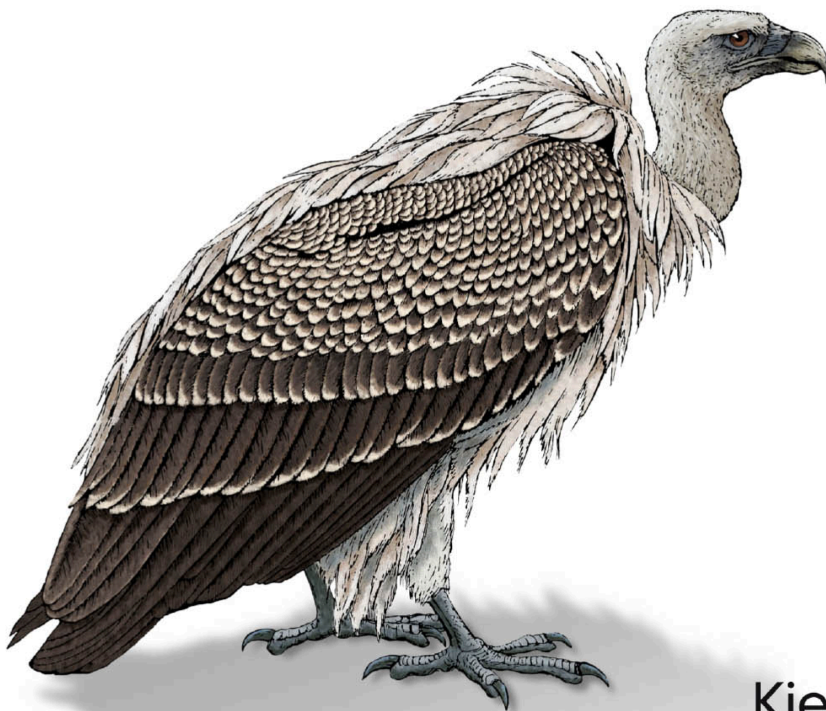
O'REILLY®

Übersetzung der 2. Auflage

Handbuch Infrastructure as Code

Prinzipien, Praktiken und Patterns für eine cloudbasierte IT-Infrastruktur



Kief Morris

Übersetzung von Thomas Demmig

Inhalt

Cover

Titel

Impressum

Inhalt

Vorwort

Warum ich dieses Buch geschrieben habe

Was in dieser Auflage neu und anders ist

Was kommt als Nächstes?

Was dieses Buch ist und was es nicht ist

Etwas Geschichte zu Infrastructure as Code

Für wen dieses Buch gedacht ist

Prinzipien, Praktiken und Patterns

Die ShopSpinner-Beispiele

In diesem Buch verwendete Konventionen

Danksagung

Teil I Grundlagen

1 Was ist Infrastructure as Code?

Aus der Eisenzeit in das Cloud-Zeitalter

Infrastructure as Code

Vorteile von Infrastructure as Code

Infrastructure as Code nutzen, um für Änderungen zu optimieren

Einwand »Wir haben gar nicht so häufig Änderungen, sodass sich Automation nicht lohnt«

Einwand »Wir sollten erst bauen und danach automatisieren«

Einwand »Wir müssen zwischen Geschwindigkeit und Qualität entscheiden«

Die Four Key Metrics

Drei zentrale Praktiken für Infrastructure as Code

Zentrale Praktik: Definieren Sie alles als Code

Zentrale Praktik: Kontinuierliches Testen und die gesamte aktuelle Arbeit ausliefern

Zentrale Praktik: Kleine einfache Elemente bauen, die Sie unabhängig voneinander ändern können

Zusammenfassung

2 Prinzipien der Infrastruktur im Cloud-Zeitalter

Prinzip: Gehen Sie davon aus, dass Systeme unzuverlässig sind

Prinzip: Alles reproduzierbar machen

Fallstrick: Snowflake-Systeme

Prinzip: Erstellen Sie wegwerfbare Elemente

Prinzip: Variationen minimieren

Konfigurationsdrift

Prinzip: Stellen Sie sicher, dass Sie jeden Prozess wiederholen können

Zusammenfassung

3 Infrastruktur-Plattformen

Die Elemente eines Infrastruktur-Systems

Dynamische Infrastruktur-Plattform

Infrastruktur-Ressourcen

Computing-Ressourcen

Storage-Ressourcen

Networking-Ressourcen

Zusammenfassung

4 Zentrale Praktik: Definieren Sie alles als Code

Warum Sie Ihre Infrastruktur als Code definieren sollten

Was Sie als Code definieren können

Wählen Sie Werkzeuge mit externalisierter Konfiguration aus

Managen Sie Ihren Code in einer Versionsverwaltung

Programmiersprachen für Infrastruktur
Infrastruktur-Scripting
Deklarative Infrastruktur-Sprachen
Programmierbare, imperative Infrastruktur-Sprachen
Deklarative und imperative Sprachen für Infrastruktur
Domänenspezifische Infrastruktur-Sprachen
Allgemein nutzbare Sprachen und DSLs für die Infrastruktur
Implementierungs-Prinzipien beim Definieren von Infrastructure as Code
Halten Sie deklarativen und imperativen Code voneinander getrennt
Behandeln Sie Infrastruktur-Code wie echten Code
Zusammenfassung
Teil II Arbeiten mit Infrastruktur-Stacks
5 Infrastruktur-Stacks als Code bauen
Was ist ein Infrastruktur-Stack?
Stack-Code
Stack-Instanzen
Server in einem Stack konfigurieren
Low-Level-Infrastruktur-Sprachen
High-Level-Infrastruktur-Sprachen
Patterns und Antipatterns für das Strukturieren von Stacks
Antipattern: Monolithic Stack
Pattern: Application Group Stack
Pattern: Service Stack
Pattern: Micro Stack
Zusammenfassung
6 Umgebungen mit Stacks bauen
Worum es bei Umgebungen geht
Auslieferungsumgebungen

Mehrere Produktivumgebungen
Umgebungen, Konsistenz und Konfiguration
Patterns zum Bauen von Umgebungen
Antipattern: Multiple-Enviroment Stack
Antipattern: Copy-Paste Environments
Pattern: Reusable Stack
Umgebungen mit mehreren Stacks erstellen
Zusammenfassung
7 Stack-Instanzen konfigurieren
Eindeutige Kennungen durch Stack-Parameter erstellen
Beispiel-Stack-Parameter
Patterns zum Konfigurieren von Stacks
Antipattern: Manual Stack Parameters
Pattern: Stack Environment Variables
Pattern: Scripted Parameters
Pattern: Stack Configuration Files
Pattern: Wrapper-Stack
Pattern: Pipeline Stack Parameters
Pattern: Stack Parameter Registry
Konfigurations-Registry
Eine Konfigurations-Registry implementieren
Eine oder mehrere Konfigurations-Registries
Secrets als Parameter nutzen
Secrets verschlüsseln
Secretlose Autorisierung
Secrets zur Laufzeit injizieren
Wegwerf-Secrets
Zusammenfassung

8 Zentrale Praktik: Kontinuierlich testen und ausliefern
Warum Infrastruktur-Code kontinuierlich testen?
Was kontinuierliches Testen bedeutet
Was sollten wir bei der Infrastruktur testen?
Herausforderungen beim Testen von Infrastruktur-Code
Herausforderung: Tests für deklarativen Code haben häufig nur einen geringen Wert
Herausforderung: Das Testen von Infrastruktur-Code ist langsam
Herausforderung: Abhängigkeiten verkomplizieren die Test-Infrastruktur
Progressives Testen
Testpyramide
Schweizer-Käse-Testmodell
Infrastruktur-Delivery-Pipelines
Pipeline-Stages
Scope von Komponenten, die in einer Stage getestet werden
Scope von Abhängigkeiten für eine Stage
Plattformelemente, die für eine Stage erforderlich sind
Software und Services für die Delivery-Pipeline
Testen in der Produktivumgebung
Was Sie außerhalb der Produktivumgebung nicht nachbauen können
Die Risiken beim Testen in der Produktivumgebung managen
Zusammenfassung
9 Infrastruktur-Stacks testen
Beispiel-Infrastruktur
Der Beispiel-Stack
Pipeline für den Beispiel-Stack
Offline-Test-Stages für Stacks
Syntax-Checks

Statische Offline-Code-Analyse
Statische Code-Analyse per API
Testen mit einer Mock-API
Online-Test-Stages für Stacks
Preview: Prüfen, welche Änderungen vorgenommen werden
Verifikation: Aussagen über Infrastruktur-Ressourcen treffen
Ergebnisse: Prüfen, dass die Infrastruktur korrekt arbeitet
Test-Fixtures für den Umgang mit Abhängigkeiten verwenden
Test-Doubles für Upstream-Abhängigkeiten
Test-Fixtures für Downstream-Abhängigkeiten
Komponenten refaktorieren, um sie isolieren zu können
Lebenszyklus-Patterns für Testinstanzen von Stacks
Pattern: Persistent Test Stack
Pattern: Ephemeral Test Stack
Antipattern: Dual Persistent and Ephemeral Stack Stages
Pattern: Periodic Stack Rebuild
Pattern: Continuous Stack Reset
Test-Orchestrierung
Unterstützen Sie lokales Testen
Vermeiden Sie eine enge Kopplung mit Pipeline-Tools
Tools zur Test-Orchestrierung
Zusammenfassung
Teil III Mit Servern und anderen Anwendungs-Laufzeitplattformen arbeiten
10 Anwendungs-Laufzeitumgebungen
Cloud-native und anwendungsgesteuerte Infrastruktur
Ziele für eine Anwendungs-Laufzeitumgebung
Deploybare Teile einer Anwendung
Deployment-Pakete

Anwendungen auf Server deployen

Anwendungen als Container verpacken

Anwendungen auf Server-Cluster deployen

Anwendungen auf Anwendungs-Cluster deployen

Pakete zum Deployen von Anwendungen auf Cluster

FaaS-Serverless-Anwendungen deployen

Anwendungsdaten

Datenschemata und -strukturen

Cloud-native Storage-Infrastruktur für Anwendungen

Anwendungs-Connectivity

Service Discovery

Zusammenfassung

11 Server als Code bauen

Was gibt es auf einem Server

Woher Dinge kommen

Server-Konfigurationscode

Code-Module für die Serverkonfiguration

Code-Module für die Serverkonfiguration designen

Server-Code versionieren und weitergeben

Serverrollen

Server-Code testen

Server-Code progressiv testen

Was Sie bei Server-Code testen

Wie Sie Server-Code testen

Eine neue Server-Instanz erstellen

Eine neue Server-Instanz per Hand erstellen

Einen Server mit einem Skript erstellen

Einen Server mit einem Stack-Management-Tool erstellen

Die Plattform für das automatische Erstellen von Servern konfigurieren
Einen Server mit einem Network-Provisioning-Tool erstellen
Server vorbereiten
Hot-Cloning eines Servers
Einen Server-Snapshot verwenden
Ein sauberes Server-Image erstellen
Eine neue Server-Instanz konfigurieren
Eine Server-Instanz ausbacken
Server-Images backen
Backen und Ausbacken kombinieren
Serverkonfiguration beim Erstellen eines Servers anwenden
Zusammenfassung
12 Änderungen an Servern managen
Patterns zum Changemanagement: Wann Änderungen angewendet werden
Antipattern: Apply on Change
Pattern: Continuous Configuration Synchronization
Pattern: Immutable Server
Wie Sie Serverkonfigurationscode anwenden
Pattern: Push Server Configuration
Pattern: Pull Server Configuration
Andere Ereignisse im Lebenszyklus eines Servers
Eine Server-Instanz stoppen und erneut starten
Eine Server-Instanz ersetzen
Einen ausgefallenen Server wiederherstellen
Zusammenfassung
13 Server-Images als Code
Ein Server-Image bauen

Warum ein Server-Image bauen?
Wie Sie ein Server-Image bauen
Tools zum Bauen von Server-Images
Online Image Building
Offline Image Building
Ursprungsinhalte für ein Server-Image
Aus einem Stock-Server-Image bauen
Ein Server-Image von Grund auf bauen
Herkunft eines Server-Image und seiner Inhalte
Ein Server-Image ändern
Ein frisches Image aufwärmen oder backen
Ein Server-Image versionieren
Server-Instanzen aktualisieren, wenn sich ein Image ändert
Ein Server-Image in mehreren Teams verwenden
Umgang mit größeren Änderungen an einem Image
Eine Pipeline zum Testen und Ausliefern eines Server-Image verwenden
Build-Stage für ein Server-Image
Test-Stage für ein Server-Image
Delivery-Stages für ein Server-Image
Mehrere Server-Images verwenden
Server-Images für unterschiedliche Infrastruktur-Plattformen
Server-Images für unterschiedliche Betriebssysteme
Server-Images für unterschiedliche Hardware-Architekturen
Server-Images für unterschiedliche Rollen
Server-Images in Schichten erstellen
Code für mehrere Server-Images verwenden
Zusammenfassung
14 Cluster als Code bauen

Lösungen für Anwendungs-Cluster
Cluster as a Service
Packaged Cluster Distribution
Stack-Topologien für Anwendungs-Cluster
Monolithischer Stack, der Cluster as a Service nutzt
Monolithischer Stack für eine Packaged-Cluster-Lösung
Pipeline für einen monolithischen Anwendungs-Cluster-Stack
Beispiel für mehrere Stacks in einem Cluster
Strategien zur gemeinsamen Verwendung von Anwendungs-Clustern
Ein großes Cluster für alles
Getrennte Cluster für Auslieferungs-Stages
Cluster für die Governance
Cluster für Teams
Service Mesh
Infrastruktur für FaaS Serverless
Zusammenfassung
Teil IV Infrastruktur designen
15 Zentrale Praktik: Kleine, einfache Elemente
Für Modularität designen
Eigenschaften gut designter Komponenten
Regeln für das Designen von Komponenten
Design-Entscheidungen durch Testen
Infrastruktur modularisieren
Stack-Komponenten versus Stacks als Komponenten
Einen Server in einem Stack verwenden
Grenzen zwischen Komponenten ziehen
Grenzen mit natürlichen Änderungsmustern abstimmen
Grenzen mit Komponenten-Lebenszyklen abstimmen

Grenzen mit Organisationsstrukturen abstimmen
Grenzen schaffen, die Resilienz fördern
Grenzen schaffen, die Skalierbarkeit ermöglichen
Grenzen auf Sicherheits- und Governance-Aspekte abstimmen
Zusammenfassung
16 Stacks aus Komponenten bauen
Infrastruktur-Sprachen für Stack-Komponenten
Deklarativen Code mit Modulen wiederverwenden
Stack-Elemente dynamisch mit Bibliotheken erstellen
Patterns für Stack-Komponenten
Pattern: Facade Module
Antipattern: Obfuscation Module
Antipattern: Unshared Module
Pattern: Bundle Module
Antipattern: Spaghetti Module
Pattern: Infrastructure Domain Entity
Eine Abstraktionsschicht bauen
Zusammenfassung
17 Stacks als Komponenten einsetzen
Abhängigkeiten zwischen Stacks erkennen
Pattern: Resource Matching
Pattern: Stack Data Lookup
Pattern: Integration Registry Lookup
Dependency Injection
Zusammenfassung
Teil V Infrastruktur bereitstellen
18 Infrastruktur-Code organisieren
Projekte und Repositories organisieren

Ein Repository oder viele?
Ein Repository für alles
Ein eigenes Repository für jedes Projekt (Microrepo)
Mehrere Repositories mit mehreren Projekten
Unterschiedliche Arten von Code organisieren
Projektsupport-Dateien
Projektübergreifende Tests
Dedizierte Projekte für Integrationstests
Code anhand des Domänenkonzepts organisieren
Dateien mit Konfigurationswerten organisieren
Infrastruktur- und Anwendungscode managen
Infrastruktur und Anwendungen ausliefern
Anwendungen mit Infrastruktur testen
Infrastruktur vor der Integration testen
Infrastruktur-Code zum Deployen von Anwendungen nutzen
Zusammenfassung
19 Infrastruktur-Code ausliefern
Auslieferungsprozess von Infrastruktur-Code
Ein Infrastruktur-Projekt bauen
Infrastruktur-Code als Artefakt verpacken
Infrastruktur-Code mit einem Repository ausliefern
Projekte integrieren
Pattern: Build-Time Project Integration
Pattern: Delivery-Time Project Integration
Pattern: Apply-Time Project Integration
Infrastruktur-Tools durch Skripte verpacken
Konfigurationswerte zusammenführen
Wrapper-Skripte vereinfachen

Zusammenfassung
20 Team-Workflows
Die Menschen
Wer schreibt Infrastruktur-Code?
Code auf Infrastruktur anwenden
Code von Ihrem lokalen Rechner aus anwenden
Code von einem zentralisierten Service anwenden lassen
Private Infrastruktur-Instanzen
Quellcode-Branches in Workflows
Konfigurationsdrift verhindern
Automatisierungs-Verzögerung minimieren
Ad-hoc-Anwendung vermeiden
Code kontinuierlich anwenden
Immutable Infrastruktur
Governance in einem Pipeline-basierten Workflow
Zuständigkeiten neu ordnen
Shift Left
Ein Beispielprozess für Infrastructure as Code mit Governance
Zusammenfassung
21 Infrastruktur sicher ändern
Reduzieren Sie den Umfang von Änderungen
Kleine Änderungen
Refaktorieren – ein Beispiel
Unvollständige Änderungen in die Produktivumgebung übernehmen
Parallele Instanzen
Abwärtskompatible Transformationen
Feature Toggles
Live-Infrastruktur ändern

Infrastruktur-Chirurgie
Expand and Contract
Zero-Downtime-Änderungen
Kontinuität
Kontinuität durch das Verhindern von Fehlern
Kontinuität durch schnelles Wiederherstellen
Kontinuierliches Disaster Recovery
Chaos Engineering
Für Ausfälle planen
Datenkontinuität in einem sich ändernden System
Sperren
Aufteilen
Replizieren
Neu laden
Ansätze zur Datenkontinuität mischen
Zusammenfassung
Index
Über den Autor
Kolophon

KAPITEL 4

Zentrale Praktik: Definieren Sie alles als Code

In Kapitel 1 habe ich drei zentrale Praktiken identifiziert, die Ihnen dabei helfen, Infrastruktur schnell und zuverlässig zu ändern: Definieren Sie alles als Code, testen Sie Ihre Arbeit kontinuierlich und liefern Sie sie aus und bauen Sie kleine, einfache Elemente.

Dieses Kapitel befasst sich genauer mit der ersten dieser zentralen Praktiken und beginnt dabei mit den banalen Fragen. Warum sollten Sie Infrastructure as Code definieren wollen? Was für Arten von Elementen können und sollten Sie als Code definieren?

Auf den ersten Blick scheint »definieren Sie alles als Code« im Kontext dieses Buchs offensichtlich zu sein. Aber die Charakteristiken der verschiedenen Arten von Sprachen sind für die folgenden Kapitel von Relevanz. Insbesondere Kapitel 5 beschreibt den Einsatz deklarativer Sprachen zum Definieren von Low-Level-Stacks (siehe »Low-Level-Infrastruktur-Sprachen« auf Seite 84) oder von High-Level-Stacks (siehe »High-Level-Infrastruktur-Sprachen« auf Seite 85), und in Kapitel 16 wird erklärt, wann deklarativer oder programmierbarer Code zum Erzeugen von wiederverwendbaren Code-Modulen und Bibliotheken am passendsten ist.

Warum Sie Ihre Infrastruktur als Code definieren sollten

Es gibt einfachere Wege, Infrastruktur zu provisionieren, als einen Haufen Code zu schreiben und ihn an ein Tool zu verfüttern. Rufen Sie einfach die webbasierte Benutzeroberfläche der Plattform Ihrer Wahl auf und sorgen Sie per Mausklicks dafür, dass ein Anwendungsserver-Cluster zum Leben erweckt wird. Oder Sie wechseln zum Prompt und nutzen Ihre Befehlszeilenmagie, um mithilfe des CLI-(Command-Line-Interface-)Tools des Anbieters einen unzerstörbaren Netzwerk-Wall zu schaffen.

Aber im Ernst: Die vorigen Kapitel haben erklärt, warum es besser ist, Ihre Systeme mithilfe von Code zu bauen – unter anderem wegen der Wiederverwendbarkeit, Konsistenz und Transparenz (siehe »Zentrale Praktik: Definieren Sie alles als Code« auf Seite 40).

Durch das Implementieren und Managen Ihres Systems als Code haben Sie die Möglichkeit, schneller zu werden und dadurch die Qualität zu verbessern. Das ist die geheime Zutat, die die High Performer einsetzen, welche durch die Four Key Metrics bestimmt werden (siehe »Die Four Key Metrics« auf Seite 39).

Was Sie als Code definieren können

Jedes Infrastruktur-Werkzeug besitzt einen anderen Namen für seinen Quellcode – zum Beispiel Playbooks, Cookbooks, Manifeste und Templates. Ich beziehe mich darauf ganz allgemein als *Infrastruktur-Code* oder manchmal auch als *Infrastruktur-Definition*.

Infrastruktur-Code spezifiziert sowohl die Infrastruktur-Elemente, die Sie haben wollen, als auch ihre Konfiguration. Sie führen ein Infrastruktur-Tool aus, um Ihren Code auf eine Instanz Ihrer Infrastruktur anzuwenden. Das Tool erstellt entweder neue Infrastruktur oder passt bestehende an, sodass sie zu dem passt, was Sie in Ihrem Code definiert haben.

Zu den Dingen, die Sie als Code definieren sollten, gehören unter anderem:

- Ein Infrastruktur-Stack, bei dem es sich um eine Zusammenstellung von Elementen handelt, die von einer Infrastruktur-Cloud-Plattform provisioniert werden. In Kapitel 3 finden Sie mehr über Infrastruktur-Plattformen, in Kapitel 5 erhalten Sie eine Einführung in das Konzept des Infrastruktur-Stacks.
- Elemente einer Serverkonfiguration, wie zum Beispiel Pakete, Dateien, User-Accounts und Services (siehe Kapitel 11).
- Eine Server-Rolle ist eine Zusammenstellung von Server-Elementen, die gemeinsam auf eine einzelne Server-Instanz angewendet werden (siehe »Serverrollen« auf Seite 212).
- Eine Server-Image-Definition erzeugt ein Image zum Erstellen mehrerer Server-Instanzen (siehe »Tools zum Bauen von Server-Images« auf Seite 247).
- Ein Anwendungspaket definiert, wie ein deploybares Anwendungsartefakt gebaut wird. Dazu können auch Container

gehören (siehe Kapitel 10).

- Konfiguration und Skripte zum Ausliefern von Services, wozu auch Pipelines und Deployment gehören (siehe »Software und Services für die Delivery-Pipeline« auf Seite 156).
- Konfiguration für Operations-Services, wie zum Beispiel Monitoring-Checks.
- Validierungsregeln, zu denen automatisierte Tests und Compliance-Regeln gehören (siehe Kapitel 8).

Wählen Sie Werkzeuge mit externalisierter Konfiguration aus

Zu Infrastructure as Code gehört per Definition das Spezifizieren Ihrer Infrastruktur in textbasierten Dateien. Sie managen diese Dateien getrennt von den eingesetzten Werkzeugen, mit denen Sie sie auf Ihr System anwenden. Sie können Ihre Spezifikationen mit beliebigen Tools lesen, bearbeiten, analysieren und manipulieren.

Noncode-Infrastruktur-Automations-Werkzeuge legen Infrastruktur-Definitionen als Daten ab, auf die Sie keinen direkten Zugriff haben. Stattdessen können Sie die Spezifikationen nur über die Tools selbst verwenden und bearbeiten. Die Werkzeuge besitzen dafür im Zweifel eine Kombination aus GUI, API und Befehlszeilen-Schnittstelle.

Das Problem bei diesen Closed-Box-Tools ist, dass sie die Praktiken und Abläufe einschränken, die Sie nutzen können:

- Eine Versionierung Ihrer Infrastruktur-Spezifikationen ist nur möglich, wenn das Tool diese selbst unterstützt.
- CI funktioniert nur, wenn das Werkzeug eine Möglichkeit anbietet, einen Job automatisch auszulösen, wenn Sie eine Änderung vornehmen.
- Sie können Delivery Pipelines nur erstellen, wenn das Tool es einfach macht, Ihre Infrastruktur-Spezifikationen zu versionieren und zu verteilen.

Lehren aus dem Softwarequellcode

Das Pattern zur externalisierten Konfiguration spiegelt die Art und Weise wider, wie der größte Teil des Softwarequellcodes eingesetzt wird. Manche Entwicklungsumgebungen verstecken den Quellcode, wie zum Beispiel Visual Basic for Applications. Aber für nichttriviale Systeme erkennen Entwicklerinnen und Entwickler, dass es viel leistungsfähiger ist, den Quellcode in externen Dateien zu verwalten.

Es ist sehr schwierig, agile Entwicklungspraktiken wie TDD, CI oder CD mit Closed-Box-Tools zum Infrastruktur-Management umzusetzen.

Ein Werkzeug, das externen Code für seine Spezifikationen einsetzt, schränkt Sie nicht auf einen spezifischen Workflow ein. Sie können eine Standard-Versionsverwaltung einsetzen, Ihren Lieblingseditor, einen beliebigen CI-Server und automatisierte Test-Frameworks nutzen. Sie können Delivery Pipelines mit dem Tool aufbauen, das für Sie am besten funktioniert.

Managen Sie Ihren Code in einer Versionsverwaltung

Definieren Sie Ihre Elemente als Code, ist es einfach und sehr hilfreich, diesen Code in einer Versionsverwaltung (VCS, Version Control System) abzulegen. Dadurch wird Folgendes möglich:

Nachverfolgbarkeit

In der Versionsverwaltung findet sich die Historie der Änderungen, einschließlich der ändernden Person und des Kontexts.[1] Diese Historie ist unbezahlbar, wenn es um das Debuggen von Problemen geht.

Rollback

Geht durch eine Änderung etwas kaputt – insbesondere, wenn mehrere Änderungen etwas kaputt machen –, ist es nützlich, Dinge wieder auf den alten Stand zurücksetzen zu können.

Korrelation

Indem Sie Skripte, Spezifikationen und die Konfiguration unter Versionsverwaltung halten, können Sie knifflige Probleme besser verfolgen und beheben. Sie können mit Tags und Versionsnummern Verbindungen zwischen Elementen herstellen.

Sichtbarkeit

Jeder kann jede Änderung sehen, die in die Versionsverwaltung eingecheckt wurde, sodass das gesamte Team ein Lagebewusstsein bekommt. Vielleicht fällt jemandem auf, dass in einer Änderung etwas Wichtiges fehlt. Kommt es zu einem Vorfall, denken die Teammitglieder darüber nach, ob die letzten Commits dafür gesorgt haben könnten.

Aktionsfähigkeit

Die Versionsverwaltung kann für jede eingecheckte Änderung automatisch eine Aktion auslösen. Solche Trigger ermöglichen CI-Jobs und CD-Pipelines.

Was Sie nicht in die Versionsverwaltung stecken sollten, sind unverschlüsselte Secrets, wie zum Beispiel Passwörter und Schlüssel. Auch wenn Ihr Quellcode-Repository privat ist, sickern die Historie und die Revisionen des Codes doch allzu leicht durch. Secrets, die über Quellcode bekannt werden, sind eine der häufigsten Ursachen für Sicherheitslecks. In »Secrets als Parameter nutzen« auf Seite 133 beschreibe ich bessere Möglichkeiten für den Umgang mit Secrets.

Programmiersprachen für Infrastruktur

Systemadministratoren nutzen schon seit Jahrzehnten Skripte, um das Infrastruktur-Management zu automatisieren. Allgemeine Skriptsprachen wie Bash, Perl, PowerShell, Ruby oder Python haben immer noch einen wichtigen Anteil am Werkzeugkasten eines Infrastruktur-Teams.

CFEngine (*https://cfengine.com*) war Pionier beim Einsatz deklarativer, domänenspezifischer Sprachen (DSL, Domain-Specific Languages, siehe »Domänenspezifische Infrastruktur-Sprachen« auf Seite 75) zum Infrastruktur-Management. Puppet (*https://puppet.com*) und dann Chef (*https://www.chef.io*) entwickelten sich parallel zur Server-Virtualisierung und zu IaaS-Clouds im Mainstream. Ansible (*https://www.ansible.com*), Saltstack (*https://www.saltstack.com*) und andere folgten.

Stack-orientierte Tools wie Terraform (*https://www.terraform.io*) und CloudFormation (*https://oreil.ly/wt4pC*) kamen ein paar Jahre später dazu, wobei das gleiche deklarative DSL-Modell zum Einsatz kam. Deklarative Sprachen haben Infrastruktur-Code vereinfacht, indem sie die Definition der gewünschten Infrastruktur davon getrennt haben, wie das zu implementieren ist.

Seit Kurzem gibt es einen Trend zu neuen Infrastruktur-Tools, die bestehende allgemein nutzbare Sprachen zur Definition der Infrastruktur verwenden.[1] Pulumi (*https://www.pulumi.com*) und das AWS CDK (Cloud Development Kit, *https://aws.amazon.com/cdk*) unterstützen Sprachen wie Typescript, Python oder Java. Diese Werkzeuge entstanden, um manche der Einschränkungen deklarativer Sprachen anzugehen.

Deklarativen und imperativen Code mischen

Imperativer Code ist ein Satz von Anweisungen, der festlegt, wie etwas passieren soll. *Deklarativer Code* definiert, was Sie haben wollen, ohne zu spezifizieren, wie das geschehen soll.[2]

Zu viel Infrastruktur-Code leidet heutzutage unter dem Vermischen deklarativen und imperativen Codes. Ich glaube, es ist ein Fehler, darauf zu bestehen, dass nur genau eines der beiden Sprachparadigmen für den gesamten Infrastruktur-Code zum Einsatz kommen sollte.

Eine Infrastruktur-Codebasis kümmert sich um viele verschiedene Aspekte – vom Definieren von Infrastruktur-Ressourcen über das Konfigurieren verschiedener Instanzen von ansonsten gleichen Ressourcen bis hin zum Orchestrieren des Provisionierens mehrerer unabhängiger Elemente eines Systems. Manche dieser Aspekte lassen sich am einfachsten über eine deklarative Sprache ausdrücken. Andere sind komplexer und werden besser über eine imperative Sprache umgesetzt.

Als Praktikerinnen und Praktiker des immer noch jungen Felds des Infrastruktur-Codes versuchen wir weiterhin herauszufinden, wo die Grenzen zwischen diesen Aspekten zu ziehen sind. Vermischt man sie, kann das zu Code führen, der Sprachparadigmen mixt. Eine Fehlermöglichkeit ist, eine deklarative Syntax wie YAML durch Bedingungen und Schleifen zu erweitern. Die zweite Fehlermöglichkeit ist, einfache Konfigurationsdaten (»2GB RAM«) in prozeduralen Code einzubinden und damit das, *was* Sie wollen, mit dem zu vermischen, *wie* es zu implementieren ist.

In mehreren Bereichen dieses Buchs weise ich darauf hin, was meiner Meinung nach die unterschiedlichen Aspekte sind und wo ein bestimmtes Sprachparadigma am besten passt. Aber unser Feld entwickelt sich immer noch weiter. Viele meiner Ratschläge werden nicht mehr gültig oder unvollständig sein. Daher möchte ich Sie dazu ermutigen, über diese Fragen nachzudenken und uns allen dabei zu helfen, herauszufinden, was am besten funktioniert.

Infrastruktur-Scripting

Bevor sich Standardtools für das deklarative Provisionieren von Cloud-Infrastruktur entwickelten, schrieben wir Skripte in allgemeinen, prozeduralen Sprachen. Unsere Skripte nutzen meist ein SDK (Software Development Kit), um mit der API des Cloud-Providers zu interagieren.

In Listing 4-1 kommt Pseudocode zum Einsatz, und das Beispiel ähnelt Skripten, die ich in Ruby mit dem AWS SDK geschrieben habe. Es erstellt einen Server namens `my_application_server` und führt dann das (fiktive) Servermaker-Tool zum Konfigurieren aus.

Listing 4-1: Beispiel für prozeduralen Code, der einen Server erstellt

```
import 'cloud-api-library'

network_segment = CloudApi.find_network_segment('private')

app_server = CloudApi.find_server('my_application_server')
```

```
if(app_server == null) {
  app_server = CloudApi.create_server(
    name: 'my_application_server',
    image: 'base_linux',
    cpu: 2,
    ram: '2GB',
    network: network_segment
  )
  while(app_server.ready == false) {
    wait 5
  }
  if(app_server.ok != true) {
    throw ServerFailedError
  }
  app_server.provision(
    provisioner: servermaker,
    role: tomcat_server
```

```
    )

}
```

Dieses Skript vermischt das, *was* zu erstellen ist, mit dem, *wie* es zu erstellen ist. Es legt Attribute des Servers fest, unter anderem die CPU- und Speicherressourcen, die provisioniert werden sollen, das Betriebssystem-Image, von dem gestartet wird, und welche Rolle angewendet werden soll. Es implementiert auch Logik und prüft, ob der Server namens `my_application_server` schon existiert, um zu verhindern, dass ein Server doppelt angelegt wird. Dann wartet es, bis der Server bereit ist, um die Konfiguration darauf anzuwenden.

Dieser Beispielcode kümmert sich nicht um Änderungen an den Serverattributen. Was passiert, wenn Sie den Speicher erhöhen müssen? Sie könnten das Skript anpassen und bei bestehendem Server jedes Attribut prüfen und bei Bedarf aktualisieren. Oder Sie könnten ein neues Skript schreiben, um bestehende Server zu finden und anzupassen.

Realistischere Szenarien sind solche, bei denen mehrere Server mit unterschiedlichem Typ im Spiel sind. Neben unserem Anwendungsserver hatte mein Team Webserver und Datenbankserver im Einsatz. Auch gab es mehrere Umgebungen, was zu mehreren Instanzen jedes Servers führte.

Teams, mit denen ich zusammengearbeitet habe, verwandelten solche einfachen Skripte wie das obige Beispielskript oft in ein Skript für viele verschiedene Zwecke. Das übernahm dann Argumente, mit denen der Typ des Servers und die Umgebung spezifiziert wurde und die dann dazu dienten, die passende Serverinstanz anzulegen. Das haben wir zu einem Skript umgebaut, das Konfigurationsdateien einliest, in denen die verschiedenen Serverattribute definiert sind.

Ich arbeitete gerade an einem solchen Skript und fragte mich, ob es sich lohnen würde, es als Open-Source-Tool bereitzustellen, als HashiCorp die erste Version von Terraform veröffentlichte.

Deklarative Infrastruktur-Sprachen

Viele Infrastruktur-Codewerkzeuge wie zum Beispiel Ansible, Chef, CloudFormation, Puppet oder Terraform nutzen deklarative Sprachen. Ihr Code definiert Ihren gewünschten Status für Ihre Infrastruktur, wie zum Beispiel, welche Pakete und User Accounts auf einem Server vorhanden sein oder wie viel

RAM- und CPU-Ressourcen bereitgestellt werden sollten. Das Tool kümmert sich um die Logik, wie dieser gewünschte Status erreicht werden kann.

In Listing 4-2 wird der gleiche Server wie in Listing 4-1 erstellt. Der Code in diesem Beispiel (wie bei den meisten Codebeispielen in diesem Buch) ist in einer fiktiven Sprache erstellt.[1]

Listing 4-2: Beispiel für deklarativen Code

```
virtual_machine:

  name: my_application_server

  source_image: 'base_linux'

  cpu: 2

  ram: 2GB

  network: private_network_segment

  provision:

    provisioner: servermaker

    role: tomcat_server
```

Dieser Code enthält keine Logik, mit der geprüft wird, ob der Server schon besteht, oder mit der darauf gewartet wird, dass der Server bereit ist, bevor der Provisioner gestartet wird. Das Tool, das Sie ausführen, um den Code anzuwenden, kümmert sich darum. Es vergleicht auch die aktuellen Infrastruktur-Attribute mit der Deklaration und ermittelt, welche Änderungen vorzunehmen sind, um die Infrastruktur auf den gewünschten Stand zu bringen. Um also in diesem Beispiel den Speicher des Anwendungsservers zu erhöhen, bearbeiten Sie die Datei und lassen das Tool erneut laufen.

Deklarative Infrastruktur-Tools wie Terraform oder Chef trennen das *Was* (Sie wollen) vom *Wie* (es erzeugt wird). Im Ergebnis ist Ihr Code sauberer und

direkter. Manche beschreiben deklarativen Infrastruktur-Code als näher am Konfigurieren denn am Programmieren.

Ist deklarativer Code echter Code?

Manche lehnen deklarative Sprachen ab, weil diese eher Konfiguration als »echter« Code sind.

Ich verwende das Wort »Code«, um mich sowohl auf deklarative als auch auf imperative Sprachen zu beziehen. Muss ich zwischen beidem unterscheiden, sage ich ganz spezifisch entweder »deklarativ« oder »programmierbar« (oder nutze eine Variante davon).

Ich finde die Diskussion darüber, ob eine Coding-Sprache Turingvollständig sein muss, müßig. Ich finde selbst reguläre Ausdrücke für bestimmte Zwecke sinnvoll – und die sind auch nicht Turingvollständig. Daher mag mir eine Ergebenheit für die Reinheit »echter« Programmierung auch einfach fehlen.

Idempotenz

Das kontinuierliche Anwenden von Code ist eine wichtige Praxis für das Pflegen der Konsistenz und Kontrolle Ihres Infrastruktur-Codes, wie in »Code kontinuierlich anwenden« auf Seite 395 beschrieben wird. Dazu gehört ein wiederholtes Wiederanwenden von Code auf die Infrastruktur, um einen Drift zu vermeiden. Code muss *idempotent* sein, um sicher kontinuierlich angewendet werden zu können.

Sie können idempotenten Code beliebig häufig wiederholt laufen lassen, ohne die Ausgabe oder das Ergebnis zu verändern. Lassen Sie ein Tool mehrfach laufen, das nicht idempotent ist, kann das die Dinge ziemlich durcheinanderbringen.

Hier ein Beispiel für ein Shell-Skript, das nicht idempotent ist:

```
echo "spock:*:1010:1010:Spock:/home/spock:/bin/bash" \
    >> /etc/passwd
```

Lassen Sie dieses Skript einmal laufen, erhalten Sie das gewünschte Ergebnis: Der Anwender `spock` wird zur Datei */etc/passwd* hinzugefügt. Lassen Sie es zehnmal laufen, erhalten Sie zehn identische Einträge für diesen gleichen Anwender.

Mit einem idempotenten Infrastruktur-Tool geben Sie an, wie das Ergebnis aussehen soll:

```
user:
  name: spock
```

```
  full_name: Spock

  uid: 1010

  gid: 1010

  home: /home/spock

  shell: /bin/bash
```

Egal wie oft Sie das Tool mit diesem Code laufen lassen, wird es sicherstellen, dass nur ein Eintrag für den Anwender `spock` in der Datei */etc/passwd* existiert. Keine unschönen Nebeneffekte.

Programmierbare, imperative Infrastruktur-Sprachen

Deklarativer Code ist super, wenn immer das gleiche Ergebnis gewünscht ist. Aber es gibt Situationen, in denen Sie abhängig von den Umständen etwas Unterschiedliches erreichen wollen. So erzeugt der folgende Code beispielsweise einen Satz an VLANs. Der Cloud-Provider des ShopSpinner-Teams hat in jedem Land unterschiedlich viele Data Center, und das Team möchte, dass der Code ein VLAN in jedem Data Center erzeugt. Also muss er dynamisch ermitteln, wie viele Data Center es gibt, und in jedem ein VLAN anlegen:

Listing 4-3: Beispiel für Infrastruktur-Code in einer imperativen Sprache

```
this_country = getArgument("country")

data_centers = CloudApi.find_data_centers(country:
this_country)

full_ip_range = 10.2.0.0/16

vlan_number = 0

for $DATA_CENTER in data_centers {
```

```
  vlan = CloudApi.vlan.apply(
    name: "public_vlan_${DATA_CENTER.name}"
    data_center: $DATA_CENTER.id
    ip_range: Networking.subrange(
        full_ip_range,
        data_centers.howmany,
        data_centers.howmany++
    )
  )
}
```

Der Code weist zudem jedem VLAN einen IP-Bereich zu, wobei er eine fiktive, aber sehr nützliche Methode namens `Networking.subrange()` verwendet. Diese Methode übernimmt den in `full_ip_range` deklarierten Adressbereich, teilt ihn basierend auf dem Wert von `data_centers.howmany` in eine Reihe kleinerer Adressbereiche auf und gibt einen dieser Adressbereiche zurück (der über die Variable `data_centers.howmany` indexiert ist).

Solche Logik lässt sich nicht durch deklarativen Code ausdrücken, daher erweitern die meisten deklarativen Infrastruktur-Tools ihre Sprachen um imperative Programmiermöglichkeiten. So ergänzt Ansible beispielsweise sein YAML um Schleifen und Bedingungen (*https://oreil.ly/-4wWs*). Die Konfigurationssprache HCL von Terraform wird oft als deklarativ beschrieben, aber eigentlich kombiniert sie drei Untersprachen (*https://oreil.ly/dFgG4*). Eine davon ist Expressions (*https://oreil.ly/qJQrd*), die Bedingungen und Schleifen enthält.

Neuere Tools, wie zum Beispiel Pulumi (*https://www.pulumi.com*) oder das AWS CDK (*https://aws.amazon.com/cdk*), wenden sich wieder programmierbaren Sprachen für die Infrastruktur zu. Ein großer Vorteil ist die Unterstützung von allgemein nutzbaren Programmiersprachen (siehe »Allgemein nutzbare Sprachen und DSLs für die Infrastruktur« auf Seite 76). Aber sie sind auch nützlich, um dynamischeren Infrastruktur-Code zu implementieren.

Statt also entweder deklarative oder imperative Infrastruktur-Sprachen als das einzig korrekte Paradigma anzusehen, sollten wir uns anschauen, für welche Aspekte die jeweilige Sprache am besten passt.

Deklarative und imperative Sprachen für Infrastruktur

Deklarativer Code ist nützlich, um den gewünschten Status eines Systems zu definieren – insbesondere, wenn es in den erwarteten Ergebnissen keine großen Variationen gibt. Normalerweise definieren Sie die Form der Infrastruktur, die Sie mit hoher Konsistenz wiederholen wollen.

So wollen Sie zum Beispiel normalerweise, dass alle Umgebungen, die einen Release-Prozess unterstützen, nahezu identisch sind (siehe »Auslieferungsumgebungen« auf Seite 96). Deklarativer Code ist daher gut für das Definieren wiederverwendbarer Umgebungen (oder Teilen davon, siehe das Reusable-Stack-Pattern in »Pattern: Reusable Stack« auf Seite 102). Sie können sogar in eingeschränktem Rahmen Variationen zwischen Infrastruktur-Instanzen umsetzen, die durch deklarativen Code definiert wurden, indem Sie Instanz-Konfigurationsparameter einsetzen (siehe Kapitel 7).

Aber manchmal wollen Sie wiederverwertbaren, gemeinsam nutzbaren Code schreiben, der abhängig von der Situation unterschiedliche Ergebnisse liefern kann. So schreibt das ShopSpinner-Team beispielsweise Code, der Infrastruktur für unterschiedliche Anwendungsserver erstellen kann. Manche dieser Server sind von außen zugänglich, daher benötigen sie passende Gateways, Firewall-Regeln, Routen und Logging. Andere Server sind nur intern erreichbar, daher gibt es für sie andere Verbindungs- und Sicherheitsanforderungen. Die Infrastruktur kann sich auch für Anwendungen unterscheiden, die Messaging, Data Storage und andere optionale Elemente nutzen.

Durch seine Unterstützung zunehmend komplexer Variationen enthält deklarativer Code auch immer mehr Logik. Irgendwann sollten Sie sich fragen, warum Sie komplexe Logik in YAML, JSON, XML oder einer anderen deklarativen Sprache schreiben.

Daher sind programmierbare, imperative Sprachen passender für das Erstellen von Bibliotheken und abstrakten Schichten, wie wir genauer in Kapitel 16

besprechen werden. Und diese Sprachen tendieren dazu, das Schreiben, Testen und Managen von Bibliotheken besser zu unterstützen.

Domänenspezifische Infrastruktur-Sprachen

Abgesehen von ihrem deklarativen Ansatz nutzen viele Infrastruktur-Tools ihre eigene DSL (Domain-Specific Language).[1]

Eine DSL ist eine Sprache, die dazu entworfen wurde, eine bestimmte Domäne zu modellieren – in unserem Fall Infrastruktur. Damit wird es einfacher, Code zu schreiben und ihn zu verstehen, weil sie die von Ihnen zu definierenden Elemente besser abbilden kann.

So besitzen beispielsweise Ansible, Chef und Puppet jeweils eine DSL zum Konfigurieren von Servern. Ihre Sprachen stellen Konstrukte für Konzepte wie Pakete, Dateien, Services und User Accounts bereit. Ein Pseudocode-Beispiel für eine DSL zur Serverkonfiguration sieht so aus:

```
package: jdk

package: tomcat

service: tomcat

  port: 8443

  user: tomcat

  group: tomcat

file: /var/lib/tomcat/server.conf

  owner: tomcat

  group: tomcat

  mode: 0644
```

```
  contents:
  $TEMPLATE(/src/appserver/tomcat/server.conf.template)
```

Dieser Code stellt sicher, dass die zwei Softwarepakete `jsk` und `tomcat` installiert werden. Er definiert einen Service, der laufen soll, legt den Port fest, an dem er lauschen soll, und bestimmt den Account und die Gruppe, unter der er aktiv ist. Schließlich definiert der Code, dass eine Serverkonfigurationsdatei aus einer Template-Datei erzeugt werden soll.

Der Beispielcode lässt sich recht einfach von jemandem verstehen, der Systemadministrationswissen besitzt, auch wenn das spezifische Tool oder genau diese Sprache ihr oder ihm nicht vertraut sind. In Kapitel 11 geht es darum, wie Sprachen zur Serverkonfiguration einzusetzen sind.

Viele Stack-Management-Tools nutzen ebenfalls DSLs, unter anderem auch Terraform und CloudFoundation. Sie bieten auf diesem Weg Konzepte aus ihrer eigenen Domäne (Infrastruktur-Plattformen) an, sodass Sie direkt Code schreiben können, der sich auf virtuelle Maschinen, Disk Volumes und Netzwerk-Routen bezieht. In Kapitel 5 finden Sie mehr Informationen über den Einsatz dieser Sprachen und Tools.

Andere Infrastruktur-DSLs modellieren Konzepte für Anwendungs-Laufzeitplattformen – zum Beispiel Anwendungs-Cluster, Service Meshes oder Anwendungen. Beispiele dafür sind Helm-Charts (*https://oreil.ly/F14uU*) und CloudFoundry App-Manifeste (*https://oreil.ly/SBpsV*).

Viele Infrastruktur-DSLs sind als Erweiterungen bestehender Markup-Sprachen wie YAML (Ansible, CloudFoundation, alles rund um Kubernetes) oder JSON (Packer, CloudFoundation) angelegt. Bei manchen handelt es sich um interne DSLs, die als Untermenge (oder Übermenge) einer allgemein nutzbaren Programmiersprache geschrieben sind. Chef ist ein Beispiel für eine interne DSL, die als Ruby-Code geschrieben wird. Andere sind externe DSLs, die durch Code in einer anderen Sprache interpretiert werden. Terraform HCL ist eine externe DSL – der Code hat nichts mit der Sprache Go zu tun, in der der Interpreter geschrieben ist.

Allgemein nutzbare Sprachen und DSLs für die Infrastruktur

Die meisten Infrastruktur-DSLs sind eher deklarative als imperative Sprachen. Eine interne DSL wie Chef ist eine Ausnahme, auch wenn Chef vor allem deklarativ ist.[1]

Einer der größten Vorteile des Einsatzes von allgemein nutzbaren Sprachen wie Java-Script, Python, Ruby oder TypeScript ist das Ökosystem an Werkzeugen.

Diese Sprachen werden durch IDEs[2] sehr gut unterstützt, die leistungsfähige Produktivitätsfeatures wie Syntax Highlighting oder Code Refactoring bieten. Eine Testunterstützung ist ein besonders nützlicher Teil des Ökosystems einer Programmiersprache.

Es gibt viele Tools zum Testen der Infrastruktur – manche davon sind in »Verifikation: Aussagen über Infrastruktur-Ressourcen treffen« auf Seite 169 und »Server-Code testen« auf Seite 213 aufgeführt. Aber nur wenige von ihnen sind in Sprachen integriert, die Unit Testing unterstützen. Wie wir in »Herausforderung: Tests für deklarativen Code haben häufig nur einen geringen Wert« auf Seite 143 besprechen werden, ist das für deklarativen Code nicht unbedingt ein Problem. Aber für Code, der variablere Ergebnisse liefert, wie zum Beispiel Bibliotheken oder Abstraktionsschichten, sind Unit Tests essenziell.

Implementierungs-Prinzipien beim Definieren von Infrastructure as Code

Um Ihre Infrastruktur-Systeme einfach und sicher aktualisieren und weiterentwickeln zu können, müssen Sie Ihre Codebasis sauber halten: Sie muss leicht zu verstehen sein, gut testbar und wartbar sein und sich einfach verbessern lassen. Codequalität ist ein vertrautes Thema in der Softwareentwicklung. Die folgenden Implementierungs-Prinzipien sind Richtlinien für das Designen und Organisieren Ihres Codes, um dieses Ziel zu unterstützen.

Halten Sie deklarativen und imperativen Code voneinander getrennt

Code, der sowohl deklarative wie auch imperative Anteile enthält, hat einen Design Smell, und Sie sollten ihn in seine jeweiligen Zuständigkeiten aufteilen.[1]

Behandeln Sie Infrastruktur-Code wie echten Code

Viele Infrastruktur-Codebasen entwickeln sich aus Konfigurationsdateien und Hilfsskripten in ein unwartbares Chaos. Allzu häufig betrachten die Leute Infrastruktur-Code nicht als »echten« Code. Sie halten dort nicht die gleiche Entwicklungsdisziplin wie bei Anwendungscode ein. Um eine Infrastruktur-Codebasis wartbar zu halten, müssen Sie sie mit der gleichen Wichtigkeit behandeln.

Designen und managen Sie Ihren Infrastruktur-Code so, dass er sich leicht verstehen und warten lässt. Setzen Sie Praktiken zur Codequalität um, wie zum Beispiel Code Reviews, Pair Programming und automatisiertes Testen. Ihr Team sollte auf technische Schulden achten und versuchen, sie zu minimieren.

In Kapitel 15 wird beschrieben, wie Sie verschiedene Softwaredesign-Prinzipien auf Infrastruktur anwenden, wie zum Beispiel ein Verstärken der Kohäsion und eine Reduktion der Kopplung. In Kapitel 18 wird erklärt, welche Möglichkeiten es zum Organisieren und Managen von Infrastruktur-Codebasen gibt, um einfacher mit ihnen arbeiten zu können.

Code als Dokumentation

Es kann zu viel Arbeit sein, Dokumentation zu schreiben und sie aktuell zu halten. Für manche Zwecke ist der Infrastruktur-Code nützlicher als »richtige« Dokumentation. Es handelt sich bei ihm um eine immer akkurate und aktuelle Beschreibung Ihres Systems:

- Neue Teammitglieder können den Code durchgehen, um etwas über das System zu lernen.
- Teammitglieder können den Code lesen und Commits begutachten, um zu erfahren, was andere getan haben.
- Technische Reviewer können den Code verwenden, um herauszufinden, was sich verbessern lässt.
- Auditorinnen und Auditoren können den Code und die Versionshistorie begutachten, um ein genaues Bild des Systems zu erhalten.

Infrastruktur-Code ist selten die einzig erforderliche Dokumentation. Eine High-Level-Dokumentation ist für den Kontext und eine Strategiebeschreibung nützlich. Vielleicht haben Sie Stakeholder, die Aspekte Ihres Systems verstehen müssen, Ihren Technologie-Stack aber nicht kennen.

Sie können auch diese anderen Dokumentationstypen als Code managen. Viele Teams schreiben Architecture Decision Records (ADRs, *https://oreil.ly/cxP4c*) in einer Markup-Sprache und halten sie unter Versionskontrolle.

Sie können nützliche Unterlagen wie Architekturdiagramme oder Parameter-Referenzen automatisch aus Code generieren. Diesen können Sie in eine Change Management Pipeline stecken, um die Dokumentation immer dann zu aktualisieren, wenn jemand eine Änderung am Code vornimmt.

Zusammenfassung

Dieses Kapitel hat die zentrale Praktik behandelt, Ihr System als Code zu definieren. Dazu gehört, sich anzuschauen, warum Sie etwas als Code definieren sollten und welche Teile Ihres Systems als Code definiert werden können. Im Zentrum dieses Kapitels stand das Begutachten unterschiedlicher Paradigmen

für Infrastruktur-Sprachen. Das mag ein wenig abstrakt wirken. Aber für das Schaffen effektiver Infrastruktur ist es von entscheidender Bedeutung, die richtige Sprache auf die richtige Art und Weise einzusetzen – eine Herausforderung, die die Branche noch nicht gelöst hat. Daher ist die Frage, welche Sprache Sie für die verschiedenen Teile Ihres Systems nutzen und welche Konsequenzen sich daraus ergeben, ein Thema, das in diesem Buch immer wieder auftauchen wird.

Index

1Password (Website) 111

5 Lessons We've Learned Using AWS (Website) 427

A

A Brief and Incomplete History of Build Pipelines (Blogpost) 152

Abgrenzung, Anwendungs-Cluster 281

Abhängigkeiten

- als Skript-Aufgabe 378
- deutlich machen 146
- entkoppeln 339
- isolieren 146
- minimieren 146
- Scope für Stages 155
- Server 225
- Test-Fixtures 172
- verkomplizieren die Test-Infrastruktur 148
- von Definitionscode trennen 339
- zirkuläre 296
- zwischen Komponenten-Stacks 329

abschließendes Testen 139

Abstraktionsschichten 327

Abtrennen, mehrere Produktivumgebungen 97

abwärtskompatible Transformationen 411

Accelerate State of DevOps Report 139

Ad-hoc-Anwendung 395

ADRs (Architecture Decision Records) 78

AKS (Azure Kubernetes Services) 59, 270, 274

Aktionsfähigkeit

von Code 68
Aktivität einer Pipeline-Stage 153
Alles reproduzierbar machen (Prinzip) 44
allgemein nutzbare Registry-Produkte 131
allgemein nutzbare Sprachen 76
Amazon 243
Amazon DynamoDB 59
Amazon Elastic Container Service for Kubernetes (EKS) 59, 270, 274
Amazon Elastic Container Services (ECS) 59, 270, 274
Amazon S3 60
Ambler, Scott
Refactoring Databases 200
Änderungen, Umfang reduzieren 402
Änderungsfehlschläge, Anteil 39
Änderungsmanagement, Anwendungs-Cluster 281
Andrew File System 60
Angstspirale der Automation 49
Animator (Website) 247
Ansible 68, 75, 209
Ansible Cloud Modules (Website) 82
Ansible for OpenShift (Website) 275
Ansible Tower 209
Ansible Tower (Website) 130
Antipattern
Apply on Change 230
Copy-Paste Environments 100
Definition 24
Dual Persistent and Ephemeral Stack Stages 179
für Infrastruktur-Stacks 86
Manual Stack Parameters 110
Monolithic Stack 86
Multiple-Environment Stack 99
Obfuscation Module 318
Spaghetti Module 322
Unshared Module 320

Anwender
  in Team-Workflows 384
Anwendungen
  als Container verpacken 193
  als Komponenten von Infrastruktur-Systemen 53
  auf Anwendungs-Cluster deployen 195
  auf Server deployen 193
  auf Server-Cluster deployen 195
  ausliefern 356
  Connectivity 200
  deploybare Teile 191
  FaaS-Serverless deployen 198
  mit Infrastruktur testen 358
  mit Infrastruktur-Code deployen 359
  Pakete zum Deployen auf Cluster 197
Anwendungs-Cluster 269
  Anwendungen deployen 195
  getrennte für Auslieferungs-Stages 282
  Governance 283
  Lösungen 270
  Stack-Topologien 272
  Strategien 280
  Teams 284
Anwendungsdaten 199
anwendungsgesteuerte Infrastruktur 190
Anwendungshosting-Cluster 59
Anwendungs-Laufzeitumgebung
  Cloud-native Infrastruktur 190
  Ziele 191
Anwendungs-Laufzeitumgebungen
  Anwendungen auf Server deployen 193
  Anwendungs-Connectivity 200
  Anwendungsdaten 199
  anwendungsgesteuerte Infrastruktur 190
  Deployment-Pakete 193

Anwendungspakete 66
Anwendungs-Runtime
- Secrets injizieren 134

Anwendungsserver 163
Apache Mesos (Website) 196, 272
Apache OpenWhisk (Website) 287
API-Gateways 62, 203
APIs
- statische Code-Analyse 167
- testen mit Mocks 167

Application-Group-Stack-Pattern 86, 89
Apply on Change (Antipattern) 230
Apply-Time-Project-Integration-Pattern 375
AppVeyor (Website) 157
Architecture Decision Records (ADRs) 78
Architektonisches Quantum 297
Artefakte 365
Artifactory (Website) 366
ASG (Auto Scaling Group) 58
asynchrones Messaging 63
Atlantis 157, 391
Aufteilen von Daten 430
Aufwärmen von Server-Images 254
Ausbacken einer Server-Instanz 224
Ausfälle einplanen 428
Ausfallmodus 428
Ausführen, als Skript-Aufgabe 379
Ausgabe einer Pipeline-Stage 154
Ausliefern 363
- Durchlaufzeit als Schlüsselmetrik für die Softwareauslieferung und Operational-Performance 39
- Infrastruktur-Code als Artefakte verpacken 365
- Infrastruktur-Code mit Repositories ausliefern 365
- Infrastruktur-Projekte bauen 364
- Konfiguration und Skripten für Services 66
- Pipeline-Software und -Services 156

Projekte integrieren 368
Umgebungen 96
Authentifizierung, Service Mesh 285
automatisiertes Testen in Pipeline-basiertem Workflow 397
Automatisierungs-Verzögerung 394
Auto-Recovery 219
Autorisierung, secretlos 133
Autoscaling 219
AWS CodeBuild (Website) 157
AWS CodePipeline (Website) 157
AWS ECS Services (Website) 197
AWS Elastic BeanStalk 56, 60
AWS Image Builder (Website) 247
AWS Lambda 59, 287
AWS SageMaker 59
AWS Subnets 62
Awspec (Website) 169
Azure App Service Plans (Website) 197
Azure Block Blobs 60
Azure Cosmos DB 59
Azure DevOps 56
Azure Functions 59, 287
Azure Image Builder (Website) 247
Azure Kubernetes Service (AKS) 59, 270, 274
Azure ML Services 59
Azure Page Blobs 60
Azure Pipelines (Website) 157
Azure Resource Manager (Website) 82
Azurite (Website) 167

B

BaaS (Backend as a Service) 287
Backen von Server-Images 225, 235
Bamboo (Website) 156
Bare Metal 58
Basis-Betriebssystem als Quelle für Server 207

batect (Website) 392
Bazel (Website) 371
Beck, Kent 293, 405
Behr, Kevin, The Visible Ops Handbook 35
Benutzer
als Code-Autoren 386
testen 159
Benutzer-Aufteilung 411
Best Practices 25
Bestätigung einer Pipeline-Stage 154
Betriebssysteme, Server-Images 264
BibliothekenStack-Elemente erstellen 315
Blameless Postmortem 399, 419
Block Storage (Virtual Disk Volumes) 60
Blue-Green-Deployment-Pattern 423
Borg 196
Bosh (Website) 82
BoxFuse (Website) 157
Branch by Abstraction 411
Break-Glass-Prozess 235
Buck (Website) 371
Builder
als Code-Autoren 386
in Team-Workflows 385
Builder-Instanzen 249
BuildKite (Website) 157
Build-Server 156
Build-Stage 165
Build-Time-Project-Integration-Pattern 369
Bundle-Module-Pattern 318, 321, 325
Butler-Cole, Ben 234

C

CaaS (Containers as a Service) 58, 406
Cache 63
Campbell, Laine

Database Reliability Engineering 200
Canary-Development-Pattern 411
can_connect (Methode) 175
Capistrano (Website) 195
CD (Continuous Delivery)
Beschreibung 137
Gründe 138
Herausforderungen 143
Infrastruktur-Delivery-Pipelines 152
progressives Testen 148
testen in der Produktivumgebung 158
CDC (Consumer-driven Contract Testing) 378
CDK (Cloud Development Kit) 69, 74, 332, 334
CDN (Content Distribute Network) 63
CD-Software 157
Centos 259
CFAR (Cloud Foundry Application Runtime) (Website) 272
CFEngine 68, 209
cfn_nag (Website) 166
CFRs (Cross-Functional Requirements) 141
Changelog 351
Changemanagement
Patterns 230
Chaos Engineering 160, 243, 427
Chaos Monkey 427
checkov (Website) 166
Chef 23, 68, 75–76, 209, 211
Chef Community Cookbooks (Website) 367
Chef Infra Server (Website) 130
Chef Provisioning (Website) 82
Chef Server 209, 366–367
chroot (Befehl) 252
CI 89
CircleCI (Website) 157
CI-Server 125

Clarity (Website) 169
Clay-Shafer, Andrew 23
Cloud Computing 55
Cloud Development Kit (CDK) *siehe* CDK (Cloud Development Kit)
Cloud Foundry Application Runtime (CFAR) (Website) 272
Cloud Native Computing Foundation 191
Cloud-agnostisch 57
CloudFormation 69, 82–83, 103, 198, 314, 334
CloudFormation Linter (Website) 166
CloudFoundation 76
CloudFoundry App-Manifeste 76
cloud-init (Website) 239
Cloud-native Infrastruktur 190, 200
Cloud-Plattform-Services 157
Cloud-Zeitalter 32
  Alles reproduzierbar machen (Prinzip) 44
  Beschreibung 43
  Prinzipien 43
  Sicherstellen, dass jeder Prozess wiederholt werden kann (Prinzip) 50
  Systeme sind unzuverlässig (Prinzip) 44
  Variationen miminieren (Prinzip) 47
  Wegwerfbare Elemente erstellen (Prinzip) 46
Cluster as a Service 270, 273
Cluster *siehe auch* Anwendungs-Cluster
Cluster *siehe auch* Server-Cluster
CMDB (Configuration Management Database) 127
CNAB (Cloud Native Application Bundle) (Website) 197
Cobbler (Website) 220
Code
  in Pipeline-basiertem Workflow 397
  in Versionsverwaltung managen 67
  kontinuierlich anwenden 395
  Qualität 141
Codebasis 69, 77, 83, 103, 122, 140, 150, 192, 210, 363, 392, 398
Coding-Sprachen

allgemein nutzbare Sprachen 76
Beschreibung 68
deklarative Infrastruktur-Sprachen 71
Domänenspezifische Infrastruktur-Sprachen (DSL) 75
High-Level-Infrastruktur-Sprachen 85
imperative Infrastruktur-Sprachen 73
Low-Level-Infrastruktur-Sprachen 84
programmierbar *siehe* imperative Infrastruktur-Sprachen
prozedurale Sprachen 70, 76
Stack-Komponenten 314
Compliance
testen 142
Computing-Ressourcen 58
ConcourseCI 124, 157
Concurrency
testen 159
Conding-Sprachen
prozedurale Sprachen 145
Configuration Management Database (CMDB) 127
Connectivity 192
Consul (Website) 131, 203, 286
Consumer 172, 296
Consumer-driven Contract Testing (CDC) 378
Container
als Code 270
Anwendungen verpacken 193
Beschreibung 58
Lösung auslagern 409
Containers as a Service (CaaS) 58, 406
Content Distribute Network (CDN) 63
Continuous Delivery (CD) *siehe* CD (Continuous Delivery)
Continuous Delivery (Humble und Farley) 137, 152
Continuous-Configuration-Synchronization-Pattern 232, 235
Continuous-Stack-Reset-Pattern 182
Copy-Paste Environments (Antipattern) 100

CoreOS rkt (Website) 193
Cowboy-IT 32
Cron 209
Cross-Functional Requirements (CFRs) 141
Crowbar (Website) 220

D

Dark Launching 411
Data Center 62, 73, 423
Database Reliability Engineering (Campbell und Majors) 200
Dateien
    Konfigurationswerte 354
    Projektsupport 351
Daten
    managen 160
    testen 159
Daten neu laden 431
Daten sperren 430
Datenbank 164
Datensatz 159, 282
Datenschemata 199
Datenstrukturen 192, 199
DBaaS (Database as a Service) 60
DBDeploy (Website) 199
dbmigrate (Website) 199
DDD (Domain-driven Design) 326
DDNS (Dynamic DNS) 202
deb-Dateien 359, 365
DebianPreseed (Website) 217, 253
Debois, Patrick 23
dedizierte Projekte für Integrationstests 353
deklarative Infrastruktur-Sprachen 71
deklarative Tests 144
deklarativer Code
    Beschreibung 69, 74, 77
    Kombinationen testen 146

variablen Code testen 145
Wiederverwenden mit Modulen 314
Delivery-Time-Project-Integration-Pattern 373
demokratische Qualität
in Pipeline-basiertem Workflow 397
Dependency Injection (DI) 339
Deployment-Häufigkeit
als Schlüsselmetrik für die Softwareauslieferung und Operational-Performance 39
Deployment-Manifest 197
Deployment-Pakete 193
Design Smell 77
Designer
in Team-Workflows 385
DevOps 23, 32, 356
Dezentralisierte Konfiguration 240
DI (Dependency Injection) 339
direkte Verbindung 62
Disaster Recovery 426
DNS (Domain Name System) 202
Do Better As Code (Website) 148
Docker (Website) 193
Dojo (Website) 392
Dokumentation
Code als 78
Domain Name System (DNS) 202
Domain-driven Design (DDD) 326
Domänenkonzepte 295, 354
doozerd (Website) 131
DORA 39
Downstream-Abhängigkeiten 172, 174
Drone (Website) 157
Dropwizard (Website) 359
DRY-Prinzip (Don't Repeat Yourself) 293
DSGVO (Website) 283
DSL (Domain-Specific Language) 75

Dual Persistent and Ephemeral Stack Stages (Antipattern) 179
Dynamic DNS (DDNS) 202

E

ECS (Amazon Elastic Container Services) 59, 270, 274
EDGE
    Value-Driven Digital Transformation (Highsmith, Luu und Robinson) 48
EDGE-Modell 48
Effektives Arbeiten mit Legacy Code (Feathers) 303
Effizenz
    Server 224
Einmal-Passwörter 135
Eisenzeit 32
EKS (Amazon Elastic Container Service for Kubernetes) 59, 270, 274
Emergency-Fix-Prozess 399
entr (Tool) 139
Environment Branches 101
Envoy (Website) 286
Ephemeral-Test-Stack-Pattern 178
Ergebnisse
    testen 171
Erkennung 428
etcd (Website) 131
Evolutionary Database Design (Sadalage) 200
Executables 191
Explosionsradius 88
Extreme Programming (XP) 137

F

FaaS (Function as a Service) 220
    Infrastruktur für Serverless 286
    Serverless Code Runtimes 59
    Serverless-Anwendungen deployen 198
Fabric (Website) 195
Facade-Module-Pattern 317, 319, 322, 325
Facade-Pattern

Facade Module 317
Facebook 347
Fan-In Pipeline-Design 374
Farley, David
Continuous Delivery 137, 152
FCS (Fictional Cloud Service) 25
Feathers, Michael 303
Feature Branching 89
Feature Toggles 413
Fehlerbehebung
Service Mesh 285
Fehlertoleranz
mehrere Produktivumgebungen 96
Fission (Website) 287
FKS (Fictional Kubernetes Service) 25, 408
Flyway (Website) 199
Foreman (Website) 220
Fowler, Martin 75, 150, 176
Patterns for Managing Source Code Branches 392
Framework-Repositories
als Quelle für Server 208
Freeman, Steve
Growing Object-Oriented Software, Guided by Tests 405
FSI (Fictional Server Image) 25
Function as a Service (FaaS) *siehe* FaaS
(Function as a Service)
Funktionalität
testen 141

G

Gateways 62
GCE Persistent Disk 60
gemeinsames Verwenden
Modularisieren von Stacks 313
Geschwindigkeit
gegenüber Qualität priorisieren 38

- Server 224

Gesetz von Conway 306, 345

Gesetz von Demeter 295

get_fixture() (Methode) 175

get_networking_subrange (Funktion) 145

get_vpc (Funktion) 145

Gillard-Moss, Peter 234

git-crypt (Website) 133

GitHub 156–157

GitLab 157

GitOps (Website) 158

GitOps-Methodik 395

Given/When/Then-Format 144

GKE (Google Kubernetes Engine) 59, 270, 274

Glass, Robert 320

GoCD 124, 157

Golden Images 222

Google 39, 347

Google Cloud Deployment Manager (Website) 82

Google Cloud Functions 59, 287

Google Cloud Storage 60

Google Kubernetes Engine (GKE) 59, 270, 274

Google ML Engine 59

Governance

- Anwendungs-Cluster 283
- in Pipeline-basiertem Workflow 396
- Kanäle in Pipeline-basiertem Workflow 398
- Server-Images 266

Governance-Spezialisten

- als Code-Autoren 387
- in Team-Workflows 385

GPG (Website) 111

Grenzen

- harte 203
- Komponenten-Lebenszyklen 304

natürliche Änderungsmuster 303
Netzwerk 311
Organisationsstrukturen 306
Resilienz fördern 307
Sicherheits- und Governance-Aspekte 311
Skalierbarkeit ermöglichen 307
zwischen Komponenten ziehen 303
Growing Object-Oriented Software, Guided by Tests (Freeman und Pryce) 405

H

Hardware-Architekturen
Server-Images 264
hartcodierte IP-Adressen 202
HashiCorp 131
HCL (Konfigurationssprache) 74
Helm (Website) 197, 361
Helm-Charts 76
Heroku 56
High-Level-Infrastruktur-Sprachen 85
Highsmith, Jim
EDGE
Value-Driven Digital Transformation 48
Hirschfeld, Rob 280
Hodgson, Pete 415
Honeycomb 160
horizontales Gruppieren 309
Hostfile-Einträge 202
Hot-Cloning 221
HPE Container Platform (Website) 271
Humble, Jez
Continuous Delivery 137, 152
Hunt, Craig
TCP/IP Netzwerk-Administration 63
hybride Cloud 57

I

IaaS (Infrastructure as a Service) 54
  dynamisch 55
Idempotenz 72
immutable Infrastruktur 395
Immutable-Server-Pattern 232, 234, 372
imperativer Code 69, 74, 77
Infrastructure as Code
  Definition 65
  Geschichte 23
  Implementierungs-Prinzipien beim Definieren 77
  Vorteile 34
  zentrale Praktiken 40
Infrastructure-Domain-Entity-Pattern 322, 325
Infrastruktur
  anwendungsgesteuert 190
  aufteilen in handhabbare Elemente 146
  ausliefern 356
  Automatisierungs-Tools-Registries 130
  Cloud-native 190, 200
  Delivery-Pipelines 152
  für Builder-Instanzen 249
  für FaaS Serverless 286
  immutable 395
  Live-Infrastruktur ändern 415
  modularisieren 297
  Projekte bauen 364
  Ressourcen 57
  sicher verändern 401
  Systemkomponenten 53
  Tools durch Skripte verpacken 378
  vor der Integration testen 359
Infrastruktur-Chirurgie 307, 417
Infrastruktur-Code testen ist langsam 146
Infrastruktur-Scripting 70

Infrastruktur-Stacks 107, 163
Beispiel 163
Beschreibung 81
Definition 66
High-Level-Infrastruktur-Sprachen 85
Lebenszyklus-Pattern für Testinstanzen 176
Low-Level-Infrastruktur-Sprachen 84
Offlinetests 165
Onlinetests 168
Patterns und Antipatterns 86
Preview von Änderungen 169
Quellcode 83
Server konfigurieren 83
Stack-Instanzen 83
Umgebungen mit mehreren Stacks bauen 104
inkrementelle Änderung 405
inotifywait (Tool) 139
Inspec (Website) 169, 215
Integration-Registry-Lookup-Pattern 336
Integrationshäufigkeit 393
Integrations-Pattern 393
Integrationstests 353
Inverses Conway-Manöver 306
IP-Adressen
hartcodiert 202
Istio (Website) 286
iterative Änderungen 405

J

Jacob, Adam 23
JavaScript 76
Jenkins 124, 156
JEOS (Just Enough Operating System) 253
JSON 76

K

Kanies, Luke 23
KeePass (Website) 111
Keeper (Website) 111
Key/Value-Paar 128
Key/Value-Store 60, 108, 131
Kim, Gene
  The Visible Ops Handbook 35
Kitchen-Terraform (Website) 185
Kohäsion 292
Komponenten
  Refaktorieren 175
  Scope in Stages getestet 154
Komposition
  Modularisieren von Stacks 313
Konfiguration
  als Skript-Aufgabe 378
  Anwendungs-Cluster 281
  Werkzeuge mit externalisierter 67
Konfigurationsdateien 118
Konfigurationsdrift 48, 394
Konfigurationsparameter 192
Konfigurations-Pattern
  verbinden 129
Konfigurations-Registries 126, 130, 202
Konfigurations-Registry 132
Konfigurationswerte 354, 379
Konsistenz
  in Pipeline-basiertem Workflow 397
  von Code 40
Kontinuität 423
  Chaos Engineering 427
  durch schnelles Wiederherstellen 425
  durch Verhindern von Fehlern 424
  für Ausfälle planen 428

in sich ändernden Systemen 430
inkrementell verbessern 429
kontinuierliches Disaster Recovery 426
Kopplung 292
Kopplung, enge 185
kops (Website) 271
Korrektur 429
Korrelation
von Code 68
Kubeadm (Website) 271
KubeCan 408
Kubeless (Website) 287
Kubernetes 271, 277
Kubernetes Borg (Website) 272
Kubernetes-Cluster 271
kubespray (Website) 271
kurzlebige Instanzen 147