Fachbereich
Elektrotechnik und Informatik

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Bachelorarbeit

Autoscaling für elastische Container Plattformen

Christian-Friedrich Stüben

Ausgabedatum: 22. Juni 2017
Abgabedatum: 22. September 2017

(Prof. Dr. rer. nat. habil Hanemann)
Vorsitzender des Prüfungsausschusses

Aufgabenstellung

In dieser Bachelorarbeit sollen existierende Open Source Monitoring Lösungen (wie z.B. der Elastic Stack, Prometheus, etc.) analysiert und mit einer bestehenden Deployer-Lösung für elastische Containerplattformen (wie Kubernetes, Apache Mesos, Docker Swarm, Nomad, etc.), zu einem Auto-Scaler integriert werden. Eine Realisierung dieses Deployers wurde bereits für Kubernetes auf Amazon AWS[1] und OpenStack umgesetzt. Es sollen Metriken der Cluster-Nodes erfasst, zusammengeführt und analysiert werden. Die Bestimmung geeigneter Metriken für Skalierungsmaßnahmen gehört nicht zum Umfang dieser Arbeit. Die Arbeit soll jedoch nachweisen, dass beliebig erfassbare Metriken (und Metrikkombinationen) für eine Autoskalierung genutzt werden können. Im Detail müssen folgende Teilaufgaben bearbeitet und dokumentiert (Bachelorarbeit) werden:

Analyse von bestehenden Monitoring-Lösungen
Es gibt diverse Tools und Tool-Stacks mit denen das Monitoring von Cluster-Nodes erfolgen kann

- Mittels einer Recherche sollen geeignete (mind. drei) Monitoring-Systeme/Stacks identifiziert und verglichen werden.
- Die Kriterien für eine vergleichende Analyse sollen auf dem Funktionsumfang (unterstützt Metriken) und der Skalierbarkeit beruhen. Ausschusskriterien sind Dienste die nur als externer Service zur Verfügung stehen und Tools unter nicht-freier Lizenz.
- Weitere Kriterien, besonders solche, die für die Umsetzung und Integration mit dem vorhandenen ECP-Deployer zusammenhängen, sollen in Absprache selbstständig identifiziert werden.

Automatisches Skalieren des Clusters (Integration)
Aktuell ist ein Deployer für Kubernetes Cluster vorhanden und für andere Plattformen, wie beispielsweise Apache Mesos und Docker Swarm, geplant. Der bestehende Deployer soll um eine Monitoring- und Skalierungs-Komponente ergänzt werden.

- Entsprechend der vergleichenden Analyse soll das geeignetste Monitoring-System installiert und integriert werden.

[1] Amazon Web Services

- Es soll ein System in Ruby umgesetzt werden, welches anhand der Daten aus dem Monitoring-System und der übergebenen Metriken die Anzahl der Cluster-Nodes nach Bedarf anpasst (Auto-Skalierung).
- Dazu muss die Cluster-Konfiguration angepasst werden und der ECP-Deployer geeignet getriggert werden.
- Es sind nach Möglichkeit einfache Datenformate wie bspw. JSON zu nutzen.

Evaluierung der Skalierbarkeit des Clusters
Anhand von mehreren erfassbaren Metriken/Metrikkombinationen soll das Skalieren des Clusters exemplarisch nachgewiesen werden.

- Es muss gezeigt werden, dass Systemzustände der Cluster-Nodes (bspw. CPU-Load, Arbeitsspeicher, Plattenplatz, Netzwerklast, etc.) erkannt und in sinnvolle Skalierungsaktionen umgesetzt werden können.
- Dazu müssen geeignete Tools zur Erzeugung von Lasten identifiziert oder ggf. erstellt werden.
- Es muss getestet werden, dass das System entsprechend der überwachten Metriken selbstständig skaliert.
- Es soll horizontales und automatisches Hoch- und Runterskalieren getestet werden.

Fachbereich Elektrotechnik und Informatik
Dekanat - Prüfungsausschuss

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Erklärung zur Bachelorarbeit

Ich versichere, dass ich die Arbeit selbständig, ohne fremde Hilfe verfasst habe.

Bei der Abfassung der Arbeit sind nur die angegebenen Quellen benutzt worden. Wörtlich oder dem Sinne nach entnommene Stellen sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, insbesondere dass die Arbeit Dritten zur Einsichtnahme vorgelegt oder Kopien der Arbeit zur Weitergabe an Dritte angefertigt werden.

Lübeck, den 14. September 2017

.....

(Unterschrift)

Zusammenfassung der Arbeit / Abstract of Thesis

Fachbereich: **Department:**	Elektrotechnik und Informatik
Studiengang: **University course:**	Informatik/Softwaretechnik
Thema: **Subject:**	Autoscaling für elastische Container Plattformen
Zusammenfassung:	Die vorliegende Arbeit untersuchte die Tauglichkeit von Monitoring Lösungen und beschreibt eine Machbarkeitsstudie zur automatischen Skalierung von elastischen Container Plattformen auf Basis virtueller Maschinen.
Abstract:	The following work is an evaluation of monitoring tools. Based on these results a software for autoscaling of elastic container platforms has been developed.
Verfasser: **Author:**	Christian-Friedrich Stüben
Betreuender ProfessorIn: **Attending professor:**	Prof. Dr. Nane Kratzke
WS / SS:	WS 2017/2018

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel werden die Hintergründe dieser Arbeit beleuchtet und anschließend die gestellte Aufgabe in Kontext zu den Hintergründen gebracht. Dies dient vor allem dazu Begriffe und Definitionen eindeutig fest zu halten.

1.1 Hintergrund

Bei modernen Software Architekturen ist es häufig, auf Grund globaler Verfügbarkeit, nötig Anwendungen horizontal skalierbar zu gestalten. Die klassischen monolithischen Systeme führen bei horizontaler Skalierung dazu, dass sämtliche Komponenten einer Software dupliziert werden müssen. Da hierbei auch Teile der Software erweitert werden, die nicht zwangsweise überlastet sind, werden unter Umständen mehr Ressourcen angefordert, als benötigt werden. Der so entstandene Überhang kann im Extremfall dazu führen, dass bei einer bessern Verteilung der Ressourcen ganze Server eingespart werden könnten.

Cloud-Computing

Für Unternehmen ist es unattraktiv für kurzweilige Projekte Rechenkapazität in Form von Servern anzuschaffen. Es bietet sich daher an Rechenkapazität nicht mehr physisch vor Ort zu haben, sondern dezentral bei einem Cloud-Provider anzumieten. Dies erlaubt den Unternehmen nicht nur die Auslagerung der Hardwarebetreuung, sondern bietet auch die Möglichkeit dynamisch auf erhöhte Nachfrage zu reagieren. Ein Beispiel hierfür ist der Internet Television Anbieter Netflix, dessen IT-Infrastruktur auf AWS basiert und so globale Verfügbarkeit seiner Inhalte sicher stellt [Ama15].

Der Vorteil von Cloud-Computing ist aber nicht nur, dass dynamisch Rechenkapazität angemietet werden kann, sondern auch, die punktgenaue Abrechnung für diese Kapazität. So kann es von Vorteil sein einen Virtuellen Server bei stündlicher Abrechnung unmittelbar vor betreten eines neuen Abrechnungszeitraumes herunter zu fahren, sofern seine Rechenleistung nicht benötigt wird. [Ama17]

Microservices

Eine effiziente Möglichkeit monolithische Systeme aufzubrechen und ihre Architektur effizient skalierbar zu gestalten, ist es die einzelnen Komponenten in sogenannte Microservices zu

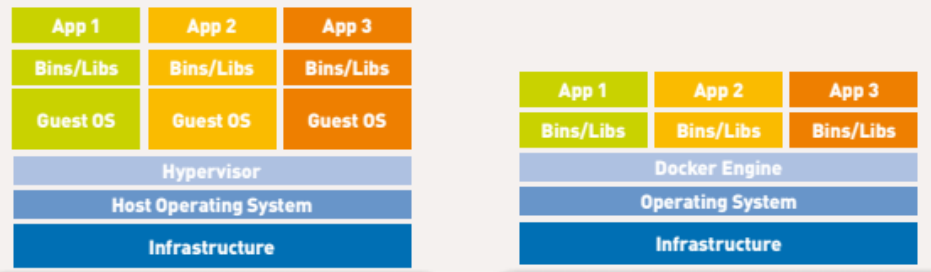

Abbildung 1.1: Vergleich VM links, Dockercontainer rechts [Kra16]

verteilen. Ein System setzt sich in diesem Fall aus einer Reihe von Microservices zusammen. Ist das System mit einer Über- oder Unterlast konfrontiert, können die einzelnen Komponenten horizontal skaliert werden, indem Kopien der Microservices auf neuen Ressourcen bereit gestellt, oder redundante Services abgeschaltet werden.

Um diese Microservices einfach benutzen zu können, werden diese in Containern gestartet. Da die Verwaltung von vielen Containern zu einer immer komplexeren Aufgabe wird, existieren sogenannte Container Plattformen, die selbstständig Container erzeugen und überwachen können. Diese Container Plattformen stellen sicher, dass jederzeit die vom Nutzer gewünschten Microservices zur Verfügung stehen.

1.2 Container

Ein Container befindet sich in einem abgetrennten für ihn abgetrennten Bereich des Kernels. Somit sind Container unabhängig von einander und es gibt keine Ressourcenkonflikte. Im Gegensatz zu Virtuellen Maschinen benötigt ein Container keinen Hypervisor[1], der ein ganzes Gastsystem emuliert. Wie in Abbildung 1.1 zu sehen, ist ein Container deutlich leichtgewichtiger und schneller zu konfigurieren als eine VM[2]. Zusätzlich beschränkt sich die Funktion eines Containers in der Regel auf eine Anwendung. Sollte diese einen Fehler verursachen oder nicht richtig funktionieren, schaltet sich der Container ab. Abbilder der Anwendungen liegen im Dateisystem des Container Programms und können so mehrfach ausgeführt oder gestartet werden.

Docker

Docker ist der Marktführer der Container Software. Erster Release war am 13.03.2013 bei PyCon Lightning talk - The future of Linux Containers. Nach einer frühen Kooperation mit Red Hat (19.09.2013), Microsoft (14.10.2014) und AWS (13.11.2014) bildete Docker, zusammen mit CoreOS, die OCI[3] am 22.06.2015, um offene Industriestandards für Containerformate

[1] Softwaretechnische Abstrahierung zwischen Hardware und Betriebssystem
[2] virtuelle Maschine
[3] Open Container Initiative

festzulegen [Doc17a].

Docker besteht im wesentlichen aus einem zentralen Daemon[4] über den Container gestartet werden können. Dazu sucht nach Konfiguration eines Containers der Daemon nach dem passenden Abbild im lokalen Dateisystem. Findet der Daemon kein passendes Abbild, verbindet er sich mit der zentralen Dockerdatenbank und lädt das gewünschte Abbild herunter.

RKT

Eingeführt durch CoreOS 2014 ist RKT eine Container Engine die für den Gebrauch in Cloud-Umgebungen entwickelt wurde. RKT ist heute schon nativ in vielen Distributionen zu finden, unter andrem Archlinux, CoreOS, NixOS und Fedora.[Cor17]

Im Gegensatz zu Docker, benötigt RKT keinen zentralen Daemon, der das Ausführen von Container-Images übernimmt, sondern startet Container direkt aus der Kommandozeile. Dies bringt RKT im Gegensatz zu Docker in Einklang mit Trennung von Prozessen und Privilegien, wie sie in Unix Systemen üblich ist. Darüber hinaus schafft RKT so die Möglichkeit Container über Initialisierungssysteme wie *systemd* oder *upstart* zu starten.

1.3 Container Plattformen

Um eine Vielzahl von Containern verwalten zu können, werden häufig Container Plattformen verwendet. Diese bestehen in der Regel aus einem oder mehreren Master-Nodes[5], die die Eingaben der Nutzer verarbeiten und auswerten. Bei den meisten Systemen können die Master-Nodes selbst Rechenleistung zur Verfügung stellen, zusätzlich jedoch gibt es Worker-Nodes[6], die ihre Ressourcen dem Master-Nodes zur Verfügung stellen. Die Master-Node kann nun die ihm zur Verfügung gestellten Ressourcen nutzen, um Container zu starten. Ein Vorteil der aus diesem Konzept entsteht ist, dass es eine zentrale Anlaufstelle für Services gibt, die durch verschiedene Container realisiert werden. So existiert zum Beispiel ein virtueller Endpunkt (IP-Adresse) für eine Vielzahl von Containern, die jeweils einen Webserver realisieren. Die Container Plattform ist dann in der Lage den Netzwerkverkehr, der bei ihr für die Webserver ankommt, gleichmäßig auf alle Container zu verteilen, ohne dass der Anwender von diesem Verhalten Notiz nimmt. Sollte eine Container überlastet sein, können weitere des gleichen Typs erstellt werden und die Last gleichmäßig verteilt werden. Dies gelingt so lange, bis die Ressourcen aller Worker-Nodes gebraucht werden.

[4] Programm unter Linux, das im Hintergrund läuft und keine Nutzerinteraktion erfordert
[5] virtuelle oder physischer Hauptserver
[6] Maschinen als Recheneinheit des Clusters

Kuberentes

Die OpenSource Plattform Kubernetes bietet die Möglichkeit zur Verwaltung von Programmen und Applikationen, die in Containern gekapselt sind. Ein Kubernetes Cluster setzt sich zusammen aus Verwaltungsservices (kubeapi-server, kube-controller-manager, kube-scheduler) so wie Rechenservices (kubelet, kube-proxy), die auf einem oder mehreren Servern laufen. Die Verwaltungsservices (in der Regel auf einem oder mehreren Master-Nodes zusammengefasst) sorgen dafür, dass Container gestartet und Ports freigegeben werde. Die Rechenservices (die auf einem oder mehreren Worker-Nodes laufen) übernehmen die Aufgabe die geforderten Container zur Verfügung zu stellen und die Kommunikation zu diesen sicher zu stellen. [The17c]

Kubernetes bietet außerdem die Möglichkeit sogenannter *Rolling-Updates*. Dies bedeutet, dass eine neue Version einer Applikation sukzessive in dem Cluster gestartet wird und die alten Container abgeschaltet werden. Dieser Prozess wird fortgesetzt bis sämtliche laufenden Container mit der neuen Version der Applikation laufen.

Mesos

Mesos von Apache ist ein OpenSource Softwareprojekt zum Verwalten und Erstellen von Compute Clustern. Mesos begann 2009 an der UC[7] Berkeley als Forschungsprojekt und wurde am 27.07.2016 von der Apache Software Foundation veröffentlicht. [The17a]

Mesos besteht aus einem Master Daemon, der auf Master-Nodes läuft, so wie aus einer Reihe von Agenten Daemons, die auf den Worker-Nodes laufen. Die Agenten Daemons stellen Leistung (CPU, RAM usw.) dem Master Daemon zur Verfügung und dieser entscheidet wie viele Ressourcen in welchem Rahmen von einer zu startenden Applikation genutzt werden. [The17b]

Docker Swarm

Mit Docker Swarm lassen sich eine Reihe an Docker Engines (die auf verschiedenen Servern laufen) zu einem Cluster zusammenfassen und verwalten. Ein Docker Swarm besteht aus einer oder mehreren Master-Nodes so wie einer definierten Anzahl an Worker-Nodes. Die Master-Nodes leiten sogenannte *Tasks* an Worker-Nodes weiter, so dass diese die gewünschten Container mit ihren Replikationen erstellen. [Doc17b]

Nomad

Nomad wird von HashiCorp entwickelt und wurde am 28.09.2015 in der ersten Version veröffentlicht. [Has17b]

[7] University of California

Bei der Entwicklung von Nomad wurde viel Wert auf Hochverfügbarkeit, so wie ein hohes Maß an Verteilung auf verschiedene Cloud-Computing Anbieter gelegt. So bietet Nomad Unterstützung für diverse Cloud-Computing Anbieter, wie AWS, GCE[8] oder Microsoft Azure. [Has17a] Wie auch Kubernetes basiert Nomad auf unterster Eben auf einer Docker Engine.

1.4 Software: ecp_deploy

Das Forschungsprojekt Cloud-TRANSIT von Prof. Dr. Nane Kratzke und Peter-Christian Quint, am Kompetenzzentrum CoSA, an der Fachhochschule Lübeck, beschäftigt sich mit Problemen und Abhängigkeiten von Providern im Bereich des Cloud-Computing.[PDNK14]

Im Zuge dieses Projektes ist es immer wieder nötig zu Testzwecken VMs zu einem Cluster zusammen zu fassen und die Services von Container Plattformen auf diesen zu integrieren. Aus diesem Grund entstand im Rahmen einer studentischen Projektarbeit im Jahr 2016 die Software **ecp_deploy**, die in der Lage ist VMs bei verschiedenen Providern anzufordern, zu konfigurieren und zu der Container Plattform Kubernetes zusammen zu fassen. Es nicht nur möglich Cluster zu starten und herunter zu fahren, sondern auch die Anzahl der Worker-Nodes zu verändern. Hierzu erkennt die Software die Differenz der Anzahl der gewünschten Maschinen zu den tatsächlich vorhandenen und beginnt, durch Herunterfahren oder Anfordern von Maschinen, die Größe des Clusters zu regulieren.

1.4.1 Funktionsweise

Der gewünschte Cluster wird in einer JSON-Datei beschrieben und die Login Credentials zu dem gewünschten Cloud-Provider hinterlegt. Anschließend ist das Programm in der Lage eine Softwarerepräsentation des gewünschten Clusters zu erzeugen und die entsprechenden virtuellen Maschinen mit entsprechender Konfiguration über die Provider-CLIs anzufordern und zu konfigurieren. Damit die Virtuellen Maschinen alle für Kubernetes benötigten Services selbstständig mit allen benötigten Parametern starten, erzeugt *ecp_deploy* Cloud-Init Skripte. Diese Skripte werden beim Anfordern einer virtuellen Maschine übergeben und bei ihrem ersten Start ausgeführt. Mit diesen Skripten laden die virtuellen Maschinen alle für Kubernetes benötigten Softwarekomponenten herunter und starten diese. Sind alle Komponenten konfiguriert und gestartet, registrieren sich die virtuellen Maschinen als Worker-Nodes in dem Cluster.

Sobald alle Virtuellen Maschinen erfolgreich gestartet und konfiguriert wurden, speichert *ecp_deploy* den aktuellen Cluster Zustand, um später Änderungen an der Größe des Cluster durchführen zu können. So können beliebig viele VMs angefordert werden, um den Cluster hoch zu skalieren. Entsprechen kann durch Terminierung einzelner Maschinen das Cluster runter skaliert werden.

[8] Google Compute Engine

1.5 Monitoring von Servern

Um die reibungslose Funktion von Servern sicher zu stellen und mit möglichst wenig personellem Aufwand dies zu überwachen, existieren auf dem Markt eine ganze Reihe an Monitoring Lösungen. Da die Software Lösung dieser Arbeit auf einer externen Monitoring Lösung basiert, muss zunächst geklärt werden welche Typen von Monitoring Lösungen existieren und welche für diese Software in Frage kommen. Da alle bisher entwickelten Programme und der Cluster selbst auf Basis von Linux entwickelt wurden, wird sich bei der Monitoring Lösung auf das Monitoring von Linux Servern beschränkt. Eine Monitoring Lösung muss also eine Reihe an Anforderungen erfüllen, damit sie in Frage kommt.

1.5.1 Passives Monitoring

Unter passivem Monitoring versteht man, dass ein Monitoring Tool existiert, an das von den überwachten Servern regelmäßig Daten geschickt werden. Das hat zur Folge, dass auf jedem überwachten Server ein Prozess laufen muss, der die Daten sammelt und an das Monitoring Tool weiter leitet. Nur wenn das Monitoring Tool diese Daten richtig versteht, ist es in der Lage Informationen der überwachten Server zu sammeln und zu verarbeiten.

1.5.2 Aktives Monitoring

Bei aktivem Monitoring sendet das Monitoring Tool in regelmäßigem Abstand Anfragen an alle überwachten Server. Auf diese Anfrage antworten alle Server mit ihren Daten. Der Vorteil von aktivem Monitoring ist, dass Serverausfälle und -fehler sehr schnell erkannt werden, wenn das Monitoring Tool vergeblich auf eine Antwort wartet.

1.5.3 Erfasste Daten

Die Daten die ein Server an den Monitor sendet, müssen in Metriken erfasst werden. Eine Metrik gibt, in Zusammenhang mit dem zugehörigen Wert eines Servers, Informationen darüber, ob der Server innerhalb der festgelegten Grenzen funktioniert. Je nach Metrik können dies einfache numerische Werte sein, wie z.B. bei der aktuellen CPU Auslastung, aber auch einfache boolsche Aussagen wie z.B. „Prozess X ist abgestürzt“.

1.5.4 Funktionale Anforderungen

Aus den vorher beschriebenen Arten von Monitor Lösungen, ergeben sich für dieses Projekt folgende Anforderungen an Funktionen, die ein Monitor erfüllen sollte.

- MF1: (Linux-)Serverdaten müssen abgefragt werden können
- MF2: Metriken für numerische Analyse (CPU Auslastung, RAM Auslastung, usw.)

- MF3: Metriken zum analysieren von Systemlogdateien

1.6 Zusammenhang zur Aufgabenstellung

Nachdem die Grundlagen und Begrifflichkeiten geklärt wurden, lässt sich die Aufgabenstellung mit diesen in Verbindung bringen. Aufgabe ist es eine Monitoring Lösung zu finden, die in der Lage ist Daten von Master- und Worker-Nodes eines Clusters zu erfassen. Zusätzlich muss ein Programm geschrieben werden, dass bei Über- oder Unterlast den Cluster horizontal skaliert. Für diese Skalierung muss eine Änderungen an der Softwarerepräsentation des Clusters vornehmen werden und das Programm *ecp_deploy* mit entsprechenden Parametern ausgeführt werden. Zusammengefasst ergeben sich folgende funktionalen Anforderungen an die entstandene Software.

- FA1: Überwachung der Cluster Infrastruktur
- FA2.1: Erkennen von Über- bzw. Unterlast
- FA2.2: Reagieren auf Über- bzw. Unterlast
- FA3: Durchführung (Scheduling) des Skalierungsprozesses

2 Recherche

Nachdem im letzten Kapitel festgelegt worden ist, welche Anforderungen ein Monitor für die virtuellen Server zu erfüllen hat, wurde eine Internetsuche durchgeführt. Bei dieser Suche wurden eine ganze Reihe von Tools und Softwarelösungen identifiziert und anschließend verglichen.

2.1 Analyse Kriterien

Alle hier betrachteten Monitoring Lösungen mussten zumindest die Anforderungen MF1 bis MF3 erfüllen. Zusätzlich soll mit einer lokalen Installation auf einem Linux-Server gearbeitet werden, da keine Abhängigkeit von einem Online-Monitoring Tool geschaffen werden sollte. Damit die zu entwickelnde Software keine Kosten verursacht und keine Abhängigkeiten von einzelnen Anbietern entstehen, sollte sich auf freie Softwarelizenzen konzentriert werden. Um stabile Rahmenbedingungen für die Entwicklung der Software zu haben, musste darauf geachtet werden, dass die verwendete Monitoring Lösung auch in einer aktuellen, stabilen Version verfügbar ist. Ein häufiges Problem von nicht kommerzieller Software ist, dass die Mitglieder der Entwicklerteams stark fluktuieren und so ganze Entwicklungsstränge nicht weiterentwickelt und fortgeführt werden. Es wurde daher genau darauf geachtet, dass die Monitoring Lösungen nach Möglichkeit das Semantic Versioning[1] verwendeten und in einer Version nach 1.0.0 vorhanden waren. Außerdem durfte nicht außer Acht gelassen werden, dass Kubernetes Cluster in der Praxis deutlich höhere Server-Zahlen haben, als es für prototypische Tests im Rahmen dieser Arbeit üblich war.

Aus diesen Anforderungen ergaben sich für die Bewertung und den Vergleich sowohl harte, als auch weiche Kriterien. Während nur die Lösungen in Frage kamen, die alle harten Kriterien erfüllten, wurden auch solche näher untersucht, die nicht alle weichen Kriterien erfüllen konnten. Die harten Kriterien gliedern sich in folgende Punkte

- **H1**: Lokale Installation möglich,
- **H2**: Lösung unterliegt einer freien Lizenz (GPL, MIT, etc.),
- **H3**: Skalierbarkeit auf große Maschinenmengen ($\geq$ 1000 Server),
- **H4**: Stabile und aktuelle Version verfügbar ($\leq$ 12 Monate alt, Version $\geq$ 1.0.0),

[1] Vergleiche http://semver.org/

Verwendete Suchmaschine	Suchbegriffe
scholar.google.com	server monitoring tool
scholar.google.com	cluster monitoring tool
www.zhb.uni-luebeck.de	server monitoring tool
www.zhb.uni-luebeck.de	cluster monitoring tool

Tabelle 2.1: Suche wissenschaftlicher Quellen

die weichen Kriterien hingegen gliedern sich in die Punkte

- **W1**: Linux binaries der Agentenprogramme für Server verfügbar (kein manuelles Kompilieren auf jeder Maschine nötig),
- **W2**: Monitor lässt sich ohne komplexe Konfiguration auf Ubuntu 16.04 installieren,
- **W3**: Monitor Lösung bietet eine Ruby API,
- **W4**: Monitor bietet ein Alarmsystem,
- **W5**: Monitor bietet die Möglichkeit Docker Container zu überwachen.

Hierbei ist vor allem zu beachten, dass das weiche Kriterien W5 lediglich dazu dient, die hier entwickelte Software Lösung gegebenen Falls später mit einer Container basierten Metrik zu nutzen.

2.2 Internetsuche

Zunächst wurde entschieden von welchen Suchmaschinen aus die Recherche gestartet werden sollte. Es wurde darauf geachtet eine sinnvolle Mischung aus wissenschaftlichen, so wie populären Suchmaschinen zu nutzen, um möglichst breit gefächerte Quellen zu finden.

Dazu wurden zur wissenschaftliche Recherche die Google-Suchmaschine (*scholar.google.com*), so wie die zentrale Hochschulbibliothek der Universität zu Lübeck (*www.zhb.uni-luebeck.de*) heran gezogen. Die benutzten Suchbegriffe der jeweiligen Suchmaschine sind in Tabelle 2.1 festgehalten.

Da die Suche bei wissenschaftlichen Quellen kaum nützliche Ergebnisse erbrachte, wurden zusätzlich die populären Suchmaschinen Bing (*www.bing.com*), so wie Google (*www.google.de*) genutzt. Auch hier wurden die benutzten Suchbegriffe dokumentiert und sind in Tabelle 2.2 festhalten.

Verwendete Suchmaschine	Suchbegriffe
www.bing.com	linux server monitoring
www.bing.com	linux cluster monitoring
www.bing.com	Open Source Linux Monitoring Tools
www.google.com	linux server monitoring
www.google.com	linux cluster monitoring
www.google.com	Open Source Linux Monitoring Tools

Tabelle 2.2: Suche populärer Quellen

Name	H1	H2	H3	H4	W1	W2	W3	W4	W5
bwatch	X	X							
Cacti	X	X		X		X			
ClusterProbe	X	X							
Elastic Stack	X	X	X	X	X	X	X		X
Ganglia	X	X	X	X	X	X	X		
Heapster	–	X	X	X	–	–			X
monitorix	X	X	X	X	X	X		X	
Nagios	X	X	X	X	X		X	X	X
NodeQuery			X	X	X			X	
Prometheus	X	X	X	X	X		X	X	X
ServerDensity			X	X	X			X	
Site24x7			X	X	X			X	X
Supermon	X	X	X						
The Dude	X	X		X					
Zabbix	X	X	X	X	X		X	X	X
Zenoss	X	X	X	X			X	X	X

Tabelle 2.3: Vergleichende Analyse

2.3 Vergleichende Analyse

Nachdem die potentiellen Monitoring Lösungen identifiziert worden waren, mussten mit den vorher festgelegten Kriterien verglichen werden, um eine geeignete Software zu finden.

In der Tabelle 2.3 sind die Suchergebnisse mit dem jeweiligen Hinweis, ob sie die harten und weichen Kriterien erfüllen, aufgelistet.

Aus der Analyse ergab sich, dass die vier markierten Lösungen *Elastic Stack*, *Nagios*, *Prometheus* und *Zabbix* alle harten, so wie mindestens vier von fünf weichen Kriterien erfüllten. Aus diesem Grund wurde im nächsten Schritt ein praktischer Vergleich angestellt, bei dem jede der Lösungen lokal auf einer virtuellen Maschine installiert und getestet wurde.

Eine Sonderrolle übernahm die markierte Monitoring Lösung *Heapster*. Heapster ist eine Monitoring Lösung, die direkt als Container in Kuberentes eingesetzt werden kann, um Kubernetes Cluster zu analysieren. [Con17] Aufgrund der Nähe zu der untersuchten Problemstellung, wurde Heapster ebenfalls praktisch auf seine Tauglichkeit geprüft.

2.4 Praktischer Vergleich

Die fünf Lösungen wurden nacheinander auf virtuellen Servern installiert. Dabei ist eine Grundkonfiguration des Ubuntu 16.04 Cloud-Image benutzt worden.

2.4.1 Elastic Stack

Ein Elastic Stack besteht im wesentlichen aus den drei Modulen **Elastic Search** (zur Suche in den Daten), **Logstash** (Filterung der Daten) und **Kibana** (grafische Weboberfläche). Zusätzlich laufen auf jedem Server, der überwacht werden soll, Prozesse, sogenannte Beats, die Daten an den Monitor senden.

Zunächst mussten alle Paketquellen zur Paketverwaltung von Ubuntu hinzugefügt werden. Nachdem alle Komponenten des Elastic Stack installiert waren, mussten nur wenige Konfigurationen vorgenommen werden. Dabei lauscht Logstash auf einem Port mit einem REST[2]-Server und sendet alle ankommenden Daten weiter an Elasticsearch. Mit Ausnahme der Visualisierung von Kibana waren nur wenige Einstellungen an den Elastic Stack Komponenten vorzunehmen, die Installation lies sich daher sehr einfach automatisieren.

2.4.2 Heapster

Heapster kann als Service in Kubernetes Clustern der Version 1.0.6 oder neuer gestartet und genutzt werden. Heapster bietet dabei eine Metrik Schnittstelle über einen HTTP Server, aus dem alle relevanten Informationen über REST-Anfragen in Erfahrung gebracht werden können.

Obwohl das Programm *ecp_deploy* Kuberentes Cluster der Version 1.5.4 integriert, basieren die genutzten Cloud-Init-Skripte (und damit auch die Struktur der installierten Services) auf der Version 1.0. Die dadurch verwendeten virtuellen Netzwerke des Kubernetes Cluster waren leider nicht in der Lage den Netzwerkverkehr so zu lenken, dass die Metriken von Heapster von externen Prozessen ausgelesen werden konnten. Der Arbeitsaufwand, die Cloud-Init Skripte nach den neusten Spezifikationen einer Kubernetes Installation neu zu schrieben, wurde als zu umfangreich eingeschätzt. Es wurde daher davon abgesehen Heapster als Monitoring Lösung zu verwenden.

2.4.3 Nagios

Damit Nagios installiert werden konnte, musste zunächst der Quellcode aus Paketquellen herunter geladen und kompiliert werden. Auch wenn das Kompilieren und Starten des Programms keine Probleme gab, wurden die benötigten Skripte unter Ubuntu 16.04 nicht korrekt als Service erkannt. Dies führte zu Problemen bei der Automatisierung der Installation. Der

[2]Representational State Transfer

praktische Vergleich zeigte keine nennenswerten Vorteile gegenüber den anderen Monitoring Lösungen, so dass die schwer handhabbare Komplexität der Installation ein Ausschlusskriterium für diese Lösung wurde.

2.4.4 Prometheus

Die Installation von Prometheus, so wie dem zugehörigen Agentenprogramm gestaltete sich als problemlos. Auch die Möglichkeit Metriken von Kubernetes direkt in Prometheus zu nutzen sah nach einer vielversprechenden Lösung aus. Jedoch lies sich Prometheus nicht so konfigurieren, dass Server, die sich in das Cluster neu anmelden, automatisch mit überwacht werden. Da andere Lösungen diese Möglichkeit anboten, wurde von Prometheus als Monitoring Lösung abgesehen.

2.4.5 Zabbix

Zabbix lies sich ohne Probleme auf einem Ubuntu 16.04 Server installieren. Die Konfiguration wurde danach über die Weboberfläche vorgenommen. In der Grundkonfiguration war Zabbix leider nicht in der Lage Daten von neu zu überwachende Maschinen ohne große Konfiguration anzunehmen und zu verarbeiten. Auch wenn sich die Installation von Agenten als problemlos erwies, war der Arbeitsaufwand beim Hinzufügen neuer Maschinen höher als bei den anderen Lösungen.

2.5 Wahl des Monitors

Am Ende des Vergleichs wurde ein **Elastic Stack** gewählt. Sowohl die Einfachheit der Installation, als auch das leichte Hinzufügen neu gestarteter Server zu den überwachten Maschinen, sprachen für diese Monitoring Lösung.

2.6 Integration

Die Analyse ergab, dass ein Elastic Stack genutzt werden sollte. Aus diesem Grund werden im Folgenden alle genutzten Komponenten näher beschrieben.

2.6.1 Der Elastic Stack

Ein Elastic Stack besteht im wesentlichen aus drei Komponenten. **Elasticsearch**, **Logstash** und **Kibana**. Zusätzlich laufen auf jedem Server, der überwacht werden soll, **Agenten** (sogenannte **Beats**), die Daten sammeln und an den Elastic Stack weiter leiten.

Elasticsearch

Elastic Search indiziert die gesammelten Daten. Über die mitgelieferte Anfragesprache lassen sich gezielt Informationen aus den Daten extrahieren.

Logstash

Logstash bietet die Möglichkeit gesammelte Daten vor zu verarbeiten. Es lassen sich Filter und Masken definieren, so dass nur die Daten an Elastic Search zur Verarbeitung weitergeleitet werden, die relevant für den Monitor sind. Dies vereinfacht das Abfragen von gewünschten Informationen.

Kibana

Kibana ist die optionale grafische Weboberfläche des Elastic Stack. Mit ihr ist es möglich Live-Daten zu visualisieren und manuell zu kontrollieren. Mit Hilfe der Entwicklerkonsole können Anfragen gesendet und sofort ausgewertet werden.

Agenten

File- und *Metricbeat* sind die zwei der Daten sammelnden Agenten des Monitors. Mit Filebeat werden wichtige Log-Dateien von den überwachten Servern auf den Monitor gesendet. Metricbeat übernimmt die Aufgabe numerische Daten, wie z.B. verwendeter Arbeitsspeicher, CPU Auslastung oder Netzwerkverkehr zu erfassen und an den Elastic Stack weiter zu leiten.

Neben File- und Metricbeat wurde auch der Einsatz von *Paketbeat* (zur Überwachung von Netzwerk Eigenschaften) und Heartbeat (für Informationen über laufende Prozesse) untersucht.

Die Kombination aus Metric- und Paketbeat erfüllte die Anforderungen MF2 und MF3.

2.6.2 Das Installations Script

Um die für das automatische Skalieren nötige Konfiguration des Elastic Stack sicher zu stellen, wurde in kurzes Bash-Skript geschrieben, dass die Installation und Konfiguration der Komponenten automatisiert. Des Weiteren bietet dieses Skript die Möglichkeit einfache Konfigurationsänderungen, wie etwa verwendete Ports oder Passwörter, zentral anzupassen.

Aufbau

Das Installationsskript gliedert sich in einen konfigurierenden und in einen ausführenden Teil. In dem Hauptverzeichnis finden sich die Dateien

- settings.conf

- setup.sh

In dem Unterverzeichnis *bin* befinden sich die untergeordneten Skripte, die jeweils eine Teilaufgabe der Installation übernehmen. Die untergeordneten Skripte sind:

- bin/repositories
- bin/elastic
- bin/logstash
- bin/kibana

settings.conf

In der Datei *settings.conf* (siehe Beschreibung 2.1) befinden sich die Konfigurationsmöglichkeiten für den Elastic-Stack. Mit ihrer Hilfe können Ports und Logins festgelegt werden. Diese Konfigurationen werden von allen untergeordneten Skripten eingeladen und verwendet.

```
1  # Kibana
2  KB_ADM=kibanaadmin
3  KB_PW=kibana
4  KB_PORT=5601
5  KB_IP=localhost
6
7  # Logstash
8  LS_PORT=5044
9  LS_IP=localhost
10
11 # Elastic search
12 ES_PORT=9200
13 ES_IP=localhost
14
15 # External IP
16 EX_IP=10.50.10.165
```

Beschreibung 2.1: Konfigurationsdatei settings.conf

setup.sh

Die Datei *setup.sh* (siehe Beschreibung 2.2) ruft nacheinander alle untergeordneten Skripte auf, die für die Installation des Elastic-Stack benötigt werden.

```
1 #!/bin/bash
2 cd $(dirname $0)
3 source settings.conf
```

```
4
5 bash bin/repositories
6 bash bin/elastic
7 bash bin/logstash
8 bash bin/kibana
```

Beschreibung 2.2: Hauptscript setup.sh

bin/repositories

Das Skript *bin/repositories* (siehe Beschreibung 2.3) macht alle Paketquellen verfügbar, die für den Elastic-Stack benötigt werden. Zusätzlich aktualisiert es in diesem Zuge die Paketquellen der Distribution und aktualisiert alle installierten Programme auf die neuste Version. Da für einen Elastic-Stack die Java 8 Version von Oracle benötigt wird, die nicht nativ in den Paketquellen von Ubuntu[3] vorhanden ist, muss neben den Elastic-Stack Komponenten auch diese spezielle Java 8 Version installiert werden.

```
1 [..]
2 echo "# Adding elasticsearch repository"
3 sleep 1
4 wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch
      | sudo apt-key add -
5 echo "deb https://artifacts.elastic.co/packages/5.x/apt
    stable main" | sudo tee -a /etc/apt/sources.list.d/elastic
    -5.x.list
6 [..]
```

Beschreibung 2.3: Einrichten der Paketquellen

bin/elastic

Die Installation von Elasticsearch (siehe Beschreibung 2.4) beginnt zunächst mit der Installation von Java 8 von Oracle. Ist dies beendet, wird zunächst Elasticsearch installiert (Zeile 4) und darauf konfiguriert (Zeile 8-12). Hier greift das Script auf die Variablen der *settings.conf* (siehe Beschreibung 2.1) zurück. Wenn der automatische Start bei Reboot eingeschaltet ist (Zeile 16) wird Elasticsearch als Service neu gestartet (Zeile 19).

```
1 [..]
2 echo "## Installing elastic search"
3 sleep 1
4 apt-get -y install elasticsearch
5
6 echo "# Configuring elastic search components"
```

[3]Ubuntu wurde als Linux Distribution der Monitor VM gewählt.

```
7  sleep 1
8  file=/etc/elasticsearch/elasticsearch.yml
9  cat << EOF > $file
10 network.host: $ES_IP
11 http.port: $ES_PORT
12 EOF
13
14 echo "## Start elastic search on reboot"
15 sleep 1
16 update-rc.d elasticsearch defaults 95 10
17
18 echo "## Restart elastic search"
19 service elasticsearch restart
```

Beschreibung 2.4: Installation von Elasticsearch

bin/logstash

Die Installation von Logstash (siehe Beschreibung 2.5) läuft analog zu der Installation von Elasticsearch (siehe Beschreibung 2.4). Nach der Installation von Java 8, wird Logstash installiert (Zeile 4) und konfiguriert (Zeile 8-15). Auch dieses Skript verwendet die Variablen der *settings.conf* (siehe Beschreibung 2.1).

```
1  [..]
2  echo "# Installing logstash"
3  sleep 1
4  apt-get -y install logstash
5
6  echo "# Configuring logstash"
7  sleep 1
8  file=/etc/logstash/logstash.yml
9  cat << EOF > $file
10 path.data: /var/lib/logstash
11 path.config: /etc/logstash/conf.d
12 http.host: "$EX_IP"
13 log.level: debug
14 path.logs: /var/log/logstash
15 EOF
16 [..]
```

Beschreibung 2.5: Installation von Logstash

bin/kibana

Im letzten Schritt der Installation des Elastic-Stack, muss Kibana installiert und konfiguriert werden (siehe Beschreibung 2.6). Zusätzlich zu Java 8 setzt Kibana die Pakete nginx, so

wie die apache2-utils voraus, die in Zeile 4 installiert werden. Anschließend wird Kibana installiert (Zeile 8) und konfiguriert (Zeile 12-16).

```
1  [..]
2  echo "# Install apache2-utils and nginx"
3  sleep 1
4  apt-get -y install apache2-utils nginx
5
6  echo "# Installing kibana"
7  sleep 1
8  apt-get -y install kibana
9
10 echo "# Configuring kibana"
11 sleep 1
12 file=/etc/kibana/kibana.yml
13 cat << EOF > $file
14 server.port: $KB_PORT
15 server.host: "$KB_IP"
16 EOF
17 [..]
```

Beschreibung 2.6: Installation von Kibana

2.6.3 Funktion

Nachdem alle Komponenten installiert wurden, können Metric- und Paketbeat Agenten auf den zu überwachenden Servern installiert werden. Unter der Angabe von der Monitor IP-Adresse und dem Logstash Port, können diese ihre Daten an den Logstash Service senden. Anschließend können die Daten über Elasticsearch bzw. Kibana abgefragt und analysiert werden.

3 Entwurf

Nachdem feststand welche Monitor Lösung verwendet werden sollte und wie die Verbindung zu dieser aussehen sollte, wurde ein Konzept für die automatische Skalierung entwickelt. Teil dieses Konzeptes ergab auch eine Reihe von Anpassungen, die an der vorhandenen Software *ecp_deploy* vorgenommen werden mussten. Die aus diesem Konzept resultierende Software wird im Folgenden „**ecp_autoscale**“ genannt.

3.1 MAPE-Zyklus

Die Wartung von Komponenten verteilter Systeme wird zu einer immer schwierigeren Aufgabe. Durch immer komplexer werdende Strukturen ist es nötig, dass ein System seinen eigenen Zustand überwachen und selbstständig auf Veränderungen der Umwelt reagieren kann. Eine Möglichkeit dieses Problem zu behandeln ist, Rahmenbedingungen für ein System fest zu legen, die es selbstständig einhält und im Fall einer Überschreitung der vorher definierten Grenzen den Nutzer informiert.

Aus diesem Grund begann IBM im Jahr 2001 damit Computer Systeme zu entwickeln, die sich selbst verwalten können [KC03]. Ausgestattet mit Sensoren (Monitoring), Aktoren (Anpassung) und einer Wissensbasis ist ein solches System in der Lage sich auf sich verändernde Umweltbedingungen anzupassen. Die einzelnen Arbeitsschritte dieses Zyklus wird häufig als Monitor-Analyze-Plan-Execute (MAPE) bezeichnet. Die für diese Arbeit entwickelte Softwarelösung implementiert vier Module, wobei jedes Modul einem der vier Arbeitsschritte entspricht.

3.2 Konzept

Ein grobes Konzept hinter der Software *ecp_autoscale* kann der Abbildung 3.1 entnommen werden. Im wesentlichen setzt die Software sich aus den vier Modulen **Monitor**, **Analyser**, **Planner** und **Executor** zusammen. Hierbei ist festzustellen, dass die Schnittstellen der Software insbesondere im Monitor Modul (Schnittstelle zur Monitoring Lösung Elastic Stack) und im Executor Modul (Schnittstelle zum ecp_deploy Programm) sind.

Im Folgenden werden die einzelnen Module näher beschrieben. Ein grober Ablauf, der einzelnen Aktivitäten, die für den MAPE-Zyklus benötigt werden, ist in Abbildung 3.2 dargestellt. Hier ist deutlich zu erkennen, dass die Schnittstellen der Software im Monitor (abfra-

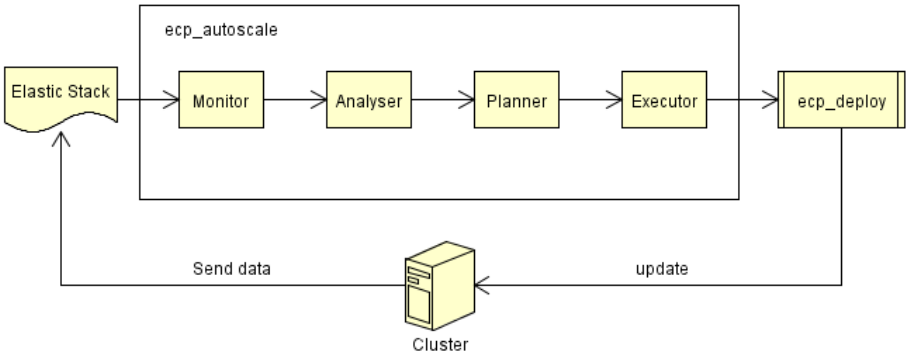

Abbildung 3.1: Konzept mit MAPE-Zyklus

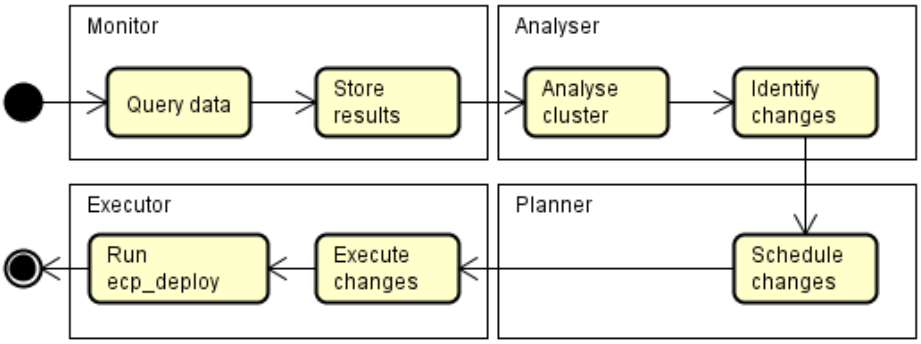

Abbildung 3.2: Aufgabenteilung des MAPE-Zyklus

gen der Daten der einzelnen Server), so wie im Executor (ausführen von ecp_deploy) liegen. Die Module sollten so weit wie möglich unabhängig von einander agieren, um einzelne Komponenten austauschbar zu gestalten. So soll es ohne Probleme möglich sein den Planner um weitere (komplexere) Scheduling Strategien zu erweitern, ohne die anderen Module anpassen zu müssen. Aber auch die externen Schnittstellen (z.B. Monitor und Metriken) müssen so implementiert werden, dass zu einem späteren Zeitpunk weitere Monitoring Lösungen genutzt werden können, ohne die Architektur wesentlich zu verändern.

3.2.1 Monitor

Das Monitormodul (Monitor) bildet die Schnittstelle zum externen Monitor. Es musst so gestaltet sein, dass sowohl die Sprache in der Anfragen geschrieben werden, als auch die Konfiguration (IP-Adresse, Port usw.) zur Laufzeit eingelesen und ausgewertet werden. Dies soll später die Möglichkeit bieten neben dem Elastic Stack auch andere Monitoring Lösungen einzubinden.

Da in dieser Arbeit ein passives Monitoring als Grundlage verwendet wird, ist es nötig, dass innerhalb dieses Moduls die Metriken der Server in regelmäßigen Abständen abgefragt

werden. Diese Art von „Polling“ ist nur nötig, wenn, wie im Fall eines Elastic Stack, kein aktives Monitoring statt findet bzw. kein Alarmsystem existiert. Aus diesem Grund existiert in der Klasse Monitor die Funktion **main_loop**.

Innerhalb dieser Funktion werden für jeden Server des Clusters eine Anfrage an den Monitor gesendet, um die Daten der vorher definierten Metriken auszulesen. Im Fall des Elastic Stack lassen sich alle Metriken in einem Schritt abfragen, dazu müssen lediglich innerhalb der Anfrage Aggregationen definiert werden, die die Werte der Metriken enthalten. Sind alle Werte und ihre Aggregationen erfasst, werden diese im nächsten Schritt in einem Ergebnis-Hash gespeichert, der für jeden Hostname die Antwort der Anfrage im JSON[1]-Format festhält.

3.2.2 Analyser

Das Analysemodul (Analyser) verarbeitet die vom Monitor ausgelesenen Metriken und prüft ob eine Skalierung vorgenommen werden muss. Dazu wird das Ergebnis des Monitors, so wie die entsprechenden Metriken an das Analysemodul übergeben. Für jeden Server wird nun überprüft, ob die Ergebnisse innerhalb der in den Metriken definierten Grenzen liegt. Stellt das Analysemodul fest, dass eine Skalierung des Clusters vorgenommen werden soll, veranlasst es diese über die vom Planungsmodul bereit gestellten Funktionen. Wichtig ist, dass die Analyse des Clusters erst statt finden kann, wenn die Daten für alle Server abgefragt werden können. Daher müssen die Ergebnisse des Monitors im Analysemodul zwischen gespeichert werden.

3.2.3 Planner

Stellt das Analysemodul fest, dass die Größe des Clusters angepasst werden muss, so übernimmt das Planungsmodul (Planner) die Ausführung dieser Anpassung. In dem Entwurf dieser Arbeit, veranlasst das Planungsmodul die Skalierung des Clusters über einen einfachen Scheduling Algorithmus. Es werden also alle Skalierungsoperationen (in Form von Cluster Aktionen) in eine Warteschlange geschrieben, die in regelmäßigen Abständen abgearbeitet wird. Aus diesem Grund existiert in diesem Modul die Funktion **execute_loop**, die parallel zu der Funktion *main_loop* des Monitors und das Abarbeiten der Warteschlange übernimmt. Das Planungsmodul stellt aber auch sicher, dass in der Warteschlange einzelne Änderungen am Cluster nicht mehrfach vorkommen.

Im letzten Schritt der Skalierung, veranlassen die Cluster Aktionen des Planungsmoduls die Modifizierung des Cluster Objektes. Dazu werden Funktionen des Ausführungsmoduls genutzt, die die Veränderung am Cluster Objektes kapseln.

[1] JavaScript Object Notation

3.2.4 Executor

Mit dem Ausführungsmodul (Executor) führt die Softwarelösung die eigentliche Skalierung des Clusters aus. Für diesen Zweck enthält das Ausführungsmodul ein Cluster Objekt, an dem Änderungen vorgenommen werden. Sind alle gewünschten Änderungen durchgeführt, wird über die Funktion **update_cluster** das neue Cluster Objekt gespeichert und als Parameter an das Programm *ecp_deploy* übergeben.

Zusätzlich muss das Ausführungsmodul in der Lage sein Agenten auf VMs zu starten. Aus diesem Grund stellt das Ausführungsmodul zusätzlich die Funktionen **send_ssh_command** und **copy_files_to_machine** bereit.

3.2.5 Cluster Aktionen

Die Cluster Aktionen (Cluster-Action) sind eine Reihe von Funktionen, die ein Cluster Objekt verändern können. Sie werden in die Warteschlange des Planungsmoduls eingereiht und nacheinander abgearbeitet. Die Cluster Aktionen gliedern sich in die Folgenden Aktionen:

- StartAction - Veranlasst den Start einer VM für den Cluster.
- KillAction - Veranlasst das Löschen einer VM aus dem Cluster.
- RestartAction - Veranlasst den Neustart einer VM in dem Cluster.
- InitializeAction - Setzt den Initialisierungsstatus einer VM des Clusters.

3.2.6 Metriken

Die Metriken der Monitoring Lösung bieten die Hauptinformationen auf deren Basis das entwickelte Programm die Skalierung des Clusters vor nimmt. Metriken umfassen sowohl numerische Werte, als auch boolsche Aussagen. Damit die Metriken der Monitoring Lösung genutzt werden können, muss eine Softwarerepräsentation dieser Metriken existieren. Da das Analysieren und Bestimmen geeigneter von Metriken zur Skalierung kein Teil der Aufgabenstellung war, wurde versucht möglichst einfache Metriken zu verwenden und diese generisch zu beschreiben. Dies dient dazu, eine spätere Erweiterung der Software an dieser Stelle zu vereinfachen.

Boolsche Metrik

Diese Art von Metrik bietet wenig Komplexität. Mit ihr kann die Software auf Serverausfälle oder Prozessabstürze innerhalb des Clusters reagieren. Enthält eine Logdatei Fehlermeldungen oder gibt ein Server keine Rückmeldungen, sorgt die Software durch Abfrage einer solchen Metrik dafür, dass der Server aus dem Cluster genommen wird und ein Server gleichen Typs herauf gefahren wird.

Dieser Typ Metrik prüft also ein bestimmtes Verhalten (zum Beispiel Existenz von Log-Einträge, oder Anzahl fehlgeschlagener HTTP Anfragen über einer bestimmten Schwelle) und teilt dem Analysemodul nach Auswertung ein einfaches Ja oder Nein mit.

Numerische Metrik

Numerische Metriken sind Werte von Eigenschaften einer Maschine, die innerhalb eines Zeitfenstern einen bestimmten Wertebereich nicht verlassen dürfen. Sollte z.B. ein Server über mehrere Minuten eine hohe CPU-Auslastung besitzen, so kann mit dieser Metrik das Analysemodul weitere Server gleichen Typs anfordern. Ebenso können Server herunter gefahren werden, wenn eine Vielzahl Server gleichen Typs existieren und jeder eine niedrige CPU-Auslastung hat.

Zu erwähne ist, dass die geplante Software auf prozentualen Werten der Metriken arbeiten soll. Die ausgelesenen Daten müssen auf den Wertebereich von Null bis Eins abgebildet werden. In vielen Fällen ist dies bereits gegeben, da prozentuale Metriken abgefragt werden können. Es muss aber auch möglich sein aus absoluten Werten eine prozentuale Auslastung zu errechnen.

Ferner kann es sein, dass sich Metriken genau invers zum gewünschten Ergebnis verhalten. Zum Beispiel lässt sich die CPU-Auslastung als Addition aller laufenden Prozesse darstellen, einfacher ist es jedoch zu überprüfen wie sich der Leerlaufprozess verhält. Hat der Leerlaufprozess einen hohen Anteil, so ist die CPU-Auslastung niedrig, ein Verhalten, das genau der umgekehrten CPU-Auslastung entspricht.

3.3 Ablauf

Bei der Entwicklung des Entwurfs zeigt sich, dass mindestens zwei asynchron laufende Prozesse für das Programm benötigt werden. Die Daten der Cluster Maschinen müssen in regelmäßigen Abständen abgefragt und ausgewertet werden, während zeitgleich die Änderung an einem Cluster vorbereitet werden müssen. Aus diesem Grund kristallisierten sich die beiden Prozesse **main_loop**, so wie **execute_loop** heraus, die diese beiden Aufgaben übernehmen.

3.3.1 Prozess 1: main_loop

Die Funktion **main_loop** stellt das Herzstück des Monitoring Moduls dar. Innerhalb dieser Funktion werden die Daten der Monitoring Software abgefragt, zwischengespeichert und anschließend an das Analysemodul weiter geleitet. Ein grober Ablauf einer solchen Analyse (mit dem Ergebnis, dass der Cluster herauf skaliert werden muss), kann der Abbildung 3.3 entnommen werden.

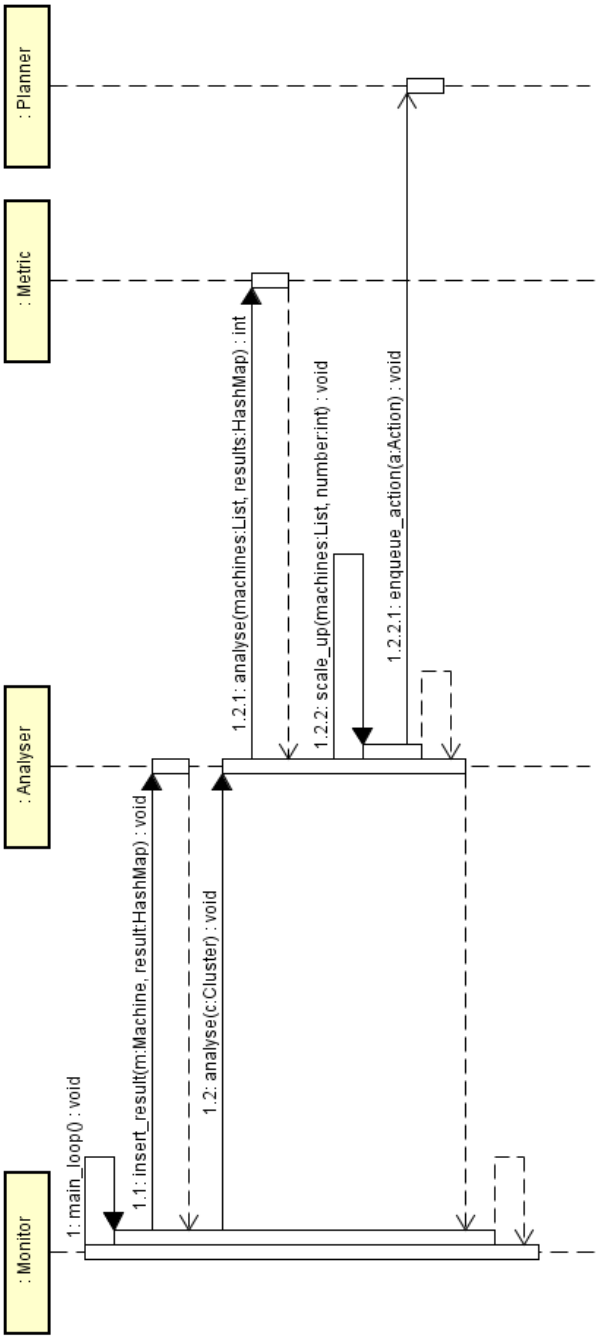

Abbildung 3.3: Sequenzdiagramm main_loop

Das Monitormodul muss zunächst die Daten abfragen und dem Analyser weiter reichen (Funktion **insert_result**). Ist dies für alle Maschinen des Clusters geschehen, wird der Analysepart des Analysemoduls gestartet (Funktion **Analyser.analyse(cluster)**). Teil dieser Funktion ist, dass jede registrierte Metrik alle Maschinen des Clusters analysiert (Funktion **Metric.analyse(machines,results)**).

Ergibt sich aus der Analyse, dass der Cluster skaliert werden muss, wird die Funktion **scale_up** bzw. **scale_down** aufgerufen. Diese Funktionen erzeugen eine Start-, bzw. KillAction und reiht diese in die Warteschlange des Planungsmoduls ein (Funktion **enqueue_action(action)**).

3.3.2 Prozess 2: execute_loop

Der zweite wesentliche Prozess des Programms, ist die Durchführung der geforderten Änderungen an dem Cluster Objekt. Innerhalb des Planungsmoduls können beliebig komplexe Strategien zur Umsetzung der Änderungen genutzt werden. Für das erste Konzept jedoch wird eine FIFO[2] Warteschlange genutzt. Das bedeutet, dass die Änderungen an dem Cluster in der Reihenfolge statt finden, wie sie das Analysemodul festgelegt hat.

Wie in Abbildung 3.4 zu sehen ist, werden nacheinander alle Aktionen der Warteschlange angesprochen und ausgelöst (Funktion **fire**). Im Fall einer StartAction, wird die Funktion **start_machine(machine)** des Ausführungsmoduls aufgerufen. Mit dieser Funktion wird die eigentliche Modifikation am Cluster Objekt ausgeführt.

Sind alle Aktionen ausgelöst worden, wird über die Funktion **update_cluster** des Ausführungsmoduls aufgerufen und mit ihr das Programm *ecp_deploy* gestartet.

3.4 Datenstrukturen

Nachdem feststand wie der Programmablauf statt finden sollte, wurde überlegt welche geeigneten Datenstrukturen genutzt werden können. Da das Programm in Ruby implementiert wurde, konnte auf die effiziente Nutzung von HashMaps und Listen zurück gegriffen werden. Ein genaues Diagramm der Klassenhierarchien kann Abbildung 3.5 entnommen werden. Dem Klassendiagramm nach schien die Komplexität des Analyse- und Ausführungsmoduls deutlich über der des Monitor- und Planungsmoduls zu liegen. Im Laufe der Implementierung bestätigte sich diese Vermutung, wie in Kapitel 4 gezeigt wird.

[2] First-In-First-Out

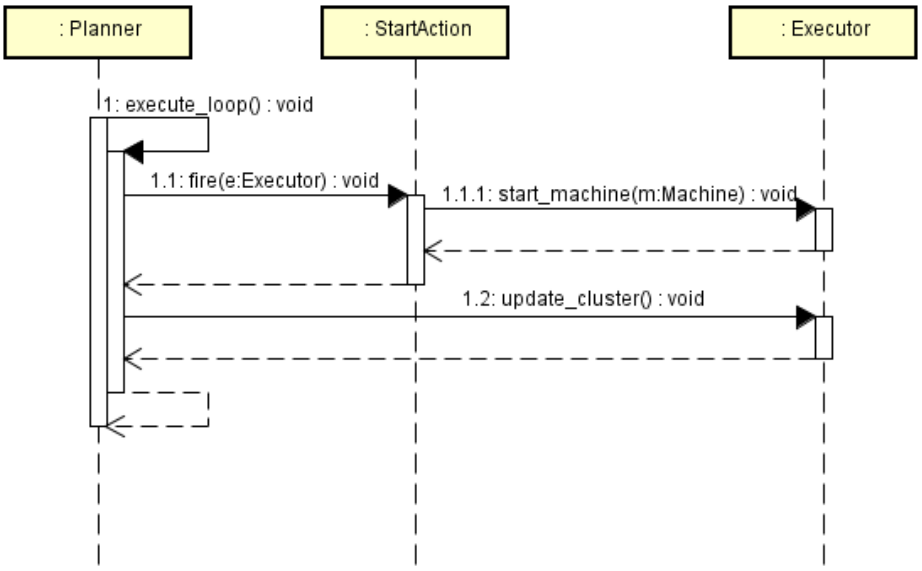

Abbildung 3.4: Sequenzdiagramm execute_loop

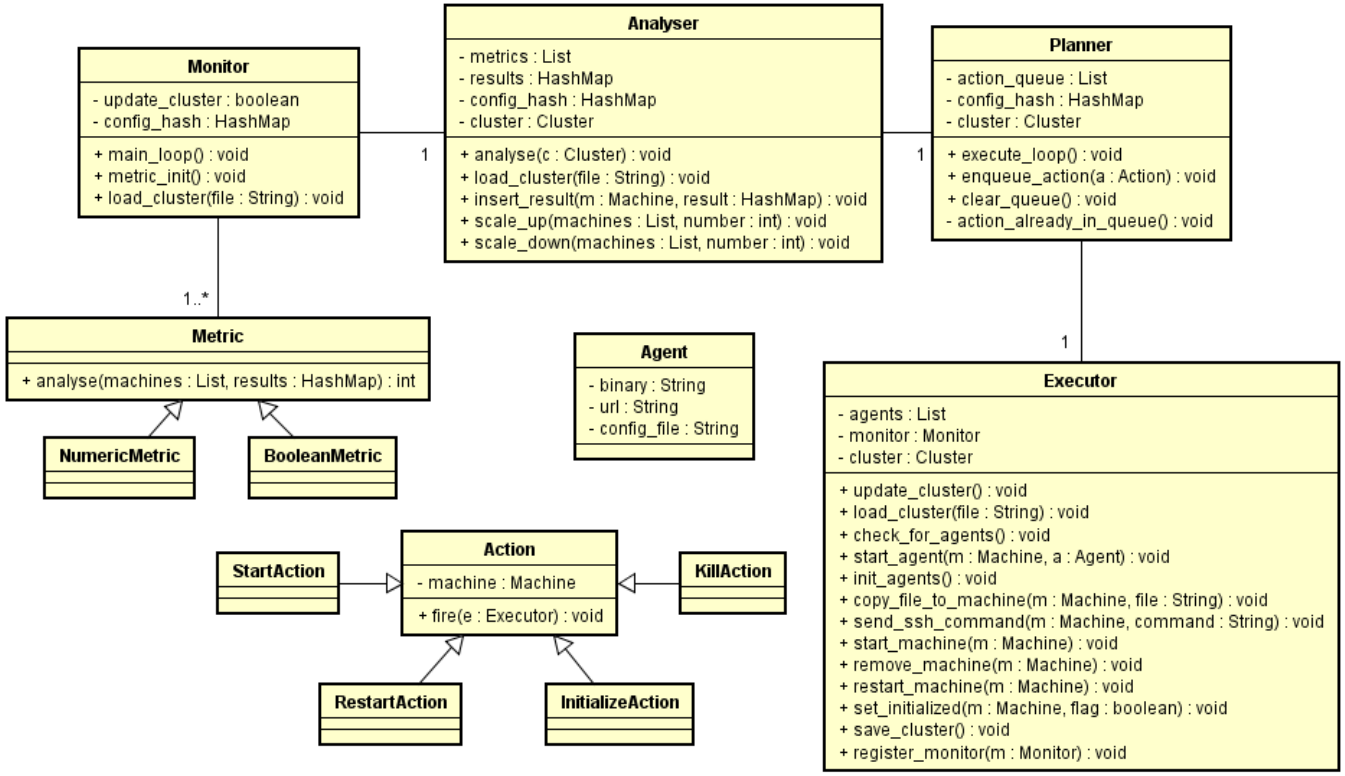

Abbildung 3.5: Klassendiagramm des Entwurfs

4 Implementierung

Um das in Kapitel 3 entwickelte Konzept auf seine Machbarkeit zu prüfen, musste das Konzept zunächst praktische Implementiert werden. Der Aufgabenstellung entsprechend wurde hierzu die Programmiersprache Ruby gewählt.

Da bereits das Programm *ecp_deploy* in Ruby entwickelt wurde, konnten nach kleinen Anpassungen die vorhandenen Datenstrukturen für die Softwarerepräsentation eines Clusters verwendet werden. Dies erleichterte insbesondere die Skalierung eines Clusters durch den Executor.

4.1 Anpassung des ecp_deployer

Da die Implementierung auf bestehende Datenstrukturen von *ecp_deploy* zurück greifen sollte, wurde die Funktionsweise des Programms *ecp_deploy* leicht geändert. Es musste die Möglichkeit bereit gestellt werden Cluster-Objekte direkt über eine Eingabe zu vergleichen, so wie Server eindeutig an ihren Hostnames zu identifizieren. Hier bot es sich auch an die Klasse Cluster um ein paar nützliche Funktionen zu erweitern, die das Arbeiten auf einem oder mehreren Maschinen Objekten erleichtern.

4.1.1 Hostnames

Daten im Elastic Stack werden nach ihren Hostnames sortiert. Daher musste der ecp_deployer so angepasst werden, dass eindeutige Hostnames an die startenden Server vergeben werden. Nur so ist es möglich, dass gezielt einzelne Server abgeschaltet oder neu gestartet werden können. In Beschreibung 4.1 ist die Funktion zur Generierung eines neuen Namen für Maschinen angegeben.

Um Namen zu generieren wird die Ruby Bibliothek *SecureRandom* verwendet. Mit ihr wird eine zufällige Kombination von hexadezimal Zahlen als Name generiert (Zeile 3). Danach wird geprüft, ob es Maschinen gibt, die bereits diesen Namen tragen (Zeile 7) und entweder die Funktion beendet, oder ein neuer Name generiert (Zeile 11).

```
def get_next_uid
  run = true
  tmp = SecureRandom::hex(NAME_LENGTH)
  while run
    contain = nil
    unless @machines.nil?
      contain = @machines.select {|m| m.hostname == "#{tmp}"}
    if contain.nil? || contain.empty?
      run = false
    else
      tmp = SecureRandom::hex(NAME_LENGTH)
    end
  end
  tmp
end
```

Beschreibung 4.1: Erzeugen eines neuen Hostnames des Clusters

4.1.2 Vergleich von Cluster-Objekten

Um Änderungen am Cluster vor zu nehmen, bietet *ecp_deploy* die Möglichkeit Cluster-Objekte zu vergleichen. In der bisherigen Implementation wurde dabei eine JSON-Datei in ein Cluster-Objekt überführt und mit dem aktuellen Stand verglichen. Um diesen Prozess effizienter zu gestalten, wurde das Programm so angepasst, dass zwei Cluster-Objekte eingelesen und verglichen werden können, ohne dass eine Cluster Beschreibung in Form einer JSON-Datei vorhanden sein muss.

4.1.3 Arbeiten auf Maschinen Rollen

Da es im Zuge des Monitoring und Auslesen von Daten immer wieder dazu kommt, dass Maschinen bestimmter Rollen in einer logischen Einheit zusammen gefasst werden müssen, wurde die Klasse Cluster um die Funktion *get_machines_by_role* (siehe Beschreibung 4.2) erweitert. Diese Funktion gibt ein Array von Maschinen zurück, die alle einer bestimmten Rolle angehören.

```
def get_machines_by_role(role)
  @machines.select{|m| m.role==role}
end
```

Beschreibung 4.2: Filter von Maschinen einer bestimmten Rolle

4.1.4 Initialisierungsstatus

Das Anfordern von neuen VMs von einem Cloud-Provider ist zwar schnell zu erledigen, bis diese VMs jedoch vollständig gestartet und erreichbar sind, vergehen wenige Minuten. Damit die Monitoring Lösung in dieser Zeit Anfragen an diese VMs unterbindet, wurde die

Maschinen Klasse des ecp_deploy Programms um den Parameter *initialized* erweitert (siehe Beschreibung 4.3, Zeile 5).

```
1 def initialize([..])
2   @provider = provider
3   [..]
4   @starting_time = Time.now.to_i
5   @initialized = false
6 end
```

Beschreibung 4.3: Erweiterte Maschinen Klasse

Mit diesem Parameter alleine ist der Umgang mit gestarteten VMs jedoch unhandlich, weshalb die Klasse Cluster um ein paar Funktionen erweitert wurde (siehe Beschreibung 4.4). Mit diesen Änderungen können Maschinen über ihre IP-Adresse erfragt werden (Zeile 1-4), ihr Initialisierungsparameter kann mit Hilfe einer IP-Adresse gesetzt werden (Zeile 6-9) und es kann eine Liste aller initialisierten Maschinen zurück gegeben werden (Zeile 11-13).

```
1  def get_machine_by_ip(ip)
2    arr = @machines.select{|m|m.private_ipv4==ip}
3    arr.nil? ? arr : arr.first
4  end
5
6  def set_initialized_by_ip(ip,flag=true)
7    m = get_machine_by_ip(ip)
8    m.initialized=flag unless m.nil?
9  end
10
11 def select_intialized
12   @machines.select{ |m| m.initialized}
13 end
```

Beschreibung 4.4: Neue Maschinen Funktionen in Cluster Klasse

Somit kann sicher gestellt werden, dass das Monitormodul nur Server überwacht, die vollständig initialisiert wurden.

4.2 Monitor

Das Monitor Modul der Software, stellt die Verbindung zu dem Elastic-Stack her. Welche Metriken abgefragt werden können, muss vorher, durch den Anwender, in JSON-Dateien festgelegt werden. Der Monitor ließt diese Dateien ein, verarbeitet ihren Inhalt und erzeugt Anfragen (vergleiche Kapitel 4.2.2), die für jede Maschine an den Stack gesendet werden. Die Ergebnisse dieser Anfragen werden, zusammen mit den Metriken, an das Analysemodul weiter gereicht, damit dieses eine Skalierung initiieren kann.

4.2.1 JSON-Metriken

Wie bereits angedeutet, werden die abgefragten Daten aller Metriken mit Hilfe des JSON-Format konfiguriert. Eine entsprechende JSON-Datei wird eingelesen und aus ihr eine Anfrage in Query DSL[1][Ela17] für Elasticsearch generiert. Eine exemplarische Beschreibung der Metrik, die den Leerlaufprozess aller Maschinen abfragt, ist in Beschreibung 4.5 angegeben. Neben dem Namen (Zeile 2), dem Typ (Zeile 3) und den Grenzen (Zeile 5-6), wird auch festgehalten, ob es sich um eine inverse numerische Metrik handelt und ob ein Absolut Wert existiert (vergleiche Kapitel 3.2.6).

```
1  {
2    "name": "CPU-Idle",
3    "type": "numeric",
4    "checks": {
5      "min": 0.85,
6      "max": 0.30,
7      "inverse":true,
8      "has-total":false
9    },
10   "aggregations": [
11     {
12       "name": "cpu-idle",
13       "field": "system.cpu.idle.pct",
14       "func": "avg"
15     }
16   ]
17 }
```

Beschreibung 4.5: Exemplarische Beschreibung einer Metrik in JSON

4.2.2 Query

Instanzen der Klasse Query (vergleiche Beschreibung 4.6) sind in der Lage eine Anfrage für Elasticsearch im Query DSL-Format zu erzeugen. Damit eine Anfrage alle Daten abruft, die von den Metriken benötigt werden, generieren die Metriken ihre Aggregationen selbst und speichern diese in der Query Klasse (Funktion *create_aggregation* Zeile 3-6).

Eine Anfrage kann nun mit Hilfe einer Maschine erzeugt werden (Funktion *generate* Zeile 17-19). Dazu wird eine Zeichenkette mit allen Aggregationen erstellt, die JSON-Konventionen entspricht (Funktion *json_query*, Zeile 8-15).

[1]Query Domain Specific Language

```
1  class Query
2    [..]
3    def create_aggregation(metric)
4      @aggs.push(metric.aggregation)
5      self
6    end
7
8    def json_query(machine)
9      tmp = ''
10     @aggs.each do |a|
11       tmp += "#{a},"
12     end
13     tmp = tmp.chop
14     "{\"query\":{\"bool\":{\"must\":[{\"term\":{\"beat.hostname\" : \"#
           {machine.hostname.to_s}\"}},{\"range\":{\"@timestamp\": {\"gte\"
           :\"now-#{@time_frame}\",\"lte\":\"now\"}}}]}},\"size\":0,\"aggs\
           ": {#{tmp}}}"
15   end
16
17   def generate(machine)
18     {index: @search_index.to_s, body: json_query(machine)}
19   end
20 end
```

Beschreibung 4.6: Query Klasse mit Aggregationen

Aggregation

Eine Aggegration beschreibt ein bestimmtes Datenfeld einer Anfrage und die mathematische Funktion, die auf die Daten angewendet werden soll. So besteht zum Beispiel die Möglichkeit den Durchschnitt der Last für Prozesse in einem Zeitfenster zu ermitteln. Bei dem erzeugen der Softwarerepräsentation der Metriken, sorgt der Monitor dafür, dass entsprechende Aggregationen erzeugt werden.

Exemplarisch ist in Beschreibung 4.7 die Erzeugung der für eine numerische Metrik benötigten Aggregation im JSON-Format zu sehen. Dazu wird jede nötige Aggregation (Zeile 4) überprüft und ihre Query DSL Befehl (Zeile 8) erzeugt.

4.2.3 Funktion: main_loop

Das Herzstück des Monitormoduls und damit auch des MAPE-Zyklus, ist die Funktion *main_loop*. In dieser Funktion werden die zuvor generisch generierten Anfragen für jede Maschine des Clusters angepasst und an den Elastic-Stack gesendet. Das Ergebnis sind Datenfelder, wie sie für die Metriken benötigt werden. Diese Datenfelder werden gespeichert und dem Analysemodul zur weiteren Verarbeitung überlassen. In Beschreibung 4.8 lässt sich der genaue Ablauf dieser Funktion ablesen. Nach der erfolgreichen Initialisierung aller benötigter Komponenten (Zeile 2-4), beginnt die Schleife für die Suchanfragen an den Elastic-Stack.

```
1  def generate_aggs(metric_hash)
2    tmp_str = ""
3    agg_field = metric_hash['aggregations']
4    agg_field.each do |agg|
5      name = agg['name']
6      func = agg['func']
7      field = agg['field']
8      tmp_str += "\"#{name}\":{\"#{func}\":{\"field\": \"#{field}\"}},"
9    end
10   @aggregation = tmp_str.chop
11 end
```

Beschreibung 4.7: Aggregationsfunktion der NumericMetric Klasse

Dies werden für alle Maschinen gesendet (Zeile 7-10) und im Analysemodul abgespeichert. Sind alle Ergebnisse erfasst, wird das Analysemodul angesprochen und eine Analyse des Clusters initiiert (Zeile 11). Anschließen legt sich der Prozess des Monitormoduls für eine in der Konfiguration definierte Zeitspanne schlafen.

```
1  def main_loop
2    interval = @config_hash['monitor-interval'].nil? ?
         DEFAULT_MONITOR_INTERVAL : Utils::interval_in_seconds(@config_hash
         ['monitor-interval'])
3    metric_init
4    @executor.register_monitor(self)
5    while @running
6      load_cluster(@cluster_file) if @update_cluster
7      @cluster.machines.each do |machine|
8        response = @es_client.search(@query.generate(machine))
9        @analyser.insert_result(machine,response)
10     end
11     @analyser.analyse(@cluster)
12     sleep interval
13   end
14 end
```

Beschreibung 4.8: Funktion main_loop des Monitormoduls

4.3 Analyser

Das Analysemodul der Software bewertet die Ergebnisse, die von dem Monitormodul geliefert werden. Es untersucht jeweils den aktuellen Zustand des Clusters und prüft, ob sich dieser, gemäß der Metriken, in einem Normalbereich befindet. Dabei wird davon ausgegangen, dass die Maschinen bestimmter Rollen zusammengefasst werden und zusammen den Vorgaben der Metriken entsprechen müssen.

4.3.1 Erzwingen der Cluster Grenzen

Eine Funktion die sich im Laufe der Entwicklung als nützlich und auch notwendig erwies, war die Möglichkeit Grenzen der Skalierung festzulegen und das Analysemodul zu nutzen, um diese Grenzen einzuhalten. So lassen sich in der Konfiguration Minimal- und Maximalwerte der Anzahl an Maschinen pro Rolle festlegen und das Analysemodul sorgt dafür, dass nicht über die Grenzen hinaus skaliert wird. Dazu wird zunächst, vor beginn der Skalierung, einmalig geprüft ob sich die Anzahl der Maschinen innerhalb dieser Grenzen befindet. Ist dies nicht der Fall, wird zur nächstgelegenen Grenze herauf oder herunter skaliert.

Die Funktion *enforce_role_boundaries* (siehe Beschreibung 4.9) stellt dieses Verhalten sicher. Dazu werden alle Maschinen einer Rolle betrachtet (Zeile 2-3) und ihre Anzahl mit den vorher definierten Grenzen verglichen (Zeile 4-11). Sollten die Grenzen überschritten werden, wird eine entsprechende Anzahl Maschinen gestartet (Zeile 6) bzw. herunter gefahren (Zeile 10).

```
1  def enforce_role_boundaries(role)
2    machines = @cluster.get_machines_by_role(role)
3    machine_numer = machines.count
4    if machine_numer < min_machines
5      Utils::LOG.debug(self){"Role #{machines.first.role} out of
           boundaries. Starting #{min_machines-machine_numer} server"}
6      scale_up(machines,min_machines-machine_numer)
7    end
8    if machine_numer > max_machines
9      Utils::LOG.debug(self){"Role #{machines.first.role} out of
           boundaries. Removing #{machine_numer-max_machines} server"}
10     scale_down(machines,machine_numer-max_machines)
11   end
12 end
```

Beschreibung 4.9: Erzwingen der Einhaltung von Cluster Grenzen

4.3.2 Initialisierung gestarteter Maschinen

Damit das Analysemodul nur auf vorhandene Daten zurück greift, analysiert es nur die Ergebnisse des Monitormoduls, die von vollständig initialisierten Maschinen gesendet werden (siehe Beschreibung 4.10). Aus diesem Grund werden zunächst alle Maschinen ignoriert, deren Initialisierungsparameter *false* ist. Behandelt das Modul eine nicht initialisierte Maschine, so prüft es, ob bereits Ergebnisse von den Metriken gesendet werden (Zeile 5-12). Ist dies der Fall, wird eine Initialisierungsaktion von dem Planungsmodul gefordert (Zeile 2).

```
1  def check_for_initialized(machine)
2    @planner.enqueue_action(InitializeAction.new(machine)) if has_results
         (machine)
3  end
4
5  def has_results(machine)
6    total = 0
7    hits = nil
8    hash = @results[machine.hostname]
9    hits = hash['hits'] unless hash.nil?
10   total = hits['total'] unless hits.nil?
11   total > 0
12 end
```

Beschreibung 4.10: Prüfen und Initialisieren von Maschinen

4.3.3 Funktion: analyse

Sobald Daten für jede Maschine vom Monitor angefordert wurde, ruft er die Funktion *analyse* des Analysemoduls auf (siehe Beschreibung 4.11). Wie in dieser Funktion zu erkennen ist, werden zunächst die verwendeten Rollen der Worker-Nodes des Clusters erfasst (Zeile 3) und geprüft, ob sämtliche Maschinen dieser Rolle, die initialisiert sind, auch Daten senden (Zeile 8-14). Wird in diesem Schritt festgestellt, dass eine initialisierte Maschine keine Daten mehr sendet, wird ihr Neustart veranlasst (Zeile 10). Alle nicht initialisierten Maschinen werden vorerst ignoriert (Zeile 15).

Danach werden zunächst die Boolschen Metriken behandelt, indem die Analyse Funktion jeder Metrik aufgerufen wird (Zeile 22). Der Rückgabe Wert dieser Funktionen ist eine Liste aller Maschinen, für die die Metrik positiv ausgewertet wurde.

Dann wird für jede Numerische Metrik geprüft, ob die Maschinen dieser Rolle in den spezifizierten Parametern arbeiten (Zeile 31-39). Jede dieser Metriken gibt darauf eine Empfehlung an den Analyser ab, ob und wie der Cluster skaliert werden soll (Variable **ret** in Zeile 33). Das arithmetische Mittel sämtlicher Empfehlungen gibt nun den Skalierungsfaktor mit dessen Hilfe die Cluster Skalierung vorgenommen wird (Zeile 40 und 42). Mit diesem Skalierungsfaktor wird die Funktion *scale_role* aufgerufen, in der die eigentliche Cluster Skalierung veranlasst wird.

4.3.4 Skalierungsfunktionen

Die für die Skalierung benötigten Cluster-Aktionen (vergleiche Kapitel 4.4.3) werden mit Hilfe von drei Funktionen, *scale_role*,*scale_up* und *scale_down*, aufgerufen. Die Funktion *scale_role* wird nur dafür verwendet den Skalierungsfaktor auf eine ganze Zahl zu runden und die Anzahl der Maschinen einer Rolle entsprechend dieser Zahl zu modifizieren. Ist die Zahl

```ruby
def analyse
  Utils::LOG.info(self){"Analysing cluster: #{@cluster.name}"}
  roles = @cluster.get_cluster_roles
  roles = roles - IGNORED_ROLES
  role_scale_value = 0.0
  roles.each do |role|
    role_machines = @cluster.get_machines_by_role(role)
    role_machines.each do |m|
      if m.initialized
        @planner.enqueue_action(RestartAction.new(m)) unless
            has_results(m)
      else
        check_for_initialized(m)
      end
    end
    role_machines.delete_if{|m| !m.initialized }
    if !greedy? && @cluster.machines.select{|m| !m.initialized && m.
        role.downcase!='master'}.count>0
      Utils::LOG.info(self){'Stopped scaling until all machines have
          been initialized'}
      role_machines.clear
    end
    unless role_machines.empty?
      @metrics.select{|m| m.class == BooleanMetric}.each do |metric|
        filtered_machines = metric.analyse(role_machines,@results)
        if filtered_machines.empty?
          Utils::LOG.debug(self){"Boolean Metric #{metric.name} found
              no machines"}
        else
          names = filtered_machines.collect{|m| m.hostname}
          Utils::LOG.info(self){"Metric: #{metric.name} filtered: #{
              names}"}
        end
      end
      metric_count = 0
      @metrics.select{|m| m.class == NumericMetric }.each do |metric|
        metric_count+=1
        ret = metric.analyse(role_machines,@results)
        if ret == 0
          Utils::LOG.debug(self){"Cluster is working within #{metric}
              limits"}
        else
          role_scale_value += ret
        end
      end
      role_scale_value /= metric_count unless metric_count==0
      Utils::LOG.debug(self){"Scaling with factor #{role_scale_value}"}
      scale_role(role_machines, role_scale_value)
    end
    role_scale_value = 0.0
  end
  @results.clear
end
```

Beschreibung 4.11: Analysieren eines Clusters

negativ, wird die Funktion *scale_down* aufgerufen, ist sie positiv, die Funktion *scale_up*. Das Verhalten der Skalierungsfunktionen kann exemplarisch in der Beschreibung 4.12 nachvollzogen werden.

Es wird zunächst geprüft, ob bereits eine Modifizierung des Clusters veranlasst wurde (Zeile 3) und wie diese Skalierung aussieht. Nachdem geklärt wurde, ob die Skalierung innerhalb der Cluster Grenzen liegt (Zeile 12), wird eine Cluster-Aktion in die Warteschlange des Planungsmoduls eingereiht (Zeile 16).

```
1  def scale_up(machines, number=1)
2    mod_number = 0
3    cluster_scale = @machine_modifier[machines.first.role]
4    if !cluster_scale.nil? && cluster_scale > 0
5      if number > cluster_scale.abs
6        mod_number = number-cluster_scale.abs
7      end
8    else
9      mod_number = number
10   end
11   machine = machines.first
12   current_machine_count = @cluster.get_machines_by_role(machines.first.
         role).count+mod_number
13   if current_machine_count > max_machines
14     Utils::LOG.info(self){'Stopping starting of machines due to cluster
           limits'}
15   else
16     @planner.enqueue_action(StartAction.new(machine,mod_number))
17   end
18 end
```

Beschreibung 4.12: Hochskalieren eines Clusters

4.4 Planner

Das Planungsteil eines MAPE-Zyklus kann sehr komplexe Methoden liefern, wie und wann Änderungen am System vorgenommen werden sollen. Da das Hauptaugenmerk dieser Arbeit nicht auf der Effizienz der Skalierungsoperation besteht, sondern gezeigt werden soll, dass eine Skalierung durchgeführt wird, wurde das Planungsmodul möglichst einfach gehalten. Aus diesem Grund besteht es im Wesentlichen aus einem Scheduler der eine Warteschlange implementiert. Durch die Änderungen am Cluster werden in dieser Warteschlange gespeichert und in regelmäßigen Abständen ausgeführt.

Für den Fall, dass in einer späteren Version die Software an dieser Stelle modifiziert werden soll, wurde bereits bei dieser Implementierung darauf geachtet die Erweiterbarkeit möglichst

einfach zu gestalten. Darum unterscheidet das Planungsmodul bereits jetzt zwischen verschiedenen Scheduling-Algorithmen, wobei bisher nur die einfache Warteschlange implementiert wurde.

4.4.1 Funktion: enqueue_action

Für die Kommunikation zwischen Planungs- und Ausführungsmodul wird eine einfache Warteschlange verwendet. Aktionen lassen sich über die Funktion *enqueue_action* in die Warteschlange einreihen (siehe Beschreibung 4.13).

```
1 def enqueue_action(action)
2   @mutex.synchronize do
3     Utils::LOG.debug(self){"Enqueued action: #{action}"}
4     @action_queue.push(action) unless action_already_in_queue(action)
5   end
6 end
```

Beschreibung 4.13: Implementierung des Einreihens von Cluster Aktionen

Damit eine Cluster-Aktion nur einmal ausgeführt wird und nicht mehrfach in die Warteschlange eingereiht wird, wurde eine einfach Abfrage implementiert, die dies verhindern soll (siehe Beschreibung 4.14). Dazu wird jede Aktion gleichen Typs betrachtet und geprüft, ob die betrachtete Aktion die gleiche Maschine betrifft (Zeile 3-5). Anschließend wird der Rückgabewert **ret** zurück gegeben.

```
1 def action_already_in_queue(action)
2   ret = false
3   @action_queue.select{|a| a.class == action.class}.each do |entry|
4     ret |= (entry.machine == action.machine)
5   end
6   ret
7 end
```

Beschreibung 4.14: Verhindern von Dopplung von Aktionen

4.4.2 Funktion: execute_loop

Die Funktion *execute_loop* (siehe Beschreibung 4.15) arbeitet regelmäßig die Aktionen ab, die der Analyser an der Architektur vornehmen möchte. Dabei wird nach eingestellten Scheduling-Algorithmus entschieden wie diese Änderungen durchgeführt werden sollen. In der einfachen Implementierung dieser Arbeit, werden die Aktionen in eine Warteschlange eingereiht und der Reihe nach abgearbeitet. Hierbei gilt ein einfaches FIFO[2]-Prinzip.

[2]First In - First Out

```
1  def execute_loop
2    interval = @config_hash['planner-interval'].nil? ?
         DEFAULT_PLANNER_INTERVAL : Utils::interval_in_seconds(@config_hash
         ['planner-interval'])
3    while @running
4      force_update = false
5      Utils::LOG.info(self){'Planning cluster update execution'}
6      if @scheduler=='default'
7        @mutex.synchronize do
8          force_update = !@action_queue.empty?
9          Utils::LOG.debug(self){'No update actions queued'} unless
               force_update
10         @action_queue.each do |action|
11           action.fire(@executor)
12         end
13         @action_queue.clear
14       end
15       sleep interval
16     end
17     if @scheduler=='round-robin'
18       raise StandardError.new('Round robin scheduling detected')
19     end
20     @executor.update_cluster if force_update && @running
21   end
22 end
```

Beschreibung 4.15: Implementierung der execute_loop

Im Zuge der Implementierung ist klar geworden, dass die Nebenläufigkeit von *main_loop* und *execute_loop* in seltenen Fällen zu Race Conditions[3] führen kann. Aus diesem Grund wurden sowohl bei der Funktion *enqueue_action* (siehe Beschreibung 4.13 Zeile 2) als auch bei der Funktion *execute_loop* (siehe Beschreibung 4.15 Zeile 8) die kritische Variable **@action_queue** durch einen Mutex[4] geschützt.

4.4.3 Cluster-Action

Das Analyse Modul ist in der Lage festzustellen, in welcher Form sich der Cluster verändern muss, um den geforderten Zustand zu erreichen. Damit der Cluster verändert werden kann, stehen dem Modul eine Reihe von Cluster Aktionen (Cluster-Action) zur Verfügung.

Jede Cluster Aktion verfügt über die Funktion *fire*. Innerhalb dieser Funktion wird die eigentliche Aktion durchgeführt und der Cluster verändert.

[3] Gleichzeitiger Zugriff auf kritische Speicherbereiche
[4] Gegenseitiger Ausschluss

Start-Action

Die Start-Action, abgeleitet von der Klasse Cluster-Action, ist in der Lage eine neue Maschine in einen Cluster einzufügen (siehe Beschreibung 4.16). Dazu muss ihr eine alte Maschine übergeben werden, deren Eigenschaften dupliziert werden soll.

Die Notwendigkeit einer Urbild Maschine stellt zwar eine Einschränkung bei der Erzeugung neuer Cluster Maschinen dar, jedoch ist dies bei horizontaler Skalierung zu vernachlässigen. Es wird niemals den Fall geben, dass eine völlig neue Maschine erzeugt werden muss, da neue Maschinen zur Entlastung vorhandener eingesetzt werden.

```
1  class StartAction < ClusterAction
2    def fire(executor)
3      @count.times do
4        executor.cluster.addMachine(duplicate_old_machine)
5      end
6    end
7    def duplicate_old_machine
8      Machine.new(@machine.provider, @machine.storage_cluster, @machine.
           image, @machine.flavor, @machine.role, "#{@machine.hostname}n")
9    end
10 end
```

Beschreibung 4.16: StartAction Klasse

Kill-Action

Eine Kill-Action (siehe Beschreibung 4.17) beschreibt das Entfernen einer speziellen Maschine aus einem Cluster. Dies ist vor allem nötig um Ressourcen in Zeiten von niedriger Auslastung zu sparen. Dabei wird der einzigartiger Hostname einer Maschine, der ihr von dem Cluster Objekt bei ihrer Erstellung gegeben wurde, genutzt.

```
1  class KillAction < ClusterAction
2    def fire(executor)
3      executor.cluster.rm_machine_by_hostname(@machine.hostname)
4    end
5  end
```

Beschreibung 4.17: KillAction Klasse

Initialize-Action

Die Initialisierungsaktion (Initialize-Action, siehe Beschreibung 4.18) setzt den Status einer Maschine auf *initialisiert*. Erkennt das Analysemodul, dass Daten von einer kürzlich gestarteten Maschine gesendet werden, wird diese Funktion aufgerufen.

```
1  class InitializeAction < ClusterAction
2    def fire(executor)
3      executor.cluster.set_initialized_by_ip(@machine.private_ipv4,true)
4    end
5  end
```

Beschreibung 4.18: InitializeAction Klasse

Restart-Action

Die Restart Aktion (Restart-Action) erzwingt den Neustart einer Maschine. Dies ist nötig, wenn eine Metrik festgestellt hat, dass Services oder Programme auf einer Maschine nicht mehr korrekt funktionieren. Außerdem wird eine solche Aktion gestartet, wenn das Analysemodul feststellt, dass eine Maschine aufgehört hat Daten zu senden. So wird sicher gestellt, dass das Monitormodul jederzeit Informationen über alle Maschinen des Clusters besitzt.

Die Restart-Action (siehe Beschreibung 4.19) macht sich dabei eine Eigenschaft des Programms *ecp_deploy* zu Nutzen. Stellt das Programm bei einem Update Befehl fest, dass eine Maschine des Clusters keine IP-Adresse besitzt, wird die entsprechende Maschine herunter gefahren und eine Maschine gleichen Typs neu gestartet. Eine Restart Aktion löscht also das Datenfeld der IP-Adresse der Maschine, wodurch diese bei einem Update neu gestartet wird.

```
1  class RestartAction < ClusterAction
2    def fire(executor)
3      machine = executor.cluster.machines.select{|m| m.hostname==@machine
           .hostname}.first
4      machine.private_ipv4='' unless machine.nil?
5    end
6  end
```

Beschreibung 4.19: RestartAction Klasse

4.5 Executor

Da sich im Laufe der Implementierung herausstellte, dass die Cloud-Init Skripte nicht dynamisch genug sind, musste eine Möglichkeit gefunden werden die Agenten-Software auf allen Mitgliedern der Cluster zu installieren, konfigurieren und zu starten. Die einfachste Möglichkeit dies zu erreichen erschien durch die Verwendung von SSH[5] Aus diesem Grund bietet das Ausführungsmodul (Executor) nicht nur die Möglichkeit Cluster Objekte zu verändern und

[5]Secure Shell

den ecp_deploy zu starten, sondern auch eine Verbindung zu den Maschinen eines Clusters herzustellen und Befehle an diese zu senden.

4.5.1 Funktion: update_cluster

Damit das eigentliche Cluster Update gestartet wird, wird von dem Planungsmodul die Funktion *update_cluster* des Ausführungsmoduls aufgerufen (siehe Beschreibung 4.20). Während durch die Cluster Aktionen verschiedene Änderungen an dem Cluster-Objekt durchgeführt wurden, speichert das Ausführungsmodul die neue Cluster Konfiguration in eine temporäre Datei (Zeile 2-3) und führt anschließend die Update Funktion des ecp_deploy CLI[6] aus (Zeile 5-7). Im letzten Schritt des Updates wird an den Monitor gemeldet, dass das Cluster Objekt sich verändert hat, damit dieser das neue Objekt einladen kann (Zeile 11-12).

```
1  def update_cluster
2    tmp_name="tmp_#{cluster.name}.yml".gsub(" ","").downcase
3    safe_cluster(tmp_name)
4    Utils::LOG.debug(self){'Starting cluster-update'}
5    exec="ecp_deploy update --credentials #{@credential_file} --
         description #{tmp_name} --status #{@cluster_file}"
6    Utils::LOG.debug(self){"Running command: #{exec}"}
7    `#{exec}`
8    Utils::LOG.debug(self){'Cluster updated. Cleaning up files.'}
9    File.delete(tmp_name)
10   Utils::LOG.debug(self){'Waiting for new server to arrive'}
11   load_cluster(@cluster_file)
12   @monitor.has_updated
13 end
```

Beschreibung 4.20: Update Aufruf des Executormoduls

4.5.2 SSH Funktionen

Das Ausführungsmodul ist in der Lage Linuxbefehle über den Konsolen basierten SSH-Client zu senden. Ferner ist es in der Lage das SSH-Protokoll zu nutzen, um Dateien (im wesentlichen Konfigurationsdateien) auf die Cluster Maschinen zu kopieren.

Mit der Funktion *ssh_command*, lassen sich Befehle generieren und über das SSH-Protokoll senden (siehe Beschreibung 4.21).

```
1  def ssh_command(machine,command, user='', params='')
2    login = (@config_hash['ssh-ip-field']=='private'? machine.
         private_ipv4 : machine.public_ipv4)
3    login="#{user}@#{login}" unless user==''
```

[6]Command Line Interface

```
4    timeout = @config_hash['ssh-timeout'].nil? ? DEFAULT_SSH_TIMEOUT :
         @config_hash['ssh-timeout']
5    params+=" -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no
         -o PasswordAuthentication=no"
6    exec = "timeout -s 9 #{timeout} ssh #{params} #{login} '#{command}' <
         /dev/null"
7    Utils::LOG.debug(self){"Executing command: #{exec}"}
8    `#{exec}`
9  end
```

Beschreibung 4.21: Senden eines Befehls per SSH-Client

Die Funktion *copy_file_to_machine* (siehe Beschreibung 4.22) kopiert eine einzelne Datei auf eine Cluster Maschine.

```
1  def copy_file_to_machine(machine,file,user='',params='')
2    ip = (@config_hash['ssh-ip-field']=='private'? machine.private_ipv4 :
         machine.public_ipv4)
3    login = ''+ip
4    login="#{user}@#{login}" unless user==''
5    timeout = @config_hash['ssh-timeout'].nil? ? DEFAULT_SSH_TIMEOUT :
         @config_hash['ssh-timeout']
6    params+=" -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no
         -o PasswordAuthentication=no"
7    exec = "timeout -s 9 #{timeout} scp #{params} #{file} #{login}:~"
8    Utils::LOG.debug(self){"Executing command: #{exec}"}
9    `#{exec}`
10 end
```

Beschreibung 4.22: Kopieren einer Datei per SSH-Protokoll

4.5.3 Funktion: check_for_agents

Die Funktion *check_for_agents* übernimmt eine Sonderrolle in dem MAPE-Zyklus. Zwar ist die Funktion Teil des Ausführungsmoduls, jedoch läuft sie in einer Schleife in ihrem eigenen Prozess. So ist gewährleistet, dass eine Zeitüberschreitung im SSH Protokoll, die Durchführung von Skalierungsoperationen des Ausführungsmoduls nicht unnötig verzögert.

Die Funktion *check_for_agents* stellt sicher, dass alle geforderten Agenten auf allen überwachten Servern laufen und Daten senden (siehe Beschreibung 4.23). Hierfür wird ein einfache Prozessliste über einen Linux Befehl (Zeile 5-7) abgefragt und geprüft, ob der Agentenprozess läuft (Zeile 8). Sollte dies nicht der Fall sein, wird versucht den Agenten auf dieser Maschine zu starten (Zeile 10).

```
1  def check_for_agents
2    Utils::LOG.info(self){'Checking agents on cluster machines.'}
3    @cluster.machines.each do |machine|
4      @agents.each do |agent|
5        command = 'ps aux'
6        params = "-i #{@config_hash['ssh-key-pair']}"
7        status = ssh_command(machine,command,USER,params)
8        agent_running = status.include?("./#{agent.binary}")
9        Utils::LOG.debug(self){ "Agent #{agent.name} running on #{machine
             .hostname}: #{agent_running}" }
10       start_agent(machine,agent) unless agent_running
11     end
12   end
13 end
```

Beschreibung 4.23: Abfrage der Agenten des Executormoduls

4.5.4 Funktion: start_agent

Stellt die Funktion *check_for_agents* fest, dass ein Agent nicht korrekt auf einer Maschine funktioniert, wird dieser Agent von der Funktion *start_agent* gestartet. Dabei wird ein Linux-Befehl zusammen gestellt, der den Agent herunterlädt, konfiguriert und als Daemon startet.

5 Verifikation

Nachdem die Implementation der Software abgeschlossen war, musste eine Möglichkeit gefunden werden ihre Funktionsweise zu überprüfen. Bei dieser Verifikation der Spezifikationen sollte heraus kommen, dass die Software sich wie gefordert verhält und alle Anforderungen erfüllt. Es mussten also folgende Anforderungen überprüft werden:

- FA 1: Überwachung der Cluster Infrastruktur
- FA 2.1: Erkennung von Über- bzw. Unterlast
- FA 2.2: Reagieren auf Über- bzw. Unterlast
- FA 3: Einfache Planung (Scheduling) und Durchführung des Skalierungsprozesses

5.1 Vorbereitung

Um jede der vier Funktionalen Anforderungen zu verifizieren, wurden zunächst Anwendungsfälle definiert, in denen die einzelnen Module reagieren sollten. Anschließend wurden diese Anwendungsfälle in Testszenarien zusammen gefasst, die bei der Überwachung eines Clusters auftreten können. Da im Zuge dieses Projektes kein Cluster unter Realbedingungen existierte, mussten Mittel gefunden werden, die Anwendungsfälle geeignet darzustellen.

5.1.1 Anwendungsfälle

Bei den folgenden Anwendungsfällen, wurden auf vereinfachte Weise die Funktionen der Software überprüft. Es wurden folgende Anwendungsfälle untersucht:

- UC1: Leerlauf - Der Cluster arbeitet ohne Last
- UC2: Überlast - Der Cluster benötigt mehr Rechenleistung als bereit gestellt
- UC3: Unterlast - Der Cluster hat mehr Ressourcen als er im Moment benötigt
- UC4: Serverausfall - Der Cluster verliert im laufenden Betrieb Rechenleistung

Leerlaufkonfiguration

Der erste Anwendungsfall beschrieb die Cluster-Infrastruktur im Betrieb ohne große Last. Der Cluster lief in diesem Fall auf der vom Nutzer eingestellten minimalen Konfiguration. Hierzu kann der Nutzer in der angegebenen Konfigurationsdatei die minimale (und maximale) Anzahl der Maschinen pro Rolle festlegen (siehe Anhang 1). Mit diesem Anwendungsfall wurde gezeigt, dass die funktionale Anforderung **FA 1** erfüllt wurde. Die Cluster-Infrastruktur wurde überwacht, aber keine unnötige horizontale Skalierung vorgenommen.

Überlast

Dieser Anwendungsfall beschrieb die Situation, in der ein hohes Maß an Rechenleistung vom Nutzer angefordert wird. Es musste auf den Maschinen der Cluster-Infrastruktur also eine erhöhte Zahl an Rechen-, so wie Lese- und Schreiboperationen durchgeführt werden. Die Software sollte in diesem Fall durch das Hinzufügen von weiteren Cluster-Worker Nodes reagieren. Mit diesem Anwendungsfall wird gezeigt, dass die funktionalen Anforderungen **FA 2.1**, **FA 2.2** und **FA 3** erfüllt waren.

Unterlast

Die Situation, dass viele Maschinen gestartet sind, jedoch keine Arbeit zu verrichten haben, wird durch den Anwendungsfall *Unterlast* abgebildet. Die Software muss in der Lage sein diese Situation zu erkennen und damit zu beginnen einzelne Maschinen, die nicht benötigt werden, herunter zu fahren. Auch durch diesen Anwendungsfall werden die funktionalen Anforderungen **FA 2.1**, **FA 2.2** und **FA 3** überprüft.

Serverausfall

Durch äußere Umstände kann es passieren, dass Server eines Clusters nicht mehr erreichbar sind. Der Anwendungsfall *Serverausfall* beschreibt die Situation, dass Maschinen ungeplant abgeschaltet werden, oder ihre Arbeit einstellen. In diesem Fall muss die Software erkennen, dass ein Problem vorliegt und durch das Starten von neuen Maschinen, die ausgefallenen ersetzen. Die korrekte Funktionsweise der funktionalen Anforderungen **FA 1** und **FA 3** werden von diesem Anwendungsfall überdeckt.

5.1.2 Benötigte Software

stress

Um Last auf Maschinen zu erzeugen, wurde der Workload-Generator stress [Wat17] von Amos Waterland verwendet. Dieses einfache Kommando Zeilen Programm erzeugt über angegebene Parameter CPU-Last, Lese- und Schreibzugriffe auf Festplatte und Speicher.

ecp_stress

Damit das Programm *stress* mit verschiedenen Konfigurationen zu verschiedenen Zeiten geplant auf Servern gestartet werden kann, wurde die kleine Anwendung *ecp_stress* entwickelt. *Ecp_stress* bietet einen einfachen Scheduling Algorithmus in Form einer Warteschlange. Über die Eingabe von JSON-Dateien, so wie einer bestehenden Cluster-Datei des Programms *ecp_deploy*, können die unter Kapitel 5.1.1 aufgelisteten Anwendungsfälle simuliert werden.

Eine genaue Beschreibung des Programms *ecp_stress* und seiner Arbeitsweise findet sich in Anhang 1.2.

5.2 Durchführung

Als alle Anwendungsfälle definiert waren, wurde die Reihenfolge der Tests festgelegt. Zu jedem Testszenario wurde zunächst festgehalten, welche Konfiguration des Clusters als Basis des Tests dienen sollte und welche Metriken verwendet werden sollten.

5.2.1 Test: Leerlaufkonfiguration

Der Test *Leerlaufkonfiguration* sollte zeigen, dass die vom Nutzer eingestellten Grenzen des Clusters eingehalten werden. Dazu wurde bewusst ein Cluster erzeugt, der mit mehreren Maschinen Rollen gegen diese Limitierung verstoßen hat.

Cluster Konfiguration

Zunächst mussten die Grenzen des Clusters mit Hilfe der Konfigurationsdatei des *ecp_autoscale* Programms festgelegt werden. Die Minimal- und Maximalwerte wurden unter dem Abschnitt *[Analyser]* in die Konfigurationsdatei eingetragen und über einen Parameter des CLI an das Programm übergeben. In Beschreibung 5.1 sind die wichtigen Eigenschaften festgehalten. Ein Beispiel einer kompletten Konfigurationsdatei und aller Parameter findet sich in Anhang 1.

```
1 [..]
2 [Analyser]
3 greedy=false
4 min-machine=2
5 max-machine=3
6 machine-timeout=2m
7 [..]
```

Beschreibung 5.1: Konfiguration *ecp_autoscale* bei Test: **Leerlauf**

Der erzeugte Cluster selbst wurde nun mit zwei Rollen, *LowPerformer* und *HighPerformer*, gestartet. Die Rolle *LowPerformer* überschreitet mit 5 Maschinen die obere Grenze des Clusters um zwei Maschinen, die Rolle *HighPerformer* unterschreitet die untere Grenze um eine Maschine. In Tabelle 5.1 ist die ursprüngliche Cluster Zusammensetzung angegeben.

Rolle	Anzahl
Master	1
LowPerformer	5
HighPerformer	1

Tabelle 5.1: Cluster Konfiguration vor Programmstart bei Test: **Leerlauf**

Ergebnisse

Nach dem Start des Programms, stellte das es direkt fest, dass die Cluster Grenzen nicht eingehalten wurden. In Beschreibung 5.2 sind die wesentlichen Passagen der Log Datei des Programms angegeben. In Zeile 2 lässt sich erkennen, dass das Programm beim Initialisieren erkannt hat, dass bei der Rolle *LowPerformer* zwei Maschinen zu viel sind. Korrekterweise hat das Programm reagiert, indem zwei mal das Ereignis *KillAction* erzeugt und eingereiht wird.

Im weiteren Verlauf des Programms, wird erkannt, dass die Rolle *HighPerformer* zu wenige Maschinen besitzt (Zeile 5) und eine Maschine gestartet werden muss. Ein entsprechendes Ereignis findet um 10:31:19 statt (Zeile 6).

```
1  [..]
2  10:31:19 DEBUG #<Analyser>: Role lowPerformer out of boundaries. Removing 2
       server
3  10:31:19 DEBUG #<Planner>: Enqueued action: #<KillAction>
4  10:31:19 DEBUG #<Planner>: Enqueued action: #<KillAction>
5  10:31:19 DEBUG #<Analyser>: Role highPerformer out of boundaries. Starting 1
       server
6  10:31:19 DEBUG #<Planner>: Enqueued action: #<StartAction>
7  [..]
8  10:31:19 DEBUG #<KillAction>: Fired kill action
9  10:31:19 DEBUG #<Executor>: Removing machine: lowPerformer
10 10:31:19 DEBUG #<KillAction>: Fired kill action
11 10:31:19 DEBUG #<Executor>: Removing machine: lowPerformer
12 10:31:19 DEBUG #<StartAction>: Fired. Duplicating machine of highPerformer 1
       times
13 10:31:19 DEBUG #<Executor>: Starting new machine highPerformer
14 [..]
```

Beschreibung 5.2: Verhalten von *ecp_autoscale* bei Test: **Leerlauf**

Nach der erfolgreichen Skalierung der Cluster Maschinen, stellte sich ein stabiler Zustand ein, die endgültige Zusammensetzung der Maschinen ist in Tabelle 5.2 festgehalten.

Rolle	Anzahl
Master	1
LowPerformer	3
HighPerformer	2

Tabelle 5.2: Cluster Konfiguration nach Skalierung bei Test: **Leerlauf**

Rolle	Anzahl
Master	1
LowPerformer	1

Tabelle 5.3: Cluster Konfiguration vor Programmstart bei Test: **Überlast mit Einzelmetrik**

5.2.2 Test: Überlast mit Einzelmetrik

Bei dem Test *Überlast mit Einzelmetrik* wurde geprüft, wie sich das Programm verhält, wenn auf einer Maschine des Clusters besonders viele Ressourcen angefordert werden. Zu diesem Zweck wurde eine inverse numerische Metrik *CPU-Idle* definiert, die prüft, ob der durchschnittliche Anteil des Leerlaufprozesses aller Maschinen einer Rolle über vierzig Prozent liegt. Sobald dieser Wert unterschritten wird, empfiehlt die Metrik mit dem Starten von Maschinen zu beginnen.

Cluster Konfiguration

Um einen einfachen Cluster zu haben, wurde für jede Rolle eine minimale Anzahl von **einer** Maschine festgelegt. Anschließend wurde der Cluster mit der in Tabelle 5.3 abgebildeten Konfiguration gestartet.

Metrik Konfiguration

Damit die CPU-Auslastung der Maschinen überwacht werden konnte, musste zunächst eine Metrik definiert werden, die der Monitor abfragen und überprüfen kann. Die in Beschreibung 5.3 definierte Metrik liest dazu den prozentualen Anteil des Leerlaufprozess jeder Maschine aus. Im Durchschnitt über alle Maschinen sollte dieser zwischen 40 % und 85 % liegen.

Zu beachten ist, dass diese Definition eine inverse Metrik ist. Überschreitet der gemessene Wert den minimal Wert, so wird durch herunter skalieren reagiert. Bei Unterschreiten des angegebenen Maximalwertes skaliert das Programm den Cluster herauf.

```
{
  "name": "CPU-Idle",
  "type": "numeric",
  "checks": {
    "min": 0.85,
    "max": 0.4,
    "inverse":true,
    "has-total":false
  },
  "aggregations": [
    {
      "name": "cpu-idle",
      "field": "system.cpu.idle.pct",
      "func": "avg"
    }
  ]
}
```

Beschreibung 5.3: Metrik Definition **CPU-Idle**

Stress Konfiguration

Damit die Last erzeugt werden konnte, musste das Programm *ecp_stress* mit einer entsprechenden Konfiguration genutzt werden. Da die Maschinen der Rolle *LowPerformer* über vier CPUs verfügten, mussten mindestens vier Last erzeugende Prozesse gestartet werden, damit eine Auslastung von hundert Prozent erreicht werden konnte. Wie in der Beschreibung 5.4 zu sehen ist, starteten dreißig Sekunden nach dem Beginn der Tests vier Last erzeugende Prozesse, die jeweils 600 Sekunden andauerten.

```
[{
  "machine": "544dbf1fc0",
  "start": 30,
  "duration": 600,
  "stress_parameter":{
    "cpu":4,
    "io":1,
    "vm": 1,
    "vm-bytes": "1024k",
    "hdd":1
  }
}]
```

Beschreibung 5.4: Stress-Schedule für Test: **Überlast mit Einzelmetrik**

Ergebnisse

Der Test begann um 11:14 und wie zu erwarten war, war der Anteil des Leerlaufprozess der Maschine *Nr. 1* zunächst bei fast neunzig Prozent. Der ausgelesene Wert der CPU-Idle

Maschine	Nr.	11:14	11:15	11:16	11:17	11:18	11:19	11:20	11:21	11:22	11:23	11:24	11:25	11:26
544dbf1fc0	1	11,40	68,70	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	44,07	11,40
83e816c0a8	2						35,60	22,50	9,12	9,40	9,45	9,30	9,22	9,33
ed4b01016f	3					42,75	10,13	21,69	8,67	8,57	8,90	8,70	8,73	8,95
⌀		11,40	68,70	100,00	100,00	71,38	48,58	48,06	39,26	39,32	39,45	39,33	20,67	9,89

Tabelle 5.4: Werte der CPU-Idle Metrik bei Test: **Überlast mit Einzelmetrik** in Prozent

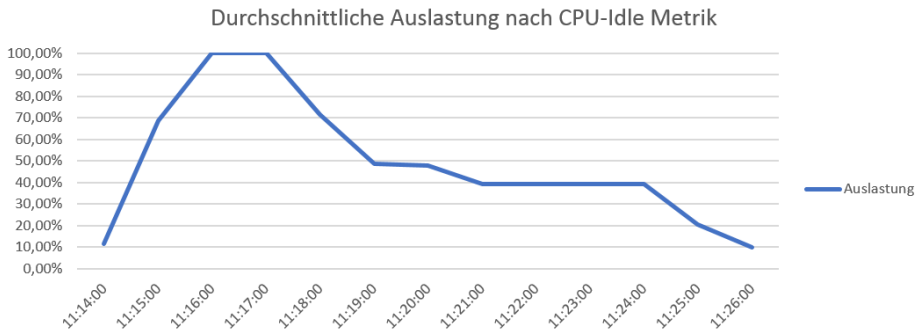

Abbildung 5.1: Auslastung bei Test: **Überlast mit Einzelmetrik** in Prozent

Metrik war bei 11,40%. Bis um 11:16 steig die ermittelte Auslastung auf einhundert Prozent und das Programm reagierte durch anfordern von zwei weiteren Maschinen. Zwischen 11:18 und 11:19 klinkten sich Maschinen *Nr.2* und *Nr. 3* in den Cluster ein. Eine zeitliche Entwicklung des Anteils des Leerlaufprozess jeder Maschine kann der Tabelle 5.4 entnommen werden.

Der durchschnittliche Anteil der Leerlaufprozesse ist letztendlich ausschlaggebend für die Skalierung der *CPU-Idle* Metrik. Ein zeitlicher Verlauf dieses Messwertes kann der letzten Zeile von Tabelle 5.4 und Abbildung 5.1 entnommen werden.

Deutlich zu erkennen ist, dass um 11:19 der gemessen Wert sich innerhalb der Grenzen der Metrik befand und das Cluster einen stabilen Zustand erreichte. Um 11:25, zehn Minuten nach Beginn des Tests, endete das Programm *ecp_stress* und die Last auf den Maschinen verringerte sich. Im weiteren Verlauf (nicht mehr dargestellt) reagierte das Programm durch herunterfahren von ungenutzten Maschinen.

Alle relevanten Informationen aus der Ereignis Log-Datei dieses Tests sind in Anhang: Beschreibung 6 festgehalten.

5.2.3 Test: Überlast mit Metrikkombination

Damit gezeigt werden kann, dass das Programm *ecp_autoscale* nicht nur mit einer einzelnen Metrik genutzt werden kann, sondern auch eine Kombination von Metriken korrektes Skalieren verursachen können, wurde ein weiterer Versuch zum Anwendungsfall Überlast unternommen.

Rolle	Anzahl
Master	1
LowPerformer	2

Tabelle 5.5: Cluster Konfiguration bei Start von Test: **Überlast mit Metrikkombination**

In diesem Fall wurde das Programm *ecp_stress* so konfiguriert, dass nicht nur eine hohe CPU-Last, sondern auch ein erhöhter Bedarf an Speicher (RAM) vorliegt. Die genaue Konfiguration kann Beschreibung 5.6 entnommen werden.

Cluster Konfiguration

Für diesen Test wurde ein einfacher Cluster herauf gefahren. Die ursprüngliche Konfiguration kann Tabelle 5.5 entnommen werden.

Metrik Konfiguration

Zusätzlich musste neben der *CPU-Idle* Metrik auch eine Metrik definiert werden, die die prozentuale Auslastung des Arbeitsspeichers einer Maschine betrachtet. Diese Metrik, im Folgenden *Memory-Use* genannt, wurde mit Hilfe der unter Beschreibung 5.5 angegebenen JSON-Datei festgehalten.

Im Gegensatz zur Metrik *CPU-Idle* (siehe Beschreibung 5.3) ist *Memory-Use* keine inverse Metrik. Ihr Maximalwert darf also nicht überschritten werden, der Minimalwert nicht unterschritten. Die Metrik stellt also sicher, dass die Speicherauslastung der Maschinen im Mittel zwischen 10 % und 50 % liegt.

Stress Konfiguration

In Beschreibung 5.6 ist festgehalten wie das Programm *ecp_stress* angesteuert wird. In Zeile 6 und Zeile 17 ist definiert, dass es, wie auch schon im letzten Test, vier Prozesse geben wird, die Last auf der CPU erzeugen. Zusätzlich werden auf jeder Maschine drei Prozesse erzeugt, die jeweils 512mb RAM anfordern (Zeile 8-9 und Zeile 19-20).

Ergebnisse

In dem Log der Ereignisse des Programms *ecp_autoscale* (siehe Anhang: Beschreibung 7) lässt sich der Verlauf des Tests nachvollziehen. Der Test begann um 11:46:26 und zunächst

```
{
  "name": "Memory-Use",
  "type": "numeric",
  "checks": {
    "min": 0.1,
    "max": 0.5,
    "has-total":true
  },
  "aggregations": [
    {
      "name": "memory-used",
      "field": "system.memory.actual.used.bytes",
      "func": "avg"
    },
    {
      "name": "memory-total",
      "field": "system.memory.total",
      "func":"max"
    }
  ]
}
```

Beschreibung 5.5: Metrik Definition **Memory-Use**

```
[{
  "machine": "745c86dbbf",
  "start": 1,
  "duration": 600,
  "stress_parameter":{
    "cpu": 4,
    "io": 0,
    "vm": 3,
    "vm-bytes": "512m",
    "hdd": 0
  }
},{
  "machine": "96a64679fd",
  "start": 30,
  "duration": 530,
  "stress_parameter":{
    "cpu": 4,
    "io": 0,
    "vm": 3,
    "vm-bytes": "512m",
    "hdd": 0
  }
}
]
```

Beschreibung 5.6: Stress-Schedule für Test: **Überlast mit Metrikkombination**

Maschine	Nr.	11:47	11:48	11:49	11:50	11:51	11:52	11:53	11:54	11:55	11:56	11:57	11:58
96a64679fd	1	11,57	65,93	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	17,63	10,57
745c86dbbf	2	9,30	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	8,50
50a47640dc	3					37,57	8,90	9,47	9,20	97,37	78,07	10,23	12,37
0cc95bb3d3	4					100,00	7,67	7,20	7,03	6,87	7,27	7,30	8,10
⌀		10,43	82,97	100,00	100,00	84,39	54,14	54,17	54,06	76,06	71,33	33,79	9,88

Tabelle 5.6: Werte der CPU-Idle Metrik bei Test: **Überlast mit Metrikkombination** in Prozent

Maschine	Nr.	11:47	11:48	11:49	11:50	11:51	11:52	11:53	11:54	11:55	11:56	11:57	11:58
96a64679fd	1	11,70	62,97	88,53	88,40	88,40	88,40	88,40	88,40	88,37	88,40	11,60	11,60
745c86dbbf	2	12,00	88,67	88,60	88,60	88,70	88,80	88,77	88,80	88,70	88,70	88,77	11,80
50a47640dc	3					11,07	11,57	11,87	12,13	13,33	13,67	13,20	13,20
0cc95bb3d3	4					5,60	11,50	11,63	11,87	12,07	12,20	12,20	12,30
⌀		11,85	75,82	88,57	88,50	48,44	50,07	50,17	50,30	50,62	50,74	31,44	12,23

Tabelle 5.7: Werte der Memory-Used Metrik bei Test: **Überlast mit Metrikkombination** in Prozent

wurde korrekt erkannt, dass der Cluster mehr Ressourcen hat, als er benötigte. Die vom Nutzer festgelegte minimale Anzahl an Worker-Nodes verhinderte jedoch, dass Maschinen herunter gefahren werden.

Um 11:47:40 wurde das Programm *ecp_stress* mit der in Beschreibung 5.6 angegebenen Konfiguration gestartet was zu einem erhöhten Ressourcen Bedarf auf den Maschinen *Nr. 1* und *Nr. 2* führte. Dieser erhöhte Bedarf wurde von dem Monitormodul um 11:47:59 korrekt erkannt. Zu diesem Zeitpunkt liefen aber beide Metriken noch in den vom Nutzer eingestellten Grenzen.

Um 11:48:35 überschritt die Messung des Speicherbedarfs diese Grenzen und der Analyser forderte zunächst eine Maschine an. Knapp zehn Sekunden später, um 11:48:45, erkannte auch die Metrik für die CPU-Auslastung die erhöhten Anforderungen und empfahl ebenfalls eine Maschine hoch zu fahren. Es wurden insgesamt zwei weitere Maschinen herauf gefahren, bis die Messungen um 11:57:16 ergaben, dass sich nun die Metriken wieder in den vorher definierten Grenzen befinden.

Ein Verlauf der vom Monitor ausgelesenen Werte befinden sich in den Tabellen 5.6 und 5.7

Nachdem das Programm *stress* auf den beiden Maschinen um 11:58:41 endete, zehn Minuten nach Start der ersten Ressourcenanforderung, begann der zweite Teil des Tests. Dieser Test bildete den Anwendungsfall *Unterlast* ab und ist näher im folgenden Abschnitt beschrieben.

5.2.4 Test: Unterlast

Ein Anwendungsfall der direkt mit den vorherigen Tests mit abgedeckt werden konnte, war das Verhalten bei Unterlast. Nachdem die Lasterzeugung durch *ecp_stress* beendet war, wur-

Rolle	Anzahl
Master	1
LowPerformer	4

Tabelle 5.8: Cluster Konfiguration zu Beginn des Tests: **Unterlast**

de das Verhalten des Programms nach dem letzten Test (Test: Überlast mit Metrikkobination) weiter beobachtet. Das erwartet Verhalten war, dass das Programm erkennt, dass die besondere Lastsituation vorbei ist und mit herunterfahren von Maschinen reagiert.

Cluster Konfiguration

Auch wenn die ursprüngliche Cluster Konfiguration in diesem Test die gleiche wie bei dem Test: *Überlast mit Metrikkombination* war, ist der relevante Zustand des Clusters nach der automatischen Hochskalierung. Der Ausgangszustand für diesen Test ist in Tabelle 5.8 abzulesen.

Stress Konfiguration

Für diesen Test wurde keine gesonderte Konfiguration des Programms *ecp_stress* vorgenommen. Es gilt die gleiche Konfiguration wie unter Beschreibung 5.6 angegeben. Zu beachten ist jedoch, dass der für diesen Test relevante Zeitraum von 11:58 Uhr bis 12:02 Uhr war und die von *ecp_stress* erzeugte Last um 11:57 endete.

Ergebnisse

Der Beobachtungszeitraum dieses Tests, begann nach beenden der ecp_stress Prozesse auf den Maschinen um 11:57:41. Zu diesem Zeitpunkt befand sich der Cluster in einem stabilen Zustand und es mussten keine Skalierungsoperationen vorgenommen werden. Im Laufe der nächsten Minute nahm der Monitor das Abnehmen des Ressourcenbedarfs wahr. Die vom Monitor gemessenen Werte der CPU-Idle Metrik sind in Tabelle 5.9 aufgeführt. Deutlich zu erkennen ist, dass um 11:59:05 von der CPU-Idle Metrik korrekt erkannt wird, dass zu viele Ressourcen zur Verfügung stehen und Maschinen heruntergefahren werden müssen. Nachdem die Skalierung auf Grund der Cluster Grenzen (zwei laufende Maschinen pro Rolle) um 11:59:31 endete, wurde auch die Aufzeichnung der Testdaten beendet.

Die für die Skalierung relevanten Informationen über die Aktionen des Programms sind in Anhang: Beschreibung 8 festgehalten.

Insgesamt ergibt sich aus den beiden Tests Unterlast und Überlast mit Metrikkombination ein Lebenszyklus des Clusters von 11:46:00-12:02:00. Ein visualisierter Verlauf der gesamten Messwerte der beiden Metriken, befindet sich in Anhang: Abbildung 1.

Maschine	Nr.	11:57	11:58	11:59	12:00
96a64679fd	1	17,63	10,57		
745c86dbbf	2	100,00	8,50	8,17	
50a47640dc	3	10,23	12,37	12,67	11,37
0cc95bb3d3	4	7,30	8,10	7,73	7,70
⌀		33,79	9,88	9,52	9,53

Tabelle 5.9: Werte CPU-Idle Metrik bei Test: **Unterlast** in Prozent

5.2.5 Test: Serverausfall

Mit dem Test *Serverausfall* musste gezeigt werden, dass das Programm eine plötzlich abreißende Verbindung zu einer Cluster Maschine erkennt. Die entsprechende Reaktion des Programms sollte es sein eine Maschine gleichen Typs hoch zu fahren und in den Cluster einzuklinken.

Cluster Konfiguration

Für diesen Test wurde zunächst ein einfacher Cluster erstellt. Dieser Cluster bestand aus einer Master-Node, so wie zwei Worker-Nodes des Typs **LowPerformer**. Da der Cluster sich im Leerlauf befand, entsprach diese Konfiguration auch genau der eingestellten Minimalgröße eines Clusters.

Stress Konfiguration

Das Herunterfahren der Maschine wurde über einen einfachen Konsolenbefehl realisiert, der über eine SSH-Verbindung gesendet wurde. Die entsprechende Konfiguration von ecp_stress befindet sich in der JSON-Konfiguration 5.7. Die Maschine mit dem Hostnamen **946df4f7d7** (vergleiche Zeile 2), bekam nach 20 Sekunde (Zeile 3) den Befehl sofort herunter zu fahren (Zeile 5).

```
1 [{
2   "machine": "946df4f7d7",
3   "start": 30,
4   "duration": 1,
5   "command": "sudo shutdown -h now"
6 }]
```

Beschreibung 5.7: Stress-Schedule für Test: **Serverausfall**

Ergebnisse

Wie in dem Auszug aus der Log-Datei des Programms *ecp_stress* in Beschreibung 5.8 zu sehen ist, wurde um 10:38:02 der Befehl an die Maschine **946df4f7d7** gesendet.

```
1  10:38:02 DEBUG #<Scheduler>: Fired: Scheduler::CommandEntry: sudo shutdown -h
       now Begin: 30
```

Beschreibung 5.8: ecp_stress Log bei Test: **Serverausfall**

In dem Auszug aus des Ereignis-Log des Programms *ecp_autoscale* (siehe Beschreibung 5.9) geht hervor, dass das Programm um 10:39:03 erkannte, dass die Maschine *946df4f7d7* neu gestartet werden musste (Zeile 5). Um 10:39:17 wurde das Programm *ecp_deploy* aufgerufen und damit das Update des Clusters ausgelöst. Dieses dauerte bis 10:39:50. Nach kurzer Zeit (um 10:41:57) war die neu gestartet Maschine *5e345195b1* vollständig initialisiert der Cluster arbeitete wieder in den definierten Grenzen (Zeile 13-15).

```
1  [..]
2  10:39:03 DEBUG -- #<Analyser>: Storing result for machine: 946df4f7d7
3  10:39:03 DEBUG -- #<Analyser>: Storing result for machine: d869bc88d3
4  10:39:03  INFO -- #<Analyser>: Analysing cluster: My Cluster
5  10:39:03 DEBUG -- #<Planner>: Enqueued action: #<RestartAction>
6  [..]
7  10:39:17  INFO -- #<Executor>: Starting cluster-update
8  10:39:17 DEBUG -- #<Executor>: Running command: ecp_deploy update --credentials
       credentials.json --description tmp_mycluster.yml --status test.yml
9  [..]
10 10:41:56 DEBUG -- #<Analyser>: Storing result for machine: d869bc88d3
11 10:41:57 DEBUG -- #<Analyser>: Storing result for machine: 5e345195b1
12 10:41:57  INFO -- #<Analyser>: Analysing cluster: My Cluster
13 10:41:57 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.35682499999999995
14 10:41:57 DEBUG -- #<Analyser>: Cluster is working within InverseNumericMetric:
       CPU-Idle limits
15 10:41:57 DEBUG -- #<Analyser>: Scaling with factor 0.0
```

Beschreibung 5.9: ecp_autoscale Log bei Test: **Serverausfall**

5.2.6 Test: Zeitüberschreitung

Der Test *Zeitüberschreitung* prüft das Verhalten der Software in dem Fall, dass eine angeforderte Maschine nicht oder unvollständig startet. Zu diesem Zweck wurde zunächst ein Cluster im Leerlauf erzeugt, der die minimal Konfiguration unterschreitet. Das vom Programm gewünschte Verhalten ist, so viele Maschinen an zu fordern, dass die eingestellten Cluster Grenzen eingehalten werden.

Eine angeforderte Maschine benötigt mehrere Minuten bis sie sich in den Cluster einklinkt. In dieser Zeit wurde die Maschine manuell über das CLI des benutzten Cloud Providers (bei

Rolle	Anzahl
Master	1
LowPerformer	1

Tabelle 5.10: Cluster Konfiguration bei Test: **Zeitüberschreitung**

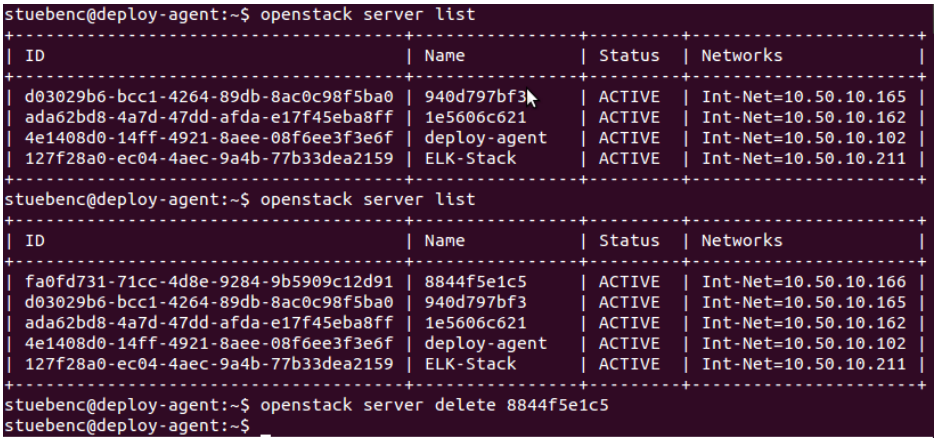

Abbildung 5.2: Manueller Eingriff bei Test: **Zeitüberschreitung**

diesem Test *OpenStack*) abgeschaltet. Die Folge ist, dass das Programm vergeblich auf die vollständige Initialisierung der Maschine wartet.

Cluster Konfiguration

Es wurde zunächst ein minimaler Cluster herauf gefahren, der nur aus einem Master und einer Maschine der Rolle *LowPerformer* bestand (siehe Cluster Konfiguration in Tabelle 5.10).

Manueller Eingriff

Vor Beginn des Tests wurden mit Hilfe des CLI alle laufenden Maschinen aufgelistet. Wie Abbildung 5.2 zu sehen ist, liefen zunächst die Maschinen *1e5606c621* und *940d797bf3* im Cluster.

Als das Programm gestartet wurde, stellte es fest, dass eine weitere Maschine herauf gefahren werden musste, um die Grenzen (siehe Beschreibung 5.10) des Clusters einzuhalten. Nach einem Update Aufruf des *ecp_deploy* Programms, wurde die Maschine *8844f5e1c5* gestartet.

Nach dem sofortigem manuellen Entfernen dieser Maschine, wurde das Verhalten des Programms beobachtet.

```
1  [..]
2  [Analyser]
3  greedy=false
4  min-machine=2
5  max-machine=4
6  machine-timeout=3m
7  [..]
```

Beschreibung 5.10: Konfiguration von *ecp_autoscale* bei Test: **Zeitüberschreitung**

Ergebnis

Wie bereits in dem Test *Leerlaufkonfiguration* (vergleiche Kapitel 5.2.1) reagierte das Programm mit dem Anfordern von einer Maschine, damit die Grenzen des Clusters eingehalten wurden. Das Ereignis-Log des Programms *ecp_autoscale* ist in Beschreibung 5.11 angegeben.

Deutlich zu erkennen ist, dass ungefähr drei Minuten nach Starten einer Maschine (Zeile 7) erkannt wurde, dass die Maschine nicht richtig herauf gefahren ist und mit Neustarten der Maschine reagiert werden muss (Zeile 9-10).

```
1  [..]
2  09:47:33 DEBUG #<Analyser>: Role lowPerformer out of boundaries. Starting 1
       server
3  09:47:33 DEBUG #<Planner>: Enqueued action: #<StartAction>
4  09:47:33 INFO #<Planner>: Planning cluster update execution
5  09:47:33 DEBUG #<StartAction>: Fired. Duplicating machine of lowPerformer 1
       times
6  [..]
7  09:48:11 DEBUG #<Executor>: Cluster updated. Cleaning up files.
8  [..]
9  09:51:45 DEBUG #<Analyser>: Time since machine 8844f5e1c5 started exceeds
       timeout limit
10 09:51:45 DEBUG #<Planner>: Enqueued action: #<RestartAction>
11 [..]
```

Beschreibung 5.11: Ereignis-Log bei Test: **Zeitüberschreitung**

5.2.7 Test: Netzwerklast

Bei den bisherigen Tests wurden nur Metriken des Agenten MetricBeat genutzt. Da die Implementierung aber auch vorsieht, dass andere der von Elastic bereit gestellten Agenten für das Programm *ecp_autoscale* nutzbar sind, wurde noch ein weiterer Versuch zum Thema Netzwerklast durchgeführt.

Ziel dieses Tests ist es nachzuweisen, dass eine Boolsche Metrik gemäß der Konfiguration einen Zustand des Clusters erkennt und reagiert. Hierzu wurde der Agent **Paketbeat** genutzt, der den vom etcd2-Service genutzten Port 2379 auf allen Maschinen überwacht. So ließen sich Informationen über die Antwortzeit einer jeden HTTP-Anfrage auf diesem Port auswerten.

Rolle	Anzahl
Master	1
LowPerformer	2

Tabelle 5.11: Cluster Konfiguration bei Test: **Netzwerklast**

Metrik Konfiguration

Die Boolsche Metrik (siehe Beschreibung 5.12) sieht vor, dass sobald die durchschnittliche Antwortzeit einer HTTP-Anfrage einen bestimmten Wert (hier 20ms, Zeile 6) überschreitet, das Programm *ecp_autoscale* dies erkennt und eine Warnung ausgibt.

```
1  [{
2    "name": "Http-response-time",
3    "type": "boolean",
4    "checks": {
5      "count":{
6        "max":20
7      }
8    },
9    "aggregations":[
10     {
11       "name": "http-error",
12       "field": "responsetime",
13       "func":"avg"
14     }
15   ]
16 }]
```

Beschreibung 5.12: Metrik Konfiguration bei Test: **Netzwerklast**

Damit auf dem Netzwerkadapter einer Maschine eine Last durch HTTP-Anfragen erzeugt werden konnte, wurde das Programm *ecp_stress* um die HTTP-Flut Funktion erweitert. Eine genaue Beschreibung von dieser Funktion befindet sich in Anhang 1.2.

Cluster Konfiguration

Für den Test **Netzwerklast** wurde zunächst ein sehr einfacher Cluster herauf gefahren. Die genaue Cluster-Beschreibung ist Tabelle 5.11 zu entnehmen.

Ergebnisse

Wie in Beschreibung 5.13 zu sehen ist begann der Test nach erfolgreiche Initialisierung aller Maschinen um 11:26:44. Zunächst befand sich die Metric http-response-time in den vom Nutzer festgelegten Grenzen (Zeile 3).

Um 11:27:25 wurde mit der Flut von HTTP-Anfragen auf der Maschine *bbdbcb3359* begonnen. Kurze Zeit später, um 11:29:39 erkannte die Metrik eigenständig, dass die Antwortzeit auf dem Port 2379 auf Maschine *bbdbcb3359* bereits bei 36,5 ms lag und gab die Maschine in die Liste der gefilterten Maschinen (Zeile 7). Im Zuge der Flut an Befehlen stieg dieser Wert auf eine maximale Antwortzeit von 86,4 ms um 11:30:05. Nach dieser Spitze senkte sich der Wert ab und war ab 11:30:36 wieder in den vom Nutzer definierten Grenzen (Zeile 14)

```
1  [..]
2  11:26:44  INFO -- #<Analyser>: Analysing cluster: My Cluster
3  11:26:44 DEBUG -- #<Analyser>: Boolean Metric Http-response-time found no
       machines
4  [..]
5  11:29:39  INFO -- #<Analyser>: Analysing cluster: My Cluster
6  11:29:39 DEBUG -- BooleanMetric:Http-response-time: Boundary exceeded:
       36.534351145038165 > 20 for machine bbdbcb3359
7  11:29:39  INFO -- #<Analyser>: Metric: Http-response-time filtered: ["
       bbdbcb3359"]
8  [..]
9  11:30:05  INFO -- #<Analyser>: Analysing cluster: My Cluster
10 11:30:05 DEBUG -- BooleanMetric:Http-response-time: Boundary exceeded:
       86.38208955223881 > 20 for machine bbdbcb3359
11 11:30:05  INFO -- #<Analyser>: Metric: Http-response-time filtered: ["
       bbdbcb3359"]
12 [..]
13 11:30:36  INFO -- #<Analyser>: Analysing cluster: My Cluster
14 11:30:36 DEBUG -- #<Analyser>: Boolean Metric Http-response-time found no
       machines
15 [..]
```

Beschreibung 5.13: ecp_autoscale Log bei Test: **Netzwerklast**

5.3 Fazit

Mit der Durchführung der Tests konnte gezeigt werden, dass die entwickelte Software die funktionalen Anforderungen erfüllt. Zwar ist die Reaktionszeit für sich verändernde Lastverhältnisse mit mehren Minuten relativ lang, jedoch fand die horizontale Skalierung des Clusters automatisch und ohne Eingreifen des Benutzers statt. Auch wenn die verwendeten Metriken nur exemplarisch genutzt wurden, ist die Architektur so entwickelt, dass es kein Problem sein sollte die Software um weitere Metriken und Metrikkombinationen zu erweitern.

Problemlos funktionierte das Erkennen von abgestürzten Maschinen, sowohl solche, die nicht korrekt starten, als auch solche, die im Laufenden Betrieb das Arbeiten einstellten. In beiden Fällen konnte das Programm schnell durch das Neustarten solcher Maschinen reagieren.

Anhang

1 Anhänge zur Implementierung

1.1 Hilfsfunktionen

Die Klasse **Utils** bietet eine Reihe von Hilfsfunktionen, die an mehreren Stellen des Programms genutzt werden. So bietet Utils eine zentrale Möglichkeit um Log Einträge zu generieren. Mit diesen Log Einträgen kann das Verhalten des Programms überwacht und kontrolliert werden.

Klasse: ConfigParser

Es wurde eine einfacher Parser für eine Konfigurationsdatei geschrieben. Innerhalb dieser Konfigurationsdatei können Standardwerte und Parameter festgelegt werden, die das Verhalten der Software beeinflussen. Ein Beispiel für eine solche Konfigurationsdatei ist unter Anhang: Beschreibung 1 angegeben.

```
1  # Default configuration
2  [General]
3  #verbose=false
4  [Monitor]
5  ip=10.50.10.211
6  ignored-roles=master
7  interval=5s
8  [Analyser]
9  greedy=false
10 min-machine=2
11 max-machine=4
12 machine-timeout=3m
13 [Planner]
14 interval=10s
15 [SSH]
16 timeout=10s
17 ip-field=private
18 key-pair=~/.ssh/deployer-key.pem
19 [Agent]
20 time-frame=1m
```

Beschreibung 1: Konfigurationsdatei der Klasse ConfigParser

Konfigurationsoptionen

In Anhang: Tabelle 1 sind alle möglichen Parameter festgehalten, die in der Konfigurationsdatei festgelegt werden können.

1.2 Lasterzeugung: ecp_stress

Das kleine Programm *ecp_stress* wurde dazu entwickelt Lastsituationen auf Cluster Maschinen zu simulieren. Im Wesentlichen besteht es aus einer sequenzieller Warteschlange in die Aktionen eingereiht werden, die auf den Maschinen ausgeführt werden sollen.
Es gibt drei verschiedene Aktionen, die auf einer Maschine ausgelöst werden können.

1. Starten des Programms *stress*[Wat17] mit spezifizierten Parametern
2. Ausführen von Linux Befehlen auf Maschinen (über SSH)
3. Fluten einer Maschine mit REST-Befehlen auf einem bestimmten Port

Das Programm wandelt die eingelesene JSON-Datei in Einträge um, die in der Warteschlange verarbeitet werden können. Jeder der Einträge besitzt das Feld *machine*, *start* und *duration*, die die Maschine festlegen, auf dem die Aktion ausgeführt wird, wann (nach Programmstart) diese Aktion durchgeführt wird und wie lange sie maximal dauert. Daneben gibt es ein Feld, dass die drei Einträge unterscheidbar macht. Die drei Arten von Einträgen werden im folgenden kurz erläutert.

Stress Eintrag

Das Programm *stress* bietet eine ganze Reihe von Möglichkeiten um Last auf CPU, RAM und Festplatte zu erzeugen. Gesteuert wird es dazu über Parameter, die bei dem Programmstart angegeben werden. Das Programm *ecp_stress* ist dazu in der Lage diese Parameter aus einer JSON-Datei einzulesen und per SSH-Protokoll das Programm *stress* mit diesen Parametern auf einer Maschine zu starten. Eine Eintrag wird wie in Anhang: Beschreibung 2 angegeben definiert.

Linux Befehl über SSH

Eine einfache Möglichkeit Aktionen auf einer Cluster Maschine auszulösen, ist es ihr Befehle über das SSH-Protokoll zu senden, die auf ihr ausgeführt werden sollen. Dazu bietet *ecp_stress* die Möglichkeit Kommandos auf Maschinen auszuführen. In Anhang: Beschreibung 3 ist die Konfiguration für den *Shutdown*-Befehl auf einer Maschine angegeben, der dazu benutzt wurde einen Serverausfall zu simulieren.

Parameter	**Wert**	**Beschreibung**
[General]		Generelle Einstellungsmöglichkeiten
verbose	*true* oder *false*	Ausführliche Ausgabe während des Programmablaufs
[Monitor]		Einstellungen des Monitormoduls
ip	String	IP-Adresse der externen Monitoringsoftware in IPv4 Notation
ignored-roles	String	Rollen die bei der Überwachung nicht beachtet werden sollen, getrennt durch Kommata
interval	Ganze Zahl	Dauer der Pause zwischen zwei Monitor Zyklen. Die Eingabe ist eine Zahl gefolgt von dem Buchstaben 's' für Sekunden oder 'm' für Minuten.
[Analyser]		Einstellungen des Analysemoduls
greedy	*true* oder *false*	Wenn *greedy* aktiviert ist, wird nach dem Starten einer Maschine nicht gewartet bis diese initialisiert ist, bevor mit der Skalierung fortgesetzt wird.
min-machines	Ganze Zahl	Minimale Anzahl von Maschinen für jede Rolle (Ausnahme bilden Maschinen der Rolle *master*)
max-machines	Ganze Zahl	Maximale Anzahl von Maschinen für jede Rolle (Ausnahme bilden Maschinen der Rolle *master*)
machine-timeout	Ganze Zahl	Dauer wie lange auf das Senden von Daten einer neu erstellten Maschine gewartet wird, wird diese überschritten, wird der Neustart angeordnet. Die Eingabe ist eine Zahl gefolgt von dem Buchstaben 's' für Sekunden oder 'm' für Minuten.
[Planner]		Einstellungen des Planungsmoduls
interval	Ganze Zahl	Dauer der Pause zwischen zwei Planungszyklen. Die Eingabe ist eine Zahl gefolgt von dem Buchstaben 's' für Sekunden oder 'm' für Minuten.
[SSH]		Einstellungen für SSH-Verbindung
timeout	Ganze Zahl	Dauer bis ein per SSH gesendeter Befehl abgebrochen wird. Die Eingabe ist eine Zahl gefolgt von dem Buchstaben 's' für Sekunden oder 'm' für Minuten.
ip-field	*private* oder *public*	Gibt an welche IP-Adressen Feld von Cluster Maschinen für die Verbindung genutzt werden soll.
key-pair	String	Speicherort des privaten Schlüssels, der zur Authentifizierung auf Maschinen über SSH genutzt wird.
[Agent]		Einstellungen für Agenten
time-frame	Ganze Zahl	Zeitfenster in dem Werte von Agenten berücksichtigt werden. Die Eingabe ist eine Zahl gefolgt von dem Buchstaben 's' für Sekunden oder 'm' für Minuten.

Tabelle 1: Konfigurationsparameter des Programms *ecp_autoscale*

```
{
  "machine": "544dbf1fc0",
  "start": 30,
  "duration": 300,
  "stress_parameter":{
    "cpu":4,
    "io":1,
    "vm": 3,
    "vm-bytes": "512m",
    "hdd":1
  }
}
```

Beschreibung 2: Beispiel für *stress*-Konfiguration

```
{
  "machine": "211472028e",
  "start": 20,
  "duration": 0,
  "command": "sudo shutdown -h now"
}
```

Beschreibung 3: Beispiel für *command*-Konfiguration

HTTP-Flut

Diese Reihe an Befehlen wird dazu genutzt eine Last auf dem Netzwerkadapter einer Maschine des Clusters zu erzeugen. Dazu wird auf der Maschine ein Service benötigt, der einen REST-Server auf einem Port zur Verfügung stellt. Im Fall von Kubernetes kann dazu die *keys*-Tabelle des etcd2-Service benutzt werden. Eine HTTP-Flut wird wie in Anhang: Beschreibung 4 angegeben beschrieben.

```
{
  "machine": "f55ed606e5",
  "start": 2,
  "duration": 10,
  "http_flood":{
    "port":2379,
    "path":"/v2/keys/foobar",
    "retries": 420,
    "delay": 0.1
  }
}
```

Beschreibung 4: Beispiel für *httpflood*-Konfiguration

Wie in Anhang: Beschreibung 5 zu erkennen ist, wird zunächst die Verbindung zu dem REST-Server hergestellt (Zeile 2-4).

Ist die Verbindung erfolgreich aufgebaut, wird so oft ein Eintrag für den REST-Server erzeugt (Zeile 8-13) und abgesendet (Zeile 14), wie in der Konfiguration angegeben (Zeile 6). Nach einer kurzen Wartezeit (Zeile 15), wird mit dem nächsten Senden begonnen. Sollte dieser Vorgang insgesamt länger dauern als eine vorher definierte Zeitspanne, wird der Prozess abgebrochen (Zeile 5).

Im letzten Schritt wird noch die erzeugten Daten von dem REST-Server gelöscht (Zeile 21), damit keine Rückstände des Tests zurück bleiben.

```
1  def run_http_flood(entry)
2    ip = @cluster.machines.select{|m| m.hostname==entry.machine}.first.
         private_ipv4
3    uri = URI.parse("http://#{ip}:#{entry.port}")
4    http = Net::HTTP.new(uri.host,uri.port)
5    Timeout::timeout(entry.duration) do
6      entry.retries.times do
7        header = {:'Content-Type' => 'application/json'}
8        data = '{'
9        4096.times do |i|
10         data+= "\"#{i}\":\"dummy-data\","
11       end
12       data = data.chop
13       data += '}'
14       http.put("#{entry.path}",data,header)
15       sleep(entry.delay) unless entry.delay==0
16     end
17   end
18 rescue Timeout::Error => e
19   Utils::LOG.info(entry){"Flood closed after #{entry.duration}s"}
20 ensure
21   http.delete("#{entry.path}")
22 end
```

Beschreibung 5: Erzeugen einer Flut von REST-Befehlen

1.3 Das Command Line Interface

Das Programm ecp_autoscale wird über ein CLI in Betrieb genommen. Für die Konfiguration der Parameter und Optionen wurde das Gem[1] „Commander“[TH16] verwendet.

[1]Eine über die Webseite www.rubygems.org verbreitete Softwarebibliothek [Rub17]

Globale Optionen

Globale Optionen sind Parameter und Einstellungen, die für jeden Befehl des Programms gelten.

- „–config-file FILE“: Lädt die Datei FILE als Konfigurationsdatei für das Programm ein.
- „–verbose“: Startet das Programm im Debug-Modus. Erlaubt zusätzliche Ausgaben und Log-Einträge

Befehl: Start

Mit dem Start Befehl wird das Programm im normalen Zustand gestartet.

Optionen

Für den Befehl Start stehen folgende Optionen zur Verfügung, die über Parameter eingestellt werden können.

- „–cluster-file FILE“: Liest die Datei FILE als ursprüngliche Cluster Konfiguration ein. Diese Cluster Konfiguration ist ein Cluster-Objekt des ecp_deploy Programms.
- „–credential-file FILE“: Die Datei FILE enthält die Login Daten, die für die Verbindung zu den VMs des Clusters benötigt werden. Hier kann die Credential-Datei des ecp_deploy verwendet werden.
- „–scheduler SCHEDULER“: Verwendet den unter SCHEDULER angegebenen Scheduling-Algorithmus im Planungsmodul.

Beispiel

Mit dem Befehl **ecp_autoscale start –config-file scale.conf –credential-file creds.json –cluster-file cluster.yaml –scheduler round-robin –verbose** lässt sich das Programm starten. Dabei wird die Konfigurationsdatei **scale.conf** eingelesen, die Cluster Datei **cluster.yaml** verwendet, das Planungsmodul nutzt Round-Robin Scheduling und es findet eine ausführliche Ausgabe von Informationen statt.

2 Anhänge zur Verifikation

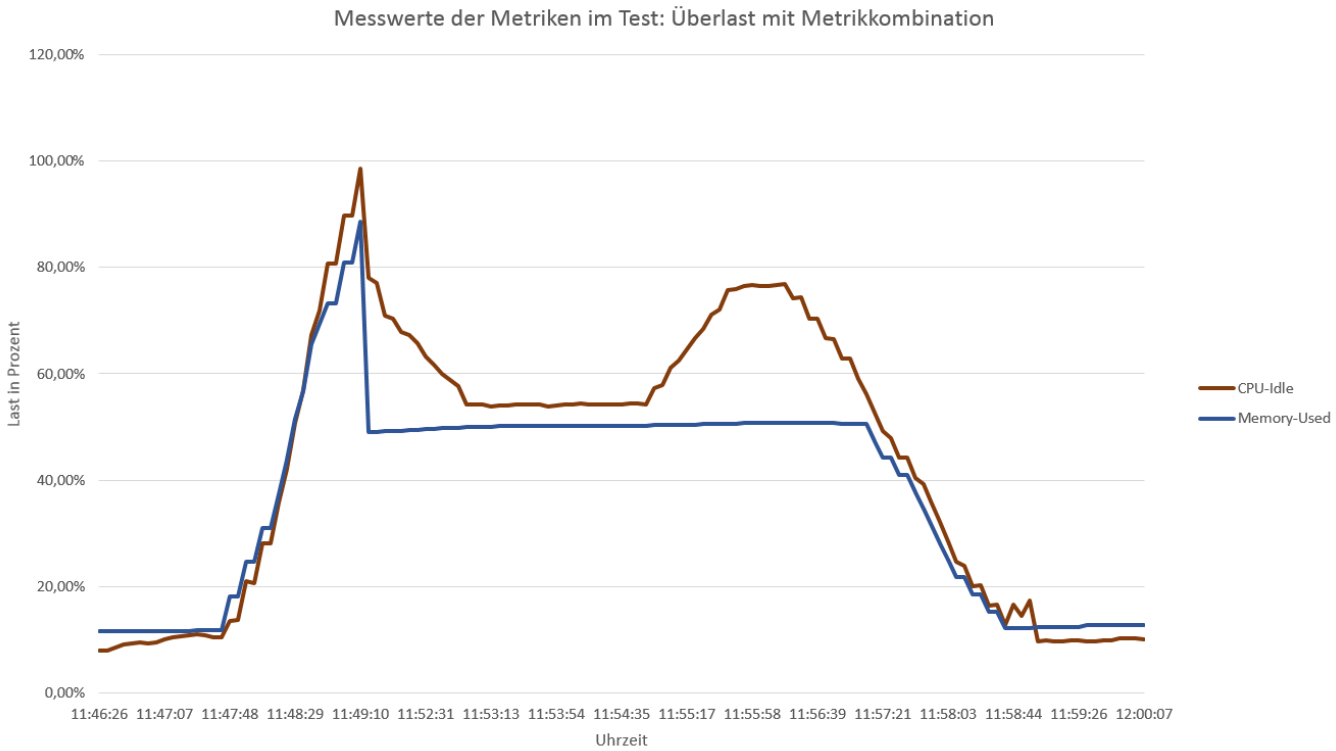

Abbildung 1: Gesamtverlauf der Metrikdaten in den Tests *Überlast mit Metrikkombination* und *Unterlast*

```
1  [..]
2  11:14:36 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.10866666666666658
3  11:14:36 DEBUG -- InverseNumericMetric: CPU-Idle: Role: lowPerformer cpu-idle
       average < 0.15000000000000002. Advice: Shutdown -1 server(s)
4  11:14:36 DEBUG -- #<Analyser>: Scaling with factor -1.0
5  11:14:36 DEBUG -- #<Analyser>: Modify cluster. Scale value for lowPerformer is
       -1.0
6  11:14:36 DEBUG -- #<Analyser>: Stopping shutdown due to cluster limits
7  [..]
8  11:16:17 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.9796666666666667
9  11:16:17 DEBUG -- InverseNumericMetric: CPU-Idle: Role: lowPerformer cpu-idle
       average > 0.7. Advice: Start 2 server(s)
10 11:16:17 DEBUG -- #<Analyser>: Scaling with factor 2.0
11 11:16:17 DEBUG -- #<Analyser>: Modify cluster. Scale value for lowPerformer is
       2.0
12 11:16:17 DEBUG -- #<Planner>: Enqueued action: #<StartAction:0x000000024999e8>
13 [..]
14 11:19:36 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.5634
15 11:19:36 DEBUG -- #<Analyser>: Cluster is working within InverseNumericMetric:
       CPU-Idle limits
16 11:19:36 DEBUG -- #<Analyser>: Scaling with factor 0.0
17 [..]
```

Beschreibung 6: Verhalten von *ecp_autoscale* im Test: **Überlast mit Einzelmetrik**

```
1  [..]
2  11:46:26 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.08046666666666663
3  11:46:26 DEBUG -- InverseNumericMetric: CPU-Idle: Role: lowPerformer cpu-idle
       average < 0.15000000000000002. Advice: Shutdown -1 server(s)
4  11:46:26 DEBUG -- NumericMetric: Memory-Use with total: memory-total: Current
       numeric metric Memory-Use is: 0.11710574510201219
5  11:46:26 DEBUG -- #<Analyser>: Cluster is working within NumericMetric: Memory-
       Use with total: memory-total limits
6  11:46:26  INFO -- #<Analyser>: Modify cluster. Scale value for lowPerformer is
       -0.5
7  11:46:26  INFO -- #<Analyser>: Stopping shutdown due to cluster limits
8  [..]
9  11:47:59 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.21050000000000002
10 11:47:59 DEBUG -- #<Analyser>: Cluster is working within InverseNumericMetric:
       CPU-Idle limits
11 11:47:59 DEBUG -- NumericMetric: Memory-Use with total: memory-total: Current
       numeric metric Memory-Use is: 0.24654504174524747
12 11:47:59 DEBUG -- #<Analyser>: Cluster is working within NumericMetric: Memory-
       Use with total: memory-total limits
13 11:47:59  INFO -- #<Analyser>: Role lowPerformer working within all limits.
       Scale value is 0.0
14 [..]
15 11:48:45 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.7191666666666667
16 11:48:45 DEBUG -- InverseNumericMetric: CPU-Idle: Role: lowPerformer cpu-idle
       average > 0.7. Advice: Start 1 server(s)
17 11:48:45 DEBUG -- NumericMetric: Memory-Use with total: memory-total: Current
       numeric metric Memory-Use is: 0.694565779287915
18 11:48:45 DEBUG -- NumericMetric: Memory-Use with total: memory-total: Role:
       lowPerformer memory-used average > 0.5. Advice: Start 1 server(s)
19 11:48:45 DEBUG -- #<Analyser>: Scaling with factor 1.0
20 [..]
21 11:57:16 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.527875
22 11:57:16 DEBUG -- #<Analyser>: Cluster is working within InverseNumericMetric:
       CPU-Idle limits
23 11:57:16 DEBUG -- NumericMetric: Memory-Use with total: memory-total: Current
       numeric metric Memory-Use is: 0.4742286740753441
24 11:57:16 DEBUG -- #<Analyser>: Cluster is working within NumericMetric: Memory-
       Use with total: memory-total limits
25 11:57:16  INFO -- #<Analyser>: Role lowPerformer working within all limits.
       Scale value is 0.0
26 [..]
```

Beschreibung 7: Verhalten von *ecp_autoscale* im Test: **Überlast mit Metrikkombination**

```
1  [..]
2  11:57:42 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.40413333333333334
3  11:57:42 DEBUG -- #<Analyser>: Cluster is working within InverseNumericMetric:
       CPU-Idle limits
4  11:57:42 DEBUG -- NumericMetric: Memory-Use with total: memory-total: Current
       numeric metric Memory-Use is: 0.37811007938610847
5  11:57:42 DEBUG -- #<Analyser>: Cluster is working within NumericMetric: Memory-
       Use with total: memory-total limits
6  11:57:42 DEBUG -- #<Analyser>: Scaling with factor 0.0
7  [..]
8  11:59:00 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.0973888888888889
9  11:59:00 DEBUG -- InverseNumericMetric: CPU-Idle: Role: lowPerformer cpu-idle
       average < 0.15000000000000002. Advice: Shutdown -1 server(s)
10 11:59:00 DEBUG -- NumericMetric: Memory-Use with total: memory-total: Current
       numeric metric Memory-Use is: 0.12405183476467319
11 11:59:00 DEBUG -- #<Analyser>: Cluster is working within NumericMetric: Memory-
       Use with total: memory-total limits
12 11:59:00 DEBUG -- #<Analyser>: Scaling with factor -0.5
13 11:59:00  INFO -- #<Analyser>: Modify cluster. Scale value for lowPerformer is
       -0.5
14 11:59:00 DEBUG -- #<Planner>: Enqueued action: #<KillAction>
15 [..]
16 11:59:31 DEBUG -- InverseNumericMetric: CPU-Idle: Current numeric metric CPU-
       Idle is: 0.09783333333333338
17 11:59:31 DEBUG -- InverseNumericMetric: CPU-Idle: Role: lowPerformer cpu-idle
       average < 0.15000000000000002. Advice: Shutdown -1 server(s)
18 11:59:31 DEBUG -- NumericMetric: Memory-Use with total: memory-total: Current
       numeric metric Memory-Use is: 0.1272091925268993
19 11:59:31 DEBUG -- #<Analyser>: Cluster is working within NumericMetric: Memory-
       Use with total: memory-total limits
20 11:59:31 DEBUG -- #<Analyser>: Scaling with factor -0.5
21 11:59:31  INFO -- #<Analyser>: Modify cluster. Scale value for lowPerformer is
       -0.5
22 11:59:31  INFO -- #<Analyser>: Stopping shutdown due to cluster limits
```

Beschreibung 8: Verhalten von *ecp_autoscale* im Test: **Unterlast**

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AWS Amazon Web Services

CLI Command Line Interface

FIFO First In - First Out

GCE Google Compute Engine

JSON JavaScript Object Notation

OCI Open Container Initiative

SSH Secure Shell

UC University of California

VM virtuelle Maschine

Query DSL Query Domain Specific Language

REST Representational State Transfer

Literaturverzeichnis

[Ama15] Amazon Web Services: *Netflix Delivers Billions of Hours of Content Globally by Running on AWS*, Februar 2015. https://www.youtube.com/watch?v=lQGHsBOZJBw, Zugriff 09.06.2017.

[Ama17] Amazon Web Services: *Amazon EC2 – Preise*, Juli 2017. https://aws.amazon.com/de/ec2/pricing/, Zugriff 05.07.2017.

[Con17] Contributors, Heapster: *Heapster*, September 2017. https://github.com/kubernetes/heapster, Zugriff 11.09.2017.

[Cor17] CoreOS: *RKT - A security-minded, standards-based container engine*, Juli 2017. https://coreos.com/rkt, Zugriff 02.07.2017.

[Doc17a] Docker: *Docker Company*, März 2017. https://www.docker.com/company, Zugriff 09.06.2017.

[Doc17b] Docker Inc: *Swarm mode key concepts*, Juli 2017. https://docs.docker.com/engine/swarm/key-concepts/, Zugriff 02.07.2017.

[Ela17] Elasticsearch: *Elasticsearch Reference - Query DSL*, Januar 2017. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html, Zugriff 16.08.2017.

[Has17a] HashiCorp: *Introduction to Nomad*, Juli 2017. https://www.nomadproject.io/intro/index.html, Zugriff 02.07.2017.

[Has17b] HashiCorp: *Nomad*, Juli 2017. https://www.hashicorp.com/blog/nomad-announcement/, Zugriff 02.07.2017.

[KC03] Kephart, J. O. und D. M. Chess: *The vision of autonomic computing*. Computer, 36(1):41–50, Jan 2003.

[Kra16] Kratzke, Prof. Dr. Nane: *Container- und Cluster-Technologien für Microservices*, November 2016. https://cosa.fh-luebeck.de/files/cloud_transit/poster/poster-container-cluster.pdf, Zugriff 09.08.2017.

[PDNK14] Prof. Dr. Nane Kratzke, Peter-Christian Quint: *Cloud TRANSIT*, Januar 2014. https://cosa.fh-luebeck.de/de/cloud/projekte/cloud-transit, Zugriff 13.07.2017.

[Rub17] Ruby Gems Contributors: *What is a gem?*, Juni 2017. http://guides.rubygems.org/what-is-a-gem/, Zugriff 19.07.2017.

[TH16] TJ Holowaychuk, Gabriel Gilder: *commander*, Dezember 2016. https://rubygems.org/gems/commander, Zugriff 19.07.2017.

[The17a] The Apache Software Foundation: *The Apache Software Foundation Announces Apache Mesos v1.0*, Juli 2017. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces97, Zugriff 02.07.2017.

[The17b] The Apache Software Foundation: *Mesos Architecture*, Juli 2017. http://mesos.apache.org/documentation/latest/architecture/, Zugriff 02.07.2017.

[The17c] The Kubernetes Authors: *What is Kubernetes?*, Juli 2017. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, Zugriff 02.07.2017.

[Wat17] Waterland, Amos: *The stress workload generator*, Juli 2017. http://people.seas.harvard.edu/~apw/stress/, Zugriff 09.07.2017.