



# Modellierung und Simulation von dynamischen Container-basierten Software-Architekturen in Palladio

Bachelorarbeit von

Nathan Hagel

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

| | |
|---|---|
| Erstgutachter: | Prof. Dr. Anne Koziolek |
| Zweitgutachter: | Prof. Dr. Ralf H. Reussner |
| Betreuender Mitarbeiter: | Dipl.-Inform Jörg Henß |
| Zweiter betreuender Mitarbeiter: | Dipl.-Inform Martina Rapp |

10. Januar 2022 – 10. Mai 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

# Zusammenfassung

Moderne verteilte Software-Systeme werden heute nicht mehr statisch auf Maschinen deployed. Die gewünschten Komponenten/Container und deren Skalierungen werden deklarativ definiert. Eine Kontrollschleife versucht dann, den vorgegebenen Zustand dynamisch durch Starten und Stoppen von Containern und Pods zu erreichen. Auch Skalierung von Diensten und Rollouts von neuen Produktversionen lassen sich auf diese Art realisieren.
Die Auswirkungen auf die Performance und Skalierbarkeit der Anwendung beim Einsatz dieser Techniken sind bisher nur schwer vorhersagbar.
Ziel dieser Arbeit ist die Verbesserung der Modellierung und Simulation von dynamischen Container-basierten Software-Architekturen in Palladio.
Mithilfe von Kubernetes lässt sich Software auf diese Art deployen.
In dieser Arbeit wird untersucht, ob sich Container, sowie grundlegende Kubernetes-Konstrukte wie Pods und Services mithilfe des Palladio Component Model (PCM) abbilden lassen. Dynamische Deployments containerisierter Software könnten so mit Palladio abgebildet werden.
Ein Teil dieser Arbeit besteht dabei aus einer PCM-Erweiterung zur Abbildung der grundlegenden Kubernetes-Konstrukte und Container.
Um die Verwendung dieser Erweiterung zu beschreiben, wurde ein Workflow definiert, um mit Kubernetes deployte Software, sowie die deklarative Beschreibung eines Wunschzustands in Palladio umzusetzen und statisch zu simulieren. Damit der Übergang von einer deklarativen Beschreibung der gewünschten Container hin zu einer konkreten Allokierung in einem Kubernetes-Cluster umgesetzt werden kann, wurde ein Pod-Allokations-Scheduler für Palladio-Modelle entwickelt und prototypisch implementiert. Abschließend wurde ein dynamisches Simulationskonzept für Palladio erarbeitet, wobei Kontrollschleifen abgebildet wurden, die den Zustand des Kubernetes-Clusters überwachen.
Zur Evaluation wurde ein Referenz-Cluster definiert, welches mithilfe der PCM-Erweiterung und des definierten Workflows abgebildet wurde.
Zusammenfassend wurde festgestellt, dass dynamische Deployments, sowie containerisierte Software-Systeme mit Palladio abbildbar und simulierbar sind.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

Moderne Software wird heutzutage nicht mehr statisch auf einem einzelnen Server bzw. Endgerät deployed. Stattdessen werden Komponenten oder Container definiert, die Dienste anbieten. Diese Komponenten beziehungsweise Dienste werden dann unabhängig von der zugrundeliegenden technischen Infrastruktur, wie ein großer oder mehrere kleine Server, bedarfsabhängig skaliert. Ein weiterer Trend ist, dass das Hosting der Dienste nicht mehr in selbstbetriebenen Rechenzentren stattfindet [39]. Als Alternative werden Cloud-Dienstleister wie Amazon Web Services (AWS) oder Google Cloud Platform (GCP) verwendet, welche Server bereitstellen. Oft bieten diese Cloud-Dienstleister bereits skalierbare Ressourcen in Kombination mit der Möglichkeit an, Software dynamisch zu deployen [3], [7]. Ein weit verbreiteter Standard hierfür ist Kubernetes.
Die Vorteile dieses Vorgehens sind vielfältig. Die wegfallende Notwendigkeit, eigene Server zu unterhalten, ebenso wie die gebannte Gefahr, viel zu groß definierte Infrastruktur zu beschaffen, nur damit seltene Lastspitzen abgefangen werden können, sind nur zwei Vorteile. Zur Umsetzung solcher dynamischen, containerbasierten Deployments gibt es Containerorchestrierungswerkzeuge wie Kubernetes. Kubernetes ermöglicht es, Deployments von Software deklarativ zu definieren, also, zum Beispiel, wie viele Instanzen eines Containers parallel laufen sollen [12]. Kubernetes übernimmt dann das Erstellen und bei Bedarf auch das Entfernen von Containern [13]. Diese Einzelcontainer müssen, obwohl sie möglicherweise zur gleichen Software gehören, nicht auf dem gleichen Endsystem laufen, sondern können auf ein Netz an Rechenressourcen verteilt werden. Ebenso können Software-Systeme mit Kubernetes dynamisch skaliert werden. Ein definierter, zur Laufzeit veränderbarer Zustand wird dabei automatisch mithilfe von Kontrollschleifen überprüft und Änderungen automatisch bei Bedarf vorgenommen [14]. Bei der Überlegung, wie eine Software deployed werden soll, stellt sich die Frage, wie sich ein containerbasiertes Deployment auf die Performance der Software auswirkt. Die Auswirkungen auf die Performance bei Einsatz von Kubernetes oder ähnlichen Werkzeugen ist bisher nur schwer zu bestimmen, insbesondere im Vergleich zu einem klassischen, statischen Deployment. Das Ziel ist die Modellierung und Simulation von dynamischen Container-basierten Software-Architekturen in Palladio zu verbessern. Dieses Vorgehen soll genauere Vorhersagen der Performance einer dynamischen containerisierten Software bereits zur Design-Zeit ermöglichen.
Die vorliegende Arbeit beschäftigt sich mit dieser Thematik, indem sie folgende Forschungsfragen vollständig oder teilweise beantwortet:

1. Forschungsfrage: *Wie können Container in Palladio modelliert werden?*

2. Forschungsfrage: *Welche Kubernetes-Konstrukte lassen sich in Palladio abbilden?*

3. Forschungsfrage: *Wie lassen sich nicht statische Deployments in Palladio abbilden und simulieren?*

4. Forschungsfrage: *Wie lassen sich Skalierung Horizontaler Pod Autoskalierer (HPA) und Rollouts simulieren?*

5. Forschungsfrage: *Wie lassen sich Auswirkungen von Throtteling und Resource-Sharing schon zur Design-Zeit vorhersagen?*

Um diese Fragen zu bearbeiten, werden zuerst die notwendigen Grundlagen in Kapitel 2 erläutert. Kapitel 3 beschreibt eine Modellerweiterung des Palladio Component Model (PCM) zur Umsetzung der Abbildung von Containern und Kubernetes-Konstrukten in Palladio. In Kapitel 4 wird später auf Basis der Modellerweiterung ein Workflow beschrieben, der die Abbildung von Kubernetes-Clustern in Palladio Schritt für Schritt ermöglicht. Da in Kubernetes nicht manuell bestimmt wird, wo welcher Teil der Anwendung läuft, wird für die Modellerweiterung ein Allokations-Scheduler entwickelt und prototypisch implementiert. Zum Abschluss wird ein Konzept zur Simulation dynamischer containerisierter Software-Architekturen mit Palladio entwickelt, bevor verwandte Arbeiten beschrieben und die Arbeit evaluiert wird.

# 2 Grundlagen

In diesem Kapitel werden die Grundlagen und zugrundeliegenden Technologien für diese Arbeit beschrieben. Das beinhaltet das Container-Orchestrierungswerkzeug Kubernetes, den Architektursimulator Palladio, sein Komponentenmodell, und die Simulationsfunktionalität SimuLizar.

## 2.1 Kubernetes

Kubernetes (griechische Übersetzung für: Steuermann, Pilot; kurz K8s) ist eine Spezifikation, die viele verschiedene Dienste, die im Kontext von containerbasierter Software bei Deployment und Organisation benötigt werden, anbietet. Kubernetes basiert auf Borg, einem proprietären Container-Orchestrierungs-Werkzeug von Google. 2014 wurde Kubernetes als Open-Source Projekt veröffentlicht und der Cloud Native Computing Foundation gestiftet. Heutzutage wird es von vielen großen Firmen beeinflusst, die diese Stiftung unterstützen. Dieser Umstand, das allgemeine Design von Kubernetes, sowie dessen einfache Erweiterbarkeit, haben dazu geführt, dass ein Ökosystem um Kubernetes entstanden ist, das offen für viele Anpassungen und Erweiterungen ist und viel Funktionalität anbietet [2, 11 ff.].
Kubernetes definiert dabei einen portablen Standard, der von Cloud-Anbietern oder Firmen für die eigenen Bedürfnisse angepasst wird. Die Grundkonzepte sind aber bei allen Varianten im Allgemeinen gleich [15]:
Die Kern-Funktionalität von Kubernetes ist die Containerorchestrierung, das Verwalten von Containern und Diensten sowie das Verteilen eingehender Last. Das bedeutet, dass in Containern gekapselte Dienste oder Anwendungen über verschiedene Endsysteme verteilt werden können, ohne dass dies der Software bekannt sein muss, oder spezielle Mechanismen zur Entwurfszeit vorgesehen werden müssen. Auch können mehrere Instanzen, die so genannten Replicas, des gleichen Dienstes konfiguriert werden, um beispielsweise einen erhöhten Durchsatz zu ermöglichen. Kubernetes abstrahiert von dieser Ebene und verteilt einkommende Anfragen auf die verschiedenen Anwendungsinstanzen.
Der Zustand eines Clusters kann in Kubernetes deklarativ definiert werden. Kubernetes überprüft dann periodisch mithilfe von verschiedenen Kontrollschleifen, ob der gewünschte Zustand des Clusters vorliegt und veranlasst bei Bedarf Veränderungen [4, 65 ff.]. Ein Beispiel hierfür wäre, dass eine bestimmte Anzahl an Pods deklarativ spezifiziert wird. Laufen zu wenige oder zu viele Instanzen des Pods im Cluster, werden weitere automatisch gestartet oder überflüssige gestoppt.
Mithilfe von Kubernetes können diese Punkte nicht nur vor dem Starten des Systems definiert werden, sondern auch zur Laufzeit [4, 186 f.]. Der aktuelle sowie der gewünschte Zustand werden in einem Key-Value-Store, dem etcd gespeichert. Die Kontrollschleifen

greifen dann auf diesen zu. Um diese Funktionalitäten anzubieten werden verschiedene Einheiten und Konzepte benötigt, die im Folgenden vorgestellt werden.

### 2.1.1 Cluster

Ein Kubernetes-Cluster ist, einfach ausgedrückt, eine Bündelung von Rechenressourcen. Hierbei ist es nicht notwendig, dass mehrere physische Maschinen an einem Cluster beteiligt sind. Konkret bedeutet das, dass ein Cluster sowohl aus einer oder mehreren physischen bzw. virtuellen Maschinen bestehen kann. Das Cluster stellt das äußerste Konzept in Kubernetes dar und enthält eigentlich alle Objekte. Auf einem Kubernetes Cluster können eine oder mehrere containerisierte Anwendungen laufen. Ein Kubernetes Cluster ist organisiert in sog. Nodes [4, p. 69f], siehe Unterabschnitt 2.1.2. Das Cluster ist die Grundlage für jede mit Kubernetes deployte Anwendung.

### 2.1.2 Nodes

Nodes (deutsch: Knoten) befinden sich in einem Kubernetes-Cluster und können physische, wie auch virtuelle Maschinen sein. Sie besitzen eine Spezifikation, wie viele Ressourcen (bspw. CPU und RAM) für die jeweilige Node zur Verfügung stehen. Es wird unterschieden zwischen zwei Arten von Nodes: Einmal die Worker-Nodes (deutsch: Arbeiter-Knoten) und die Master-Node (deutsch: Meister-Knoten). Die Master-Node ist für die Organisation und Einhaltung der definierten Spezifikationen der Softwaresysteme im Cluster zuständig. Sie beinhaltet unter anderem verschiedene Kontrollschleifen, die den Zustand des Clusters überprüfen sowie in vielen Fällen den Key-Value-Store etcd. Eine Komponente, die diese Aufgabe übernimmt ist der kube-controller-manager. Dieser kube-controller-manager erkennt zum Beispiel neue Knoten im Cluster. Eine andere wichtige Komponente auf der Master-Node ist der kube-scheduler. Dieser verteilt nicht allokierte Pods siehe Unterabschnitt 2.1.3 auf andere Nodes [4, 38 ff.].
Die Worker-Nodes sind für die Ausführung der Container, gekapselt in Pods, zuständig. Das bedeutet, dass Pods üblicherweise auf Worker-Nodes deployed werden. Bei der Konfiguration von Kubernetes-Clustern und dem Deployen von Systemen mithilfe von Kubernetes, wird mit der Master-Node, bzw. ihrer Schnittestelle kommuniziert [4, p. 69f].

### 2.1.3 Pods

Pods (deutsch: Gruppe/Schote) sind die kleinsten deploybaren Einheiten die in Kubernetes erstellt und verwaltet werden können [16]. Pods beinhalten einen oder mehrere Container, die in der gleichen Umgebung laufen. Das bedeutet, dass sie beispielsweise einen gemeinsamen Speicher teilen. Verschiedene Pod-Instanzen können auf verschiedenen Endsystemen laufen. Die Container einer Pod-Instanz müssen dagegen auf der selben Maschine laufen. Es ist auch erlaubt, dass mehrere Instanzen des gleichen Pods auf derselben Node laufen. Da Pods Container beinhalten, in denen Systeme laufen, muss sichergestellt werden, dass genügend Ressourcen zur Verfügung stehen, bevor ein weiterer Pod auf einer Node deployed werden kann. Hierzu kann spezifiziert werden, wie viele Ressourcen ein Pod benötigt. Dies ist die Summe der benötigten Ressourcen der Container, die auf dem Pod laufen [17].

Der Typ und die Menge an Pods die in einem Cluster laufen sollen kann deklarativ mithilfe eines Deployments, siehe Unterabschnitt 2.1.11, spezifiziert werden. Dabei handelt es sich bei einer Deployment-Spezifikation um eine Datei, die die gewünschten Eigenschaften enthält. Pods können auch manuell gestartet und gestoppt werden [16].

### 2.1.4 Container

Bei Containern handelt es sich um eine Art Betriebssystem-Virtualisierung, ähnlich wie virtuelle Maschinen, jedoch leichtgewichtiger. Ein Container enhält dabei alles was er benötigt, um als eigenständige Software-Komponente zu funktionieren. Container unterstützen Techniken zur Isolierung von Ressourcen, wie control groups (cgroups), siehe Unterabschnitt 2.1.14 [4, p. 9f]. Durch die Verwendung von Containern kann die zugrundeliegende Plattform, auf welcher der Container läuft, ausgetauscht werden, ohne dass sich das Verhalten ändert. Ein Container kann ein Container-Image enthalten. Dieses Container-Image beinhaltet die Anwendung, sowie alle Abhängigkeiten die benötigt werden. In Kubernetes können Container als Teil eines Pods deployed werden, nicht aber direkt ohne Pod, siehe Unterabschnitt 2.1.3. Das Deployment von Containern in Kubernetes besitzt in vielen Fällen sogenannte Requests und Limits, siehe Unterabschnitt 2.1.6 und Unterabschnitt 2.1.7. Diese beschreiben die Anforderungen an den Knoten, auf welchem Sie deployed werden sollen [18].

### 2.1.5 Resource Units in Kubernetes

Kubernetes definiert Ressourcen, wie CPU und Speicher, mithilfe von CPU-Einheiten und Speicher in Bytes. Bei der Angabe von Speicher wird die benötigte Menge in Bytes angegeben. Bei CPU-Einheiten verwendet Kubernetes CPU-Kerne als Basiseinheit. Hierbei kann ein Kern (virtuell oder physisch) aufgeteilt werden. Eine Angabe von 0.5 bedeutet: 50% der CPU-Rechenzeit eines zur Verfügung stehenden Kerns wird von diesem Container benötigt. Als Alternative zu Angaben als Gleitkommazahl findet man oft die Verwendung von millicores (m). Hierbei entspricht 1.0 = 1000m bzw. die oben verwendeten 0.5 entsprechen 500m. Die kleinste zuweisbare Einheit ist 1m, also vergleichbar zu einem tausendstel CPU-Kern. Es können auch Werte größer 1 angefordert werden, also mehr als ein CPU-Kern [19]. Kubernetes garantiert dabei nicht, dass man dabei einen dedizierten Kern erhält. Für eine Angabe von 1000m wären vier viertel-CPUs ebenso korrekt.

### 2.1.6 Requests

Requests definieren, wie viele Ressourcen, meist CPU und Speicher, von einem Container oder Pod benötigt werden. Hierbei verwendet Kubernetes Requests als weiche Grenze, wenn ein Pod bereits auf einem Knoten läuft. Um zu überprüfen, ob ein Pod beispielsweise auf einer Node noch Platz hat, werden die Requests jedoch als Metrik verwendet um diese Überprüfung durchzuführen. Das bedeutet, dass Nodes CPU-Einheiten und Speicher zur Verfügung haben, aber nur ein weiterer Pod auf eine Node deployed werden kann, wenn noch genug Speicher und CPU-Einheiten zur Verfügung stehen, siehe Abbildung 2.1. Das ist unabhängig von der aktuellen Auslastung der Node, bzw. der darauf laufendenen

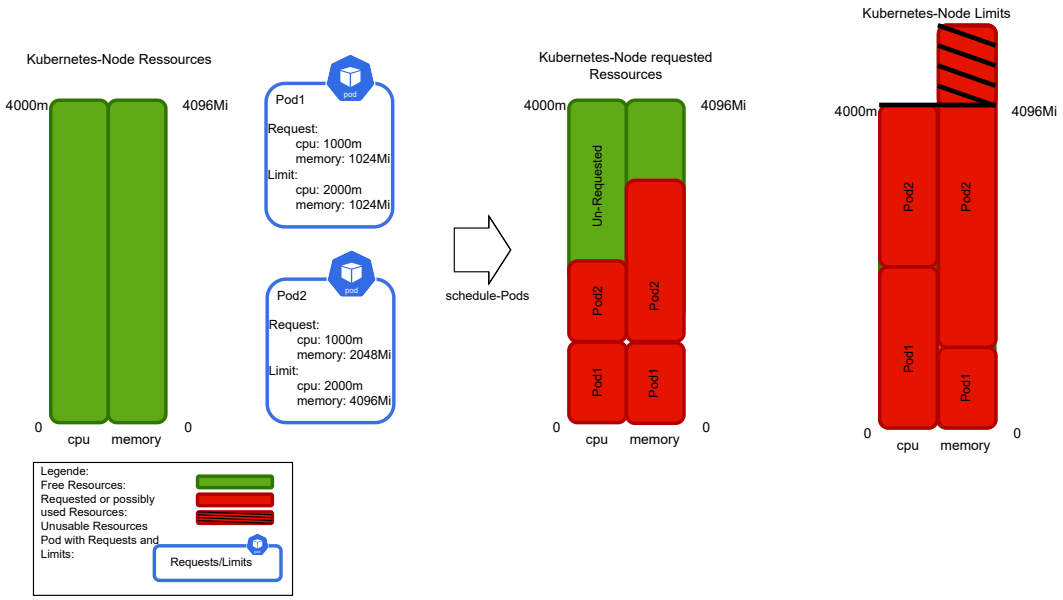


Abbildung 2.1: Überblick Requests und Limits

Pods. Selbst wenn alle Pods gerade keine Arbeit hätten und dementsprechend theoretisch Ressourcen zur Verfügung stünden, kann ein weiterer Pod nur dann zur Node hinzugefügt werden, wenn die Summe der bereits existierenden Requests, addiert mit der Request des neuen Pods kleiner ist als die der Node zur Verfügung stehenden Ressourcen. Unter gewissen Umständen können Pods aber mehr Ressourcen verwenden, als in der Request definiert. Dies funktioniert nur so lange, wie andere Pods gerade weniger verbrauchen bzw. noch überschüssige Ressourcen vorhanden sind. Es können auch zusätzliche Ressourcen definiert werden, die über CPU und Speicher hinausgehen. Als Beispiel kann definiert werden, dass ein Pod gerne eine SSD zur Verfügung hätte. [17]. In Abbildung 2.1 sind auf der linken Seite die Ressourcen einer Node als Balken dargestellt. Sie besitzt 4000m also 4 Kerne und 4096Mi Speicher. Daneben wurden Requests und Limits von Pods spezifiziert, die auf der rechten Seite des Pfeils auf dem Pod allokiert wurden. Dabei werden einmal die reservierte Ressourcen durch die Requested-Resources dargestellt. Man erkennt, dass noch Pods mit maximal 2000m CPU und 1024Mi Memory auf diesem Knoten allokiert werden könnten. Auf der rechten Seite sieht man dabei eine Abbildung der Limits der beiden allokierten Pods. Man erkennt, dass laut den Limits kein Platz mehr für weitere Pods existieren, bzw. bereits jetzt die Limits größer sind als die Ressourcen der Node. Das ist aber nicht problematisch, da Limits nicht garantierte Ressourcen darstellen im Gegensatz zu Requests. Es kann trotzdem sein, dass Pod2 in der Abbildung nahe 4096Mi Memory zur Ausführung erhält, wenn Pod1 gerade keine Ressourcen benötigt.

### 2.1.7 Limits

Limits definieren in Kubernetes harte Grenzen, wie viele Ressourcen von einem Container oder Pod verwendet werden dürfen. Das bedeutet, dass das Limit eines Containers höher sein kann, als die Request, die für die Scheduling-Entscheidung eines Pods auf die Nodes verwendet wird. Limits bedeuten also, dass ein Pod von Kubernetes daran gehindert wird, weitere Ressourcen zu benutzen, obwohl möglicherweise weitere Ressourcen frei verfügbar sind. Eine mögliche Auswirkung bei der Verwendung von zu viel CPU ist eine Drosselung bzw. bei der Verwendung von zu viel Speicher eine Abschaltung des Pods [17].

### 2.1.8 Allokations-Scheduling in Kubernetes

Kubernetes übernimmt die Auswahl der Node, auf der ein Pod laufen soll. Die Auswahl übernimmt dabei der kube-scheduler. Jeder Pod stellt bestimmte notwendige und teilweise optionale Anforderungen an die Node, auf der er laufen soll. Ein wichtiges Beispiel und für diese Arbeit die wichtigste Anforderung ist die Ressourcenanforderung des Pods.
Der kube-scheduler überprüft in den zwei Schritten: 1. Filtering (Filtern) und 2. Scoring (Punktevergabe), welche Node für einen zu allokierenden Pod ausgewählt wird. Im ersten Schritt (Filtering), wird ermittelt welche Nodes überhaupt die notwendigen Anforderungen erfüllen und schließt die anderen aus. Ein Beispiel hierfür ist, dass wenn eine Node bereits all ihren Speicher mit Requests von anderen Pods belegt hat, wird diese Node im Filtering-Schritt vom kube-scheduler bereits ausgeschlossen, solange der zu allokierende Pod mindestens eine Einheit Speicher anfordert. Nach diesem ersten Schritt wird für jede übrig gebliebene Node eine Punktzahl errechnet, wie gut die noch zur Auswahl stehenden Nodes sich eignen. Der Knoten mit der höchsten Punktzahl wird zurückgegeben und der Pod dann auf diesem erstellt. Sollten zwei Knoten die gleiche Punktzahl erreichen, wird einer von beiden zufällig ausgewählt.
Die Berechnung der Punktzahl kann nach definierbaren Regeln erfolgen. Wenn die einzige Anforderung an das Scheduling ist, dass genügend Ressourcen frei sind, dann würde jede Node, die diese Anforderung erfüllt auch die gleiche Punktzahl im Scoring erhalten. Für den Fall, dass es keine Node gibt, die die notwendigen Anforderungen, also beispielsweise die notwendigen in der Request festgelegten Ressourcen, erfüllt, wird der Pod vorerst auf keine Node zugeteilt. Dieser Pod verbleibt dann im Cluster und wird möglicherweise in der nächsten Iteration der hierfür zuständigen Kontrollschleife ge-scheduled, falls der Scheduler dann eine Node findet. Die Scheduling-Strategie kann überschrieben werden [20]. Ein Beispiel hierfür wäre, dass das Scheduling auf weiteren Informationen beruhen soll, als mit dem kube-scheduler evaluiert werden.

### 2.1.9 Quality-of-Service-Klassen

Kubernetes verwendet sogenannte Quality of Service (QoS)-Klassen, um zu entscheiden, welcher Pod abgeschaltet wird, sollten wegen zu wenig zur Verfügung stehenden Speichers Pods abgeschaltet werden müssen. Die QoS Klassen beschreiben gleichzeitig, welche Ressourcen-Zusagen für einen ge-scheduleten Pod gelten. Jedem Pod wird bei dessen Erstellung eine QoS-Klasse zugeteilt. Kubernetes verwendet drei verschiedene Klassen:

Guaranteed, Burstable und BestEffort.

Die Klasse **Guaranteed** wird vergeben, wenn jeder Container eines Pods ein Speicher- und ein CPU-Limit und ein Speicher- und ein CPU-Request hat. Außerdem müssen das Speicher- und das CPU-Limit und Request exakt den gleichen Wert haben. Konkret bedeutet dies, dass Kubernetes, sollte der Pod ge-scheduled werden, auf jeden Fall die spezifizierten Ressourcen reserviert, sodass der Pod diese dann auch verwenden kann.
Im Abschnitt über die Kubernetes Resource Units wurde beschrieben, dass Pods oder Container CPU-Anteile zugewiesen bekommen. Es besteht aber auch die Möglichkeit dedizierte CPU-Kerne einzelnen Pods zuzuweisen. Notwendig hierfür ist, dass sich der Pod in der QoS-Klasse *Guaranteed* befindet. Zusätzlich muss die CPU Request ganzzahlig sein (`CPU Request: 1`). Andernfalls wird, trotz einer Spezifikation von 1000m als CPU-Request, dem Pod nur anteilig CPU Ressourcen im Wert von einem CPU Kern aber kein dedizierter Kern zugewiesen.

**Burstable** wird vergeben, wenn die Guaranteed-Kriterien nicht zutreffen, jedoch mindestens einer der Container innerhalb des Pods ein CPU- oder Speicher-Request hat.

Als letztes kommt die Klasse **BestEffort**. Hierfür darf kein Container innerhalb des Pods ein Speicher- oder CPU-Request oder Limit haben. Das bedeutet, dass Kubernetes, solange Ressourcen frei sind, nach gewissen Scheduling-Regeln diesen Containern Ressourcen zur Verfügung stellt, sollte Last an diesen Container bzw. Pods kommen [21].

### 2.1.10 Replica-Sets

Ein Replica-Set in Kubernetes beschreibt, wie viele Pod-Instanzen eines bestimmten Pods im Cluster laufen sollen. Hierfür werden Angaben über einen Selector zur Identifizierung der Pods, die zu diesem Replica-Set gehören, gemacht. Außerdem wird die Anzahl der Pods angegeben, die laufen sollen. Das beschreibt den gewünschten Zustand des Clusters bezogen auf diese Pods. Ein Beispiel hierfür wäre: „Es sollen 3 Pods vom Typ A im Cluster laufen“Um diesen Zustand zu erreichen, werden Pods durch das Replica-Set erstellt oder gelöscht. Diese Veränderungen werden mithilfe einer Kontrollschleife getriggert. Diese Kontrollschleife vergleicht zur Laufzeit den aktuellen Zustand des Clusters mit dem im Replica-Set definierten gewünschten Zustand. Wenn der Zustand nicht übereinstimmt, werden bei zu vielen Pod-Instanzen Pods gelöscht bzw. bei zu wenigen, neue Pods erstellt. Um zu definieren, auf welchen Pod sich das Replica Set bezieht, wird das Image eines Pods angegeben. Replica-Sets können von Deployments, siehe Unterabschnitt 2.1.11, einem höheren Konzept, verwaltet werden. Im Allgemeinen gilt die Best-Practice nicht Replica-Sets direkt zu verwenden, sondern Deployments für die Beschreibung des gewünschten Cluster Zustands. Diese erstellen und verwalten dann ihre eigenen Replica-Sets, bieten zusätzlich aber noch verschiedene nützliche Funktionen, siehe Unterabschnitt 2.1.11 [13].

### 2.1.11 Deployments

Deployments beschreiben, ähnlich wie Replica-Sets einen gewünschten Zustand eines Clusters. Dieser beinhaltet beispielsweise, dass 3 Pods eines bestimmten Typs im Cluster laufen sollen. Das Deployment bietet in Kubernetes darüber hinaus viele weitere Funktionalitäten an, weshalb die Nutzung eines Deployments statt Replica-Sets empfohlen wird [13].
Es können zum Beispiel Pods aktualisiert werden, indem das Pod-Template, also das Image auf welches in der Deployment-Spezifikation verwiesen wird, geändert wird. Der Deployment-Controller, der diese Änderungen vornimmt, führt dann die notwendigen Schritte aus, sodass im Betrieb diese Änderungen vorgenommen werden können. Auch können nach Veränderungen eines Deployments diese wieder umgekehrt werden. Das kann beispielsweise notwendig sein, wenn die neue Version nicht stabil läuft. Um wieder zur alten Version zu gelangen, kann auf die von Kubernetes automatisch erstellte Historie zugegriffen werden und ein sogenannter Rollback durchgeführt werden.
Auch hier werden diese Schritte dann automatisch durch den Deloyment-Controller durchgeführt.
Typischerweise benötigen alle Änderungen und Updates Zeit. Der Deployment-Controller überprüft, ähnlich wie der Replica-Set-Controller, nur in einer definierbaren Wiederholungsrate, ob der gewünschte Zustand, der definiert wurde, aktuell im Cluster vorliegt. Um zu identifizieren, welche Pods zu welchem Deployment gehören, wird ein Selector-Feld angegeben. Die Pods besitzen dann den Selector und der Deployment-Controller kann diese damit identifizieren [12]. Der Deployment-Controller macht dann die Änderungen im Cluster. Auf einem Cluster können mehrere Deployments laufen, jedoch kann ein Deployment nur ein Pod-Image referenzieren, also wird für jeden Pod-Typ der in dem Cluster laufen soll mindestens ein Deployment benötigt [13]. In Listing 1 ist ein Beispiel einer Deployment-Spezifikation zu sehen. Der Name des Deployments ist `nginx-deployment`. Als label, welches zur Identifikation der Pods dient, dass das Deployment selbst diese findet, wurde hier `metadata.labels.app: nginx` gewählt. Das Deployment erstellt drei Replicas vom angegebenen Pod durch die `template`-Felder. Diese enthalten in diesem Beispiel einen Container, der ein Container-Image referenziert in der angegebenen Version.

### 2.1.12 Service

Ein Service stellt eine Möglichkeit dar, eine Anwendung über eine Schnittstelle zu erreichen. Hierbei übernimmt Kubernetes die Service-Discovery sowie das Loadbalancing für einkommende Anfragen dieses Services. Hintergrund ist, dass in Kubernetes nur ein gewünschter Zustand des Clusters vom Nutzer definiert werden kann. Der Zustand kann sich aber dynamisch ändern. Das bedeutet, dass die Menge an laufenden Pods nicht fest ist. Da jeder Pod bei der Erstellung eine IP-Adresse bekommt, ändern sich also die IP-Adressen, die einen spezifischen Dienst anbieten, unter Umständen immer wieder. Der Service selbst ist dagegen unter einer festen IP-Adresse innerhalb des Clusters zu finden. Die Anwendungen oder Pods, die einen gewissen Service benötigen, brauchen durch das Service-Konzept in Kubernetes also keine Informationen, welcher Pod gerade wo läuft. Stattdessen wird ein Service definiert, der angeboten wird und die eingehenden Requests

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Listing 1: Deployment-Spezifikation Beispiel [12]

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Listing 2: Service Specification [22]

werden auf die Pods, welche diesen Service anbieten, verteilt. Das bedeutet, ein Service in Kubernetes übernimmt auch das Loadbalancing. Eine Möglichkeit ist, die einkommenden Requests an einen zufälligen, beim Service registrierten Pod, zu senden.
Pods die zu einem Service gehören werden mithilfe eines Selectors, siehe Listing 2, identifiziert. Für diese Pods wird dann ein Endpoint erstellt. Der Service überprüft, welche Pods oder Deployments die gleichen Label wie der Service definiert haben. Ein Beispiel für ein Label aus Listing 2 ist `app: MyApp` oder die Spezifikation eines Namens. Weitere Informationen in einer Service-Spezifikation sind ports, wie `port: 80` im Beispiel. Diese Angabe spezifiziert, unter welchem Port der Service erreichbar ist. Der Target-Port spezifiziert, unter welchem Port der Pod für diesen Service erreicht werden soll [22].

In Kubernetes gibt es verschiedene Typen von Services: Der Standardfall (optionale Typspezifikation) ist der ClusterIP-Service, wobei diese Services nur innerhalb des Clusters aber nicht von außen erreicht werden können. Desweiteren gibt es Headless-Services, NodePort-Services und LoadBalancer-Services. Headless-Services werden verwendet, wenn der anfragende Client direkt mit einem bestimmten Pod oder Pods untereinander mit einem spezifischen Pod kommunizieren wollen oder müssen. Der Standardfall ClusterIP, wie oben beschrieben, würde bei einer Anfrage an den Service, die Anfrage an einen zufälligen Pod weiterleiten. Es kann aber von Nöten sein, bei nicht zustandslosen Appliktionen oder Pods, dass eine Anfrage immer wieder an den selben Pod gehen muss. Hierfür werden Headless-Services verwendet. NodePort-Services sind Services, die in einem Knoten existieren und unter einem statischen Port der Node erreichbar sind. Das bedeuetet, dass dieser NodePort-Service im Gegensatz zu ClusterIP-Services auch von außerhalb des Clusters erreichbar ist. Eine Alternative zu den NodePorts sind die LoadBalancer-Services. Diese verwenden externe Load-Balancer der Cloud-Provider. Dadurch wird der Zugriff auf die Worker-Nodes, die einen NodePort-Service anbieten, beschränkt. Von außen kann also nicht direkt auf die Worker-Nodes über Node-IP und Port zugegriffen werden, sondern der Load-Balancer des Cloud-Providers macht das. Das ist eine sicherere Variante im Vergleich zu NodePort-Services, da das Öffnen von Ports ins Cluster eine Sicherheitslücke darstellen kann und externe Loadbalancer oft ein ausgereifteres Loadbalancing umsetzen [22].

### 2.1.13 Ingress

Ein Ingress beschreibt eine Schnittstelle von außen in das Cluster beziehungsweise für die Dienste, die das Cluster anbietet. Ein Ingress verteilt dabei eingehende Anfragen auf die Services im Cluster, die diese wiederum an die Pods verteilen. Üblicherweise wird ein Ingress statt eines NodePort-Services verwendet, um Dienste des Clusters von außen zugänglich zu machen, da es üblicherweise nicht gewollt ist, dass externe Clienten direkt mit Nodes im Cluster kommunizieren [25].

### 2.1.14 cgroups v2 in Kubernetes

Dieser Abschnitt liefert eine sehr oberflächliche Erklärung, was cgroups sind. Für eine detailliertere Betrachtung kann die Kernel-Spezifikation verwendet werden, siehe [11]. Im Rahmen der vorliegenden Arbeit ist in erster Linie ein grobes Verständnis über diese Technologie notwendig. Deshalb wird nicht weiter auf die technischen Feinheiten von cgroups eingegangen, sondern diese nur grob eingeordnet. cgroups stellen eine Möglichkeit dar, Ressourcen eines Systems hierarchisch zu organisieren und zu konfigurieren. Auf cgroups basieren Containertechnologien wie Docker. Auch Kubernetes als Containerorchestrierungswerkzeug benutzt cgroups zur Zuweisung von Ressourcen an Pods und Container. cgroups erstellen eine Baumstruktur, in welcher Systemressourcen wie Speicher oder CPU verteilt werden können. In dieser Baumstruktur können Teil-Bäume existieren, die beispielsweise die Ressourcen eines Pods organisieren. Die Prozesse eines Pods laufen dann in dieser cgroup. Container die in dem Pod existieren, können als Blätter des Teilbaums angesehen werden. Diese können dann ebenso ihre eigenen Ressourcenspezifikationen haben. Die in Unterabschnitt 2.1.6 und Unterabschnitt 2.1.7 beschriebenen Requests und Limits eines Containers können so mithilfe von cgroups realisiert werden [11], [23].

## 2.2 Palladio

In diesem Kapitel werden die für die vorliegende Arbeit wichtigen Grundlagen von Palladio erklärt. Dabei wird insbesondere auf das PCM eingegangen, welches im Rahmen dieser Arbeit erweitert wird. Die Modellierung von PCM-Modellen kann mit der Palladio-Bench durchgeführt werden. Sie besteht aus verschiedenen Eclipse-Erweiterungen, die die Domain Specific Language (DSL) PCM unterstützen. Mithilfe der Palladio-Bench können Software Architekturen in Eclipse entweder über einen graphischen Editor oder einen Baumeditor erstellt oder bestehende Modelle aus XML-Dateien geladen werden. Auf Basis dieser können dann Performance-Analysen durchgeführt werden, die mithilfe eines gegebenen Szenarios die Antwortzeit, die Ressourcenausnutzung oder den Durchsatz bestimmen können. Mithilfe der Palladio-Bench können so verschiedene Softwarearchitekturen in der Entwurfsphase miteinander verglichen werden [33].

### 2.2.1 Palladio Component Model

Das PCM (deutsch: Palladio Komponenten Modell) ist eine domänenspezifische Sprache (DSL), um für Software bzw. Software-Architekturen bereits zur Entwurfszeit Vorhersagen

über die Performanz des zu entwickelnden Systems zu treffen. Das PCM basiert auf dem Ansatz, wie der Name schon beinhaltet, dass Software-Komponenten zu einem größeren System zusammengesetzt werden können. Die Performance von Software-Komponenten hängt von vielen Faktoren ab: der konkreten Implementierung der Komponente, dem Nutzungsverhalten, der Laufzeitumgebung und anderen Diensten oder Komponenten, die für die Ausführung des eigenen Dienstes notwendig sind. Palladio bietet die Möglichkeit Einflussfaktoren zu spezifizieren, beziehungsweise im PCM abzubilden. Für die Laufzeitumgebung können beispielsweise die Hardwarespezifikationen geändert werden. Um das Nutzungsverhalten zu simulieren, werden probabilistische Verteilungsfunktionen verwendet. Diese können verändert werden, um verschiedenste Nutzungsverhalten zu simulieren. Eine Veränderung der Allokation von Komponenten auf den zur Verfügung stehenden Hardwareknoten kann ebenso zu einer Veränderung der Performance eines Systems führen. Das PCM bietet auch hierfür eine Lösung zur Simulation an [33].

### 2.2.2 Palladio-Komponenten und -Konzepte

Dieses Kapitel beschreibt die für diese Arbeit wichtigen PCM-Komponenten und -Konzepte. Diese werden auf Metamodell-Ebene beschrieben. Zudem wird auf deren Implementierung eingegangen, sofern das für diese Arbeit relevant ist. Da es sich bei dieser Arbeit um eine Erweiterung des PCM auf den verschiedenen Ebenen (*Viewpoints*) von Palladio handelt, wird hier ein Großteil der grundlegenden Palladio-Konzepte beschrieben, um später die Verwendung oder Erweiterung des Metamodells zu beschreiben.

**Palladio Viewpoints und Viewtypes** Der Palladio-Ansatz ist aufgeteilt in drei sogenannte *Viewpoints* [34, p. 43ff.]. Diese *Viewpoints* definieren jeweils Belange, welche von verschiedenen, mit dem Modell oder System interagierenden Rollen, benötigt werden. Für diese Arbeit sind nur die *Architectural Viewpoints* in Palladio relevant, da Viewpoints nicht auf Palladio oder die Modellierung einer Software beschränkt sind. Die Viewpoints lauten: *Structural*, *Behavioral* und *Deployment* [34, p. 43f.]. Der *Decision Viewpoint* wird dabei in dieser Arbeit nicht weiter betrachtet, da dieser nicht erweitert wird.

**Structural Viewpoint** Der *Structural Viewpoint* beschreibt die statischen Belange sowie die Struktur eines Systems. Der Structural Viewpoint beinhaltet sowohl den *Repository View Type* als auch den *Assembly View Type*. Er wird dargestellt durch das *Repository Model*.

**Repository Model:** Das *Repository Model* enthält Software-Komponenten, Schnittstellen (Interfaces) und Beziehungen zwischen den Komponenten und Schnittstellen. Diese Beziehungen sind *provides* und *requires* Beziehungen und beschreiben, dass eine Komponente eine Schnittstelle anbietet oder benötigt. Im PCM sind Komponenten und Schnittstellen voneinander getrennt. Ein Interface wird dadurch als allgemeine Schnittstelle angesehen und ist nicht direkt gekoppelt mit einer Komponente. Erst eine Beziehung zwischen Komponente und Interface beschreibt das Anbieten oder Benötigen eines Interfaces. Im *Repository* können zusätzlich Datentypen definiert werden. Das *Repository* beschreibt welche Komponenten zur Verfügung stehen und welche Schnittstellen benötigt oder angeboten werden.

Ein *Repository* ist dabei systemunabhängig. Komponenten und Interfaces können also für verschiedene Software-Systeme verwendet bzw. definiert werden. Zu den wichtigsten Komponententypen im *Repository Model* zählen die *Basic Component*, die *Composite Component* und das *SubSystem*. Die *Basic Component* beschreibt eine Software-Komponente. *Provides-* und *Requires*-Verbindungen von den Komponenten zu Interfaces beschreiben die angebotenen bzw. benötigten Schnittstellen. Eine Schnittstelle hat dabei Operationen. Für jede Operation der benötigten Schnittstellen einer *Basic Component* wird ein *Verhalten* spezifiziert [34, p. 48]. Diese Vehalten, die auch Hardwareressourcenanforderungen enthalten können werden genauer auf Seite 18 beschrieben. Die *Composite Component* ist eine Sonderform der einfachen Komponente. Eine *Composite Component* wird aus Komponenten zusammengesetzt. Sie kann ebenso wie *Basic Components* Schnittstellen anbieten und benötigen. Diese Schnittstellen können dann an die inneren Komponenten delegiert werden. *Composite Components* können dann, ebenso wie *Basic Components* im *System Model* mithilfe eines *Assembly Contexts* instanziiert werden. Ein wichtiger Unterschied zu einem *System* oder *SubSystem* ist, dass eine *Composite Component* nicht auf verschiedenen Hardware-Umgebungen deployed werden kann. Die *Composite Component* muss also immer als ganze Einheit auf einem Hardwareknoten allokiert werden. Man spricht hier von einer Black-Box-Ansicht für den *System Deployer* [28]. Eine Alternative zur *Composite Component* ist das *SubSystem*. Ein *SubSystem* ist aus Sicht des *System Deployers* eine White-Box. Der *System Deployer* kann also die inneren Komponenten sehen und diese auch auf verschiedenen Hardwareknoten allokieren. Das *SubSystem* ist hierbei eine Gruppierung von *Basic Components* und *Composite Components*. Das *SubSystem* kann bei der logischen Aufteilung und Wiederverwendung hilfreich sein [34, p. 44ff.]. Strukturell gibt es keinen großen Unterschied zwischen einer *Composite Component* und einem *SubSystem*, die White-Box / Black-Box aus Sicht des *System Deployers* einmal ausgenommen [28].

Der *Assembly View Type* definiert die Struktur eines Systems. Konkret werden Instanzen von Komponenten aus dem Repository in Form von so genannten *Assembly Contexts* modelliert. Diese *Assembly Contexts* bieten die *provided Interfaces* an bzw. benötigen die *Required Interfaces* aus dem *Repository Model*. Im *System Model* werden diese Schnittstellen als *Operation Provided/Required Role* dargestellt. Diese Rollen referenzieren dann das entsprechende *Interface*.

**System Model:** Ein System wird im *System Model* modelliert. Ein Beispiel eines *System Models* aus dem Standard-Palladio-Template des MediaStores aus den Screencasts [32] ist in Abbildung 2.2 zu sehen. In diesem Beispiel wurden Komponenten, wie die *MediaManagement*-Komponente in einem *Assembly Context* instanziiert. In einem *System Model* kann es mehrere Instanzen der gleichen Komponente geben. Diese können dann bei Bedarf unterschiedlich benannt werden. *Assembly Contexts* bieten die *provided Interfaces* als *provided Roles* an. Die *required Interfaces* werden zu *required Roles*. Diese Rollen werden dargestellt in der „lollipop notation“ [34, p. 49]. Die *Provided Role IMediaManagement* des *Assembly Contexts MediaManagement* ist als Lollipop dargestellt. Die *Required Role IMediaAccess* desselben *Assembly Contexts* wird als Fassung abgebildet. Eine Verbindung zwischen einer *Provided Role* und einer *Required Role* wird mithilfe eines Pfeils dargestellt. Dieser Pfeil wird *Assembly Connector* genannt. Ein *Assembly Connector* kann nur

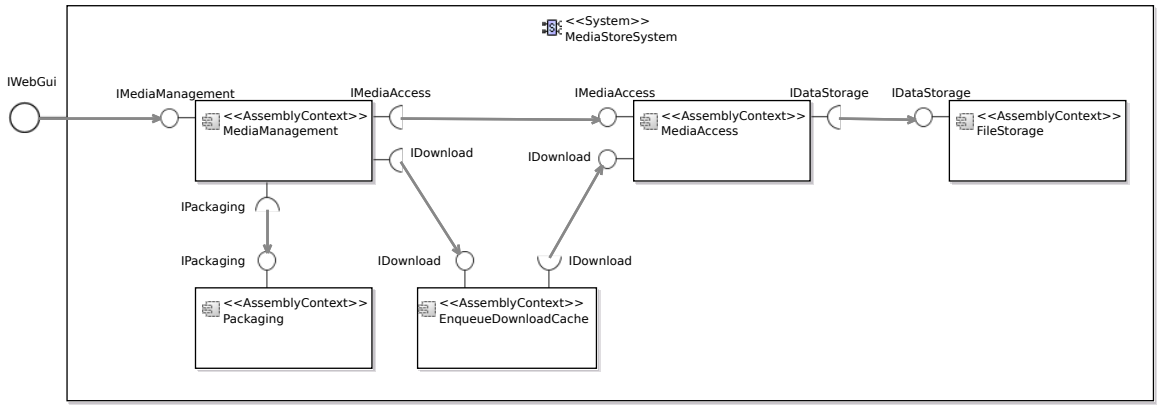


Abbildung 2.2: System Model des Beispiels MediaStore [32]

Rollen mit integrierten Schnittstellen des selben Typs miteinander verbinden. *Assembly Connectors* dienen dem Verbinden und Weiterleiten von Anfragen. Die Funktionalitäten sind dabei in den Komponenten gekapselt, nicht in den Übergängen.

In Abbildung 2.2 befinden sich die *Assembly Contexts* in einem *System*. Wie in Abbildung 2.2 zu sehen ist, bietet das *System MediaStoreSystem* auch eine Rolle und darin enthalten eine Schnittstelle an. Die angebotetene Rolle *IWebGui* beinhaltet das *Interface IMediaManagement*. Diese Schnittstelle wird mithilfe eines sogenannten *Delegation Connectors* an eine Rolle innerhalb des Systems delegiert. Auch hier muss das in der Rolle enthaltene Interface vom gleichen Typ sein, wie die innere Rolle, an welche weiterdelegiert wird. Diese Funktionalität gibt es auch für benötigte Schnittstellen, die dann nach außen delegiert werden können. *Delegation Connectoren* werden allgemein verwendet, um eine Schnittstelle einer *Composed Structure* mit einer Schnittstelle bzw. passenden Rolle eines *Assembly Contexts* zu verbinden. Weitere *Composed Structures* sind bspw. die *Composite Component* und das *SubSystem*. Ein *System* muss mindestens eine Schnittstelle anbieten [34, p. 50]. Zusätzlich kann ein *System* nicht Teil einer anderen zusammengesetzten Komponente sein. Grund hierfür sind die *Quality of Service*-Annotationen für die Systemschnittstellen. Diese charakterisieren die Qualität eines benötigten Diensts, beispielsweise mithilfe einer Antwortzeit. *Quality of Service*-Annotationen können nur an *Required Roles* vergeben werden. Die zugehörigen angebotenen Schnittstellen in Form von *Provided Roles* können dann von *Usage Models* verwendetet werden. Ein *System* kann wiederum auf Hardware allokiert werden, also alle *Assembly Contexts* die im *System Model* zu finden sind, werden Hardwareknoten zugeordnet, siehe Seite 19. Die inneren Komponenten eines *Systems* können einzeln auf Hardware deployed werden. Es muss also nicht das gesamte System auf demselben Hardwarecontainer laufen [29].

**System vs. SubSystem vs. Composite Component** In diesem Abschnitt werden die Unterschiede zwischen *System*, *SubSystem* und *Composite Component* herausgearbeitet. Ziel dieses Abschnittes ist eine fundierte Gegenüberstellung dieser Elemente, um die Frage zu

beantworten, welches bestehende Konzept sich am besten für die Abbildung eines Pods bzw. eines Containers eignet und den Gedankengang für die Abbildung dieser beiden Konzepte im Rahmen der vorliegenden Arbeit besser darzustellen. Die Beantwortung dieser Frage ist für die Arbeit, welche sich unter anderem mit dem Scheduling von Pods beschäftigen soll, von großer Wichtigkeit und auf Seite 22 beziehungsweise 24 zu finden.

Tabelle 2.1 zeigt die Gemeinsamkeiten und Unterschiede der drei Palladio-Konzepte *System*, *SubSystem* und *Composite Component*. Um die Frage zu beantworten, welches dieser drei Konzepte am besten für die Abbildung eines Pods oder Containers geeignet ist, muss überlegt werden, was auf dieser Ebene die notwendigen Eigenschaften eines Containers oder Pods sind. Diese Eigenschaften sollten dann auch das entsprechende Palladio-Konzept haben beziehungsweise das Palladio-Konzept sollte keine der wichtigsten Eigenschaften verbieten. Pods, siehe Unterabschnitt 2.1.3 bzw. Container, siehe Unterabschnitt 2.1.4, sind für die Betrachtung auf der *Structural View* im Allgemeinen gleich. Beide können eine oder mehrere Komponenten, (im Fall von Pods sind es eben Container) enthalten. Zusätzlich müssen sie auf derselben Hardware-Ressource deployed werden und können mehrfach instanziiert werden. Ein Blick in die Tabelle 2.1 zeigt, dass nur die *Composite Component* all diese Eigenschaften enthält. Es liegt also nahe, Pods und Container mithilfe von *Composite Components* abzubilden oder diese als Grundlage für die Erweiterung bzw. Neueinführung eines Containers oder Pods auf der *Structural View* zu verwenden. Zusätzlich ist die Sicht auf eine *Composite Component* aus Sicht des *System Deployers* auch logisch korrekt, wenn wir Pods und Container ausschließlich als Deploymentkonzept betrachten. Die weiteren Sichten der verschiedenen Rollen, *Software Architect* und *Component Developer*, sind ebenso in der Tabelle 2.1 aufgeführt, haben jedoch keinen Einfluss auf die Auswahl der *Composite Component* an dieser Stelle. Als Beispiel einer Abbildung aus dem MediaStore könnte eine beliebige *Basic Component*, wie z.B. MediaManagement in eine *Composite Component* verpackt werden. Diese *Composite Component* kann dann als Container, welcher in diesem Fall nur MediaManagement enthält, verwendet werden. Um einen Pod aus mehreren Containern darzustellen, bietet es sich dann an, eine *Composite Component* mit Inhalt einer oder mehrerer anderer *Composite Components* zu definieren. Diese äußere *Composite Component* kann dann zum Beispiel der Spezifikation eines Pods genügen und möglicherweise für ein Deployment des MediaStores mithilfe von Kubernetes verwendet werden. Je nach Anforderung in der Deployment-Spezifikation kann dieser Pod dann einmal oder mehrmals mithilfe eines *Assembly Contexts* im *System Model* instanziiert werden. In einem späteren Schritt, dem Allokations-Scheduling der Pods, ist dann durch die Eigenschaften der *Composite Component* gewährleistet, dass dieser Pod als *Composite Component* vollständig, ohne Aufteilung in die Einzelkomponenten, auf der gleichen Hardware-Ressource allokiert wird. Die Frage, an welcher Stelle die Ressourcen Requests und Limits in diese Überlegung einfließen, wird später auf seite 28 betrachtet. Nichtsdestotrotz liefert der in diesem Paragraphen angesprochene Ansatz: „Composite Components als Oberklasse von Pods **und** Containern zu verwenden“ einige Schwächen im Bezug auf das akkurate Scheduling im Bezug auf cgroups. Die tatsächlich gewählte Lösung wird in Absatz 3.1.1 und Absatz 3.1.1 besprochen und im weiteren Verlauf der Arbeit diskutiert.

| **Eigenschaft** | **System** | **SubSystem** | **Composite Component** |
|---|---|---|---|
| Sicht des System-Deployers | White-Box | White-Box | Black-Box |
| Sicht des Software-Architects | Setzt zusammen | Setzt zusammen | Verwendet |
| Sicht des Component-Developers | Irrelevant | Irrelevant | Entwickelt Composite Components |
| Einzelteile auf getrennten Resource Containern / Maschinen deploybar? | Ja | Ja | Nein |
| Mehrere Instanzen auf derselben Resource Containern / Maschinen deploybar? | Ja | Ja | Ja |
| Darf Teil eines Systems (PCM) sein? | Nein | Ja | Ja |
| Hat Schnittstellen nach außen? | Ja | Ja | Ja |

Tabelle 2.1: Gegenüberstellung von System, SubSystem und Composite Component [34, p. 48f], [29], [28]

**Behavioral Viewpoint** Der *Behavioral Viewpoint* ist die zweite Betrachtungsebene des PCM. Er beschreibt das Verhalten der Komponenten, wie die Einzelkomponenten zusammenarbeiten und Szenarien zur Nutzung des Systems.
Die *Service Effect Specification (SEFF)* beschreibt das innere Verhalten von Einzelkomponenten. Hierbei betrachten die *SEFFs* die Interna einer Komponente auf einer eher hohen Abstraktionsebene. Insbesondere die Verbindungen zwischen den angebotenen Schnittstellen, sowie den benötigten Schnittstellen können mithilfe einer *SEFF* dargestellt werden [34, p. 53f.]. Ein Beispiel hierfür wäre, welche Anfragen an andere Komponenten aufgerufen werden, um einen angebotenen Dienst zu erfüllen.

Eine Spezialform der SEFF ist die *Resource Demanding SEFF*. Diese Form der *SEFF* verbessert die Möglichkeiten der Analyse der Performance und der Zuverlässigkeit von Systemen. Bei der *Resource Demanding SEFF* handelt es sich um eine Mischung aus *SEFF* und *Resouce Demanding Behavior* [27]. Der *Resource Demanding Behavior*-Teil beschreibt hierbei die *Resource Demands* der einzelnen Aktionen. *Resource Demands* beschreiben an dieser Stelle jedoch nicht die benötigten Ressourcen, wie zum Beispiel 1 Kern mit 2 GHz oder referenzieren ein Prozessor-Modell. Stattdessen werden auf einer abstrakteren Ebene *Resource Types* definiert, die benötigt werden, wie zum Beispiel CPU, Festplatte oder eine LAN-Verbindung. Diese *Resource Types* werden dann im *Deployment Viewpoint* vom *System Deployer* verwendet um sie sogenannten *Processing Resources* zuzuweisen. Diese *Processing Resources* gehören zu konkreten *Resource Environments*. Auf diesem Weg ist eine Entkopplung von Komponenten und den benötigten Ressourcen möglich. Insbesondere ist zum Zeitpunkt der Beschreibung des Verhaltens der Software mit *Resource Demanding SEFFs* oft noch nicht klar, auf welcher Hardware die Komponenten konkret laufen [34, p. 109].

Eine Komponente besitzt für jede angebotene Operation eine *Resource Demanding SEFF*. Im Kontext des Beispiels MediaStore, hat die *Basic Component* Media Management drei verschiedene *Resource Demanding SEFFs* für die drei Operationen, die durch das *Interface* IMediaManagement angeboten werden. Da die *Basic Component* MediaManagement eine *provides*-Verbindung zu diesem Interface hat, müssen diese drei Operationen angeboten werden und sind somit durch eine *Resource Demanding SEFF* genauer spezifiziert. Die drei Operationen lauten: `upload`, `download` und `getFileList`. Die *Resource Demanding SEFF* für die angebotene Operation `getFileList` ist in Abbildung 2.3 zu sehen. Zu Beginn der *SEFF* ist ein *Resource Demand* zu sehen. Dieser spezifiziert hier die benötigten Ressourcen und wird später zur genaueren Simulation verwendet.

**Usage Model:** Das *Usage Model* ist auch Teil des *Behavioral Viewpoints* [34, p. 56f.]. Es beschreibt die Interaktionen von Nutzern oder anderen Systemen mit dem modellierten System. Ein *Usage Model* besteht aus *Usage Scenarios*, sowie allgemeinen Nutzerdaten. Ein *Usage Scenario* enthält einen *Workload* und ein *Scenario Behavior* [30]. Hierbei beschreibt ein *Scenario Behavior* eine Abfolge von Anfragen an die angebotenen Dienste des Systems. Diese werden in einem *Scenario Behavior* angegeben, inklusive möglicher Parameter, welche auf Zufallsvariablen basieren können [26]. Der *Workload* als zweiter Teil eines

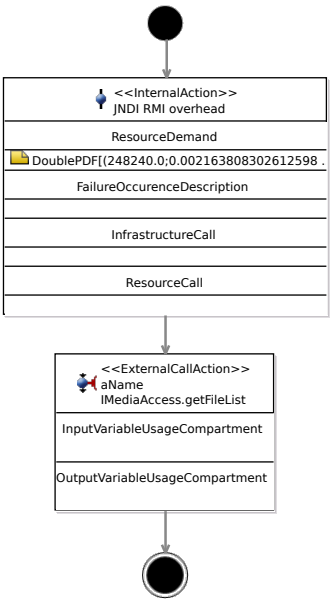


Abbildung 2.3: Resource Demanding SEFF der Komponente MediaManagement für die Operation getFileList

*Usage Scenarios* beschreibt die Intensität der Nutzung des Systems. Dies ist beispielsweise möglich durch die Angabe, wie viele Nutzer das System zeitgleich benutzen [31].

**Deployment Viewpoint** Nach der Beschreibung der Einzelkomponenten, der Orchestrierung zu einem fertigen System, sowie einer Beschreibung des Verhaltens der Komponenten und möglicher Nutzungsszenarien, fehlt noch der finale Schritt für Performance-Simulationen: die Allokation auf Hardware. Dieser Punkt wird im *Deployment Viewpoint* behandelt. Der *Deployment Viewpoint* wird durch den *System Deployer* beschrieben. Der *System Deployer* spezifiziert, welche Ausführungsumgebungen (*Resource Environments*) zur Verfügung stehen [34, p. 107ff.]. Später werden dann die Systeme, aufgeteilt in Komponenten, auf den verschiedenen *Resource Containern* allokiert.

Die Ausführungsumgebung wird durch das *Resource Environment Model* beschrieben. Dieses Modell beschreibt *Resource Container*, die physische sowie virtuelle Hardwareknoten beschreiben können. Das PCM unterscheidet jedoch an dieser Stelle nicht zwischen physischen oder virtuellen Hardwareknoten [34, p.57]. Ein Resource Container kann verschiedene *Processing Resources* besitzen mit einem *Resource Type*, wie z.B. CPU oder HDD. Die verschiedenen *Resource Container* können dann mithilfe einer *Linking Resource* verbunden werden. Diese kann durch Eigenschaften wie Latenz, Durchsatz oder Fehlerhäufigkeit beschrieben werden und so Einfluss auf die Simulation nehmen. Die Möglichkeit zur Beschreibung von Netzwerk Ressourcen und Latenzen ist insbesondere bei verteilten Systemen mit verschiedenen Knoten und viel Kommunikation zwischen den Knoten wichtig und kann einen großen Einfluss auf die Performance eines solchen Systems haben. Die Latenz einer *Linking Resource* kann aus Erfahrungswerten bereits laufender Systeme ge-

wonnen werden und beschreibt den round-trip-delay eines verschickten Netzwerk-Pakets. Jede Anfrage über diese *Linking Resource* benötigt dann um den Wert der Latenz länger, bis sie abgearbeitet ist [34, p. 109]. Die *Processing Ressources* können auch abgesehen von einer *Processing Rate*, mit beispielsweise der Anzahl an Replicas oder einer Scheduling-Strategie genauer spezifiziert werden [34, p. 107ff.].

Die *Resource Environment Models* sind im PCM wiederverwendbar und somit auch unabhängig von konkreten Systemen oder möglichen Allokationen von Komponenten. Diese Allokationen, also konkrete Zuweisungen der Bestandteile eines Systems, werden durch das *Allocation Model* beschrieben. Im *Allocation Model* werden Systembestandteile auf Hardwareknoten allokiert. Es ist wichtig zu beachten, wie bereits oben erwähnt, dass *Basic Components* sowie *Composite Components* immer als Ganzes allokiert werden müssen. Ein *System* beziehungsweise ein *SubSystem* kann auf verschiedenen Hardwareknoten verteilt sein. Die Hardwareknoten müssen über *Linking Resources*, die nicht im *Allocation Model* aufgeführt werden, verbunden werden. Dies ergibt sich unter anderem aus der weiter oben angesprochenen Black-Box Eigenschaft von *Composite Components* für den *System Deployer*.

### 2.2.3 Rollen im Palladio Component Model

Im Folgenden werden die Rollen des Palladio-Entwicklungsprozesses kurz vorgestellt. Hintergrund ist, die Verantwortlichkeiten der Rollen zu definieren um sie in den PCM / Kubernetes-Workflow, siehe Kapitel 4, einzubinden. Im Palladio-Entwicklungsprozess gibt es folgende fünf Rollen: den *Software Architect*, den *Component Developer*, den *Domain Expert*, den *System Deployer* und den *Quality Analyst*. In diesem Abschnitt werden nur die drei Rollen *Software Architect, Component Developer* und *System Deployer* beschrieben. Die anderen beiden Rollen sind für die vorliegenden Arbeit nicht relevant. Der *Software Architect* entwickelt den Entwurf der Software. Dieser Schritt passiert beispielsweise im *System Model*, wo Komponenten assembliert werden. Der *Component Developer* spezifiziert und implementiert Komponenten. Er kümmert sich auch um die Spezifikation von Performance und Zuverlässigkeit. Diese Angaben müssen jedoch oft auf abstrakter Ebene gemacht werden, da üblicherweise nicht bekannt ist, auf welchen Maschinen die Software deployed wird. Der *System Deployer* verwendet die Spezifikationen der Komponenten um Ressourcen Umgebungen zu definieren. Außerdem allokiert er die im *System Model* assemblierten Komponenteninstanzen auf *Resource Container* der *Resource Environment* [34, p. 203ff].

# 3 Konzeption Modellerweiterung

Dieses Kapitel beschäftigt sich mit konkreten Konzepten und Ansätzen zur Abbildung und Simulation von containerisierten Anwendungen und Kubernetes in Palladio. Ziel des Kapitels ist es, die Unterschiede auf der Modellebene auf Basis der Erklärungen des Grundlagen-Kapitels herauszuarbeiten, sowie die gewählte Entwurfsentscheidung zu dokumentieren. Hierbei wird weniger stark auf die Erklärung der Palladio- oder Kubernetes-Konzepte eingegangen, sondern Eigenschaften herausgegriffen, welche eine Abbildung ermöglichen bzw. erschweren, oder nicht gut abgebildet werden können. Zu Beginn des Kapitels werden die wichtigsten Konzepte aus Kubernetes als Containerorchestrierungswerkzeug aufgezählt, für welche eine Abbildung im PCM erarbeitet wurde. Danach wird die Erweiterung des Metamodells, aufgeteilt auf die drei *Viewpoints Structural, Behavioral* und *Deployment*, beschrieben. Dabei werden die gewählten Entwurfsentscheidungen diskutiert und begründet sowie beschrieben, warum dieses Konzept für die Abbildung und Simulation notwendig ist. Abschließend wird eine Gesamtübersicht über die Modellerweiterung gegeben.

## 3.1 Modell-Konzept

Diese Arbeit erweitert das PCM um die Möglichkeit, Systeme, welche mit Containerorchestrierungswerkzeugen wie Kubernetes deployed werden abzubilden und zu simulieren. Die abgebildeten Konzepte sind: Container, Pods, Cluster, Knoten, Deployments, Requests und Limits und das Service-Konzept. Dazu kommt das Scheduling von Pods auf Knoten auf Basis der Requests und Limits. Die Erweiterungen erstrecken sich über die *Viewpoints Structural*, *Behavioral* sowie *Deployment*. Der *Deployment Viewpoint* spielt hierbei eine besondere Rolle, da mit Containerorchestrierungswerkzeugen wie Kubernetes, Software in einer anderen Form deployed wird, als bei einem statischen Deployment auf Virtual Machine (VM)s, wie es bisher durch das PCM unterstützt wurde.

Die Modellerweiterungen und Veränderungen sind in die drei Ebenen bzw. *Viewpoints* von Palladio aufgeteilt. Gewisse Konzepte aus Kubernetes werden in mehreren *Viewpoints* modelliert. Das bedeutet, die Abbildung gewisser Konzepte beschränkt sich nicht auf Änderungen in nur einem *Viewpoint*, sondern erfordert Änderungen auf mehreren Ebenen.

Das Metamodell als UML-Diagramm der Erweiterung des PCM ist in Abbildung 3.6 zu sehen. Es wird im Verlauf des Kapitels genau beschrieben und soll hier bereits einen Überblick über die Modelle liefern.

### 3.1.1 Modellverwendung und -veränderungen / Erweiterungen auf Ebene des Structural Viewpoint

Der *Structural Viewpoint* umfasst das *Repository Model* sowie das *System Model*. Mithilfe dieser Modelle im *Structural Viewpoint* werden folgende Konzepte vollständig oder teilweise modelliert: Container, Pods, Services sowie Requests und Limits.

**Container** Der Hauptbestandteil der vorliegenden Arbeit ist, containerisierte Anwendungen und Containerorchestrierungswerkzeuge wie Kubernetes in Palladio abzubilden und zu simulieren. Da Container das Grundkonzept für diese Fragen darstellen, ist die Abbildung dieser im PCM für diese Arbeit grundlegend wichtig.
Im PCM werden zum aktuellen Zeitpunkt Softwarekomponenten, sog. *Basic Components* als kleinster Bestandteil einer Applikation verwendet. Mehrere *Basic Components* können dann zu einem ganzen *System*, einem *SubSystem* oder einer *Composite Component* aggregiert werden.
Eine containerisierte Anwendung besteht, im Vergleich dazu, aus einem oder mehreren Containern, die Anwendungscode enthalten. Hierbei kann ein Container bereits die gesamte Anwendung enthalten oder die Anwendung kann auf mehrere Container verteilt sein.
Das *Repository Model* beschreibt grundlegend alle *Basic Components*, *SubSystems* und *Composite Components*, die es für eine Anwendung gibt oder geben soll, mit ihren Schnittstellen. Diese werden dann im *Assembly Viewtype* mithilfe des *System Models* zusammengesetzt, um das *System* dann zu deployen. Im Allgemeinen stellt ein Container eher eine Möglichkeit dar Anwendungen zu deployen bzw. ist als Deployment-Konzept bekannt. Im Palladio-Ansatz würde man dieses Konzept also in dem *Deployment Viewtype* sehen. Der *Deployment Viewtype* beinhaltet aber nur das *Resource Environment Model*, sowie das *Allocation Model*. Betrachtet man den Container als reines Deployment Konzpet, losgelöst von der Verwendung von Kubernetes mit Requests und Limits sowie Services, liegt es nahe einen Container als *Nested Resource Container* darzustellen. Im Folgenden werden *Resource Container* oder *Nested Resource Container* des *Resource Environment Model* nie mit nur Container abgekürzt. Bei der Verwendung des Wortes Container handelt es sich also immer um das Konzept des Containers, das hier abgebildet werden soll. Dabei könnten alle zu einem Container gehörenden Software-Komponenten, im PCM also *Basic Components* oder *Composite Compontents*, die im *System Model* assembliert wurden, auf diesem *Nested Resource Container*, der dann den im Cluster laufenden Container darstellt, allokiert werden. Dieser Ansatz stellt auf jeden Fall eine einfache Möglichkeit dar, Container allein mithilfe des PCM abzubilden. Für diese Arbeit geht es aber nicht allein um die Abbildung von Containern als Deployment-Konzept, sondern um die Modellierung und Simulation von Container-basierten Software-Architekturen. Das bedeutet, dass der Container als Konzept bereits zur Entwurfszeit berücksichtigt werden kann und sollte, da für eine Container-basierte Software-Architektur Container als abgeschlossene gekapselte Einheit verstanden werden. Diese bieten dann einen Dienst an und das Gesamtsystem wird aus einem oder mehreren Containern zusammengesetzt. Der Container wird als wiederverwendbare Komponente beschrieben. Unter anderem aus diesem Grund wurde

der Container bereits in das *Repository Model* aufgenommen. Ein weiterer Grund dafür ist, dass der Container, welcher mehrere *Basic Components* oder auch *Composite Components* kapselt, dann im System-Entwurf wiederverwendet werden kann.
Eine Analogie zum Rollensystem von Palladio wäre, dass der *Component Developer* jetzt ganze Container entwickelt, welche vom *Software Architect* definiert wurden. Der *Software Architect* kann diese dann im *System Model* zum Gesamtsystem zusammen setzen und zum Beispiel definieren, wie viele Instanzen eines Containers benötigt werden. Diese Tatsache erleichtert auch die Verwendung von Containern zur Definition containerisierter Systeme/Architekturen im PCM im Vergleich zur oben angesprochenen Allokation der Einzelkomponenten auf einen Container erst zum Deployment-Zeitpunkt. Der *System Deployer* allokiert dann die definierten Container auf der Ausführungsumgebung. Dabei wird pro Container ein *Nested Resource Container* erstellt, der den Container zur Laufzeit enthält.
Es stellt sich nun die Frage, wie man einen Container im *Repository Model* entwirft. Die Anforderungen eines Containers für das PCM im Rahmen dieser Arbeit wurden mithilfe einer Integritätsregel [36] folgendermaßen definiert:

**IR 10:** Ein Container muss durch
keine oder beliebig viele Basic Components
UND keine oder beliebig viele Composite Components
UND ein oder kein Standard Limit
UND ein oder kein Standard Request
UND keine oder beliebig viele zusätzliche Limits
UND keine oder beliebig viele zusätzliche Requests
beschrieben werden.

Diese Modellierung ist in Abbildung 3.6 auf der linken Seite im Paket `kubernetesModel.repository` als UML-Diagramm zu sehen.
Die ersten beiden Eigenschaften des Containers implizieren, dass es sich um eine zusammengesetzte Komponente handeln muss. Im PCM stehen zwei Konzepte hierfür zur Verfügung: Einmal das *SubSystem* und die *Composite Component*. Der Unterschied wurde in Tabelle 2.1 ausgeführt. Für den Container wurde hierbei die *Composite Component* gewählt, welche eine Allokation der inneren Komponenten auf verschiedenen *Resource Containern* verbietet. Gleichzeitig entspricht die Black-Box-Eigenschaft der *Composite Component* zur Deployment- und Allokationszeit der Sicht des *System Deployers* auf Container. Eine Allokation der inneren Komponenten eines Containers auf verschiedenen Hardwareknoten wiederspricht den Grundeigenschaften des Containers und ist deshalb nicht sinnvoll.
Die Eigenschaften, dass es Limits und Requests geben kann, wurde gewählt um dynamische Container-basierte Systeme abbilden und simulieren zu können. Die Standard Requests und Limits sind eine Referenz auf jeweils K8sStandardRequestLimit-Objekte. Diese beschreiben einen Memory- und CPU-Wert, wie in Kubernetes üblich. Ein Container kann somit Speicher- sowie CPU-Anteile anfordern bzw. durch diese beschränkt werden. Die Möglichkeit diesen Sachverhalt abzubilden spielt eine wichtige Rolle beim Allokationsscheduling, siehe Kapitel 5. Da in Kubernetes das Scheduling nicht nur für Speicher und CPU-Anteile, sondern auch durch weitere Eigenschaften eines Hardware-

knotens beeinflusst werden kann (bei Wahl eines entsprechenden Schedulers), wurde als weiteres Attribut eine Sammlung an AbstractK8sRequestLimit gewählt. Eine mögliche Spezialisierung davon, die auch im UML-Diagramm bereits vorgesehen wurde, ist das Setzen einer einfachen Eigenschaft mithilfe eines K8sAdditionalRequestLimit. Diese können dann das Allokationsscoring beeinflussen, sodass bei entsprechender Spezifikation zum Beispiel Knoten mit einer SSD bevorzugt werden.
Die Verwendung des Containers in den weiteren *Viewpoints* ist einmal die Assemblierung mithilfe von *Assembly Contexts* im *System Model*, sowie die spätere Allokation der Container auf *Resource Containern*. Für die Verwendung des Containers in dynamischen Systemen mithilfe von Kubernetes, wird der Container in Pods assembliert. Die Pods werden dann zum Gesamtsystem zusammengebaut und allokiert. Um den *Nested-Resource-Container-Scheduler* verwenden zu können, müssen die Container einzeln auf *Nested Resource Containern* allokiert werden. Um das zu ermöglichen, wird für jeden Container der in einem Pod verwendet wird, ein *Nested Resource Container* erstellt, siehe 36.

**Pod** Der Pod wird definiert, um dynamische Systeme auf Basis von Kubernetes abbildbar und simulierbar zu machen. Ein wichtiger Punkt dieser Arbeit ist der Pod Allocation Scheduler, da in einem dynamischen System nicht bekannt ist, wo welcher Pod laufen wird. Dies macht die Umsetzung des Pods in dieser Arbeit unabdingbar. Die Argumente, warum der Pod in das *Repository Model* gehört und nicht ausschließlich in den *Deployment Viewpoint* sind die gleichen wie für den Container, siehe Seite 22, weshalb an dieser Stelle diese Punkte nicht erneut aufgeführt werden.
Die Anforderungen für den Pod wurden folgendermaßen definiert:

**IR 20** Ein Pod muss durch
keinen oder mehrere Container
beschrieben werden.

Ein Pod ist einem Container konzeptionell sehr ähnlich, mit dem Unterschied, dass statt Software-Komponenten nun ganze Container in einem Pod gekapselt und gemeinsam deployed werden.
Die Anforderungen an den Pod sind eher einfach. Grund hierfür ist, dass einige Eigenschaften eines Pods durch Container oder Deployments teilweise implizit umgesetzt werden. Ein Beispiel hierfür ist eine Identifikation des Pods. Diese wird durch das `podReference`-Attribut des Deployments, siehe Seite 35, umgesetzt, damit ersichtlich werden kann, welcher Pod zu welchem Deployment gehört. Die Requests und Limits werden in Kubernetes durch Container definiert, weshalb ein Pod auch in diesem Modell die Requests und Limits implizit durch die Summe der Requests und Limits der assemblierten Container auf diesem Pod hat.
Die einzige Anforderung an den Pod ist also, dass er keinen bis beliebig viele Container enthalten können muss. Es liegt also nahe erneut eine zusammengesetzte Struktur wie die *Composite Component* oder das *SubSystem* als Eltern-Klasse zu verwenden. Für den Pod wurde, im Gegensatz zum Container, das *SubSystem* verwendet. Grund hierfür ist,

dass wie bereits auf Seite 22 erwähnt, jeder Container in einem Pod auf einem *Nested Resource Container* allokiert wird, bzw. auch für jeden auf einem allokierten Pod enthaltenen Container immer ein *Nested Resource Container* auf dem Knoten (Node) des Pods exisiteren muss. Diese Besonderheit steht im Zusammenhang mit der Verwendung des bereits implementierten Nested-Resource-Container-Schedulers. Die Allokation der im Pod enthaltenen Container geschieht also auf unterschiedlichen *Resource Containern*. Mit einer *Composite Component* ist die Allokation der inneren Komponenten (hier Container) auf verschiedenen *Resource Containern* nicht vorgesehen bzw. ein entscheidender Unterschied zwischen *SubSystem* und *Composite Component*. Da das aber möglich sein muss, wird der Pod als Spezialisierung des *SubSystems* dargestellt.
Eine wichtige Einschränkung muss bei der Verwendung des Modells aber eingehalten werden: Die Container dürfen ausschließlich auf den extra dafür im *Resource Environment Model* erstellten *Nested Resource Containern* des Knotens (Node), auf welchem der Pod allokiert ist, allokiert werden. Eine Allokation auf verschiedenen *Resource Containern* / Kubernetes Nodes würde die Grundidee des Pods verletzen und wäre nicht mit dem in dieser Arbeit erstellten Modell kompatibel. Man kann sagen, dass der Pod, modelliert als Spezialisierung des *SubSystems* die Eigenschaften eines *SubSystems* verletzt. Insbesondere die Zusicherung, dass eine Allokation der Einzelteile auf verschiedenen *Resouce Containern* erlaubt ist. Eine Alternative zum gewählten Ansatz wäre es also auf der gleichen Ebene, auf welcher die *Composite Component* sowie das *SubSystem* definiert wurden, ein drittes zusammengesetztes Konstrukt Pod zu erstellen. Das würde dieses Problem lösen. Für die Umsetzung in dieser Arbeit wurde sich jedoch für die Spezialisierung des *SubSystems* entschieden, da das die Integration in das PCM mithilfe von Child-Extendern vereinfacht.

**Service** Services in Kubernetes agieren als Schnittstelle zwischen Pods. Das bedeutet, wenn ein Pod einen Service benötigt beziehungsweise anfragt, macht er das typischerweise nicht direkt an den Pod, da sich dessen IP-Adresse und Allokation regelmäßig ändern kann, sondern schickt seine Anfrage an einen Service. Dieser wählt dann einen Pod aus, der diese Anfrage bearbeiten soll. Die konkreten Anforderungen an den Service bzw. die Funktionalität die abgebildet werden soll ist also folgende:

FA10: Ein Service muss Anfragen eines definierten Typs anbieten.
FA20: Ein Service muss Anfragen eines definierten Typs an Pods weiterleiten können.
FA30: Ein Service muss eine Menge an Pods besitzen, die die Schnittstelle, die der Service anbietet, ebenfalls anbieten.
FA40: Ein Service muss neue Pods registrieren können.
FA50: Ein Service muss eine Auswahl treffen können, welcher registrierte Pod die Anfrage erhält.
FA60: Ein Service muss Pods, an die Anfragen weitergeleitet werden sollen, entfernen können.

Die Anforderung FA40 beschreibt, dass ein Service Pods registrieren können muss. Gemeint ist damit, dass in Kubernetes ein Service die Service-Discorvery übernimmt. Als

Ergebnis daraus, „weiß“ der Service sozusagen, welche Pods diesen Service anbieten um damit später Anfragen auf diese Pods zu verteilen. Für die Abbildung in das PCM wurde aber die Formulierung „registrieren“ gewählt, da ein Modell zur Laufzeit immer einen Zustand abbildet und dieser fest ist, ein abgebildeter Service also ohne Service-Discovery weiß, an welche Pods Anfragen weitergeleitet werden dürfen.

Es wurde entschieden, dass es nicht notwendig ist, für einen Service ein neues Metamodell-Objekt anzulegen. Stattdessen können bereits existierende Funktionalitäten des PCM verwendet werden um die obigen Anforderungen zu erreichen. Um alle Anforderungen erfüllen zu können, erfordert es jedoch mehrere Schritte um dieses Konzept abzubilden. Die Schritte sind in Abbildung 3.2 zu sehen und werden im Folgenden erklärt.

Zuerst muss ein *Interface* definiert werden, welches die Operationen des Services darstellt und entsprechende Methoden anbietet. Als zweiter Schritt wird eine *BasicComponent* erstellt, welche die Service-Komponente abbildet. Diese Service-Komponente muss das oben erstellte *Interface* anbieten und benötigen. Da der Service Anfragen an Pods entgegennimmt und weiterdelegiert ist das gleichzeitige Anbieten des *Interfaces*/Services nötig, damit andere Pods Anfragen an die Service-Komponenten senden können. An die *Required Roles* der Service-Komponente werden später die Pods gehängt, an welche die Anfragen delegiert werden sollen. Deshalb muss die Service-Komponente das *Interface* sowohl anbieten also auch benötigen. Danach können *Provided-* und *Required Roles* zur Komponente hinzugefügt werden. Es ist wichtig zu beachten, dass bei einer Änderung des Clusterzustands zur Laufzeit/Simulationszeit, welche Pods betrifft, die den Service anbieten, sich auch die Service-Komponente ändert. Diese Änderungen betreffen die interne *SEFF* sowie unter anderem die Anzahl der *Provided-* und *Required Roles* der Service-Komponente. Alle Änderungen werden in Kapitel 6 beschrieben. Aus diesem Grund sind die konkrete Anzahl bspw. der *Required Roles* Cluster-Zustands-abhängig. Nachdem die *Required Roles* erstellt wurden, kann die *SEFF* der Service-Komponente erstellt werden. Ein Beispiel für eine Load-Balancing-*SEFF*, welche eingehende Anfragen gleichmäßig auf in diesem Fall zwei registrierte Pods verteilt, die diesen Service anbieten ist in Abbildung 3.3 gegeben. Wichtig ist, dass pro registriertem Pod eine *Probabilistic Branch Transition* existiert. Diese hat dann die Wahrscheinlichkeit

$$\frac{1}{AnzahlDerRegistriertenPods}$$

bei einer gleichmäßigen Verteilung. Diese *Probabilistic Branch Transition* referenziert dann in ihrer inneren *External Call Action* die *Required Role*, an der der Pod hängt, der ausgewählt wurde. Im Beispiel ist in der *External Call Action* „delegateToPod1“ also die *Required Role* ausgewählt, an welcher im *System Model* auch die Instanz Pod1 hängt. Nachdem die *SEFFs* des Services, welche auch das Loadbalancing definiert erstellt wurde, ist die Service-Komponente im Repository Model vollständig definiert. Es ist wichtig hervorzuheben, dass pro Operation, die das *Interface*, welches der Service anbietet eine *SEFF* definiert wird. Grundsätzlich ist ein Service nicht darauf eingeschränkt nur ein *Interface* anzubieten. Das bedeutet, sollte es notwendig sein, können die angebotenen Operationen eines Services auch auf mehrere *Interfaces* verteilt werden. Im nächsten Schritt wird der Service dann

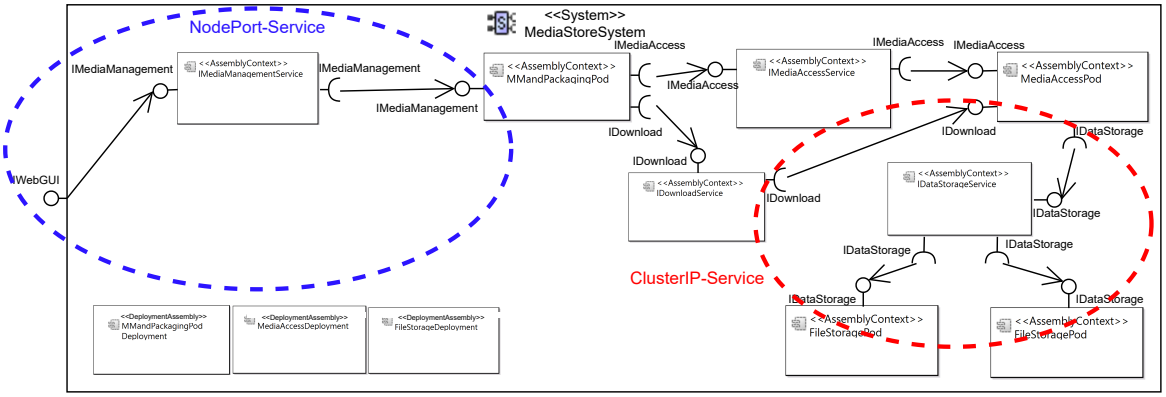


Abbildung 3.1: System Model mit Abbildung eines NodePort-Services, sowie eines ClusterIP-Services

im *System Model* instanziiert. Danach werden alle *Required-* und *Provided Roles* mit den entsprechenden Instanzen verbunden. Konkret wird also ein *Assembly Connector* erstellt, der die *Provided Role* des Pod1, der den Service anbietet, mit der zugehörigen *Required Role* des Services verbindet. Das wird mit allen Pods gemacht, die den Service anbieten. Für alle Pods die den Service benötigen wird ein *Assembly Connector* erstellt, der die *Required Role* des Pods mit der *Provided Role* des Services verbindet. Als letzter Schritt muss die Service Instanz allokiert werden, da es sich um eine einfache Komponente handelt. Die Allokation muss also auf einem *Resource Container* stattfinden. Eine Option hierfür wäre die Master-Node eines Clusters. Jenachdem ob gewünscht ist, dass das Loadbalancing des Services auch Ressourcen benötigt, können *Resource Demands* hinzugefügt werden.

Die obige Beschreibung zur Umsetzung eines Services beschreibt den Standardfall des ClusterIP Services, siehe Unterabschnitt 2.1.12. Es gibt noch drei weitere Service-Arten, von welchen mit obiger Beschreibung zwei ebenso umgesetzt werden können mit kleinen Veränderungen. Für den Headless Service, welcher bspw. bei zustandsabhängigen Systemen oder Teilsystemen wichtig ist, muss garantiert sein, dass alle Anfragen immer an denselben Pod gehen. Das ist bspw. möglich, indem entweder eine direkte Verbindung vom anfragenden Pod zum anbietenden Pod mithilfe eines *Assembly Connectors* erstellt wird oder ein Service-Objekt erstellt wird, welches nur genau einen Pod hat und dadurch alle Anfragen an diesen Pod delegiert. Die zweite Variante stellt jedoch eher die tatsächliche Umsetzung in Kubernetes dar, da trotz eines Headless Services immer noch ein Service erstellt wird. Die dritte umsetzbare Service-Art ist der NodePort-Service. Dieser ermöglicht es Anfragen an Pods von außerhalb des Clusters zu senden. Um diesen Service ebenso umzusetzen muss nur ein Service-Objekt wie oben beschrieben erstellt werden und statt dass nur interne *Assembly Contexts* die Pods instanziieren mit der *Provided Role* des Services verbunden werden, kann ein *Delegation Connector* von der *System Provided Role* zur *Provided Role* des Services verwendet werden. Die Struktur des *System Models* bei der

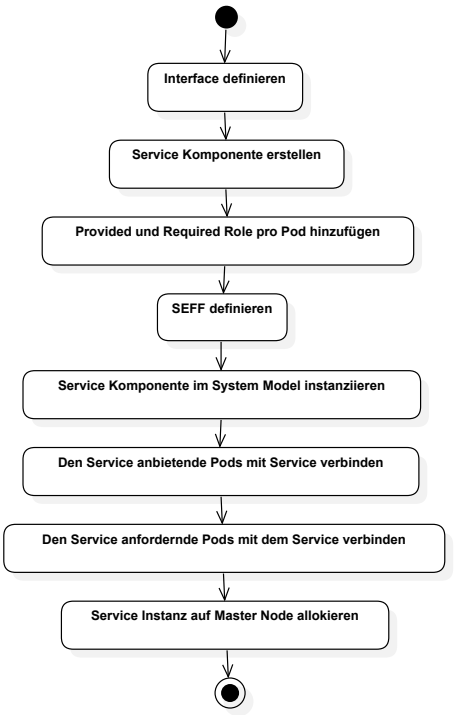


Abbildung 3.2: Aktivitätsdiagramm zur Modellierung eines Services im PCM

Umsetzung eines NodePort-Services (eingekreist in blau) und eines ClusterIP-Services (eingekreist in rot) kann in Abbildung 3.1 verglichen werden. Der letzte Service LoadBalancer stellt für das PCM keinen konzeptuellen Unterschied zum NodePort-Service dar. Deshalb ist dieser grundsätzlich auch abbildbar. Der eigentlich vorgeschaltete Loadbalancer des Cloud-Providers, kann in der Service-Komponenten im System/Cluster integriert sein.

**Ingress** Die Abbildung eines Ingress mithilfe des PCM ähnelt stark der Abbildung eines NodePort-Services. Dabei wird, statt dass die *System Provided Role* direkt mit dem NodePort-Service verbunden wird, noch eine Ingress-Komponente zwischengeschaltet. Diese wird im *Repository Model* definiert und verteilt die eingehenden Anfragen auf die Service-Komponenten.

**Requests und Limits** Requests und Limits, siehe Unterabschnitt 2.1.6 und Unterabschnitt 2.1.7, beschreiben, wie viele Ressourcen (zum Beispiel CPU und Speicher) für einen Container zur Verfügung stehen. Mit Requests wird angegeben, wie viel Ressourcen einem Container garantiert werden, sozusagen eine untere Grenze für Ressourcen. Ein Beispiel hierfür ist, dass ein Container nur auf einer Node allokiert wird, die noch genügend nicht-reservierte Ressourcen zur Verfügung hat. Limits definieren eine harte obere Grenze für Ressourcen, die der Container nutzen darf. Ein Szenario hierfür ist, dass, wenn ein Container mehr CPU benutzt, als im Limit spezifiziert, dieser gedrosselt wird. Bei zu viel genutztem Speicher wird der Container automatisch terminiert. Die Angabe von Requests und Limits ist nicht auf CPU und Speicher limitiert. Es kann ebenso eine

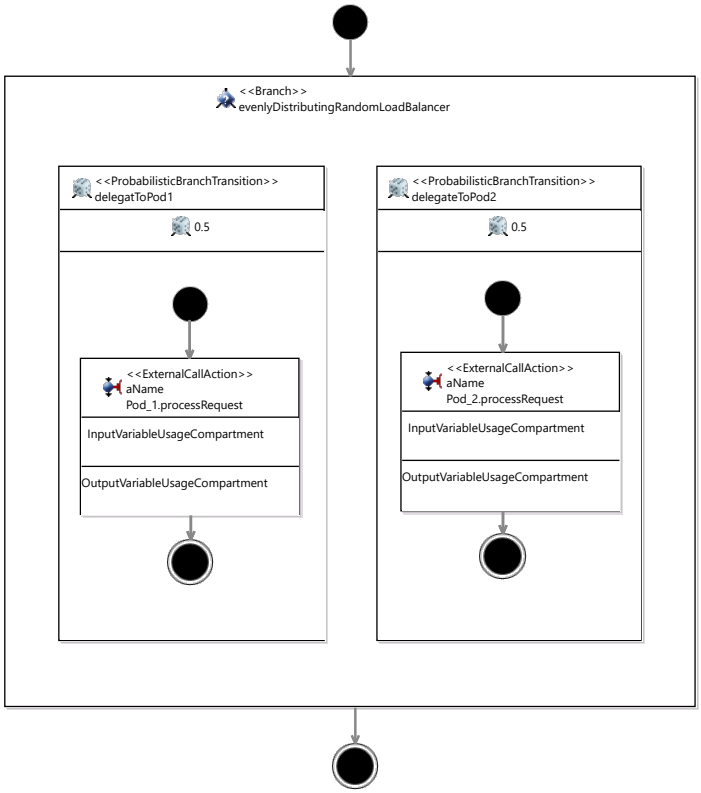


Abbildung 3.3: Gleichmäßig zufällig verteilende Load-Balancing SEFF eines Services mit 2 Pods

beliebige Ressource, wie beispielsweise das Vorhandensein einer SSD als Request definiert werden. Damit haben Requests und Limits einen relevanten Einfluss auf die Performance einer Software. Gleichzeitig spielen Requests eine große Rolle bei der Allokation von Pods auf Nodes. Diese beiden Punkte, sowie die Forschungsfrage, welchen Einfluss Requests und Limits auf die Performance einer containerisierten Anwendung / Pods in Kubernetes haben, motivieren die Abbildung dieser Konzepte im PCM.

Die Anforderungen an Requests und Limits im Rahmen dieser Arbeit wurden folgendermaßen definiert:

FA70: Das PCM muss dem Nutzer die Möglichkeit bieten, eine CPU-Request für einen Container zu definieren.
FA80: Das PCM muss dem Nutzer die Möglichkeit bieten, eine Speicher-Request für einen Container zu definieren.
FA90: Das PCM muss dem Nutzer die Möglichkeit bieten, beliebige Request-Attribute für einen Container textuell zu definieren.
FA100: Das PCM muss dem Nutzer die Möglichkeit bieten, ein CPU-Limit für einen Container zu definieren.
FA110: Das PCM muss dem Nutzer die Möglichkeit bieten, ein Speicher-Limit für einen Container zu definieren.
FA120: Das PCM muss dem Nutzer die Möglichkeit bieten, beliebige Limit-Attribute für einen Container textuell zu definieren.

Ein Request beziehungsweise ein Limit wird immer für einen Container spezifiziert, nicht für einen Pod. Die Requests bzw. Limits eines Pods können aus den inneren Containern abgeleitet werden. Diese Verbindung zwischen Pods und darin enthaltenen Containern ist für den Allokations-Scheduler von Pods relevant, siehe Kapitel 5. Die Anforderungen FA70, FA80, FA100 und FA110 wurden explizit ausgewiesen, da eine Speicher- und CPU-Spezifikation Best-Practice bei der Definition eines Cluster-Zustands ist. Deshalb ist anzunehmen, dass diese beiden Angaben in den meisten Fällen gemacht werden und eine explizite Ausweisung dieser Attribute die Benutzbarkeit der PCM-Erweiterung verbessert. Zusätzlich wurde in der vorliegenden Arbeit ein Allokations-Scheduler für Pods entwickelt, der auf den CPU- und Speicher-Requests der Pods basiert. Dieser Scheduler bildet den Standardfall in Kubernetes ab, sollten keine zusätzlichen Ressourcentypen definiert worden sein.
Um Container mit Requests und Limits zu versehen, wurden mehrere Ansätze in Betracht gezogen:
1. Es können einfache Integer- / Long-Attribute in der Container-Definition verwendet werden, um die CPU- und Speicher-Requests und Limits abzubilden. Hierbei müsste, um die Anforderungen FA90 und FA120 umzusetzen, ein weiteres mehrwertiges Attribut erstellt werden, welches entweder den Typ String zur textuellen Beschreibung hat, oder es müsste ein mehrwertiges Attribut von einem RequestLimit-Objekt hinzugefügt werden. Dieses RequestLimit-Objekt müsste dann ein Feld für eine textuelle Beschreibung besitzen.
2. Es werden dedizierte RequestLimit-Objekte verwendet. Diese haben, je nach Typ des

Requests beziehungsweise Limits, Felder vom Typ Integer / Long für CPU oder Speicher, und vom Typ String für die textuelle Beschreibung weiterer Requests und Limits. Ein Container hat dann eine mehrwertige Referenz zu einer Elternklasse aller RequestLimit-Typen.
3. Es werden wie im 2. Ansatz spezielle Request- und Limit-Klassen erstellt, mit einem Super-Typ für eine spätere Gleichbehandlung. Ein Container hat insgesamt vier Referenzen, die die Requests und Limits für diesen Container darstellen. Zuerst eine `standardRequest`, dann ein `standardLimit` und zwei, mehrwertige Attribute `additionalRequests`, sowie `additionalLimits` vom Typ der Oberklasse `AbstractK8sRequestLimit`. Die Standardattribute beschreiben nur ein CPU- bzw. Memory-Request/Limit und sind vom Typ `K8sStandardRequestLimit`. Dabei enthält der Typ `K8sStandardRequestLimit` ein Integer-Attribut für die Angabe von CPU in milicores und ein Long-Attribut für die Angabe der Speicher-Spezifikation in kB. Die `K8sAdditionalRequestLimit` Klasse enthält nur ein Attribut vom Typ String.
Für die Abbildung der Requests und Limits von Containern wurde Variante 3 gewählt, welche auch im UML-Diagramm, siehe Abbildung 3.6 zu sehen ist.
Der Grund, warum es nicht den Typ Request und den Typ Limit getrennt gibt ist, dass sich Requests und Limits bei der Spezifikation nicht unterscheiden, solange klar ist, dass ein RequestLimit-Objekt als Request oder als Limit verwendet werden soll. Diese Unterscheidung wird für das standardRequest beziehungsweise das standardLimit bei der Auswahl beziehungsweise Spezifikation des entsprechenden RequestLimit-Objektes getroffen. Die zusätzlichen Requests bzw. Limits wurden hinzugefügt, um die Erweiterbarkeit der Requests und Limits um neue Ressourcentypen zu ermöglichen, wie beispielsweise eine SSD. Hierbei wird die Unterscheidung zwischen Request oder Limit ebenso durch zwei mehrwertige Attribute umgesetzt. Einmal für zusätzliche Requests und einmal für zusätzliche Limits. Die Definition zusätzlicher Requests und Limits wurde bewusst offen für Erweiterungen gewählt, ohne große Einschränkungen zu machen, damit bei Bedarf eine einfache, den dann vorliegenden Anforderungen entsprechende Erweiterung möglich ist. Zum jetzigen Zeitpunkt berücksichtigt der Allokations-Scheduler die zusätzlichen Requests und Limits nicht, ist aber so entworfen, dass eine einfache Erweiterung möglich wäre.
Ansatz 1 wurde nicht gewählt, da dieser Ansatz wenig Spielraum für Erweiterungen offen lässt. Ansatz 2 wurde nicht gewählt, da die üblicherweise benutzten standardRequest und standardLimit-Felder nicht explizit ausgewiesen wären, was die Benutzbarkeit einschränken kann. Zusätzlich vereinfacht dieses explizite Definieren der standardRequest-/Limit-Felder die Implementierung des Allokations-Schedulers. Wie in Abbildung 3.6 zu sehen ist, sind die Request- beziehungsweise Limit-Attribute alle optional. Es ist also ebenso keine Definition von Requests und Limits möglich.

Die Referenzen eines Containers zu Requests und Limits wurden als *containement* dargestellt, siehe Abbildung 3.6. Das bedeutet sie sind in einem Container enthalten und werden speziell für einen Container erstellt und sind somit in der Baumstruktur eines Modells Kinder eines Containers. Ein weiterer wichtiger Punkt ist die oben erwähnte Einschränkung der Einheiten bei der Angabe der standardRequests bzw. standardLimits. Es wurde entschieden, für den CPU-Wert den Datentyp Integer zu verwenden und nur eine Spezifikation in millicores zuzulassen. Die Angabe in dieser Einheit wird bei der Spe-

zifikation von Requests und Limits oft verwendet. Diese Einschränkung entfernt jedoch die Möglichkeit, den Sonderfall umzusetzen, in dem ein Container dedizierte Kerne zugewiesen bekommt, wie in Unterabschnitt 2.1.9 beschrieben. Für Speicherangaben wurde oben erwähnt, dass diese in kB erfolgen sollen. Grundsätzlich ist diese Einschränkung aber nicht zwingend notwendig. Die Einheit bei der Spezifikation der Kubernetes-Nodes, siehe Seite 34, muss nur mit der Angabe bei den Request- und Limit-Objekten übereinstimmen, wobei die Einheit nicht explizit angegeben wird.
Es existiert bereits eine prototypische Implementierung eines Nested-Resource-Container-Schedulers, siehe [10]. Dieser setzt bereits CPU-Requests bei der Zuteilung des CPU-Shares um. Hier muss erwähnt werden, dass die Angaben in Kubernetes bei millicores zur Basis 1000 gemacht werden. Der Scheduler verwendet die *processing rate* zur Basis 1024. Diese Umrechnung muss bei der Verwendung dieser Attribute durch den Scheduler berücksichtigt werden, ist aber sonst kompatibel.
Mit der in diesem Abschnitt beschriebenen Umsetzung wurden die spezifizierten Anforderungen umgesetzt.

### 3.1.2 Modellverwendung und -veränderungen / Erweiterungen auf Ebene des Behavioral Viewpoint

Der *Behavioral Viewpoint* enthält das *Usage Model* sowie die *SEFF* der Komponenten. Für diese Modelle ist keine Erweiterung des PCM vorgesehen. Dennoch werden in diesen Modellen, je nach Anwendungsfall, Spezifikationen vorgenommen, um die Abbildung von dynamischen, containerisierten Anwendungen zu verbessern. Möglichkeiten zur Abbildung des Kubernetes-spezifischen Overheads wären in diesem Viewpoint denkbar. Eine Notwendigkeit hierfür besteht, da bei der Verwendung von Kubernetes, unter Umständen, Pods oft terminiert und neu gestartet werden. Diese Häufigkeit könnte aus Erfahrungswerten abgeschätzt werden. Im nächsten Schritt kann dann eine dummy-Komponente im *Repository Model* erstellt werden, zusammen mit einem dummy-Interface. Dieses *Interface* beinhaltet beispielsweise die Operation `startUp()`. Für diese Operation kann nun eine *Resource Demanding SEFF* definiert werden, die den Overhead beim Starten eines Pods grob abschätzt. Die Häufigkeit der Aufrufe dieser Operation kann dann im *Usage Model* definiert werden. Die dummy-Komponente kann entweder einmalig für das Cluster instanziiert werden oder pro Kubernetes Knoten. Hierbei müsste dann das *Usage Model* entsprechend angepasst werden. Ein Kritikpunkt an diesem Ansatz ist, dass jegliche Abbildung des Overheads nur auf geschätzten oder in einem tatsächlichen Cluster gemessenen Häufigkeiten basiert. Eine höhere Genauigkeit würde erreicht werden, wenn dieser Overhead nur genau dann getriggert wird, wenn auch tatsächlich ein neuer Pod hochgefahren wird. Eine Umsetzung dieser Idee ist nur möglich, wenn es dynamische Veränderungen im Cluster zur Simulationszeit gibt, also tatsächlich Pods hinzukommen oder verschwinden. Um einen derartigen Overhead abzubilden, bieten sich *Actions* an, da diese an tatsächliche Veränderungen im System gekoppelt sind, siehe [37, p. 143ff]. Mithilfe von *Actions* lassen sich nicht nur Rekonfigurationen von Systemen umsetzen, sondern sie erlauben auch eine Abbildung des zugehörigen Overheads. Dieser Overhead kann bei der Spezifikation einer *Action* definiert werden. Wie hoch so ein Overhead ausfällt,

kann für eine genaue Simulation mithilfe von Experimenten ermittelt werden. In einer Action wird ebenso definiert, was rekonfiguriert werden soll. Werden für jedes vorkommende Szenario, beispielsweise „Erstelle neuen Pod“ oder „Entferne bestehenden Pod“ *Actions* definiert, so kann in diesen *Actions* bereits der Overhead definiert werden, was zu einer erhöhten Genauigkeit der Simulation dynamischer containerbasierter Systeme führen kann. Auch der Ausführungszeitpunkt kann in *Actions* angegeben werden [37, p. 143ff].

Weitere Änderungen, die im *Behavioral Viewpoint* durchgeführt werden, beziehen sich in erster Linie auf *SEFF* von Service-Komponenten. Ansonsten müssen im *Behavioral Viewpoint* keine Änderungen umgesetzt werden.

### 3.1.3 Modellverwendung und -veränderungen / Erweiterungen auf Ebene des Deployment Viewpoint

Der *Deployment Viewpoint* umfasst das *Resource Environment Model*, sowie das *Allocation Model* [34, p. 57]. Mithilfe dieser Modelle werden die Kubernetes-Konzepte Cluster, Node und Deployment abgebildet.

**Cluster** Ein Kubernetes-Cluster ist eine Bündelung von Rechenressourcen, beziehungsweise Rechenressourcen, die zur Allokation von Komponenten genutzt werden können, siehe Unterabschnitt 2.1.1.

Die Anforderungen an ein Cluster im Rahmen dieser Arbeit wurden folgendermaßen definiert:

IR30: Ein Cluster muss durch
beliebig viele Kubernetes-Nodes beschrieben werden.
FA140: Das PCM muss dem Nutzer die Möglichkeit bieten, einem Cluster neue Kubernetes-Nodes hinzuzufügen.
FA150: Das PCM muss dem Nutzer die Möglichkeit bieten, aus einem Cluster Kubernetes-Nodes zu entfernen.

Als Abbildung eines Clusters wurde das *Resource Environment Model* gewählt. Dieses beschreibt *Resource Container* mit ihren Ressourcen-Spezifikationen. Die *Resource Container* können physische, wie auch virtuelle Maschinen abbilden. Kubernetes-Nodes entsprechen aus der Sicht des PCM *Resource Containern*. Diese Tatsache, zusammen mit der Umsetzung von Kubernetes-Nodes als Erweiterung von *Resource Containern*, erfüllt bereits alle definierten Anforderungen für das Cluster. Es sprechen jedoch noch weitere Gründe für eine Abbildung eines Clusters mithilfe des *Resource Environment Models*. Der erste ist die im PCM existierende Unabhängigkeit des *Resource Environment Models* vom darauf laufenden System. In Kubernetes ist ein Cluster unabhängig von den darauf laufenden Pods beziehungsweise Systemen. Ein Cluster definiert nur die zur Verfügung stehenden Kubernetes-Nodes. Ein weiterer Grund ist, dass somit für die Umsetzung eines Clusters kein weiteres PCM-Konzept erstellt werden muss. Das begünstigt die Möglichkeit auch

Cluster-/Kubernetes-externe Komponenten abzubilden, die nicht Teil der containerisierten Anwendung sind. Ein Beispiel hierfür wäre ein nicht in einem Pod laufender Datenbank-Server. Im *Resource Environment Model* könnte ein weiterer *Resource Container* erstellt werden, auf welchem im *Allocation-Model* die externe Datenbank-Komponente allokiert werden kann. Abschließend ist also zu sagen, dass das Cluster, unter der Bedingung, dass Kubernetes-Nodes als Spezialisierung von *Resource Containern* abgebildet werden, keine Änderung im Metamodell benötigt.

**Kubernetes-Nodes** Kubernetes-Nodes sind physische oder virtuelle Hardwareknoten. Es wird zwischen Worker-Nodes und Master-Nodes unterschieden. Die Anforderungen an Kubernetes-Nodes für die vorliegende Arbeit wurden mithilfe einer Integritätsregel [36] folgendermaßen definiert:

Eine Kubernetes-Node muss durch
eine Angabe einer Speicher-Spezifikation
UND einer Angabe einer CPU-Spezifikation in millicores
UND einer Spezifikation, ob es sich um eine Master-Node handelt
beschrieben werden.

Als Funktionale Anforderung wurde folgende Anforderung definiert:

FA160: Das PCM muss dem Nutzer die Möglichkeit bieten, Pods auf Kubernetes-Nodes im Allocation Model zu allokieren.
FA170: Das PCM muss dem Nutzer die Möglichkeit bieten, Pods, die auf Kubernetes-Nodes allokiert sind, aus dem Allocation Model zu entfernen.

Eine Kubernetes-Node ist einem *Resource Container* sehr ähnlich. Beide stellen Ressourcen zur Verfügung. Auf beiden können Software-Komponenten, beziehungsweise Teile von Systemen allokiert werden. Der *Resource Container* ist bereits in die funktionierende Simulationsanwendung SimuLizar integriert. Die notwendigen Erweiterungen eines *Resource Containers* zu einer Kubernetes-Node halten sich also stark in Grenzen. Deshalb bietet es sich an, die Kubernetes-Node als Spezialisierung des *Resource Containers* abzubilden.
Dieser Ansatz wurde gewählt und ist in Abbildung 3.6 zu sehen. Als Erweiterung zum *Resource Container* wurden Attribute für die Angabe des Speichers, der CPU in millicores und eine Option, ob es sich um eine Master-Node handelt, ergänzt. Bei diesen Attributen ist die Angabe des zur Verfügung stehenden Speichers vom Typ Long und muss, wie auf Seite 28 beschrieben, der gleichen Größenordnung beziehungsweise Einheit folgen, in welcher die Speicher-Requests und Limits angegeben werden sollen. Ansonsten ist bei dieser Angabe die Einheit nicht relevant. Für das millicores-Attribut wurde der Typ Integer gewählt. Diese Angabe soll die Anzahl der zur Verfügung stehenden millicores auf dieser Node beschreiben. 1000 millicores entsprechen einem Kern. Die Angabe, was

ein Kern ist, wird in Kubernetes nicht direkt spezifiziert. Es kann sich dabei um einen virtuellen Kern oder einen physichen Kern handeln. Wie stark dieser Kern ist, hängt vom zugrundeliegenden System oder Cloud-Provider ab. Ein Beispiel hierfür sind für AWS die AWS-vCPU oder bei Azure ein vCore [24]. In Palladio kann die Anzahl der Kerne mithilfe der Replicas einer *CPU Processing Resource* definiert werden. Ein Ansatz daraus die millicores einer Kubernetes-Node abzuleiten, wäre:

**Anzahl der Replicas · 1000 = millicores der Kubernetes-Node**

Die Leistungsfähigkeit der Kerne kann dann mit der *Processing Rate* spezifiziert werden. Die Anforderungen FA160 und FA170 sind automatisch erfüllt, da ein Pod eine Spezialisierung des *SubSystems* ist und eine Kubernetes-Node eine Spezialisierung des *Resource Container* ist. Auf einem *Resource Container* können *SubSysteme* allokiert werden, somit auch auf einer Kubernetes-Node.

**Deployment** Die Deployments in Kubernetes beschreiben den gewünschten Cluster-Zustand. Dabei werden Pod-Templates angegeben, sowie die Anzahl der Replicas und üblicherweise Requests und Limits der Container im referenzierten Pod. Auch wenn das Deployment in Kubernetes mächtiger ist als die Angabe der genannten Spezifikationen, ist das die Grundlage für alle weiteren Funktionalitäten. Im Rahmen dieser Arbeit werden nur diese Angaben abgebildet. Die Anforderungen an das Deployment wurden mithilfe einer Integritätsregel [36] definiert:

Ein Deployment muss durch
Einen Pod
UND eine Anzahl gewünschter Replicas
UND eine oder keine CPU-Request
UND eine oder keine Speicher-Request
UND ein oder kein CPU-Limit
UND ein oder kein Speicher-Limit
UND beliebig vielen zusätzlichen Requests
UND beliebig vielen zusätzlichen Limits
beschrieben werden.

Pods und damit indirekt Pod-Templates wurden bereits eigenständig abgebildet, siehe Seite 24. Um ein Pod-Template zu referenzieren, erfordert es lediglich eine Referenz auf einen im *Repository Model* definierten Pod. Auch die Requests und Limits von Containern sind eingenständig als Konzept in dieser Arbeit zu finden, siehe Seite 28. In der gewählten Abbildung des Pods in das PCM beschreibt ein Pod bereits mithilfe der in ihm assemblierten Container die Requests und Limits des Deployments. Somit wird ein Teil einer Deployment-Spezifikation bereits bei der Definition der Container im *Repository Model* erledigt. Eine Deployment-Spezifikation muss in dieser Arbeit also nur noch den gewünschten Zustand mithilfe der Replicas beschreiben und dabei einen Pod aus dem *Repository Model* referenzieren. Diese Umsetzung ist im UML-Diagramm unter der Assoziation podReference, siehe

Abbildung 3.6, zu sehen. Die Anzahl der gewünschten Instanzen des Pods ist mithilfe eines Integer-Attributs replicas umgesetzt. Grundsätzlich wäre die Definition des Deployment-Konzepts an dieser Stelle fertig. Es wurde aber entschieden, ähnlich wie bei den Requests und Limits, eine Modellsicht hinzuzufügen, die alle Deployments enthält. Hierfür wurde die Klasse Deplyoments erstellt, welche aus beliebig vielen Deployment-Objekten besteht. Es besteht jetzt also die Möglichkeit in einer Deployment-Sicht beziehungsweise einem Deployment-Modell alle Deployments zu definieren, die den gewünschten Cluster-Zustand bilden. Mithilfe einer Kontrollschleife zur Simulationszeit könnte dann der gewünschte Zustand im Deployment-Modell mit dem tatsächlichen Zustand im *System Model* und *Allocation Model* verglichen werden. Bei einem Unterschied könnten dann entsprechende Rekonfigurationen vorgenommen werden. Es wurde jedoch entschieden, die Deployment-Modelle als Deployment-Repository zu definieren. Ein Deployment ist dann ein möglicher Baustein, um einen Cluster-Zustand zu definieren. Die tatsächliche Zusammensetzung des gewünschten Cluster-Zustands geschieht im *System Model* mithilfe des neu erstellten DeploymentAssembly-Objekts, siehe Abbildung 3.6 im unteren Bereich. Dieses DeploymentAssembly ist eine Spezialisierung eines *Assembly Contexts*. Dabei referenziert das DeploymentAssembly ein Deployment. Alle assemblierten Deployments im *System Model* stellen dann den gewünschten Zustand dar. Dieser Zustand muss dann konsistent mit den assemblierten Pods sein.
Mit dieser Umsetzung des Deployment-Konzepts können definierte Deployments schnell wiederverwendet werden, sind also System-unabhängig. Zusätzlich werden alle oben erwähnten Anforderungen an das Deployment erfüllt und es wird so eine deklarative Definition eines gewünschten Cluster-Zustands ermöglicht.

**Nested Resource Container** In dieser Arbeit wurden hierarchische Pods mit integrierten Containern vorgestellt. Für diese Konzepte wurden Abbildungsmöglichkeiten in das PCM gefunden. Containern können Requests und Limits zugewiesen werden, die die Ressourcennutzung oder Zuweisung von Ressourcen einschränken. Ein Ziel dieser Arbeit wurde definiert als:
„Vorhersage des Einflusses von Requests und Limits auf die Performance einer containerisierten Anwendung/Pods in Kubernetes."
Die Ressourcen-Zuweisung auf Container basiert in den meisten Implementierungen auf hierarchischen cgroups, siehe Unterabschnitt 2.1.14. Hierbei werden Ressourcen auf control-groups zugeteilt. Ein Container kann dabei eine cgroup sein und somit Ressourcen, wie Speicher und CPU, zugewiesen bekommen. Für hierarchische cgroups existiert eine prototypische Implementierung eines Schedulers für Palladio, siehe [10]. Dieser verwendet *Resource Container*, welche als *Nested Resource Container* wiederum *Resource Container* enthalten können, und verteilt Ressourcen hierarchisch nach den Angaben der einzelnen Resource Container, beispielsweise der *processing rate*. Es liegt also nahe, für eine Simulation von containerisierten Anwendungen diesen Scheduler zu verwenden. Da dieser Scheduler auf einer Verschachtelung von *Resource Containern* basiert, müssen die Container in den Pods eigene *Resource Container* erhalten, denn ein Container besitzt üblicherweise Ressourcen-Requests und -Limits. Das ist auch der Grund, warum der Pod als Spezialisierung des *SubSystems* und nicht der *Composite Component* umgesetzt

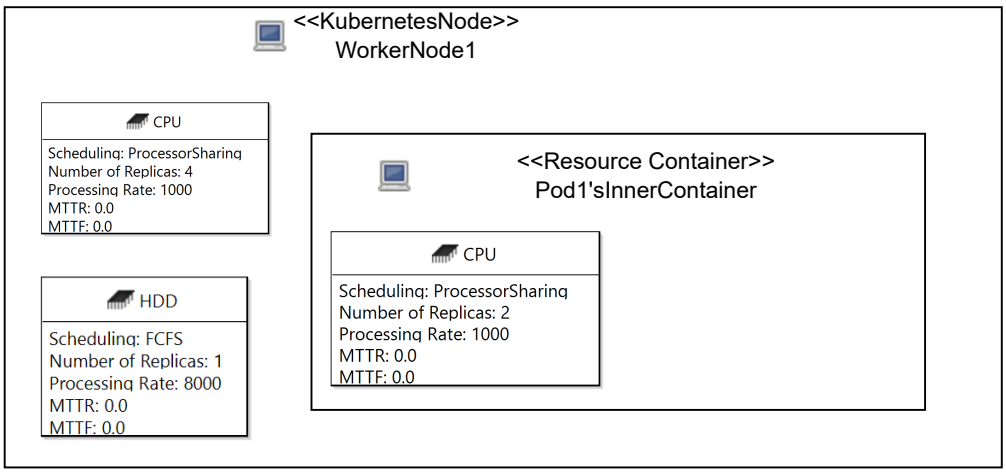


Abbildung 3.4: Resource Environment Model mit innerem Resource Container

wurde. Durch die Umsetzung als *SubSystem* können die inneren Container auf andere *Resource Container* verteilt werden, als auf den Pod selbst. Die inneren Container eines Pods dürfen trotzdem nur auf die inneren *Resource Containern* des *Resource Containers* allokiert werden, auf dem der Pod allokiert ist. Um die Allokation von Pods mithilfe von *Nested Resource Containern* besser darzustellen, wird im Folgenden ein Beispiel erläutert: Abbildung 3.4 bildet einen Ausschnitt eines Clusters (*ResourceEnvironment*) ab, mit einem *Nested Resource Container*. Der äußere *Resource Container* ist vom Typ Kubernetes-Node. Er stellt eine Worker-Node dar und besitzt eine Ressource-Spezifikation für HDD und CPU. Die Anzahl der Kerne der Worker-Node beträgt 4, ausgedrückt durch die Angabe der *Replicas* der *Processing Resource CPU*. Der innere *Resource Container* ist für die Allokation des inneren Containers von Pod1 aus dem *Repository Model* gedacht. Dieser innere *Resource Container* hat zwei Kerne zur Verfügung, ebenso ausgedrückt durch die Anzahl der *Replicas*. Dieser Pod wurde im *System Model* einmalig instaziiert und enthält genau einen Container mit einer CPU-Request von 2000m. Eine *HDD-Processing-Rate* hat der innere *Resource Container* nicht. Im *Allocation Model* welches dieses Szenario allokiert, siehe Abbildung 3.5, wurde Pod1 auf der äußeren KubernetesNode allokiert. Der innere Container von Pod1 wurde korrekt auf dem inneren *Resource Container* allokiert. Mit dieser Allokation kann der Nested-Resource-Container-Scheduler verwendet werden. Da dieser Scheduler Requests bei der Zuweisung der verfügbaren CPU-Zeit grundsätzlich umsetzt, können CPU-Requests in Simulationen bereits berücksichtigt werden. Für Hauptspeicher gibt es keine *Processing Resource* in Palladio. Deshalb werden Speicher-Requests und Limits bei der Simulation nicht berücksichtigt. Diese spielen nur für das Allokations-Scheduling von Pods eine Rolle, siehe Kapitel 5.

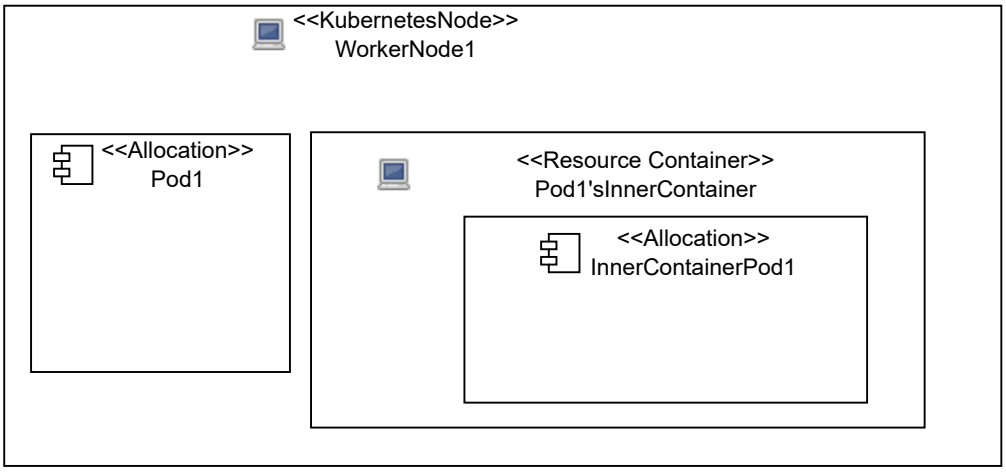


Abbildung 3.5: Allocation Model mit Pod und Container in innerem Resource Containers

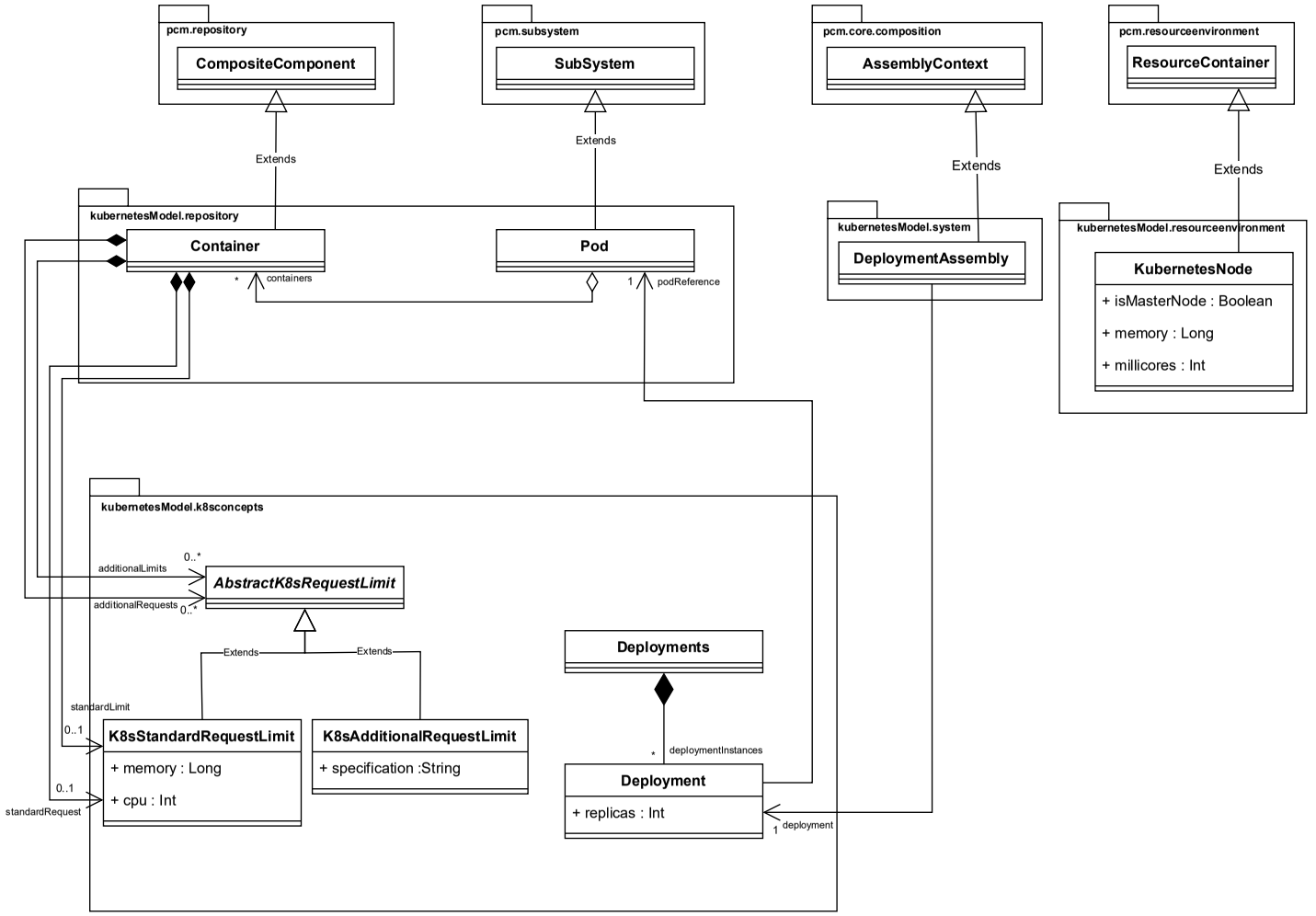


Abbildung 3.6: Klassendiagramm des Entwurfs für die Modellerweiterung

## 3.2 Implementierung

Die Implementierung der Modellerweiterung wurde mithilfe eines EMF-Modells als Erweiterung des bestehenden PCM umgesetzt.
Dabei wurde für neue Objekte das `kubernetesModel.k8sconcepts` Paket erstellt. Die anderen Pakete wurden mit führendem `kubernetesModel` nach den Modellen benannt, die sie erweitern. Nach der Generierung des Java-Codes mithilfe des zugehörigen genmodels konnten die neu eingeführten Konzepte in der Palladio-Bench verwendet werden.

# 4 PCM-Kubernetes-Workflow

Dieses Kapitel beschreibt die Verwendung der Modellerweiterung bei der Modellierung von Software mithilfe des PCM unter Verwendung der erarbeiteten Kubernetes-Konzepte. Es wird dabei auch diskutiert, welche der aus dem Palladio Ansatz bekannten Rollen für welchen Schritt (mit-)verantwortlich ist.

Der PCM-Kubernetes-Workflow ist aufgeteilt in drei Hauptaktivitäten, siehe Abbildung 4.1. Diese lauten: System definieren, Cluster definieren und System deployen. Dabei beinhaltet die Aktivität *System definieren* den eigentlichen Entwicklungsprozess der Software. Der zweite Schritt des Workflows begibt sich bereits in die Richtung des Deployens des Systems, indem dort die Ausführungsumgebung, also das Cluster definiert wird. Im letzten Schritt werden die Deployment-Spezifikationen erstellt. Diese Aufteilung in drei Aktivitäten ermöglicht eine einfache Austauschbarkeit des Clusters bzw. des Deployments. Vorangegangene Schritte können hier einfach wiederverwendet werden. Im Folgenden werden die drei Hauptaktivitäten des Workflows weiter spezifiziert und erklärt. Es ist wichtig zu erwähnen, was nicht im Workflow abgebildet wird. Das beinhaltet das Requirements-Engineering beziehungsweise die Definition, was für eine Software gebaut oder abgebildet werden soll, sowie die Erstellung einer System-Architektur. Es wird für diesen Workflow also angenommen, dass vorliegt, welche Schnittstellen und Komponenten es geben soll und wie diese zu einem System kombiniert werden.

## 4.1 Struktur des Systems erstellen

Das Aktivitätsdiagramm der geschachtelten Aktivität *System definieren: Struktur des Systems erstellen* ist in Abbildung 4.2 zu sehen und wird im Folgenden beschrieben. Die erste Aktivität lautet *Interfaces definieren* und beinhaltet das Erstellen der *Interfaces* inklusive iherer Operationen im *Repository Model*. Diese Interfaces werden im nächsten Schritt *Software-Komponenten erstellen* benötigt, um die Einzel-Komponenten, aus denen das System beziehungsweise die Container gebaut werden, vollständig zu definieren. Danach werden in *RDSEFF der Komponenten erstellen* alle *Resource Demanding SEFFs* der Komponenten spezifiziert. Bis zu diesem Schritt sind keine Änderungen im Vergleich zur Abbildung einer Software ohne Kubernetes-Erweiterung erfolgt, sondern nur ein möglicher Ablauf definiert. In der vierten Aktivität *Container ohne Requests und Limits erstellen* werden die Container der Software definiert. Das bedeutet, dass die Komponenten, die in einem Container gekapselt werden sollen, zu einer zusammengesetzten Komponente, dem Container, assembliert werden. Hierbei ist zu erwähnen, dass der Container dabei ein Container-Diagram enthält. In diesem werden mithilfe von *Assembly Contexts* Software-Komponenten aus dem *Repository Model* zu einem Container assembliert. Dabei wird die

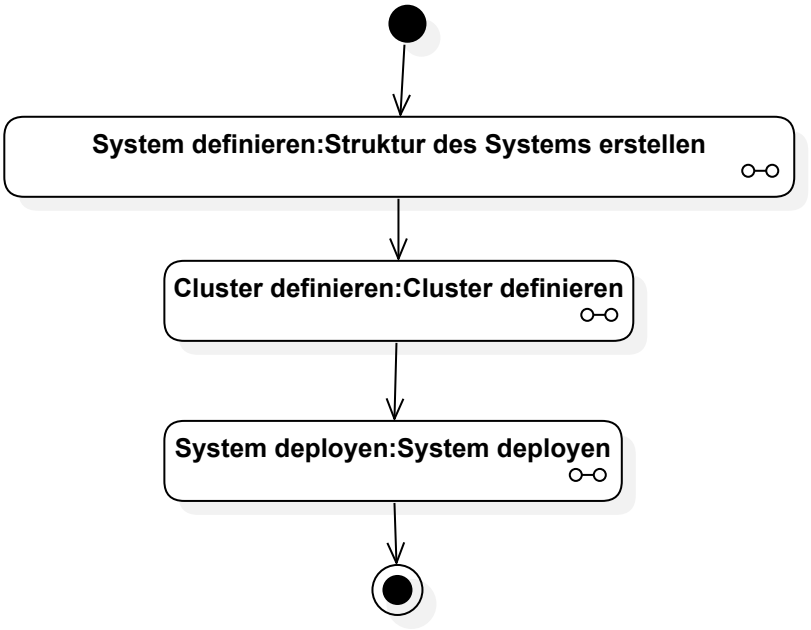


Abbildung 4.1: Aktivitätsdiagramm des PCM-Kubernetes-Workflows

innere Kommunikation zwischen den Software-Komponenten abgebildet. Um darzustellen welche Dienste ein Container anbietet oder benötigt, werden *Provided Roles* und *Required Roles* zu den *Interfaces*, die die Operationen des Diensts spezifizieren, erstellt. Von außen sind nur diese Rollen sichtbar. Anfragen werden mithilfe von *Delegation-Connectoren* an die inneren Software-Komponenten delegiert, die die jeweilige Anfrage abarbeiten können. Wichtig ist, dass an dieser Stelle die Request- und Limit-Referenz des Containers leer bleibt.
Nach erfolgreicher Definition aller Container werden die Pods in der Aktivität *Pods erstellen* spezifiziert. Die Information, welche Container zu Pods zusammengefasst werden, liegt an dieser Stelle vor, oder kann erarbeitet werden, ist im Workflow aber nicht explizit ausgewiesen. Das Gleiche gilt bei der Erstellung der Container aus Software-Komponenten. Grundsätzlich sind zwei Arten von Pods möglich. Es ist üblich, nur einen Container in einem Pod zu kapseln (Single-Container-Pod). Jedoch können ebenso mehrere Container einen Pod bilden (Multi-Container-Pod). Das ist sowohl von Kubernetes, als auch mit dieser PCM-Modell-Erweiterung abbildbar. Die Container eines Pods werden in diesem Schritt, ähnlich wie für den Container mit Software-Komponenten, im eigenen Container-Diagram in *Assembly Contexts* zusammengesetzt. Für genau einen Container im Pod wird nur ein *Assembly Context* benötigt. In diesem Fall erhält der Pod die gleichen *Provided-* und *Required-Roles* wie der innere Container. Die Rollen des Containers werden dann mit *Delegation Connectoren* im Pod-Diagram mit den äußeren Rollen des Pods verbunden. An dieser Stelle ist die Spezifikation eines Single-Container-Pods in diesem Schritt des Workflows abgeschlossen. Für einen Multi-Container-Pod müssen mehrere Container in *Assembly Contexts* im Pod assembliert werden. Die Kommunikation zwischen den inneren Containern wird wie bei der Spezifikation eines Containers aus Software-Komponenten

durch *Assembly Connectoren* dargestellt. Ergibt sich hieraus, dass eine Schnittstelle von einem inneren Container benötigt wird, die ein anderer innerer Container des Pods anbietet und dieser Dienst auch nicht von außerhalb des Pods erreicht werden soll, dann muss dieser Pod keine Beziehung zum zugehörigen *Interface* im *Repository Model* erhalten. Für alle Dienste die der Pod anbietet oder benötigt, werden wie beim Container *Provided-* und *Required-Roles* angegeben. Im Pod-Diagram werden diese mithilfe von *Delegation Connectoren* mit den entsprechenden Rollen der inneren Container verbunden.
Im nächsten Schritt des Workflows wird sich um die Service-Definition gekümmert. Auf Seite 25 wurde beschrieben, welche Schritte notwendig sind, um einen Kubernetes-Service mithilfe dieser Erweiterung abzubilden. Diese Schritte werden in der Aktivität *Interne Service-Komponenten erstellen* für jeden im System benötigten Service durchgeführt. Das beinhaltet die Defintion der Service-Komponente, der Angabe des zugehörigen Interfaces, beziehungsweise bei Bedarf der Erstellung eines passenden Interfaces, sowie der Definition einer initialen *SEFF*, die das Load-Balancing abbildet. Hierbei wird die auf Seite 25 beschriebene Struktur zunächst nur für eine Instanz umgesetzt. Im späteren Verlauf des PCM-Kubernetes-Workflows wird diese *SEFF* an die gewünschte Deployment-Spezifikation angepasst.
Nach der Spezifikation der internen Services wird die Schnittstelle nach außen spezifiziert. Gemeint ist hierbei die Definition der von außerhalb des Systems erreichbaren Schnittstelle. In Kubernetes wird so etwas mit einem Ingress umgesetzt. Für die Abbildung in das PCM handelt es sich konzeptuell um eine Service-Komponente, die im System Model instanziiert wird und einen *Delegation Connector* zur System-Schnittstelle hat. Diese Schnittstelle muss jedoch erst im *Repository Model* als *Interface* erstellt werden.
Dann werden in der nächsten Aktivität *Service-Loadbalancing-SEFFs erstellen* die *SEFFs* dieser Komponente erstellt. Dabei referenzieren die Aufrufe nach innen aber nicht direkt Pods, sondern Rollen der bereits erstellten Service-Komponenten.
Darauffolgend wird im *System Model* in der Aktivität *Ingress Provided Role erstellen* die den Ingress repräsentierende *System Operation Provided Role* erstellt. Nach einer Instanziierung der Ingress-Service-Komponente im Schritt *Ingress Loadbalancer instanziieren* werden die Service-Komponenten aus dem *Repository Model* ebenso im *System Model* instanziiert. Dies geschieht so oft, wie verschiedene Service-Instanzen tatsächlich im System benötigt werden, denn es können mehrere Services des gleichen Typs existieren. Als letzter Schritt werden in *System-Struktur mit einer Instanz pro Pod definieren* die benötigten Pods instanziiert. In diesem Schritt wird die Architektur des Systems auf Pod-Ebene im *System Model* abgebildet. Mit einer Instanz pro Pod ist gemeint, dass jeder Pod zu einem Deployment wird. Die Anzahl der Deployments wird damit in diesem Schritt definiert. Wie viele Replicas es von einem Pod am Ende gibt, kann dann skaliert beziehungsweise in der Deployment-Spezifikation angegeben werden.
An dieser Stelle ist die Hauptaktivität *System definieren*, siehe Abbildung 4.1 abgeschlossen.

## 4.2 Cluster definieren

Das Aktivitätsdiagramm dieser Hauptaktivität ist in Abbildung 4.4 zu sehen und wird im Folgenden beschrieben. In der vorangegangenen Hauptaktivität wurde die Struktur des

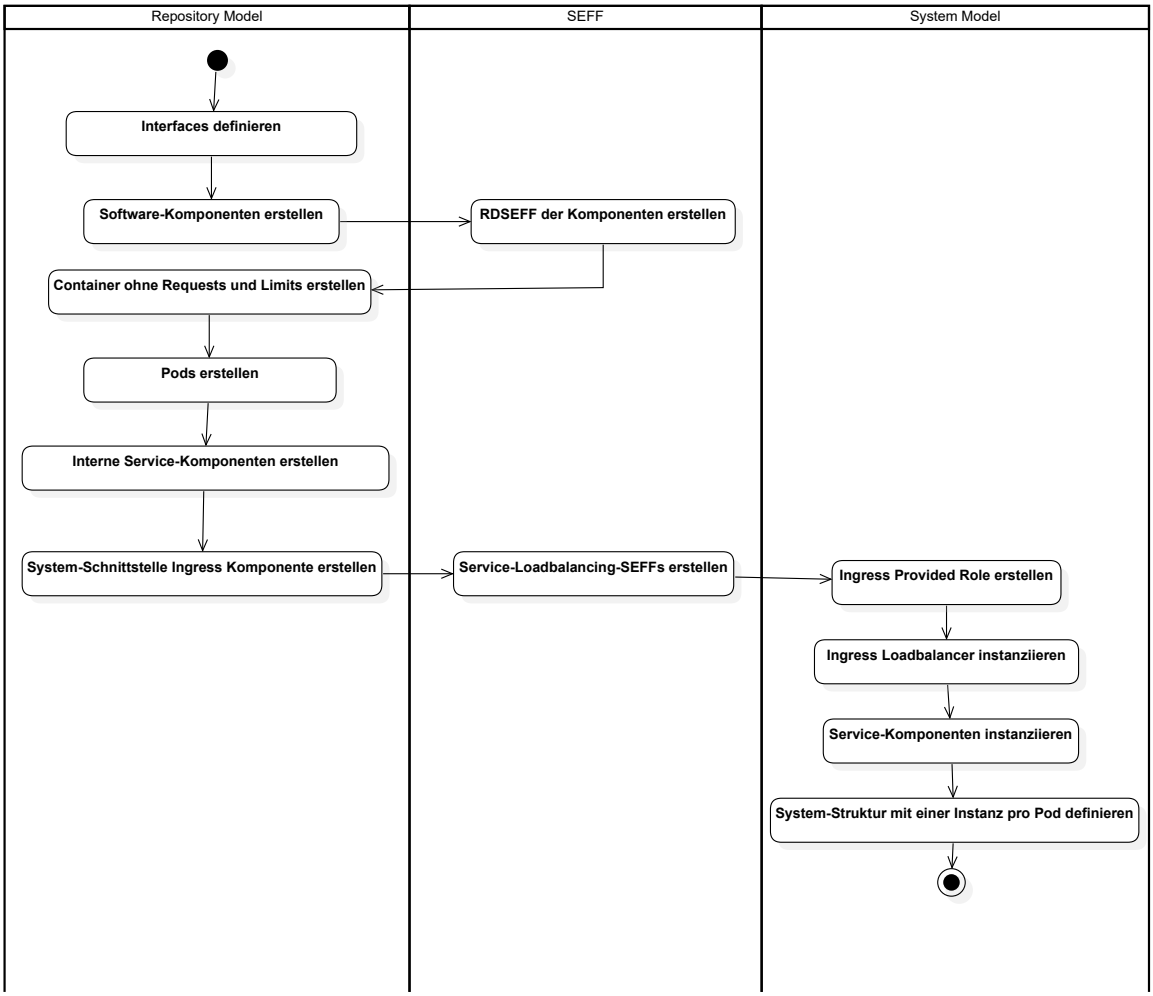


Abbildung 4.2: Aktivitätsdiagramm der Aktivität „System definieren“ siehe 4.1

Systems definiert. Diese Angaben sind unabhängig von der Ressourcenumgebung, in der das System später läuft. Da es sich bei dieser Modellerweiterung um eine Kubernetes-Erweiterung handelt, wird als *Resource Environment* ein Cluster definiert. Dieses Cluster kann selbstverständlich gegen ein anderes Cluster ausgetauscht werden. Die Struktur des Systems ist auch in containerisierter Form unabhängig von der Ausführungsumgebung.
Um ein Cluster zu definieren, muss im ersten Schritt *Zur Verfügung stehende Rechenressourcen analysieren* bestimmt werden, welche Rechenressourcen oder Maschinen überhaupt zur Verfügung stehen. Unter Umständen müssen diese erst definiert werden, weil es sich beispielsweise um einen Cloud-Anbieter handelt, der praktisch so viele Ressourcen zur Verfügung stellen kann, wie angefragt werden. Grundsätzlich schränkt man sich hier aber auf eine bestimmte Konfiguration ein und hat dann eine feste Menge an physischen oder virtuellen Maschinen vorliegen um das Cluster zu erstellen.
Im nächsten Schritt wird definiert, wie viele Kubernetes-Nodes erstellt werden sollen oder zur Verfügung stehen. Nachdem alle Informationen über das verwendete oder zu Simulationszwecken definierte Cluster vorliegen, können im Schritt *Kubernetes-Nodes im Resource Environment Model erstellen* die Kubernetes-Node-Instanzen in Palladio erstellt werden. Daraufhin müssen die CPU- und Speicher-Werte angegeben werden. Diese sind für die Allokation der Pods auf den Nodes wichtig. Es kann zusätzlich angegeben werden, ob es sich bei den Knoten um Master-Nodes handelt. Sind die Angaben über die Ressourcen zur Allokation erledigt, können die Palladio *Processing Resources* definiert werden. Bei diesen handelt es sich um die von SimuLizar verwendeten Ressourcen. Im letzten Schritt werden Linking Resources zwischen den Kubernetes-Nodes erstellt. Im Cluster werden alle Knoten miteinander verbunden. Nach diesem Schritt ist die Beschreibung des *Resource Environments* abgeschlossen. Ein Beispiel eines definierte *Resource Environment Model* ist in Abbildung 4.3 zu sehen. In diesem Modell gibt es zwei Worker-Nodes und eine Master-Node. Auf den zwei Worker-Nodes wurden bereits *Resource Container* erstellt, um jeweils zwei Pods mit ihren inneren Containern zu allokieren. Die Pods enthalten jeweils nur einen Container.

## 4.3 System deployen

Die letzte Hauptaktivität des PCM-Kubernetes-Workflows ist das Deployen des Systems. Hierbei werden Deployment-Spezifikationen definiert, Requests und Limits definiert, sowie das *System Model* auf den in den Deplyoments definierten Zustand gebracht. Am Ende werden die Pod-Replicas im *Allocation Model* allokiert. Das Aktivitätsdiagramm dieser Aktivität ist in Abbildung 4.5 zu sehen. Es ist aufgeteilt in die Lanes: *Repository Model, System Model, SEFFs, Resource Environment Model, DeploymentsRepository* und *Allocation Model*. Jede dieser Lanes beschreibt das Modell in welchem Änderungen vorgenommen werden. Aktivitäten in einer Lane beschreiben, dass diese Aktivität in diesem Model zu Veränderungen führt oder dieses Model hautpsächlich benötigt wird.
Das Deployments Repository ist ein eigenständiges Modell, in dem alle Deployments definiert werden. Hintergedanke ist, dass so Deployments schnell ausgetauscht werden können, da man diese bereits definiert vorliegen haben kann. Ziel des gesamten Schrittes *System deployen* ist, den Prozess der Definition eines Deployments bis zur tatsächlichen

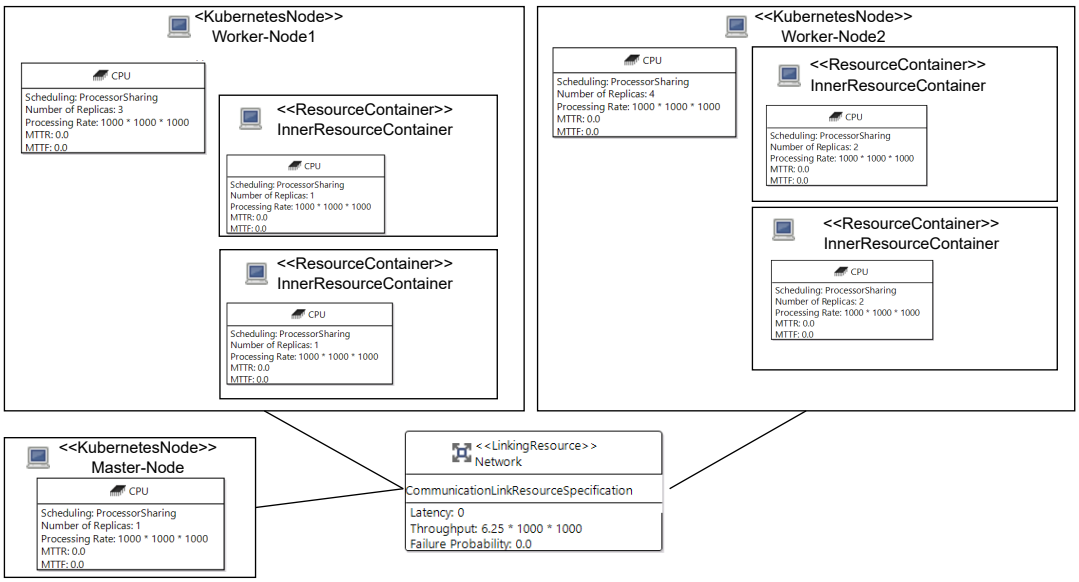


Abbildung 4.3: Resource Environment Beispiel aus der Evaluation

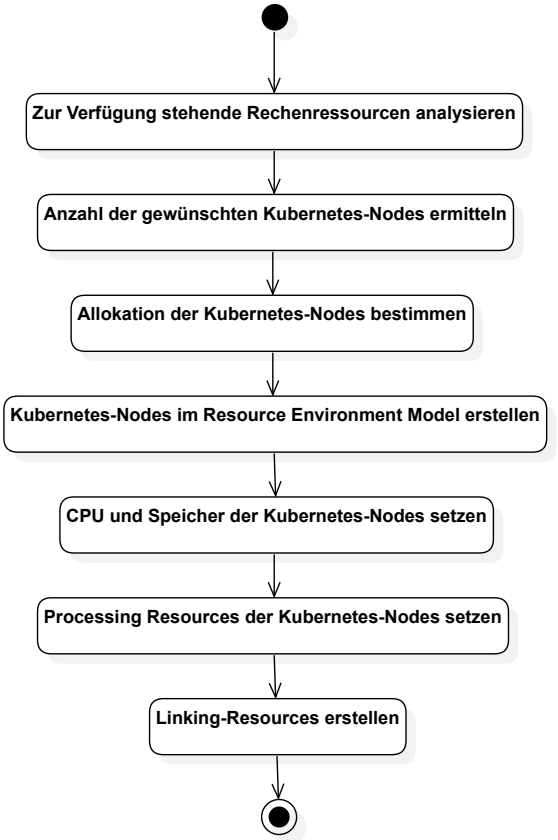


Abbildung 4.4: Aktivitätsdiagramm der Aktivität „Cluster definieren“ siehe 4.1

Ausführung in Kubernetes abzubilden. Hierfür wird mit der Definition des gewünschten Cluster-Zustands begonnen. Im Schritt *Deployments definieren* werden im Deployments Repository die Deployment-Objekte erstellt und spezifiziert, die später benötigt werden. In der Aktivität *Struktur des Systems erstellen*, siehe Abschnitt 4.1, wurde bereits pro benötigter Pod-Gruppierung, welche ein Deployment spezifizieren soll, ein Pod erstellt. Dieser Pod wird im Deployment angegeben und stellt eine Art Pod-Template, beziehungsweise Image dar. Zusätzlich wird die Anzahl der gewünschten Replicas des Pods definiert. In einer Deployment-Spezifikation werden üblicherweise die Requests und Limits der Container des Pods definiert. Aus diesem Grund werden im Schritt *Requests für Container erstellen* die Request für jeden Container erstellt. Dabei werden insbesondere die CPU- und Speicher-Requests angegeben. Nach den Requests werden in *Limits für Container erstellen* alle Limits erstellt. Nachdem die Requests und Limits definiert wurden, ist die Deployment Spezifikation abgeschlossen. Es wurde ein Pod im Deployment referenziert, Request- und Limit-Angaben gemacht (implizit über die Container), wobei ein Deployment aber eine spezielle Pod-Instanz im Repository referenziert. Dadurch ist auch diese Angabe eindeutig. Um Deployments unabhängig von einem möglichen gewünschten Zustand des Clusters definieren zu können, wurde, wie für Software-Komponenten, die Entwurfs-Entscheidung getroffen, Deployments in Repositories auszulagern. Es wurde entschieden, dass die Definition eines Deployments im DeploymentRepository noch nicht bedeuetet, dass es auch tatsächlich verwendet wird. Stattdessen muss hierfür ein DeploymentAssembly im System-Model verwendet werden. Alle Deployment-Assemblies definieren dann den gewünschten Zustand des Clusters. Im DeploymentsRepository liegen nur die Bausteine dafür vor. Im Schritt *Gewünschte Deployments assemblieren* werden also genau die Deployments mithilfe von DeploymentAssemly-Objekten definiert, die dann die gewünschte Anzahl an Replicas beziehungsweise zusammen genommen den gewünschten Gesamt-Zustand des Clusters beschreiben. Nach diesem Schritt ist also bekannt, wie viele Replicas es von welchem Pod und damit auch bezogen auf welchen Service geben soll. Dadurch kann im Schritt *Service-Komponenten Provided- und Required-Roles erstellen* für jede Pod-Instanz, die einen bestimmten Service anbietet, in diesem Service eine *Required Role* erstellt werden. Diese Required-Role bewirkt dann, dass in der Loadbalancing *SEFF* an diese *Required Role* eingehende Anfragen weitergeleitet werden können. Für jede Pod-Instanz die einen bestimmten Service benötigt, wird eine *Provided Role* erstellt, damit diese Pod-Instanzen ihre Anfragen an den Service senden können. Nachdem alle Rollen der Service-Komponenten erstellt wurden, können im Schritt *Loadbalancing-SEFF anpassen* die *SEFFs* der Services angepasst werden. Je nach Loadbalancing-Strategie müssen diese unterschiedlich verändert werden. Der auf Seite 25 vorgestellte Ansatz mit einem zufälligen, gleichmäßigen Scheduler führt dazu, dass pro *Required Role* des Services, also pro Pod-Instanz die diesen Service anbietet, eine weitere *Probabilistic Branch Transition* erstellt wird. Die Wahrscheinlichkeit wird entsprechend der gewünschten Pod-Instanzen angebenen. Nach der Anpassung der *SEFFs* an die definierten Pod-Instanzen, sowie der Anpassung der Service-Komponenten um die Anzahl der tatsächlich benötigten Rollen, können die Pods instanziiert werden. Dies geschieht im *System Model* im Schritt *Pod-Replicas instanziieren*. Hierbei wird die Anzahl an *Assembly Contexts* pro Pod erstellt, wie in der Replica-Spezifikation des Deployments angegeben. Die *Assembly Contexts* der Pod-Instanzen werden dann noch mithilfe von *Assembly Connectoren* mit den entsprechenden, zuvor erstellten Rollen der Service-Komponenten

verbunden. Nach diesem Schritt ist das *System Model* vollständig definiert und enthält die gewünschte Anzahl an Pod-Instanzen, die Deployment Assemblies, die den gewünschten Cluster-Zustand definieren und die Service-Komponenten.
Jetzt müssen die Pods, wie auch alle anderen Komponenten, allokiert werden. Dies geschieht in den darauffolgenden Schritten: *Allocation der Pods bestimmen, Pods allokieren* und *Innere Container allokieren* im Allocation Model. Zusätzlich müssen, für die Verwendung des Nested-Container-Schedulers, innere *Resource Container* im *Resource Environment Model* erstellt werden. Der Schritt *Allocation der Pods bestimmen* beinhaltet die Verwendung des Pod-Allocation-Schedulers, siehe Kapitel 5. Dieser ermittelt eine Allokation der Pods auf den Kubernetes-Nodes im Cluster. Ist eine Verteilung der Pods auf die Kubernetes-Nodes erfolgt, können im nächsten Schritt *Pods allokieren* die Pod-Instanzen auf den Kubernetes-Nodes im *Allocation Model* nach den Vorgaben des Schedulers allokiert werden. Gleichzeitig werden die Service-Komponenten entweder auf der Master-Node oder auf einer beliebigen Kubernetes-Node allokiert. Je nachdem ob für die Service-Komponenten *Resource Demands* spezifiziert wurden oder nicht, hat die Allokation dieser einen oder keinen Einfluss auf die Simulation.
Wie in Absatz 3.1.3 beschrieben, müssen für die inneren Container der Pods extra *Resource Container* im *Resource Environment Model* erstellt werden. Aus dem vorangegangenen Schritt ist klar, welche Pod-Instanz sich auf welcher Kubernetes-Node befindet. Auf Basis dieser Informationen werden die inneren *Resource Container* erstellt, um die inneren Container im Schritt *Innere Container allokieren* final zu allokieren.

Nach der Allokation der inneren Container wurde das System, nach den Angaben der Deployments, der Spezifikation der Pods und Container, sowie der Definition des Systems im *System Model* vollständig mithilfe des PCM und der Kubernetes-Erweiterung abgebildet. Definiert man jetzt ein *Usage Model*, sowie die zu messenden Werte im *Monitor Repository* von Palladio, kann das containerisierte System simuliert werden. Dabei wurde die Allokation der Komponenten mithilfe des Allocation-Schedulers berechnet. An dieser Stelle ist eine Abbildung und Simulation von containerisierten, statischen mit Kubernetes deployten Systemen mithilfe von Palladio möglich.

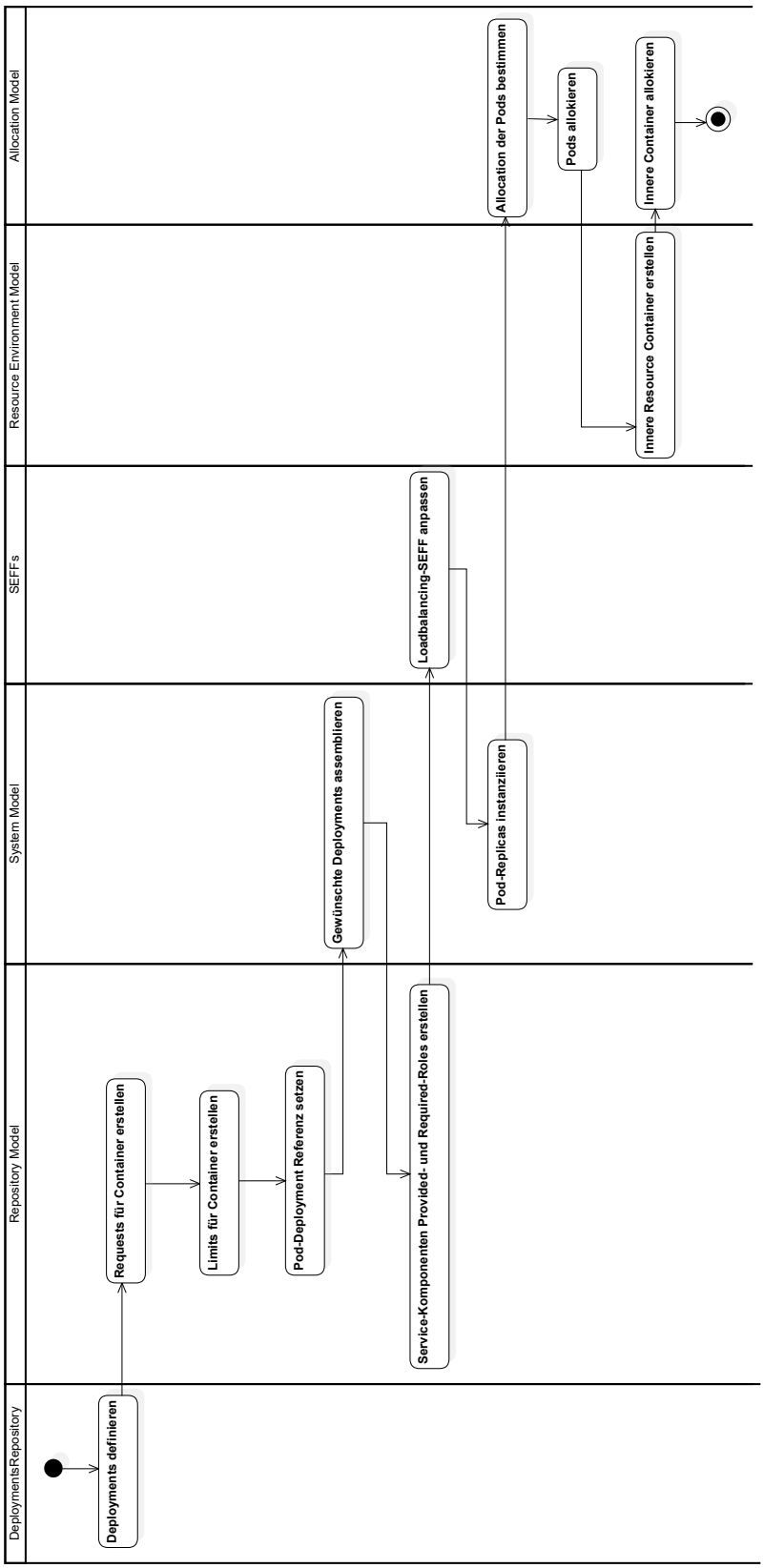


Abbildung 4.5: Aktivitätsdiagramm der Aktivität „System deployen“ siehe 4.1

# 5 Pod-Allokations-Scheduler

In diesem Kapitel wird der Pod-Allokations-Scheduler für die PCM-Kubernetes-Erweiterung beschrieben. Die Grundlagen zur Allokation von Pods in Kubernetes werden in Unterabschnitt 2.1.8 beschrieben. Das Pod-Scheduling in Kubernetes ist aufgeteilt in drei Schritte: Filtern, Bewerten, Auswählen, siehe Abbildung 5.1. Für die Implementierung eines Schedulers für das PCM wurde diese Struktur beibehalten. Als Eingabedaten für den Scheduler muss bekannt sein, welcher Pod gescheduled werden muss, wie der aktuelle Zustand des Clusters aussieht, sowie welche Spezifikationen die Kubernetes-Nodes des Clusters haben. Im Schritt Filtern, werden alle Kubernetes-Nodes herausrausgefiltert, die nicht genügend Ressourcen zur Verfügung haben. Das bedeutet, dass die Differenz aus der Summe der Requests der bereits auf ihnen allokierten Pods, abgezogen von den auf der Kubernetes-Node verfügbaren Ressourcen, zu gering ist um einen weiteren Pod zu reservieren. Danach wird eine Reihenfolge für jede „Feasible Node“ auf Basis einer Score-Bewertung bestimmt. Zuletzt wird die Node gewählt, welche die höchste Punktzahl erhalten hat. Bei gleicher Punktzahl wird zufällig gewählt. Die Scheduling-Kriterien können angepasst werden, wobei für die Implementierung dieses Schedulers eine einfache Logik gewählt wurde. Jede Node, die noch genügend nicht-reservierte (unrequested) Ressourcen hat, wird als „Feasible “ vermerkt. Da es keine weiteren Regeln gibt, erhalten alle Knoten die gleiche Bewertung in Schritt zwei, sodass im dritten Schritt (Selection) eine zufällige Node gewählt wird. Die Regeln des Schedulers sind sowohl in Kubernetes, als auch in Palladio austauschbar, sodass auch eine auf weiteren Informationen basierende Bewertung umsetzbar wäre, beispielsweise „Bewerte die Node am besten, die prozentual noch am meisten Ressourcen frei hat“.

**Implementierung / Umsetzung mithilfe der Modellerweiterung** Die prototypische Implementierung des Allokations-Schedulers wurde in Java umgesetzt. Der Scheduler ist als eigenständiges Palladio-Plugin zu verwenden und ist nicht Teil der Modell-Erweiterung auf Metamodell-Ebene. Das bedeutet, dass die Funktionalitäten, wie beispielsweise den freien Platz auf einer Kubernetes-Node zu berechnen, keine Funktionalität der Klasse `KubernetesNode` darstellt, sondern im Scheduler implementiert ist. Um die Schnittstelle des Schedulers im Rahmen einer QVTo-Transformation nutzen zu können, wurde eine Black-Box Schnittstelle definiert, die aus der Transformation heraus aufgerufen werden kann. Dabei soll der aktuelle Ist-Zustand in Form der Modelle zur Laufzeit als Eingabedaten übergeben werden. Die relevanten Modelle sind das *Repository Model*, das *Allocation Model*, das *Repository Model* und das *System Model*. Zurückgegeben wird eine Empfehlung, wo welcher nicht-allokierte Pod gescheduled werden soll. In der Umsetzung der aktuellen Implementierung wurde auch das Erkennen von nicht-allokierten Pods umgesetzt. Dabei wird der Zustand des *System Models* mit dem Zustand des *Allocation Models* verglichen.

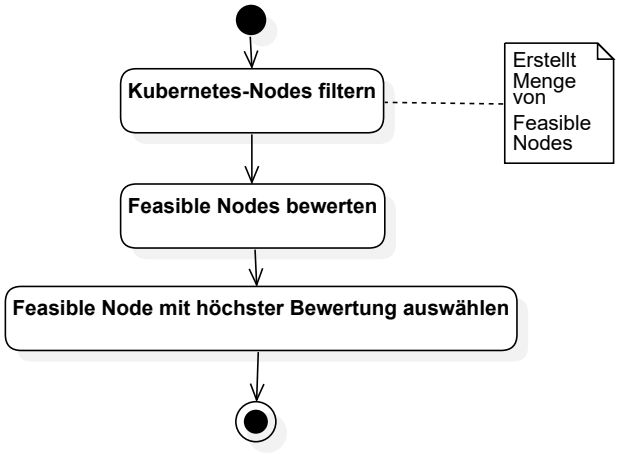


Abbildung 5.1: Pod-Allokations-Scheduler Aktivitätsdiagramm

Es wird überprüft, ob es *Assembly Contexte* im *System Model* gibt, für die kein *Allocation-Objekt* im *Allocation Model* existiert. Den in einem solchen *Assembly Context* enthaltenen Pod müsste man dementsprechend allokieren. Die Black-Box-Methode der Bibliothek gibt ein Tupel als Map<Assembly Context, KubernetesNode> zurück. Jedes darin enthaltene Paar in der Rückgabe der Black-Box-Methode zur Transformation kann dann allokiert werden. Grund hierfür ist, dass während der Ausführung der schedule Methode zuerst überprüft wird, welche Pods allokiert werden müssten. Dieser Vergleich bezieht sich auf die im *System Model* instanziierten Pods. Ein Vergleich zwischen *System Model* und definierten Deployments könnte zusätzlich noch gemacht werden. Für die Ergebnis-Tupel können dann in der QVT Operational (QVTo)-Transformation die notwendigen Modellveränderungen durchgeführt werden. Grund für diese Aufteilung in Java-Code und QVTo-Transformation ist einmal die gute Unterstützung von QVTo-Transformationen als Teil von Reconfigurations in SimuLizar, da QVTo viele Mechanismen mitbringt um Modelle zu transformieren. Die Logik eines Schedulers, wie des Allokations-Schedulers, zu implementieren ist jedoch in Java einfacher und besser testbar. Zusätzlich kann diese Bibliothek anderweitig wiederverwendet werden. Da QVTo die Verwendung von externen Java-Methoden erlaubt, wurde eine Kombination aus Java und QVTo gewählt und somit der Baustein des Allokations-Schedulers in Java implementiert.
Um die Funktionalitäten des Schedulers zu testen wurden Unit-Tests zusammen mit Beispiel-Modellen erstellt.

# 6 Simulation dynamischer containerisierter Software-Architekturen

In dieser Arbeit wurde erarbeitet, wie containerisierte Software-Systeme modelliert werden können. Das vorgestellte Konzept beschreibt bisher aber nur eine statische Abbildung, sowie die Entscheidungsfindung, wo welcher Pod allokiert wird. Diese statischen Modelle können mit Palladio simuliert werden, um beispielsweise die Antwortzeiten verschiedener Konfigurationen zu messen. Moderne verteilte Software-Systeme werden jedoch in vielen Fällen nicht mehr statisch, sondern dynamisch deployed und rekonfiguriert. Diese Tatsache motiviert die Entwicklung eines dynamischen Simulationskonzepts um die Auswirkungen auf die Performance einer dynamisch deployten Software, beispielsweise mit Kubernetes, bereits frühzeitig zu analysieren.
Mithilfe der bereits erwähnten Modell-Erweiterung können bereits ohne automatisierte Simulation, dynamische Systeme simuliert werden. Dies ist möglich, indem man bei jeder Änderung am Deployment, die Simulation stoppt, Änderungen an den Modellen manuell durchführt und dann die Simulation wieder startet. Nachteil dieses Ansatzes ist, dass die Simulation einer automatisierten Skalierung durch die vielen Modelländerungen sehr aufwendig ist. In diesem Kapitel wird ein dynamisches Simulationskonzept für containerisierte Anwendungen mithilfe von Palladio beschrieben, welches automatisiert auf Ereignisse reagieren kann, die den System-Zustand beeinflussen.
Es stellt sich jedoch die Frage, welche Anforderungen es für eine dynamische Simulation gibt, insbesondere, welche Szenarien simulierbar sein sollen. Hierfür wurden vier Szenarien gefunden, die zur Laufzeit eines Systems eintreten können und eine dynamische Anpassung des Cluster-Zustands erfordern:

1. Pod wird durch das System terminiert, da er beispielsweise zu viel Speicher verwendete (Out-of-Memory (OOM)-Killer) oder eine Kubernetes-Node ist ausgefallen wegen eines Hardware-Fehlers.
2. Das System wird durch steigende oder geringer werdende eingehende Last durch das Nutzungsszenario skaliert beziehungsweise durch Fehler im System werden mehr Kapazitäten benötigt.
3. Ein neuer Dienst wird mithilfe eines Deployments in Betrieb genommen (rolling release), beziehungsweise durch ein Update eines Dienstes wird ein neues Deployment mit neuer Pod-Version erstellt und die Pods des alten Serivces werden graduell ersetzt.
4. Ein Deployment wird wegen eines Updates entfernt, da für die neue Version ein neues Deployment erstellt wurde.

Es werden also folgende Anforderungen an das dynamisches Container Simulationskonzept (DCSK) definiert:

FA160: Das DCSK muss in der Lage sein, Unterschiede zwischen Cluster-Zustand und Deployments festzustellen.
FA170: Wenn ein allokierter Pod durch das System terminiert wurde, muss das DCSK überprüfen, ob der Pod neu allokiert werden muss.
FA180: Wenn entschieden wurde, dass ein Pod allokiert werden muss, muss das DCSK überprüfen können, ob eine Allokation im aktuellen Cluster möglich ist.
FA190: Wenn das DCSK eine Allokation für einen Pod gefunden hat, muss das DCSK in der Lage sein diesen Pod zu allokieren.
FA200: Wenn ein Pod zu viel zwischen dem definierten Zustand in den Deployments und dem Cluster-Zustand festgestellt wurde, muss das DCSK in der Lage sein diesen Pod zu deallokieren.
FA210: Wenn ein Pod zu wenig zwischen dem definierten Zustand in den Deployments und dem Cluster-Zustand festgestellt wurde, muss das DCSK in der Lage sein eine Allokation für diesen Pod zu triggern.
FA220: Das DCSK muss die Überprüfungen, ob Änderungen notwendig sind, zur Simulationszeit in definierbaren Abständen durchführen.

## 6.1 Technische Umsetzbarkeit

Mithilfe der Palladio-Simulation SimuLizar können bereits Veränderungen zur Simulationszeit an den Modellen vorgenommen werden. Häufig verwendete Möglichkeiten hierfür sind: Reconfigurations mit QVTo, *Actions* und Henshin einer Sprache zur Definition von in-place EMF-Modell-Transformationen [1]. Das wahrscheinlich wichtigste Konzept sind die QVTo-Reconfigurations. Mithilfe von QVTo lassen sich Modell-Transformationen kompakt ausdrücken. In Palladio können diese Transformationen zur Simulationszeit verwendet werden. Dabei können durch eine Messung der Simulation ausgelöst, die spezifizierten Rekonfigurationen ausgeführt werden. Hierbei kann jedes Modellelement verändert werden. Die in Unterabschnitt 3.1.2 angesprochenen *Actions* können ebenso Veränderungen zur Simulationszeit durchführen. *Actions* kapseln dabei mehrere Schritte einer Modell-Veränderung und können als eine Art Bausteine wiederverwendet werden. Es kann auch ein Einfluss auf die Performance bei der Durchführung der Action angegeben werden. Es wurde beispielsweise ein horizontales-Scaling bereits mit *Actions* umgesetzt. *Actions* basieren dabei aber ebenso auf QVTo-Transformationen, die die Modellveränderungen umsetzen. Für das Triggern der Ausführung einer *Action* können Bedingungen definiert werden [37, p. 146 ff.].
Modellveränderungen zur Simulationszeit sind also möglich, was eine Simulation der obigen Szenarien ermöglichen sollte. Die Erstellung des Simulationskonzepts orientierte sich an den Möglichkeiten der Verwendung von Rekonfigurationen mit QVTo in Kombi-

nation mit Black-Box Methoden in Java. Ob dieses Konzept ebenso mit *Actions* oder einem anderen Mechanismus umsetzbar ist, ist anzunehmen, wurde aber nicht weiter betrachtet.

## 6.2 Simulationskonzept

Für dieses Konzept wird angenommen, dass das zugrundeliegende Modell mithilfe des PCM-Kubernetes-Workflows, siehe Kapitel 4, erstellt wurde, oder dass eine äquivalente Struktur vorliegt. In Kubernetes werden die Überprüfungen des aktuellen Zustands des Clusters mithilfe verschiedener Kontrollschleifen umgesetzt. Um eine ständige Überprüfung des Modell-Zustands zu ermöglichen, müssen die notwendigen Überprüfungen also ebenso in einer Schleife ausgelöst werden. Bei der Verwendung von Reconfigurations mit QVTo wäre ein möglicher Auslöser die Ausführung der Reconfigurations nach jeder Messung. Eine Ausführung vor der ersten Messung, da bereits bei Start der Simulation die Zustände verschieden sein könnten, könnte auch umgesetzt werden, da die Rekonfigurationen auch simulationsunabhängig ausführbar sind.
Ein Aktivitätsdiagramm der definierten Kontrollschleife für das dynamische Simulationskonzept unter Verwendung der Kubernetes-Erweiterung ist in Abbildung 6.1 zu sehen. Eine mögliche Umsetzung wäre, dass zwischen jeder Messung diese Kontrollschleife ausgelöst wird, bis es keine Veränderungen mehr in den Modellen gibt. Die Aktivitäten referenzieren hierbei direkt die neuen Metamodell-Elemente. Im Folgenden wird dieses Aktivitätsdiagramm beschrieben.
Die Kontrollschleife wird dabei beispielsweise zwischen zwei Messungen gestartet und läuft solange, bis keine Veränderungen durchzuführen sind. Der nächste Start würde dann nach der nächsten Messung geschehen. Damit würde für jede Messung der Modell-Zustand fest definiert werden, beziehungsweise alle Transformationen würden zwischen zwei Messungen stattfinden.
Die erste Aktivität *Deployment-Assemblies mit Pod-Instanziierungen vergleichen* betrachtet das *System-Model*. Es werden die Deployment-Assemblies mit den *Assembly Contexts*, die Pods enthalten, verglichen. Wird eine Differenz zwischen den Deployments und den instanziierten Pods festgestellt, muss eine Veränderung im *System Model* vorgenommen werden, was zur nächsten Entscheidung führt. Diese überprüft, ob es einen Pod zu viel oder zu wenig im *System Model* gibt. Wenn eine Pod-Instanz zu viel instanziiert ist, also die Anzahl Replicas die das Deployment spezifiziert niedriger ist, als die Anzahl Instanziierungen, muss ein *Assembly Context*, der zu viel ist, aus dem *System Model* entfernt werden. Ist der Pod bereits allokiert, müssen sowohl die Allokation des Pods, wie auch die Allokationen auf den inneren *Resource Containern* entfernt werden. Zusätzlich müssen im nächsten Schritt die *Resource Container* aus dem *Resource Environment Model* entfernt werden. Ein Beispiel für einen „zu viel“ instanziierten Pod ist ein horizontales Skalieren nach innen, welches seit der letzten Iteration der Kontrollschleife durchgeführt wurde, indem die Anzahl der Replicas eines Deployments erhöht wurde.
Wird festgestellt, dass es einen Pod zu wenig gibt, wird für die fehlende Pod-Instanz ein *Assembly Context* erstellt. Die Service-Komponente, bei der der neue Pod registriert werden muss (In Kubernetes würde hier die Service-Discovery diesen Schritt übernehmen), muss *Provided Roles* und *Required Roles* für den neuen Pod erhalten. Zusätzlich

muss die Service-SEFF angepasst werden, damit dieser neue Pod ebenso im Loadbalancing berücksichtigt wird. Nach diesen Änderungen am *System Model* wurde die Differenz zwischen Deployment-Assemblies und Pod-Instanziierungen behoben, weshalb auf die rechte Seite der Abbildung 6.1 gewechselt wird, zur Entscheidung „Allocation für jeden Pod gefunden?“Hier nimmt der Kontrollfluss den „Nein“ Zweig, da ein neuer Pod gerade instanziiert wurde, für den es noch keine Allokation geben kann. Dieser wird in den darauf folgenden Schritten allokiert. Der Zweig „Pod-Instanz zu wenig –> Ja“ würde beispielsweise gewählt werden, wenn eine horizontale Skalierung nach außen skaliert, also Pods hinzugefügt werden. Eine Skalierung nach innen oder außen bedeutet hier, dass die Deployment-Replicas verändert wurden. Die Schritte, was passiert, wenn ein Unterschied zwischen den Deployment-Assemblies und den Pod-Instanziierungen im *System Model* vorliegt, wurden damit erklärt. Es kann aber auch sein, dass im *System Model* kein Unterschied zwischen Deployment-Assemblies und Pod-Instanzen vorliegt, also dass das *System Model* die Deployments richtig abbildet und damit der gewünschte Zustand des Clusters korrekt definiert ist. Dennoch können Änderungen an den Modellen notwendig sein. Ein Szenario hierfür ist, dass ein Pod terminiert wurde, beispielsweise weil er zu viel Speicher verwendete. Dann ist der gewünschte Zustand aus dem *System Model* noch korrekt, das *Allocation Model* stimmt aber nicht mehr mit dem definierten Zustand überein. In diesem Fall wird der rechte Zweig „Differenz –> [Nein] “ gewählt und dann das *System Model* mit dem *Allocation Model* verglichen. Wird in diesem Schritt für jeden Pod aus dem *System Model* eine Allokation gefunden, geht die Kontrollschleife in die nächste Iteration. Ist das aber nicht der Fall, dann wird die geschachtelte Aktivität *Allocation für Pod bestimmen:Allocation Scheduler* siehe Abbildung 5.1 ausgeführt. Diese verwendet den bereits definierten Allokations-Scheduler für Pods und bestimmt eine Kubernetes-Node auf welcher der Pod allokiert werden kann, solange es eine Node mit genügend freien Ressourcen gibt. Ist das nämlich nicht der Fall, geht die Kontrollschleife einfach in die nächste Iteration. Der Pod bleibt dann unallokiert solange bis eine mögliche Allokation gefunden wird. Sollte es aber eine mögliche Allokation geben, wird eine *Allocation* im *Allocation Model* für den Pod erstellt. Zusätzlich müssen die Nested Resource Container für diesen Pod auf der Kubernetes-Node erstellt werden, um danach die inneren Container des Pods auf den inneren *Resource Containern* zu allokieren. Sind auch die inneren Container allokiert, wird wieder überprüft, ob jeder Pod eine Allokation hat, bis alle nicht-allokierten, aber allokierbaren Pods allokiert sind. Danach geht es in die nächste Iteration.
Der Prozess, sowie die Schritte die durchgeführt werden müssen, sind implementierungs unabhängig. Ob Reconfigurations oder *Actions* oder Java oder etwas anderes gewählt werden. Ein Vorschlag zur Umsetzung wäre jedoch, die Logik, also das Vergleichen der Modelle mithilfe von Black-Box-Methoden in Java umzusetzen. Die Veränderungen an den Modellen lassen sich einfacher mit QVTo realisieren. Es ist sicher auch möglich, *Actions* zu definieren, die das darstellen, wie in [37, p. 154ff.] mit einem horizontalen Skalierer gezeigt wurde. Mithilfe von *Actions* würde sich auch der Kubernetes-spezifische Overhead, beispielsweise durch das Starten oder Schedulen eines Pods gut darstellen lassen.

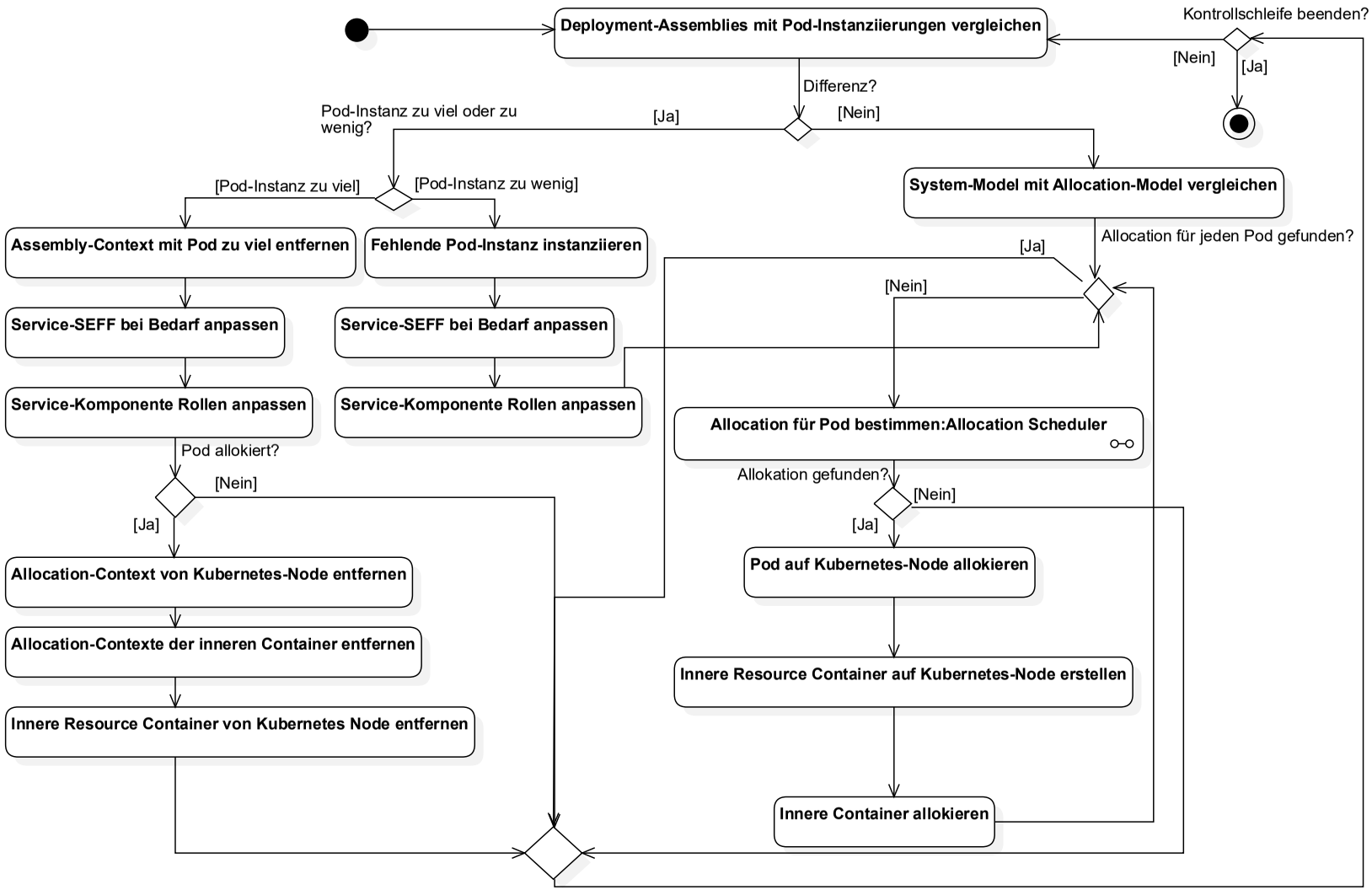


Abbildung 6.1: Aktivitätsdiagramm zur Abbildung einer Kubernetes-PCM-Kontrollschleife zur Einhaltung des definierten Cluster-Zustands

## 6.3 Veränderungen des Systems simulieren

In diesem Kapitel wurde bisher beschrieben, wie Veränderungen durch das System (Pod wird wegen OOM-Error terminiert) oder durch den Nutzer (Deployment wird skaliert, erstellt oder entfernt), erkannt und durchgeführt werden können. Hierbei handelt es sich aber um Szenarien, die unabhängig vom bisher vorgestellten Simulationskonzept auftreten. In realen Kubernetes-Clustern oder laufenden Anwendungen passieren diese Szenarien immer wieder, entweder durch das System oder den Nutzer. Für eine realistische Simulation müssen diese Szenarien ebenso simuliert werden können. Im Folgenden wird deshalb für jedes der oben vorgestellten Szenarien vorgeschlagen, wie man dieses Szenario in eine Simulation einbinden könnte.

1. **Pod wird terminiert:** Reconfiguration mit QVTo erstellen, die zu einem zufälligen Zeitpunkt oder mehreren Zeitpunkten getriggert wird (beispielsweise abhängig von den bisher durchgeführten Messungen) und eine beliebige *Allocation* aus dem *Allocation Model* entfernt, die einen Pod enthält.

2. **System wird skaliert (Horizontaler Pod Autoskalierer):** Reconfiguration mit QVTo erstellen, die zu einem zufälligen Zeitpunkt oder festen Zeitpunkt die Anzahl der Replicas eines oder mehrerer assemblierten Deployment Assemblies erhöht oder verringert. Der Zeitpunkt kann ebenso abhängig von den Ergebnissen der letzten Messung definiert werden. Beispiel: Aus „Antwortzeit des Systems zu hoch“ folgt „Replicas werden erhöht“ und umgekehrt. Solch eine Reconfiguration würde einen automatischen Skalierer beschreiben.

3. **Neues Deployment wird erstellt:** Reconfiguration erstellt ein neues DeploymentAssembly.

4. **Deployment wird entfernt:** Reconfiguration entfernt ein DeploymentAssembly aus dem *System Model*.

Mit diesen Szenarien und entsprechenden Reconfigurations lassen sich viele Real-Welt-Szenarien, die ein Cluster durchläuft, abbilden und simulieren.

# 7 Verwandte Arbeiten

## 7.1 Kubernetes-Abbildung für Palladio

Ghirardin et al. [6] stellen einen Ansatz vor, Kubernetes bzw. Kubernetes-ähnliche Cloud-Cluster-Architekturen mithilfe von Palladio zu simulieren. Es wurde ein Modell entwickelt, welches eine lauffähige Simulation von verschiedenen Konfigurationen eines Kubernetes-Clusters ermöglicht. Dieses Modell soll dazu dienen, Entwicklern die Möglichkeit zu geben, gute Konfigurationen für ihre Anwendung zu finden. Es hätte sich im Rahmen der Bachelorarbeit angeboten, die Modelle genauer zu betrachten und evtl. zu erweitern. Leider sind die Modelle nicht verfügbar und auf Nachfrage an die Autoren wurde mitgeteilt, dass der Hauptautor nicht mehr an der Universität von Bozen sei und es auch sonst keine Möglichkeit gäbe, in die Arbeit einzusehen. Der dort gewählte Ansatz befasst sich nicht mit einer Erweiterung des PCM sondern vielmehr mit einer Möglichkeit, Pods oder ein Kubernetes-Cluster mithilfe von Palladio abzubilden. Die Abbildungsmöglichkeiten von Services oder der Allokation von Pods wurde im Rahmen dieser Arbeit nicht ausführlich behandelt.

## 7.2 Palladio-Simulation mit Kubernetes

Bei SimulationAutomation [5] handelt es sich um ein Projekt, welches die Ausführung von Palladio mit Kubernetes ermöglicht. Es handelt es sich hierbei nicht um eine Simulation von Kubernetes oder Ähnlichem mithilfe von Palladio. Diese Arbeit ist als Abgrenzung aufgeführt.

## 7.3 ContainerCloudSim

ContainerCloudSim ist ein Ansatz, um auf Basis des Cloud-Simulations-Werkzeuges CloudSim, Container-basierte Infrastrukturen testen zu können. CloudSim selbst beschränkt sich auf VMs und die Darstellung und Simulation von Containern, welche immer mehr an Relevanz gewinnen. Die Arbeit von Saleh und Mashaly versucht diese Lücke in der Simulation von Cloud-Infrastrukturen auf Container-Basis zu schließen [35]. In Abgrenzung zu dieser Arbeit liegt hier der Fokus auf der reinen Erweiterung der bestehenden Simulation um Containerkonzepte mit einem Fokus der Simulation von containerbasierten Infrastrukturen. Die vorliegende Arbeit dagegen erweitert eine andere Simulations-Software um die Möglichkeit Container, aber auch Kubernetes-Konzepte abzubilden und dynamisch zu simulieren.

## 7.4 Allokations Scheduler

Zur Simulation der Pod-Allokation gibt es bereits Simulatoren. Beispiele hierfür sind Hidehito et al. [40], mit deren Hilfe Cluster simuliert werden können ohne sie zu deployen. Das umfasst auch den Pod Scheduler. Alternativ dazu gibt es eine durch Alibaba Open Source entwickelte Software die speziell für die Simulation der Kubernetes-Pod-Allokation entwickelt wurde, [38]. Mithilfe dieser Software lässt sich in Spezifikationsdateien ein Cluster definieren. Auf deren Basis kann dann eine Allokation von Pods durchgeführt werden. Grundsätzlich wird hier auch eine Änderungen der Scheduler-Regeln unterstüzt.

## 7.5 Dynamische Simulation im PCM und transiente Effekte

Die vorliegende Arbeit beschäftigt sich mit der dynamischen Simulation von containerisierten Software-Architekturen. Dabei kann es von Bedeutung sein, den Overhead, der beispielsweise durch das Hochfahren eines neuen Pods entsteht, ebenso zu simulieren. In dieser Arbeit wurden Möglichkeiten erwähnt, diesen Overhead abzubilden. Eine Arbeit die sich speziell mit transienten Effekten bei der dynamischen Simulation von Systemen beschäftigt, ist die Dissertation von Christian Stier [37]. In dieser wird ein Modell vorgestellt, mit welchem sich Rekonfigurationen von Modellen kapseln lassen, sowie auch ein synchroner oder asynchroner Overhead simuliert werden kann [37, p. 143 ff.]. Mithilfe dieser Erkenntnisse können auch Reconfigurationen, wie sie in dieser Arbeit beschrieben wurden, umgesetzt werden und gleichzeitig ein einfach definierbarer Overhead spezifiziert werden.

# 8 Evaluation

Im Verlauf der vorliegenden Arbeit wurde das PCM um Container und Kubernetes-Konstrukte erweitert. Um die Erweiterung, sowie die Abbildung containerisierter Software-Architekturen besser nutzbar und verständlicher zu machen wurde ein Workflow zur Abbildung in das PCM entwickelt. Um containerisierte Software-Architekturen nicht nur abbilden, sondern auch dynamisch simulieren zu können, wurde ein Pod-Allokations-Scheduler, sowie ein dynamisches Simulationskonzept erarbeitet. Die Motivation dahinter ist, dass moderne Software-Systeme oft nicht mehr statisch, sondern mithilfe einer deklarativen Spezifikation dynamisch deployed werden. Um die Möglichkeit einer deklarativen Spezifikation von Systemen mit Palladio, sowie der anschließenden Simulation dieser umzusetzen, wurden folgende Forschungsfragen definiert:

1. Forschungsfrage: *Wie können Container in Palladio modelliert werden?*
2. Forschungsfrage: *Welche Kubernetes-Konstrukte lassen sich in Palladio abbilden?*
3. Forschungsfrage: *Wie lassen sich nicht statische Deployments in Palladio abbilden und simulieren?*
4. Forschungsfrage: *Wie lassen sich Skalierung HPA und Rollouts simulieren?*
5. Forschungsfrage: *Wie lassen sich Auswirkungen von Throtteling und Resource-Sharing schon zur Design-Zeit vorhersagen?*

Dieses Kapitel evaluiert die entwickelte Modellerweiterung und die Simulationskonzepte. Die Evaluation ist mithilfe von Goal-Question-Metric / Ziel-Frage-Metrik (GQM)-Plänen organisiert, die Ziele definieren, welche zur Beantwortung der in Kapitel 1 definierten Forschungsfragen beitragen.

## 8.1 GQM-Modellerweiterung

**Goal 1:** Ausarbeitung und Implementierung einer Modellerweiterung um Container und Konzepte von Containerisierungswerkzeugen wie Kubernetes mithilfe des PCM abbilden.
**Adressierte Forschungsfragen:** 1, 2, 3

**Question 1.1:** Können Container in Palladio modelliert werden?

**Metric 1.1.1:** Umgesetzter Anteil der Anforderungen an das Modellelement Container in Prozent.

**Metric 1.1.2:** Abdeckung des Modells für beispielhafte Definition eines Containers.

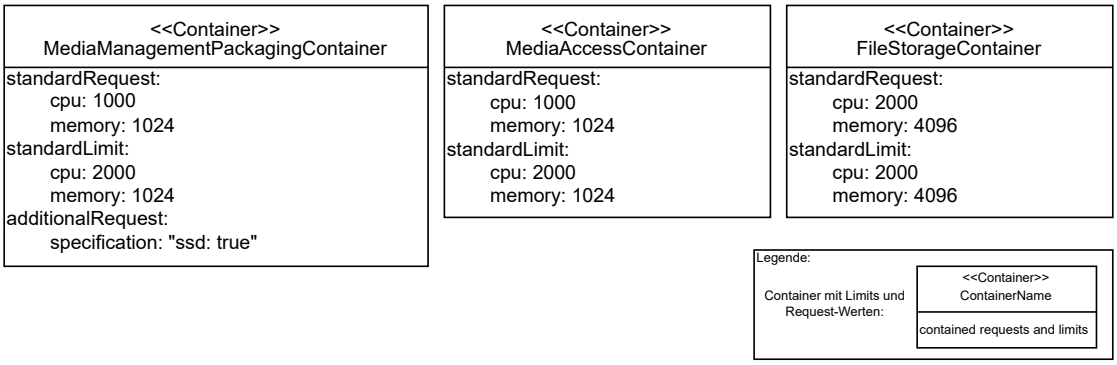


Abbildung 8.1: Container mit Requests und Limits

**Question 1.2:** Lassen sich alle wichtigen Grundkonzepte von Kubernetes in Palladio abbilden?

**Metric 1.2.1:** Übersicht der abgebildeten Konstrukte im Vergleich zu den wichtigsten Grundkonzepten aus Kubernetes. Anteil berechnen.

**Metric 1.2.2:** Definition eines Referenz-Cluster-Zustands, welcher mithilfe der Erweiterung abgebildet wird. Modell auf Übereinstimmung mit Referenz-Cluster prüfen, Anzahl der Unterschiede zählen.

**Question 1.3:** Lässt sich ein mithilfe von Deployments deklarativ definierter Cluster-Zustand abbilden?

**Metric 1.3.1:** Mithilfe von Deployments beschriebenen Cluster-Zustand mit Palladio abbilden und Anzahl der Abweichungen bestimmen.

**Question 1.1:** Die Anforderungen an einen Container wurde mithilfe der Integrationsregel IR10 aus 3.1.1 definiert. Die Modellerweiterung für den Container, siehe Seite 22, enthält alle Anforderungen. Die Umsetzung dieser Anforderungen kann in der Abbildung der Metamodell-Erweiterung, siehe Abbildung 3.6, welche implementiert wurde, betrachtet werden. Sie beträgt 100%. Es wurden alle definierten Anforderungen abgebildet. Für die zweite Metrik dieser Frage, wurde folgendes Szenario definiert:

**Szenario:** Ein Nutzer möchte die Cacheless-MediaStore-Anwendung aus den Palladio Screencasts [32] containerisieren um sie später mithilfe von Kubernetes zu deployen. Zuvor möchte er diese Änderungen im Palladio-Simulator abbilden. Er entscheidet, die Komponenten MediaManagement und Packaging in einem Container zu kapseln, wohingegen die MediaAccess-Komponente sowie die FileStorage-Komponente einen eigenen Container erhalten. Aus Erfahrungswerten der Auslastung des bisherigen Deployments auf zwei Servern, werden die Requests und Limits für die Container wie in Abbildung 8.1 abgebildet, gesetzt.

Das Ergebnis der Umsetzung dieses Szenarios führt zum in Abbildung 8.2 abgebildeten Ergebnis. In der Abbildung sind die Request- und Limit-Objekte, die in den Containern enthalten sind, nicht abgebildet. Diese sind aber wie in Abbildung 8.1 erstellt worden.

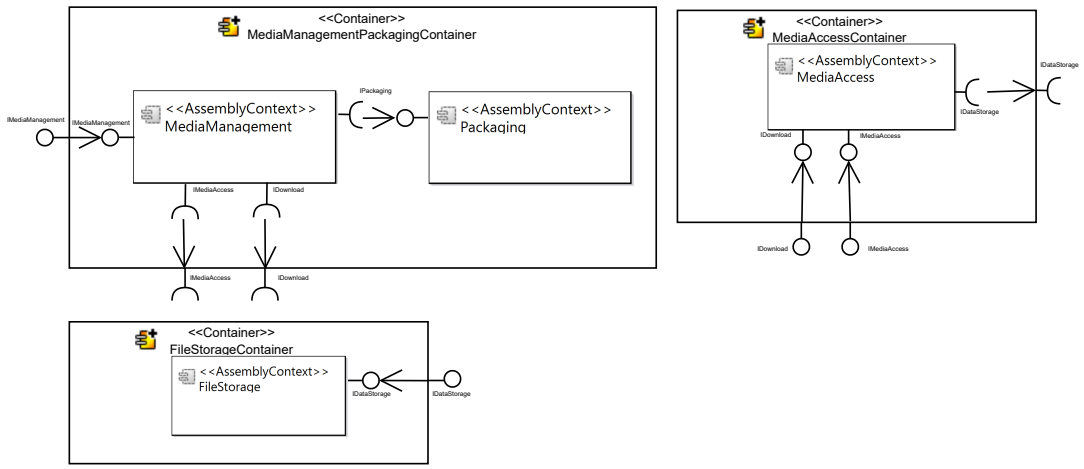


Abbildung 8.2: Ergebnis des Szenarios zur Containerisierung des Cacheless-MediaStores - Innere Container-Modelle

Diese Container können nun wie normale *Composite Components* verwendet werden. Die Requests und Limits werden bei einer reinen Abbildung eines Containers nicht berücksichtigt, da diese lediglich bei der Verwendung von Kubernetes eine Rolle spielen. Die Frage, wie Container in Palladio abgebildet werden können, ist damit beantwortet und mithilfe eines Beispiels evaluiert.

**Question 1.2:** Im Rahmen dieser Arbeit wurden Lösungen erarbeitet, wie wichtige Kubernetes-Konstrukte in Palladio abbildbar sind. Dafür wurden teilweise neue Modellelemente hinzugefügt. Teilweise konnten Konzepte aber bereits aus den vorhandenen Möglichkeiten des PCM abgebildet werden. In Tabelle 8.1 findet sich eine Übersicht mit allen in dieser Arbeit behandelten Kubernetes-Konstrukten. Grundsätzlich gibt es noch deutlich mehr, jedoch handelt es sich hier um die relvantesten, um Kubernetes-Cluster abzubilden und zu simulieren. Kubernetes, als Plattform mit verschiedenen Versionen, ist sicherlich nie vollständig abbildbar, da es beliebig erweiterbar ist, jedoch konnte für die Grundkonzepte eine Abbildung gefunden werden. Die Metrik 1.2.1 „Übersicht der abgebildeten Konstrukte im Vergleich zu den wichtigsten Grundkonzepten aus Kubernetes. Anteil berechnen.“ kann somit mit 91,7% beantwortet werden, wobei dieser Anteil sich nur auf die in dieser Arbeit betrachteten Konzepte bezieht, welche jedoch eine gute Abbildung ermöglichen. Für Konstrukte, wie die Control Plane Components, welche Entscheidungen über das Cluster treffen, beziehungsweise Änderungen vornehmen, wurde beispielsweise keine Abbildung gesucht. Stattdessen sind diese, wie der key-value store etcd, der alle Cluster Daten enthält, implizit durch das statische Modell abgebildet, welches ein Cluster in Palladio beschreibt. Im Rahmen dieser Arbeit wurden solche Komponenten nicht explizit ausgewiesen.

| Kubernetes-Konstrukt | Abgebildet | Neues Modellelement | Siehe Kapitel |
|---|---|---|---|
| Cluster | Ja | Nein | 3.1.3 |
| Master-Node | Ja | Ja | 3.1.3 |
| Worker-Node | Ja | Ja | 3.1.3 |
| Pod | Ja | Ja | 3.1.1 |
| Container | Ja | Ja | 3.1.1 |
| Requests | Ja | Ja | 3.1.1 |
| Limits | Ja | Ja | 3.1.1 |
| Service | Ja | Nein | 3.1.1 |
| Ingress | Ja | Nein | 3.1.1 |
| Deployment | Ja | Ja | 3.1.3 |
| Replica Set | Nein | Nein | - |
| Kubernetes Pod Scheduler | Ja | Nein (Palladio Plug-In) | 5 |

Tabelle 8.1: Übersicht der abgebildeten Kubernetes-Konstrukte

Zur Verwendung von Metrik 1.2.2 wird erneut ein Szenario definiert, in welchem ein Kubernetes-Cluster-Zustand beschrieben wird. Dabei handelt es sich um einen Schnappschuss eines bereits laufenden Clusters mit stattgefundener Allokation von Pods. Dieser wird dann mithilfe der Kubernetes-Erweiterung versucht abzubilden um zu überprüfen ob es Differenzen zwischen dem definierten Cluster-Zustand und dem Modell gibt. Das Szenario basiert erneut auf dem Cacheless-MediaStore aus den Screencasts [32] und ist als Fortsetzung des Szenarios auf Seite 62 zu sehen.

**Szenario:** Nachdem der Nutzer die Anwendung containerisiert hat, deployed er diese auf einem Kubernetes-Cluster. Das Cluster wird ihm zur Verfügung gestellt und besteht dabei aus drei Kubernetes-Nodes. Eine davon ist eine Master-Node, auf welcher keine Pods der Anwendung deployed werden. Die beiden anderen sind Worker-Nodes. Die Ressourcenspezifikationen der Nodes sowie die Cluster-Struktur sind in Abbildung 8.3 abgebildet. Der Nutzer beschließt, nur Single-Container-Pods zu verwenden, also für jeden der bereits definierten Container einen eigenen Pod zu erstellen. Er erstellt hierfür drei Deployments. Er beschließt pro angebotener Rolle der Container, und damit Pods, eigene Services zu erstellen. Dabei entscheidet er sich, für den IMediaManagement-Service, welcher die Verbindung der System-Schnittstelle nach außen darstellt, einen NodePort-Service zu verwenden. Die anderen Services für IDownload, IMediaAccess und IDataStorage werden als ClusterIP-Services definiert. Für IPackaging erstellt er keinen Service, da diese Rolle nur innerhalb eines Containers verwendet wird und der Container diese auch nicht anbietet, siehe Abbildung 8.2. Für die Angabe der benötigten Instanzen entscheidet er sich einen MediaManagementPackaging-Pod zu erstellen, einen MediaAccess-Pod und zwei FileStorage-Pods. Die Requests und Limits für die Deployment-Spezifikation übernimmt er aus den bereits definierten Requests und Limits der Container, siehe Abbildung 8.1. Dann deployed er das Ganze auf dem Cluster und erhält einen Cluster-Zustand, welcher in Abbildung 8.3 abgebildet ist. Ihm fällt ein, dass er nun die Performance überprüfen möchte, hat aber keine Möglichkeit, Anfragen an das System zu senden. Jedoch erinnert er sich an den Palladio-Simulator, mit welchem er jetzt versucht, den Zustand des Clu-

sters statisch abzubilden, da mit diesem Anfragen einfach simuliert werden können. Er verwendet hierfür den PCM-Kubernetes-Workflow, siehe Kapitel 4, wobei er Teile davon bereits umgesetzt hat oder diese schon aus dem Palladio-Modell der Screencasts vorliegen. So sind die *Interfaces* der Komponenten zusammen mit den *Basic Components* und ihren *Resource Demanding SEFF* schon in Palladio abgebildet. Ebenso hat er bereits die Container spezifiziert, siehe Abbildung 8.2. Nach dem Abarbeiten des Workflows liegt ein Cluster als *Resource Environment Model* siehe Abbildung 8.5, ein *System Model* mit den spezifizierten Deployments, den Pod-Instanzen in *Assembly Contexten*, sowie den Service-Komponenten vor, siehe Abbildung 8.4. Ebenso liegt ein *Alloaction Model* vor. Dieses ist in Abbildung 8.5 direkt auf dem *Resource Environment Model* graphisch für ein besseres Verständnis abgebildet.
Im Folgenden wird der gewünschte Zustand des Clusters, siehe Abbildung 8.3, mit den aus dem Szenario entstandenen Modellen verglichen. In der Abbildung 8.3 ist nicht nur das Cluster, sondern auch die bereits requesteten Ressourcen der Nodes durch die allokierten Pods mithilfe von Balken beschrieben. Der rote Anteil bedeutet, dass Ressourcen belegt sind. Der grüne Anteil bedeutet, dass diese noch requested werden können.

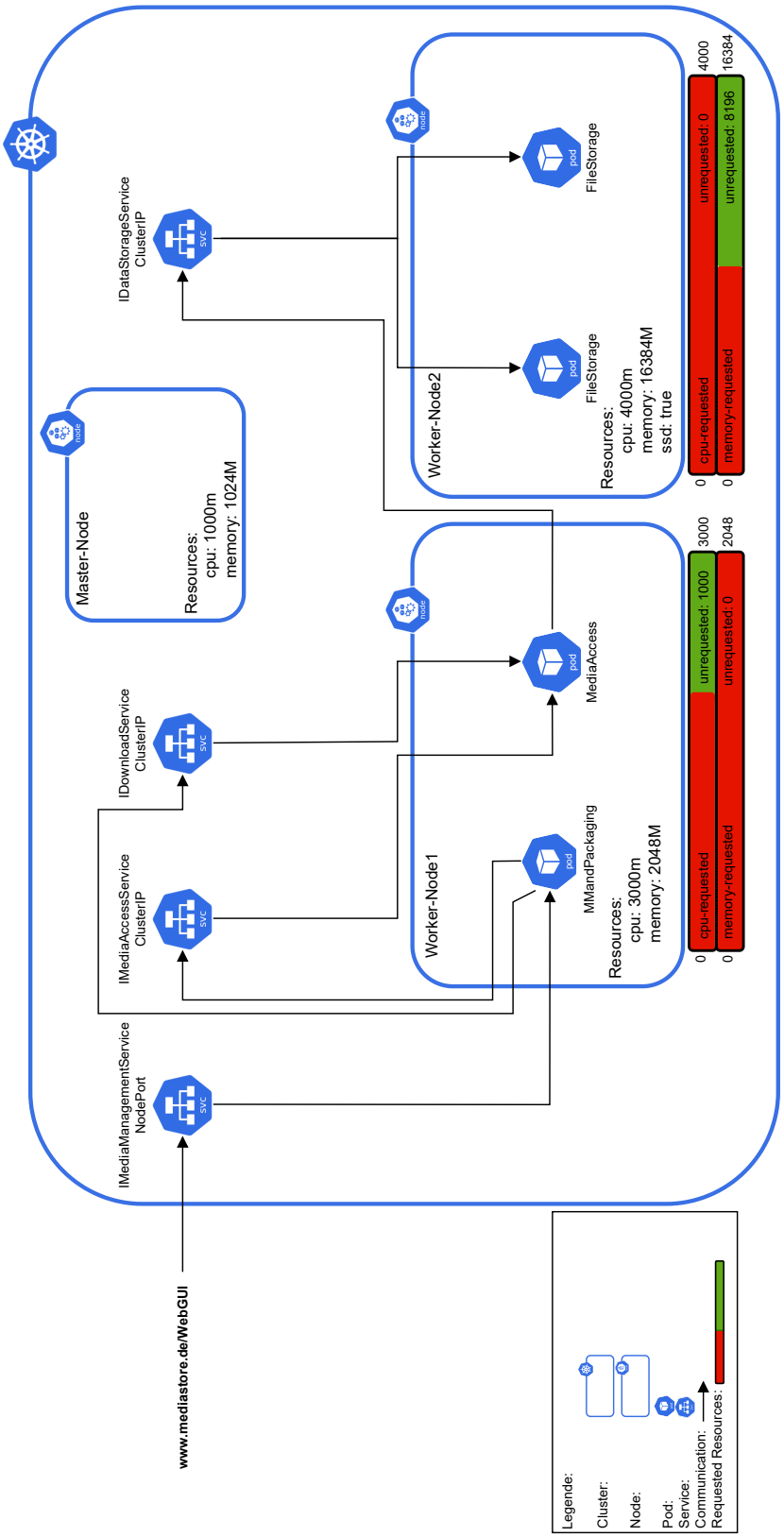


Abbildung 8.3: Cluster-Zustand-Snapshot des MediaStores

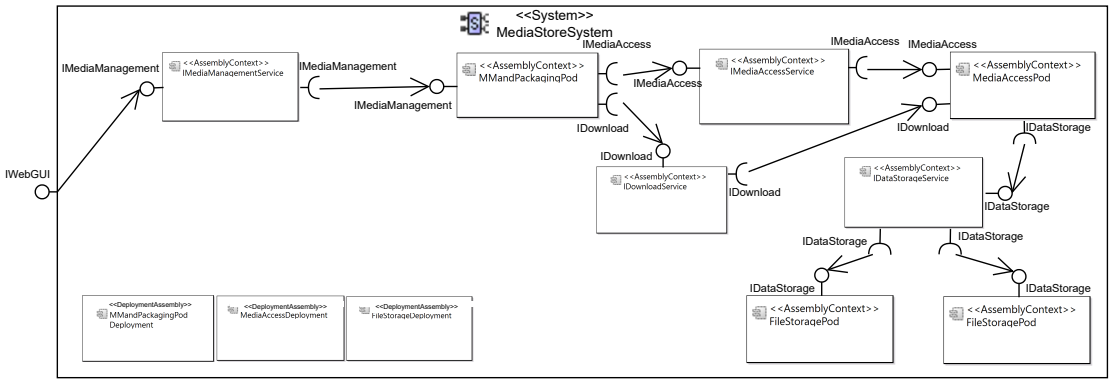


Abbildung 8.4: System Model abgeleitet aus Evaluationsszenario, erstellt mithilfe des PCM-Kubernetes-Workflows

Vergleicht man die Abbildungen der Modelle beziehungsweise die Modelle aus Abbildung 8.4 und Abbildung 8.5 mit dem definierten Cluster-Zustand im Szenario beziehungsweise in Abbildung 8.3, so stellt man fest, dass jedes erwähnte Kubernetes-Konstrukt abgebildet wurde. Für die Metrik kann man dennoch sagen, dass der Speicher, sowie das Vorhandensein einer SSD, wie es für die FileStorage-Pods gewünscht war. nur in den Requests abgebildet war. Die Kubernetes-Node im Modell hatte keine SSD, weder als *Processing Resource* noch als anderes Attribut. Auch die Memory-Spezifikation, wie sie im Cluster spezifiziert wurde, wurde im *Resource Environment* nicht abgebildet. Diese Unterschiede liegen daran, dass im Modell Kubernetes-Nodes nicht mit weiteren Spezifikationen als der Angabe von Speicher und CPU ausgestattet werden können. Eine Erweiterung um ein allgemeines Spezifikationsmodell der Nodes zur Umsetzung benutzerdefinierter Ressourcen würde diesen Unterschied beheben. Alternativ könnte ebenso eine Liste von Strings als weiteres Attribut diesen Teil übernehmen. Für das Abbilden von Hauptspeicher in den *Resource Containern* liegt zurzeit keine *Processing Resource* vor. Dadurch kann man die Metrik 1.2.2 mit *2* beantworten. Die Question 1.2 ist mithilfe der Tabelle 8.1 sowie dem umgesetzten Szenario somit folgendermaßen beantwortet: *Es können das Cluster, Master-Nodes, Worke- Nodes, Pods, Container, Requests und Limits, Services, Ingress, und Deployments abgebildet werden.* Das erstellte Modell kann unter [8] heruntergeladen werden.

**Question 1.3** Diese Frage kann, nach den Erkenntnissen aus der Beantwortung der Questions 1.1 und 1.2 mit *Ja* beantwortet werden. Die für diese Arbeit wichtigen Informationen in einem Deployment sind die Pod-Referenz sowie die Angabe von Requests und Limits. Beides wurde im MediaStore Modell erstellt. Die erstellten Deployments wurden als DeploymentAssembly im *System Model*, siehe Abbildung 8.4 assembliert und beschreiben, wie viele Instanzen der Pods erstellt werden sollen. Diese Spezifikationen wurden im MediaStore Modell aus dem Szenario umgesetzt: FileStorage mit zwei Replicas, sowie die anderen Pods mit jeweils einem. Für die Metrik 1.3.1 gilt: Abweichungen konnten für den Rahmen dieser Arbeit nicht gefunden werden.

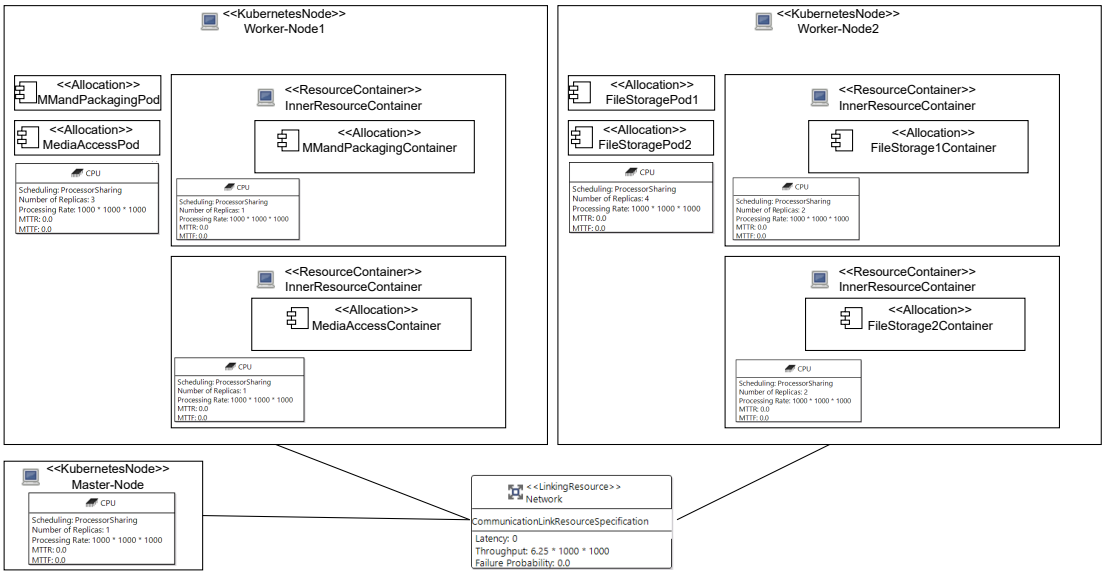


Abbildung 8.5: Resource Environment des Evaluationsszenarios mit Alloaction Model graphisch kombiniert

**Goal 1:** Der in diesem Kapitel verwendete GQM-Plan konnte somit erfolgreich bearbeitet werden, weshalb Goal 1 als erreicht anzusehen ist. Mithilfe dieser Evaluation und der in dieser Arbeit erstellten Modellerweiterung konnten auf die Forschungsfragen 1, 2 und 3 Antworten gefunden werden. Für den zweiten Teil der Forschungsfrage 3 muss erwähnt werden, dass in diesem Kapitel nur die Abbildung beschrieben wurde. Die Simulation eines statischen Deployments, wie es im Szenario als Schnappschuss eines Cluster-Zustands definiert wurde ist jedoch mit Palladio mithilfe eines nach dem PCM-Kubernetes-Workflow definierten Modells ebenso möglich.

## 8.2 GQM Pod Allokation

**Goal 2:** Nachbildung der Kubernetes Pod-Allokierung in Palladio.
**Adressierte Forschungsfragen:** 2, 3, 4

**Question 2.1:** Erzeugt der PCM-Pod-Scheduler eine gültige Allokation der Pods auf Kubernetes-Nodes?

**Metric 2.1.1:** Überprüfung auf Korrektheit des Scheduling-Ergebnisses durch Vergleich mit erwartetem Ergebnis.

**Question 2.2:** Welche Veränderungen müssen an einem Modell geschehen um einen nicht-geschedulten Pod im Modell korrekt zur Simulationszeit zu allokieren?

**Metric 2.2.1:** Auflistung der Schritte und Bewertung der Umsetzbarkeit in einer Reconfiguration.

Mithilfe von Kubernetes werden Systeme deklarativ beschrieben. Dabei ist die Allokation der Pods auf den Kubernetes-Nodes nicht bekannt. Bei der Ausführung eines Deployments werden die darin spezifizierten Pods im Cluster deployed. Dabei bestimmt Kubernetes, wo die Pods allokiert werden. Diese Allokation muss bei einer dynamischen Simulation in Palladio ebenso umgesetzt werden können, weshalb Goal 2 definiert wurde. Mithilfe eines Palladio-Plugins wurde deshalb ein Scheduler prototypisch implementiert, welcher auf Basis des *System Models* und des aktuellen *Allocation Models* eine Empfehlung für die Allokation eines nicht-allokierten Pods gibt. Einzelteile der Implementierung wurden mithilfe von Unit-Tests getestet. Zusätzlich wurden zur Evaluation des Schedulers auf Basis des Szenarios von Seite 65 ein Modell erstellt, welches den Scheduler in vier Schritten testet, welches unter [9] zusammen mit dem AllocationScheduler-Code heruntergeladen werden kann. Für dieses Modell wurde entschieden, dass die drei Deployments aus dem Szenario in der Reihenfolge: MMandPackagingPodDeployment -> MediaAccessDeployment -> FileStorageDeployment deployed werden. Dabei wird ein Pod immer vollständig allokiert, bevor die nächste Scheduler-Entscheidung getroffen wird. Daraus resultieren vier *System*- bzw. *Allocation*-Modelle, die den Zustand vor jedem einzelnen Schritt beschreiben. Nachdem der Scheduler ein Ergebnis zurückgegeben hat, wird dieser Pod allokiert, woraus das *Allocation Model* zur Eingabe in den nächsten Schritt entsteht. Für das *System Model* werden lediglich die Pods der Deployments nacheinander instanziiert. Die Abbildung 8.6 bildet Schritt für Schritt die Allokation der Pods bei Ausführung der Deployments in der angegebenen Reihenfolge ab.

**Question 2.1** Die Question 2.1 „Erzeugt der PCM-Pod-Scheduler eine gültige Allokation der Pods auf Kubernetes- Nodes?“ kann nach der Durchführung des Tests der Allokation mit einem *Ja* beantwortet werden, denn die in Abbildung 8.6 gezeigte Allokation der Pods ist ein gültiges Scheduling-Ergebnis. Das Endergebnis entspricht auch dem erwarteten Ergebnis, dass alle 4 Pods eine Allokation im Cluster finden.

**Question 2.2** Diese Frage wird in Abbildung 6.1 beantwortet. Auf der linken Seite der Abbildung sind alle Schritte aufgelistet. Die Metric 2.2.1 kann nicht abschließend angewendet werden, da keine Reconfiguration implementiert wurde, die diese Schritte umsetzt. Aus den Ergebnissen der prototypischen Entwicklung des Allocation-Schedulers, sowie den Möglichkeiten, die Reconfigurations mit QVTo bieten, ist aber stark anzunehmen, dass eine Umsetzung dieser Schritte vergleichsweise leicht sein sollte.

## 8.3 GQM Kontrollschleifen in Kubernetes und dynamische Simulation

**Goal 3:** Abbildung der Kontrollschleifen von Kubernetes in Palladio zur Vorhersage der Performance von dynamischen Deployments.
**Adressierte Forschungsfragen:** 2, 3, 4

**Question 3.1:** Wie lässt sich eine Skalierung (HPA) und Rollouts simulieren?

**Metric 3.1.1:** Bewertung der Umsetzbarkeit mithilfe von Reconfigurations.

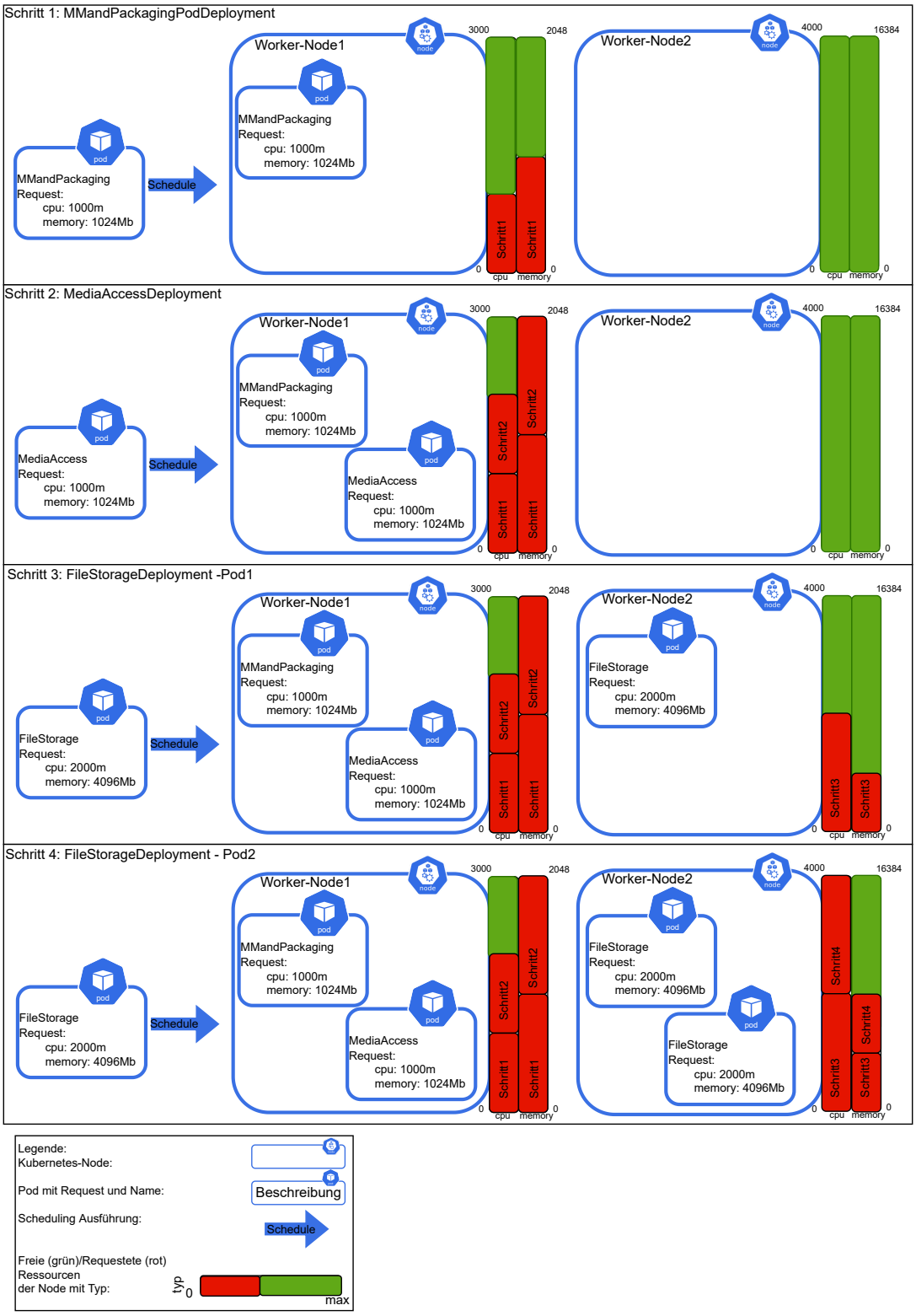


Abbildung 8.6: Ressourcenreservierung und Allokation der Worker-Nodes Schritt-für-Schritt

Um dynamische containerisierte Software-Architekturen mithilfe von Palladio zu simulieren, müssen Kontrollschleifen, die den Cluster-Zustand überwachen und bei Bedarf Änderungen am Zustand vornehmen, entworfen und implementiert werden. Im Rahmen der vorliegenden Arbeit wurde ein Simulationskonzept entwickelt, welches grundlegende Kontrollschleifen die Kubernetes hat, umsetzt. Diese Kontrollschleifen können auf verschiedene Arten implementiert werden, um sie in die Simulation aufzunehmen.

**Question 3.1** Die Question 3.1 lässt sich mithilfe des Simulationskonzepts beantworten, siehe Kapitel 6. Eine Skalierung eines HPA stellt lediglich eine Veränderung der Anzahl der Replicas eines Deployments dar. Diese Veränderung würde von der definierten Kontrollschleife erkannt und notwendige Änderungen an den Modellen gestartet werden. Ein Rollout eines Updates, also einer neuen Version von Pods kann durch das Ersetzen eines Deployments, durch ein anderes realisiert werden. Durch das Fehlen des ursprünglichen Deployments werden die dazugehörigen Pods entfernt. Durch das neue Deployment mit der neuen Version der Pods wird von der Kontrollschleife eine Differenz zwischen Ist- und Soll-Zustand festgestellt. Die notwendigen Änderungen werden dann ebenso gestartet. Die technische Umsetzbarkeit wurde bereits in Abschnitt 6.1 beschrieben und kann deshalb entsprechend als realisierbar angenommen werden. Sollten die Möglichkeiten, die QVTo anbietet nicht ausreichen, so besteht auch die Möglchkeit weitere Teile in Black-Box-Methoden auszulagern oder alles in Java umzusetzen.

## 8.4 Evaluation des Systems in Bezug auf Forschungsfrage 5

Die Forschungsfrage 5: *Wie lassen sich Auswirkungen von Throtteling und Resource-Sharing schon zur Design Zeit vorhersagen?* wurde im Rahmen dieser Arbeit nicht explizit bearbeitet. Stattdessen wurde eine Erweiterung des PCM entwickelt, die die Definition von Ressourcen-Einschränkungen für Container erlaubt. Die Erweiterung ist so definiert, dass sie mit dem bereits implementierten Nested Resource Container Scheduler [10] kompatibel ist. Dieser setzt Scheduling inspiriert von cgroups um und berücksichtigt bereits Requests von CPU bei der Spezifikation und Simulation von Containern. Diese Berücksichtigung und gegebenfalls Drosselung einzelner Container, die sich die Ressourcen einer Kubernetes-Node teilen, lässt sich also mithilfe des Modells bei entsprechender Spezifikation in die Simulation mit einbeziehen. Es wurde jedoch keine ausführliche Evaluation für eine konkrete Beantwortung dieser Forschungsfrage durchgeführt und auch der Nested Resource Container Scheduler wurde nicht in aller Ausführlichkeit analysiert. Dadurch kann diese Frage nicht abschließend beantwortet werden. Auf Basis der vorliegenden Arbeit und mithilfe des Nested Resource Container Schedulers beziehungsweise einer möglichen Erweiterung dieses Schedulers, kann aber an der Beantwortung dieser Frage weitergearbeitet werden.

Für die Beantwortung der anderen Forschungsfragen kann gesagt werden, dass diese Arbeit Frage 1 und 2 (Abbildung von Containern bzw. Kubernetes-Konstrukte) ausführlich beantwortet. Auch für Forschungsfrage 3: *Wie lassen sich nicht statische Deployments in Palladio abbilden und simulieren?* konnte ein Ergebnis erarbeitet und dieses evaluiert werden. Forschungsfrage 4: *Wie lassen sich Skalierung HPA und Rollouts simulieren?* wurde

mithilfe des dynamischen Simulationskonzepts bearbeitet. Die Ergebnisse dieser Arbeit konnten sinnvoll und mit positivem Ergebnis evaluiert werden.

# 9 Fazit

In dieser Arbeit wurde eine Abbildung für containerisierte Systeme mit Palladio entwickelt. Zusätzlich wurden grundlegende Kubernetes-Konstrukte zur deklarativen Beschreibung eines Deployments und als Vorbereitung für dynamische Simulation containerisierter Software mit Palladio konzipiert und implementiert. Desweiteren wurde ein Pod-Allokations-Scheduler entwickelt und prototypisch implementiert, der auf Basis eines Palladio-Modell-Zustands identifiziert, ob ein Pod geschedulet werden muss oder kann. Für diesen wird dann eine Allokation bestimmt. Um die Verwendung der Kubernetes-Erweiterung in Palladio zu definieren und die Struktur, sowie die Zusammenhänge zwischen den implementierten Konzepten darzustellen wurde ein Workflow entwickelt. Mit diesem Workflow lassen sich sowohl bereits definierte Kubernetes-Cluster mithilfe von Palladio abbilden, als auch Software-Systeme, die von einem klassischen Deployment migriert werden. Auch für die Verwendung einer neu definierten Software eignet sich diese Vorgehensweise um die Containerisierung darzustellen und mögliche Deployments mit Kubernetes abzubilden. Ziel dieser Arbeit war ebenso die Entwicklung eines dynamischen Simulationskonzeptes für deklarativ beschriebene Deployments mit Palladio. Als Teil dieses Simulationskonzepts wurde eine Kontrollschleife konzeptioniert, die verschiedene Kontrollschleifen aus Kubernetes vereint und beschreibt, welche Veränderungen an PCM-Modellen vorgenommen werden müssen, wenn sich dynamische Änderungen in den Modellen ergeben. Szenarien die dabei berücksichtigt wurden, sind unter anderem, das Terminieren eines Pods, das Skalieren von Deployments HPA, sowie das Erstellen und Entfernen von Deployments zur Simulationszeit. Auch für die Abbildung derartiger Szenarien wurden Vorschläge zur Implementierung dieser erarbeitet. Abschließend wurden die erarbeiteten Konzepte und Implementierungen evaluiert. Die fünf in der Einleitung definierten Forschungsfragen konnten im Rahmen dieser Arbeit entweder vollständig oder teilweise beantwortet werden.

## 9.1 Künftige Arbeiten

Als künfitge Arbeiten können verschiedene Bereiche, die in dieser Arbeit betrachtet wurden erweitert, verbessert oder weiter evaluiert werden. Eine Möglichkeit die die Kubernetes-Modellerweiterung noch erweitern würde, wäre das ausführlicherere Modellieren von benutzerdefinierten Ressourcen, sowohl bei der Angabe von Requests und Limits, als auch bei der Spezifikation der Pods. Damit einhergehend bietet es sich an, weitere Pod-Allokations-Scheduler zu entwickeln und zu implementieren. In Kubernetes sind die meisten Konstrukte austauschbar, so auch der Scheduler und je nach Anwendungsfall kann es von Interesse sein, weitere spezielle Ressourcen abzubilden und in der Allokation berücksichtigen zu können.
Um die Nutzbarkeit der Erweiterung und des Schedulers zu verbessern, bietet es sich

an, eine Reconfiguration umzusetzen, die auf Basis des in dieser Arbeit prototypisch implementierten Allokations-Schedulers, deklarativ spezifizierte Deployments im PCM voll-automatisch assembliert, sowie die Allokation vornimmt. Als Erweiterung dieser Verbesserung der Nutzbarkeit, kann auf dem dynamischen Simulationskonzept beruhend, oder erweiternd, eine Reconfiguration umgesetzt werden, mit welcher Hilfe eine Skalierung der Deployments möglich ist. Ähnliche Ansätze wurden bereits in [37] umgesetzt.
Als weiterer Punkt, der sich insbesondere mit der nicht vollständig beantworteten Forschungsfrage 5 beschäftigt, wäre eine Evaluation der Modellerweiterung mithilfe des Nested Resource Container Schedulers [10]. Eine Erweiterung dieses Schedulers in Kombination mit einer Umsetzung der Berücksichtigung von RAM im Scheduling, beispielsweise mit passiven Ressourcen wäre ebenso denkbar. Abschließend könnte eine ausführliche Nutzerstudie zur Evaluation des PCM-Kubernetes-Workflows zur Verbesserung der Nutzbarkeit der Erweiterung beitragen.

# Literatur


[1] Thorsten Arendt u. a. „Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations“. In: *Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part* (2010). URL: https://link.springer.com/content/pdf/10.1007/978-3-642-16145-2.pdf.

[2] John Arundel und Justin Domingus, Hrsg. *Cloud Native DevOps with Kubernetes*. Englisch. O'Reilly Media, Inc., 2019. 344 S. ISBN: 9781492040767.

[3] *AWS. Amazon Elastik Kubernetes Service (EKS)*. 2022. URL: https://aws.amazon.com/de/eks/ (besucht am 09. 05. 2022).

[4] Jonathan Baier, Gigi Sayfan und Jesse White, Hrsg. *The Complete Kubernetes Guide*. Englisch. Packt Publishing, 2019. 616 S. ISBN: 9781838647346.

[5] Niko Benkler. „SimulationAutomation - Palladio Simulation mit Kubernetes“. In: *github*
Commit: 316bcc111ac08ce8b36d9e65ff9234c158e738e4 (2020). URL: https://github.com/Benkler/SimulationAutomation.

[6] Federico Ghirardini u. a. „Model-Driven Simulation for Performance Engineering of Kubernetes-Style Cloud Cluster Architectures“. In: *Advances in Service-Oriented and Cloud Computing*. Bd. 1. 2018, S. 7–20. DOI: https://doi.org/10.1007/978-3-030-63161-1.

[7] *Google Cloud. Google Kubernetes Engine*. 2022. URL: https://cloud.google.com/kubernetes-engine?hl=de (besucht am 09. 05. 2022).

[8] Nathan Hagel. „Example Model from Evaluation GQM - Goal 1“. In: *GitHub* (9.05.2022). URL: https://github.com/Yabbies/PCMKubernetesExampleModels.

[9] Nathan Hagel. „PCM-K8s-Allocation Scheduler with tests and evaluation model“. In: *GitHub* (9.05.2022). URL: https://github.com/Yabbies/org.palladiosimulatorg.kubernetes.allocationscheduler.

[10] Jörg Henß. „Mosaic Container Scheduler“. In: *GitHub*
Commit: 27e5a22770bb77eb3b7fe3f32b19ee856a726a3d (3.05.2022). URL: https://github.com/PalladioSimulator/Palladio-Simulation-Scheduler/tree/mosaic-container-scheduler.

[11] Tejun Heo. „Control Group v2“. In: (1.05.2022). URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.

[12] *Kubernetes Documentation. Deployments*. 2022. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (besucht am 08. 04. 2022).

[13] *Kubernetes Documentation. Replica Set.* 2022. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (besucht am 07. 04. 2022).

[14] *Kubernetes Documentation. kube-controller-manager.* 2022. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/ (besucht am 09. 05. 2022).

[15] *Kubernetes Documentation. Was ist Kubernetes.* 2022. URL: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/ (besucht am 27. 04. 2022).

[16] *Kubernetes Documentation. Kubernetes Pods.* 2022. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (besucht am 05. 01. 2022).

[17] *Kubernetes Documentation. Kubernetes Resource Management for Pods and Containers.* 2022. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ (besucht am 05. 01. 2022).

[18] *Kubernetes Documentation. Containers.* 2022. URL: https://kubernetes.io/docs/concepts/containers/ (besucht am 27. 04. 2022).

[19] *Kubernetes Documentation. Resource units in Kubernetes.* 2022. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#resource-units-in-kubernetes (besucht am 05. 01. 2022).

[20] *Kubernetes Documentation. Kubernetes Scheduler.* 2022. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/ (besucht am 06. 01. 2022).

[21] *Kubernetes Documentation. Create a Pod that gets assigned a QoS class of Burstable.* 2022. URL: https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/#create-a-pod-that-gets-assigned-a-qos-class-of-burstable (besucht am 27. 01. 2022).

[22] *Kubernetes Documentation. Service.* 2022. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (besucht am 25. 04. 2022).

[23] *Kubernetes Documentation. Container Runtimes.* 2022. URL: https://kubernetes.io/docs/setup/production-environment/container-runtimes/ (besucht am 01. 05. 2022).

[24] *Kubernetes Documentation. CPU units.* 2022. URL: https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/#cpu-units (besucht am 02. 05. 2022).

[25] *Kubernetes Documentation v1.19. Ingress.* 2022. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/ (besucht am 09. 05. 2022).

[26] Palladio Component Model. *Palladio Source Code Documentation - EntryLevelSystemCall.class from pcm.ecore.*
Commit: cb6bf866ef391b2ed3fe737612775932d222a33f. 1.03.2022.

[27] Palladio Component Model. *Palladio Source Code Documentation - ResourceDemandingBehavior.class from pcm.ecore.*
Commit: cb6bf866ef391b2ed3fe737612775932d222a33f. 25.02.2022.

[28] Palladio Component Model. *Palladio Source Code Documentation - SubSystem.class from pcm.ecore*.
Commit: cb6bf866ef391b2ed3fe737612775932d222a33f. 23.02.2022.

[29] Palladio Component Model. *Palladio Source Code Documentation - System.class from pcm.ecore*.
Commit: cb6bf866ef391b2ed3fe737612775932d222a33f. 23.02.2022.

[30] Palladio Component Model. *Palladio Source Code Documentation - UsageModel.class from pcm.ecore*.
Commit: cb6bf866ef391b2ed3fe737612775932d222a33f. 1.03.2022.

[31] Palladio Component Model. *Palladio Source Code Documentation - Workload.class from pcm.ecore*.
Commit: cb6bf866ef391b2ed3fe737612775932d222a33f. 1.03.2022.

[32] Palladio. „Palladio Simulator Overview - Screencast“. In: *Palladio* (29.04.2022). URL: https://www.palladio-simulator.com/tools/screencasts/.

[33] Ralf Reussner. „Softwaretechnik II Software Components“. In: (11.12.2018), S. 43–65. URL: https://ilias.studium.kit.edu/goto.php?target=file_1422209_download&client_id=produktiv.

[34] Ralf Reussner u. a., Hrsg. *Modeling and Simulating Software Architectures – The Palladio Approach*. Englisch. MIT Press, 2016. 408 S. ISBN: 978-0-262-03476-0.

[35] Noorhan Saleh und Maggie Mashaly. „A Dynamic Simulation Environment for Container-based Cloud Data Centers using ContainerCloudSim“. In: *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*. 2019, S. 332–336. DOI: 10.1109/ICICIS46948.2019.9014697.

[36] Christian Pikalek und SOPHIST. „Besondere Business Rules: Integritätsregeln“. In: *SOPHIST* (14.04.2010). URL: https://blog.sophist.de/2010/04/14/besondere-business-rules-integritatsregeln/.

[37] Christian Stier. *Adaptation-aware architecture modeling and analysis of energy efficiency for software systems*. Karlsruhe, [2019]. URL: https://doi.org/10.5445/KSP/1000086089http://dx.doi.org/10.5445/KSP/1000086089%20;%20https://nbn-resolving.org/urn:nbn:de:0072-860895%20;%20http://d-nb.info/1181185602/34%20;%20http://www.ksp.kit.edu.

[38] Zhiheng Sun, Liper und Qizhen Weng. „alibaba - open-simulator“. In: *GitHub - Alibaba Open Source*
Commit: a706f8dfb538e5d9a8a3c10170f72d8c958cb4eb (9.05.2022). URL: https://github.com/alibaba/open-simulator.

[39] Bimlesh Wadhwa, Aditi Jaitly und Bharti Suri. „Cloud Service Brokers: An Emerging Trend in Cloud Adoption and Migration“. In: *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*. Bd. 2. 2013, S. 140–145. DOI: 10.1109/APSEC.2013.129.

[40] Hidehito Yabuuchi, Daisuke Taniwaki und Shingo Omura. „k8s-cluster-simulator: A simulator for evaluating Kubernetes schedulers“. In: *Preferred Networks Research & Developement Blog* (9.05.2022). URL: `https://tech.preferred.jp/en/blog/k8s-cluster-simulator-release/`.

# Abkürzungsverzeichnis

**AWS** Amazon Web Services.

**cgroups** control groups.

**DCSK** dynamisches Container Simulationskonzept.

**DSL** Domain Specific Language.

**GCP** Google Cloud Platform.

**GQM** Goal-Question-Metric / Ziel-Frage-Metrik.

**HPA** Horizontaler Pod Autoskalierer.

**OOM** Out-of-Memory.

**PCM** Palladio Component Model.

**QoS** Quality of Service.

**QVTo** QVT Operational.

**SEFF** Service Effect Specification.

**VM** Virtual Machine.