HAW HAMBURG

Grundseminararbeit

Tobias Schweisfurth

Infrastructure as Code: Cluster as a Service und Betrachtung von automatisierter Aufsetzung von Mandantenfähigkeit

Betreuung durch: Prof. Dr. Kai von Luck / Prof. Dr. Jan Sudeikat
Eingereicht am: 28. Februar 2019

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Inhaltsverzeichnis

1 Vorwort

Diese schriftliche Ausarbeitung ist als Prüfungsleistung des Master-Studiengangs Informatik an der HAW im Grundseminar entstanden. Sie dient der Einarbeitung in das Themengebiet Infrastructure as Code und Ermöglichung von automatisierter Mandantenfähigkeit in Form von Cluster as a Service.

2 Einleitung

Infrastructure as Code ist die Art und Weise wie einem Projekt zugrunde liegende Infrastruktur und dessen Konfiguration durch Code beschrieben werden kann. Der Grundgedanke dahinter ist, dass für jedes Produkt die Infrastruktur einfach funktionieren muss. Beachtet man aber das anwachsende Portfolio von Möglichkeiten neue Infrastruktur zu provisionieren, so kann es für das Unternehmen oder den Administrator immer schwieriger werden, auf einem aktuellen Stand zu bleiben und dabei trotzdem sicher zu stellen, dass die Infrastruktur nicht in sich zusammen bricht.

In der DevOps-Bewegung wird ein organisatorischer Umschwung beschrieben, wobei Entwicklung, Qualitätssicherung und Operations besser [6]. Dabei sollen anstelle von verteilten Silos crossfunktionale Teams daran arbeiten, dass kontinuierlich Features ausgeliefert werden. Das bedeutet, dass Entwicklung und Operations stärker miteinander zusammen arbeiten müssen, fördert aber auch die Problemlösung, da es in solchen DevOps orientierten Organisationen weniger zu Missverständnissen zwischen den unterschiedlichen Teams kommt[8].

Wenn in DevOps orientierten Teams Operations und Entwicklung enger miteinander zusammen arbeiten, erscheint es logisch, dass Praktiken aus dem jeweiligen Bereich versucht werden auf den anderen anzuwenden. Infrastructure as Code beschreibt dabei den Ansatz die Automatisierung von Infrastruktur basierend auf Praktiken der Softwareentwicklung zu ermöglichen bzw. zu erleichtern.

In dieser Seminararbeit soll sich mit dem Grundgedanken von Infrastructure as Code auseinander gesetzt werden. Dafür soll im Abschnitt 3 darauf eingegangen werden, was Infrastructure as Code ist und welche Praktiken und Tools momentan zur Verfügung stehen. Weiterhin soll auf das Thema Continuous Configuration Automation (CCA) eingangen werden, welches als Erweiterung oder Implementierung von Infrastructure as Code angesehen werden kann. Die Umsetzung von Infrastructure as Code soll im Abschnitt 4 anhand von dem Thema "Multi-tenancy in der Cloud“ besprochen werden. Dabei soll in Hinsicht auf an diese Arbeit anschließende Projekte angesprochen werden, inwieweit sich Cluster-Infrastruktur automatisieren lässt und Mandantenfähigkeit in solchen Clustern automatisiert umgesetzt werden kann. Abschließend soll mit einem Fazit auf die weitere Forschung eingegangen werden.

3 Infrastructure as Code

Infrastructure as Code ist die Umsetzung und Provisionierung von Infrastruktur durch Code. Das bedeutet auf einer abstrakten Ebene, dass Konfigurationen für Infrastruktur genauso wie Software programmiert werden. Ziel ist es dabei, dass etablierte Mechanismen aus der Softwareentwicklung für die Entwicklung und Umsetzung von Infrastruktur genutzt werden können. Speziell wird dabei im Abschnitt 3.1 auf Praktiken agiler Softwareentwicklung eingegangen. Traditionelle Prozesse, Strukturen und Überwachung, die verwendet wurden, als sich noch nicht mit Cloud und Automatisierung auseinander gesetzt wurde, tendieren dazu, dass sie nicht mehr greifen oder zu langsam sind. Agile Entwicklung basiert darauf auf sich ändernde Anforderungen reagieren zu können[9]. Daher findet man in der Entwicklung von Infrastructure as Code häufig Verweise auf agile Softwareentwicklungsmethoden. Beispielsweise kann durch Transparenz als agiler Wert Angst vor Veränderung der Infrastruktur genommen werden. Alle Anpassungen, die gemacht wurden und gemacht werden sollen, sind im Code beschrieben und einsehbar.

Infrastucture as Code sorgt für die Umsetzung einer Infrastruktur und die Anpassungsfähigkeit an eine sich immer weiterentwickelnde Cloud- und Automatisierungslandschaft[9]. Das bedeutet, dass für das Produkt benötigte Infrastruktur durch diesen Code konfiguriert wird. Ein solches Code-Snippet kann zum Beispiel beinhalten, wie die zugrunde liegende Cloud konfiguriert ist oder wie viele virtuelle Maschinen benötigt werden[1]. Im Code wird beschrieben, welche und wie viele Geräte vorhanden sein sollen, welche Software und Middleware auf diesen Geräten installiert sein soll, damit die eigentliche Software darauf ausgeführt werden kann und startet ggf. Prozesse, die neben der eigentlichen Software laufen sollen[1].

Durch die Umsetzung von Infrastruktur durch Source-Code können typische Softwareentwicklungsprozesse und Praktiken greifen. Dazu gehören unter anderem Versionskontrolle, (automatisiertes) Testing von Code (sowohl Unit- als auch Integrationstest) und Praktiken wie test-getriebene Softwareentwicklung (engl. test-driven development) und continiuous integration (CI) und continiuous delivery (CD)[9]. Infrastructure as Code ermöglicht weiterhin eine Art von Debugging. Ist eine automatisierte Umsetzung von Infrastruktur erst einmal in Code geschrieben, so lässt es sich auf verschiedenen Maschinen ausführen und testen[1]. Weiterhin kann der Code ähnlich wie in der Softwareentwicklung auf Schwächen oder (Syntax-)Fehler untersucht werden. Dazu gehören unter anderem Bemühungen Code-Smells[12], Anti-Patterns[10] und Charakteristken für defekten Skripten[11] herauszufinden. Versionskontrolle ermöglicht außerdem, dass auf vorherige

Versionen - zum Beispiel im Falle einer fehlerhaften Konfiguration - zurückgekehrt werden kann.

Die Ziele von Infrastructure as Code lassen sich grob darunter zusammenfassen, dass der DevOps-Gedanke übernommen werden soll und Infrastruktur wie Software-Code behandelt wird und sich Infrastruktur zusätzlich zum Produkt weiter entwickelt. Keif Morris zählt in "Infrastructure as Code"[9] als weitere Ziele auf, dass Veränderung von Infrastruktur ermöglicht und unterstützt wird, Veränderungen als Routine aufgefasst werden können, damit die Angst vor Infrastruktur-Veränderungen genommen werden kann, Infrastruktur von den Entwicklern definiert, provisioniert und gemanaged werden kann, ohne dass ein previligiertes Team dafür benötigt wird, und dass Lösungen zu Problemen bei Aufsetzung von Infrastruktur durch Implementierung, Testing und Evaluation überprüft werden können.

3.1 Praktiken von Infrastructure as Code

3.1.1 Definitionsdatei

Der Grundbaustein von Infrastructure as Code ist eine Definitionsdatei (engl. definition file). Sie beschreibt was und wie etwas (eine Ressource) konfiguriert werden soll. Üblicherweise handelt es sich dabei um text-basierte Dateien, die entweder in Form von Standardformaten wie YAML, XML oder JSON oder in Form eine domain-spezifischen Sprache (engl. domain-specific language (DSL)) gespeichert werden[9, 8]. Text-basierte Dateien haben den Vorteil, dass sie einfach in Versionskontrollsystemen versioniert werden können, anders als Konfigurationen, die in der Tool-eigenen Datenbank abgelegt sind.

Verschiedene Tools verwendenen Definitionsdateien unterschiedlich. Mit Puppet[1] beschreibt man einen gewünschten Zustand der Infrastruktur. Dieser wird von einem Master Server mit Agenten auf jedem Client erzielt. Dabei wird eine DSL ähnlich zu JSON verwendet, um Zustände in Definitionsdateien zu definieren. Im Gegensatz dazu werden bei Ansible Playbooks in YAML für die Konfiguration geschrieben, welche von Ansible mithilfe von SSH auf die jeweiligen Clients übertragen wird. Als dritte von vielen Tools werden Definitionsdateien bei Chef[2] in einer Ruby ähnlichen DSL geschrieben[8]. Dabei

[1] puppet.io
[2] chef.io

können bei Chef viele Ressourcen in Klassen und viele Klassen in Modulen zusammengefasst werden[1].

3.1.2 Versionskontrolle

In der Softwareentwicklung ist es üblich, dass Versionskontrollsysteme (VCS) eingesetzt werden, um Veränderungen über die Zeit hinweg zu protokollieren und Kollaboration zwischen Entwicklern zu ermöglichen. Dabei steht besonders im Fokus, dass zu jederzeit auf vergangene Versionen und Änderungen zugegriffen werden kann[4].

Mithilfe von Definitionsdateien ist es bei Infrastructure as Code möglich, dass Änderungen der Infrastruktur in einem VCS festgehalten werden. Das bedeutet, dass Aussagen über den Zustand der Infrastruktur immer anhand der Versionen der Definitionsdateien im VCS getroffen werden. Das impliziert auch, dass alle Infrastrukturänderungen aus dem VCS heraus geschehen müssen, was bedeutet, dass es keine einmaligen Änderungen oder Fixes auf Ressourcen in der Infrastruktur geben kann, die nicht ins VCS eingecheckt wurden.

Ein großer Vorteil davon ist Transparenz. Jeder Entwickler ist in der Lage den genauen Zustand einer Ressource anhand der Version im VCS auslesen zu können. Er muss sich nicht mehr mit der Ressource verbinden, um diese Informationen zu erhalten. Gleichzeitig kann man anhand der Historie in VCS feststellen, welche Änderungen gemacht wurden und von wem. Bezieht man eine Commit[3]-Nachricht mit ein, so hat man viele Informationen, wenn es zu Debugging kommt. Genauso kann in VCS auf eine vorherige Version zurückgekehrt werden. Sollte demnach eine Konfiguration der Infrastruktur nicht funktionieren, kehrt man auf eine funktionierende Version zurück. Wenn man zusätzlich hinzuzählt, wie sehr VCS die Prozesse bzw. den Workflow eines Teams prägen, so kann man hinzuzählen, dass Code-Reviews verstärkt aufgenommen werden. Bevor eine Änderung in Produktion geht - also von einer Verzweigung im VCS auf den Hauptzweig gepusht wird - wird sichergestellt, dass ein weiterer Entwickler diesen Änderungen zustimmt und keine Fehler auffindet.

Zuletzt kann man noch hinzufügen, dass über moderne CI/CD-Tools wie Jenkins[4] in Verbindung mit GIT weitere Automatisierung geschaffen werden kann. Liegt ein neuer Com-

[3] Commit in GIT-Versionskontrolle. Andere Systeme können unterschiedliche Formen der Festhaltung einer Version in der Historie verwenden.

[4] https://jenkins.io/ - Neben Jenkins gibt es noch weitere CI/CD-Build-Tools wie Travis CI und GitLab

mit vor, so kann das CI/CD-Tool direkt mit der Ausführung der CI- und CD-Pipelines zu beginnen.

3.1.3 Automatisiertes Testen von Änderungen

Wenn Infrastruktur wie Code behandelt wird, spielt das Testen von Definitionsdateien eine große Rolle. Durch Tests wird sichergestellt, dass Änderungen den gewünschten Effekt und keine Seiteneffekte haben. Durch automatisiertes Testen kann auch sichergestellt werden, dass zerstörerische Veränderungen nicht in Produktion gelangen. Im einfachsten Fall kann sichergestellt werden, dass es keine Syntax-Fehler gibt. Im komplexeren Szenario kann über Integrationstests das Verhalten der Veränderung überprüft werden. Im Idealfall gibt es Testumgebungen, wo Veränderungen eingespielt werden, bevor sie in Produktion übergehen.

3.2 Continuous Configuration Automation

Continiuous Configuration Automation (CCA) ermöglicht die Automatisierung von Konfigurationseinstellungen sowohl on-premise als auch in Cloud-Umgebungen. Es kann als Implementierung der Infrastructure as Code Prinzipien angesehen werden und wird in vielen Arbeiten synonym verwendet[12]. CCA-Tools haben die Aufgabe die Konfiguration und Provisionierung der Infrastruktur über Tasks im CCA-Tools durchzuführen. In deklarativer oder imperativer[5] Schreibweise werden Definitionsdateien (s. 3.1.1) in einem für das Projekt angelegten VCS abgelegt und vom CCA-Tools abgearbeitet.

Änderungen erfolgen dabei ausschließlich nur noch über das CCA-Tool. Alle temporären Lösungen oder Änderungen der Infrastruktur, die als Hotfix oder Test erstellt worden sein können, werden bei der nächsten Synchronisation mit dem CCA-Tool verworfen[9]. Damit werden die Konsistenz und Transparenz als Richtlinien von Infrastructure as Code sichergestellt. Durch CCA-Tools können gleichzeitig eine Vielzahl an Ressourcen konfiguriert werden.

Zu den bekannten CCA-Tools gehören Chef, Puppet, Ansible und SaltStack. Auf diese soll im Folgenden genauer eingegangen werden.

[5] Methoden der Interaktion mit den anzupassenden Systemen variert zwischen den CCA-Tools

3.2.1 Puppet

Puppet ist ein CCA-Tool, welches nach der agent-master Architektur vorgeht. Auf einem oder mehreren Servern wird die Puppet master Applikation installiert und auf allen zu konfigurierenden Clients wird die Puppet agent Applikation installiert, welche im Hintergrund laufen kann oder manuell getriggert wird.

Konfigurationen werden dabei in Form von Puppet-Manifesten programmiert. Solche Manifeste beinhalten Informationen über Ressourcen und Abhängigkeiten dieser Ressourcen. Der Puppet agent sendet in regelmäßigen Abständen oder manuell "facts" an den Puppet master. Dafür verwendet Puppet den sogenannten Facter[6]. Der Puppet master hält die Manifeste für die jeweiligen Clients vor, kompiliert diese zu einem Catalog, baut Graphen der Ressourcen und ihrer Abhängigkeiten auf und beantwortet schlussendlich die Anfrage mit dem für den Client passenden Catalog. Der Puppet agent kann daraufhin überprüfen, welche Anforderungen und Konfigurationen durch den Catalog beschrieben werden und führt diese aus, sofern man sich noch nicht im gewünschten Zustand befindet. Am Ende wird dem Puppet master mitgeteilt, welche Änderungen vorgenommen wurden und ob Fehler aufgetreten[7].

Puppet verwendet für die Programmierung der Puppet-Manifeste eine domänenspezifische Sprache, die auf Ruby basiert[14].

3.2.2 Chef

Ähnlich wie Puppet agiert Chef auch nach dem agent-master Prinzip. Dabei werden auf den Master Knoten oder Server sogenannte Rezepte (engl. recipes) hochgeladen. Die Rezepte werden dann auf einzelnen Clients deployt. Dafür verwendet Chef zusätzlich eine Workstation[14], welche sich mit den Clients über SSH verbindet und über Zertifikate authentifiziert[2].

Konfigurationen werden in Chef in Ruby programmiert. Auch bietet Chef die Möglichkeit Kochbücher (engl. cookbook) zu erstellen und zu importieren, welche komplette Systeme beschreiben. Vorteil ist, dass Konfigurationen, die bereits von anderen Unternehmen vorgenommen und getestet wurden, übernommen werden können[2].

[6] https://puppet.com/docs/facter

3.2.3 Ansible

Im Gegensatz zu Chef und Puppet verfolgt Ansible einen agentenlosen Ansatz. Daher muss auf den einzelnen Clients, die konfiguriert werden sollen, keine zusätzliche Software installiert werden. Alle Aktionen von Ansible werden stattdessen über SSH ausgeführt.

Dafür muss dennoch ein Ansible Master aufgesetzt werden, welchem alle zu konfigurierenden Clients in einer Inventar-Liste eingepflegt werden. Auf jedem Client wird ein authorisierte Schlüssel für SSH hinterlegt[14]. Die Kommunikation über SSH hat sowohl den Vorteil, dass keine weiteren Ports außer für SSH geöffnet werden müssen[2], als auch den Nachteil, dass nicht zwangsläufig alle Systeme SSH unterstützen. Für diesen Fall kann Ansible Sudo-Credentials hinterlegen, um Befehle als root auf diesen Systemen auszuführen[14]. Wenn ein Client konfiguriert werden soll, wird er aus der Inventar-Liste ausgewählt und vorher definierte Anweisungen über SSH ausgeführt.

Anweisungen werden in Form von YAML programmiert und als sogenannte Playbooks festgehalten. Einzelne Module für die Konfiguration ganzer Systeme können in jeder Programmiersprache geschrieben werden, solange man valides JSON als Ausgabe erhält [14, 13].

Ansible stellt in Form von Ansible Tower ein User-Interface, welches über ein Dashboard den aktuellen Zustand darstellt und über REST API-calls Ausführungen von Playbooks erlaubt.

3.2.4 SaltStack

Bei SaltStack gibt es auch wieder Master, wobei die Besonderheit bei SaltStack darin liegt, dass die einzelnen Clients (hier Minions) nicht explizit im Master hinterlegt werden müssen, sondern sich über eine Anfrage beim Master anmelden. Akzeptiert der Master, so kann der Client von diesem Master kontrolliert werden[2]. Dafür können Konfigurationen wie bei Ansible auch direkt über die CLI eingepflegt werden oder Definitionsdateien (in SaltStack: ”states“) in YAML geschrieben werden, welche SaltStack abarbeiten kann.

SaltStack bietet die Möglichkeit unterschiedliche Level von Master zu haben und ist in der Lage Lasten und Redundanzen ausreichend zu verteilen[2, 14]. Zusammen mnit der Fähigkeit von asynchronen File-Server kann somit eine hohe Skalierbarkeit und Resilienz ermöglicht werden[14].

4 Automatisierte Aufsetzung von Mandantenfähigkeit

Unter Mandantenfähigkeit (engl. Multi-tenancy) versteht man allgemein, dass alle Nutzer auf der gleichen Plattform ungestört voneinander arbeiten. Es wird dabei nicht jedem Kunden eine eigene Infrastruktur bereitgestellt (Single-Tenant-Architektur), sondern eine Software bedient verschiedene Nutzer auf Software as a Service (SaaS) Ebene. Es werden auch nicht verschiedene Software-Instanzen für verschiedene Nutzer instanziiert. Jeder Nutzer arbeitet auf einer für ihn passenden Anwendungsinstanz.

Im Zuge von Infrastruktur-Automatisierung ist dabei interessant, inwiefern kundenspezifische Anwendungsinstanzen erstellt werden. Im Zuge dessen soll auf Einweg-Cluster und Job-basierte Infrastruktur in 4.1 eingegangen werden und danach Sustainable Programming in 4.3 erwähnt werden, welches in diesem Zusammenhang Relevanz findet.

4.1 Einweg-Cluster und Job-basierte Infrastruktur

Wenn sich DevOps orientierte Teams mit Applikationsentwicklungs al auch mit Organisation und Konfiguration von Infrastruktur beschäftigen, ist es naheliegend, dass Infrastructure as Code Definitionsdateien neben dem Applikationscode aufbewahrt werden. Eine einheitliche Codebase hat den Vorteil, dass alle Dateien, die für das Deployment einer Applikation benötigt werden, sich an einem Ort befinden. Ein Automation Server könnte auf dieser Codebasis jederzeit das Setup starten.

In Abbildung 1 ist so ein Ablauf beispielhaft dargestellt. Basierend auf einer Codebase sollte der Automation Server in der Lage sein Ressourcen zu konfigurieren, mögliche Vernetzungen einzelner Komponenten sicherzustellen, ganze Cluster aufzusetzen und letztendlich die Applikation auszuliefern.

Job-basiert kann zum Beispiel die Anforderung sein, wenn zu einem festen Zeitpunkt Daten zur Verarbeitung ankommen. Anstatt das Setup für die Zeit neben der Verarbeitung aufrecht zu erhalten, kann es herunter skaliert werden (s. Abbildung 2). Das Einweg-Cluster treibt diesen Gedanken auf die Spitze, indem Ressourcen nach Vollendung der Aufgabe komplett herunterskaliert oder sogar freigegeben werden.

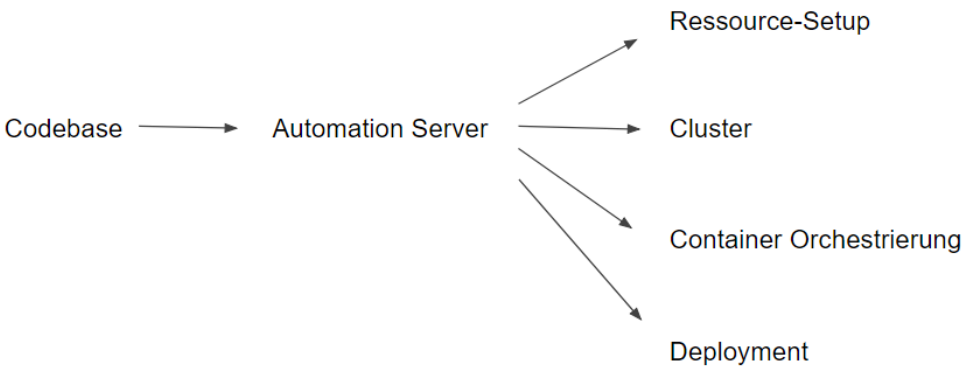

Abbildung 1: Beispielhafte Job-basierte Infrastruktur-Konfiguration

Abbildung 2: Zyklischer Ablauf eines theoretischen Anwendungsfalles für ein Cluster as a Service Szenario

4.2 Cluster as a Service

Eine Plattform steht vor der Herausforderung eine heterogene Applikationslandschaft zu unterstützen, Skalierbarkeit und Elastizität zu unterstützen und gleichzeit effizient alle Ressourcen aufzuteilen[5]. Automatisierte Ressourcenverteilung und die Möglichkeit auf Anfragen Cluster zu erstellen, fallen unter Cluster as a Service (ClaaS).

Die Autoren von von "Cluster as a Service: A Resource Sharing Approach for Private Cloud"[3] und "Cluster as a Service: a Container based Cluster Sharing Approach with multi-user Support"[5] schlagen dabei vor, dass die Cluster-Umgebung für die verschiedenen Applikationsframeworks virtualisiert wird und haben basierend auf leichtgewichtigen Containern ein ClaaS-System namens Docklet entwickelt.

Die grundsätzliche Idee hinter Cluster as a Service soll sein, dass ein fertig konfiguriertes Cluster auf Anfrage bereitgestellt werden kann.

4.3 Sustainable Programming

Sustainable Programming beschäftigt sich mit der Art und Weise wie so effizient programmiert wird, dass insgesamt Strom und somit Kosten gespart werden können. Durch gute Programmierung und damit einhergehender effizienter Nutzung von Hardware soll der Stromverbrauch einer Anwendung reduziert werden.

Betrachtet man dies nicht aus Sicht eines Anwendungsentwicklers[7], sondern global im Cloud Kontext, so findet das Thema schnell mehr Relevanz. Anwendungen sind auf verschiedene Rechenzentren verteilt und möglicherweise gegen Ausfälle mit Redundanz abgesichert.

Ziel kann es jetzt bei der Planung und Umsetzung von Infrastruktur-Konfigurationen sein, dass die Nachhaltigkeit dieser Software mit Mitteln der Automatisierung gefördert wird. Für einmalige Aufgaben, die immer wiederkehrend sind, könnte man überlegen, Job-basierte Infrastruktur aufzusetzen. Ist die Aufgabe erledigt, können Ressourcen wieder freigegeben werden oder möglicherweise sogar heruntergefahren werden.

[7] Sustainable Programming konzentriert sich eigentlich auf Softwareentwicklung, für allgemein IT wird der Begriff ”Green IT“ verwendet

5 Ausblick und Forschung

Forschung im Bereich Infrastructure as Code bewegt sich momentan in Richtung Fehlererkennung von defekten Skripten. Dabei gibt es Ausarbeitungen, die sich mit Code-Smells auseinandersetzen [12], und Ausarbeitungen, die sich mit defekten Skripten auseinandersetzen [11].

Eine mögliche Forschungsfrage ist der Wandel in Richtung Automatisierung. Dabei könnten Herausforderungen erarbeitet werden, vor welchen Unternehmen stehen, die ihre Infrastruktur automatisieren wollen.

Weiterhin kann mithilfe von CCA-Tools ein Wandel von Infrastructure as a Service und Plattform as a Service in Richtung Software as a Service eingeleitet werden. Beispielsweise können Cluster-Konfigurationen über Software gesteuert werden und solle Software von Infrastruktur- oder Plattform-Betreibern angeboten werden.

5.1 Weiterführende akademische Arbeit

Für die Projekte und weiterführende Arbeit soll eine Einarbeitung in verschiedene Infrastructure as Code Implementierungen in Form von CCA-Tools stattfinden, um eine Aussage treffen zu können, welches Tool für die Verwendung in Projekten am besten geeignet ist.

In den Projekten soll sich mit dem Thema Einweg-Cluster bzw. Cluster as a Service beschäftigt werden. Dabei gibt es die Vision, dass Kataloge für das Konfigurieren von ganzen Cluster (bspw. Kubernetes-Cluster) abhängig vom Anwendungsfall erstellt werden. Beispielhaft bestände die Möglichkeit, dass eine gewisse Infrastruktur zur Verfügung steht, die einzelnen Entwicklern unabhängig voneinander zu Verfügung gestellt werden soll. Wenn dann für die Bearbeitung eines Projekts ein Kubernetes-Cluster benötigt wird, soll dieses von einer zuständigen Person - oder sogar automatisiert - auf dieser Infrastruktur erstellt werden können, wobei alle Konfigurationsarbeit abgenommen wird. Die Auseinandersetzung mit Mandantenfähigkeit und dessen Automatisierung soll Fokus weiterer wissenschaftlicher Arbeit sein.

Literatur

[1] Artač, Matej ; Borovšak, Tadej ; Di Nitto, Elisabetta ; Guerriero, Michele ; Tamburri, Damian A.: DevOps: Introducing Infrastructure-as-code. In: *Proceedings of the 39th International Conference on Software Engineering Companion*. Piscataway, NJ, USA : IEEE Press, 2017 (ICSE-C '17), S. 497–498. – ISBN 978-1-5386-1589-8

[2] Benson, J. O. ; Prevost, J. J. ; Rad, P.: Survey of automated software deployment for computational and engineering research. In: *2016 Annual IEEE Systems Conference (SysCon)*, April 2016, S. 1–6

[3] Cao, D. ; Liu, P. ; Cui, W. ; An, B.: Cluster as a Service: A Resource Sharing Approach for Private Cloud, March 2016

[4] Chacon, Scott: *Los geht's - Wozu Versionskontrolle?*. – URL https://git-scm.com/book/de/v1/Los-geht%E2%80%99s-Wozu-Versionskontrolle%3F

[5] Cui, W. ; Zhan, H. ; Li, B. ; Wang, H. ; Cao, D.: Cluster as a Service: A Container Based Cluster Sharing Approach with Multi-user Support. In: *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, March 2016, S. 111–118

[6] Debois, Patrick ; Shafer, Andrew C.: *Agile Infrastructure Operations*. 2008. – URL http://www.jedi.be/presentations/agile-infrastructure-agile-2008.pdf

[7] Docs, Puppet: *Overview of Puppet's architecture*. 2018. – URL https://puppet.com/docs/puppet/4.6/architecture.html

[8] Ebert, C. ; Gallardo, G. ; Hernantes, J. ; Serrano, N.: DevOps. In: *IEEE Software* 33 (2016), May, Nr. 3, S. 94–100. – ISSN 0740-7459

[9] Morris, Kief: *Infrastructure as Code - Managing Servers in the Cloud*. Sebastopol, CA : O'Reilly, 2016. – First edition

[10] Rahman, A.: Anti-Patterns in Infrastructure as Code. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, S. 434–435

[11] Rahman, Akond: Characteristics of Defective Infrastructure As Code Scripts in DevOps. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. New York, NY, USA : ACM, 2018 (ICSE '18), S. 476–479. – ISBN 978-1-4503-5663-3

[12] Schwarz, J. ; Steffens, A. ; Lichter, H.: Code Smells in Infrastructure as Code. In: *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, Sep. 2018, S. 220–228

[13] UpGuard: *Ansible vs Ansible Tower*. 2017. – URL https://www.upguard.com/articles/ansible-vs.-ansible-tower

[14] Venezia, Paul: *Puppet vs. Chef vs. Ansible vs. Salt*. 2013. – URL https://www.networkworld.com/article/2172097/puppet-vs--chef-vs--ansible-vs--salt.html