dem Anwendungsmodell Placeholder-Components für die Hosting-Umgebung eingefügt. Die Placeholder-Components enthalten nur minimalen Informationen, die zwischen den Organisationen ausgetauscht werden müssen. Schritt drei umfasst die Verarbeitung des Gesamtmodells durch jeden Teilnehmer. Jeder Teilnehmer verarbeitet das Gesamtmodell und ersetzt in seinem individuellen Subgraph alle Placeholder-Components durch konkrete Middleware- und Infrastruktur-Components. Diese Ersetzung kann automatisiert über ein Requirement-Capability Matching-Verfahren ausgeführt werden. Das partiell ersetzte Modell bezeichnen Wild et al. [WBK+20] als *local multi-participant deployment model* (LDM). In Schritt vier berechnet jeder Teilnehmer basierend auf seinem individuellen LDM einen lokalen Workflow, der alle lokalen Components instantiiert. Der Workflow muss alle lokalen Komponenten in der richtigen Reihenfolge instanziieren und alle Input-Parameter jeder Operation bereitstellen. Die Input-Parameter können dabei aus nicht lokalen Teilen des Modells kommen und sind teilweise erst während des Deployments bekannt. Diese Parameter müssen daher während des Deployments von einem anderen Teilnehmer übertragen werden. Gleichzeitig müssen lokale Outputs von Operationen anderen Teilnehmern während des Deployments zur Verfügung gestellt werden. Weiterhin muss das Gesamt-Deployment implizit durch die einzelnen LDMs koordiniert werden. Wild et al. [WBK+20] beschreiben in Ihrer Arbeit ein Verfahren, um diese lokalen Workflows für jeden Teilnehmer zu generieren und notwendige Aktivitäten zur Kommunikation automatisch zu integrieren. Das Deployment kann in Schritt fünf von einem Teilnehmer initiiert werden und die lokalen Workflows

werden innerhalb jeder Organisation von einem Orchestrator ausgeführt. Zur Validierung wurde ein OpenTosca[17]-basierter Prototyp entwickelt.
Da jeder Teilnehmer selbst die konkreten Informationen über seine Hosting-Umgebung in sein LDM integriert, bietet der Ansatz von Wild et al. [WBK+20] den Vorteil, dass nur wenig Informationen geteilt werden müssen. Nur Informationen, die für Verbindungen, zwischen den Components benötigt werden, müssen ausgetauscht werden. Weiterhin kommt der Ansatz ohne einen zentralen Orchestrator für alle Organisationen aus. Die für jeden Teilnehmer generierten Workflows müssen trotzdem von einem organisationsinternen Orchestrator ausgeführt werden. Einzelne Geräte und Instanzen haben keine Autonomie von dem jeweiligen organisationsinternen Orchestrator. Dadurch ist mehr Kommunikation zwischen dem Orchestrator und jedem Gerät notwendig, was sich bei batteriebetriebenen und über Funk verbundenen Geräten negativ auf die Lebensdauer der Batterie auswirkt [PK00]. Auch muss der organisationsinterne Orchestrator eine direkte Verbindung zu jedem Gerät aufbauen können. Geräte können jedoch innerhalb einer Organisation durch eine Firewall, welche eingehende Verbindungen blockiert, geschützt sein. Auch können zum Beispiel Smart-Home-Anwendungen in Kundennetzwerken, welche durch NAT keine eingehende Verbindungen erlauben, nicht erreicht werden. Weiterhin muss jeder Teilnehmer im Vorfeld alle möglichen Placeholder-Ersetzungen für alle auftretenden Requirements definieren, um das Requirement-Capability Matching-Verfahren automatisiert ausführen zu können.

2.5.3 Deploymentautomatisierung von verteilten Anwendungen durch die Kombination von mehrere Deploymenttechnologien

Wurster et al. [WBB+21] beschreiben in Ihrer Arbeit, dass oft mehrere Deploymenttechnologien benötigt werden, um ein Deployment auszuführen. Unterschiedliche Technologien setzen unterschiedliche Schwerpunkte und haben jeweils Vor- und Nachteile. Für das Provisionieren von Amazon Web Services[9] (AWS) Ressourcen eignet sich zum Beispiel CloudFormation[10] [WBB+21]. Im Gegensatz dazu fokussiert sich Chef[11] auf die Installation und Konfiguration von Anwendungen auf bereits provisionierter Infrastruktur. Komplexe Anwendungen bestehen oft aus mehreren Komponenten, die auf unterschiedlichen Infrastrukturen betrieben werden. Ein Deployment benötigt daher oft verschiedene Deploymenttechnologien. Anwendungsteile in verschiedenen Deploymenttechnologien zu modellieren, ist aufwendig. Diese separat zu provisionieren, ist fehleranfällig und teuer. Weiterhin wird Expertise für jede Deploymenttechnologien benötigt. Um diese Probleme zu lösen, wird in der Arbeit die Forschungsfrage gestellt: „How to seamlessly model and automate the deployment of a complex application distributed across heterogeneous environments that requires different deployment technologies?“ [WBB+21, S. 2].
Das Konzept der Arbeit besteht aus sechs Schritten, um ein Deployment mit unterschiedlichen Deploymenttechnologien auszuführen. Als Erstes wird die Gesamtanwendung in dem technologieneutralen von Wurster et al. [WBF+19] beschriebenen EDMM modelliert. Eine Kurzbeschreibung des EDMM befindet sich in Abschnitt 2.5.2 und eine ausführliche Beschreibung in der Arbeit von Wurster et al. [WBF+19]. Im zweiten Schritt wird jeder Teil des EDMM-Modells mit der gewünschten Deploymenttechnologie annotiert. Dazu haben Wurster et al. [WBB+21] eine Erweiterung der EDMM-Syntax entwickelt, die es erlaubt jeder Component eine spezifische Deploymenttechnologien

[9] https://aws.amazon.com/de/ (zuletzt besucht 28.09.2022)
[10] https://aws.amazon.com/de/cloudformation/ (zuletzt besucht 28.09.2022)
[11] https://www.chef.io/ (zuletzt besucht 28.09.2022)

zuzuweisen. Das EDMM-Modell besteht nach dem Annotieren aus mehreren Gruppen von Components, die jeweils mit einer Deploymenttechnologie provisioniert werden. So kann zum Beispiel ein Teil des Modells mit CloudFormation und ein anderer mit Chef annotiert sein. Um ein Deployment auszuführen, muss das Gesamtmodell der Anwendung in Schritt drei in mehrere technologiespezifische Gruppen aufgeteilt werden. Da zwischen den verschiedenen technologiespezifischen Gruppen zyklische Abhängigkeiten bestehen können, ist es nicht immer möglich alle Components, die einer Deploymenttechnologie zugeordnet sind, auf einmal zu provisionieren. Daher haben Wurster et al. [WBF+19] einen auf der Arbeit von Saatkamp et al. [SBKL20] basierenden Algorithmus entwickelt, der die minimale Anzahl an Deploymentgruppen bestimmt, sodass keine zyklischen Abhängigkeiten entstehen. Eine Deploymentgruppe ist dabei eine Menge von Components, die von einer Ausführung der ihnen zugeordneten Deploymenttechnologie provisioniert werden kann. Diese Deploymentgruppen werden in Schritt vier von dem technologieneutralen EDMM-Modell in die spezifische Deploymenttechnologie transformiert. Dafür erweitert die Arbeit das von Wurster et al. [WBB+20] vorgestellte *EDMM Transformation Framework*. Die Erweiterung ermöglicht, dass Konfigurationsparameter anhand des EDMM-Modell aufgelöst und dynamisch zur Laufzeit als In- und Outputs abgebildet werden. Zum Beispiel können IP-Adressen dynamisch während des Deployments vergeben werden, deshalb müssen diese dann von einem technologiespezifischen Teil des Deployments in einen anderen übertragen werden, um eine Verbindung zu ermöglichen. In Schritt fünf wird die Deploymentreihenfolge der Deploymentgruppen bestimmt. Um das zu ermöglichen wird in der Arbeit ein Verfahren basierenden auf den Arbeiten von Breitenbücher et al. [BBK+14] und Saatkamp et al. [SBKL20] vorgestellt, um die Deploymentgruppen so zu sortieren, dass die Abhängigkeiten zwischen den Gruppen sowie ihre In- und Outputs berücksichtigt werden. Um das Deployment in Schritt sechs ausführen und koordinieren zu können, stellen Wurster et al. [WBB+21] eine Systemarchitektur vor.
Der von Wurster et al. [WBB+21] beschriebene Ansatz hat den Vorteil, unterschiedliche Deploymenttechnologien einsetzen zu können. Um organisationsübergreifende Deployments in heterogene Umgebungen umzusetzen, kann das vorteilhaft sein, da unterschiedliche Organisationen verschiedene Deploymenttechnologien benötigen könnten. Der vorgestellte Ansatz basiert auf einem zentralen Orchestrator, der das Deployment koordiniert. Daher kann keine Software auf Geräte provisioniert werden, die sich hinter einer Firewall befinden. Auch müssen alle benötigten Zugangsdaten im EDMM-Modell enthalten sein und alle am Deployment beteiligten Organisationen müssen Schnittstellen für ihre interne Infrastruktur für den zentralen Orchestrator öffnen.

2.5.4 Formalisierungen und Algorithmen für Topologie Modelle von verteilten Cloud Deployments

Saatkamp et al. [SBKL20] beschreiben in ihrer Arbeit mehrere Algorithmen, um topologische Deploymentmodelle wie TOSCA in verschiedene Teile aufzuteilen. Zentrale Orchestratoren oder Provisioning Engines benötigen Zugang zu Low-Level Schnittstellen von allen an einem Deployment beteiligen Cloud-Providern, um das Deployment auszuführen. Um das zu vermeiden, präsentieren Saatkamp et al. [SBKL20] in ihrer Arbeit eine Methode, um Deploymentmodelle in verschiedene Teile aufzuteilen. Cloud-Provider können die einzelnen Teile separat ausführen und müssen weniger oder keine Schnittstellen für eine Provisioning Engine nach außen öffnen.
Um ein Deploymentmodelle zu erstellen, es in mehrere Teile aufzuteilen und in einem koordiniertem Deployment auszuführen, beschreiben Saatkamp et al. [SBKL20] ein Verfahren basierend auf acht Schritten. Die ersten fünf Schritte basieren auf der früheren Arbeit [SBKL17] und sind

ebenfalls in der Arbeit [SBKL20] beschrieben. In Schritt eins wird ein Topology Modell bestehend aus den Components und Relations einer Anwendung erstellt. Ein solches Modell enthält zum Beispiel Application-Specific Components, *Middleware Components* wie Webserver oder *Infrastructure Components* wie zum Beispiel eine virtuelle Maschine. Schritt zwei umfasst das Hinzufügen von Labels zu mindestens allen Application-Specific Components. Jedes Label definiert die Zielumgebung, wie zum Beispiel ein Cloud-Provider, auf der die Component provisioniert wird. Schritt drei überprüft ob, eine über die Label definierte Aufteilung möglich ist. Zum Beispiel müssen alle Application-Specific Components mit einem Label versehen sein. Weiterhin kann keine Component auf einer Component, die einem anderen Provider zugeordnet ist, gehostet sein. In Schritt vier wird das Gesamtmodell basierend auf den Labels in mehrere Teile aufgeteilt. Sind zwei Application-Specific Components unterschiedlichen Providern zugeordnet, müssen zum Beispiel Middleware Components auf denen sie im Gesamtmodell gemeinsam gehostet werden, für beide Provider dupliziert werden. In Schritt fünf ersetzt ein Algorithmus in jedem Teil des Modells die Infrastructure oder Middleware Components durch Provider spezifische Components. So kann zum Beispiel ein Webserver im Modell durch AWS Elastic Beanstalk[12] ersetzt werden. Die Schritte sechs bis acht sind neu und noch nicht von vorausgegangenen Arbeiten beschreiben.
Schritt sechs berechnet die Reihenfolge in der die Provider ihre Teile des Modells provisionieren. Dafür werden Gruppen von Components erstellt, die jeweils einem Label zugeordnet sind und auf einmal provisioniert werden können. Die Reihenfolge, in der die Gruppen provisioniert werden, hängt von den Verbindungen zwischen Components die unterschiedlichen Providern zugeordnet sind ab. Wenn eine Verbindung aufgebaut werden soll, muss erst das Ziel der Verbindung und dann die Quelle provisioniert werden. Dadurch muss die Gruppe mit dem Ziel der Verbindung zuerst provisioniert werden. Eine Gruppe kann nicht immer alle Components von einem Provider umfassen, da unter anderem sonst zyklische Abhängigkeiten zwischen den Gruppen entstehen können. In Schritt sieben wird für jede Gruppe ein Deploymentmodell erstellt. Das Modell umfasst alle Components in der Gruppe, alle Ziele einer providerübergreifenden Verbindung und die der Ziel-Component zugeordneten Infrastructure oder Middleware Components. Im letzten Schritt werden die Deploymentmodelle der Gruppen zu den jeweiligen Providern übertragen und in der in Schritt sechs bestimmten Reihenfolge ausgeführt. Dabei müssen auch Inputs, die erst während des Deployments bestimmt werden können, wie IP-Adressen zwischen den Providern ausgetauscht werden.
Saatkamp et al. [SBKL20] formalisieren diese Schritte anhand eines topologischen Meta-Modells und beschreiben für Schritt drei bis sieben jeweils einen Algorithmus in Pseudocode. Weiterhin implementieren die Autoren einen Prototyp, basierend auf dem grafischen Modellierungswerkzeug Winery[13].
Der vorgestellte Ansatz erlaubt ein verteiltes Deployment einer Anwendung mit mehreren Orchestratoren in unterschiedlichen Organisationen. Weiterhin erlaubt dieser Ansatz, dass Organisationen gemeinsam ein Deployment ausführen können, ohne Schnittstellen nach außen zu öffnen. Jedoch enthält die Arbeit keinen Ansatz, um das Deployment automatisiert auszuführen. Zum Beispiel werden Inputs manuell zwischen den Organisationen ausgetauscht. Weiterhin gibt es einen zentralen Orchestrator pro Organisation, daher müssen alle Geräte und Instanzen innerhalb dieser Organisation

[12] https://aws.amazon.com/de/elasticbeanstalk/ (zuletzt besucht 28.09.2022)
[13] https://github.com/eclipse/winery/ (zuletzt besucht 28.09.2022)

für den zentralen Orchestrator erreichbar sein. Dazu kommt, dass einzelne Geräte keine Autonomie vom jeweils zentralen Orchestrator haben, dadurch muss jede Ausführung jeder Operation auf allen Geräte vom zentralen Orchestrator kontrolliert werden.

2.5.5 Deploymentautomatisierung von Multicloud-Anwendungen in föderierte Cloud Umgebungen

Panarello et al. [PBL+17] beschreiben in Ihrer Arbeit ein Konzept, um das Deployment von Multicloud-Anwendungen in eine föderierte Cloud Umgebung zu ermöglichen. Cloud Föderation wird von den Autoren als ein Konzept beschreiben, bei dem Cloud-Provider dynamisch eigene und die Ressourcen von anderen Cloud-Providern kombinieren, um die Anforderungen von Kunden zu erfüllen. Die Cloud-Provider sind dabei separate Entitäten, die interagieren, um ihre jeweils ungenutzten Ressourcen zu teilen. Dadurch können Cloud-Provider die Auslastung ihrer Ressourcen verbessern und Lastspitzen mithilfe von Ressourcen anderer Cloud-Provider ausgleichen. In der Arbeit werden mehrere Probleme identifiziert, welche diese Zusammenarbeit behindern. So ist es problematisch, dass verschiedene Cloud-Provider unterschiedliche Management Systeme und Metamodelle, um Anwendungen zu beschreiben, einsetzen. Weiterhin gibt es keine Standards, um die verfügbaren Ressourcen der Cloud-Provider zu koordinieren und Anwendungen in die föderierte Umgebung zu provisionieren.
Deshalb haben Panarello et al. [PBL+17] ein Konzept und eine Architektur für die Zusammenarbeit zwischen den Cloud-Providern und das Deployment von Anwendungen entwickelt. Um das konzeptionell beschriebene System umzusetzen, kombinieren die Autoren mehrere Standards und existierende Technologien. Die Architektur beschreibt einen zentralen Messaging-Exchange-Server, über den alle Cloud-Provider miteinander kommunizieren können. Weiterhin betreibt jeder Cloud-Provider ein *Application Deployment Toolkit*. Das Toolkit empfängt Anwendungen zum Deployment von Endnutzern und entscheidet, welche Teile der Anwendung lokal vom jeweiligen Cloud-Provider selbst betrieben und welche Teile zu anderen Providern ausgelagert werden. Um diese Entscheidung zu treffen, sendet das Toolkit eine Anfrage über die Art, die Anzahl und den Preis der verfügbaren Ressourcen an die anderen Cloud-Provider. Für diese Kommunikation wählen die Autoren das XMPP[14] Protokoll aus und beschreiben ein Nachrichtenformat für die Ressourcenanfragen und Antworten. Um die Ressourcenanfragen zu erhalten und Antworten zu senden, betreiben die Cloud-Provider Agenten, die in einem gemeinsamen XMPP Chatraum kommunizieren. Anhand der Antworten entscheidet das Toolkit, welche Cloud-Provider für die nicht lokalen Teile der Anwendung genutzt werden und provisioniert diese auf den verfügbaren Ressourcen der anderen Cloud-Provider. Die Arbeit von Panarello et al. [PBL+17] enthält kein Konzept, um zu entscheiden wo welche Anwendungsteile platziert werden, sondern verweist auf existierende Arbeiten. Da manche Cloud-Provider diese Interaktion nicht nativ unterstützen gibt es in dem Konzept Plugins und Message-Oriented Middlewares welche die Kommunikation und Interaktion ermöglichen. Die Autoren verwenden die CLoud-Enabled-Virtual-EnviRonment (CLEVER) [CFV13] Message-Oriented Middleware, welche sich durch Cloud-Provider spezifische Plugins erweitern lässt. Um ein Deployment durchzuführen, müssen die Anwendungsstacks einer Anwendung um Cloud-Provider spezifische Infrastrukturkomponenten erweitert werden. Ein Anwendungsstack ist dabei zum Beispiel ein auf einem virtuellen Server gehosteter Teil der Anwendungstopologie,

[14] https://xmpp.org/ und https://www.rfc-editor.org/rfc/rfc6120 (zuletzt besucht 28.09.2022)

welcher einem Cloud-Provider zugeordnet ist. Die Autoren führen dazu ein Deployment Model Completion Manager ein, der vor dem Deployment jeden Stack um die Cloud-Provider spezifischen Infrastrukturkomponenten erweitert. Um die nötigen Infrastrukturkomponenten und ihre Properties zu bestimmen, gibt es eine Registry, die für jeden Cloud-Provider die benötigten Informationen über die Infrastruktur und Anmeldedaten enthält. Neue Cloud-Provider werden dabei manuell der Registry hinzugefügt. Anschließend wird das erweiterte Modell von einer Laufzeitumgebung ausgeführt. Die Laufzeitumgebung nutzt Schnittstellen, um die Komponenten bei den jeweiligen Cloud-Providern zu provisionieren. Anwendungen werden in TOSCA modelliert und nach den Ergänzungen des Deployment Model Completion Manager mit einer TOSCA Laufzeitumgebung provisioniert. Panarello et al. [PBL+17] nutzen für das Provisionieren OpenTOSCA.
Das Konzept von der Arbeit ermöglicht die Zusammenarbeit von verschiedenen Cloud-Providern, um Ressourcen zu teilen und Anwendungen in föderierte Cloud Umgebung zu provisionieren. Da alle Cloud-Provider die Details ihrer Infrastruktur und die benötigten Anmeldedaten in einer Registry speichern müssen, eignet sich das Konzept nicht für ein organisationsübergreifendes Deployment, bei dem die Teilnehmer diese Informationen nicht teilen können. Weiterhin wird für den Informationsaustausch eine Messaging-basierte Lösung eingesetzt, welche trotz Firewalls oder NAT Kommunikation erlaubt. Jedoch wird für das Deployment aber ein zentraler Orchestrator eingesetzt, sodass Geräte oder Instanzen hinter Firewalls nicht erreicht werden können.

2.5.6 Weitere Arbeiten

Neben den genauer betrachteten Arbeiten gibt es zum Beispiel [LVCD13], [FBH+17], [HBS+16] oder [SBK+16], welche sich mit Softwaredeployments auf IoT-Geräte mit TOSCA beschäftigen. Die Arbeiten [TPM+17], [TCDM21] oder [CCP15] nutzen TOSCA für Multicloud Deployments. Alle diese Arbeiten verwenden einen zentralisierten Orchestrator, der für organisationsübergreifende Deployments nicht geeignet ist [WBK+20].
Die Arbeit von Zimmermann et al. [ZBF+17] beschäftigt sich mit dem Deployment von Datenanalysesoftware von einem externen Dienstleister zur lokalen Datenverarbeitung in einer Organisation. Der Dienstleister kennt dabei die lokale Infrastruktur und die zu verarbeiteten Daten nur teilweise. Die Arbeit bietet mehrere Ansätze, um ein TOSCA Modell ohne die genaue Kenntnis der lokalen Infrastruktur zu erstellen. Jedoch verwendet auch dieser Ansatz einen zentralen Orchestrator [ZBF+17].
Des Weiteren gibt es andere dezentrale Agentensysteme, um Deployments auszuführen. Zum Beispiel nutzen Herry et al. [HAR14] einen Planning-basierten Ansatz, um Workflows für Agenten aus einem Anwendungsmodell zu generieren. Diese Agenten arbeiten Push-based, daher können sie nicht hinter Sicherheitsbeschränkungen eingesetzt werden, welche eingehende Kommunikation verhindern.
Die Arbeit von Hall et al. [HHW99] erarbeitet ein Schema, um eine Software mit ihren Abhängigkeiten in einem provisionierbaren Format zu definieren. Darüber hinaus gibt es in dem System von Hall et al. [HHW99] eine Komponente, welche die verfügbare Software zusammen mit den definierten Abhängigkeiten bereitstellt. Agenten, welche mit der Software mitgeliefert werden, interpretieren das Schema und führen das Deployment aus. Die Arbeit enthält aber kein Konzept, um das Deployment von mehrere Komponenten einer Software auf unterschiedlichen Geräten zu koordinieren. Weiterhin wird für jedes Deployment ein eigener Install-Agent übertragen. Das wirkt sich negativ auf die Lebensdauer eines batteriebetriebenen IoT Geräts aus.

Juve und Deelman [JD12] beschreiben ein agentenbasiertes Deployment System und eine Beschreibungssprache für Deployments. Es werden dabei gesamte virtuelle Maschinen als Nodes modelliert. Dieses System kann aber für organisationsübergreifende Deployments nicht eingesetzt werden, da ein zentraler Koordinator Zugriff auf alle beteiligten Cloud-Provider von allen Organisationen benötigt, um virtuelle Instanzen zu erstellen.
Die Arbeit von Sampaio und Mendonça [SM11] erlaubt das Modellieren von Multicloud-Anwendungen basierend auf dem Open Virtualization Format[15] und nutzt für das Deployment verschiedene Plugins für unterschiedliche Cloud-Provider. Auch in dieser Arbeit kommt eine zentrale Komponente zum Einsatz, die das Deployment ausführt.
Terraform[25] Cloud Agents ist eine cloudbasierte Lösung, um ein Terraform-Deployment in private Clouds oder interne Infrastruktur zu ermöglichen, ohne eingehende Verbindungen zu benötigen [WWG22]. Diese Lösung hat den Nachteil, dass alle Agenten einer Organisation global für alle Teilnehmer zugreifbar sind [WWG22]. Das betrifft auch Agenten, die aktuell nicht am Deployment teilnehmen [WWG22]. Das schränkt die organisationsübergreifende Nutzung ein [WWG22].
Die Arbeiten [MN10] und [FRL05] verwenden beide mobile Agenten, um Software auf Geräte zu installieren. Aber beide Arbeiten erlauben nicht das koordinierte Deployment von unterschiedlichen Komponenten einer Software.
Ein Konzept, um imperative Deploymentmodelle aufzuteilen wurde von Kopp und Breitenbücher [KB17] präsentiert.
Die Arbeit von Arcangeli et al. [ABL15] bietet einen Überblick zu automatisierten Deploymentlösungen in verteilten Umgebungen. Aber keine der vorgestellten Lösungen bietet TOSCA-basierte Agenten, die (teil)autonom ein Deployment bestehend aus mehrere Komponenten provisionieren und koordinieren können.

Andere verbreitete Lösungen wie Ansible[19] oder Terraform[25] nutzen einen zentralen Manager, der Befehle auf Ressourcen pushed [KBL+19]. Diese Lösungen funktionieren nicht, wenn sich die Ressourcen hinter Sicherheitsbeschränkungen befinden. Puppet[16] und Chef[11] verwenden einen zentralen Manager und verteilte Agenten [KBL+19; Pup]. Beide Lösungen unterstützen aber kein TOSCA und verwenden HTTP zur Kommunikation zwischen dem Manager und den Agenten [MSDC22; Pup]. HTTP eignet sich jedoch schlechter für IoT-Geräte mit unzuverlässiger Verbindung als asynchrones Messaging.

2.5.7 Andere TOSCA Orchestratoren

Neben dem in dieser Arbeit entwickelten Orchestrator, der im Agentenprototyp integriert ist, gibt es andere Implementierungen für TOSCA. Es werden im Folgenden mehrere ausgewählte Orchestratoren betrachtet.

OpenTOSCA OpenTOSCA[17] ist ein Ökosystem bestehend aus Eclipse Winery[13] und dem TOSCA-1.0-basierten OpenTOSCA Container[8] [BEK+16; II]. Eclipse Winery wird zum grafischen Modellieren von Anwendungen verwendet und der Container zum Provisionieren und Managen

[15] http://www.dmtf.org/standards/ovf und ISO Standart 17203 (zuletzt besucht 28.09.2022)
[16] https://puppet.com/ (zuletzt besucht 28.09.2022)
[17] https://www.opentosca.org/ (zuletzt besucht 28.09.2022)

der in TOSCA modellierten Anwendungen. Ergänzend zu dem zentralen Orchestrator gibt es viele wissenschaftliche Arbeiten wie [KBL+19] oder [WBK+20], die Konzepte erforschen oder den Funktionsumfang erweitern.

xOpera Bei xOpera[18] handelt es sich um ein Open Source in Python entwickelten Orchestrator [LCM22]. Dieser verwendet TOSCA 1.3 und unterstützt Ansible[19] Playbooks als Implementierungen für Operationen [LCM22].

Cloudify Cloudify[20] ist eine Open Source Plattform mit TOSCA Orchestrator und einem Modellierungswerkzeug [LSC20]. Cloudify verwendet eine eigene auf TOSCA basierende Domain Specific Language (DSL) [EoB+20]. Neben der eigenen DSL hat Cloudify viele eingebaute abstrakte Node Types, die das Modellieren vereinfachen [NtM+22]. Cloudify bietet zum Beispiel Plugins für Skripte, Ansible Playbooks, AWS[9] oder Kubernetes[7]. Manche dieser Plugins stellen vorgefertigte Node Types bereit, um Ressourcen zu modellieren [ENo+22]. Für das Deployment unterstützt Cloudify verteilte Agenten [NAoi22]. Die Agenten kommunizieren per AMQP[21] Messaging und HTTPS mit einer zentralen Managerkomponente [NAoi22]. Dadurch ist Kommunikation trotz Sicherheitsbeschränkungen, welche eingehende Verbindungen blockieren, möglich. Jedoch können Cloudify-Agenten immer nur einzelne Operationen ausführen und nicht autonom ganze Subgraphen mit mehreren Node Templates provisionieren.

Yorc Yorc[22] ist ein Open Source Orchestrator für TOSCA 1.2 [AGB21]. Er unterstützt als Implementierungen für Operationen Bash und Python Skripte sowie Ansible Playbooks [AGB21]. Darüber hinaus hat der Orchestrator eine Plugin Schnittstelle, um weitere Implementierungsarten zu unterstützen [AGB21]. Außerdem verwendet das Alien4Cloud[23] Projekt Yorc als einen TOSCA Orchestrator [fra21].

Unfurl Unfurl[24] ist ein Open Source Orchestrator, der für ein Projekt das TOSCA 1.3 Modell, alle Implementierungen, Workflows und den Status der verwendeten Ressourcen und Services in einem Git-Repository speichert [Oneb]. Er unterstützt unter anderem Ansible Playbooks, verschiedene Skriptsprachen und Terraform[25] in Operationen und zum Beispiel Kubernetes[7] oder Docker[26] über mitgelieferte Node Types. Unfurl verfolgt einen Ansatz, der keine Server oder Agenten verwendet [Sj22]. Der Orchestrator arbeitet mit Befehlen über eine Kommandozeilenschnittstelle und verwaltet Projekte in Ordnern [Sj22]. Unfurl ermöglicht einen GitOps-basierten Workflow, da Änderungen am Zustand oder der Konfiguration der verwalteten Ressourcen und Komponenten in ein Git-Repository synchronisiert werden [Onea].

[18] https://github.com/xlab-si/xopera-opera/ (zuletzt besucht 28.09.2022)
[19] https://www.ansible.com/ (zuletzt besucht 28.09.2022)
[20] https://cloudify.co/ (zuletzt besucht 28.09.2022)
[21] https://www.amqp.org/ und ISO/IEC Standard 19464 (zuletzt besucht 28.09.2022)
[22] https://github.com/ystia/yorc/ (zuletzt besucht 28.09.2022)
[23] https://alien4cloud.github.io/ (zuletzt besucht 28.09.2022)
[24] https://unfurl.run/ (zuletzt besucht 28.09.2022)
[25] https://www.terraform.io/ (zuletzt besucht 28.09.2022)
[26] https://www.docker.com/ (zuletzt besucht 28.09.2022)

3 Konzept für die Entwicklung eines Multiagentensystems

Um die in Abschnitt 1.1 definierten Ziele für ein dezentralisiertes und TOSCA-basiertes Softwaredeployment in eine heterogene organisationsübergreifende Multicloud- und IoT-Umgebung zu ermöglichen, werden in diesem Kapitel mehrere Konzepte erarbeitet. Das Deploymentkonzept setzt auf kooperierende Agenten, die jeweils möglichst autonom Teile des Deployments ausführen. Wooldridge [Woo02] beschreibt einen Agenten als ein System, das sich in einer Umgebung befindet und autonom Aktionen in dieser ausführen kann, um sein Ziel zu erreichen. Es gibt aber unterschiedliche Definitionen für einen Agenten in unterschiedlichen Fachbereichen [Woo02].
Um das verteilte Deployment über Agenten zu ermöglichen, muss ein Konzept erarbeitet werden, um Node Templates in einer mit TOSCA modellierten Anwendung, einem Agenten zuzuordnen. Die Zuordnung sollte dabei mit existierenden TOSCA Features umgesetzt werden und nicht durch die Einführung einer Erweiterung. Dadurch ist das resultierende Service Template kompatibel zu anderen Modellierungswerkzeugen. Das Konzept, um diese Zuordnung zu realisieren, ist in Abschnitt 3.1 näher beschrieben.
Für modellierte Anwendungen werden außerdem keine Geräte oder Instanzen als Node Templates in das Service Template integriert. Im Service Template sind nur Anwendungskomponenten enthalten und diese werden direkt auf dem Gerät installiert, auf dem sich der Agent befindet. Dadurch müssen Organisationen keine Details über die verwendeten Geräte oder ihre Infrastruktur untereinander teilen.
Die Agenten erlauben ein teilautonomes und asynchrones Deployment auszuführen. Dabei kann ein Agent alle Komponenten einer Software, die auf dem lokalen Gerät installiert werden sollen, anhand des TOSCA Modells bestimmen und die notwendigen Operationen weitgehend autark ausführen. Dabei ist es wichtig, dass nur die notwendigen Teile des Deploymentmodells auf jeden Agenten übertragen werden. Um das zu ermöglichen werden in, Abschnitt 3.2 mehrere Ansätze vorgestellt und miteinander verglichen, um ein TOSCA Modell in unterschiedliche Subgraphen aufzuteilen.
Um ein dezentrales Deployment auszuführen, gibt es Anforderungen an die Kommunikation der Agenten untereinander. Diese wurden von Kopp und Breitenbücher [KB17] für die verteilte Ausführung von Workflows beschrieben. Basierend darauf werden in Abschnitt 3.3 die Kommunikationsanforderungen für ein deklaratives Modell erarbeitet und eine MQTT-basierte Kommunikationsarchitektur beschrieben. Um den gleichen CSAR mehrfach provisionieren zu können, wird in Abschnitt 3.3.1 ein Konzept präsentiert, um *DeploymentIDs* in die Kommunikationsarchitektur zu integrieren.
Um zu verhindern, dass im Fall eines kompromittierten Agenten ein Angreifer Zugriff auf alle Properties und Attribute bekommt, wird die Kommunikationsarchitektur in Abschnitt 3.3.2 um ein Sicherheits- und Visibilitykonzept für Properties und Attribute erweitert. Das Sicherheitskonzept verhindert auch, dass ein Agent über MQTT Zugriff auf alle Subgraphen erhält.
Viele Cloud-Provider bieten eine automatisierte Skalierung der verwendeten Ressourcen an. So können zum Beispiel neue virtuelle Maschinen gestartet werden, wenn mehr Leistung benötigt

wird. Um diesen Anwendungsfall abzubilden, ist in Abschnitt 3.4 ein Konzept beschrieben, um das dynamische Hinzufügen weiterer Agenten für einen bestimmten Subgraphen zu unterstützen.
Der letzte Abschnitt dieses Kapitels beschreibt einen GitOps-basierten Workflow für TOSCA, um die Zusammenarbeit an einem zentralen Modell für alle Entwickler zu erleichtern. Auch organisationsübergreifende Zusammenarbeit wird erleichtert, da die Entwickler unterschiedlicher Organisationen in einem zentralen Repository zusammenarbeiten können. Der Workflow ermöglicht es Änderungen einem Review zu unterziehen und in einer Versionshistorie zu speichern.

3.1 Integration von Agenten in TOSCA Modelle

Um das Deployment eines Service Templates über verteilte Agenten zu ermöglichen, muss jeder Agent wissen, welches Node Template ihm zugeordnet ist. Diese Zuordnung ist eine Liste von Node Templates für jeden Agenten. Diese Liste kann entweder separat vorliegen oder direkt in das Service Template integriert werden. Die Integration in ein Service Template bietet Vorteile gegenüber einer separaten Liste. Durch die Integration kann die Zuordnung mit den gleichen Werkzeugen modelliert werden, mit denen sonst auch das Service Template selbst bearbeitet und erstellt wird. Das reduziert die Wahrscheinlichkeit von Fehlern, wenn entweder das Service Template oder die Zuordnung bearbeitet wird. Weiterhin ist direkt anhand des Service Templates ersichtlich, welche Node Templates welchem Agenten zugeordnet sind.
Es ist daher Ziel dieser Arbeit, die Zuordnung zwischen Node Templates und Agenten direkt in das Service Template zu integrieren. Weiterhin ist es vorteilhaft, wenn die Repräsentation der Zuordnung kompatibel mit der TOSCA Spezifikation ist. Spezielle Erweiterungen des Sprachumfangs wie zum Beispiel die Erweiterung für EDMM von Wurster et al. [WBB+21] hätten den Nachteil, dass diese nicht von anderen TOSCA-kompatiblen Werkzeugen unterstützt werden. Es ist daher das Ziel, die Zuordnung von Node Templates zu Agenten mit vorhanden TOSCA Features in Service Templates zu integrieren.
Agenten sollten nicht als eigene Entitäten, wie zum Beispiel Node Templates modelliert werden, da sie mehrere Node Templates beinhalten können. Weiterhin sind die Agenten selbst keine Komponente der zu modellierenden Anwendung. Es gibt mehrere Möglichkeiten eine Zuordnung zwischen Node Template und Agenten vorzunehmen. Die folgenden Features wurden identifiziert, um eine solche Zuordnung zu repräsentieren: spezielle Properties, spezielle Metadaten, Directives, Policies und Groups. Die einzelnen Möglichkeiten werden im Folgenden erklärt.

Spezielle Properties Properties sind Schlüsselwertpaare, die einem Node Template zugeordnet werden können. Um Properties für die Zuordnung von Node Templates zu Agenten zu benutzen, wird ein Property mit einem speziellen Namen definiert. Das Property kann zum Beispiel „AGENT_NAME“ heißen. Diesem Property wird in der Definition des Node Templates der Namen des für ihn zuständigen Agenten als Wert zugewiesen. Während des Deployments werden alle Node Templates mit dem Property „AGENT_NAME“ identifiziert und anhand des gesetzten Namens einem Agenten zugeordnet. Die Verwendung von Properties hat den Nachteil, dass jedes Node Template um das neue Property erweitert werden muss. Auch muss das Property in jedem Node Template angepasst werden, wenn die Node Templates in einem anderem TOSCA Modell wiederverwendet werden. Weiterhin beschreibt die TOSCA Spezifikation Properties als Inputs für Node Templates [OAS20b]. Diese Inputs repräsentieren den gewünschten Zustand, in dem sich ein Node Template nach dem Deployment befinden soll [OAS20b]. Wird ein Property als Indikator für

die Zuordnung zu einem Agenten verwendet, ist das Property kein Input für das Node Template, sondern für den Orchestrator. Daher widerspricht diese Verwendung der von TOSCA vorgesehenen Semantik. Eine weitere Einschränkung dieser Lösung ist, dass sich die Zuordnung von Agenten auf Node Templates nicht weiter beschränken lässt. Es kann zum Beispiel nicht modelliert werden, dass Agenten nur spezielle Node Templates unterstützen.

Spezielle Metadaten Die Verwendung von speziellen Metadaten funktioniert ähnlich wie die von Properties. Metadaten können in TOSCA für jedes Node Template definiert werden. Um ein Node Template einem Agenten zuzuordnen, wird in jedem Node Template ein spezielles Schlüsselwertpaar innerhalb der Metadaten definiert. Dieses Schlüsselwertpaar enthält den Namen des Agenten. Diese Lösung hat einerseits den Vorteil, dass sie anders als die spezielle Properties nicht gegen eine in der TOSCA Spezifikation definierte Semantik verstößt. Andererseits ist die Semantik der in den Metadaten enthaltenen Information, nicht genauer durch die TOSCA Spezifikation definiert. Dadurch können Modellierungswerkzeuge, welche für die TOSCA Spezifikation ohne diese Erweiterung entwickelt wurden, keine Unterstützung bei der Erstellung der Metadaten bieten.

Directives Directives sind eine Liste von Strings, die jedem Node Template zugeordnet werden können, um einem Orchestrator weitere Instruktionen zur Verarbeitung mitzuteilen [OAS20b]. Die Spezifikation enthält eine Liste der möglichen Strings, welche für die Zuordnung von Node Templates zu einem Agenten erweitert wird. Jeder neue String enthält eine Zuordnung. Wird das Node Template „TemperaturSensor“ dem Agenten „TemperaturAgent“ zugeordnet, lautet der Directive String „AgentMapping:TemperaturAgent“. Dieser String wird dem „TemperaturSensor“ Node Template als Directive hinzugefügt. Nachteilig an dieser Lösung ist, dass die Spezifikation eine Liste aller definierten Directives enthält. Die Verwendung nicht definierter Directive Strings könnte zu Verarbeitungsproblemen in externen Modellierungs- und Validierungswerkzeugen führen.

Policies und Groups Policies und Groups erlauben es, Node Templates in Gruppen zusammenzufassen. Bei Policies wird der Keyname *targets* verwendet, um eine Liste an Node Templates zu definieren, auf welche die Policy angewendet wird [OAS20b]. Bei Groups wird dagegen der *members* Keyname verwendet, um Node Templates einer Gruppe zuzuweisen. Policies und Groups sind typisierte TOSCA Features [OAS20b]. Groups haben einen Group Type und Policies haben einen Policy Type. In den Typen ist es möglich, Einschränkungen der erlaubten Node Types zu definieren. Dadurch können nur Node Templates mit einem erlaubten Node Type der Group oder Policy hinzugefügt werden. So ist es unter anderem möglich sicherzustellen, dass modellierte externe Dienste wie zum Beispiel Software-as-a-Service (SaaS) Anwendungen nicht einem Agenten zum Provisionieren zugeordnet werden können.
Modellierungswerkzeuge können diese Einschränkungen ohne Anpassungen unterstützen, da sie in der TOSCA Spezifikation enthaltene Features nutzen [OAS20b]. Weiterhin können Modellierungswerkzeuge bei der Erstellung von Policies oder Groups unterstützen. Dafür müssen die Modellierungswerkzeuge keine Agenten unterstützen, sondern nur die TOSCA Spezifikation. Aus syntaktischer Sicht eignen sich beide Features, um die Zuordnung abzubilden. Die TOSCA Spezifikation definiert aber unterschiedliche Semantiken für beide Features. Policies sind laut der TOSCA Spezifikation nicht funktionale Anforderungen wie Quality-of-Service Ziele oder Zahlungsbedienungen, die auf eine Menge an Node Templates angewendet werden [OAS20b]. Groups sind als logische Gruppen, typischerweise für Managementziele, beschrieben [OAS20b].

Die Semantik des Group Features passt besser, da es sich bei der Zuordnung zwischen Node Templates und Agenten um eine funktionale Anforderung handelt. Werden die Node Templates nicht korrekt verteilt, können manche Funktionalitäten nicht umgesetzt werden. Weiterhin lassen sich Policies auf Groups anwenden. Dadurch ist es möglich eine Policy zum Beispiel auf alle Node Templates eines Agenten anwenden, da alle Node Templates in einer Group enthalten sind.

Alle gezeigten Möglichkeiten erlauben es, Node Templates den Agenten zuzuordnen. Am besten geeignet ist das Group Feature der TOSCA Spezifikation, da die Semantik des TOSCA Features am besten den Anforderungen entspricht und sich die kompatiblen Node Types einschränken lassen. Auch können Modellierungs- und Validierungswerkzeuge dieses Feature ohne spezielle Anpassungen für die Konzepte dieser Arbeit unterstützen. Deshalb wird in dieser Arbeit das Group Feature ausgewählt, um die Zuordnung von Node Templates zu Agenten umzusetzen. Dazu wird der Group Type „AgentGroupDefinition“ definiert. Jede Group von diesem Type repräsentieren jeweils einen Agenten. Einem Agenten können Node Templates zugeordnet werden, indem sie der *members* Liste hinzugefügt werden. Ein Service Template mit diesen Groups ist zu der TOSCA Spezifikation kompatibel. Orchestratoren, die keine Unterstützung für Agenten bieten, ignorieren diese Groups. Unterstützt ein Orchestrator das verteilte Deployment über Agenten, erkennt er den speziellen Group Type mit dem Namen „AgentGroupDefinition“.

3.2 Generierung von Modell-Subgraphen

Dieser Abschnitt beschreibt, wie ein deklaratives TOSCA Modell einer Anwendung in mehrere Subgraphen aufgeteilt werden kann. Dazu werden im Folgenden mehrere Ansätze vorgestellt und miteinander verglichen. Es wird für jeden Agenten ein individueller Subgraph erstellt, der alle dem Agenten zugewiesenen Node und Relationship Templates umfasst. Aus den individuellen deklarativen Subgraphen kann jeder beteiligte Agent selbstständig lokal seinen eigenen Deployment-Workflow mit dem von Breitenbücher et al. [BBK+14] beschriebenen Verfahren ableiten. Der Workflow umfasst alle zur Provisionierung notwendigen Operationen für jedes Template des individuellen Subgraphs. Die im Folgenden präsentierten Ansätze, um Subgraphen aus der gesamten Anwendungstopologie zu generieren, haben unterschiedliche Vor- und Nachteile. Es kann zum Beispiel vorteilhaft sein, in den Subgraph jedes Agenten noch weitere Informationen aus anderen Teilen der Topologie einzubetten. Agenten können mit Zugriff auf mehr Informationen oft autonomer arbeiten. So kann eine Operation auf einem Agent Properties von Node Templates, die einem anderen Agenten zugeordnet sind, benötigen. Sind diese Properties direkt im Subgraph des ersten Agenten enthalten, müssen die Properties nicht während des Deployments von dem anderen Agenten abgerufen werden. Das spart den Overhead einer Nachrichtenübertragung und erhöht die Autonomie des Agenten. Jedoch hat es auch Nachteile, mehr Informationen in ein Subgraph einzubetten. Einerseits erhöhen nicht benötigte Informationen unnötig die Größe der Subgraphen, die auf die Agenten übertragen werden. Anderseits sind mehr Informationen bedroht, wenn ein Agent gestohlen oder kompromittiert wird.
Die Ansätze unterscheiden sich nicht nur in der Menge der Informationen, die zusätzlich in die Subgraphen eingebettet werden, sondern auch in weiteren Punkten. Für manche Ansätze müssen zum

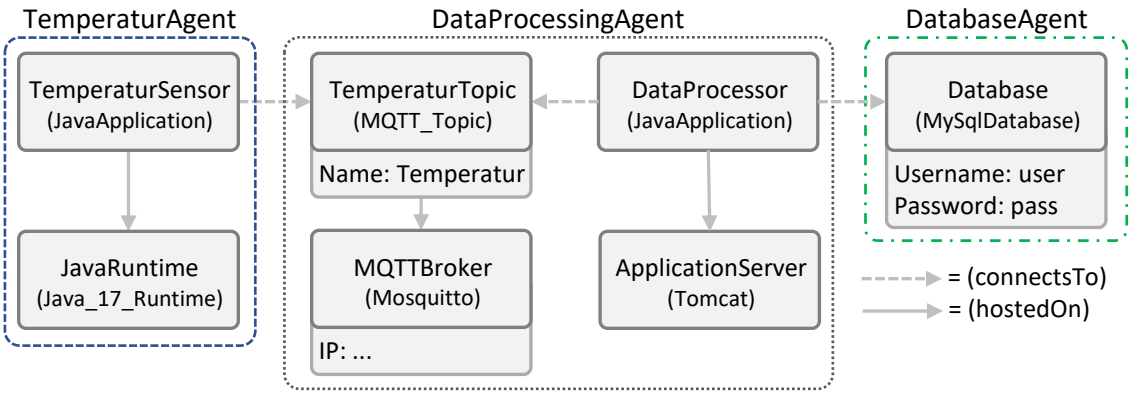

Abbildung 3.1: Beispiel für die Repräsentation von Agenten in einem Service Template. Die Node Templates sind in drei Gruppen aufgeteilt, die jeweils einem Agenten zugeordnet sind. Attributes haben den Wert „...“, da diese erst während des Deployments bestimmt werden. [Notation basiert auf Vino4TOSCA BBK+12]

Beispiel bei der Erstellung des Modells der Anwendung bestimmte Regeln oder Einschränkungen beachtet werden. Die einzelnen Ansätze werden in den Abschnitten 3.2.1 bis 3.2.5 näher beschrieben und in den Abschnitten 3.2.6 und 3.2.7 miteinander verglichen.

In Abbildung 3.1 ist eine Anwendung abgebildet, die als Beispiel für die weiteren Abschnitte verwendet wird. Einige Properties und alle Operationen wurden zur Vereinfachung nicht abgebildet. Jedes Node und Relationship Template hat aber alle Operationen des in der TOSCA Spezifikation Version 1.3 [OAS20a] beschriebenen Lifecycle Interfaces, um den Lebenszyklus der Templates kontrollieren zu können. Die Anwendung besteht aus einer Java Anwendung, die Temperaturdaten über einen Sensor erfasst und auf der JavaRuntime läuft. Die Temperaturdaten werden über MQTT an einen Temperatur Topic weitergegeben. Der „DataProcessor“ ist auf diesen Topic subscribed und verarbeitet die Daten, bevor sie in einer Datenbank gespeichert werden. Der Topic ist auf einem MQTT Broker gehostet und der DataProcessor auf einem Tomcat[1] Application Server. Die Node Templates sind auf Agenten verteilt, welche jeweils für das Deployment der ihnen zugeordneten Node Templates zuständig sind.
Die Node Templates, die einem Agenten zugeteilt sind, werden im Folgenden als *lokale Node Templates* des Agenten bezeichnet. Relationship Templates, die zwei lokale Node Templates verbinden, werden *lokale Relationship Templates* genannt. Node Templates von anderen Agenten, sind *externe Node Templates*. So ist der TemperaturSensor und die JavaRuntime dem TemperaturAgent zugeordnet. Für den TemperaturAgent sind beide Node Templates lokal, während sie für den DatabaseAgent extern sind. Dem DatabaseAgent ist das Node Template Database zugeordnet. Der Rest aller abgebildeten Node Templates wird vom DataProcessingAgent provisioniert. Nicht alle Node Templates müssen einem Agenten zugeordnet sein. Externe Dienste wie SaaS Angebote, die nicht während des Deployments gestartet werden müssen, können als Node Template modelliert werden, benötigen aber keinen Agenten.
Jeder Agent muss über alle Informationen über die ihm zugeordneten Node und Relationship Templates verfügen, sonst kann er sie nicht provisionieren. Er muss alle lokalen Operationen kennen, um sie ausführen zu können. Weiterhin benötigt er zum Beispiel alle Node Types, Interfaces und

[1] https://tomcat.apache.org/ (zuletzt besucht 28.09.2022)

Artifacts welche von lokalen Node Templates verwendet werden. Deshalb werden bei jedem Ansatz die Subgraphen als Service Templates in einem CSAR übertragen. Die Service Templates enthalten alle in dem Subgraph verwendeten Types und Interfaces. Die für das Deployment notwendigen Implementation und Deployment Artifacts befinden sich zusammen mit dem Service Template im CSAR.
Relationship Templates können beliebige Typen mit unterschiedlicher Semantik aufweisen. Breitenbücher et al. [BBK+14] identifizieren in ihrer Arbeit jedoch zwei unterschiedliche Kategorien von Relationship Templates. Die erste Kategorie drückt eine Abhängigkeit zwischen zwei Node Templates aus. Darunter fallen zum Beispiel die Typen „hostedOn“, „runsOn“ oder „installedOn“. Die zweite Kategorie drückt eine Verwendung eines Node Templates von einem anderen Node Template aus. Diese Art umfasst zum Beispiel die Typen „connectsTo“, „invokes“ oder „calls“. In dieser Arbeit werden stellvertretend für alle Typen aus den Kategorien nur Relationship Templates vom Type „hostedOn“ oder „connectsTo“ verwendet. Die Konzepte und Ansätze lassen sich jedoch auf alle Relationship Types in den jeweiligen Kategorien anwenden. Zum Beispiel könnten auch Relationship Templates vom Type installedOn anstelle von hostedOn verwendet werden. [BBK+14]
Die TOSCA Spezifikation enthält keine Einschränkungen, wie viele ausgehende hostedOn Relationship Templates ein Node Template haben kann [OAS20b]. Jedoch werden in dieser Arbeit nur Node Templates mit maximal einem ausgehendem hostedOn Relationship Template erlaubt, da die Semantik eines Node Templates, das auf mehreren Node Templates gehostet wird, nicht klar definiert ist [KBL+19]. Weiterhin sind Verbindungen vom Type hostedOn nur zwischen lokalen Node Templates auf einem Agenten möglich. Zum Beispiel kann eine Java Anwendung nicht von einer Java Runtime ausgeführt werden, welche sich auf einem anderen Agenten, also einem anderen Gerät befindet. Relationship Templates vom Relationship Type connectsTo sind zwischen zwei Node Templates von unterschiedlichen Agenten möglich. Um ein connectsTo Relationship Template zu instanziieren, werden die Operationen des Relationship Templates auf dem Agenten mit dem Ursprungs-Node-Template ausgeführt.

3.2.1 Kein Split Ansatz

Der *kein Split* Ansatz hat den Vorteil, dass keine Aufteilung des gesamten Anwendungsmodells in mehrere Subgraphen notwendig ist. Jeder Agent erhält das komplette Modell und provisioniert nur die dem Agenten zugeordneten Node Templates. Dadurch muss kein Modellierungwerkzeug oder zentraler Agent das Gesamtmodell verarbeiten und einzelne Subgraphen generieren. Weiterhin stehen Agenten während des Deployments alle im Modell enthalten Informationen zur Verfügung und es müssen keine Properties von anderen Agenten abgerufen werden. Dadurch ist während des Deployments weniger Kommunikation notwendig und eine größere Autonomie der Agenten möglich. Anders als Properties werden Attributes oft erst während des Deployments von Operationen bestimmt. Deshalb müssen Attributes während des Deployments zwischen den Agenten ausgetauscht werden. Wenn in dem Beispiel in Abbildung 3.1 die connectsTo Verbindung von dem TemperaturSensor die IP des MQTT Brokers benötigt, welche erst während des Deployments bekannt wird, muss der TemperaturAgent eine Anfrage an den DataProcessingAgent senden.
Operationen von Node Templates können beliebige Attributes lesen, aber nur die eigenen Attributes des Node Templates schreiben [OAS20b]. Deshalb müssen Attributes von anderen Agenten gelesen, aber nicht auf andere Agenten geschrieben werden.
Weitere Kommunikation ist bei der Instantiierung von connectsTo Verbindungen zwischen Node

Templates auf verschiedenen Agenten notwendig. Hier muss sichergestellt werden, dass eine Verbindung erst erstellt wird, wenn die Quelle und das Ziel der Verbindung gestartet wurden [BBK+14]. Das die Quelle instantiiert und gestartet wurde, kann der Agent selbst verifizieren, da er das Ursprungs-Node-Template gestartet hat. Um sicherzustellen, dass das Ziel einer connectsTo Verbindung instantiiert und gestartet wurde, muss der Agent mit dem Zielagenten kommunizieren. Auch ist der Ansatz einfach zu implementieren, da der ursprüngliche CSAR ohne Weiterverarbeitung verwendet werden kann.
Nachteilig ist, dass alle Agenten die komplette Struktur und alle Properties der Anwendung kennen. So muss der TemperaturAgent keinen Zugriff auf den Username und das Password der Datenbank erhalten. Darüber hinaus sind im originalen CSAR auch alle *Artifacts* enthalten. Dadurch müssen Artifacts, die zum Beispiel Operationen implementieren, aber nur von einem Agenten benötigt werden, auf alle Agenten übertragen werden. Es muss insgesamt das komplette Modell und alle Artifacts auf jeden Agenten übertragen werden, das erhöht die Datenmenge, die zu jedem Agenten übertragen wird.
Eine einfache Erweiterung dieses Ansatzes ist, dass für jeden Agenten individuelle CSARs generiert und die jeweils nicht verwendete Artifacts entfernt werden. Das spart Bandbreite, Speicherplatz und verhindert, dass alle Implementierungsdetails jedem Gerät zur Verfügung stehen, benötigt aber eine Vorverarbeitung vor dem Deployment.

3.2.2 Proxy Replacement Subgraph Ansatz

Beim *Proxy Replacement Subgraph* (PRS) Ansatz wird für jeden Agenten ein individualisierter Subgraph aus dem kompletten Modell generiert. Das kann bei einem Vorverarbeitungsschritt nach dem Erstellen des Modells oder automatisiert von einem Agenten im Netzwerk übernommen werden. Um ein individuellen PRS zu generieren, werden alle nicht lokalen Node und Relationship Templates des Agenten im kompletten Modell durch *Proxy Node und Relationship Templates* ersetzt. Relationship Templates von einem lokalen zu einem externen Node Template sind davon nicht betroffen. Ein Proxy Node Template ist ein Template, das über alle Properties des zu ersetzenden Node oder Relationship Templates verfügt. Dabei werden alle Interfaces, Inputs, Operationen, Artifacts und Types die nicht zu einem lokalen Node Template gehören entfernt. Auch wird jeder Subgraph zu einem validen Service Template vervollständigt. Dazu muss der „ProxyType“ Node Type dem Service Template hinzugefügt werden. Ein Beispiel PRS für den TemperaturAgent aus Abbildung 3.1 sieht man in Abbildung 3.2. Es wurden alle für den TemperaturAgent nicht lokalen Node Templates durch Proxys ersetzt. Gegenüber dem vorherigen Ansatz reduziert sich dadurch die Datenmenge, die zu jedem einzelnen Agenten übertragen werden muss, erheblich. Weiterhin sind in einem PRS alle Properties enthalten und so müssen diese nicht über das Netzwerk von einem anderen Agenten abgefragt werden, falls das Property in lokalen Operationen als Inputs benötigt wird. Attributes können erst während des Deployments bestimmt werden, daher müssen nicht lokale Attributes trotzdem erst während des Deployments von dem zuständigen Agenten abgefragt werden. Auch müssen die Agenten den Deploymentvorgang über Netzwerk koordinieren. Da im PRS die Zuordnung zwischen Agenten und den Proxy Templates enthalten bleibt, weiß jeder Agent mit welchem anderen Agent er für Attributesanfragen oder zur Koordination kommunizieren muss. Nachteilig ist, dass jeder Agent die komplette Struktur der modellierten Anwendung und alle Properties erhält. Ähnlich wie im vorherigen Ansatz muss der TemperaturAgent keinen Zugriff auf die Properties des Database Node Templates haben. Jedoch könnte jede Operation jedes Property

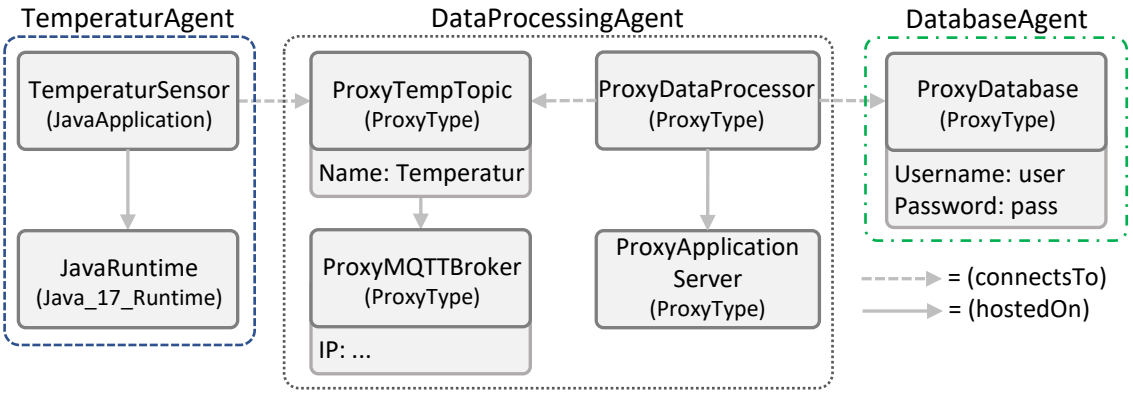

Abbildung 3.2: Subgraph für den TemperaturAgent unter Verwendung des PRS Ansatz. Alle nicht lokalen Node und Relationship Templates wurden durch Proxy Templates ersetzt. [Notation basiert auf Vino4TOSCA BBK+12]

als Input verwenden [OAS20b]. Deshalb können Properties ohne spezielle Modellierungsregeln nicht einfach aus dem PRS entfernt werden. Diese Modellierungsregeln werden in Abschnitt 3.2.3 näher beschrieben.

3.2.3 Reduced Proxy Subgraph Ansatz

Beim Reduced Proxy Subgraph (RPS) Ansatz werden im Gegensatz zu dem vorherigen PRS Ansatz weniger Node Templates in den Subgraphen für die jeweiligen Agenten hinzugefügt. Die Subgraphen enthalten jeweils alle lokalen Node und Relationship Templates sowie alle ausgehenden Relationship Templates. Weiterhin sind alle Ziel Node Templates von ausgehenden Relationship Templates als Proxys enthalten. Die Generierung eines Subgraphen für einen Agenten besteht aus vier Schritten. Im ersten Schritt werden alle lokalen Node und Relationship Templates des Agenten dem Subgraph hinzugefügt. In Schritt zwei wird für jedes lokale Node Template überprüft, ob es über Relationship Templates zu externen Node Templates anderen Agenten verfügt. Für jedes identifizierte externe Node Template wird im dritten Schritt ein Proxy Node Template generiert und zusammen mit dem verbindenden Relationship Template zum Subgraph hinzugefügt. Ein Proxy Template ist wie in Abschnitt 3.2.2 ein neues Node Template, das nur über die Properties des ersetzten Node Templates verfügt. Abbildung 3.3 zeigt ein Beispiel Subgraph für den TemperaturAgent. Um den Subgraph in Abbildung 3.3 für den TemperaturAgent aus dem Ausgangsmodell in Abbildung 3.1 zu erstellen, wurden dem RPS zunächst die Node und Relationship Templates des TemperaturAgent hinzugefügt. Danach wurde für das TemperaturSensor und JavaRuntime Node Template überprüft, ob sie über ein Relationship Template verfügen, das ein externes Node Template als Ziel hat. Dabei wurde das TemperaturTopic Node Template als externes Ziel identifiziert und als Proxy zusammen mit dem Relationship Template hinzugefügt. Im Vergleich mit dem Proxy Replacement Subgraph müssen im RPS Ansatz weniger Information übertragen werden und ein einzelner Agent erhält weniger Informationen über die Struktur der Anwendung. Wie bei den anderen Ansätzen ist für den Attributeszugriff und für die Koordination des Deployments Netzwerkkommunikation notwendig. Da die Zuordnung zwischen Proxy Node Templates und Agenten bei diesem Ansatz auch im Subgraph enthalten ist, weiß jeder Agent welcher andere Agent für die Koordination einer connectsTo Verbindung kontaktiert werden muss.

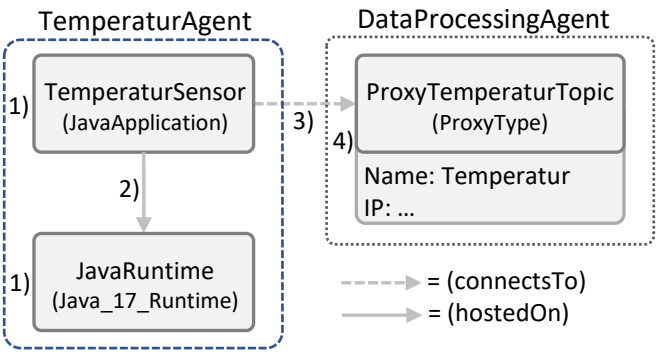

Abbildung 3.3: Subgraph für den TemperaturAgent unter Verwendung des RPS Ansatz. Node Templates welche, direkt mit einem lokalen Node Template verbunden sind, werden durch ein Proxy Template ersetzt. [Notation basiert auf Vino4TOSCA BBK+12]

Der RPS Ansatz reduziert die übertragene Datenmenge und versteckt Teile der Struktur der Anwendung. Aber durch Property und Attribute Funktionen kann bei jeder Operation jedes Property und Attribute in einem Service Template gelesen werden. Der TemperaturSensor könnte zum Beispiel den Username und das Passwort der Datenbank über eine Property Funktion abfragen, obwohl er keine direkte Verbindung zur Datenbank hat. Da der RPS nicht alle Node Templates mit ihren Properties enthält, kann zum Beispiel eine Property Funktion keine Properties von manchen externen Node Templates lesen.
Daher muss eine Modellierungsregel für Modellautoren eingeführt werden, um den RPS Ansatz zu ermöglichen. Die Regel schränkt die Reichweite von Property und Attribute Funktionen ein und erlaubt nur Zugriff auf externe Properties und Attributes von Node Templates, welche direkt über eine Relationship mit einem lokalen Node Template verbunden sind. Dadurch können zum Beispiel Operationen der *connectsTo* Verbindung zwischen TemperaturSensor und dem Proxy Template alle Properties und Attributes lesen, welche direkt im Proxy Template definiert sind. Jedoch keine vom DataProcessor oder von der Database. Modellautoren müssen darauf achten, dass alle Properties und Attributes entweder in lokalen Node Templates oder im Ziel Node Template einer Verbindung vorhanden sind. Deshalb wurde in Abbildung 3.3 das Attribute „IP“ von dem MQTT Broker zu dem Topic verschoben. Dadurch hat die *connectsTo* Verbindung Zugriff auf alle Informationen, welche sie zur Ausführung benötigt. Diese Modellierungsregel hat neben dem Modell auch Auswirkungen auf die Implementierungen der Operationen. Da IP nun ein Attribute vom TemperaturTopic ist, muss eine Operation in TemperaturTopic erstellt werden, welche die IP während der Instantiierung oder des Startens setzt.

3.2.4 Total Separated Subgraph Ansatz

Der *Total Separated Subgraph* (TSS) Ansatz verwendet keine Kommunikation zur Koordination des Deployments oder zur Übertagung von Properties und Attributes. Weiterhin ermöglicht er die kleinsten Subgraphen aller Ansätze, benötigt jedoch mehrere Modellierungsregeln für die Erstellung des Gesamtmodells der zu provisionierenden Anwendung. Durch die kleinere Subgraphgröße pro Agent und dem Verzicht auf Kommunikation während des Deployments, reduziert sich die übertragene Datenmenge, dass kann den Stromverbrauch vor allem bei über Funk verbundenen Geräten reduzieren [PK00]. Jedoch lassen sich nicht alle Anwendungen unter Verwendung dieses

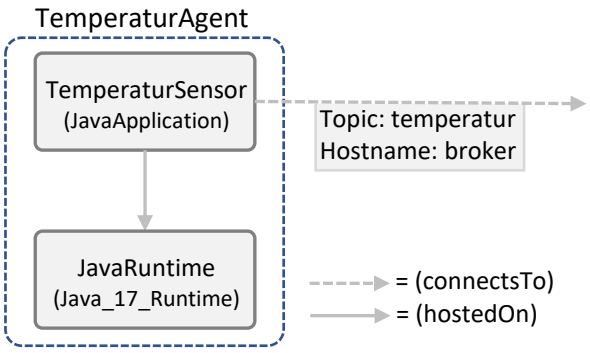

Abbildung 3.4: Der TSS des TemperaturAgent. Das ausgehende connectsTo Relationship Template hat kein Ziel Node Template. Die zwei Properties Topic und Hostname sind jetzt in dem Relationship Template enthalten. [Notation basiert auf Vino4TOSCA BBK+12]

Ansatzes modellieren. Bei dem TSS Ansatz enthält ein Subgraph eines Agenten alle lokalen Node und Relationship Templates sowie ausgehende Relationship Templates. Anders als beim RPS Ansatz werden keine Proxy Node Templates für die Ziele der ausgehenden Relationship Templates eingefügt. Abbildung 3.4 zeigt den Total Separated Subgraph des TemperaturAgent. Dieser Subgraph enthält das TemperaturSensor Node Template, das JavaRuntime Node Template und das verbindende hostedOn Relationship Template. Weiterhin ist das connectsTo Relationship Template, das im Gesamtmodell den TemperaturSensor mit dem TemperaturTopic verbindet, enthalten. Da das Ziel dieses Relationship Template nicht im Subgraph enthalten ist, stellt der Subgraph keine valide TOSCA Topologie dar. Das aus dem Subgraph erzeugte Service Template ist nicht valide und ein verarbeitender Orchestrator muss speziell angepasst werden.
Da bei diesem Ansatz Kommunikation vermieden wird, müssen alle Properties und Attributes, welche von Operationen benötigt werden, im Subgraph enthalten sein und können nicht von einem anderen Agenten abgerufen werden. Dadurch kann der TemperaturSensor alle Attributes der JavaRuntime lesen, aber keine von dem TemperaturTopic. Um Verbindungen zwischen Templates auf unterschiedlichen Agenten zu ermöglichen, müssen alle Informationen, um die Verbindung herzustellen, im TSS enthalten sein. Dazu muss eine Modellierungsregel eingeführt werden, die agentenübergreifende Property- und Attributeszugriffe verbietet. Das kann dazu führen, dass Properties an mehreren Stellen dupliziert werden müssen. Aufgrund dieser Modellierungsregel verfügt in diesem Beispiel das *connectsTo* Relationship Template über die zwei Properties „Topic“ und „Hostname“. Würde mehr als ein Relationship Template das TemperaturTopic Template als Ziel haben, dann müsste das Property Topic in alle Relationship Templates kopiert werden. Weiterhin wurde das dynamisch während des Deployments bestimmte Attribute IP wurde durch ein statisches Property namens Hostname ersetzt.
Es ist oft möglich dynamische Attribute wie IP-Adressen durch statische Properties zu ersetzen. Da es für Modellautoren nicht immer möglich ist alle externen Attributes durch lokale Properties zu ersetzen, lassen sich mit dem TSS Ansatz nicht alle Anwendungen modellieren. Weiterhin müssen die Operationen des *connectsTo* Relationship Templates über einen automatischen Retry-Mechanismus verfügen, da keine Koordination zwischen den Agenten während des Deployments stattfindet. Dadurch kann beispielsweise der TemperaturAgent nicht sicherstellen, dass das Ziel Template des *connectsTo* Relationship Templates von dem DataProcessingAgent schon instantiiert und gestartet

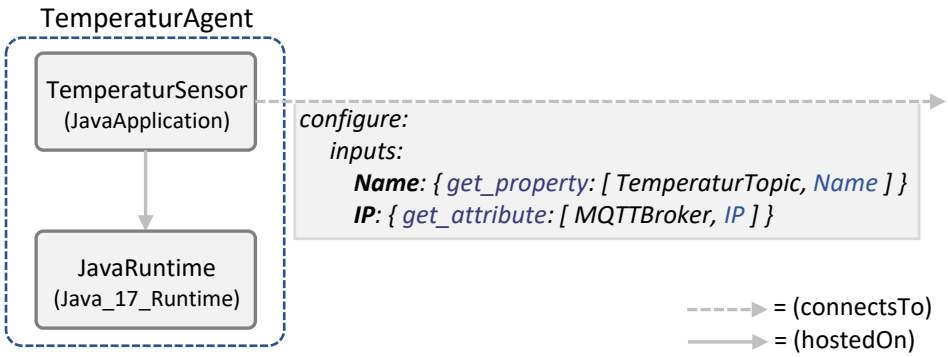

Abbildung 3.5: Der Subgraph des TemperaturAgent unter Verwendung des TSSN Ansatz. Das ausgehende connectsTo Relationship Template hat kein Ziel Node Template. Die Inputs der configure Operation werden über eine Property- und eine Attribute-Funktion von externen Node Templates gelesen. [Notation basiert auf Vino4TOSCA BBK+12]

wurde. Ein automatischer Retry-Mechanismus versucht bei einem Verbindungsfehler erneut eine Verbindung aufzubauen, bis der TemperaturTopic vom DataProcessing Agent provisioniert wurde. Die Anfragen der Retry-Versuche können gespart werden, wenn als Ziel ein externer Dienst wie ein SaaS Angebot zum Einsatz kommt. Der TSS Ansatz eignet sich nicht für Anwendungen mit vielen externen Verbindungen oder vielen dynamischen Attributen, die von anderen Agenten benötigt werden. Weiterhin kann es notwendig sein, Implementierungen von Operationen anzupassen, um zum Beispiel Hostnamen anstelle von IP-Adressen und den Retry-Mechanismus zu unterstützen. Jedoch bietet dieser Ansatz die größte Autonomie der einzelnen Agenten. Weiterhin benötigt dieser Ansatz keine Netzwerkkommunikation zwischen den Agenten für das Deployment. Darüber hinaus hat ein Agent wenig Informationen über die anderen Agenten, welche bei einer Kompromittierung gestohlen werden können. Dieser Ansatz eignet sich am besten für Anwendungen auf IoT-Geräten mit schlechter Netzwerkverbindung.

3.2.5 Total Separated Subgraph with Network Ansatz

Der letzte in dieser Arbeit präsentierte Ansatz stellt eine Erweiterung des in Abschnitt 3.2.4 präsentierten TSS Ansatzes dar. Auch bei dem Total Separated Subgraph with Network (TSSN) Ansatz wird für jeden Agenten ein individueller Subgraph generiert. Bei dem TSSN Ansatz werden die Subgraphen für die Agenten genau wie beim TSS Ansatz generiert.
Der Unterschied zwischen dem TSS und dem TSSN Ansatz besteht darin, dass im TSSN Ansatz Attribute und Properties von anderen Agenten abgefragt werden können. Für den TemperaturAgent, entsteht beispielsweise der in Abbildung 3.5 abgebildete Subgraph. Der Subgraph enthält die Node Templates TemperaturSensor und JavaRuntime, sowie das *hostedOn* Relationship Template zwischen den beiden Knoten. Weiterhin wurde das *connectsTo* Relationship Template zwischen dem TemperaturSensor und dem TemperaturTopic hinzugefügt. Zum Verbinden benötigen die Operationen des *connectsTo* Relationship Templates den Namen des Topics und die IP des MQTT Brokers. Beide Inputs sind als Property oder Attribute Funktionen definiert, welche die jeweiligen Werte lesen. Die Resultate der Auswertung von Funktionen können selbst wieder eine Funktion darstellen. Die *get_property* Funktion liest das Property „Name“ von dem Node Template „TemperaturTopic“

und die *get_attribute* Funktion liest das Attribute IP von dem Node Template *MQTTBroker*. Beide Informationen sind nicht im TSSN enthalten. Dadurch muss, um das Property Namen und das Attribute IP zu lesen, eine Anfrage über das Netzwerk gesendet werden.
Diese könnte von dem DataProcessingAgent beantwortet werden, sobald die Informationen während des Deployments vorliegen. Properties sind schon vor dem Deployment bekannt, es könnte also sofort geantwortet werden. Für Attribute ist das nicht möglich, da diese meistens erst während des Deployments, von einer Operation bestimmt werden. Weiterhin könnte der angefragte Agent antworten, nachdem eine Operation das Attribute erstmalig gesetzt hat, jedoch können nachfolgende Operationen das Attribute überschreiben. Wird mit der Antwort für Attributes gewartet, bis das Ziel Node Template instantiiert und gestartet wurde, ist sichergestellt, dass die Operationen *create*, *configure* und *start* des Lifecycle Interfaces schon ausgeführt wurden. Es ist möglich Attributes auch durch zum Beispiel die *stop* Operation zu setzen, welche erst beim Freigeben des Node Templates ausgeführt wird. Aber die Spezifikation beschreibt Attributes als den tatsächlichen Zustand einer TOSCA Entität, nach dem diese provisioniert und instantiiert wurde [OAS20b]. Aus diesem Grund wird im Weiteren davon ausgegangen, dass Attributes nach dem Instanziieren und Starten des Templates den gewünschten Wert erhalten haben. Wird die Antwort für Properties und Attributes erst nach der Instantiierung und dem Starten des Ziel Node Templates gesendet, entsteht zusätzlich eine implizite Koordination des Deploymentprozesses. Dadurch kann sichergestellt werden, dass bei einer connectsTo Verbindung das Ziel- und Ursprungs-Node-Template instantiiert und gestartet wurden, bevor die Verbindung aufgebaut wird. Dazu muss das Relationship Template ein Property oder Attribute des Ziel Node Templates lesen.
Zum Beispiel beginnt der TemperaturAgent nach dem Deployment des TemperaturSensors mit dem Deployment des *connectsTo* Relationship Templates. Da die *configure* Operation in diesem Beispiel, das Property Name und das Attribute IP liest, muss eine Anfrage über das Netzwerk zu dem DataProcessingAgent gesendet werden. Der DataProcessingAgent antwortet nach dem Deployment des MQTTBroker Node Templates mit der IP und nach dem Deployment des TemperaturTopics mit dem Wert des Property Name. Da die Antwort für das Property Name erst nach dem erfolgreichen Deployment des Ziel Node Templates abgesendet wird und der TemperaturAgent mit dem Deployment des Relationship Templates auf die Antwort warten muss, ist sichergestellt die korrekte Deploymentreihenfolge eingehalten wird. Es müssen immer erst das Ziel- und das Ursprungs-Node-Template provisioniert werden, bevor ein Relationship Template gestartet werden kann.
Bei diesem Ansatz wird angenommen, dass jedes connectsTo Relationship Template mindestens ein Property oder Attribute des Ziel Node Templates als Input benötigt. Falls ein Relationship Template keine Inputs vom Ziel Node Template benötigt, der Modellautor aber trotzdem eine Deploymentreihenfolge sicherstellen möchte, kann er ein Property in dem Ziel Node Template anlegen. Das neue Property wird dann als Input für eine der Relationship Template Operationen gesetzt. Dieser Ansatz hat den Vorteil, dass Operationen des connectsTo Relationship Template, welche keine oder nur innerhalb des TSSN verfügbare Inputs benötigen, schon ausgeführt werden, während parallel der Agent des externen Node Templates sein Deployment vornimmt. Koordination passiert nur, wenn eine Datenabhängigkeit zwischen den individuellen Subgraphen der Agenten besteht. Sonst kann jeder Agent autonom arbeiten.
Die Property und Attribute Funktionen spezifizieren nicht, welchem Agenten ein Node Template zugeordnet ist. Weiterhin ist diese Information auch nicht in den Subgraph enthalten. Daher müssen einzelne Node und Relationship Templates im Kommunikationssystem über ihre Namen adressiert werden können, um Anfragen für Properties und Attributes zu senden. Die Zuordnung zwischen den Agenten und den Node Templates wird dem Subgraph aus Platz- und Sicherheitsgründen

nicht hinzugefügt. So müssen die Zuordnungen zwischen Node Templates und Agenten nicht über begrenzte Netzwerkkapazitäten übertragen werden und ein Angreifer, der zum Beispiel ein IoT-Gerät stiehlt, erhält weniger Informationen über die Struktur der Gesamtanwendung.
Der TSSN Ansatz hat den Vorteil, dass initial wenig Daten übertragen werden müssen und externe Properties oder Attribute nur bei Bedarf übertragen werden. Weiterhin wird eine implizite Deploymentkoordination über die Datenabhängigkeiten der einzelnen Operationen realisiert. Operationen ohne Abhängigkeiten können auf unterschiedlichen Agenten parallel ausgeführt werden. In dem Beispiel in Abbildung 3.5 benötigt nur die configure Operation des Relationship Templates Informationen von außerhalb des Subgraphs. Eine eventuell vorhandene *create* Operation, welche zum Beispiel benötigte Software installiert, kann unabhängig nach dem Deployment des Temperatur-Sensor Node Templates ausgeführt werden. Die Operation muss nicht auf das TemperaturTopic Node Template warten. Auch sind für diesen Ansatz im Gegensatz zum TSS Ansatz weniger Modellierungsregeln zu beachten und Implementierungen von Operationen müssen nicht angepasst werden. Da Properties und Attributes wie Passwörter oder Verbindungsinformationen erst beim Deployment abgerufen werden, können kompromittierte Geräte vor dem Deployment gesperrt werden und erhalten dadurch keinen Zugriff auf diese Informationen.
Weiterhin lassen sich über Regeln in der Messaging Middleware der Zugriff auf einzelne oder alle Properties und Attributes eines Node Templates für bestimmte Agenten sperren oder nur zu bestimmten Zeiträumen während des Deployments erlauben. Da bei diesem Ansatz im Subgraph keine Ziel Node Templates von den externen Verbindungen enthalten sind, repräsentieren diese keine valide TOSCA Topologie. Der TSSN Ansatz erlaubt, dass diese Subgraphen im Kontext der Gesamttopologie Zugriff auf Attributes und Properties haben, obwohl die Gesamttopologie für den lokalen im Agenten eingebauten Orchestrator nicht sichtbar ist.

3.2.6 Vergleich

In diesem Abschnitt werden die vorherigen Ansätze aus den Abschnitten 3.2.1 bis 3.2.5, um einen individuellen Subgraphen für jeden Agenten aus einem ganzen Service Template zu generieren, miteinander vergleichen. Die gezeigten Lösungen haben unterschiedliche Vor- und Nachteile. Manche Ansätze eignen sich für bestimmte Anwendungsszenarien besser als andere. Für manche muss ein Service Template Autor bestimmte Modellierungsregeln bei der Erstellung des Gesamtmodells beachten. Andere benötigen weniger Netzwerkkommunikation. Die Bewertungskriterien für den Vergleich sind aus den Zielen dieser Arbeit (siehe Abschnitt 1.1) abgeleitet und im Folgenden näher beschrieben. Die Ansätze werden in der Tabelle 3.1 anhand der Bewertungskriterien verglichen.

Benötigt keine Vorverarbeitung oder zentralen Agent? Dieses Kriterium beschäftigt sich mit der Frage, ob ein Service Template in einem Vorverarbeitungsschritt in mehrere Subgraphen aufgeteilt werden muss. Das Aufteilen auf mehrere Subgraphen kann vor dem Deployment von einem Modellierungswerkzeug oder während des Deployment von einem Agenten ausgeführt werden.

Unterstützt Deploymentkoordination? Viele Anwendungsszenarien benötigen eine Koordination dies Deploymentvorgangs über mehrere Agenten hinweg. Bei *connectsTo* Relationship Templates zwischen Node Templates auf verschiedenen Agenten muss sichergestellt werden, dass Ziel- und Ursprungs-Node-Template bereits gestartet wurden, bevor die Verbindung aufgebaut

wird. In manchen Fällen kann auf eine Koordination des Deploymentprozesses verzichtet werden, indem das Gesamtmodell und die Implementierungen der jeweiligen Operationen der Relationship Templates angepasst werden. Dies ist aber nicht immer möglich und daher sollte es ein Ansatz ermöglichen, den Deploymentvorgang über mehrere Agenten hinweg zu koordinieren.

Müssen Service Template Autoren spezielle Modellierungsregeln befolgen? Entwickler, die Service Templates für das Deployment mit Agenten konzipieren, sollten so wenig wie möglich zusätzliche Regeln für die Modellierung einer Anwendung befolgen müssen. So bleiben Service Templates zwischen verschiedenen Orchestrator Implementierungen portable. Auch müssen Entwickler, die bereits Service Templates erstellen können, weniger zusätzliche Modellierungsregeln erlernen und existierende Service Templates können einfacher für das Deployment mit Agenten angepasst werden. Wenn wenig spezielle Modellierungsregeln benötigt werden, können auch für TOSCA 2.0 entwickelte Modellierungswerkzeuge mit keinen oder nur wenigen Anpassungen verwendet werden. Am besten ist, wenn keine zusätzlichen Modellierungsregeln notwendig sind und existierende Service Templates übernommen werden können. So benötigt zum Beispiel der TSS Ansatz starke Anpassungen, da keine Attribute von anderen Agenten abgerufen werden können. Beim kein Split Ansatz dagegen sind keine Anpassungen notwendig, da Service Templates direkt an die Agenten weitergegeben werden. Das Hinzufügen von Agenten zu Node Template Zuordnungen wird bei dieser Betrachtung nicht als zusätzliche Modellierungsregel betrachtet, da diese Anpassung bei allen Ansätzen notwendig ist. Modellierungsregeln schränken Autoren bei der Erstellung des initialen Service Templates ein, um das Generieren von Subgraphen zu ermöglichen. Sie werden daher bei der Erstellung des gesamten Service Template angewendet und nicht auf das Erstellen der jeweiligen Subgraphen selbst.

Müssen Operationsimplementierungen angepasst werden? Dieses Kriterium beschreibt, ob Änderungen an der Implementierung der Operationen oder den Anwendungskomponenten nötig sind, welche für die Verwendung mit einem anderen Orchestrator nicht notwendig sind. Zum Beispiel müssen Anwendungen im TSS Ansatz so entwickelt sein, dass sie keine Attributes von externen Node Templates benötigen.

Kann jeder Anwendungsfall modelliert werden? Es ist nicht mit jedem Lösungsansatz möglich jeden Anwendungsfall zu modellieren, der sich auch mit der uneingeschränkten TOSCA Spezifikation modellieren lässt. Manche Einschränkungen, die für die jeweiligen Ansätze gelten, machen es unmöglich bestimmte Anwendungen zu modellieren. Beim RPS Ansatz ist es zum Beispiel nicht möglich, beliebige externe Attributes zu lesen. Benötigt eine Anwendung Zugriff auf im Graph weit verteilte Attributes, lässt sich diese Anwendung nicht modellieren. Weiterhin können Anwendungen, die eine exakte Deploymentreihenfolge benötigen, nicht mit dem TSS Ansatz umgesetzt werden, da hier keine Koordination der Agenten stattfindet.

Initiale Übertragungsgröße Die initiale Übertragungsgröße beschreibt, wie viele Daten für jeden Subgraph an jeden Agenten im Vergleich mit den anderen Lösungsansätzen übertragen werden müssen. Die genauen Unterschiede hängen vom modellierten Anwendungsszenario ab. Wenn ein Service Template provisioniert werden soll, welches jedem Agenten nur ein einziges Node Template zuweist und alle Node Templates über Relationship Templates verbunden sind, dann gibt es zum

Beispiel keinen Unterschied zwischen der initialen Übertragungsgröße von dem PRS Ansatz und dem RPS Ansatz. Weiterhin gibt es keine Unterschiede zwischen den einzelnen Ansätzen, wenn nur ein Agent zum Einsatz kommt.

Kommunikationsaufwand während der Ausführung des Deployments Der Kommunikationsaufwand während des Deployments beschreibt, wie oft Agenten untereinander kommunizieren müssen, um ein Deployment erfolgreich abzuschließen. Die initiale Datenmenge zum Übertragen der individuellen Subgraphen auf die Agenten wird bei diesem Kriterium nicht erfasst. Diese wird von dem vorherigen Kriterium abgedeckt. Bei dieser Betrachtung hat jeder Agent den initialen Subgraph schon erhalten und der Vorgang zur Ausführung des Deployments wurde gestartet. Bei allen Ansätzen außer bei dem TSS Ansatz müssen zum Beispiel Nachrichten ausgetauscht werden, um das Deployment einer connectsTo Verbindung zwischen zwei Agenten zu koordinieren. Für den TSS Ansatz fällen keine Übertragungen während des Deployments an, da dieser Ansatz auf starke Autonomie der einzelnen Agenten ausgelegt ist. Beim TSSN Ansatz sind je nach Anwendungsfall mehr Übertragungen notwendig, als bei anderen Ansätzen, da nicht schon ein Teil der externen Properties in der initialen Übertragung des Subgraphs enthalten sind. Da diese Übertragungen jedoch auch zur Koordination des Deployments eingesetzt werden, hängt die Menge an Daten, die zusätzlich übertragen werden muss, von der modellierten Anwendung ab. Je nach Anwendungsfall nehmen die Nachrichten zur Koordination und zum Property und Attribute Austausch aber nur wenig Übertragungskapazität im Vergleich zu dem Subgraphen, den Implementation Artifacts und den Deployment Artifacts ein.

Menge an Informationen, auf die ein einzelner Agent Zugriff hat? Ein Agent muss immer alle lokalen Node Templates und deren Relationship Templates kennen, sonst kann er diese nicht provisionieren. Die Ansätze unterscheiden sich bei der Menge an Informationen, die ein Agent über die Node Templates der anderen Agenten erhält. Sind viele Informationen lokal verfügbar, sind autonomere Deployments möglich, da zum Beispiel Properties nicht über Netzwerk von einem anderen Agenten abgefragt werden müssen. Falls weniger Informationen lokal verfügbar sind, können weniger Informationen kompromittiert werden und es müssen auch weniger Daten auf einen Agenten übertragen werden.

Autonomie während des Deployments Ein Agent berechnet aus seinem lokalen deklarativen Modell seinen individuellen Deploymentplan. Eine Subgraph der keine externen Verbindungen hat und keine externen Informationen benötigt, kann von einem Agenten allein ohne Interaktion mit anderen Agenten provisioniert werden. Mehr Autonomie hat den Vorteil, dass Agenten weniger Kommunikation benötigen und der Deploymentvorgang zuverlässiger und schneller abläuft. Netzwerkkommunikation ist besonders in IoT Umgebungen fehleranfällig und langsam. Wird mehr Kommunikation benötigt, kann das außerdem den Energieverbrauch eines IoT Geräts steigern [PK00]. Weiterhin könnten IoT-Geräte offline oder in einem Energiesparmodus sein und damit Deployments, die im Netzwerk ausgeführt werden, durch Abhängigkeiten blockieren. Bei dem TSS Ansatz muss nie auf einen anderen Agenten gewartet werden, es müssen keine Properties oder Attributes von anderen Agenten abgefragt werden und es wird auch keine Koordination benötigt.

	Kein Split	**Proxy Replacement Subgrap**	**Reduced Proxy Subgraph**	**Total Separated Subgraph**	**Total Separated Subgraph with Network**
Benötigt keine Vorverarbeitung oder zentralen Agent?	✓	✗	✗	✗	✗
Unterstützt Deploymentkoordination?	✓	✓	✓	✗	✓
Müssen Service Template Autoren spezielle Modellierungsregeln befolgen?	keine	keine	wenige	viele	wenige
Müssen Operationsimplementierungen angepasst werden?	✗	✗	✓	✓	✗
Kann jeder Anwendungsfall modelliert werden?	✓	✓	✗	✗	✓
Initiale Übertragungsgröße	hoch	hoch	mittel	gering	gering
Kommunikationsaufwand während der Ausführung des Deployments	gering	gering	gering	keiner	gering
Menge an Informationen, auf die ein einzelner Agent Zugriff hat?	sehr hoch	hoch	wenig	sehr wenig	sehr wenig
Autonomie während des Deployments	mittel	mittel	mittel	hoch	mittel*
Können Attribute anderer Agenten gelesen werden?	✓	✓	teilweise	✗	✓
Implementierungskomplexität	gering	mittel	mittel	gering	mittel

Tabelle 3.1: Vergleich der einzelnen Modellierungsansätze aus den Abschnitten 3.2.1 bis 3.2.5. *Je nach Anwendungsfall kann die Autonomie auch geringer ausfallen, zum Beispiel, wenn viele Properties von anderen Agenten abgefragt werden, ohne, dass diese gleichzeitig auch zu Koordination genutzt werden können.

Können Attribute anderer Agenten gelesen werden? Für die Provisionierung jedes Node oder Relationship Templates kann jedes Attribute im Service Template benötigt werden. Deshalb müssen Attributes entweder zwischen Agenten ausgetauscht werden können oder die Verwendung von Attributes muss über Modellierungsregeln eingeschränkt werden.

Implementierungskomplexität Komplexere Lösungen benötigen mehr Aufwand bei der Konzipierung, Implementierung, Wartung und beim Testen. Die Ansätze sollten einfach zu implementieren sein, da es zum Beispiel notwendig ist, die Agentensoftware für mehrere Plattformen zu implementieren.

3.2.7 Fazit

Die vorgestellten Ansätze haben Vor- und Nachteile für unterschiedliche Anwendungsszenarien. Der TSS Ansatz bietet sich an, wenn die Netzwerkkommunikation reduziert werden soll. Dieser Ansatz ist für Anwendungen, welche viele Verbindungen haben, ungeeignet. Der RPS Ansatz benötigt kleinere Subgraphen als der PRS Ansatz, erfordert aber spezielle Regeln bei der Erstellung des Service Templates. Der geeignetste Ansatz, um die Ziele dieser Arbeit zu erreichen, ist der TSSN Ansatz, der in Abschnitt 3.2.5 beschrieben wird. Dieser benötigt wenig Modellierungsregeln, dadurch ist das Portieren von existierenden TOSCA Modellen einfacher. Weiterhin lassen sich alle Anwendungen, die auch mit TOSCA modelliert werden können, mit diesem Ansatz umsetzen. Auch sind für den TSSN Ansatz keine Anpassungen an existieren Operationsimplementierungen notwendig. Welcher Ansatz insgesamt mehr Daten für ein Deployment übertragt, hängt stark vom Anwendungsfall ab. Agenten, die nach dem TSSN Ansatz arbeiten, übertragen die benötigten Properties erst während des Deployments. Jede Anfrage für ein Property benötigt aber zwei Nachrichten. Diese Nachrichten können je nach Anwendungsfall gleichzeitig für die Koordination des Deployments eingesetzt werden. Andere Ansätze integrieren die Properties in die initiale Übertragung des Subgraphen zum Agenten. Das spart den Overhead einer Nachrichtenübertragung für jedes Property. In dem Subgraph können je nach modellierter Anwendung auch nicht benötigte Properties enthalten sein. Es kommt auf den Anwendungsfall an, ob die unnötigen Properties oder der Nachrichtenoverhead mehr Daten benötigt.
Nachteilig an dem TSSN Ansatz gegenüber dem kein Split Ansatz ist, dass aus dem Service Template mehrere Subgraphen erstellt werden müssen. Dadurch wird ein weiterer Verarbeitungsschritt zwischen dem Modellieren und dem Deployment notwendig. Durch diesen weiteren Schritt ist es aber möglich, dass jeder einzelne Agent nur Zugriff auf so wenig Informationen wie möglich erhält. Ein Agent erhält keine Kenntnis über die Gesamtstruktur der Anwendung oder Properties, die er nicht direkt benötigt. Auch ist es jederzeit möglich den Zugang zu Properties von anderen Agenten zu sperren, da diese nicht initial im Subgraph enthalten sind. Aufgrund dieser Vorteile wurde der TSSN Ansatz für Realisierung der Konzepte und des Prototypen in dieser Arbeit ausgewählt.

3.3 Kommunikationsarchitektur

Um den in Abschnitt 3.2.5 beschriebenen und in Abschnitt 3.2.7 ausgewählten Ansatz umzusetzen, müssen die Agenten untereinander kommunizieren können. Daher wurde die im Folgenden beschriebe Kommunikationsarchitektur entwickelt. Sie ermöglicht den am Deployment teilneh-

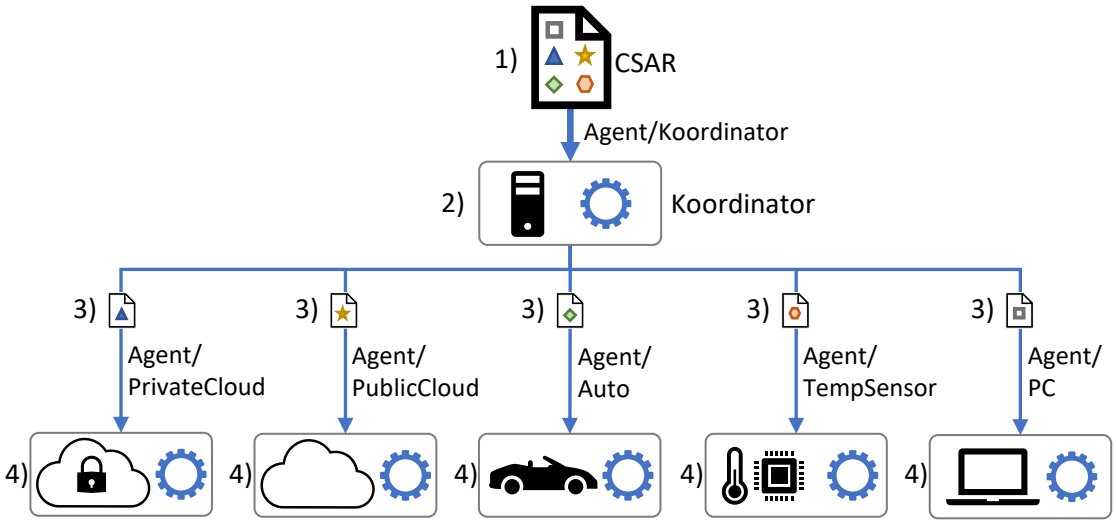

Abbildung 3.6: Kommunikationsübersicht eines Deployments, dass von mehrere Agenten ausgeführt wird. Ein blaues Zahnrad repräsentiert einen Agenten. Ein Icon neben dem Zahnrad repräsentiert den Ort, an dem der Agent installiert ist. Beschriftungen neben den Pfeilen repräsentieren MQTT Topics.

menden Agenten eine zuverlässige und asynchrone Kommunikation. Nachrichten dürfen nicht verloren gehen, sonst funktioniert die Koordination des Deployments nicht und Agenten könnten ihr Deployment nicht abschließen. Wenn Nachrichten nicht ankommen, kann ein Agent zum Beispiel nicht verifizieren, dass alle Deploymentvorbedingungen für ein Relationship Template erfüllt sind. Weiterhin bekommt der Agent nicht alle Inputdaten, um lokale Operationen des Relationship Templates auszuführen. Deshalb wird in dieser Arbeit das MQTT Protokoll mit mindestens dem Quality of Service 1 „At least once delivery“ [CBB+19] verwendet. Weiterhin bietet MQTT durch eine asynchrone nachrichtenbasierte Kommunikation eine Entkoppelung von Sender und Empfänger [EFGK03]. Ein Agent kann eine Nachricht senden, während der Empfänger gerade in einem Energiesparmodus ist oder keine Netzwerkverbindung hat. Der Empfänger kann antworten, wenn er wieder über eine Internetverbindung verfügt. Da Sicherheitsmaßnahmen, Firewalls und NAT eingehende Kommunikation verhindern können, ist es Ziel dieser Arbeit eine Kommunikationsarchitektur zu entwickeln, die ohne eingehende Netzwerkverbindungen für die Agenten auskommt. Um Kommunikation ohne eingehende Netzwerkverbindungen für Agenten zu ermöglichen, wird ähnlich wie bei dem hybriden Konzept von Képes et al. [KBL+19] ein Message Broker eingesetzt, der Nachrichten zwischen den Agenten vermittelt. Alle Agenten müssen diesen Broker erreichen können, brauchen dafür jedoch nur ausgehende Verbindungen. Das führt dazu, dass verschiedene Organisationen bei einem Deployment keine Zugänge in ihren Sicherheitsbeschränkungen öffnen müssen. Der Message Broker erlaubt Kommunikation zwischen den Agenten ohne, dass die Teilnehmer den Standort ihrer Infrastruktur, die IP-Adressen oder sonstige Informationen teilen. Die Agenten müssen nur eine gemeinsame Topicstruktur und den Message Broker zur Kommunikation nutzen.

Um die weiteren Kommunikationsanforderungen zu ermitteln, wird im Folgenden ein TSSN-basierter Deploymentvorgang beispielhaft beschrieben. Dieser ist in Abbildung 3.6 abgebildet. Der Vorgang beginnt in Schritt 1) mit der Übermittlung der in TOSCA modellierten Anwendung in einem CSAR an den koordinierenden Agenten. Der CSAR enthält ein Service Template und alle Artifacts, die für das Deployment benötigt werden. Der Koordinator teilt in Schritt 2) die Anwen-

dungstopologie nach dem in Abschnitt 3.2.5 beschrieben TSSN Ansatz in mehrere Subgraphen auf. Außerdem ermittelt er pro Subgraph alle Artefakte und Typen, welche für die Ausführung benötigt werden und erstellt für jeden Agenten einen individuellen CSAR. Der koordinierende Agent kann bei jedem Deployment individuell und dynamisch ausgewählt werden und kann auch selbst ein lokales Deployment von Node Templates vornehmen. Der Koordinator ist bei jedem Deployment der Agent, welcher das komplette Service Template empfängt. In Schritt 3) übertragt der koordinierende Agent alle individuellen CSARs zu den jeweiligen auf den Geräten und Instanzen installierten Agenten. Jeder CSAR enthält unter anderem die lokalen Node und Relationship Templates des Zielagenten. Die hier provisionierte Anwendung enthält zum Beispiel Templates für den PrivateCloud Agent, den PublicCloud Agent und den Auto Agent. Schritt 4) umfasst das lokale Deployment der Subgraphen durch die Agenten. Dafür wird lokal ein Workflow berechnet und die für das Deployment notwendigen Operationen ausgeführt. Auch während des Deployments ist Kommunikation zwischen den Agenten notwendig, um unter anderem Attributes und Properties zu übertragen und um das Deployment zu koordinieren.
Der TSSN Ansatz kombiniert das Übertragen von Werten mit der Koordination des Deployments. Um das beschriebene Deployment ausführen zu können, muss jeder Agent CSARs empfangen und der Koordinator zusätzlich noch CSARs senden können. Weiterhin muss jeder Agent Anfragen für Properties und Attributes an jeden anderen Agent senden können. Ein einziger Topic würde ausreichen, dass alle Agenten Nachrichten austauschen können. Das hätte aber den Nachteil, dass jede Nachricht immer an jeden Agenten übertragen wird. So wird jeder individuelle CSAR und alle Anfragen für zum Beispiel Attributes an alle Agenten übertragen. Dadurch müssen jeweils alle außer ein Agent die Nachrichten verwerfen. Darüber hinaus könnte ein kompromittierter Agent den CSAR von jedem anderen Agenten empfangen. Da CSARs sensible Informationen wie Zugangsdaten oder Implementierungsdetails enthalten können, muss eine Topicstruktur verwendet werden, die CSARs und Anfragen für Properties und Attribute nur ihren Zielagenten zugänglich macht.

Um Nachrichten an individuelle Agenten senden zu können, subscribed sich jeder Agent auf den Topic „Agent/<AgentName>“. Der Platzhalter „<AgentName>“ wird dabei von jedem Agenten durch seinen eigenen Namen ersetzt. Die Abbildung 3.6 enthält die verwendeten Topics für alle Agenten im Beispiel. Das Service Template des Koordinators enthält eine Zuordnung zwischen jedem Node Template und dem zuständigen Agenten. Daher kann der Koordinator bestimmen, welchen Subgraph er an welchen Topic senden muss. Dadurch werden keine unnötigen Daten übertragen. Auch erhalten Agenten kein Zugriff auf Daten, welche sie nicht benötigen. Werden Artifacts in mehreren Subgraphen verwendet, können diese auch über eine URL eingebunden werden. Dadurch sind die Artifacts nicht direkt im CSAR enthalten, sondern werden von den Agenten über HTTP abgerufen. Dadurch können HTTP Caching Mechanismen[2] eingesetzt werden, um die benötigte Datenübertragungsmenge zu reduzieren. Nehmen zum Beispiel viele Geräte aus einer Organisation an einem Deployment teil, kann in der Organisation ein HTTP Caching Proxy eingesetzt werden. Das gleiche Verfahren kann für mehrfach verwendete CSARs eingesetzt werden. Während des Deployments müssen Agenten nicht nur mit dem Koordinator, sondern auch untereinander kommunizieren, um zum Beispiel Properties oder Attributes auszutauschen. Die Agenten in dem TSSN Ansatz haben keine Informationen über die anderen Agenten. Deshalb kann ein Agent nicht wissen, welcher andere Agenten kontaktiert werden muss, um ein bestimmtes Property oder Attribute eines externen Templates abzufragen. Das hat den Vorteil, dass nicht die komplette Zuordnung zwischen Node Templates und Agenten in jedem individuellen CSARs

[2] https://www.rfc-editor.org/rfc/rfc7234 (zuletzt besucht 28.09.2022)

übertragen werden muss. Daher wird in dieser Arbeit ein Adressierungsverfahren präsentiert, das auf Node und Relationship Templatenamen basiert, um mit einzelnen Agenten zu kommunizieren. Da Node und Relationship Templates nur innerhalb eines CSARs eindeutig sein müssen und mehrere CSARs gleichzeitig provisioniert sein können, muss auch der Name des CSARs in dem Adressierungsverfahren verwendet werden. Deshalb werden für Property und Attribute Anfragen die Topics „<CsarName>/<TemplateName>“ verwendet. Benötigt zum Beispiel der Agent „TempSensor“ aus Abbildung 3.6 ein Property von dem Node Template „TemperaturProcessor“ im CSAR „TemperaturGatherApp“ sendet er eine Anfrage an „TemperaturGatherApp/TemperaturProcessor“. Um Property und Attribute Anfragen erhalten zu können, subscribed sich jeder Agent für jedes Template auf den Topic „<CsarName>/<TemplateName>“. Für eine Antwort wartet der Agent wie vom TSSN Ansatz vorgesehen, bis das angefragte Node Template instantiiert und gestartet wurde. Um die Antwort an den richtigen Agenten zu senden, wird der in der MQTT Spezifikation [CBB+19] beschriebene Request / Response Mechanismus verwendet. Dazu enthält die Anfrage ein „Response Topic“ und „Correlation Data“. Der Response Topic wird für jeden Agenten individuell konfiguriert und kann zum Beispiel zufällig gewählt werden. Dadurch erhält ein Agent durch den Response Topic keine Informationen über die Zuordnung zwischen Agenten und Node Templates. Die Correlation Data sind zufällig vom Sender generiert und helfen die Antwort einer Anfrage zuzuordnen.

3.3.1 Mehrfachdeployment

Die in Abschnitt 3.3 beschriebene Kommunikationsarchitektur erlaubt, dass mehrere Anwendungen in unterschiedlichen CSARs parallel auf Agenten provisioniert werde können. Jedoch ist es nicht möglich, dass der gleiche CSAR mehrmals provisioniert wird. Oft kann es notwendig sein, den gleichen CSAR mehrfach zu provisionieren, um mehrere Instanzen einer Anwendung zu erhalten. Um das zu ermöglichen, muss die Kommunikationsarchitektur erweitert werden. Um ein Deployment eines CSARs erneut zu ermöglichen, müssen die im Service Template verwendeten Agenten mehrfach existieren. Dadurch gibt es mehrere Agenten und Node sowie Relationship Templates mit dem gleichen Namen. Das führt in der beschriebenen Topicstruktur zu Kollisionen und Nachrichten können nicht mehr dem richtigen Agenten zugeordnet werden. Um Kollisionen bei der Nutzung von Topics zu vermeiden, wird die Topicstruktur um ein weiteres Level erweitert. Dazu wird vor jedem verwendeten Topic eine DeploymentID angefügt. So werden mit dieser Erweiterung die Topics „<DeploymentID>/<CsarName>/<TemplateName>“ und „<DeploymentID>/Agent/<AgentName>“ verwendet. Die DeploymentID ist ein String, der alle Agenten einer Anwendung einem Deployment zuordnet. Dadurch kann der gleiche CSAR mehrfach provisioniert werden und mehrere Agenten können mit dem gleichen Namen existieren. Die DeploymentID wird dem Agenten zusammen mit dem Agentennamen beim Starten übergeben. Durch diese Erweiterung kann zum Beispiel die gleiche Anwendung an mehreren Standorten provisioniert werden. Dazu kann der Name des Standorts als DeploymentID verwendet werden. So wird dann ein auf einem Roboter installierter Agent am Standort Stuttgart den Topic „Stuttgart/Agent/Roboter“ nutzen. Die DeploymentID ist frei wählbar, muss aber auf einem MQTT Broker einzigartig sein.

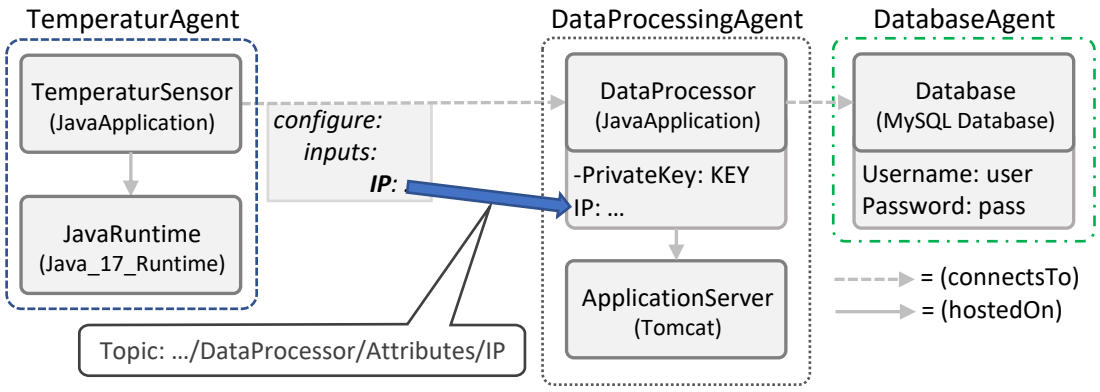

Abbildung 3.7: Vereinfachte Darstellung des Zugriffs auf das Attribute „IP“ von dem Temperatur-Agent über ein MQTT Topic. Das Attribute wird von der configure Operation des connectsTo Relationship Template benötigt. [Notation basiert auf Vino4TOSCA BBK+12]

3.3.2 MQTT-basiertes Sicherheitskonzept

Ein Vorteil der Erstellung von separaten Subgraphen für jeden Agenten ist neben den Einsparungen beim Übertragungsvolumen, dass ein kompromittierter Agent durch die Übertragung seines Subgraphen kein Zugang zu Properties und Attributes in anderen Teilen der Topologie erhält. Durch die in Abschnitt 3.3 beschriebene Kommunikationsarchitektur kann ein kompromittierter Agent aber alle Properties und Attributes von anderen Agenten abfragen. Darüber hinaus könnte ein kompromittierter Agent den MQTT Wildcard Topic „#“ nutzen und alle CSARs, Properties und Attributes, die ausgetauscht werden, empfangen. Um das zu verhindern, muss in dem Broker eine Liste der erlaubten Topics für jeden Agenten hinterlegt sein. Die genaue Konfiguration um Agenten den Zugriff auf bestimmte Topics zu erlauben oder verbieten ist Broker spezifisch, aber die MQTT Spezifikation empfiehlt, dass Broker diese Funktionalität implementieren [CBB+19]. Bei dem verbreiteten Eclipse Mosquitto[3] Broker heißt dieses Feature „Access Control List“ (ACL) [KGR20; Lig]. Weiterhin sollten Verbindungen zum MQTT Broker verschlüsselt und authentifiziert sein. Über eine ACL kann kontrolliert werden, welcher MQTT Client Zugriff, auf welchen Topic hat. Durch die in Abschnitt 3.3 beschriebene Topicstruktur lässt sich der Zugriff auf bestimmte Node oder Relationship Templates kontrollieren.
Das in Abbildung 3.7 abgebildete Beispiel zeigt eine vereinfachte Version des in Abbildung 3.1 abgebildete Service Templates. Der TemperaturSensor baut eine verschlüsselte Verbindung zu dem DataProcessor auf, wofür das Relationship Template die IP als Input für die *configure* Operation benötigt. Das Property „PrivateKey“ wird vom DataProcessor selbst für die Verschlüsselung benötigt und darf nicht kompromittiert werden. Darüber hinaus benötigt der TemperaturAgent keinen Zugriff auf den Username und das Passwort der Database. Diese Anmeldedaten benötigt nur der DataProcessingAgent für sein connectsTo Relationship Template. Eine ACL Regel kann den Zugriff des TemperaturAgents auf das DataProcessor Node Template begrenzen. Dafür wird eine ACL Regeln angelegt, die dem TemperaturAgent ausschließlich Schreibzugriff auf den Topic „<csarName>/DataProcessor“ gewährt. Dadurch kann er Anfragen für die Properties und Attributes

[3] https://mosquitto.org/ (zuletzt besucht 28.09.2022)

des DataProcessor Node Templates stellen. Zusätzlich muss der Agent seinen eigenen Response Topic lesen und wenn notwendig auf die Response Topics der anderen Agenten schreiben können. Weiterhin benötigt der TemperaturAgent Lesezugriff auf den Topic „Agent/TemperaturAgent“ um CSARs empfangen zu können. Außerdem muss der koordinierende Agent Nachrichten auf alle Topics aus dem Pattern „Agent/<AgentName>“ senden können, um die individuellen CSARs auf die einzelnen Agenten zu übertragen.
Durch diese Konfiguration hat ein kompromittierter TemperaturAgent kein Zugriff auf den Username oder das Passwort der Database. Aber der PrivateKey des DataProcessors kann gelesen werden. Um ein erweitertes Berechtigungskonzept zu ermöglichen und den Zugriff auf einzelne Properties oder Attribute beschränken zu können, muss die Topicstruktur erweitert werden.
Um das erweiterte Berechtigungskonzept umzusetzen, kann die Topicstruktur „<csarName>/<TemplateName>“ um zwei weitere Topic Levels erweitert werden. Das erste weitere Level erlaubt Zugriffsbeschränkungen auf entweder alle Properties oder alle Attributes umzusetzen. Das zweite Level steuert den Zugriff auf einzelne Werte. In der erweiterten Topicstruktur werden Attribute über den Topic „<csarName>/<TemplateName>/Attributes/<AttributeName>“ und Properties über „<csarName>/<TemplateName>/Properties/<PropertyName>“ gelesen. Ein Agent kann so zum Beispiel Zugriff auf alle Attributes eines Node Template erhalten, wenn die Konfiguration dem Agenten Zugang zum Topic „<csarName>/<NodeTemplateName>/Attributes/#“ erlaubt. Gibt es mehrere Attributes und ein Agent soll nur Zugriff auf ein einzelnes erhalten, so kann nur der Zugriff zu dem Topic „<csarName>/<NodeTemplateName>/Attributes/<AttributeName>“ erlaubt werden. Daraus folgt, dass der Agent das Attribute mit dem Namen <AttributeName> lesen kann, aber kein anderes. Im Beispiel in Abbildung 3.7 benötigt der TemperaturAgent Zugriff auf den Topic „<csarName>/DataProcessor/Attributes/IP“. Dadurch kann vom TemperaturAgent nur das IP Attribute gelesen werden, nicht aber der PrivateKey oder die Properties der Datenbank.
Über die erweiterte Topicstruktur kann für jeden Agenten der Zugriff auf jedes einzelne Property oder Attribute jedes anderen Agenten kontrolliert werden. Die ACL Regeln müssen für jeden CSAR erstellt und vor dem Deployment im Broker konfiguriert werden.
Weitere Arbeiten könnten untersuchen, ob das ACL Regelset durch statische Analyse direkt aus dem Service Template generiert werden kann. Falls das nicht möglich ist, könnte erarbeitet werden, welche Einschränkungen der Spezifikation notwendig sind um, dass zu ermöglichen. Problematisch für eine statische Analyse sind zum Beispiel Attributes, welche Funktionen enthalten. Die Funktionen werden bei Lesezugriffen auf die Attributes ausgewertet und können wiederum andere Properties und Attributes lesen. Da die in Attributes enthaltenen Funktionen durch Operationen dynamisch während des Deployments verändert oder komplett überschreiben werden können, ist eine statische Auswertung schwierig. Weiterhin muss der Quellcode der Operationen nicht bekannt sein, da Implementierungen auch nur kompilierter Form vorliegen können.

Visibility von Attributes und Properties

Bei Properties wie dem „PrivateKey“, die ausschließliche lokal auf einem Agenten genutzt werden, ist es vorteilhaft, die Zugriffskontrolle direkt im Service Template zu modellieren. Dadurch ist der Schutz des Property oder Attribute nicht von der Konfiguration des Brokers abhängig. Deshalb wird in dieser Arbeit ein Visibilitykonzept für Properties und Attributes eingeführt. Die Visibility eines Properties oder Attributes kontrolliert, ob der Wert von einem anderen Agenten gelesen oder ob der Wert nur lokal verwendet werden kann. Dafür werden die zwei Access Modifier *private* und *public* eingeführt. Ein privates Property oder Attribute kann nicht mit einem anderen Agenten geteilt

werden. Alle Anfragen zu diesem Property oder Attribute werden von dem jeweiligen Agenten nicht oder mit einem Fehler beantwortet. Für das Beispiel in Abbildung 3.7 wurde die Vino4TOSCA [BBK+12] Notation um Visibilities erweitert. Ein Minuszeichen kennzeichnet private Properties oder Attributes und ein Pluszeichen wird für Öffentliche verwendet. Properties oder Attributes sind standardmäßig öffentlich. Deshalb kann das Pluszeichen weggelassen werden. Zum Beispiel private Schlüssel oder Anmeldedaten zu lokalen Softwarekomponenten und SaaS Angeboten sollten als privat gekennzeichnet sein. Wie bei der Zuordnung von Node Templates zu Agenten das Ziel, die Kompatibilität zu anderen TOSCA Modellierungs- und Validierungswerkzeugen zu erhalten. Deshalb wird die Visibility nicht über eine Erweiterung der TOSCA Syntax abgebildet, sondern über das Speichern der Access Modifier in den Metadaten der jeweiligen Attributes und Properties.

3.4 Agentenbasierte Skalierung

Cloud Computing erlaubt es, die gebuchten Ressourcen On-Demand den aktuellen Anforderungen anzupassen [MG11]. Es können dynamisch mehr Instanzen oder andere Ressourcen gebucht werden, wenn zum Beispiel eine erhöhte Nachfrage nach einer Anwendung entsteht. Dafür bieten Cloud Plattformen Funktionen, um automatisiert die gebuchten Ressourcen anzupassen und zum Beispiel mehr VMs zu starten, wenn mehr Anfragen gestellt werden. Weiterhin können nicht benötigte VMs automatisiert gestoppt werden, um Kosten zu sparen [MLH10].
Es gibt verschiedene Ansätze, um die automatische Skalierung der gebuchten Ressourcen zu implementieren, wie zum Beispiel regelbasierte Ansätze, queuebasierte Ansätze oder auch Reinforcement-Learning-basiert Ansätze [LML14]. Die Ansätze haben je nach gewünschtem Ziel, den zur Verfügung stehenden Metriken, der Konfiguration und den Anwendungskomponenten, die skaliert werden, unterschiedliche Vor- und Nachteile [LML14]. Das konkret verwende Skalierungsverfahren ist nicht Teil dieser Arbeit und muss von der Organisation, welche die Agenten betreibt, anhand von zum Beispiel Service-Level-Agreements, Kostenüberlegungen oder der zum Einsatz kommenden Plattformen ausgewählt werden. Viele Cloud Plattformen bieten integrierte Lösungen an. Die Arbeiten [LML14] und [QCB19] bieten einen Überblick über verschiedene Skalierungsverfahren. Weiterhin können unterschiedliche Organisationen innerhalb eines Deployments unterschiedliche Strategien nutzen. Auch IoT Anwendungen haben eine wechselnde Zahl an beteiligten Geräten. Manchmal fallen Geräte aus oder neue werden kurzzeitig hinzugefügt.
Um das automatisierte horizontale Skalieren von Instanzen und Geräten zu unterstützen, muss es möglich sein, dynamisch neue Agenten einem bestehenden Deployment hinzuzufügen oder zu entfernen. Ein erneutes provisionieren der Gesamtanwendung sollte dazu nicht notwendig sein. Um das zu ermöglichen, wird im Folgenden ein Konzept, basierend auf zwei unterschiedlichen Rollen präsentiert.
Es gibt einen Primary Agenten und beliebig viele Replica Agenten für einen Subgraph. Das erste Deployment umfasst nur Primary Agenten. Es gibt für jeden Subgraph genau einen Agenten in der Primary Rolle. Alle Agenten in den vorherigen Abschnitten sind implizit in der Primary Rolle. Neue Instanzen oder Geräte, welche von den Skalierungverfahren oder manuell einem bestehenden Deployment hinzugefügt werden, verwenden die Agentensoftware in der Replica Rolle mit dem gleichen Agentennamen wie der Primary Agent. Nach dem Starten registrieren sich alle Replica Agenten beim Primary Agent über MQTT. Dafür wird über den Topic „<Deployment-ID>/Agent/<AgentName>/register“ eine Anfrage nach dem verwendeten CSARs des Primary Agent gesendet. Der Primary Agent verwaltet die Registrierungen der Replicas und übertragt seinen

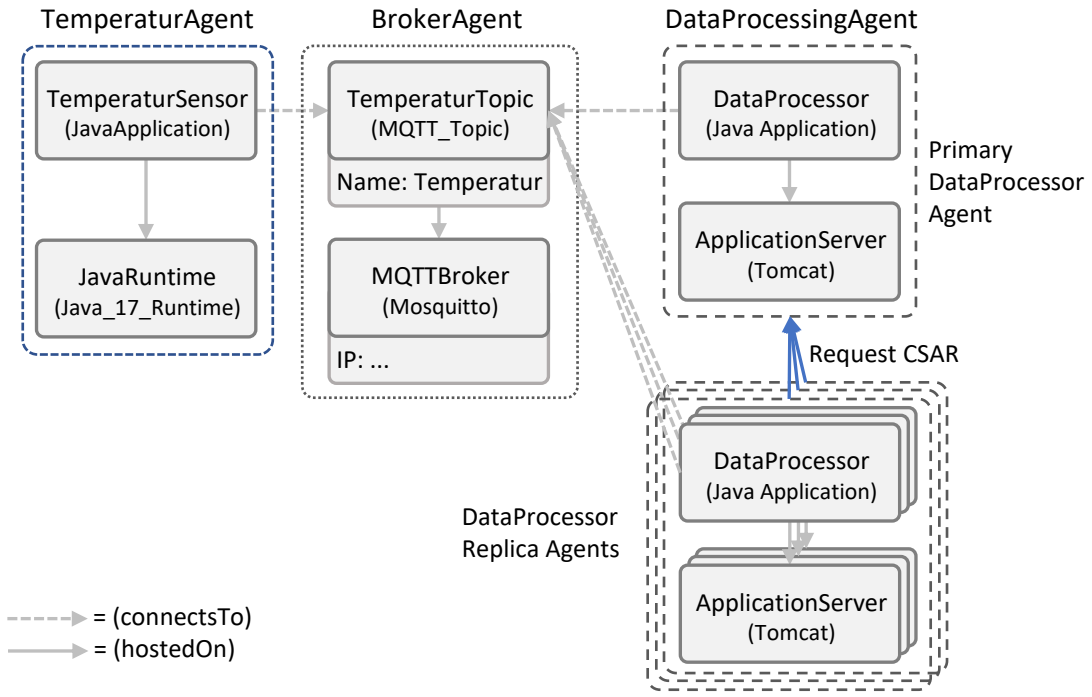

Abbildung 3.8: Beispiel für Replica Agenten, die basierend auf einem Primary Agenten gestartet wurden. Die Replicas rufen ihren CSAR vom Primary Agenten ab. [Notation basiert auf Vino4TOSCA BBK+12]

eigenen CSAR an die Replica Agenten. Dadurch erhält jeder Replica Agent den gleichen CSAR wie der jeweilige Primary Agent. Für diesen Vorgang können die in Abschnitt 3.3 beschriebenen Caching Mechanismen für Artifacts und CSARs verwendet werden. Die Registrierung enthält den jeweils individuellen Response Topic für jeden Replica Agent. Über die individuellen Response Topics kann der Primary Agent nicht nur seinen CSAR an jede Replica übertragen, sondern auch Management Operationen wie zum Beispiel *stop* oder *delete* weitergeben. Da Replicas jederzeit gestartet und gestoppt werden können, wird bei einem Löschen des Deployments eines CSARs nicht auf Rückmeldung aller Replicas gewartet, da diese schon gestoppt sein könnten. Das Konzept würde auch funktionieren, wenn sich alle Replica Agenten eines Deployments bei dem koordinierenden Agenten registrieren und über diesen ihre CSARs erhalten. Jedoch entsteht die Gesamtlast dann bei einem Agenten und nicht bei vielen Primary Agenten verteilt.

In Abbildung 3.8 wurden drei weitere Instanzen des DataProcessingAgent als Replicas gestartet, welche bei ihrem Primary DataProcessorAgent den CSAR abfragen. Neue Instanzen dieses Agenten haben den gleichen Namen, werden aber in der Replica Rolle gestartet. Es kann dabei so viele Primary- und Replica-Gruppen geben, wie es Agenten im Service Template gibt. Es könnten zum Beispiel auch Replicas vom TemperaturAgent gestartet werden, welche unabhängig von den Replicas des DataProcessingAgent agieren.

Replica Agenten bearbeiten dabei keine Property oder Attribute Anfragen, diese werden ausschließliche vom Primary Agenten beantwortet. Dadurch werden Anfragen nicht mehrfach bearbeitet. Während des Deployments können Replica Agenten aber Anfragen für Properties und Attributes an andere Agenten stellen.

Nicht alle Teile eines Service Templates können einfach skaliert werden. Würde zum Beispiel der

TemperaturTopic nur eine einzige Verbindung von einem DataProcessor akzeptieren, könnten sich die Replicas nicht verbinden. Node Templates die einen Server Session State für Verbindungen verwalten, können nicht einfach in Replicas verwendet werden [QCB19]. Da diese Node Templates nicht immer gestoppt werden können, ohne aktive Sessions zu unterbrechen [QCB19]. Ein Skalierungsverfahren müsste vor dem Beenden einer Replica warten, bis alle Sessions beendet wurden [QCB19]. Daher sollten alle Node Templates, die auf einer Replica eingesetzt werden stateless, sein. Weiterhin sollten Load Balancer oder Messaging Systeme eingesetzt werden, um dynamisch Verbindungen zwischen einzelnen Teilen der Anwendungstopologie zu ermöglichen. In dem in Abbildung 3.8 abgebildeten Beispiel sind alle Replica Agenten auf den TemperaturTopic subscribed. Sie nutzen dabei das Shared Subscription Feature von MQTT, mithilfe dessen Nachrichten jeweils nur an einen Subscriber gesendet werden, sodass jede Replica nur ein Teil aller Nachrichten bearbeiten muss [CBB+19].
Um das in Abschnitt 3.3.2 beschriebene Sicherheitskonzept auf Replicas anzuwenden, kann weitgehend das für den Primary Agent erstellte Topic Regelset übernommen werden. Zusätzlich zu diesen Regeln muss ein Replica Agent Nachrichten an den Topic „Agent/<EigenerAgentName>/register“ senden können, um sich beim Primary zu registrieren und seinen CSAR anzufordern.

3.5 GitOps mit TOSCA

Um die Vorteile eines automatisierten Deploymentsystems in größeren Teams oder in einem organisationsübergreifenden Umfeld zu nutzen, sollte es mit einem Prozess, der die Zusammenarbeit regelt, kombiniert werden. Einerseits müssen Änderungen am gemeinsamen Service Template einer Anwendung von allen beteiligten Teilnehmern nachvollzogen werden können. Andererseits müssen Änderungen einfach oder automatisch provisioniert werden können. Dadurch können Änderungen schneller umgesetzt werden und das Modell entspricht der provisionieren Anwendung [Wea]. Weiterhin sollte der Prozess Reviews und Rollbacks zu früheren Versionen unterstützen. Um diese Anforderungen zu erfüllen wird in dieser Arbeit ein auf GitOps-basierter Workflow für TOSCA präsentiert und mit dem in den vorherigen Abschnitten gezeigten agentenbasierten Deploymentkonzepten kombiniert. Um das Konzept für TOSCA anzupassen, müssen einige Veränderungen an dem Prozess wie er von Richardson [Ric17b] beschrieben wird vorgenommen werden. Dabei sollen die für GitOps gewünschten Prinzipien erhalten bleiben [Wea]. Deshalb wird der Prozess für TOSCA anhand dieser vier Prinzipien entwickelt.
Das erste Prinzip von GitOps beschreibt, dass das gesamte System in einem deklarativen Modell beschreiben wird. Der in dieser Arbeit beschriebene Ansatz um Anwendungen mit TOSCA zu provisionieren verwendet daher die deklarativen Features der TOSCA Spezifikation [OAS20b].
Das zweite GitOps Prinzip umfasst, dass der gewünschte Zustand des Systems in Git gespeichert wird [Wea]. Das modellierte Service Template beschreibt den gewünschten Systemzustand, der nach dem Deployment erreicht werden soll [OAS20b]. Die Node und Relationship Templates in der Topologie repräsentieren dabei die zu instanziierenden Komponenten und ihre Beziehungen. Um das zweite Prinzip zu erfüllen wird das Service Template in einem Git Repository gespeichert. Dadurch kann der gewünschte Zustand über Git versioniert werden und Rollbacks zu früheren Versionen sind jederzeit möglich. Außerdem können Änderungen über Review Mechanismen freigegeben werden [Ric17a]. Das ist für organisationsübergreifende Deployments wichtig, da Änderungen dadurch für

alle Teilnehmer sichtbar sind. Weiterhin ist es möglich Plattformen wie GitHub[4] oder GitLab[5] so zu konfigurieren, dass Änderungswünsche in Form von Pull Requests von allen oder von mindestens einer bestimmten Anzahl an Teilnehmern bestätigt werden müssen. So kann keine Organisation alleine unerwünschte Änderungen durchführen. Weiterhin lassen sich Commits signieren, sodass die Herkunft verifiziert werden kann [Wea].
Das dritte von Weaveworks [Wea] beschriebene Prinzip fordert, dass freigegebene Änderungen automatisch auf das Zielsystem angewendet werden können. Dadurch reflektiert der gewünschte Systemzustand, der in Git gespeichert ist, den wirklichen Zustand der provisionieren Anwendung. Um das Prinzip zu realisieren, wurde für diese Arbeit eine prototypische Synchronisierungsanwendung entwickelt, welche automatisiert geänderte Service Templates auf ein Deployment anwenden kann. Zur Demonstration dieser Anwendung wurde sie in einen GitHub Actions[6] Workflow integriert. Bei Änderungen wird automatisch oder manuell das Git Repository ausgecheckt, ein neuer CSAR generiert und dann automatisch auf die konfigurierte Umgebung via MQTT angewendet. Dabei müssen durch das agentenbasierte System keine Passwörter für Cloud-Provider, SSH-Schlüssel für IoT-Geräte oder Schnittstellen von interner Infrastruktur im Repository oder während des Deployments bekannt sein. Dadurch brauchen Entwickler keine Anmeldedaten oder Zugänge, um Änderungen am System vornehmen zu können [Wea]. Wenn darüber hinaus der Zugang zum MQTT Broker nicht in GitHub gespeichert werden soll, kann die Synchronisierungsanwendung auch auf eigener Infrastruktur betrieben werden.
Das letzte Prinzip beschreibt, dass Softwareagenten die Korrektheit des Deployments überwachen [Wea]. Dabei sollen die eingesetzten Agenten den Nutzer benachrichtigen, wenn die Realität nicht mit dem erwarteten Zustand übereinstimmt [Wea]. Um dieses Prinzip umzusetzen, wurde die Synchronisierungsanwendung so konzipiert, dass jeder Fehler während des Deployments zum Abbruch der GitHub Action führt. Dazu melden die einzelnen Agenten jedes Problem dem Koordinationsagenten und dieser leitet den Fehler an die Synchronisierungsanwendung weiter. Die korrekten Schritte, um auf ein Fehler im Deployment zu reagieren, sind von vielen Faktoren wie die Art der Anwendung, Service-Level-Agreements und der verwendeten Infrastruktur abhängig. Deshalb sollte jedes Entwicklerteam eine individuelle Lösung umsetzen, um auf Fehler zu reagieren. Über den GitHub Workflow ist es zum Beispiel möglich, E-Mail-Benachrichtigungen einzurichten. Es wäre auch möglich bei einem Fehler automatisch zu einer früheren Version zurückzukehren oder das Deployment zu wiederholen. Ein automatisiertes Wiederholen kann aber auf batteriebetriebenen IoT-Geräten wertvolle Energie verschwenden, wenn es sich um einen permanenten erneut auftretenden Fehler handelt. Des Weiteren müssen Implementierungen in TOSCA nicht idempotent sein und Node Templates repräsentieren nicht nur Container, sondern fast beliebige Softwarekomponenten, Ressourcen oder Geräte. Daher ist die korrekte Reaktion auf ein Fehler vom Anwendungsfall abhängig. Hier ist weitere Arbeit notwendig, um eine automatische Konvergenz zum korrekten Zustand zu ermöglichen und automatisiert auf Fehler oder eine Abweichung des gewünschten Zustands zu reagieren. Auch könnten in Zukunft weitere Konzepte erarbeitet werden um Observability während des gesamten Betriebs über TOSCA zu gewährleisten.

[4] https://github.com/ (zuletzt besucht 28.09.2022)
[5] https://gitlab.com (zuletzt besucht 28.09.2022)
[6] https://github.com/features/actions (zuletzt besucht 28.09.2022)

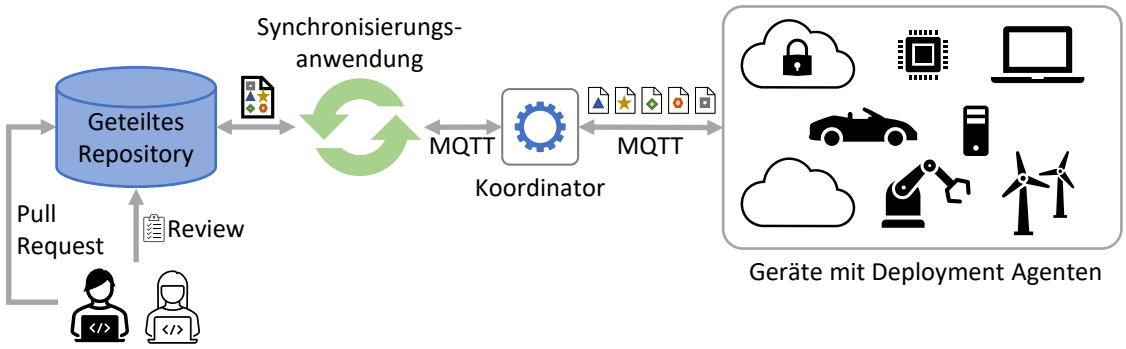

Abbildung 3.9: GitOps-basierter Workflow für TOSCA-basierte Deployments. Entwickler erstellen ein Service Template, das bei Änderungen von der Synchronisierungsanwendung an einen koordinierenden Agenten übergeben wird. Der Koordinator teilt das Service Template in mehrere Subgraphen auf und führt das Deployment mithilfe der verteilten Agenten auf jedem Gerät aus.

Abbildung 3.9 zeigt eine vereinfachte Pipeline, welche den GitOps-basierten Prozess für TOSCA abbildet. Hier arbeiten zwei Entwickler aus unterschiedlichen Organisationen an einem Service Template und den benötigten Artifacts, die in einem gemeinsamen Repository abgelegt sind. Das Repository wird auf einer Plattform gehostet, das neben Git zusätzlich noch soziale Komponenten wie Teamverwaltung, Pull Requests, Reviews, Kommentare und Issuestracking bieten sollte [Ric17b]. Die Anwendung läuft auf Ressourcen aus beiden Organisationen, wobei die Organisationen aufgrund von Compliance-Richtlinien keine Anmeldedaten, Schlüssel oder Zugänge zu internen Schnittstellen austauschen können. Um Änderungen am Service Template oder anderen Teilen des CSARs vorzunehmen, stellt einer der Entwickler ein Pull Request mit seinen Änderungen. Der Pull Request muss dann je nach Konfiguration von allen oder einigen Entwicklern einem Review unterzogen werden. Verläuft das Review erfolgreich und alle nötigen Entwickler stimmen der Freigabe des Pull Requests zu, kann der Request gemerged werden. In diesem Beispiel muss immer der jeweils andere Entwickler den Pull Request zustimmen. Änderungen sind für alle Entwickler nachvollziehbar und können über Zusatzinformationen wie Kommentare dokumentiert werden. Wird die Änderung gemerged wird die Synchronisierungsanwendung über ein Eventsystem in der Git Plattform ausgeführt. Diese erstellt einen CSARs aus dem Service Template und den Artifacts. Anschließend übertragt die Anwendung den CSAR per MQTT an einen konfigurierbaren koordinierenden Agenten. Dieser Agent kann Teil der zu provisionierenden Anwendung sein, ist aber in dem Beispiel in Abbildung 3.9 zur besseren Übersichtlichkeit separat dargestellt. Der Koordinator generiert dann aus der kompletten Topology individuelle Subgraphen für jeden Agenten und verpackt diese wiederum in CSARs. Diese CSARs werden an die einzelnen Agenten auf zum Beispiel IoT- und Edge-Geräte oder in Clouds übertragen. Nachdem jeder CSAR übertragen ist, führen die Agenten das Deployment aus. Treten während des Deployments Fehler auf, so informiert der betroffene Agent den Koordinator und dieser die Synchronisierungsanwendung. Die Synchronisierungsanwendung bricht das Deployment ab und löst konfigurierbare Schritte aus, um auf den Fehler zu reagieren.
Durch das Speichern des Service Templates und seiner Abhängigkeiten in einem Git-Repository entsteht eine „single source of truth“ [Wea], die Zusammenarbeit vereinfacht, das Recovery im Fehlerfall erleichtert und einen automatischen Audit Trail hinterlässt [Wea].

3.5.1 Ausblick

Der Workflow in Abbildung 3.9 ist so entworfen, dass er mit zukünftigen Arbeiten um mehr Funktionalität erweitert werden kann. Zum Beispiel könnte der von Krieger et al. [KBKL18] vorgestellte Ansatz verwendet werden, um Änderungen am Service Template während des Review-Prozesses auf Compliance mit einem vorher definierten Regelwerk zu überprüfen. Darüber können zum Beispiel Regeln modelliert werden, um Standards wie ISO 27018 [Int19] oder die Datenschutz-Grundverordnung [16] einzuhalten [KBKL18]. Außerdem könnte der Ansatz von Saatkamp et al. [SBK18] in den Review-Prozess integriert werden, um basierend auf formalisierten Pattern Probleme im Service Template zu erkennen. Auch eine syntaktische Prüfung des Service Templates könnte von Validierungs- oder Modellierungswerkzeugen vorgenommen werden.
Weiterhin könnten die erhobenen Requirements mit in dem Repository dokumentiert werden, was es erlaubt, bei Änderungen auf das jeweilige Requirement zu verweisen und automatisch eine Änderungshistorie der Requirements zu erhalten. Dadurch kann der Grund der Änderung direkt auf ein Requirement nachvollzogen werden [Ric17a].
Abhängig von den verwendeten Softwarekomponenten kann es notwendig sein, bestimmte Implementation- oder Deployment-Artifacts zu kompilieren oder zu übersetzen. Werden zum Beispiel in Java entwickelte Implementation Artifacts als Quellcode direkt im Repository gespeichert, lassen sich Änderungen einfacher nachvollziehen. Jedoch müssen diese Artifacts vor der Verwendung von einem Java Compiler in Bytecode übersetzt werden. Dazu kann der Workflow zum Beispiel durch eine CI Pipeline ergänzt werden, welche alle nötigen Übersetzungen und Tests vornimmt. Die finalen Artifacts können dann in den CSAR integriert oder zum Beispiel in einem Maven[7] Repository gespeichert werden. Die Artifacts werden dann im Service Template referenziert.

[7] https://maven.apache.org/ (zuletzt besucht 28.09.2022)

4 Implementierung des Prototyps

Dieses Kapitel beschäftigt sich mit der Implementierung des Prototyps für die Agentensoftware. Der Prototyp wird genutzt, um mithilfe des in Abschnitt 4.2 beschriebene Anwendungsszenario die praktische Umsetzbarkeit der in dieser Arbeit beschreiben Konzepte zu validieren. Er unterstützt die in Abschnitt 3.1 beschriebene Erweiterung der TOSCA 2.0 Spezifikation [OAS20b], um die für das Deployment verantwortlichen Agenten in einem Service Template zu repräsentieren. Die Agentensoftware wird auf jedem Gerät, jeder virtuellen Maschine und jeder Plattform, auf die Node oder Relationship Templates provisioniert werden soll, vorinstalliert. Dazu kann der Agent in ein Image oder in ein Installationsskript integriert werden. Zur korrekten Funktion muss zusätzlich der zugewiesene Agentenname, der gewünschte MQTT Broker und die DeploymentID per Umgebungsvariablen konfiguriert werden.
Während eines Deployments übernimmt die Instanz des Prototypes, an die ein CSAR gesendet wird, die Rolle des Koordinators. Der Koordinator wendet das in Abschnitt 3.2.7 ausgewählte TSSN Verfahren an, um ein Service Templates in individuelle Subgraphen aufzuteilen. Nach der Aufteilung wird jeder Subgraph um die verwendeten Typen und Artefakte ergänzt und in ein CSAR verpackt. Alle am Deployment beteiligten Agenten erhalten ihren individuellen CSAR vom Koordinator per MQTT. Sie führen dann das Deployment teilautonom aus und nutzen, wenn notwendig, die in Abschnitt 3.3 beschriebene Kommunikationsarchitektur, um Attributes und Properties auszutauschen. Dabei ist es die Aufgabe des MQTT Brokers und seiner Konfiguration sicherzustellen, dass die Agenten keine unberechtigten Zugriffe auf Werte ausführen können. Aufgabe der Agenten ist neben dem Deployment sicherzustellen, dass die Visibilities der lokalen Properties und Attributes eingehalten werden.
Der Prototyp ist in Java implementiert, um auf allen von der Java Virtual Maschine[1] unterstützten Plattformen genutzt werden zu können. Der Prototyp Agent bietet ein MQTT Interface zur Steuerung der Agenten und zur Kommunikation der Agenten untereinander. Weiterhin bietet er ein HTTP-basiertes REST Interface zur Kontrolle und Nutzerinteraktion. Der Prototyp implementiert nicht die gesamte TOSCA 2.0 Spezifikation [OAS20b], sondern nur einen minimalen Orchestrator. So wurde zum Beispiel die Importfunktionalität und Vererbung nicht implementiert. Es wurde die Funktionalität ausgewählt, welche für die Demonstration des vorgesehenen Anwendungsfalls als Agent in einem organisationsübergreifenden heterogenen Deployment benötigt wird. Manche Funktionen sind auch in dem aktuellen Entwurf „Draft 03“ der TOSCA 2.0 Spezifikation nicht enthalten und konnten daher nicht implementiert werden [OAS20b]. Weiterhin werden für manche TOSCA Entitäten, unter anderem nicht alle Keynames unterstützt. Als Implementation Artifacts für Operationen werden vom Prototyp Bash-Skripte unterstützt. Um weitere Implementation Artifacts zu unterstützen, kann ein Java Interface im Prototyp für den jeweiligen Artifact Type implementiert werden. Alle verwendeten Werte wie Properties, Attributes und Inputs werden als

[1] Es gibt mehrere Java Virtual Maschine Implementierungen, zum Beispiel von Oracle https://www.oracle.com/java/technologies/downloads/ oder Eclipse https://www.eclipse.org/openj9/ (zuletzt besucht 28.09.2022)

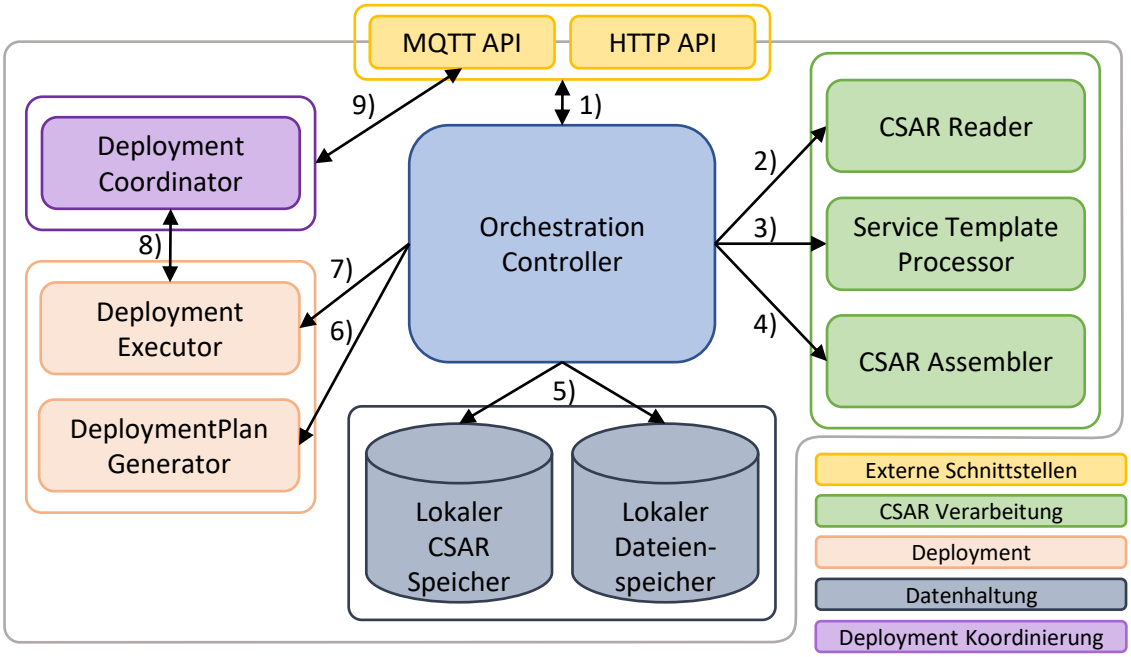

Abbildung 4.1: Architekturübersicht mit den Interaktionen zwischen den Komponenten während eines Deployments.

„Strings“ gehandhabt, da Bash-Skripte nur Strings zur Aus- und Eingabe von Daten verwenden können. Inputs werden per Umgebungsvariable an die Bash-Skripte übergeben und Outputs werden als Schlüsselwertpaare aus der Ausgabe des Skripts geparst.

4.1 Architektur

Um die ausgewählten Ansätze mit dem Agentenprototyp zu demonstrieren, müssen mehrere Funktionalitäten implementiert werden. Der Prototyp muss CSARs verarbeiten und speichern, Deployments ausführen, Deployments koordinieren, Properties und Attributes austauschen und per HTTP-Statusinformationen bereitstellen. Im Folgenden werden anhand der Schritte, welche für ein verteiltes Deployment notwendig sind und der in Abbildung 4.1 abgebildeten Architekturübersicht alle Architekturkomponenten erklärt. Beginnt ein neues Deployment eines CSARs wird dieser per MQTT zu einem der teilnehmenden Agenten übertragen. Dieser Agent erhält den CSAR über die MQTT API und übernimmt die Rolle des Koordinators in diesem Deployment. Die *MQTT API* bietet den anderen Komponenten eine Abstraktionsschicht über das verwendete Kommunikationsprotokoll und erlaubt das Senden und Empfangen von CSARs, Attributes, Properties und Befehlen über MQTT. Der empfangene CSAR wird in Schritt 1) an den *Orchestration Controller* übergeben. Die Aufgaben des Orchestration Controllers umfassen die Koordination der Verarbeitung und des Deployment des CSARs. Der Orchestration Controller verwendet, um das zu erreichen die anderen Komponenten. In Schritt 2) wird der CSAR von dem *CSAR Reader* entpackt. Der CSAR Reader verarbeitet auch die im CSAR enthaltenen Metainformationen und stellt diese dem Orchestration Controller zusammen mit dem im CSAR enthaltenen Service Template und den anderen Artifacts zur Verfügung. Der Orchestration Controller übergibt in Schritt 3) dem *Service Template Processor* das gelesene Service

Template in Textform. Der Service Template Processor parst das YAML-basierte Service Template und erstellt draus Java Objekte, die das Service Template repräsentieren. Das Service Template wird nach dem Parsen mit dem in Abschnitt 3.2.5 beschrieben Verfahren in verschiedene Subgraphen aufgeteilt. Diese Fragmente eines Service Template sind an sich keine validen Service Templates, da alle nicht lokalen Node Templates entfernt wurden und somit nicht alle Relationship Templates ein valides Ziel haben. Der Prototyp ist aber in der Lage, diese Fragmente im Kontext des gesamten Service Templates auszuführen. Um die Ausführung der Subgraphen zu ermöglichen, müssen die einzelnen Subgraphen zu einem neuen Service Template vervollständigt werden. Dazu werden alle im Subgraphen benötigten Node-, Relationship-, Interface-, Capability- und Artifact-Types zu dem neuen Service Template hinzugefügt. Ist die Erstellung der neuen Service Templates für jeden Agenten abgeschlossen, übergibt der Orchestration Controller alle Service Templates dem *CSAR Assembler*. Der CSAR Assembler erstellt in Schritt 4) für alle neuen Service Templates ein CSAR. In diesen CSAR wird jeweils ein Service Template, alle referenzierten Artifacts und CSAR Metadaten verpackt. Die neuen CSARs werden an alle anderen Agenten per MQTT übermittelt. Jeder Agent, der ein CSAR enthält, durchläuft die gleichen bisher beschriebenen Schritte und meldet dann das erfolgreiche Verarbeiten des CSARs an den Koordinator zurück. Schritt 5) umfasst das Speichern des CSARs und der aus dem Service Template erzeugten Objekte im lokalen CSAR Speicher. Alle Artifacts, wie zum Beispiel Skripte oder Binaries werden im lokalen Datenspeicher im Dateisystem abgelegt. Wenn alle Agenten per MQTT den Erhalt ihres individuellen CSARs bestätigt haben, kann das Deployment beginnen.

Um das Deployment zu beginnen, sendet der Koordinator allen Agenten ein Befehl per MQTT. Daraufhin generiert jeder Agent mit dem *Deployment Plan Generator* in Schritt 6) aus seinem Subgraph einen individuellen Deploymentplan. Der Deployment Plan Generator verwendet das von Breitenbücher et al. [BBK+14] vorgestellte Verfahren, um die deklarativ beschriebene Topologie in ein imperativ ausführbaren Plan umzuwandeln. Der Plan enthält für jedes Node und Relationship Template alle Operationen, die zum erfolgreichen Instanziieren und Starten notwendig sind. Der Orchestration Controller übergibt den generierten Deploymentplan in Schritt 7) dem *Deployment Executor* der diesen ausführt. Dafür führt der Deployment Executor jede im Plan enthaltene Operation aus, indem er das jeweilige Implementation Artifact ausführt. Werden nur im lokalen Subgraph enthaltene Attributes und Properties benötigt, kann ein Agent sein Deployment ohne Interaktion mit anderen Agenten autonom und autark abschließen. Benötigt eine Operation ein Property oder ein Attribute aus einem anderen Subgraph muss dieses über MQTT erfragt werden. Dazu werden in Schritt 8) wie in Abschnitt 3.3 beschreiben Anfragen über den *Deployment Coordinator* abgesendet. Der Deployment Coordinator nutzt in Schritt 9) die MQTT API, um Properties und Attributes anzufragen oder Befehle per MQTT zu empfangen. Die Ergebnisse der Anfragen werden jeweils zurück an den Deployment Executor übergeben. Das Deployment auf den verschiedenen Agenten wird implizit durch die angefragten Properties und Attributes koordiniert. Die HTTP API wird während des Deployments nicht benötigt, sie erlaubt einem Nutzer aber jederzeit das Abfragen von Statusinformationen. Die API bietet Informationen zu den gespeicherten CSARs, erlaubt es neue CSARs hochzuladen oder existierende zu löschen. Auch der Fortschritt eines Deployments und der Status der einzelnen Node und Relationship Templates kann überprüft werden.

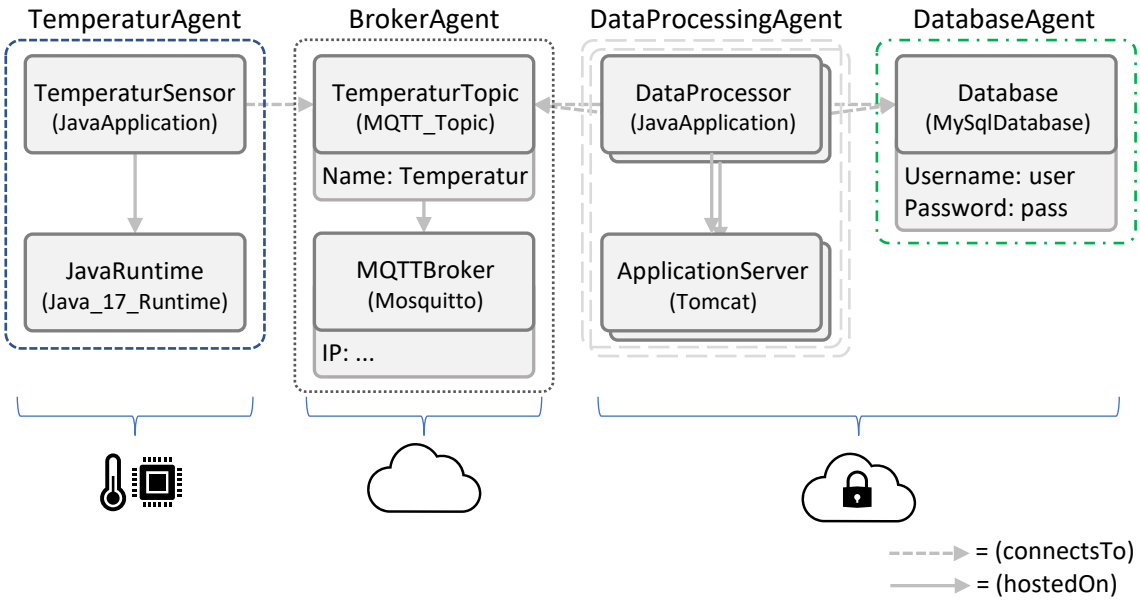

Abbildung 4.2: Das Anwendungsszenario beschreibt ein kombiniertes IoT- und Cloud-Szenario, das Temperaturwerte erfasst, verarbeitet und in einer Datenbank speichert. Dazu werden Agenten auf einem IoT-Gerät sowie in einer Public- und einer Private-Cloud verwendet. [Notation basiert auf Vino4TOSCA BBK+12]

4.2 Anwendungsszenario

Das Anwendungsszenario beschreibt eine Multicloud- und IoT-Anwendung, welche mit den in dieser Arbeit beschriebenen Konzepte provisioniert wird. Dazu wird der in diesem Kapitel beschriebene Prototyp genutzt. Das Anwendungsszenario ist zur besseren Verständlichkeit an die in dieser Arbeit verwendeten Beispiele angelehnt. Die Abbildung 4.2 enthält eine Übersicht über die verwendeten Node und Relationship Templates, die eingesetzten Agenten und ihre Verteilung auf die eingesetzte Infrastruktur. Der TemperaturAgent befindet sich auf einem Raspberry Pi[2] 3 mit einem angeschlossenen Temperatursensor. Ein Raspberry Pi ist ein kleiner Einplatinencomputer der Raspberry Pi Foundation [Ras]. Es kommt eine Java Anwendung zum Einsatz, die den Sensor in regelmäßigen Abständen ausliest und die erfassten Daten zu dem TemperaturTopic sendet. Diese wird durch das Node Template TemperaturSensor im Service Template repräsentiert. Der TemperaturTopic ist ein MQTT Topic und wird zusammen mit dem MQTT Broker Mosquitto in einer Public Cloud provisioniert. Der DataProcessor liest die Temperaturdaten von dem Topic und verarbeitet diese weiter. Um gegebenenfalls eine größere Menge an Sensordaten auszuwerten, können je nach Bedarf mehrere Instanzen des DataProcessing Agent provisioniert werden. Die erste Instanz ist dabei der Primary Agent und alle weiteren Instanzen sind Replica Agenten. Primary und Replica Agenten werden in einer Private Cloud betrieben und nutzen ein von der Cloud bereitgestelltes Skalierungsverfahren, das je nach Bedarf neue Replica Agenten startet oder alte stoppt. Nach der Verarbeitung der Sensordaten durch einen DataProcessor werden die Daten in

[2] https://www.raspberrypi.org/ (zuletzt besucht 28.09.2022)

einer MySQL[3] Database gespeichert. Die Database wird auch in der Private Cloud ausgeführt.
Der TemperaturAgent wird direkt auf dem Raspberry Pi installiert. Die anderen Agenten werden in virtuellen Maschinen in den jeweiligen Clouds ausgeführt. Die Instanzen in der Private Cloud befinden sich hinter einer Firewall und können nicht von außerhalb erreicht werden. In dem Netzwerk des Raspberry Pis kommt NAT zum Einsatz, daher kann auch dieser keine eingehenden Verbindungen empfangen. Die unterschiedlichen Infrastrukturteile könnten in diesem Szenario von unterschiedlichen Organisationen betrieben werden. Dazu müssten keine Zugänge oder Schnittstellen zwischen den Organisationen geteilt werden. Das Service Template und alle notwendigen Abhängigkeiten werden über den in Abschnitt 3.5 präsentierten Prozess entwickelt und in einem Git-Repository gespeichert. Wird eine Änderung gemerged, beginnt die Synchronisierungsanwendung mit der Erstellung des CSARs und überträgt diesen zum BrokerAgent. Der BrokerAgent übernimmt die Rolle des Koordinators und generiert aus der Gesamttopologie individuelle Subgraphen für die einzelnen Agenten. Die Subgraphen werden in CSARs verpackt und zu den einzelnen Agenten übertragen.
Die Node Templates in der unteren Hälfte in Abbildung 4.2 können vom jeweiligen Agent unabhängig von den anderen Agenten parallel provisioniert werden. Für das Deployment der Node und Relationship Templates der oberen Hälfte müssen die Abhängigkeiten der Templates untereinander berücksichtigt werden. Zum Beispiel kann die Database ohne weitere Abhängigkeiten provisioniert werden. Die Relationship Templates, welche vom DataProcessor und vom TemperaturSensor ausgehen, können jedoch erst provisioniert werden, wenn ihre jeweiligen Ursprungs- und Ziel-Node-Templates provisioniert sind. Alle Operationen, ihre Inputs und manche Properties sind in der Abbildung aus Übersichtlichkeitsgründen nicht dargestellt. Aber jedes Relationship Template hat mindestens eine Operation, welche mindestens ein Attribute oder Property des jeweiligen Ziel Node Templates benötigt. Zum Beispiel benötigt die configure Operation des vom TemperaturSensor ausgehenden Relationship Templates das Property Name des TemperaturTopics. Durch die Verwendung des TSSN Ansatz entsteht durch das Übertragen der Werte eine implizite Deploymentkoordination zwischen den Agenten. Die Agenten nutzen MQTT und einen weiteren Mosquitto Broker Instanz, um die Attributes und Properties auszutauschen.
In diesem Broker wurde eine Access Control List konfiguriert, sodass ein Agent nur die für ihn notwendigen Werte abrufen kann. Zum Beispiel erlaubt es die Regel „topic write csarName/TemperaturTopic/Properties/Name“ dem TemperaturAgent Anfragen für das Property Name des TemperaturTopic zu stellen. Wenn keine Regel für einen Zugriff erstellt wurde, kann ein Agent keine externen Werte lesen.
Für dieses Szenario wird davon ausgegangen, dass der Raspberry Pi und die virtuellen Maschinen in den Clouds schon von ihren jeweiligen Organisationen gestartet wurden und auf allen der Agentenprototyp ausgeführt wird.

[3] https://www.mysql.com/de/ (zuletzt besucht 28.09.2022)

5 Zusammenfassung und Ausblick

In dieser Masterarbeit wurden mehrere Konzepte präsentiert, um eine verteilte Anwendung organisationsübergreifend zu entwickeln, automatisiert in heterogene Umgebungen zu provisionieren und Änderungen kontrolliert vorzunehmen. Das automatisierte Deployment kann dabei verschiedene Softwarekomponenten auf IoT- und Edge-Geräte sowie unterschiedliche Cloud oder On-Premises Plattformen von verschiedenen Organisationen umfassen. Um den am Deployment beteiligten Organisationen die Souveränität über Ihre Infrastruktur zu ermöglichen, präsentierte diese Arbeit ein dezentralisiertes Deploymentkonzept, das auf teilautonome Agenten für das Softwaredeployment setzt. Die Agenten erlauben ein koordiniertes Deployment, ohne dass Organisationen ihre internen Schnittstellen für einen zentralen Orchestrator öffnen oder Anmeldedaten für ihre Infrastruktur oder Cloud Plattformen austauschen müssen. Um ein verteiltes auf agentenbasiertes Deployment zu ermöglichen, wurden in dieser Arbeit mehrere Konzepte erarbeitet und verglichen, um Agenten in TOSCA Service Templates zu repräsentieren. Der gewählte Ansatz, welcher die Agenten im Service Template über das TOSCA Group Feature repräsentiert, erlaubt die Kompatibilität zur TOSCA Spezifikation zu erhalten.
Weiterhin wurden mehrere Konzepte präsentiert und gegeneinander abgewägt, um ein TOSCA Modell in Subgraphen für unterschiedliche Agenten aufzuteilen, sodass einzelne Agenten nur Zugriff auf den für sie relevanten Teil des Modells erhalten und die Subgraphen möglichst autonom provisioniert werden können. Das reduziert die benötigte Übertragungskapazität und Energie, um das Modell an die einzelnen Agenten zu übertragen. Auch sind sensible Informationen in anderen Teilen des TOSCA Modells, im Fall einer Kompromittierung eines Agenten, geschützt. Die fünf Ansätze, um die Subgraphen aus dem Gesamtmodell zu generieren, unterschieden sich dabei hauptsächlich in der Mengen an Informationen, die jedem Agenten initial zur Verfügung steht, wie viele Informationen während des Deployments von anderen Agenten abgerufen werden und welche zusätzlichen Regeln bei der Erstellung des Gesamtmodells beachtet werden müssen.
Der gewählte TSSN Ansatz erlaubt es, Subgraphen verteilt auf unterschiedlichen Agenten im Kontext der Gesamttopologie auszuführen, erzeugt kleine Subgraphen und erlaubt die Übertragung von Attributes und Properties für die Koordination des Deploymentprozesses zu nutzen. Darüber hinaus benötigt der Ansatz nur Netzwerkkommunikation, wenn eine Datenabhängigkeit zwischen zwei Subgraphen vorliegt und es müssen wenig zusätzliche Modellierungsregeln eingehalten werden.
Um die Kommunikation zwischen den Agenten zu ermöglichen, wurde ein Konzept, basierend auf einem MQTT Broker präsentiert. Das Konzept ermöglicht Kommunikation zwischen den Agenten auch in Netzwerken mit NAT oder Sicherheitsbeschränkungen, die keine eingehenden Verbindungen erlaubten. Weiterhin erlaubt ein auf MQTT-Topic-basiertes Sicherheitskonzept eine präzise Zugangskontrolle zu einzelnen Properties und Attributes für jeden Agenten. Das Kommunikationskonzept wurde weiterhin um ein in dieser Arbeit entwickeltes Visibilitykonzept für Properties und Attributes erweitert, durch das einzelne Werte direkt im Modell für externe Zugriffe gesperrt werden können.
Cloud Computing ermöglicht das dynamische Skalieren von Ressourcen, um sich unvorhersehbaren Workloads anzupassen. Auch in IoT-Umgebungen können neue Geräte hinzugefügt oder entfernt

werden. Aus diesem Grund wurde in dieser Arbeit ein Konzept präsentiert, um einem existierenden Deployment dynamisch neue Instanzen hinzufügen zu können oder alte Instanzen zu stoppen.
Um die Zusammenarbeit zwischen verschiedenen Organisationen nicht nur während des Deployments, sondern auch während der Entwicklung und Anpassung einer Anwendung zu vereinfachen, wurden in dieser Arbeit ein GitOps-basierter Workflow präsentiert. Durch diesen Workflow können unterschiedliche Organisationen einfach an einem Modell zusammenarbeiten, Änderungen einem Review Prozess unterziehen und automatisiert auf ein Deployment anwenden. Weiterhin entsteht durch Git automatisch ein Audit Trail über alle vorgenommenen Änderungen und bei Fehlern kann zu einer älteren Version zurückgekehrt werden [Wea].
Um die Machbarkeit der Konzepte in dieser Arbeit zu validieren, wurde die Agentensoftware konzipiert und mit einem Prototyp in Java implementiert. Am Ende dieser Arbeit wurde der Prototyp mit einem Anwendungsszenario getestet.

Ausblick

Weiterführende Forschung könnte die in dieser Arbeit erstellten Konzepte an unterschiedlichen Stellen erweitern. Zum Beispiel könnte ein Modellierungswerkzeug für die Erstellung von Service Templates entwickelt werden, welches die in dieser Arbeit erweiterte Semantik des Group Features unterstützt. Das Werkzeug könnte einen Entwickler bei der Zuordnung von Node Templates zu Agenten visuell unterstützen. Das Modellierungswerkzeug könnte weiterhin das Erstellen von Access Control Lists vereinfachen, indem es zum Beispiel für jedes Attribute oder Property eine Auswahl alle Agenten bietet und den Benutzer auswählen lässt, welcher Agent auf den Wert Zugriff hat. Auch könnte in einer Arbeit untersucht werden ob, ACL Regeln direkt über statische Analyse im Modellierungswerkzeug bestimmt werden können. Dieses Werkzeug könnten dann anhand der manuell oder automatisch bestimmten Regeln, die passende Konfiguration für unterschiedliche Broker Implementierungen generieren. Darüber hinaus könnte das Werkzeug die visuelle Notation für das Visibilitykonzept von Attributes und Properties unterstützen. Weiterhin wäre es möglich, den von Harzenetter et al. [HBF+18] vorgestellten Ansatz, um Deploymentmodelle über technologie- und herstellerneutrale Pattern zu definieren, in dieses Modellierungswerkzeug zu integrieren.

Die Software, welche mit den in dieser Arbeit gezeigten Konzepten provisioniert wird, umfasst oft viele Komponenten auf unterschiedlichen Geräten oder Plattformen. Auch nach einem erfolgreichen Deployment können Software- oder Hardwarefehler auftreten, wodurch Teile oder die gesamte Anwendung unbenutzbar wird. Darüber hinaus können Engpässe bei der verfügbaren Leistung das Benutzererlebnis verschlechtern. Aktuell gibt es keine in der TOSCA 2.0 Spezifikation definierte Lösung, um einheitlich Logs und Metriken von Node und Relationship Templates zu erheben, diese zentral zu sammeln, auszuwerten und zu visualisieren. In zukünftigen Arbeiten könnte eine einheitliche Schnittstelle für TOSCA definiert werden, um Logs und Metriken zu erheben. Für diesen Zweck könnten wichtige Indikatoren für den Zustand eines Node oder Relationship Templates anhand der gesammelten Metriken erarbeitet werden. Dadurch wäre es einfacher, den Gesamtzustand der Anwendung zu erfassen und bei Fehlern Gegenmaßnahmen zu ergreifen.

Literaturverzeichnis

[16] „Verordnung (EU) 2016/679 des Europäischen Parlaments und des Rates vom 27. April 2016 zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, zum freien Datenverkehr und zur Aufhebung der Richtlinie 95/46/EG (Datenschutz-Grundverordnung)“. In: *Amtsblatt der Europäischen Union* L 119 (4. Mai 2016), S. 1–88. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj (besucht am 28. 09. 2022) (zitiert auf S. 64).

[ABL15] J.-P. Arcangeli, R. Boujbel, S. Leriche. „Automatic deployment of distributed software systems: Definitions and state of the art“. In: *Journal of Systems and Software* 103 (Mai 2015), S. 198–218. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.01.040. URL: https://www.sciencedirect.com/science/article/pii/S0164121215000308 (besucht am 28. 09. 2022) (zitiert auf S. 34).

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica et al. „A view of cloud computing“. In: *Communications of the ACM* 53.4 (2010), S. 50–58. ACM New York, NY, USA (zitiert auf S. 19, 20).

[AGB21] L. Albertin, L. Ganne, S. Benoist. *TOSCA support in Yorc*. 28. Juni 2021. URL: https://yorc.readthedocs.io/en/v4.3.0/tosca.html (besucht am 28. 09. 2022). Github Quelltext https://github.com/ystia/yorc/blob/c4597cc65549b33e5b0854954d2552ad97894d53/doc/tosca.rst (zitiert auf S. 35).

[AIM10] L. Atzori, A. Iera, G. Morabito. „The internet of things: A survey“. In: *Computer networks* 54.15 (2010). Elsevier, S. 2787–2805 (zitiert auf S. 15).

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. „Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA“. In: *On the Move to Meaningful Internet Systems: OTM 2012*. Hrsg. von R. Meersman, H. Panetto, T. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, I. F. Cruz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 416–424. ISBN: 978-3-642-33606-5. DOI: 10.1007/978-3-642-33606-5_25 (zitiert auf S. 22, 41, 44–47, 57, 59, 60, 68).

[BBK+13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, J. Wettinger. „Integrated Cloud Application Provisioning: Interconnecting Service-centric and Script-centric Management Technologies“. In: *Proceedings of the 21st International Conference on Cooperative Information Systems (CoopIS 2013)*. Springer, 2013. DOI: 10.1007/978-3-642-41030-7_9 (zitiert auf S. 15).

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. „Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA“. In: *2014 IEEE International Conference on Cloud Engineering*. IEEE, 2014, S. 87–96. DOI: 10.1109/IC2E.2014.56 (zitiert auf S. 15, 23, 27, 30, 40, 42, 43, 67).

[BCS18] A. Brogi, A. Canciani, J. Soldani. „Fault-aware management protocols for multi-component applications“. In: *Journal of Systems and Software* 139 (2018), S. 189–210. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.02.005. URL: https://www.sciencedirect.com/science/article/pii/S016412121830030X (besucht am 28. 09. 2022) (zitiert auf S. 15).

[BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. „The OpenTOSCA Ecosystem – Concepts & Tools“. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016* (2016), S. 112–130. DOI: 10.5220/0007903201120130 (zitiert auf S. 34).

[BH21] F. Beetz, S. Harrer. „GitOps: The Evolution of DevOps?“ In: *IEEE Software* 39.4 (8. Okt. 2021), S. 70–75. DOI: 10.1109/MS.2021.3119106 (zitiert auf S. 24).

[BSW14] A. Brogi, J. Soldani, P. Wang. „TOSCA in a Nutshell: Promises and Perspectives“. In: *Service-Oriented and Cloud Computing*. Hrsg. von M. Villari, W. Zimmermann, K.-K. Lau. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, S. 171–186. ISBN: 978-3-662-44879-3. DOI: 10.1007/978-3-662-44879-3_13 (zitiert auf S. 15).

[CBB+19] R. Coppen, A. Banks, E. Briggs, K. Borgendale, R. Gupta. *MQTT Version 5.0*. 7. März 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html (besucht am 28. 09. 2022) (zitiert auf S. 23, 24, 54, 56, 57, 61).

[CCP15] J. Carrasco, J. Cubo, E. Pimentel. „Towards a Flexible Deployment of Multi-cloud Applications Based on TOSCA and CAMP“. In: *Advances in Service-Oriented and Cloud Computing*. Hrsg. von G. Ortiz, C. Tran. Cham: Springer International Publishing, 28. Feb. 2015, S. 278–286. ISBN: 978-3-319-14886-1. DOI: 10.1007/978-3-319-14886-1_26 (zitiert auf S. 33).

[CFV13] A. Celesti, M. Fazio, M. Villari. „SE CLEVER: A secure message oriented Middleware for Cloud federation“. In: *2013 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2013, S. 000035–000040. DOI: 10.1109/ISCC.2013.6754919 (zitiert auf S. 32).

[Cis20] Cisco. *Cisco Annual Internet Report (2018–2023) White Paper*. 9. März 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (besucht am 28. 09. 2022) (zitiert auf S. 15).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. „Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications“. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, S. 22–27 (zitiert auf S. 22).

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec. „The many faces of publish/subscribe“. In: *ACM computing surveys (CSUR)* 35.2 (2 Juni 2003), S. 114–131. DOI: 10.1145/857076.857078 (zitiert auf S. 23, 24, 54).

[ENo+22] EarthmanT (Github User), J. Niezgoda, ofercloudify (Github User), kubama (Github User), M. Shnizer, AdarShaked (Github User), tyacbovi (Github User). *Official Plugins*. 17. Juli 2022. URL: https://docs.cloudify.co/6.4.0/working_with/official_plugins/ (besucht am 28. 09. 2022). Github Quelltext https://github.com/cloudify-cosmo/docs.getcloudify.org/blob/dd6063b653b6773c28e89ceb3c35ffe949d395b1/content/working_with/official_plugins/_index.md (zitiert auf S. 35).

[EoB+20] EarthmanT (Github User), ofercloudify (Github User), P. Brodsky, J. Niezgoda, AdarShaked (Github User). *Blueprint Files and Packages*. 2. Dez. 2020. URL: https://docs.cloudify.co/6.4.0/developer/blueprints/ (besucht am 28. 09. 2022). Github Quelltext https://github.com/cloudify-cosmo/docs.getcloudify.org/blob/d67674aaa2f870f7a0e06bbbf9a236ce5483529c/content/developer/blueprints/_index.md (zitiert auf S. 35).

[FBH+17] A. C. Franco da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, R. Steinke. „Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER)*. ScitePress. SciTePress Digital Library, 2017, S. 358–367. DOI: 10.5220/0006243303580367 (zitiert auf S. 33).

[fra21] francoischapuzot (Github User). *Orchestrators*. 26. Dez. 2021. URL: https://alien4cloud.github.io/common/features.html#/documentation/3.5.0/orchestrators/orchestrators.html (besucht am 28. 09. 2022). Github Quelltext https://github.com/alien4cloud/alien4cloud.github.io/blob/c7a5649dcabc4aaa7d5a1069c9c8848d38f63cd7/documentation/3.5.0/orchestrators/orchestrators.html (zitiert auf S. 35).

[FRL05] C.-L. Fok, G.-C. Roman, C. Lu. „Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications“. In: *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE, Juni 2005, S. 653–662. ISBN: 0-7695-2331-5. DOI: 10.1109/ICDCS.2005.63 (zitiert auf S. 34).

[FSK05] B. Ford, P. Srisuresh, D. Kegel. „Peer-to-Peer Communication Across Network Address Translators.“ In: *USENIX Annual Technical Conference, General Track*. 2005, S. 179–192 (zitiert auf S. 16).

[Goy14] S. Goyal. „Public vs private vs hybrid vs community-cloud computing: a critical review“. In: *International Journal of Computer Network and Information Security* 6.3 (2014), S. 20–29. DOI: 10.5815/ijcnis.2014.03.03 (zitiert auf S. 20).

[HAR14] H. Herry, P. Anderson, M. Rovatsos. „Choreographing configuration changes“. In: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. IEEE, 30. Jan. 2014, S. 156–160. DOI: 10.1109/CNSM.2013.6727828 (zitiert auf S. 33).

[HBF+18] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, F. Leymann. „Pattern-based Deployment Models and Their Automatic Execution“. In: *11th IEEE/ACM International Conference on Utility and Cloud Computing UCC 2018, 17–20 December 2018, Zurich, Switzerland*. IEEE Computer Society, 2018, S. 41–52. DOI: 10.1109/UCC.2018.00013 (zitiert auf S. 72).

[HBS+16] P. Hirmer, U. Breitenbücher, A. C. F. da Silva, K. Képes, B. Mitschang, M. Wieland. „Automating the Provisioning and Configuration of Devices in the Internet of Things." In: *Complex Systems Informatics and Modeling Quarterly* 9 (Dez. 2016), S. 28–43. ISSN: 2255-9922. DOI: 10.7250/csimq.2016-9.02 (zitiert auf S. 33).

[HHW99] R. S. Hall, D. Heimbigner, A. L. Wolf. „A Cooperative Approach to Support Software Deployment Using the Software Dock". In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE '99. Los Angeles, California, USA: Association for Computing Machinery, Mai 1999, S. 174–183. ISBN: 1581130740. DOI: 10.1145/302405.302463 (zitiert auf S. 33).

[HP11] G. Halkes, J. Pouwelse. „UDP NAT and Firewall Puncturing in the Wild". In: *NETWORKING 2011*. Hrsg. von J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont, C. Scoglio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 1–12. ISBN: 978-3-642-20798-3 (zitiert auf S. 16).

[HYA+15] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, S. U. Khan. „The rise of "big data" on cloud computing: Review and open research issues". In: *Information systems* 47 (2015), S. 98–115. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2014.07.006. URL: https://www.sciencedirect.com/science/article/pii/S0306437914001288 (besucht am 28. 09. 2022). Elsevier (zitiert auf S. 19).

[II] Institute of Architecture of Application Systems (IAAS), Institute for Parallel and Distributed Systems (IPVS). *OpenTOSCA*. End-to-end toolchain for your cloud applications. Universität Stuttgart. URL: https://www.opentosca.org/ (besucht am 28. 09. 2022) (zitiert auf S. 34).

[Int19] International Organization for Standardization. *ISO/IEC 27018:2019*. Information technology - Security techniques - Code of practice for protection of personally identifiable information (PII) in public clouds acting as PII processors. Techn. Ber. Jan. 2019. URL: https://www.iso.org/standard/76559.html (besucht am 28. 09. 2022) (zitiert auf S. 64).

[JD12] G. Juve, E. Deelman. „Automating Application Deployment in Infrastructure Clouds". In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE, 19. Jan. 2012, S. 658–665. ISBN: 978-1-4673-0090-2. DOI: 10.1109/CloudCom.2011.102 (zitiert auf S. 34).

[KB17] O. Kopp, U. Breitenbücher. „Choreographies are key for distributed cloud application provisioning". In: *Proceedings of the 9th Central European Workshop on Services and their Composition (ZEUS 2017)*. 2017, S. 67–70 (zitiert auf S. 34, 37).

[KBKL18] C. Krieger, U. Breitenbücher, K. Képes, F. Leymann. „An Approach to Automatically Check the Compliance of Declarative Deployment Models". In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSoC 2018)*. IBM Research Division, 2018, S. 76–89 (zitiert auf S. 64).

[KBL+19] K. Képes, U. Breitenbücher, F. Leymann, K. Saatkamp, B. Weder. „Deployment of Distributed Applications Across Public and Private Networks". In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 30. Dez. 2019, S. 236–242. DOI: 10.1109/EDOC.2019.00036 (zitiert auf S. 15, 16, 26, 27, 34, 35, 42, 54).

[KGR20] H. Koziolek, S. Grüner, J. Rückert. „A Comparison of MQTT Brokers for Distributed IoT Edge Computing“. In: *Software Architecture*. Hrsg. von A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, O. Zimmermann. Cham: Springer International Publishing, 20. Sep. 2020, S. 352–368. ISBN: 978-3-030-58923-3. DOI: https://doi.org/10.1007/978-3-030-58923-3_23 (zitiert auf S. 57).

[LCM22] A. Luzar, M. Cankar, A. Maslennikov. *xOpera TOSCA orchestrator*. 30. März 2022. URL: https://github.com/xlab-si/xopera-opera/blob/96a8954810f11d7656450aae7a8f452d014a3f93/README.md (besucht am 28. 09. 2022) (zitiert auf S. 35).

[Lig] R. Light. *mosquitto.conf man page*. URL: https://mosquitto.org/man/mosquitto-conf-5.html (besucht am 28. 09. 2022) (zitiert auf S. 57).

[LML14] T. Lorido-Botran, J. Miguel-Alonso, J. A. Lozano. „A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments“. In: *Journal of Grid Computing* 12.4 (1. Dez. 2014), S. 559–592. ISSN: 1572-9184. DOI: 10.1007/s10723-014-9314-7 (zitiert auf S. 59).

[LPB+15] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, P. Manzoni. „A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks“. In: *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*. IEEE, 16. Juli 2015, S. 931–936. DOI: 10.1109/CCNC.2015.7158101 (zitiert auf S. 16).

[LS18] A. Luoto, K. Systä. „Fighting network restrictions of request-response pattern with MQTT“. In: *Iet Software* 12.5 (1. Okt. 2018), S. 410–417. DOI: 10.1049/iet-sen.2017.0251 (zitiert auf S. 16).

[LSC20] A. Luzar, S. Stanovnik, M. Cankar. „Examination and comparison of tosca orchestration tools“. In: *Software Architecture. Communications in Computer and Information Science*. European Conference on Software Architecture. Hrsg. von H. Muccini, P. Avgeriou, B. Buhnova, J. Camara, M. Caporuscio, M. Franzago, A. Koziolek, P. Scandurra, C. Trubiani, D. Weyns, U. Zdun. Bd. 1269. Springer. Springer International Publishing, 7. Sep. 2020, S. 247–259. ISBN: 978-3-030-59155-7. DOI: 10.1007/978-3-030-59155-7_19 (zitiert auf S. 35).

[LVCD13] F. Li, M. Vögler, M. Claeßens, S. Dustdar. „Towards Automated IoT Application Deployment by a Cloud-Based Approach“. In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications* (16. Dez. 2013). IEEE, 2013, S. 61–68. DOI: 10.1109/SOCA.2013.12 (zitiert auf S. 33).

[MCRG15] E. Markoska, I. Chorbev, S. Ristov, M. Gusev. „Cloud portability standardization overview“. In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2015, S. 286–291. DOI: 10.1109/MIPRO.2015.7160281 (zitiert auf S. 15).

[Mer14] D. Merkel. „Docker: Lightweight Linux Containers for Consistent Development and Deployment“. In: *Linux Journal* 2014.239 (März 2014). ISSN: 1075-3583 (zitiert auf S. 25).

[MG11] P. Mell, T. Grance. *The NIST Definition of Cloud Computing*. Techn. Ber. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2011 (zitiert auf S. 17, 19, 20, 59).

[MLH10] M. Mao, J. Li, M. Humphrey. „Cloud auto-scaling with deadline and budget constraints“. In: *2010 11th IEEE/ACM International Conference on Grid Computing*. 2010, S. 41–48. DOI: 10.1109/GRID.2010.5697966 (zitiert auf S. 59).

[MN10] U. Manzoor, S. Nefti. „QUIET: A Methodology for Autonomous Software Deployment using Mobile Agents“. In: *Journal of Network and Computer Applications* 33.6 (Nov. 2010): *Advances on Agent-based Network Management*, S. 696–706. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2010.03.015 (zitiert auf S. 34).

[MSDC22] I. Maddaus, T. Smith, T. Duffield, S. Chisamore. *Chef Infra Client Security*. 7. Juni 2022. URL: https://docs.chef.io/chef_client_security/ (besucht am 28. 09. 2022). Github Quelltext https://github.com/chef/chef-web-docs/blob/3e2cf4475802f518b6e772fc0b79bd07859278c8/content/chef_client_security.md (zitiert auf S. 34).

[NAoi22] J. Niezgoda, AlexanderDesmond(Github User), ofercloudify (Github User), idobcloudify (Github User). *Agents*. 1. Aug. 2022. URL: https://docs.cloudify.co/6.4.0/install_maintain/agents/ (besucht am 28. 09. 2022). Github Quelltext https://github.com/cloudify-cosmo/docs.getcloudify.org/blob/d67674aaa2f870f7a0e06bbbf9a236ce5483529c/content/install_maintain/agents/_index.md (zitiert auf S. 35).

[NtM+22] M. Neumann, tyacbovi (Github User), L. Maksymczuk, L. Maksymczuk, ofercloudify (Github User). *Built-in Node Types*. 17. Aug. 2022. URL: https://docs.cloudify.co/6.4.0/developer/blueprints/built-in-types/ (besucht am 28. 09. 2022). Github Quelltext https://github.com/cloudify-cosmo/docs.getcloudify.org/blob/d67674aaa2f870f7a0e06bbbf9a236ce5483529c/content/developer/blueprints/built-in-types.md (zitiert auf S. 35).

[OAS20a] OASIS. *TOSCA Simple Profile in YAML Version 1.3*. OASIS Standard. Hrsg. von M. Rutkowski, C. Lauwers, C. Noshpitz, C. Curescu. 26. Feb. 2020. URL: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html (besucht am 28. 09. 2022). Chairs: P. Lipton, C. Lauwers (zitiert auf S. 15, 23, 41).

[OAS20b] OASIS. *TOSCA Version 2.0. Committee Specification Draft 03*. OASIS Standard. Hrsg. von C. Lauwers, C. Curescu. 28. Sep. 2020. URL: https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd03/TOSCA-v2.0-csd03.docx (besucht am 28. 09. 2022). Chairs: P. Lipton, C. Lauwers (zitiert auf S. 15, 20–23, 38, 39, 42, 44, 48, 61, 65).

[OAS20c] OASIS. *TOSCA Version 2.0. Committee Specification Draft 01*. OASIS Standard. Hrsg. von C. Lauwers, C. Curescu. 23. Apr. 2020. URL: https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd01/TOSCA-v2.0-csd01.docx (besucht am 28. 09. 2022). Chairs: P. Lipton, C. Lauwers (zitiert auf S. 21).

[OGP03] D. Oppenheimer, A. Ganapathi, D. A. Patterson. „Why do Internet services fail, and what can be done about it?“ In: *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*. 2003, S. 1–16 (zitiert auf S. 15).

[Onea] OneCommons Co. *Ensembles*. URL: https://docs.unfurl.run/ensembles.html (besucht am 28. 09. 2022) (zitiert auf S. 35).

[Oneb] OneCommons Co. *How it works*. URL: https://unfurl.run/howitworks.html (besucht am 28. 09. 2022) (zitiert auf S. 35).

[OST16] J. Opara-Martins, R. Sahandi, F. Tian. „Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective“. In: *Journal of Cloud Computing* 5.1 (15. Apr. 2016), S. 1–18. ISSN: 2192-113X. DOI: 10.1186/s13677-016-0054-z (zitiert auf S. 20).

[PBL+17] A. Panarello, U. Breitenbücher, F. Leymann, A. Puliafito, M. Zimmermann. „Automating the Deployment of Multi-Cloud Applications in Federated Cloud Environments“. In: *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS'16. ACM, Mai 2017. ISBN: 978-1-63190-141-6. DOI: 10.4108/eai.25-10-2016.2266363 (zitiert auf S. 26, 32, 33).

[PK00] G. J. Pottie, W. J. Kaiser. „Wireless integrated network sensors“. In: *Communications of the ACM* 43.5 (2000), S. 51–58. DOI: 10.1145/332833.332838 (zitiert auf S. 16, 29, 45, 51).

[PS13] D. Palma, T. Spatzier, Hrsg. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. OASIS Standard. 25. Nov. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (besucht am 28. 09. 2022). Charis: P. Lipton, S. Moser (zitiert auf S. 21).

[Pup] Puppet, Inc. *Overview of Puppet's architecture*. URL: https://puppet.com/docs/puppet/5.5/architecture.html (besucht am 28. 09. 2022) (zitiert auf S. 34).

[QCB19] C. Qu, R. N. Calheiros, R. Buyya. „Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey“. In: *ACM Computing Surveys* 51.4 (Juli 2019). ISSN: 0360-0300. DOI: 10.1145/3148149 (zitiert auf S. 59, 61).

[Ras] Raspberry Pi Foundation. *Raspberry Pi 3 Model B+*. URL: https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/ (besucht am 28. 09. 2022) (zitiert auf S. 68).

[Ric17a] A. Richardson. *GitOps - Operations by Pull Request*. 7. Aug. 2017. URL: https://www.weave.works/blog/gitops-operations-by-pull-request (besucht am 28. 09. 2022) (zitiert auf S. 24, 61, 64).

[Ric17b] A. Richardson. *The GitOps Pipeline - Part 2*. 30. Aug. 2017. URL: https://www.weave.works/blog/the-gitops-pipeline (besucht am 28. 09. 2022). Blog (zitiert auf S. 24, 25, 61, 63).

[SBK+16] A. C. F. da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann. „OpenTOSCA for IoT: Automating the Deployment of IoT Applications Based on the Mosquitto Message Broker“. In: *Proceedings of the 6th International Conference on the Internet of Things*. IoT'16. New York, NY, USA: Association for Computing Machinery, 7. Nov. 2016, S. 181–182. ISBN: 9781450348140. DOI: 10.1145/2991561.2998464 (zitiert auf S. 33).

[SBK18] K. Saatkamp, U. Breitenbücher, F. Kopp Oliverand Leymann. „Application Scenarios for Automated Problem Detection in TOSCATopologies by Formalized Patterns“. In: *Papers From the 12th Advanced Summer School on Service-OrientedComputing (SummerSOC'18)*. IBM Research Division, 2018, S. 43–53 (zitiert auf S. 64).

[SBKL17] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. „Topology Splitting and Matching for Multi-Cloud Deployments“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, Apr. 2017, S. 247–258. ISBN: 978-989-758-243-1 (zitiert auf S. 30).

[SBKL20] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. „Method, formalization, and algorithms to split topology models for distributed cloud application deployments“. In: *Computing* 102.2 (Feb. 2020), S. 343–363. ISSN: 1436-5057. DOI: 10.1007/s00607-019-00721-8 (zitiert auf S. 26, 30, 31).

[Sj22] A. Souzis, jozo (Github User). *Introduction*. 24. Mai 2022. URL: https://github.com/onecommons/unfurl/blob/2cb17707828f28d527548c553456fd6a13df42d7/README.md (besucht am 28. 09. 2022) (zitiert auf S. 35).

[SM11] A. Sampaio, N. Mendonça. „Uni4Cloud: An Approach Based on Open Standards for Deployment and Management of Multi-Cloud Applications“. In: *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*. SECLOUD '11. ACM, Mai 2011, S. 15–21. ISBN: 9781450305822. DOI: 10.1145/1985500.1985504 (zitiert auf S. 34).

[SY16] S. H. Shah, I. Yaqoob. „A survey: Internet of Things (IOT) technologies, applications and challenges“. In: *2016 IEEE Smart Energy Grid Engineering (SEGE)*. IEEE. 2016, S. 381–385. DOI: 10.1109/sege.2016.7589556 (zitiert auf S. 15).

[TCDM21] O. Tomarchio, D. Calcaterra, G. Di Modica, P. Mazzaglia. „TORCH: a TOSCA-Based Orchestrator of Multi-Cloud Containerised Applications“. In: *Journal of Grid Computing* 19.1 (18. Feb. 2021), S. 5. ISSN: 1572-9184. DOI: 10.1007/s10723-021-09549-z (zitiert auf S. 33).

[TPM+17] G. Tricomi, A. Panarello, G. Merlino, F. Longo, D. Bruneo, A. Puliafito. „Orchestrated Multi-Cloud Application Deployment in OpenStack with TOSCA“. In: *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2017, S. 1–6. DOI: 10.1109/SMARTCOMP.2017.7947027 (zitiert auf S. 33).

[VPM+21] T. Vaiyapuri, V. S. Parvathy, V. Manikandan, N. Krishnaraj, D. Gupta, K. Shankar. „A novel hybrid optimization for cluster-based routing protocol in information-centric wireless sensor networks for IoT based mobile edge computing“. In: *Wireless Personal Communications* (27. Jan. 2021), S. 1–24. DOI: 10.1007/s11277-021-08088-w (zitiert auf S. 16).

[WBB+20] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. „The EDMM Modeling and Transformation System“. In: *Service-Oriented Computing – ICSOC 2019 Workshops. Lecture Notes in Computer Science*. Hrsg. von S. Yangui, A. Bouguettaya, X. Xue, N. Faci, W. Gaaloul, Q. Yu, Z. Zhou, N. Hernandez, E. Y. Nakagawa. Bd. 12019. Cham: Springer International Publishing, 24. Apr. 2020, S. 294–298. ISBN: 978-3-030-45989-5. DOI: 10.1007/978-3-030-45989-5_26 (zitiert auf S. 30).

[WBB+21] M. Wurster, U. Breitenbücher, A. Brogi, F. Diez, F. Leymann, J. Soldani, K. Wild. „Automating the Deployment of Distributed Applications by Combining Multiple Deployment Technologies“. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. Science und Technology Publications, Lda, 2021, S. 178–189. ISBN: 978-989-758-510-4. DOI: 10.5220/0010404301780189 (zitiert auf S. 26, 28–30, 38).

[WBF+19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. „The essential deployment metamodel: a systematic review of deployment automation technologies“. In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (26. Aug. 2019), S. 63–75. ISSN: 2524-8529. DOI: 10.1007/s00450-019-00412-x (zitiert auf S. 20, 28–30).

[WBK+20] K. Wild, U. Breitenbücher, K. Képes, F. Leymann, B. Weder. „Decentralized Cross-Organizational Application Deployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models“. In: Lecture Notes in Computer Science 12127 (Juni 2020), S. 20–35. DOI: 10.1007/978-3-030-49435-3_2 (zitiert auf S. 15, 16, 22, 26–29, 33, 35).

[Wea] I. Weaveworks. *Guide To GitOps*. What you need to know. URL: https://www.weave.works/technologies/gitops/ (besucht am 28. 09. 2022) (zitiert auf S. 24, 25, 61–63, 72).

[Wea18] Weaveworks. *What Is GitOps*. 21. Aug. 2018. URL: https://www.weave.works/blog/what-is-gitops-really (besucht am 28. 09. 2022). Blog (zitiert auf S. 24).

[Wea22] Weaveworks. *Getting Started With Weave GitOps*. 21. Juni 2022. URL: https://www.weave.works/blog/getting-started-with-weave-gitops (besucht am 28. 09. 2022). Blog (zitiert auf S. 17, 24).

[Woo02] M. Wooldridge. „Intelligent Agents: The Key Concepts“. Lecture Notes in Computer Science. In: *Multi-Agent Systems and Applications II*. Hrsg. von V. Mařík, O. Štěpánková, H. Krautwurmová, M. Luck. Bd. 2322. Berlin, Heidelberg: Springer Berlin Heidelberg, 10. Apr. 2002, S. 3–43. ISBN: 978-3-540-45982-8. DOI: 10.1007/3-540-45982-0_1 (zitiert auf S. 37).

[WWG22] K. Wang, K. Wang, H. Gill. *Terraform Cloud Agents*. 22. Juni 2022. URL: https://www.terraform.io/cloud-docs/agents (besucht am 28. 09. 2022). Github Quelltext https://github.com/hashicorp/terraform-docs-agents/blob/b99e60fbe5dd1ea46bffcd0b87b161e081f8964f/website/docs/cloud-docs/agents/index.mdx (zitiert auf S. 34).

[ZBF+17] M. Zimmermann, U. Breitenbü, M. Falkenthal, F. Leymann, K. Saatkamp. „Standards-based Function Shipping - How to use TOSCA for Shipping and Executing Data Analytics Software in Remote Manufacturing Environments“. In: *Proceedings of the 2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC 2017)*. IEEE Computer Society, 2017, S. 50–60. DOI: 10.1109/EDOC.2017.16 (zitiert auf S. 33).

Alle URLs wurden zuletzt am 28. 09. 2022 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift