CONFIGURATION MANAGEMENT

Puppet, Ansible, Chef & Co. aus der Vogelperspektive

Configuration-Management-Werkzeuge bilden den Zwischenschritt zwischen manueller Installation auf Servern und einer vollkommenen Automatisierung der Infrastruktur.

Wer mit unterschiedlichen Betriebssystemen gearbeitet hat, weiß auch, dass es unmöglich ist, ein Installationsskript zu schreiben, das auf allen Betriebssystemen läuft. Nicht nur sind Windows und Linux offensichtlich unterschiedliche Welten, auch die verschiedenen Linux-Distributionen unterscheiden sich voneinander. Wer etwa auf einer Standardinstallation von Ubuntu versucht, Pakete mit yum zu installieren, beißt sich die Zähne aus.

Klar können Unterschiede auf Betriebssystemebene in Skripten mit *if/then/else* gelöst werden, im Sinne von „Wenn Distribution Ubuntu, dann apt, sonst yum." Das geht dann so lange gut, bis ein weiteres Paketmanagementsystem auf den Plan tritt, wie zum Beispiel dnf bei Fedora. Natürlich kann man hier mit verschachtelten *ifs* und *elses* oder gar mit *switch*-Statements argumentieren, aber das resultierende Ergebnis ist in jedem Fall komplex und muss gewartet werden. Eine komplette Installation mit Skripten zu versehen ist zwar möglich, aber auf Dauer aufwendig und nervenaufreibend.

Automatisierung durch IaC

Wir leben in einer Zeit, in der die Cloud immer dominanter wird. Mittlerweile steht nicht mehr nur zur Debatte, einzelne Updates auf lokalen oder entfernten Servern zu automatisieren. Der Trend geht über Schlagwörter wie Cloud Native und Elastizität in die Richtung, ganze Serverinstanzen jederzeit neu aufzubauen und auch wieder zerstören zu können.

Als finaler Schritt kann dann letztlich eine komplett selbstheilende Systeminfrastruktur angesehen werden. Über eine vollständige Orchestrierung aller Workflows in einer Systemlandschaft können automatisch neue Instanzen hoch- und heruntergefahren werden, um beispielsweise ein höheres Lastaufkommen zu bewältigen oder defekte Knoten auszutauschen. Eingriffe durch einen System Engineer in ein Produktionssystem werden auf wenige Ausnahmefälle reduziert.

Ziel dieses Artikels ist, zu zeigen, welche Features für ein Configuration-Management-Werkzeug essenziell sind und wie sich eine Grenze zwischen Configuration Management und Provision Management ziehen lässt.

Container werden im Folgenden nicht behandelt, da die Behandlung von Containern und Servern den Rahmen des Artikels sprengen würde. Die hier vorgestellten Methoden lassen sich aber problemlos auf Container übertragen.

Wer jedoch vorrangig an einer kompletten Orchestrierung mit Containern interessiert ist, kann sofort Kubernetes auch als Standardlösung für Container evaluieren, da Kubernetes viele Konzepte, die über Configuration Management geschaffen wurden, übernommen und erweitert hat (vergleiche den Kasten Begriffsdefinition).

Provisionieren oder konfigurieren?

Historisch gesehen bedeutet ein Update in der IT, dass in einem System eine oder mehrere Komponenten mit neueren Versionen versehen werden. Die nicht vom Update betroffenen Komponenten im System werden dabei nicht angerührt.

Das mag für erfahrene Anwender banal klingen, aber mittlerweile – vor allem getrieben durch die Cloud – gibt es auch einen weiteren Weg. Nämlich selbst bei einem Upgrade von nur einzelnen Komponenten eine bestehende Serverinstanz zu dekommissionieren und parallel ein Serverimage mit aktualisierter Software auszurollen, welches das alte Image ersetzt.

Nehmen wir an, alle laufenden Serverinstanzen in einem Unternehmen sollen von Python 3.6 auf die Version 3.8 erhöht werden. Folgt die Organisation dem sogenannten Mutable-Infrastructure-Paradigma, erhält nur Python auf dem Server das Upgrade. Der Mehrwert der im Artikel vorgestellten Configuration-Management-Werkzeuge gegenüber einem händischen Upgrade ist, dass man sich nicht manuell auf alle Server verbinden muss, um den Update-Befehl auszuführen. Ein Update von einzelnen installierten Diensten und Software wird von einer Stelle aus getriggert und parallel automatisiert auf allen Knoten ausgeführt.

Ein Systemadministrator kann jedoch andererseits die Serverinstanz, wie davor angedeutet, auch komplett neu ausrollen. Hier würde nur ein Servertemplate aktualisiert werden und die resultierenden Serverinstanzen ausgerollt werden. Das wird Immutable-Infrastructure-Paradigma genannt. Man spricht dann auch nicht mehr vom Konfigurieren von Servern, sondern vom Provisionieren.

Für Personen, die vor dem DevOps-Paradigma bereits in der IT tätig waren, kann das ständige Provisionieren von neuen Serverinstanzen aufwendig und vor allem auch riskant klingen. Was, wenn irgendein System Engineer Anpassungen direkt am Server durchgeführt hat, die nicht automatisiert wurden? Wenn die alte Serverinstanz bei einem Update

gelöscht wird, könnten in einem solchen Fall sogar Änderungen verloren gehen. In einer Produktionsumgebung könnte das eine Katastrophe auslösen, wenn plötzlich Anpassungen fehlen, die ein früheres Produktionsproblem gelöst haben.

Es gibt jedoch auch eine starke Argumentation für unveränderliche Infrastrukturen, die vor allem in verteilten Umgebungen relevant sind. Die erste Argumentation lautet, Configuration Drift zu vermeiden. Werden mehrere Server verwaltet, können mit der Zeit immer wieder kleinere Abweichungen entstehen. Nehmen wir an, wir haben einen Kafka-Cluster und auf einem Knoten unterscheidet sich die Konfiguration in Details. Selbst kleine Veränderungen können ein nichtdeterministisches Verhalten zur Folge haben. Ein Albtraum für diejenigen Personen, die den Fehler suchen müssen.

Ein weiteres Phänomen ist Elastizität. Dienste sind aktuell zum Beispiel auf vier Knoten verteilt. Was aber, wenn wir nun zehn Knoten mit klar festgelegter Konfiguration brauchen?

Manuelle Änderungen – selbst wenn diese dokumentiert sind – sind streng genommen Antipatterns und stellen ein hohes Risiko dar. Manuelle Änderungen können verloren gehen, oder ihre Dokumentation wird möglicherweise übersehen. Das könnte auch dazu führen, dass Systeme nicht von Grund auf neu installiert werden können.

Configuration-Management-Werkzeuge wie Chef, Puppet, Ansible und SaltStack bauen typischerweise auf dem Mutable-Infrastructure-Paradigma auf. Diese Werkzeuge stammen teilweise aus einer Zeit, als die Cloud noch nicht die heute vorherrschende Dominanz besaß. Soll Python ein Upgrade erhalten, triggern diese Werkzeuge demnach ein Update, das auf zugeordneten Servern durchgeführt wird.

Als Gegenbeispiel kann Terraform gesehen werden, das als Provisionierungswerkzeug bei einem Update ein neues Serverimage ausrollt und die alten Serverimages löscht.

Prozedural contra deklarativ

Ein prozeduraler Stil kann als Schritt-für-Schritt-Anleitung verstanden werden, um ein Ziel zu erreichen. Ein deklarativer Stil beschreibt einen Endzustand, und das Tool muss selbst entscheiden, wie dieser Zustand erreicht werden soll.

Wie sieht das in der Praxis aus? Es sollen zehn EC2-Instanzen ausgerollt werden. In einem prozeduralen Stil könnte man das wie folgt definieren:

```
- ec2:
  count: 10
  image: ami-v1
  instance_type: t2.micro
```

Oder übersetzt würde man sagen: Instanziere zehn Images vom Typ *t2.micro*. Dieses Beispiel stammt von Ansible. Im Vergleich dazu schauen wir uns nun den deklarativen Stil an:

```
resource "aws_instance" "example" {
  count = 10
  ami = "ami-v1"
  instance_type = "t2.micro"
}
```

Begriffsdefinition

Der Begriff Configuration Management kann für Verwirrung sorgen, da der Begriff Konfigurationsmanagement in der Vergangenheit auch Werkzeuge wie Git umfasste. Heute hat sich der Begriff Version Control System für Tools durchgesetzt, die Versionen von Softwareständen verwalten.

Sprechen wir von Configuration-Management-Werkzeugen, so sind hier Werkzeuge gemeint, um Software auf bestehenden Servern zu installieren und zu verwalten. Die in diesem Artikel angeführten Werkzeuge sind Chef, Puppet, Ansible und SaltStack. Es gibt aber noch weitere Tools [2].

Von diesen abgegrenzt werden oft Werkzeuge wie CloudFormation und Terraform, die entwickelt wurden, um Server selbst zu provisionieren. Hier ist es schwierig, eine Grenze zwischen Provisioning und Configuration zu ziehen, weil sich die Domänen überlappen. Als wesentlichstes Unterscheidungsmerkmal kann angesehen werden, dass eben bei einer Provisionierung gleich ein ganzes Image ausgetauscht wird.

Rollen wir beide Lösungen ein erstes Mal (auf separaten Systemen) aus, so gibt es keine Unterschiede im Ergebnis. Am Ende laufen zehn neue Instanzen, oder es wird eine Fehlermeldung zurückgegeben.

Die Unterschiede werden bei Updates klar, denn der deklarative Ansatz gibt an, zehn Instanzen zu verwalten. Der prozedurale Ansatz hingegen hat die Aufgabe, zehn Instanzen zu erstellen.

Prozedurale Ansätze wie der von Ansible eignen sich für veränderliche Infrastrukturen. Jedes Ansible-Playbook beschreibt, wie ein System adaptiert wird. Man kann jederzeit noch einmal zehn weitere Instanzen hinzufügen.

Ziel von deklarativen Ansätzen ist es, einen gewünschten Ist-Zustand zu beschreiben, jedoch nicht, eine Veränderung als Delta zum existierenden Zustand auszuführen. Statt zehn neue Instanzen zu erstellen, würde ein deklarativer Ansatz bei einem erneuten Aufruf feststellen, dass zehn Instanzen bereits da sind, und entsprechend nichts tun, sofern nicht jemand Instanzen heruntergefahren oder neu hinzugefügt hat.

Die Quintessenz ist, dass die Auswahl eines Werkzeugs zur Philosophie einer Organisation passen muss. DevOps ist ein Weg von einem Do-it-yourself-IT-Betrieb hin zu einer Automatisierung. Für Systemadministratoren, die in alteingesessenen Unternehmen arbeiten und vielleicht darauf sozialisiert wurden, allem und jedem zu misstrauen, kann ein deklarativer Ansatz ein Horror sein.

Ein Nachteil von prozeduralen Ansätzen ist, dass der Letztstatus der Infrastruktur nicht im Code widergespiegelt sein muss. Man kann auch mit vielen Updates, die in einzelnen Playbooks enthalten sind, zu einem Ergebnis kommen. Wer Infrastrukturcode von unveränderlichen Infrastrukturen sieht, sieht immer den Letztstand.

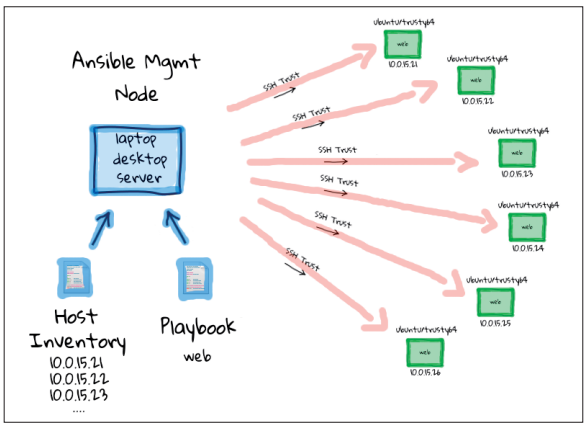
In deklarativen Sprachen ist man dafür etwas im Ausdruck eingeschränkt. Logische Operationen wie *if/then/else* sind ▶

die Domäne von prozeduralen Ansätzen und lassen sich nur schwer in deklarativen Ansätzen integrieren.

Master oder masterlos

Wir wissen also, dass Infrastruktur als Code beschrieben wird. Es stellt sich die Frage, ob die Informationen, wie eine Serverinfrastruktur auszusehen hat, zentral verwaltet werden sollen.

Ein Master-Server schafft hier eine Zentralisierung, die zu einer klaren Rollenverteilung führt. Nur vom Master aus werden auch Änderungen getriggert und Clients aktualisiert.



Der Ansible-Workflow im Überblick (Bild 1)

Zu diesen Master-Servern gibt es dann auch noch Web-UIs, um nicht allein auf die Kommandozeile reduziert zu sein.

Auch kann ein Master eine Schaltstelle sein, um im Hintergrund Clients zu verwalten. Wird ein Client verändert, kann der Ursprungszustand wiederhergestellt werden.

Tatsächlich lässt sich ein deklaratives System mit einem Master (und Agents) mit einem menschlichen Körper vergleichen. Was beim Menschen als Homöostase bekannt ist, kann auch gedanklich auf das Verwalten von Nodes übertragen werden: Ein System versucht immer wieder, auf einen bestimmten Normalzustand zu gelangen.

Ein Nachteil ist der Bedarf an zusätzlicher Infrastruktur. Man braucht einen Extraserver, auf dem der Master installiert werden muss. Dieser Server muss gewartet werden. Manche Systeme haben deswegen – analog zu Git und zentralen Repositories wie GitHub – auch optionale Masterknoten (Bild 1).

Agent versus agentless

Ob ein System agentless sein kann, ist vielleicht das markanteste Entscheidungsmerkmal. Zumindest wenn es nach StackShare [1] geht, wo viele Anwender an Puppet kritisieren, dass Agents erforderlich sind.

Chef, Puppet und SaltStack sind Werkzeuge, die mit Agents arbeiten. Diese werden dort Chef Client, Puppet Agent beziehungsweise Salt Minion genannt. Diese Agents sind auf allen Clients zu installieren, mit denen gearbeitet wird.

Der Agent gleicht kontinuierlich den Stand mit den Client-Spezifikationen des Masters ab. Gibt es eine Abweichung, wird der Ursprungszustand wiederhergestellt. Der Master schickt einen Katalog an die Clients, und die Clients stellen den gewünschten Zustand wieder her.

Viele sehen jedoch die Vorstellung, Agents installieren zu müssen, als einen Nachteil an. Agents müssen – wie alle Dienste – gewartet werden, und sie können auch ein System beeinflussen. Zudem müssen Agents jederzeit mit anderen Agents oder mit dem Master kommunizieren können. Das ist nicht immer erwünscht, ob aus Sicherheitsbedenken oder aufgrund von Wartungsaufwand und Komplexität.

Der Trend geht weg von Systemen, die Agents voraussetzen. Daher gibt es auch bei Werkzeugen, die bislang Agents benötigten, nun Versionen, die ohne Agents arbeiten können: Ein bekanntes Beispiel ist Bolt bei Puppet (vergleiche Bild 2).

Ansible ist eines der Werkzeuge, die ohne Agents arbeiten. Ganz ohne Software auf den Clients geht es natürlich auch nicht. Die Kommunikation zwischen Master und Clients erfolgt bei Ansible über SSH, und die Ausführung der Logik auf den entfernten Systemen basiert auf Python. Auf allen von Ansible verwalteten Zielsystemen müssen daher zumindest ein Python-Interpreter und ein SSH-Server installiert sein.

Community-Unterstützung

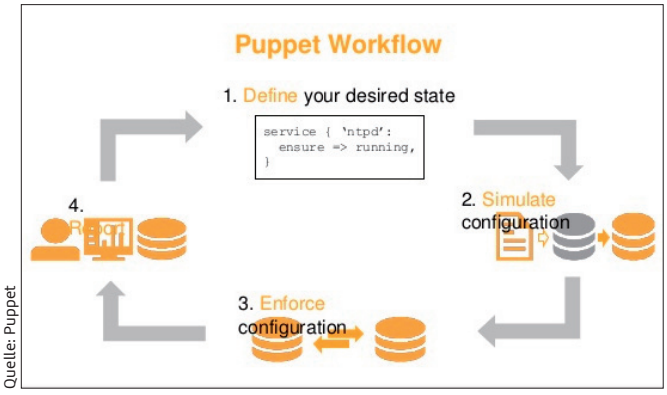
Mit jeder Technologieentscheidung wählt man auch eine Community aus. Dieser Aspekt einer Entscheidung kann von größerer Bedeutung sein als die Wahl der Technologie selbst.

Was tut ein Unternehmen, das sich einem Framework verschreibt, wenn es irgendwann keine Committer mehr gibt, die das Projekt weiter vorantreiben, oder nicht ausreichend Problemlösungen auf Stack Overflow verfügbar sind?

Wenn ein Team eine neue Technologie ausrollen will, gibt es dazu oft Module und Plug-ins. Ein Referenzbeispiel hierfür ist Confluent, eine Kafka-basierte Plattform. Auf der Confluent-Seite findet man offiziell unterstützte Ansible-Playbooks. Zu Puppet gibt es zwar Lösungen auf Puppet Forge, die Confluent deployen, diese werden aber nicht offiziell von Confluent supportet. Wer also auf Puppet setzt, arbeitet ohne Unterstützung des Herstellers.

Liegt dieser Umstand bei Confluent daran, dass Ansible mehr Aufmerksamkeit genießt? Um diese Frage beantworten zu können, gilt es zunächst herauszufinden, wie man ermittelt, ob eine Lösung einen starken Community-Support hat.

Hierfür lohnt es sich, die Werkzeuge auf StackShare, Stack Overflow und GitHub zu suchen. Auch Referenzen auf Jobbörsen sind hilfreich. Allerdings kann es bei der Untersuchung von Jobbörsen im Hinblick auf Configuration-Ma-



Quelle: Puppet

Deployment mit Puppet (Bild 2)

nagement-Werkzeuge Unschärfen geben, da Begriffe wie Puppet oder Chef bei Jobbörsen Ergebnisse zurückliefern können, die nicht direkt mit der DevOps-Technologie zusammenhängen müssen.

Was bleibt, ist: Für eine Technologieentscheidung auf Basis eines Community-Supports müssen mehrere Quellen berücksichtigt werden. Bei Features kann es unterschiedliche Meinungen geben. Entwickler gewinnen aber relativ schnell eine Übersicht zu Trends in der Community, weil mehrere Quellen sich oft bestätigen.

Würde man etwa die Häufigkeit der Nennungen von Ansible, Chef und Puppet in Stack Overflow, GitHub und StackShare vergleichen, würde sich klar zeigen, dass Ansible die meiste Aufmerksamkeit bekommt (vergleiche Bild 3).

Mature versus Cutting Edge

Schwieriger ist es, den Reifegrad bei Configuration-Werkzeugen festzulegen. Normalerweise gilt: Je älter, desto reifer. Puppet wurde das erste Mal im Jahr 2005 releast, zu einer Zeit, als auch AWS – als erster Cloud-Provider – noch in den Kinderschuhen steckte. Ist dieses Werkzeug deswegen auch das reifste aller Configuration-Management-Werkzeuge?

Im Fall von Configuration Management können wir die Metrik aber auch ignorieren, da modernere Systeme oft als Provisionswerkzeuge kategorisiert sind.

Implementierungsdetails

Implementierungsdetails können ebenfalls für manche Entscheidungen relevant sein. In welcher Sprache wird Infrastruktur codiert? YAML scheint ein Standard geworden zu sein, um Infrastrukturen zu beschreiben. JSON oder gar XML findet man immer seltener.

Auch die Programmiersprache kann relevant sein, in der ein Werkzeug geschrieben ist. Ruby hatte in den Millenniumsjahren einen Hype. In dieser Zeit sind Puppet und Chef entstanden, und sie sind in Ruby implementiert.

Ansible und Terraform, die später auf den Markt gekommen sind, setzen auf Python beziehungsweise Go. Wer in die Interna seines Tools schauen will, für den kann die verwendete Sprache durchaus relevant sein.

Technologiespezifische Lösungen

Manche Lösungen wurden für bestimmte Plattformen gebaut. Mit Ambari Blueprints kann man Hadoop-Cluster spezifizieren, während sich mit CloudFormation AWS-Cluster designen lassen. Wer sich definitiv einer Technologie verschreibt, für den kann eine solche Option interessant sein. Für diesen Artikel werden diese Lösungen nicht berücksichtigt.

Lizenz

Nicht immer wird nach der Lizenz gefragt, unter der eine Software zur Verfügung steht. Für viele ist es aber ein wichtiges Merkmal, unter welcher Lizenz eine Lösung läuft.

Die meisten Lösungen haben einen Open-Source-Kern und eine Enterprise-Version, die zusätzliche Funktionalität bietet. In diesem Sinne ist die Lizenz vielleicht zweitrangig, da die Entwickler eines Werkzeugs versuchen, ihre Kunden mit Open-Source-Funktionalität vertraut zu machen, um dann gute Karten für einen Supportvertrag zu haben, wenn eine Lösung in Produktion gehen soll.

Fastest rising tech skills, 2014-2019

% of tech jobs, Sept 2014-to-Sept 2019, at least 0.1% in each period

Rank	Skill	2014 share	2019 share	% change
1	docker	0.1%	5.1%	4162%
2	iot	0.1%	2.1%	1994%
3	ansible	0.2%	2.8%	1292%
4	kafka	0.2%	2.4%	1216%
5	azure	0.6%	6.9%	1107%
6	spark	0.3%	3.5%	1068%
7	artificial intelligence	0.3%	2.0%	701%
8	redshift	0.2%	1.2%	564%
9	swift	0.2%	1.1%	481%
10	machine learning	1.3%	7.0%	439%
11	angular	0.9%	4.9%	427%
12	aws	2.7%	14.2%	418%
13	elasticsearch	0.3%	1.4%	333%
14	servicenow	0.2%	1.0%	333%
15	tableau	0.8%	2.9%	275%
16	gradle	0.2%	0.7%	254%
17	jenkins	1.4%	5.0%	251%
18	splunk	0.5%	1.7%	238%
19	scala	0.6%	2.1%	235%
20	jira	1.5%	4.9%	232%

Quelle: Indeed

Die IT-Skills mit dem größten Zuwachs an Beliebheit (Bild 3)

Zusammenfassung

In diesem Artikel haben wir uns weniger den Implementierungsdetails einer Technologie gewidmet. Stattdessen haben wir vorgestellt, wie sich die Ansätze unterscheiden.

Wer nur Updates auf mehreren Servern automatisiert ausrollen und dazu auf das Tool setzen will, das hierfür am weitesten verbreitet ist, der trifft mit Ansible eine gute Wahl.

Puppet kann eine gute Alternative sein, wenn man sicherstellen will, dass der Ausgangszustand wiederhergestellt wird, sobald eine Konfiguration manuell verändert wird.

Die Popularität von Ansible kann aber auch anderes unterstreichen. Ansible ist vielleicht das Werkzeug mit der geringsten Lernkurve, und auch das Werkzeug, das einem DevOps-Engineer am wenigsten einen Prozess aufzwingt. Eine mögliche Erklärung kann entsprechend sein, dass DevOps zwar in aller Munde ist, aber viele Firmen dann doch einen Weg bevorzugen, der ihnen viele Freiheiten lässt. ■

[1] StackShare, www.dotnetpro.de/SL2006Configuration1
[2] Comparison of open-source configuration management software, www.dotnetpro.de/SL2006Configuration2

Stefan Papp
arbeitet als freiberuflicher Big-Data-Architekt und -Evangelist und begleitet zahlreiche Kunden dabei, sich mit der Analyse von neuen Datenquellen selbst neu zu definieren.

dnpCode A2006Configuration