

Nane KRATZKE

2. Auflage



CLOUD-NATIVE COMPUTING

Software Engineering von
Diensten und Applikationen
für die Cloud

HANSER



Bleiben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine.

Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:

www.hanser-fachbuch.de/newsletter



Nane Kratzke

Cloud-native Computing

Software Engineering von Diensten und Applikationen für die Cloud

2., überarbeitete Auflage

HANSER

Der Autor:

Prof. Dr. rer. nat. Dipl.-Inform. Nane Kratzke

Technische Hochschule Lübeck, Fachbereich Elektrotechnik und Informatik

Alle in diesem Werk enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Werk enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht. Ebenso wenig übernehmen Autoren und Verlag die Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt also auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

Die endgültige Entscheidung über die Eignung der Informationen für die vorgesehene Verwendung in einer bestimmten Anwendung liegt in der alleinigen Verantwortung des Nutzers.

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers (m/w/d) verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – mit Ausnahme der in den §§ 53, 54 URG genannten Sonderfälle –, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2024 Carl Hanser Verlag München, www.hanserfachbuch.de

Lektorat: Sylvia Hasselbach

Copy editing: Sandra Gottmann, Wasserburg

Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München

Umschlagrealisation: Max Kostopoulos

Titelmotiv: Tom West, unter Verwendung von Grafiken von © Max Kostopoulos

Layout: Manuela Treindl, Fürth

Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Göttingen

Printed in Germany

Print-ISBN: 978-3-446-47914-2

E-Book-ISBN: 978-3-446-47925-8

epub-ISBN: 978-3-446-48029-2

Inhalt

| | |
|---|-------------|
| Vorwort | XIII |
| 1 Einleitung | 1 |
| 1.1 An wen sich dieses Buch richtet | 2 |
| 1.2 Was dieses Buch behandelt | 3 |
| 1.3 Sprachliche Konventionen | 5 |
| 1.4 Notationskonventionen | 6 |
| 1.5 Ergänzende Materialien | 7 |
| Teil I: Grundlagen | 9 |
| 2 Cloud Computing | 11 |
| 2.1 Service-Modelle | 12 |
| 2.1.1 Infrastructure as a Service (IaaS) | 15 |
| 2.1.2 Platform as a Service (PaaS) | 15 |
| 2.1.3 Software as a Service (SaaS) | 16 |
| 2.2 Cloud-Ökonomie | 19 |
| 2.2.1 Eignung von unterschiedlichen Arten von Workloads | 19 |
| 2.2.2 Effekt von Zuteilungsdauer und Ressourcengröße | 22 |
| 2.3 Entwicklung der letzten Jahre | 24 |
| 3 DevOps | 27 |
| 3.1 Prinzipien des Flow | 29 |
| 3.1.1 Prinzip 1: Arbeit sichtbar machen | 29 |
| 3.1.2 Prinzip 2: Work in Progress beschränken | 30 |
| 3.1.3 Prinzip 3: Flaschenhälse minimieren | 31 |
| 3.2 Prinzipien des Feedbacks | 32 |
| 3.2.1 Prinzip 4: Probleme früh erkennen | 32 |
| 3.2.2 Prinzip 5: Probleme sofort lösen | 32 |
| 3.2.3 Prinzip 6: Probleme professionell verantworten | 33 |
| 3.3 DevOps-geeignete Architekturen | 33 |
| 3.3.1 Randbedingungen für die Entwicklung | 34 |

| | | |
|--|--|-----------|
| 3.3.2 | Nutzung von Orchestrierungsplattformen | 34 |
| 3.3.3 | Randbedingungen im Betrieb | 35 |
| 4 | Cloud-native | 37 |
| 4.1 | Definitionen in Industrie und Forschung | 39 |
| 4.2 | Die Cloud-native-Definition dieses Buchs | 40 |
| 4.3 | Zusammenfassung und Ausblick auf Teil II bis IV | 41 |
| Teil II: Everything as Code | | 45 |
| 5 | Einleitung zu Teil II | 47 |
| 6 | Deployment-Pipelines | 49 |
| 6.1 | Deployment-Pipelines as Code | 50 |
| 6.1.1 | Phasen-Pipelines | 51 |
| 6.1.2 | Gerichtete Pipelines | 52 |
| 6.1.3 | Hierarchische Pipelines | 53 |
| 6.1.4 | Steuerung von Pipelines | 54 |
| 6.2 | DevOps-geeignete Branching-Strategien | 56 |
| 6.2.1 | Git-Flow | 57 |
| 6.2.2 | GitHub-Flow | 58 |
| 6.2.3 | Trunk-basierte Entwicklung | 59 |
| 6.3 | Zusammenfassung | 60 |
| 7 | Infrastructure as Code | 63 |
| 7.1 | Virtualisierung | 65 |
| 7.1.1 | Virtualisierung von Hardware-Infrastruktur | 65 |
| 7.1.2 | Virtualisierung von Software-Infrastruktur | 66 |
| 7.2 | Provisionierung | 68 |
| 7.2.1 | Immutable Infrastructure | 68 |
| 7.2.2 | IaC-Ansätze | 69 |
| 7.2.3 | Provisionierung von lokalen Umgebungen | 72 |
| 7.2.4 | Provisionierung von Multi-Host-Umgebungen | 74 |
| 7.3 | Zusammenfassung | 77 |
| 8 | Standardisierung von Deployment Units (Container) | 79 |
| 8.1 | Hintergrund (PaaS) | 79 |
| 8.2 | Betriebssystem-Virtualisierung | 82 |
| 8.3 | Container Runtime Environments | 83 |
| 8.3.1 | Kernel-Namespaces | 84 |
| 8.3.2 | Process Capabilities | 85 |
| 8.3.3 | Control Groups | 86 |

| | | |
|-------|--|----|
| 8.3.4 | Union Filesystem | 86 |
| 8.3.5 | High-Level- und Low-Level-Container-Laufzeitumgebungen | 87 |
| 8.4 | Bau und Bereitstellung von Container-Images. | 88 |
| 8.5 | Faktoren gut betreibbarer Container | 90 |
| 8.5.1 | Codebase | 91 |
| 8.5.2 | Abhängigkeiten und Konfigurationen | 91 |
| 8.5.3 | Unterstützende Services und Port Binding | 92 |
| 8.5.4 | Build-, Release- und Run-Phase. | 93 |
| 8.5.5 | Horizontale Skalierung über Prozesse. | 94 |
| 8.5.6 | Umgebungen, Logs und Betrieb | 95 |
| 8.6 | Zusammenfassung | 96 |

9 Container-Plattformen **99**

| | | |
|---------|---|-----|
| 9.1 | Scheduling | 100 |
| 9.1.1 | Heterogenität von Workloads | 101 |
| 9.1.2 | Scheduling-Algorithmen | 102 |
| 9.1.2.1 | Einfache Scheduling-Algorithmen. | 102 |
| 9.1.2.2 | Multidimensionale Scheduling-Algorithmen | 103 |
| 9.1.2.3 | Kapazitätsbasierte Scheduling-Algorithmen | 103 |
| 9.1.3 | Scheduling-Architekturen | 104 |
| 9.1.3.1 | Monolithischer Scheduler | 105 |
| 9.1.3.2 | 2-Level-Scheduler | 105 |
| 9.1.3.3 | Shared-State Scheduler. | 106 |
| 9.2 | Orchestrierung | 107 |
| 9.2.1 | Definition von Betriebszuständen. | 107 |
| 9.2.2 | Regelkreis: Desired versus Current State. | 108 |
| 9.3 | Inside Kubernetes | 109 |
| 9.3.1 | Kubernetes-Architektur | 110 |
| 9.3.2 | Verwaltete Ressourcen und Basis-Blueprint | 112 |
| 9.3.3 | Schedulbare Workloads | 114 |
| 9.3.3.1 | Deployments | 114 |
| 9.3.3.2 | (Cron-)Jobs | 116 |
| 9.3.3.3 | Daemon-Sets | 117 |
| 9.3.3.4 | Stateful-Sets | 118 |
| 9.3.4 | Scheduling Constraints | 121 |
| 9.3.4.1 | Angabe des Ressourcenbedarfs mittels Requests und Limits | 121 |
| 9.3.4.2 | Knoten-Selektoren. | 122 |
| 9.3.4.3 | Knotenaffinitäten | 123 |
| 9.3.4.4 | Pod-(Anti-)Affinitäten | 124 |
| 9.3.5 | Automatische Skalierung von Workloads | 125 |
| 9.3.6 | Exponieren von Workloads als interne und externe Services | 126 |
| 9.3.7 | Health Checking | 129 |
| 9.3.8 | Persistenz | 132 |

| | | |
|---------|--|-----|
| 9.3.9 | Isolation von Workloads | 133 |
| 9.3.9.1 | Namespaces und Role-based Access Model (Multi-Tenancy) | 133 |
| 9.3.9.2 | Quotas und Limit Ranges | 134 |
| 9.3.9.3 | Network Policies | 135 |
| 9.4 | Zusammenfassung | 137 |

10 Function as a Service 141

| | | |
|--------|---|-----|
| 10.1 | FaaS-Plattformen | 143 |
| 10.1.1 | Das FaaS-Programmiermodell | 145 |
| 10.1.2 | Zu berücksichtigende Randbedingungen | 146 |
| 10.1.3 | Veranschaulichung des FaaS-Programmiermodells | 147 |
| 10.2 | Plattformagnostische FaaS-Frameworks | 149 |
| 10.3 | Ereignisbasierte Autoskalierung | 152 |
| 10.4 | Zusammenfassung | 155 |

Teil III: Cloud-native Architekturen 157

11 Einleitung zu Teil III 159

12 Microservice und Serverless-Architekturen 161

| | | |
|----------|--|-----|
| 12.1 | Eigenschaften von Microservices | 162 |
| 12.2 | Integrationsmuster für Microservices | 166 |
| 12.2.1 | Datenbankbasierte Integration | 167 |
| 12.2.2 | (g)RPC-basierte Interprozesskommunikation | 167 |
| 12.2.3 | Representational State Transfer (REST) | 170 |
| 12.2.4 | Ereignisbasierte Integration (asynchron) | 173 |
| 12.2.5 | API-Versioning | 175 |
| 12.3 | Architekturelle Sicherheit | 178 |
| 12.3.1 | Circuit-Breaker | 178 |
| 12.3.2 | Bulkhead | 179 |
| 12.3.3 | Idempotente API-Operationen | 180 |
| 12.4 | Skalierung von Microservices | 180 |
| 12.4.1 | Load Balancing | 181 |
| 12.4.2 | Messaging | 181 |
| 12.4.3 | Skalierung zustandsbehafteter Komponenten | 183 |
| 12.4.3.1 | Scaling for Reads | 184 |
| 12.4.3.2 | Scaling for Writes (Sharding) | 184 |
| 12.4.3.3 | Command Query Responsibility Segregation (CQRS) | 185 |
| 12.4.4 | Caching | 186 |
| 12.5 | Prinzipien zur Entwicklung von Microservices | 187 |
| 12.5.1 | Prinzip 1: Bilde Modelle um Geschäftskonzepte | 187 |
| 12.5.2 | Prinzip 2: Erschaffe eine Kultur der Automatisierung | 187 |

| | | |
|-----------|---|------------|
| 12.5.3 | Prinzip 3: Blende interne Implementierungsdetails aus | 188 |
| 12.5.4 | Prinzip 4: Dezentralisiere | 188 |
| 12.5.5 | Prinzip 5: Definiere unabhängig aktualisierbare Einheiten | 188 |
| 12.5.6 | Prinzip 6: Isoliere Fehler | 189 |
| 12.5.7 | Prinzip 7: Baue gut beobachtbare Services | 189 |
| 12.6 | Serverless-Architekturen | 190 |
| 12.6.1 | Architekturelle Konsequenzen von Serverless-Limitierungen | 191 |
| 12.6.2 | Das API-Gateway-Pattern | 193 |
| 12.6.3 | Abgrenzung zu Microservices | 195 |
| 12.7 | Zusammenfassung | 196 |
| 13 | Beobachtbare Architekturen | 199 |
| 13.1 | Konsolidierung von Telemetriedaten | 200 |
| 13.2 | Instrumentierung von Systemen | 202 |
| 13.2.1 | Logging | 202 |
| 13.2.2 | Monitoring | 204 |
| 13.2.2.1 | Metrikarten | 206 |
| 13.2.2.2 | Empfehlungen für die Metrikinstrumentierung | 207 |
| 13.2.3 | Tracing | 207 |
| 13.2.3.1 | Empfehlungen für die Instrumentierung | 209 |
| 13.2.3.2 | Tracing-Instrumentierung und Erzeugung von Spans | 212 |
| 13.2.3.3 | Serverseitiges Tracing und Extraktion von Span-Kontexten | 213 |
| 13.2.3.4 | Clientseitiges Tracing und Weiterreichen von Span-Kontexten | 214 |
| 13.3 | Automatisierte Instrumentierung | 215 |
| 13.3.1 | Eigenschaften von Service-Meshs | 216 |
| 13.3.2 | Traffic-Management | 218 |
| 13.3.3 | Resilienz | 221 |
| 13.3.4 | Sicherheit | 223 |
| 13.3.5 | Management und Analyse von Verkehrstopologien | 226 |
| 13.4 | Zusammenfassung | 227 |
| 14 | Domain-driven Design | 229 |
| 14.1 | Fachlichkeit | 230 |
| 14.2 | Strategisches Design | 232 |
| 14.2.1 | Subdomänen | 233 |
| 14.2.1.1 | Kerndomäne (Core Subdomain) | 233 |
| 14.2.1.2 | Unterstützende Subdomäne (Supporting Subdomain) | 234 |
| 14.2.1.3 | Generische Subdomänen (Generic Subdomain) | 234 |
| 14.2.1.4 | Anmerkungen am Beispiel einer Fallstudie | 234 |
| 14.2.2 | Ubiquitous Language | 236 |
| 14.2.2.1 | Eine gemeinsame Sprache als Schlüssel zu einem gemeinsamen Verständnis | 237 |
| 14.2.2.2 | Mehrdeutige und synonime Begriffe | 238 |

| | | |
|----------|---|-----|
| 14.2.3 | Bounded Contexts | 239 |
| 14.2.4 | Context Mapping | 241 |
| 14.2.4.1 | Partnerschaftliche Kooperationsmuster (Partners und Shared-Kernel) | 241 |
| 14.2.4.2 | Customer-Supplier-Kooperation | 243 |
| 14.2.4.3 | Separate Ways | 244 |
| 14.2.4.4 | Context Maps als Landkarte von Machtverhältnissen | 245 |
| 14.3 | Taktisches Design | 246 |
| 14.3.1 | Oft genutzte Pattern für Geschäftslogik | 246 |
| 14.3.1.1 | Das ETL-Pattern (primär Supporting Subdomains) | 246 |
| 14.3.1.2 | Das Active Record-Pattern (primär Supporting Subdomains) | 247 |
| 14.3.1.3 | Das Domain Model-Pattern (primär Core Subdomains) | 248 |
| 14.3.1.4 | Das Event-Sourcing-Pattern (primär Core Subdomains) | 250 |
| 14.3.2 | Oft genutzte Pattern für die Architektur | 251 |
| 14.3.2.1 | Die Ebenen-Architektur | 252 |
| 14.3.2.2 | Das Ports & Adapter-Pattern | 253 |
| 14.3.2.3 | Das CQRS-Pattern | 253 |
| 14.4 | Zusammenfassung | 256 |

Teil IV: Sichere Cloud-native Anwendungen **259**

| | | |
|----|---------------------------------|------------|
| 15 | Einleitung zu Teil IV | 261 |
|----|---------------------------------|------------|

16 Härtung Cloud-nativer Anwendungen **263**

| | | |
|----------|---|-----|
| 16.1 | Härtung (virtueller) Infrastrukturen | 267 |
| 16.1.1 | Tool-gestütztes System Hardening | 268 |
| 16.1.2 | Kontinuierliche Aktualisierung von virtuellen Maschinen | 270 |
| 16.1.3 | Sichere Authentifizierung mittels SSH | 271 |
| 16.1.4 | Kontinuierliche Überwachung virtueller Maschinen | 273 |
| 16.1.4.1 | Verhaltensbasierte Intrusion Detection mittels Auditing | 273 |
| 16.1.4.2 | Signaturbasierte Intrusion Detection | 274 |
| 16.1.4.3 | Log Forwarding | 276 |
| 16.1.5 | Einsatz von Sicherheitsgruppen und Firewalls | 277 |
| 16.2 | Härtung containerisierter Workloads | 279 |
| 16.2.1 | Absicherung von Public Endpoints mittels Ingresses | 280 |
| 16.2.2 | Namespace-basierte Netzwerkséparation | 286 |
| 16.2.3 | Pod Hardening | 289 |
| 16.2.4 | Erhöhung der Container Runtime Isolation | 299 |
| 16.2.5 | Volume Hardening | 300 |
| 16.2.6 | Workload Policing | 303 |
| 16.2.7 | Intrusion Detection | 307 |
| 16.2.8 | Sicherung der Supply Chain | 310 |

| | |
|---|------------|
| 16.2.8.1 Statische Software Composition Analysis (SCA) | 311 |
| 16.2.8.2 Static Application Security Testing (SAST) | 314 |
| 16.2.8.3 Kontinuierliches Schwachstellen-Scanning von Container- Plattformen | 316 |
| 16.3 Zusammenfassung | 319 |
| 17 Regulatorische Anforderungen..... | 321 |
| 17.1 Cloud Compliance und Zertifizierungen | 322 |
| 17.1.1 ISO 9001 | 324 |
| 17.1.2 BSI-IT-Grundschutz und BSI-Standards | 325 |
| 17.1.3 BSI-C5-Zertifizierung..... | 325 |
| 17.1.4 ISO/IEC 27001 und 27017/27018 | 326 |
| 17.1.5 CSA STAR | 327 |
| 17.1.6 CISPE Code of Conduct | 328 |
| 17.1.7 EU Cloud Code of Conduct..... | 329 |
| 17.1.8 SOC 1-3 (Service Organization Control)..... | 330 |
| 17.1.9 FedRAMP..... | 331 |
| 17.1.10 HIPAA..... | 331 |
| 17.1.11 PCI DSS | 332 |
| 17.1.12 Zusammenfassung | 333 |
| 17.2 Aus der DSGVO sich ergebende Anforderungen | 335 |
| 17.2.1 Personenbezogene Daten..... | 335 |
| 17.2.2 Grundsätze der Verarbeitung personenbezogener Daten..... | 336 |
| 17.2.3 Auftragsverarbeitung..... | 338 |
| 17.2.4 Datenschutz-Folgenabschätzungen..... | 340 |
| 17.2.5 Internationale Datentransfers in Drittländer..... | 341 |
| 17.3 Europäischer Datenschutz und Drittländer | 343 |
| 17.3.1 Probleme am Beispiel des CLOUD Act | 344 |
| 17.3.2 Lösungen trotz SCHREMS I + II | 345 |
| 17.4 Zusammenfassung | 346 |
| 18 Schlussbemerkungen | 349 |
| Literaturverzeichnis..... | 359 |
| Stichwortverzeichnis..... | 365 |

Vorwort

Dieses Buch basiert auf zwei Vorlesungen, „*Cloud-native Programmierung*“ und „*Cloud-native Architekturen*“, die ich an der Technischen Hochschule Lübeck gebe. Während der Recherchen für diese beiden Hochschulmodule war ich natürlich auch auf der Suche nach geeigneter Literatur. Das Resultat war ein Literaturumfang, der – auf einem Schreibtisch gestapelt – leider mehr als einen halben Meter Höhe eingenommen hätte.

Meine Recherche mag unzureichend oder meine Anforderungen zu spezifisch gewesen seien, aber ich fand leider nicht die eine oder zwei geeigneten Quellen, die man jemandem als Lehrbuch zum Thema Cloud-native Computing hätte empfehlen und an die Hand geben können; nur eben diesen *Bücherstapel*. Diese Literaturliste hätte mir aber vermutlich diverse kritische Blicke meiner Studentinnen und Studenten eingebracht. Auch wenn ich grundsätzlich kein Freund des Prinzips „*Setze dich zwischen zweier Bücher Mitte und schreib das Dritte*“ bin, war genau dies in diesem Fall der Anstoß zum Schreiben eines ersten Skripts, aus dem letztlich dieses Buch für die beiden oben genannten Lehrveranstaltungen entstanden ist.

Dieses Buch hat somit auch einen gewissen Handbuch-Charakter, auch wenn es kein Handbuch im klassischen Sinne ist. Es kann dennoch bis zu einem gewissen Grad als Nachschlagewerk genutzt werden, da es eine Vielzahl an hervorragender – aber eben leider isolierter – Literatur zum Thema Cloud-native Computing zusammenfasst.

Ich möchte mich an dieser Stelle u. a. bei Dr. Josef Adersberger von der QAware GmbH bedanken, der eine ähnliche Publikationsidee hatte, dann aber letztlich keine Zeit fand, sein Projekt auch umzusetzen, und der mich daraufhin mit dem Hanser Verlag in Kontakt brachte, um es an seiner Stelle zu versuchen. Zu danken ist auch seinen Mitarbeitern. Deren auf GitHub bereitgestellte Vorlesungsunterlagen „*Cloud Computing*“ (Adersberger u. a. 2018) waren insbesondere für den Teil II dieses Buchs wertvolle Inspiration und Gliederungshilfe. Dank gebührt daher auch dem Hanser Verlag und hier vor allem Sylvia Hasselbach, die sich auf diese Kontaktvermittlung und das damit einhergehende Wagnis denn auch eingelassen hat und insbesondere in der Produktionsphase viel Unterstützung geleistet hat.

Besonderer Dank gebührt auch meinen Studierenden, die die undankbare Betatester-Rolle für die praktischen Anteile (Labs) dieses Buchs übernommen haben und mir während der – aufgrund Corona leider nur online stattfindenden – Vorlesungen und Praktika dennoch mit vielen wertvollen Rückmeldungen geholfen haben, die Struktur und den Inhalt des Manuskripts für die anvisierte Zielgruppe zu optimieren. Dabei sind insbesondere Jannik Kühnemundt, Felix Lohse, Lucian Schultz und Jana Schwieger zu nennen, die mehrere vertiefende Labs entwickelt und für Folgejahrgänge zur Verfügung gestellt haben.

Nane Kratzke

1

Einleitung

In Zeiten des digitalen Wandels ist Cloud Computing heute mehr und mehr die primäre Option und nicht mehr nur eine von vielen technischen Möglichkeiten. Insbesondere Start-ups wählen kaum noch den Weg über den Aufbau „klassischer“ On-Premise-Lösungen (Rechenzentren). Insbesondere in frühen Phasen eines Unternehmens wird der Aufbau eigener Serverkapazitäten als zu kapital- und personalintensiv empfunden.

Der weltweite Markt für Public Cloud Services wächst folgerichtig Jahr um Jahr; 2019 beispielsweise um etwa 17 Prozent auf insgesamt mehr als 214 Milliarden Dollar. Das am schnellsten wachsende Marktsegment sind hierbei Infrastruktur-Dienste (IaaS) und Plattform-Dienste (PaaS). 70 % dieses Marktes teilen sich dabei die „sogenannten“ Big Five. Dies waren 2020 Amazon, Microsoft, Alibaba, Google und IBM, also alles nichteuropäische Anbieter. Diese sogenannten Hyperscaler unterliegen rechtlichen Regularien ihrer Heimatländer. Amazon, Microsoft und Google unterliegen beispielsweise dem US CLOUD Act. Der CLOUD Act verpflichtet Provider, US-Behörden Zugriff auf gespeicherte Daten zu gewähren, auch wenn die Speicherung nicht in den USA erfolgt. Alibaba unterliegt den Regularien der Volksrepublik China mit vergleichbaren staatlichen Regularien. Solche Regularien decken sich nicht notwendig mit europäischen oder nationalen Datenschutz-Auffassungen und -Interessen.

Auf europäische Initiative versucht man daher seit 2020, mit GAIA-X eine wettbewerbsfähige, sichere und vertrauenswürdige Dateninfrastruktur der „nächsten Generation“ für Europa aufzubauen, die eine „Datensouveränität“ gewährleisten soll. Dabei sollen insbesondere branchenübergreifende Kooperationen unterstützt werden, um faire und transparente Geschäftsmodelle zu fördern. Flankiert wird dies durch Regeln und Standards für kooperative und rechtskonforme Nutzung von Daten. Solche gemeinsamen Modelle und Regeln sollen die Komplexität und die Kosten der Kommerzialisierung von Daten reduzieren und deren Rechtskonformität erhöhen. Der europäische Ansatz ist also – im Vergleich zum aktuell von wenigen großen US-Hyperscalern dominierten Cloud-Computing-Markt – ein deutlich dezentralerer und kooperativerer Ansatz. Ob dieser Ansatz zu einer umfangreicheren Standardisierung und besseren Interoperabilität von Cloud-Diensten führt, muss die Zukunft allerdings erst noch zeigen. Es wird schwierig werden, in einer sehr agilen, bottom-up geprägten und lösungsorientierten Industrie durchaus fehlende und berechtigte Top-down-Standardisierungs-, Interoperabilitäts- und Datenschutzstrategien zu verankern.

Man kann daher dem Cloud-Computing-Hype durchaus kritisch gegenüberstehen und sich Fragen stellen; zum Beispiel: Ist der GAIA-X Ansatz erfolgversprechend? Wie kritisch ist die Dominanz von US-Hyperscalern? Sollte man die Datenverarbeitung Firmen anvertrauen, die Regularien von nicht demokratisch geprägten Staaten wie der Volksrepublik China unterworfen sind? Solche und ähnliche Fragen sind durchaus berechtigt.

Cloud Computing bleibt aber dennoch heute für viele Unternehmen, Organisationen, Behörden und Forschungseinrichtungen aus rein pragmatischen Gründen die primäre Option, digitalisierte Lösungen (schnell) realisieren zu können. Die Pandora ist nun einmal aus der Büchse. Cloud Computing wird nicht einfach wieder verschwinden. Die Corona-Krise hat dies noch einmal eindrucksvoll gezeigt. Cloud-basierte Lösungen waren in vielen Bereichen die „Strohhalme“, mit denen ganze Volkswirtschaften im Lockdown irgendwie den Kopf über Wasser halten konnten (Kratzke 2020). Man stelle sich nur einmal vor, was passiert wäre, wenn die Corona-Krise 30 Jahre vorher ausgebrochen wäre! Wie hätten wir uns dann im Homeoffice organisiert? Wäre Homeoffice überhaupt möglich gewesen? Es hätte definitiv keine hochskalierbaren Videokonferenzlösungen von Anbietern wie Zoom, Azure Teams oder Google Meet gegeben, die weltweit in kürzester Zeit mit unglaublichen Rechenressourcen hinterlegt werden konnten.

Dieses „Primäre“ hat mittlerweile sogar einen Namen bekommen. Man nennt es „Cloud-native“ (Kratzke und Quint 2017; Kratzke 2018). Als Cloud-native Systeme werden in diesem Buch verteilte Systeme bezeichnet, die bewusst für Cloud-Infrastrukturen und Cloud-Plattformen entwickelt werden und nur effektiv in diesen betreibbar sind. Da diese Systeme primär variable Kosten durch ihren Ressourcenverbrauch generieren (Pay-as-you-go-Kostenmodell, siehe auch Abschnitt 2.3), haben sich Designmuster entwickelt, wie solche Systeme möglichst kostengünstig (d. h. ressourcenschonend) betrieben werden können. Dies hat massiven Einfluss auf die verwendeten Technologien, aber auch auf Architekturen genommen, mit denen solche Cloud-nativen Systeme entwickelt und betrieben werden. Um diese Hintergründe und Mechanismen Cloud-nativer Systeme soll es in diesem Buch gehen.

■ 1.1 An wen sich dieses Buch richtet

Dieses Buch richtet sich insbesondere an Studierende in Informatik- oder verwandten (Master-)Studiengängen. Gleichermaßen adressiert es Dozenten, die derartige Themen im Hochschul- oder beruflichen Weiterbildungskontext vermitteln. Insbesondere die ergänzenden Online-Materialien sind für diese Zielgruppe von Interesse. Aber auch Autodidakten oder IT-Seiteneinsteiger, die im IT-Umfeld Funktionen im weiteren Bereich der Softwareentwicklung innehaben oder diese anstreben, werden adressiert.

Vor dem Hintergrund dieser Zielgruppen werden Themen des Cloud-native Computings mit dem Ziel, einen soliden und kritisch einordnenden Überblick zu geben, überwiegend auf Einsteigerniveau behandelt. Allerdings werden fundiertes softwaretechnisches Basiswissen und Programmierkenntnisse in mindestens einer prozeduralen oder objektorientierten Programmiersprache vorausgesetzt. Idealerweise ist es Python, aber andere Programmiersprachen sind auch gut anzuwenden. Ferner sind Erfahrungen mit unixoiden Betriebssystemen wie beispielsweise Linux von Vorteil.

Dieses Buch richtet sich auch an technische Projektleiter, Berater, Softwarearchitekten und -entwickler, die Cloud-native Technologien und korrespondierende DevOps-Praktiken in Projekten oder Abteilungen einführen wollen oder damit erste Erfahrungen sammeln.

Teil I ist auch für (bzw. die Kommunikation mit) C-Level-Funktionen (z. B. CIO, CTO) in Unternehmen geschrieben worden und kann Softwareentwicklern Argumente liefern, um Cloud-basierte Entwicklungsansätze kritisch, konstruktiv und lösungsorientiert mit Unternehmensleitungen zu diskutieren.

■ 1.2 Was dieses Buch behandelt

Das Buch ist in drei Teile gegliedert, die sich mit unterschiedlichen Bereichen des Cloud-native Computings befassen. Dies erstreckt sich von den theoretischen und konzeptionellen Grundlagen des Cloud Computings über praktisches „Hands-on“-Wissen der Basistechnologien bis hin zu architekturellen Überlegungen und dem verlässlichen Betrieb großer und komplexer Cloud-nativer Systeme.

In **Teil I** werden primär die theoretischen und konzeptionellen **Grundlagen** des Cloud Computings behandelt.

- **Kapitel 2** geht auf die bekannten **Service-Modelle** IaaS, PaaS und SaaS ein, die sich seit mehr als einer Dekade als eine feste konzeptionelle Gliederung im Cloud Computing bewährt haben. Behandelt wird ferner die „Cloud-Ökonomie“ und deren Einfluss auf den Technologiestack und Architekturen von Cloud-nativen Systemen, die wir heutzutage vorfinden.
- **Kapitel 3** geht auf **DevOps** als eine treibende Philosophie des Cloud Computings ein. Insbesondere die DevOps-Prinzipien des Flow motivieren dabei den **Teil II (Everything as Code)** dieses Buches. Die DevOps-Prinzipien des Feedbacks legen die Grundlage für den **Teil III (Observable Architectures)**. Das Kapitel 3 kann also durchaus als gedankliche Klammer gesehen werden.
- Das **Kapitel 4** geht auf den Begriff **Cloud-native** an sich ein, um diesen Begriff präzise zu fassen und als Leitmotiv für dieses Buch aufzugreifen. Dieses Kapitel bildet damit den Einstieg in **Teil II**.

Teil II betrachtet das „Handwerk“ der Cloud-nativen Programmierung. Die Basisfähigkeiten des **Everything as Code** bilden die Grundlage für Cloud-native Systementwicklung größerer Systeme. Das Buch überträgt hier das bewährte Vorgehen der Informatikausbildung vom Programming-in-the-Small zum Programming-in-the-Large auf Cloud-native Systeme. In der Informatikausbildung werden üblicherweise erst die Basisfertigkeiten des Programmierens im Kleinen vermittelt und erst anschließend die Fertigkeiten und Feinheiten der Erstellung von Softwarearchitekturen für größere und komplexere Softwaresysteme. Andernfalls würden Softwarearchitekturen vermutlich als vollkommen realitätsferne und theoretische „Software-Philosophie“ abgetan werden. Das Buch verfolgt also ganz bewusst einen Bottom-up-Ansatz und keinen der häufig anzutreffenden Top-down-Ansätze, die sich oft auf architekturelle Aspekte wie Microservices und Serverless-Architekturen fokussieren, aber im Rahmen dessen dann nur recht isoliert technische Einzelkomponenten tiefer liegender Ebenen wie Container, Functions, Orchestrationsplattformen, Deployment-Pipelines, Infrastruktur Provisioning usw. widmen.

- **Kapitel 6** befasst sich dazu mit der softwaredefinierten Entwicklung von **Deployment Pipelines**, die das Herzstück der Automatisierung in vielen DevOps-Ansätzen von Cloud-native Systemen bilden.
- In **Kapitel 7** wird Infrastructure as Code, also die automatisierte Bereitstellung softwaredefinierter Infrastrukturen, behandelt.
- **Kapitel 8** erläutert, wie Komponenten (inklusive aller Abhängigkeiten) Cloud-native Systeme als standardisierte Deployment Units gebaut und bereitgestellt werden können.
- **Kapitel 9** zeigt, wie aus vielen feingranularen Containern bestehende Cloud-native Systeme mittels Cluster-Technologien orchestriert und mittels Self-Healing-Mechanismen betrieben werden können.
- **Kapitel 10** geht auf noch kleinere und lastabhängig skalierende Einheiten (Functions) ein, die die darunterliegende Infrastruktur im Sinne der sogenannten Serverless-Philosophie sogar komplett wegabstrahieren können.

In **Teil III** geht es um die architekturelle Ebene von Cloud-native Systemen. Dabei werden Architekturansätze und Methodiken vorgestellt, wie Cloud-native Systeme und Anwendungen gebaut werden können, die gemäß den im Kapitel 2 gezeigten ökonomischen Gesetzmäßigkeiten kosteneffektiv betrieben und evolutionär weiterentwickelt werden können.

- **Kapitel 12** befasst sich hierzu mit den beiden das Cloud-native Umfeld prägenden Architekturstilen Microservices und Serverless Architectures.
- In **Kapitel 13** geht es hingegen stärker um den Betrieb und die Beobachtbarkeit von Cloud-native Anwendungen zur Laufzeit. Es werden Themen wie Logging, Monitoring, Tracing, Service Meshs sowie die Analyse und das Management von Verkehrstopologien behandelt.
- **Kapitel 14** bildet den Abschluss des gewählten Bottom-up-Ansatzes dieses Buchs und befasst sich damit, wie sich Cloud-native Anwendungen unter anderem mittels des Domain Driven Designs methodisch entwerfen lassen.

In **Teil IV** geht es um die Sicherheit Cloud-native Systeme. Dabei werden zum einen die technische Sicherheit, aber auch regulatorische Aspekte betrachtet.

- **Kapitel 16** befasst sich eher mit der technischen Sicherheit Cloud-native Systeme vor allem im Sinne einer Härtung von Systemen entlang von sogenannten Angriffsvektoren. Diese Systemhärtung betrachtet vor allem die Ebene virtueller Maschinen in Cloud-Infrastrukturen und die Ebene von Containern in Orchestrationsplattformen.
- **Kapitel 17** fokussiert regulatorische Aspekte, die sich vor allem aus dem deutschen bzw. dem europäischen Rechtsraum (also vor allem aus der Datenschutz-Grundverordnung, DSGVO), aber auch aus internationalen Regularien zum Cloud Computing ergeben, denen vor allem die Hyperscaler aus den USA unterworfen sind und damit auch deren Kunden betreffen.

Teil IV schließt mit Tabelle 18.1, in der alle in diesem Buch behandelten Pattern und Best Practices zur Entwicklung Cloud-native Anwendungen und Dienste aufgeführt sind. Dadurch finden sich insbesondere für die eher „nachschlagenden“ Leser relevante Stellen gegebenenfalls etwas zielgerichteter, als das rein über das Inhaltsverzeichnis möglich wäre. Ohne ein gewisses Überblickswissen ist Tabelle 18.1 aber vermutlich von eher geringerem Wert.

■ 1.3 Sprachliche Konventionen

Insbesondere Autoren deutschsprachiger IT-Literatur stellt sich die Frage, ob man sich der englischen Originalterminologie oder deutscher Übersetzungen bedient. Schnell wirkt die „dogmatische“ Nutzung deutscher Begrifflichkeiten sperrig und wenig intuitiv für den Leser. Daher werden nur in (wenigen) Fällen gängige und direkt herleitbare Übersetzungen genutzt. Ansonsten wird der englischen Originalterminologie gefolgt – zumindest immer dann, wenn es um technische Terminologie geht.

Es wird also beispielsweise der Begriff Service genutzt, wenn ein IT-Dienst (z. B. Webdienst) gemeint ist und damit primär die technische Dimension des Dienstes fokussiert wird. *Microservices* werden also nicht zu Mikrodiensten. Schlicht und ergreifend, weil der englische Begriff *Microservice* eindeutiger und anerkannt in der Domäne ist. In anderen Fällen ist es manchmal schwieriger. So ist der Begriff *Load Balancing* gut und verständlich mit Lastausgleich übersetzbbar. Allerdings wird die hierfür erforderliche Komponente gerne als *Load Balancer* bezeichnet. Man müsste also von Lastausgleicher, Komponente für den *Lastausgleich*, oder *lastausgleichende Komponente* sprechen. Das wirkt – zumindest im Sprachempfinden des Autors – unbeholfen. Daher wird auch in diesen Fällen die Originalterminologie genutzt und so vermieden, dass vom „Lastausgleich durch einen Load Balancer“ gesprochen werden müsste. Ähnlich sieht es beim häufigen Vorgang des *Deployens* aus – also dem technischen Vorgang, eine Ausführungsinstanz auf eine Ressource zu übertragen, zu installieren und dort zur Ausführung zu bringen. *To deploy* lässt sich sogar hervorragend mit mehreren Begriffen wie *ausbringen*, *bereitstellen*, *einsetzen* oder *anwenden* übersetzen. Hier ist die (semi-)synonyme Begriffsvielfalt das Problem, da dem Leser nun schnell durchrutscht, was genau gemeint ist. In solchen Fällen wird daher auch die Originalterminologie bevorzugt. Hierzu müssen dann allerdings Verben gegebenenfalls auch mal „eingedeutscht“ werden – d. h., wir *deployen* dann auch schon mal eine Komponente. Auch Begrifflichkeiten wie *Circuit-Breaker* lassen sich mit Begriffen wie *Sicherung* oder *Schutzschalter* übersetzen. Das Problem ist hier, dass diese Begrifflichkeiten im Deutschen meist in völlig anderen Kontexten gebraucht werden. Von einer Sicherung spricht man meist im Kontext der Elektrik (z. B. wenn wieder eine „Sicherung rausgeflogen ist“) und selten im Kontext von Entwurfsmustern zur Isolation von Fehlern in Teilkomponenten von Softwaresystemen. Auch deutschsprachige Software-experten machen das selten und nutzen meist den englischen Begriff.

Einzig und allein bei Begriffen, bei denen sich gute, direkte und bekannte Übersetzungen etabliert haben, werden diese natürlich genutzt. Daher reden wir nicht über *Fault Tolerance*, sondern über Fehlertoleranz von Systemen. Wir sprechen auch nicht von *Resilient Systems*, sondern von *resilienten Systemen*, wenn wir widerstandsfähige Systeme meinen, die z. B. auf Lastveränderungen durch Skalierung ihrer Ressourcen reagieren oder Fehler in Teilsystemen isolieren können, um diese zu hindern, sich auszubreiten.

■ 1.4 Notationskonventionen

Lange Quelltext-Beispiele werden in diesem Buch grundsätzlich vermieden und in den praktischen Übungen (*Labs*) außerhalb des Buchs auf einer Website isoliert. Dennoch kann es an der ein oder anderen Stelle erforderlich werden, Dinge an konkreten Quelltext-Beispielen zu erläutern. Um das babylonische Sprachwirrwarr im Cloud-nativen Umfeld zumindest in diesem Buch etwas einzudämmen, werden die meisten dieser Beispiele in Python erläutert werden. Listing 1.1 zeigt das übliche Syntax-Highlighting in der Darstellung solcher Quelltexte:

Listing 1.1 Python-Codebeispiel

```
# Example function
def function(data, context):
    return f"I am a FaaS-function to process some { data }"
```

Sie müssen allerdings kein Python-Experte sein, um diese Beispiele nachvollziehen zu können. Die Beherrschung einer imperativen/objektorientierten Programmiersprache ist jedoch sicher von Vorteil.

Insbesondere in Abschnitt 9.2 werden viele YAML-Beispiele genutzt werden. Die YAML-Notation ist eine auf Einrückung basierendes Textformat zur Darstellung von strukturierten Objekten und Listen. Sie wird üblicherweise wie in Listing 1.2 gezeigt verwendet.

Listing 1.2 Die übliche YAML Long-Notation

```
list:
  - A
  - B
object:
  attr1: value1
  attr2: value2
  nested_list:
    - v1
    - v2
    - v3
```

Um die Beispiele in diesem Buch jedoch so kompakt wie möglich zu halten, wird die eher aus JSON bekannte und weniger gebräuchliche YAML Inline-Notation (siehe Listing 3) verwendet. Diese Inline-Notation wird für Objekte und Listen immer dann genutzt, wenn dadurch ein Zusammenhang in eine Zeile passt. Das Beispiel aus Listing 1.2 kann also äquivalent wie in Listing 1.3 ausgedrückt werden.

Listing 1.3 Die weniger gebräuchliche YAML Inline-Notation

```
list: ["A", "B"]
object: { attr1: "value1", attr2: "value2", nested_list: ["v1", "v2", "v3"] }
```

Sowohl die Form aus Listing 1.2 und Listing 1.3 sind beides valide YAML-Darstellungen, die von YAML-Parsern gelesen werden können. Diese beiden Notationsformen werden in diesem Buch so kombiniert, dass einerseits Definitionsstrukturen schnell ersichtlich werden und dennoch der Zeilenumfang minimiert wird. Meist wird daher die YAML Inline-Notation auf Listen von Werten angewendet und bei Objekten nur, wenn diese aus einem einzigen Attribut bestehen.

Listing 1.4 zeigt diesen „Stilmix“ für eine möglichst kompakte, aber lesbare Darstellung von Definitionsstrukturen.

Listing 1.4 Der YAML-Stil dieses Buches

```
list: ["A", "B"]
object:
    attr1: "value1"
    attr2: "value2"
    nested_list: ["v1", "v2", "v3"]
another: { name: "Objekt" }
```

■ 1.5 Ergänzende Materialien

Der Autor weiß aus eigener Erfahrung, dass Dozenten und Trainer Lehr- und Schulungs-inhalte aus zahlreichen Quellen zusammenträgen, kombinieren und diese Quellen als in sich stimmiges Lernerlebnis aufbereiten. Dies ist nicht einfach nur ein „Zusammenkopieren“, denn ein in sich stimmiges Resultat erfordert jede Menge Aufwand.

Um diesen oft nach außen so unscheinbaren Aufwand anzuerkennen und zu honorieren, werden alle ergänzenden Materialien auf der Website zu diesem Buch für Leser, aber vor allem für Dozenten und Trainer in einer universellen Creative Commons-Lizenz (CC0) und in einer bearbeitbaren Form bereitgestellt und können daher für eigene Veranstaltungen als Ganzes oder nur in Teilen verwendet und mit eigenen Inhalten kombiniert oder ergänzt werden. Dies ist unabhängig davon, ob es sich um kommerzielle Schulungen oder Veranstaltungen an öffentlichen Einrichtungen wie Universitäten oder Hochschulen handelt.

Da sich die Technologien im Cloud-Computing extrem dynamisch entwickeln, werden konkrete Technologien, Tools und Frameworks nur dort und im Sinne von Tyrepräsentanten verwendet, wo es der Vermittlung der Inhalte dienlich ist. Da es natürlich dennoch sinnvoll ist, Inhalte praktisch zu vertiefen, werden vertiefende Labs und Übersichten von Tools extern auf einer Website zum Buch gepflegt und aktualisiert, sodass sichergestellt werden kann, dass die Inhalte des Buchs auch immer mit aktuellen Versionen von Tools und Frameworks nachvollzogen werden können. Andernfalls würde dieses Buch sehr schnell aktualisiert werden müssen.

Jedes Kapitel von Teil II, III und IV schließt daher immer mit einer Zusammenfassung, in der folgende Punkte aufbereitet werden:

1. Wichtigste Inhalte des Kapitels
2. Diskussion weiterführender Literatur und Quellen
3. Verweise auf vertiefende Labs
4. Verweise auf Toolübersichten inklusive Produktlinks (falls vorhanden)
5. Hinweise für Dozenten und Trainer inklusive Links auf Slides für den eigenen Unterricht.

Im eher einordnenden und Überblick gebenden Teil I wird diese Übersicht allerdings nur im Abschnitt 4.3 zusammenfassend für den gesamten Teil I gegeben.

Die Website zu diesem Buch
findet sich unter diesem QR-Code:



<https://bit.ly/39LOBA3>

Teil I: Grundlagen

2

Cloud Computing

„It's the economy, stupid!“

Bill Clinton, 42. Präsident der USA

Gemäß der sogenannten NIST-Definition versteht man unter Cloud Computing einen „*all-gegenwärtigen, bequemen, bedarfsgerechten Netzwerkzugriff auf einen gemeinsamen Pool konfigurierbarer Rechenressourcen, die schnell und mit minimalem Verwaltungsaufwand oder Interaktion mit Service-Providern bereitgestellt, aber auch wieder freigegeben werden können*“ (Mell und Grance 2011).

Cloud Computing ordnet sich damit im Spektrum verteilter Systeme im Bereich des Service Computings und weniger im Bereich des High Performance bzw. Super-Computings ein, auch wenn die Einflussfaktoren mittlerweile mannigfaltig und keinesfalls mehr als trennscharf zu bezeichnen sind (siehe Bild 2.1). Insbesondere im NoSQL- sowie Machine Learning-/Big-Data-Bereich gehen Super-Computing und Service Computing zunehmend mehr ineinander über.

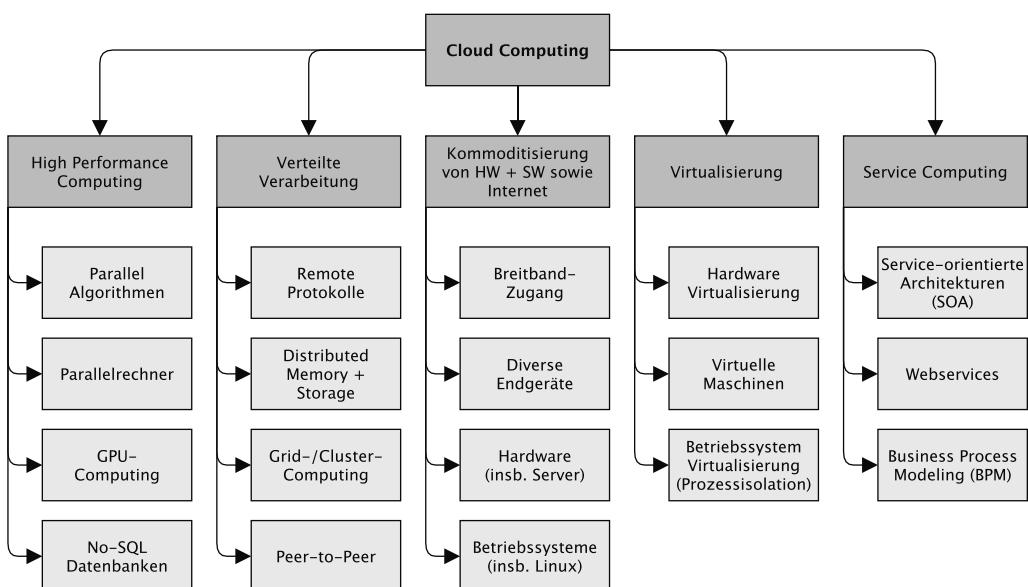


Bild 2.1 Einflussfaktoren auf das Cloud Computing

Während Super-Computing eine wichtige Rolle im Bereich der computergestützten Wissenschaften (Computational Science) spielt und für eine Vielzahl rechenintensiver wissenschaftlicher Aufgaben in verschiedensten Bereichen eingesetzt wird (z. B. Quantenmechanik, Wettervorhersage, Klimaforschung, physikalische Simulationen usw.), verstehen wir unter Service Computing eher einen interdisziplinären Ansatz, der sich mit der Frage beschäftigt, wie Informationstechnologien die geschäftsrelevante Erzeugung von Produkten und Dienstleistungen substanzial unterstützen können. Dabei finden im Service Computing u. a. Webservices, Service-orientierte Architekturen (SOA), Geschäftsprozessmodellierung, Transformations- und Integrationstechnologien – aber eben auch vermehrt „Enabling Technologies“ wie Cloud Computing – Anwendung, die durchaus substanzialen Einfluss auf Architekturen und Systeme haben. So hat sich beispielsweise SOA aufgrund des Cloud Computing-Einflusses in den letzten Jahren mehr und mehr zu einem Microservice-basierten Architekturansatz fortentwickelt. Warum das so ist, werden wir unter anderem in Abschnitt 2.3 und Abschnitt 2.4 sehen.

■ 2.1 Service-Modelle

Im Allgemeinen werden, wie in Bild 2.2 gezeigt, im Cloud Computing fünf wesentliche Service-Merkmale, vier Deployment-Modelle und drei Service-Modelle unterschieden (Mell und Grance 2011). Wir werden im weiteren Verlauf sehen, dass diese Darstellung an der ein oder anderen Stelle verfeinert werden kann (siehe beispielsweise Abschnitt 8.1 und Bild 8.3). Dennoch ist das zugrunde liegende NIST-Modell des Cloud Computings (Mell und Grance 2011) so prägend, dass es Sinn macht, sich an diesem Modell, seinen Merkmalen, Bereitstellungsformen und Service-Modellen zu orientieren.

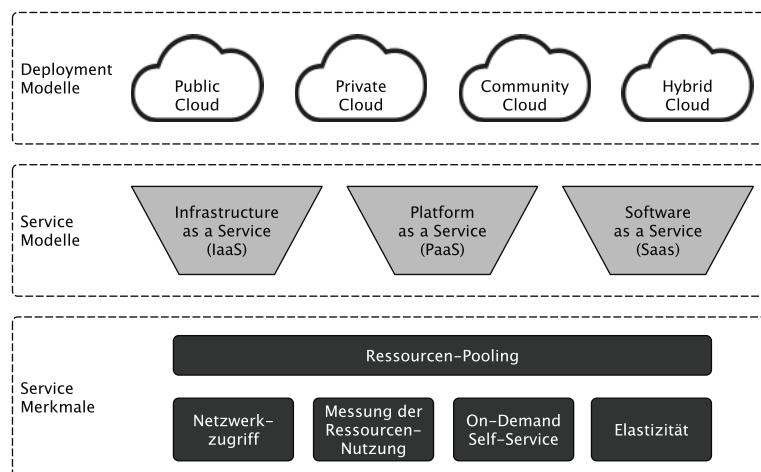


Bild 2.2 NIST-Modell des Cloud Computings

Zu den fünf wesentlichen Merkmalen des Cloud Computings sind die folgenden zu zählen:

1. **On-Demand Self-Service:** Ein Verbraucher kann Ressourcen, wie z. B. Serverzeit und Netzwerkspeicher, nach Bedarf automatisch anfordern, ohne dass hierfür eine manuelle Tätigkeit aufseiten des Cloud-Service-Providers erforderlich ist.
2. **Netzwerkzugriff:** Die Ressourcen werden über öffentliche Netzwerke bereitgestellt und der Zugriff auf diese Ressourcen erfolgt über standardisierte und weitverbreitete Internetprotokolle, die die Nutzung von Cloud-Ressourcen durch heterogene Client-Plattformen ermöglichen.
3. **Elastizität:** Ressourcen können schnell und bedarfsgerecht bereitgestellt, aber auch wieder freigegeben werden. Für den Verbraucher erscheinen die für die Bereitstellung verfügbaren Ressourcen virtuell unbegrenzt und können in beliebiger Menge und zu jeder Zeit angefordert werden. Dies fördert horizontale Skalierungsformen.
4. **Messung der Ressourcennutzung:** Cloud-Systeme steuern und optimieren automatisch ihre Ressourcennutzung, indem sie den Ressourcenverbrauch auf einer geeigneten Abstraktionsebene messen (z. B. Speicherverbrauch, Processing-Cycles, Bandbreite, aktive Benutzerkonten usw.). Die Überwachung und Messung der Ressourcennutzung schafft sowohl für den Service-Provider als auch für den Nutzer von Cloud Services Transparenz.
5. **Ressourcen-Pooling:** Die Computing-Ressourcen des Providers werden gepoolt, um mehrere Kunden mit einem Multi-Tenant-Modell zu bedienen. Dabei werden physische und virtuelle Ressourcen dynamisch den Nutzern zugewiesen und bei Bedarf auch realisiert. Der Kunde hat im Allgemeinen keine detaillierte Kontrolle oder Kenntnis über den genauen Standort der bereitgestellten Ressourcen, kann aber den Standort auf einer höheren Abstraktionsebene (z. B. Land, Region oder Rechenzentrum) angeben.

Cloud Services werden zumeist in Private- bzw. Public Cloud-Formen unterschieden. Die ebenfalls existierenden Hybrid- und Community-Formen sind oft nicht so präsent in der öffentlichen Diskussion, vermutlich weil sie im Service Computing kaum ihre Stärken ausspielen können.

- Unter einer **Public Cloud** versteht man eine Cloud-Infrastruktur für die offene Nutzung durch die Allgemeinheit. Sie kann im Besitz einer geschäftlichen, akademischen oder staatlichen Organisation oder einer Kombination davon sein und von dieser verwaltet und betrieben werden. Sie befindet sich auf den Liegenschaften des Cloud-Anbieters (d. h. Off-Premise für die Cloud-Nutzer).
- Unter einer **Private Cloud** versteht man hingegen eine Cloud-Infrastruktur, die für die exklusive Nutzung durch eine einzelne Organisation mit mehreren Verbrauchern (z. B. Geschäftseinheiten) betrieben wird. Sie kann sich im Besitz der Organisation, eines Dritten oder einer Kombination aus beiden befinden. Dabei ist es unerheblich, ob die Infrastruktur sich auf den Liegenschaften der Organisation (d. h. On-Premise für die Cloud-Nutzer) oder nicht befindet.
- Unter der weniger bekannten Form der **Community Cloud** wird eine Cloud-Infrastruktur verstanden, die für die exklusive Nutzung durch eine bestimmte Gemeinschaft von Verbrauchern aus Organisationen betrieben wird. Diese Gemeinschaft hat meist gemeinsame Anliegen (z. B. Mission, Sicherheitsanforderungen, Richtlinien und Compliance-Überlegungen). Sie kann im Besitz einer oder mehrerer Organisationen in der Community, einer dritten Partei oder einer Kombination von ihnen sein und von diesen verwaltet und

betrieben werden. Dabei ist es unabhängig, ob die Community Cloud ausschließlich auf den Liegenschaften der Gemeinschaft betrieben wird. Community Clouds können also sowohl On-Premise als auch Off-Premise betrieben werden.

- Schließlich wird als **Hybrid Cloud** eine Cloud-Infrastruktur verstanden, die eine Komposition aus zwei oder mehreren oben genannter Cloud-Infrastruktur-Formen (private, public, community) bildet. Diese bleiben eigenständige Einheiten, werden aber durch standardisierte oder proprietäre Technologie miteinander verbunden, die die Portabilität von Daten und Anwendungen ermöglicht (z. B. Cloud Bursting für den Lastausgleich zwischen Cloud-Infrastrukturen).

Mittels Cloud-Computing lassen sich Teile der IT-basierten Wertschöpfung an externe Dienstleister (Cloud-Provider) auslagern. Der Auslagerungsumfang wird dabei häufig in die Kategorien Infrastructure as a Service (IaaS, siehe Abschnitt 2.1.1), Platform as a Service (PaaS, siehe Abschnitt 2.2) und Software as a Service (SaaS, siehe Abschnitt 2.2.1.1) eingeteilt. Von IaaS über PaaS zu SaaS wird dabei der ausgelagerte Anteil immer größer, wie Bild 2.3 zeigt. Mit dem Umfang der Auslagerung wird allerdings auch die potenzielle Abhängigkeit (Vendor Lock-in) eines Kunden zu einem Cloud-Provider größer. Unter einem Lock-in-Effekt versteht man generell eine enge Kundenbindung an Produkte/Dienstleistungen eines Anbieters in Form einer technisch-funktionalen Kundenbindung, die es dem Kunden wegen entstehender Wechselkosten und sonstiger Wechselbarrieren erschwert, ein Produkt oder einen Service eines Anbieters mit dem Produkt oder Service eines anderen Anbieters auszutauschen. Im Cloud Computing entsteht dieser Effekt meist durch nichtstandardisierte Cloud-Service APIs der einzelnen Provider. Je höher man in den Schichten kommt, desto spezifischer und damit weniger austauschbar werden die bereitgestellten Cloud-Services, und desto höher ist die Lock-in-Gefahr.

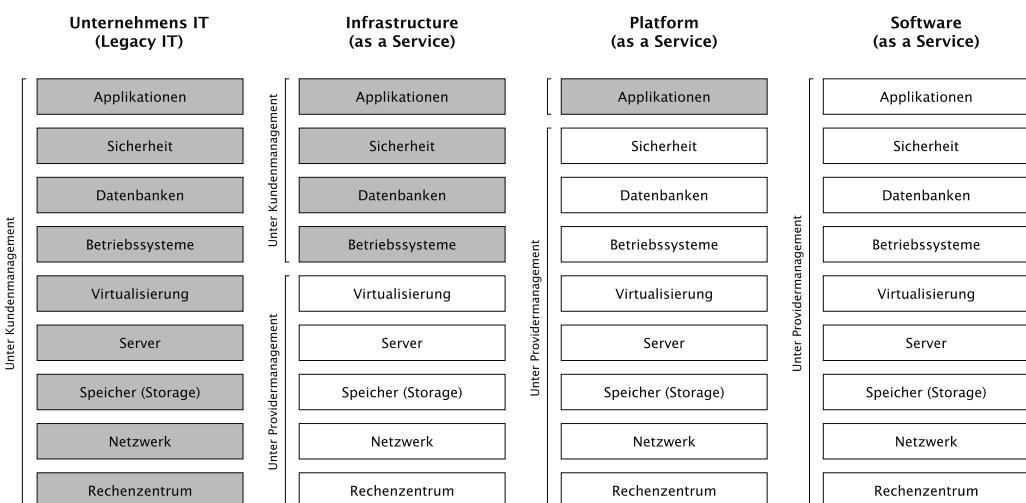


Bild 2.3 Auslagerung der Wertschöpfung bei IaaS, PaaS und SaaS

2.1.1 Infrastructure as a Service (IaaS)

Beim IaaS-Modell bietet ein Provider physische und virtuelle Hardware wie Server, Speicher und Netzwerkinfrastruktur an, die über eine Self-Service-Schnittstelle schnell bereitgestellt und außer Betrieb genommen werden kann. Dies ermöglicht es z. B., im Rahmen von periodischen Workloads mit wiederkehrenden Lastspitzen IT-Ressourcen flexibel und vor allem lastgetrieben bereitzustellen.

Die Fähigkeit, die dem Kunden zur Verfügung gestellt wird, besteht also in der schnellen und elastischen Bereitstellung von Verarbeitungs-, Speicher-, Netzwerk- und anderen grundlegenden Rechenressourcen, auf denen der Kunde beliebige Software, einschließlich Betriebssystemen und Anwendungen, einsetzen und ausführen kann.

Der Kunde verwaltet oder kontrolliert die zugrunde liegende Cloud-Infrastruktur zwar nicht, hat aber die Kontrolle über Betriebssysteme, Speicher und bereitgestellte Anwendungen sowie möglicherweise eine begrenzte Kontrolle über ausgewählte Netzwerkkomponenten (z. B. Host-Firewalls).

In Anlehnung an (Fehling u. a. 2014) bezeichnen wir das zugehörige Service-Offering als **elastische Infrastruktur** zum Zwecke des Bereitstellungs von virtuellen Servern, persistenten Speicher und Netzwerkkonnektivität. Eine elastische Infrastruktur bietet zumeist vorkonfigurierte virtuelle Server-Images, persistenten Speicher und Netzwerkkonnektivität, die von Kunden über eine Self-Service-Schnittstelle angefordert werden können. Ferner werden Last- und Nutzungsdaten vom Provider bereitgestellt, um über die Ressourcenauslastung zu informieren, die für eine nachvollziehbare Abrechnung und die Automatisierung von Verwaltungsaufgaben erforderlich ist.

2.1.2 Platform as a Service (PaaS)

Beim PaaS-Modell stellen Provider IT-Ressourcen in Form einer Applikations-Hosting-Umgebung für Kunden bereit. Ein Cloud-Provider bietet hierfür verwaltete Betriebssysteme und Middleware an. Auch viele Betriebsvorgänge werden vom Anbieter übernommen, wie z. B. die elastische Skalierung und Ausfallsicherheit gehosteter Anwendungen.

Die dem Kunden zur Verfügung gestellte Fähigkeit besteht somit darin, in einer Cloud-Infrastruktur vom Kunden erstellte oder erworbene Anwendungen bereitzustellen, die mit vom Anbieter unterstützten Programmiersprachen, Bibliotheken, Diensten und Tools erstellt wurden. Der Kunde verwaltet oder kontrolliert somit zwar nicht die zugrunde liegende Cloud-Infrastruktur, hat aber die Kontrolle über die bereitgestellten Anwendungen.

In Anlehnung an (Fehling u. a. 2014) bezeichnen wir das zugehörige Service-Angebot als **elastische Plattform** und verstehen dies als eine Middleware zur Ausführung benutzerdefinierter Anwendungen, deren Kommunikation und Datenspeicherung über eine netzwerk basierte Self-Service-Schnittstelle angeboten wird. Auf diese Weise können Anwendungskomponenten verschiedener Kunden auf einer gemeinsamen Middleware gehostet werden, die vom Anbieter bereitgestellt und gewartet wird. Diese Vereinheitlichung ermöglicht die gemeinsame Nutzung von Ressourcen und eine Automatisierung bestimmter Verwaltungsaufgaben auf Provider-Seite, z. B. die Bereitstellung von Anwendungen und die Verwaltung von Updates.

2.1.3 Software as a Service (SaaS)

Beim SaaS-Modell stellen Anbieter IT-Ressourcen in Form von für Menschen nutzbare Anwendungssoftware für Kunden bereit, um Self-Service, schnelle Elastizität und Pay-per-Use-Preise zu ermöglichen. Insbesondere kleine und mittlere Unternehmen verfügen oft nicht über die Arbeitskraft und das Know-how, um individuelle Softwareanwendungen zu entwickeln. Ferner sind viele Anwendungen zu Massenware geworden, die von vielen Unternehmen verwendet werden, aber kaum dazu beitragen, sich von Wettbewerbern abzuheben (siehe Abschnitt 14.2.1). Dies umfasst z. B. Office-Suiten, Software für die Zusammenarbeit oder Kommunikationssoftware.

Die dem Verbraucher zur Verfügung gestellte Fähigkeit besteht also bei SaaS darin, Anwendungen eines Anbieters zu nutzen, ohne die dafür erforderliche Infrastruktur oder Plattform betreiben zu müssen. Der Zugriff auf die Anwendungen erfolgt zumeist von verschiedenen Client-Geräten, wie z. B. einem Webbrowser (z. B. webbasierte E-Mail) oder über eine Programmschnittstelle.

Der Verbraucher verwaltet oder steuert die zugrunde liegende Cloud-Infrastruktur oder Cloud-Plattform einschließlich Netzwerk, Server, Betriebssystem, Speicher oder sogar einzelne Anwendungsfunktionen somit nicht selbst. Es sind jedoch – meist in sehr begrenztem Umfang – benutzerspezifische Konfigurationseinstellungen möglich (z. B. Anpassung der Benutzeroberfläche an Unternehmens-Styleguide-Vorgaben).



Anmerkungen für IT-Manager: Cloud-natives Denken funktioniert auch „ohne Cloud“

Obwohl Public Cloud Computing in vielen Fällen sehr vorteilhaft sein kann, gibt es auch Anwendungsfälle, die als problematisch gelten und bei denen es schwierig ist, die Vorteile des Deployment-Modells Public Cloud zu nutzen, wie folgende Beispiele zeigen:

- **Kritische Infrastrukturen:** In Bereichen wie der Energieversorgung, dem Gesundheitswesen oder der öffentlichen Sicherheit werden kritische Infrastrukturen betrieben, deren Ausfall schwerwiegende Folgen haben kann. Hier ist oft ein Höchstmaß an Kontrolle und Sicherheit erforderlich, das schwierig mit Public Clouds zu erzielen ist.
- **Datenschutz und Compliance:** Insbesondere Unternehmen, die personenbezogene Daten (DSGVO) verarbeiten, müssen sicherstellen, dass ihre Daten sicher sind und den geltenden Datenschutz- und Compliance-Anforderungen entsprechen. Die Verarbeitung personenbezogener Daten in Public Clouds kann hier problematisch sein (siehe auch Kapitel 17). Es kann schwierig sein, sicherzustellen, dass die Daten sicher sind und die geltenden Vorschriften insbesondere bei internationalen Datentransfers eingehalten werden.
- **Kosten:** Obwohl öffentliche Clouds in vielen Fällen kostengünstiger sein können als die Bereitstellung und Verwaltung einer eigenen IT-Infrastruktur, gibt es genauso Anwendungsfälle, in denen die Nutzung der öffentlichen Cloud unwirtschaftlich sein kann. Dies gilt insbesondere dann, wenn die Anwendung hohe

Anforderungen an Leistung, Speicherplatz oder Bandbreite hat (also eine gewisse kritische Größe erreicht hat). Aber es trifft auch für Anwendungsfälle zu, die wenig Skalierungsbedarf haben und durch eine relativ konstante Grundlast gekennzeichnet sind. Hier können Public Clouds ihre Kostenvorteile oft nur teilweise ausspielen.

Oft wird (unbewusst) unter einem Cloud-native Ansatz implizit die Bereitstellung in Public Clouds angenommen. Das ist aber eine verkürzte Betrachtung, denn dabei wird übersehen, dass es auch die Bereitstellungsmodelle Private, Hybrid oder Community Cloud gibt. Cloud-native Anwendungen funktionieren mit allen genannten Bereitstellungsmodellen gleichermaßen und kommen damit grundsätzlich auch für eher als problematisch angesehene Anwendungsfälle in Betracht, die sich oft mittels Private-Cloud-Ansätzen in den Griff bekommen lassen.

Cloud-native fokussiert nicht primär das Bereitstellungsmodell

Auch für Private Clouds bieten Cloud-native Technologien viele Vorteile. Hier geht es dann weniger um die Migration in die Public Cloud, sondern vielmehr um die Modernisierung und Standardisierung einer bestehenden, oft historisch gewachsenen Infrastruktur und die Standardisierung des Betriebs von Software. Insbesondere für Manager, die in einer eher klassisch geprägten Unternehmens-IT groß geworden sind, ist es wichtig zu verstehen, dass Cloud-native Technologien nicht nur mit Public Clouds funktionieren. Vielmehr handelt es sich um einen Ansatz, der auf modernen Entwicklungsmethoden, Containerisierung und Automatisierung basiert und unabhängig davon eingesetzt werden kann, ob die Infrastruktur privat, öffentlich, vor Ort oder gänzlich anders bereitgestellt wird.

Cloud-native fokussiert die Standardisierung des Betriebs von Anwendungen

Der Einsatz von Cloud-nativer Technologie ermöglicht es, eine agile und flexible Infrastruktur aufzubauen, die auf die Bedürfnisse von Entwicklern und Benutzern zugeschnitten ist. Mit Containern und Microservices lassen sich Anwendungen schneller entwickeln und bereitstellen, was zu einer höheren Produktivität und einem besseren Kundenerlebnis führt.

Hierfür werden etablierte Technologien und Methoden wie Container-Laufzeitumgebungen, Container-Orchestratoren wie Kubernetes und Deployment-Pipelines für die Bereitstellung von standardisierten Anwendungskomponenten (Container) in einer privaten Cloud-Umgebung genutzt. Diese Technologien ermöglichen es Unternehmen, ihre Anwendungen in einer hochverfügbaren und skalierbaren Umgebung zu containerisieren und wesentlich standardisierter zu betreiben.

Cloud-native fördert Agilität und You-Build-It-You-Run-It Ansätze

Ein weiterer, oft übersehener Vorteil der Cloud-nativen Technologie ist die Möglichkeit, Entwicklern und Anwendern ein Self-Service-Modell anzubieten, ähnlich wie bei der öffentlichen Cloud. Dadurch können sie schnell und einfach neue Anwendungen und Dienste bereitstellen, ohne auf die Unterstützung zentraler IT-Teams angewiesen zu sein. Dies beschleunigt insbesondere agile Bottom-up-Entwicklungen.

Das Konzept des Service Ownership ist dabei eine hilfreiche agile Arbeitsweise, die auf der Idee basiert, dass das Team, das für die Entwicklung einer Anwendung (Dev) oder eines Dienstes verantwortlich ist, auch für deren Betrieb und Wartung (Ops) zuständig ist und daher oft als DevOps bezeichnet wird.

„Cloud-native Denken“ ist also ...

... etwas anderes als die Auswahl eines Cloud-Anbieters. Bei Cloud-native geht es in erster Linie darum, Anwendungen und Dienste von Grund auf für Cloud-Infrastrukturen und -Plattformen zu entwickeln und zu optimieren. Im Gegensatz zu herkömmlichen Anwendungen, die oft auf lokalen Servern oder in Rechenzentren auf virtuellen Maschinen laufen, werden Cloud-native Anwendungen speziell für die Skalierbarkeit von Cloud-Infrastrukturen und -Plattformen konzipiert und optimiert.

Ein Unternehmen kann dabei problemlos sein eigener Provider sein, was manchmal aufgrund von Kritikalität oder regulatorischen Anforderungen durchaus sinnvoll ist. Da insbesondere im Public Cloud Computing eine hohe Preistransparenz aufgrund des Pay-as-you-go-Kostenmodells existiert, basiert das Cloud-Native-Modell auf einer Reihe von Grundsätzen, die darauf ausgerichtet sind, die Ressourcen möglichst effektiv zu nutzen (und damit bspw. auch den CO₂-Footprint zu reduzieren). Davon profitieren auch Anwendungen, die ausschließlich in Private Clouds betrieben werden sollen:

- **Skalierbarkeit:** Cloud-native Anwendungen müssen schnell und effektiv auf sich ändernde Anforderungen reagieren. Sie präferieren horizontale gegenüber vertikaler Skalierung.
- **Verfügbarkeit durch Automatisierung:** Cloud-native Anwendungen müssen in der Lage sein, mit Ausfällen oder Störungen umzugehen, indem sie ihre Funktionalität beibehalten oder schnell wiederherstellen, einschließlich der Verwendung verteilter Architekturen und Automatisierung, um Ausfälle zu minimieren.
- **Agilität:** Cloud-native Anwendungen sollten schnell und agil entwickelt, getestet und bereitgestellt werden können, was eine enge Zusammenarbeit zwischen Entwicklern, IT-Betrieb und anderen beteiligten Teams erfordert.
- **Microservices:** Cloud-native Anwendungen werden häufig in kleinere, unabhängige Dienste, sogenannte Microservices, unterteilt. Diese Dienste können unabhängig voneinander entwickelt, getestet und bereitgestellt werden, was die Flexibilität und Skalierbarkeit erhöht.
- **Standardisierung des Betriebs:** Container sind leichtgewichtige, portable Einheiten, mit denen sich Anwendungen und Dienste schnell erstellen, testen, bereitstellen und standardisiert betreiben lassen.

Cloud-native ist also vielmehr eine Denkweise, wie Anwendungen entwickelt und standardisiert betrieben werden sollen, die ressourceneffizient und ausfallsicher sind. Dabei wird die Betriebserfahrung von Generationen von IT-Administratoren in Form automatisierter Orchestrationsprozesse externalisiert und genutzt. Dabei spielt es keine Rolle, ob nun Amazon, Google, Microsoft, Alibaba oder das eigene Rechenzentrum diese Systeme betreibt. Weder der Standort noch ein Anbieter ist entscheidend für Cloud-natives Denken!

■ 2.2 Cloud-Ökonomie

Alle genannten Service-Modelle (IaaS, PaaS, SaaS) folgen dabei denselben wirtschaftlichen Gesetzmäßigkeiten. Beim sogenannten Pay-as-you-go-Kostenmodell werden nur die Ressourcen abgerechnet, die auch tatsächlich von einem Kunden angefordert werden. Aus Sicht des Kunden besteht also das wirtschaftliche Interesse vor allem darin, Cloud-Systeme mit einem möglichst geringen „Over-Provisioning“ zu betreiben, also Lastkurven mittels Skalierung möglichst eng und schnell folgen zu können (siehe Bild 2.4). Dies ist in klassischen Rechenzentren nicht – oder nur sehr begrenzt – möglich.

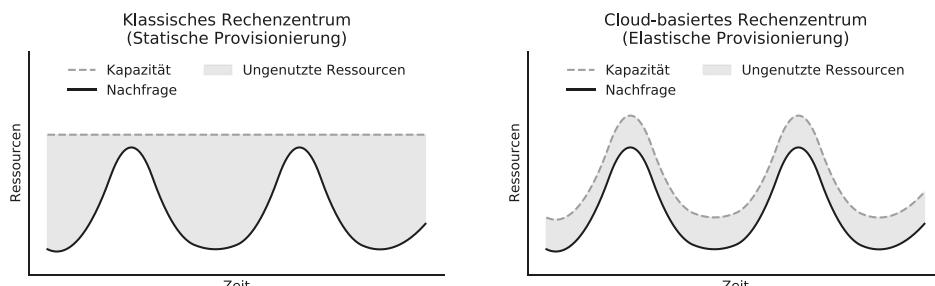


Bild 2.4 Statische und elastische Provisionierung von Ressourcen

2.2.1 Eignung von unterschiedlichen Arten von Workloads

Die Betrachtung von Workloads ist naturgegeben immer sehr anwendungsspezifisch, und man muss vorsichtig sein, nicht zu übergeneralisierende Ratschläge zu geben. Dennoch lassen sich unterschiedliche Workload-Arten ausmachen, die ökonomisch unterschiedlich geeignet für Cloud Computing sind. Dem Leser sei an dieser Stelle das Studium von (Weinman 2011) empfohlen, dessen Überlegungen hier zusammenfassend dargestellt werden.

Eine Pay-per-Use-Lösung macht immer dann offensichtlich Sinn, wenn die Stückkosten für On-Demand-Cloud-Services c niedriger sind als dedizierte, eigene Kapazitäten d . Oft können Cloud-Provider diesen Kostenvorteil bieten – aber nicht immer. Dies hängt leicht nachvollziehbar von den internen Kostenstrukturen eines Unternehmens ab und ist somit hochgradig unternehmensspezifisch.

Obwohl es kontraintuitiv erscheint, macht eine reine Cloud-Lösung aber auch in Szenarien Sinn, in denen die Stückkosten c höher als die Kosten für eigene Kapazitäten d sind. Allerdings nur, solange das Verhältnis von Spitzenlast p zu Durchschnittslast a der Nachfragekurve höher ist als das Kostenverhältnis der Stückkosten von On-Demand-Kapazität c zu dedizierter Kapazität d .

$$\frac{c}{d} < \frac{p}{a} \Leftrightarrow c < d \frac{p}{a} \Rightarrow c_{\max} := d \frac{p}{a}$$

Mit anderen Worten: Selbst wenn Cloud-Dienste doppelt so viel kosten wie In-House-Dienste, ist eine reine Cloud-Lösung für solche Bedarfskurven sinnvoll, bei denen das Verhältnis von

Spitzenwert zu Durchschnittswert zwei zu eins oder höher ist. Dies ist in einer Vielzahl von Branchen öfter der Fall, als man annehmen würde. Der Grund dafür ist, dass die dedizierte Lösung mit fester Kapazität für den Spitzenbedarf gebaut werden muss, während die Kosten der On-Demand-Pay-per-Use-Lösung proportional zum Durchschnitt sind (siehe auch Bild 2.4).

Je größer das Peak-to-Average-Verhältnis $\frac{p}{a}$ also ist, desto eher ist ein Anwendungsfall (rein ökonomisch betrachtet) für cloud-basierte Lösungen interessant. Betrachten wir vor diesem Hintergrund einmal die folgenden prototypischen Workloads, die so entweder in Reinform oder in überlagerten Kombinationen (z. B. periodischer Workload, der durch einen kontinuierlich steigenden Workload überlagert wird) im echten Leben häufig anzutreffen sind.

Statische Workloads (siehe Bild 2.5 A) sind durch ein mehr oder weniger flaches Lastprofil über die Zeit innerhalb bestimmter Grenzen gekennzeichnet. Eine Anwendung mit statischem Workload wird kaum von elastischen Infrastrukturen oder Plattformen profitieren können, da die Anzahl der benötigten Ressourcen konstant ist. Diese Arten von Workloads sind aber eher selten.

Häufiger sind hingegen periodische Aufgaben und Routinen (siehe Bild 2.5 B), zum Beispiel monatliche Gehaltsabrechnungen, monatliche Telefonrechnungen, jährliche Autoinspektionen, wöchentliche Statusberichte oder die tägliche Nutzung der öffentlichen Verkehrsmittel während der Hauptverkehrszeit. Solche Aufgaben und Routinen treten in wohldefinierten Intervallen auf und erzeugen daher **periodische Workloads** in der Nutzung involvierter IT-Systeme. Aus Kundensicht besteht das Kosteneinsparungspotenzial bei periodischen Lasten in der Außerbetriebnahme von Ressourcen in Nicht-Spitzenzeiten.

Als Spezialfall der periodischen Workloads können die Spitzen der periodischen Auslastung in einem sehr langen Zeitraum auch in Form **einmaliger/seltener Workloads** auftreten (siehe Bild 2.5 C). Oft ist diese Spur im Voraus bekannt, da sie mit einem bestimmten Ereignis (z. B. olympische Spiele alle vier Jahre) oder einer Aufgabe korreliert. In solchen Szenarien können die Bereitstellung und Außerbetriebnahme von IT-Ressourcen oft als manuelle Aufgaben realisiert werden, da sie zu einem bekannten Zeitpunkt erfolgen.

Zufällige Workloads sind eine Verallgemeinerung der periodischen Workloads, da sie Elastizität erfordern, aber nicht vorhersehbar sind (siehe Bild 2.5 D). Solche Workloads treten in der realen Welt recht häufig auf. Hier sind die ungeplante Bereitstellung und Außerbetriebnahme von IT-Ressourcen erforderlich. Die notwendige Bereitstellung und Außerbetriebnahme von IT-Ressourcen müssen daher automatisiert erfolgen, um die Anzahl der Ressourcen an die sich ändernde Last anzupassen.

Bei vielen Anwendungen ändert sich auch die Last kontinuierlich über einen längeren Zeitraum. Häufig sind solche Lasten in Form eines Basistrends als Hintergrund-Workload in anderen Workloads (z. B. periodischen Workloads) enthalten. Sich **kontinuierlich ändernde Workloads** sind durch ein kontinuierliches Wachstum oder einen kontinuierlichen Rückgang der Auslastung gekennzeichnet (siehe Bild 2.5 E/F). Rein wirtschaftlich ist es dabei egal, ob ein Workload steigt oder sinkt, denn der Flächeninhalt (also die Einsparung) ergibt sich ja aus der Differenz der statischen und elastischen Provisionierungskurven. Der Bedarf persistenten Speichers unterliegt oft solch einem kontinuierlich wachsenden Trend. Es wird in vielen Anwendungsfällen eben mehr gespeichert als gelöscht.

Wenn man diese Workloads hinsichtlich ihres $\frac{p}{a}$ aufsteigend sortiert, erhält man folgende rein ökonomische Eignungsreihenfolge von Workloads für das Cloud Computing:

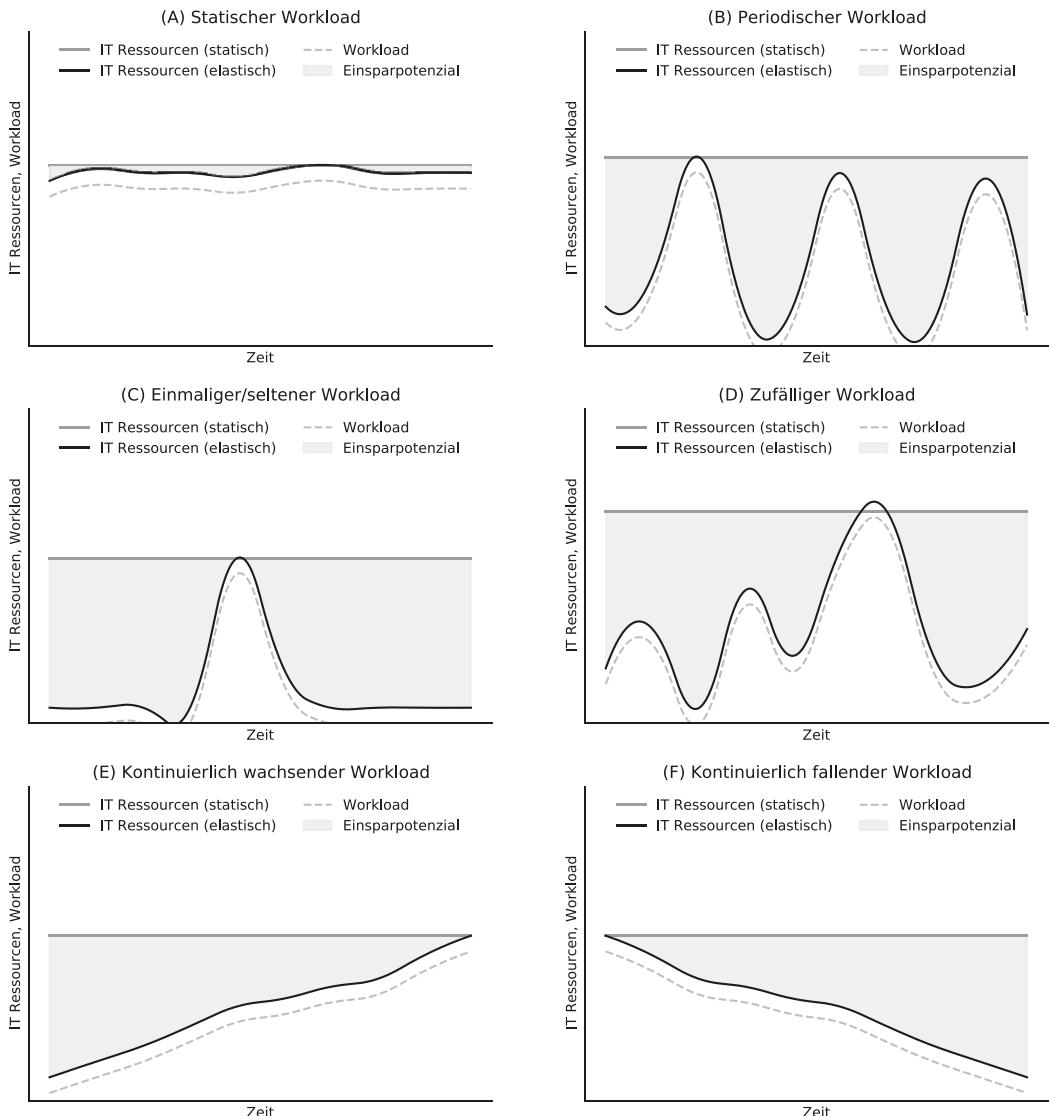


Bild 2.5 Zu berücksichtigende Workloads im Cloud Computing

- Statische Workloads (eher ungeeignet, siehe Bild 2.5 A)
- Kontinuierlich steigende/sinkende Workloads (siehe Bild 2.5 E/F)
- Zufällige und periodische Workloads (siehe Bild 2.5 B/D)
- Einmalige/seltene Workloads (extrem geeignet, Bild 2.5 C)

Für einen konkreten Anwendungsfall ist dieses $\frac{p}{a}$ natürlich immer genau zu bestimmen.

Dennoch hilft das Verständnis dieser grundsätzlichen Zusammenhänge erheblich dabei, überhaupt erst einmal interessante Anwendungsfälle zu identifizieren und uninteressante

Anwendungsfälle auszuschließen. Grundsätzlich ermöglicht die Elastizität von Cloud-Infrastrukturen und -Plattformen, Ressourcen mit der gleichen Rate bereitzustellen oder freizugeben, mit der sich die Arbeitslast eines Dienstes ändert, um diese Effekte für sich zu nutzen.

2.2.2 Effekt von Zuteilungsdauer und Ressourcengröße

Wie wir also sehen, sind Cloud-Ressourcen vor allem dann wirtschaftlich, wenn Lastschwankungen in einem Anwendungsfall auftreten. Die Kosten pro Cloud-Ressource können sogar deutlich höher als die In-House-Kosten liegen – solange das Verhältnis von Cloud zu In-House-Kosten nicht das Verhältnis von Spitzen- zu Durchschnittslast übersteigt.

Ziel ist also, im Betrieb eine möglichst niedrige Durchschnittslast zu ermöglichen (bzw. die Fläche zur Abdeckung der Lastkurve zu minimieren). Hierzu strebt man im Betrieb an, Lastkurven möglichst eng zu folgen. Kann man sich möglichst eng an Lastkurven „anschmiegen“, erzeugt dies wenig Over-Provisioning. Viele Innovationen des Cloud-native Computings wie beispielsweise Container- und FaaS-Technologien sind im Kern auf diese Erkenntnis zurückzuführen. Bei der Ressourcenzuteilung lässt sich dabei letztlich an zwei Stellschrauben drehen.

1. Man kann Ressourcen feingranularer zuteilen (vertikale Stellschraube).
2. Man kann Ressourcen kürzer zuteilen (horizontale Stellschraube).

Bild 2.6 zeigt den Effekt beider Stellschrauben (Ressourcengröße und Zuteilungsdauer) auf den Ressourcenverbrauch (und damit die Kosten) am Beispiel eines synthetischen periodischen Workload-Verlaufs.

Wie Bild 2.6 zeigt, ermöglichen es kleinere Ressourcengrößen und kürzere Zuteilungsdauern, Lastkurven enger folgen zu können. Damit kann das Over-Provisioning verringert werden. Dies spart letztlich Geld im Betrieb eines Cloud-nativen Systems. An dem – zugegeben synthetischen – Beispiel von Bild 2.6 zeigt sich dennoch, dass sich durch die Reduzierung von Ressourcengrößen und kürzere Zuteilungsdauern der rechnerische Ressourcenbedarf durchaus halbiert lässt. Dies ist natürlich immer von den dahinterliegenden Workload-Arten und dem Anwendungsfall abhängig. Auch noch größere Einsparungen sind nicht ungewöhnlich.

Diese einfache Erkenntnis hatte in den letzten Jahren einen tiefgreifenden Einfluss auf Cloud-native Architekturen und Technologien (Kratzke und Quint 2017). So konnte man in den vergangenen Jahren beobachten, wie diese beiden Stellschrauben (Zuteilungsdauer und Ressourcengröße) systematisch reduziert wurden. Während in der Anfangszeit des Cloud Computings virtuelle Maschinen üblicherweise auf Stundenbasis abgerechnet wurden, ist dies im Verlaufe der Zeit auf eine dreißigminütige, dann fünfzehnminütige bis schließlich zu einer minutengenaue oder mittlerweile sogar einer sekundengenaue Abrechnung bei vielen Providern umgestellt worden. Auch die Ressourcengröße wurde durch Technologien reduziert. Mittels IaaS kommt man nicht wirklich effizient unter die Auflösung von einer vCPU. Doch mittels der zunehmend beliebteren Container-Technologie sind wesentlich feingranularere Ressourcen möglich (siehe Kapitel 8), mit denen man problemlos unter diese 1 vCPU-Schwelle kommt. Auch die seit einigen Jahren beliebter werdende Technologie Function as a Service (FaaS, siehe Kapitel 10) kombiniert letztlich feingranularere Container mit einer Reduktion der zeitlichen Zuteilungsdauer im Subsekunden-Bereich. FaaS erlaubt es sogar, Ressourcen komplett auf null zu skalieren, wenn ein System in einem Zeitintervall

keine Aufgaben zu verarbeiten hat. Daran zeigt sich, dass viele Trendtechnologien zur fein-granulareren Ressourcenallokation im Cloud-native Umfeld ihren Grund auch immer in der innewohnenden Cloud-Ökonomie haben – auch wenn dies häufig nicht (mehr) bewusst wahrgenommen wird.

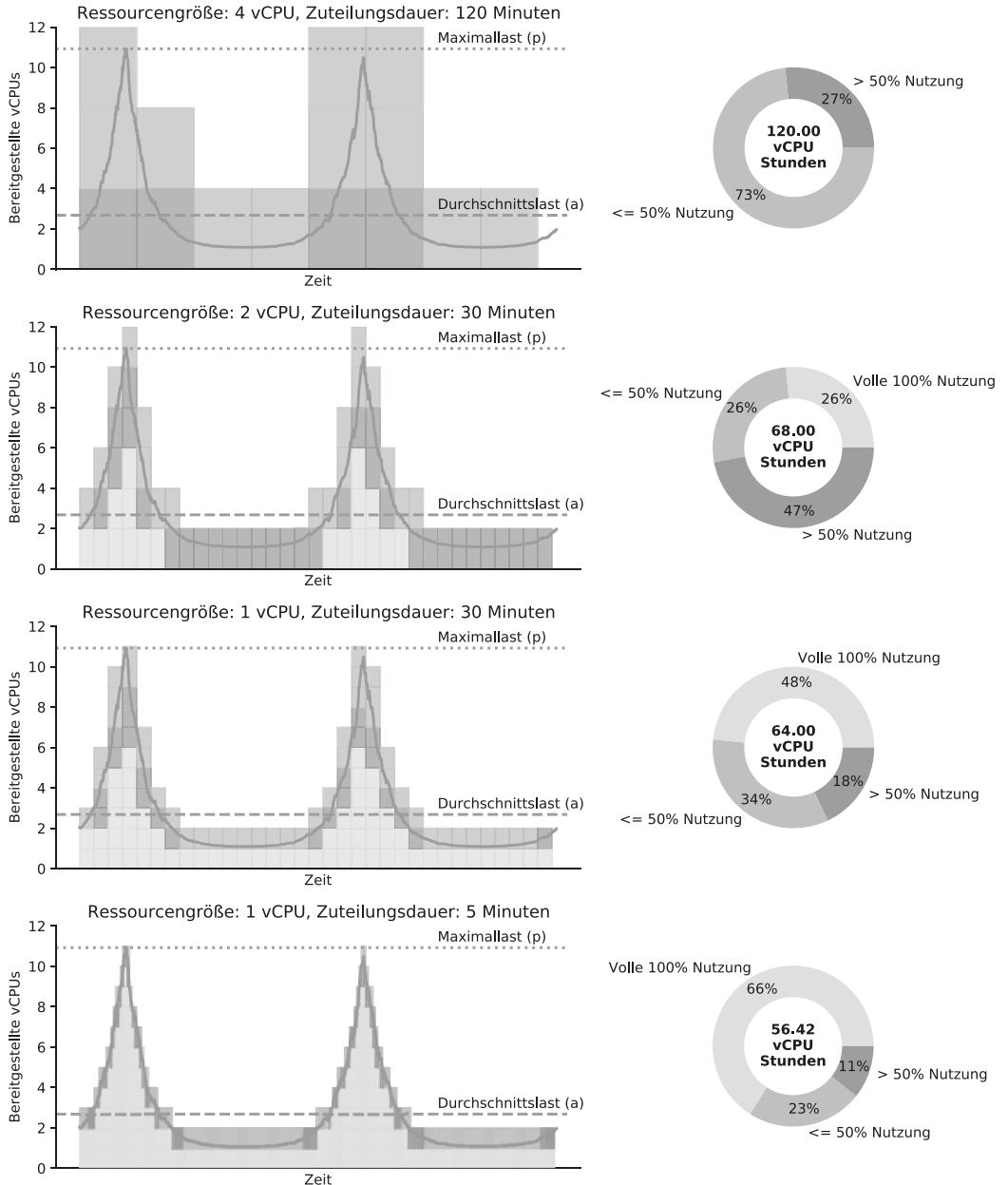


Bild 2.6 Effekt von Ressourcengröße und Zuteilungsdauer

■ 2.3 Entwicklung der letzten Jahre

Cloud Computing ist vor etwa zehn bis 15 Jahren entstanden. Dabei wurden in der ersten Adoptionsphase bestehende IT-Systeme lediglich in Cloud-Umgebungen übertragen, ohne das ursprüngliche Design und die Architektur dieser Anwendungen zu ändern. Multi-Tier-Anwendungen wurden lediglich von dedizierter Hardware auf virtualisierte Hardware in der Cloud migriert. Cloud-Systemingenieure haben im Laufe der Jahre allerdings bemerkenswerte Verbesserungen an Cloud-Plattformen (PaaS) und -Infrastrukturen (IaaS) vorgenommen und mehrere technische Trends etabliert, die derzeit zu beobachten sind. Ein wesentlicher Treiber hierfür sind die erläuterten ökonomischen Gesetzmäßigkeiten des Pay-per-use-Prinzips. Wer Cloud-native Systeme wirtschaftlich betreiben will, muss die Ressourcennutzung optimieren und minimieren.

Cloud-Infrastrukturen (IaaS) und -Plattformen (PaaS) sind daher insbesondere für den elastischen Betrieb von Cloud-nativen Anwendungen gebaut, um Over-Provisioning von Ressourcen zu vermeiden. Unter Elastizität versteht man den Grad, in dem sich ein System an Laständerungen anpasst, indem es automatisch Ressourcen bereitstellt und entnimmt. Ohne diese Elastizität ist Cloud Computing aus wirtschaftlicher Sicht sehr oft nicht sinnvoll. Mit der Zeit lernten Systemingenieure, diese Elastizitätsoptionen moderner Cloud-Umgebungen besser zu verstehen. Schließlich wurden Systeme für solche elastischen Cloud-Infrastrukturen von Grund auf entworfen, die dank neuer Deployment- und Design-Ansätze wie Container (siehe Kapitel 8), Microservices oder serverloser Architekturen (siehe Kapitel 12) den bereitzustellenden Ressourcenbedarf der zugrunde liegenden Computing-Infrastrukturen minimieren. Diese Designabsicht wird oft unbewusst mit dem Begriff „Cloud-native“ ausgedrückt.

Die Maschinenvirtualisierung hat sich insbesondere deshalb durchgesetzt, um eine Vielzahl von Bare-Metal-Maschinen zu konsolidieren und so die physischen Ressourcen in Rechenzentren effizienter nutzen zu können. Diese Maschinenvirtualisierung bildet bis heute das technologische Rückgrat des (IaaS-)Cloud Computings. Virtuelle Maschinen sind zwar leichtgewichtiger als Bare-Metal-Server, aber sie sind nicht unbedingt als leichtgewichtig zu bezeichnen, vor allem in Bezug auf ihre Image-Größen. Diese IaaS-Ebene wird vor allem in Kapitel 7 behandelt.

Vor diesem Hintergrund wurden leichtgewichtigere Container entwickelt. Container erlebten ihren Siegeszug primär, weil sie einerseits die Art und Weise der standardisierten Bereitstellung von Anwendungskomponenten vereinfachen. Container erhöhen aber auch die Auslastung der virtuellen Maschinen, da sie auf leichtgewichtigeren Betriebssystem-Virtualisierungskonzepten beruhen. Man kann also meist deutlich mehr Container auf einem physischen Host betreiben als virtuelle Maschinen. Wir werden uns mit diesen Aspekten vor allem in Kapitel 8 und in Kapitel 9 befassen. Dennoch sind Container, obwohl sie leichtgewichtig und schnell skalierbar sind, immer noch Always-on-Komponenten. Es muss also immer einen „letzten“ Container geben, der Requests bearbeiten kann. Zumaldest dieser „letzte“ Container fällt damit weiterhin in den Bereich eines statischen Workloads, also dem aus Kundensicht teuersten Workload für Cloud Computing.

Daher wurden Function-as-a-Service-(FaaS-)Ansätze entwickelt, die eine Art Time-Sharing von Containern auf darunterliegenden Container-Plattformen anwenden. Wir werden uns vor allem in Kapitel 10 mit diesen Aspekten befassen. Bei FaaS werden nur Einheiten (Funktionen) ausgeführt, die Requests zu bearbeiten haben. Durch diese zeitlich geteilte Ausführung von Containern auf der gleichen Hardware ermöglicht FaaS sogar eine Skalierbarkeit bis auf null. Studien konnten diese verbesserte FaaS-Ressourceneffizienz sogar monetär messen (Villamizar u. a. 2017). All dies hat letztlich mit der Minimierung der statischen Workload-Anteile zu tun, die den ineffektivsten Workload für Cloud Computing ausmachen.

Rückblickend betrachtet wurde der Technologie-Stack zur Verwaltung von Ressourcen in der Cloud also im Laufe der Zeit durch zusätzliche Ebenen (Virtualisierung, Container Runtime, FaaS Runtime) erweitert und damit immer komplexer. Das folgte aber einem grundsätzlichen Trend – mehr Workload auf der gleichen Anzahl physischer Maschinen auszuführen, also die Ressourceneffizienz insgesamt zu erhöhen.

3

DevOps

„It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change.“

Charles Darwin, Naturwissenschaftler und Begründer der Evolutionstheorie

Warum befassen wir uns an dieser Stelle zunächst mit DevOps und nicht gleich definitorisch mit Cloud-native Computing? Für ein Buch über Cloud-native Computing wäre dies schließlich der „direkte“ Weg. DevOps und Cloud-native Computing sind zwei eng verwandte Konzepte und in der modernen Softwareentwicklung weit verbreitet. Sie können zwar unabhängig voneinander existieren, werden aber häufig zusammen eingesetzt und ergänzen sich gegenseitig. DevOps bezieht sich auf eine Methode zur Verbesserung der Zusammenarbeit zwischen Entwicklungs- und Betriebsteams, um die Softwareentwicklung und -bereitstellung zu beschleunigen. DevOps umfasst eine Reihe von Praktiken wie Continuous Integration, Continuous Delivery und Continuous Deployment, um sicherzustellen, dass Änderungen an der Software schnell und fehlerfrei bereitgestellt werden können. Cloud-native Computing hingegen bezieht sich auf eine Architektur, die speziell für die Cloud entwickelt wurde und welche die Vorteile der Cloud nutzt.

DevOps und Cloud-native Computing verfolgen grundsätzlich ähnliche Ziele – die schnelle und zuverlässige Bereitstellung von Software. DevOps erleichtert dabei die Implementierung von Cloud-nativen Anwendungen, da es die agile und automatisierte Bereitstellung von Software fördert. Ebenso kann Cloud-native Computing DevOps unterstützen, indem es Entwicklern und Betriebsteams eine Plattform bietet, auf der DevOps-Praktiken und -Prozesse effektiv angewendet werden können. Um Cloud-native Computing zu verstehen, ist es daher hilfreich, zunächst die Treiber hinter DevOps und die daraus resultierenden Implikationen für Cloud-native Anwendungen und DevOps-fähige Architekturen zu verstehen.

Viele Unternehmen, z. B. Banken, datieren ihre IT häufig in Rhythmen (etwa einmal pro Quartal, pro Woche, pro Monat, pro Jahr) und nur in (schweren) Ausnahmefällen anlassbezogen auf. Die Dauer solcher Updates kann schon mal (z. B. für alle Bankautomaten der Deutschen Bank) mehrere Stunden oder sogar Tage dauern. Häufig legt man solche Updates in Zeiten, in denen wenig Publikumsverkehr erwartet wird, z. B. Sonntagmorgen um 04:00 Uhr. Bei Bankautomaten akzeptieren wir das meist. Häufig gibt es auch einen Automaten einer anderen Bank in der Nähe. Der Hintergrund dieser langen Zyklen (zum Vergleich: Amazon soll angeblich mehrere Hundert Updates pro Tag in seine Produktivsysteme einspielen) ist meist der nicht vollständig automatisierte Übergang von der Entwicklung (Dev) in den Betrieb (Ops). Dieser Übergang ist häufig von vielen manuellen und damit langwierigen Planungs-, Test-, Integrations- und Installationstätigkeiten geprägt.

Bei Online-Diensten wie beispielsweise Netflix (Video-Streaming), Amazon (Online-Ver sandhandel, Prime etc.) oder Google (z. B. Online-Navigation per Smartphone) ist jedoch die Akzeptanz für solche update-bedingten Downtimes heutzutage nicht mehr wirklich gegeben. Dies hat maßgeblich dazu beigetragen, dass die Entwicklung (Dev) und der Betrieb (Ops) mittlerweile mehr und mehr als Einheit gedacht und auch Architekturen auf die damit zusammenhängenden Erfordernisse konsequent ausgerichtet werden.

Die wenigsten Netflix-Kunden würden es beispielsweise akzeptieren, wenn sie ihre Lieblingsserie streamen möchten, aber stattdessen mit einer Fehlermeldung vertröstet würden, die da besagt, dass Netflix gerade aufdatiert wird und Streaming wieder ab morgen früh um 08:30 Uhr möglich ist. Bei Netflix mag dies ein „Luxusproblem“ sein. Problematischer wird es schon, wenn man zum nächsten Kundentermin mittels Google Maps navigieren möchte und die aktuelle Verkehrslage berücksichtigen muss, um auch pünktlich zu sein. Spätestens bei Katastrophenwarndiensten wie Nina oder Katwarn wird deutlich, dass Downtimes sogar „tödlich“ werden können. Katastrophen halten sich nämlich nicht an geplante Wartungsintervalle von Service-Anbietern, sondern passieren einfach unerwartet.

Lassen Sie uns diesen Hintergrund als das „**DevOps**“-Problem bezeichnen. Das DevOps-Problem haben nicht nur, aber insbesondere Service-Betreiber, die weltweit in allen Zeitzonen tätig sind, deren Dienstleistung überwiegend online erbracht wird und deren Dienste gegebenenfalls überproportional häufig außerhalb der üblichen Geschäftszeiten nachgefragt werden (Video-Streaming, nachts, abends und am Wochenende). Meist ist die Konkurrenz nur „einen Mausklick entfernt“ (Amazon Prime, Google Play, Netflix etc.). In diesem Setting findet man einfach keine geeigneten Zeitfenster mehr, um Services für Updates herunterfahren zu können. Man ist also gezwungen, seine Dienste im „Live-Betrieb“ zu aktualisieren. Damit dies überhaupt funktioniert und möglichst risikoarm abläuft, ist es erforderlich, die Entwicklung (Dev) und den Betrieb (Ops) von Diensten nicht mehr organisatorisch zu trennen, sondern als ein verschränktes Problem zu betrachten. Wenn man Dev und Ops trennt und nicht als Einheit betrachtet, wird man um Service-Downtimes für Updates vermutlich nicht herumkommen.

Unter DevOps ist also vor allem eine Kultur der Automatisierung und Verantwortung für den Betrieb zu verstehen, die kurze Releasezyklen, eine zuverlässige Inbetriebnahme durch risikoarme Releases und eine hohe Verfügbarkeit ermöglichen soll. Dies wird insbesondere durch eine verbesserte Kooperation zwischen Entwicklung und Betrieb mittels eines höheren Automatisierungsgrads erreicht.

Damit DevOps allerdings auch reibungslos funktioniert, müssen Softwaresysteme konsequent darauf ausgelegt werden. Dies umfasst u. a. die folgenden Ebenen:

- Kultur und Arbeitsorganisation
- Architektur
- Deployment-Pipeline
- Deployment Environments
- Telemetrydaten

Aus diesen grundsätzlichen DevOps-Überlegungen leiten sich viele implizite Anforderungen für Cloud-native Softwaresysteme und deren Betrieb ab. Diese werden im Weiteren betrachtet werden und mit den entsprechenden Kapiteln in Teil II und Teil III in Bezug gesetzt, in denen diese Erfordernisse aufgegriffen werden. Einen Überblick im Sinne eines grafischen Inhaltsverzeichnisses liefert auch Bild 3.1.

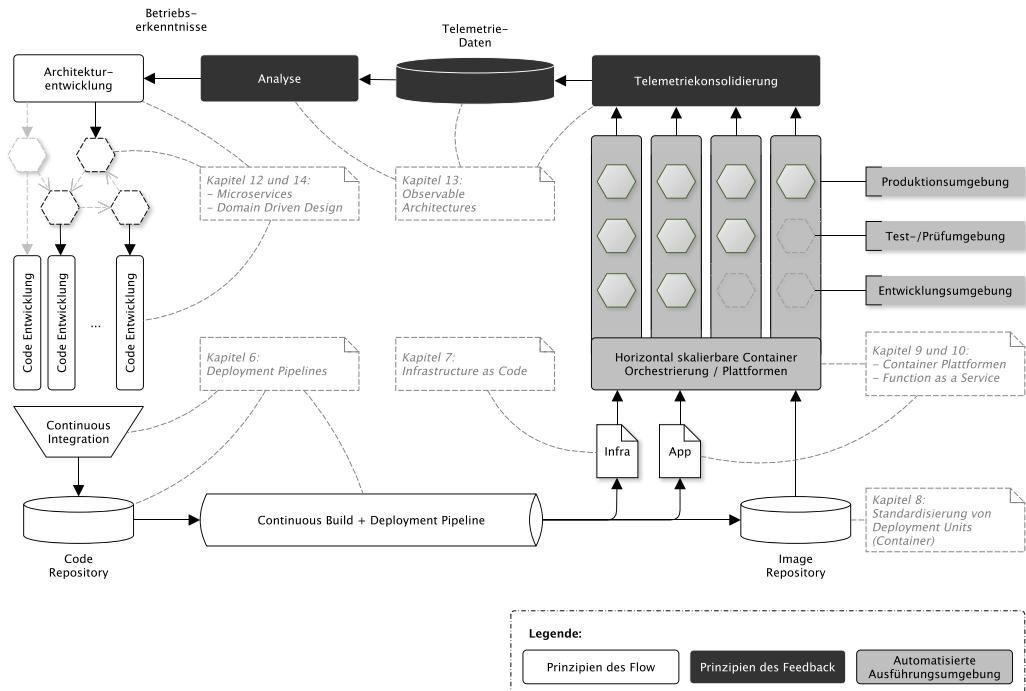


Bild 3.1 Der DevOps-Zyklus im Kontext Cloud-nativer Systeme

■ 3.1 Prinzipien des Flow

Um Softwareentwicklung möglichst reibungsfrei ablaufen zu lassen, muss man versuchen, einzelne Arbeitspakete möglichst reibungsarm „abarbeiten“ zu lassen. Hierzu empfehlen (Kim u. a. 2017), mehreren Prinzipien des Flow zu folgen. Lassen Sie uns diese im Einzelnen ansehen und insbesondere für den Teil II im Hinterkopf behalten.

3.1.1 Prinzip 1: Arbeit sichtbar machen

In der Informationstechnologie (anders als in der Produktion) kann Arbeit häufig per Mausklick bewegt werden. Weil das so einfach ist, werden manche Arbeitspakete zwischen Teams wie Pingpong hin und her bewegt.

Eine Möglichkeit, solche dysfunktionalen Prozessteile und Verhaltensweisen zu identifizieren, ist, den Fluss der Arbeit zu visualisieren. Hierzu wird häufig auf visuelle Arbeitsboards (z. B. Kanban-Boards) zurückgegriffen. Moderne Versionsverwaltungssysteme wie beispielsweise GitLab können solche Übersichten automatisiert erstellen (siehe Bild 3.2).

Arbeitspakete, die zwischen Phasen immer wieder hin und her wandern und nicht von links nach rechts „fließen“, zeigen Probleme auf, deren Ursachen man angehen sollte, da diese den Arbeitsfluss stören und „Blindleistung“ erzeugen.

The screenshot shows a GitLab Issue Board titled 'Development' with several columns: 'Reported', 'Reviewed', 'Planned', and 'In Progress'. Each column contains multiple issues, each with a title, description, and a set of labels. The 'Reported' column has issues like 'Do a better job of communicating when MR is blocked by a locked file.' and 'Usable to see user to add him to repositories'. The 'Reviewed' column has issues like 'Import project by URL form error hides the url field' and 'Contribution calendar label is out of date'. The 'Planned' column has issues like 'Wiki Page History appears to direct to wrong link and 404s' and 'Attachment is always "null" in Notes API'. The 'In Progress' column has issues like 'The buttons to resolve a discussion are malformed on Firefox under my Debian Stretch' and 'Do a better job of communicating when MR is blocked by a locked file.'. The interface includes a sidebar with navigation links like Overview, Repository, Issues, Boards, Labels, Service Desk, Milestones, Merge Requests, CI / CD, Snippets, and Settings.

Bild 3.2 Beispiel eines Issue Boards zum Workflow Tracking (Quelle: Gitlab)

Dabei sollte immer die komplette Wertkette einer Feature-Entwicklung abgebildet werden. Das bedeutet, von der ersten Idee, dem Einplanen (oder auch Verwerfen) und Priorisieren, der Entwicklung, dem Testen bis letztlich zum Release und zu dem Zeitpunkt, ab dem ein Feature tatsächlich produktiv läuft. An allen diesen Übergängen innerhalb der Wertschöpfungskette können nämlich Störungen entstehen, die man im Blick haben sollte, um Störungen frühzeitig zu identifizieren.

3.1.2 Prinzip 2: Work in Progress beschränken

Allgemein bekannt und akzeptiert ist, dass Unterbrechungen bei der Herstellung physischer Güter sehr kostenintensiv sind. Hier ist unmittelbar einsichtig, dass das Umrüsten von Maschinen Zeit kostet und Arbeitskraft bindet. Die „Unsichtbarkeit“ geistiger Tätigkeiten (wie beispielsweise in der Softwareentwicklung) lässt zu häufig diese „Umrüstkosten“ in nicht-industriellen Fertigungsprozessen aus dem Blick geraten. Studien haben gezeigt, dass sich die Zeit zum Erledigen selbst einfacher Aufgaben (wie dem Sortieren geometrischer Formen) beim Multitasking deutlich verlängert. Multitasking sollte daher reduziert werden, um zu vermeiden, dass zwischen zu vielen Schritten und Kontexten in der Softwareentwicklung hin und her gesprungen wird.

Denn auch in der Softwareentwicklung gibt es diese „Umrüstkosten“, die durch Multitasking verursacht werden, auch wenn diese vielleicht weniger offensichtlich sind als die „Umrüstkosten“ einer Maschine. Die Menge an umzusetzenden Features, die Entwickler zeitgleich „auf dem Schreibtisch“ liegen haben sollten, sollte begrenzt sein. Andernfalls ist die Gefahr zu groß, dass man zwischen zu vielen Aufgaben hin und her springen muss. Vor allem sollte

man vermeiden, Aufgaben ungeplant („Können Sie bitte mal schnell ...“) in die Entwicklung zu geben. Andernfalls läuft man Gefahr, Entwickler zu überlasten, die sich nur durch „taktische Fouls“ (Aufgaben-Pingpong, siehe Prinzip 1) retten können.

3.1.3 Prinzip 3: Flaschenhälse minimieren

Flaschenhälse sind in der Softwareentwicklung häufig durch manuelle Tätigkeiten begründet. Während die kreative Tätigkeit, Anforderungen in Programmcode zu übersetzen, meist (noch) nicht automatisiert werden kann, sind viele andere Tätigkeiten von Entwicklern durchaus automatisierbar und sollten daher auch automatisiert werden. Insbesondere der Automatisierungsgrad von Tätigkeiten wie

- der Erstellung von Test- und Produktivumgebungen,
- dem Testen von Änderungen
- oder dem Ausführen von Code-Deployments

sollten in einer DevOps-Kultur kontinuierlich steigen, um Flaschenhälse zu minimieren. Damit dies funktioniert, müssen allerdings auch Architekturen von Anfang an konsequent darauf ausgelegt werden.

Insbesondere sollte auf die langwierige (manuelle und damit fehleranfällige) Erstellung von Test- und Produktivumgebungen verzichtet werden. Stattdessen sollte die Erstellung von Produktiv- und Testumgebungen im Self-Service und auf Anforderung erfolgen und idealerweise automatisiert ausgeprägt werden. Insbesondere Infrastructure-as-Code-Ansätze bieten sich hier an, auf die noch im Detail in Kapitel 7 eingegangen werden wird.

Auch auf manuelles (und somit langwieriges und fehleranfälliges) Code-Deployment sollte verzichtet werden. Vielmehr sollten hierfür automatisiert ablaufende Deployment-Pipelines genutzt werden, die unter anderem in Kapitel 6 behandelt werden. Gleches gilt für langwieriges und manuelles Einrichten und Durchführen von Tests. Automatisiert ablaufende Deployment-Pipelines können so eingesetzt werden, dass auch Tests vor den eigentlichen Deployment Schritten automatisiert und parallelisiert ablaufen können, sodass die Testrate mit der Code-Entwicklungsrate mithält.

Damit Änderungen mittels solcher Deployment-Pipelines auch schnell und risikoarm deployt werden können, sollten ferner zu stark gekoppelte Architekturen vermieden werden. In solchen monolithischen Architekturen müssen häufig lokale Änderungen mit vielen Beteiligten abgestimmt werden. Dies ist häufig sehr zeitintensiv und reduziert dadurch den Entwicklungsdurchsatz. Lose gekoppelte Architekturen (z. B. Microservices, siehe Kapitel 12) haben das Ziel, Systeme so zu konzipieren, dass Änderungen nur lokale Auswirkungen haben und deswegen mit mehr Autonomie durchgeführt werden können. Dies reduziert den Abstimmungsbedarf bei Änderungen und erhöht somit den Entwicklungsdurchsatz.

■ 3.2 Prinzipien des Feedbacks

Um Probleme im laufenden Betrieb von Softwaresystemen möglichst frühzeitig zu erkennen, sollte man kontinuierlich Feedback einholen. Hierzu empfehlen (Kim u. a. 2017), mehreren Prinzipien des Feedbacks zu folgen, die insbesondere den Teil III motivieren.

3.2.1 Prinzip 4: Probleme früh erkennen

Komplexe Systeme sind nicht vollständig von einer einzelnen Person überschaubar und verstehtbar. Daher sollten Annahmen, die beim Design getroffen wurden, kontinuierlich geprüft werden. Dies geschieht z. B. mit Chaos Engineering, um Vertrauen in die Fähigkeit des Systems aufzubauen, Fehler- und Ausfallsituationen standzuhalten. Hierfür benötigt man zur Überprüfung u. a. Feedback-Schleifen im Produktivsystem, die kontinuierlich und automatisiert Telemetriedaten erheben.

Für Entwickler ist es ein Ziel, stabile, sichere und fehlerfreie Software zu entwickeln, die in Produktion zuverlässig läuft. Um dieses Ziel zu erreichen, werden üblicherweise Unit- und Integrationstests eingesetzt, die das erwartete Verhalten überprüfen und sicherstellen, dass die getesteten Ausfallmuster zu keinen Fehlerkaskaden führen. In Cloud-native Architekturen finden sich allerdings viele Komponenten, die variabel und austauschbar sind. Es kann durchaus unbekannte Services oder Komponenten geben, die es dennoch schaffen, das Gesamtsystem zum Ausfall zu bringen.

So hat u. a. das Streaming-Portal Netflix maßgeblich das Thema Chaos Engineering geprägt und viel zu seiner wachsenden Bedeutung beigetragen. Dabei wurde auch die Entwicklung von Chaos Monkey vorangetrieben. Diese Testkomponente hat die auf den ersten Blick widersinnige Aufgabe, laufende virtuelle Maschinen oder Container per Zufall zu terminieren. Dies hat einen disziplinierenden Effekt auf Softwarearchitekten und -entwickler. Das Wissen, dass die Instanzen jederzeit ausfallen können, führt bei Entwicklern, im Betrieb und beim Applikationsdesign zu einem Umdenken. Die Wahrscheinlichkeit, von einem Ausfall betroffen zu sein, erhöht sich durch Chaos Engineering deutlich und wird daher nicht mehr als „seltenes Ereignis“ abgetan, sondern wird durch den Einsatz passender Pattern aus dem Werkzeugkasten des Resilient Software Design proaktiv berücksichtigt (siehe auch Abschnitt 12.3).

Architekturen sollten hierfür vorbereitet und resilient sein. Ferner sollten Cloud-native Architekturen es ermöglichen, Telemetriedaten aus dem Betrieb zu erheben und für Analysen aufzubereiten.

3.2.2 Prinzip 5: Probleme sofort lösen

Dieses Prinzip hat seinen Ursprung vor allem in der japanischen Automobilindustrie. Jeder Mitarbeiter am Band hat die Pflicht, das Band zu stoppen, wenn er ein Problem erkennt. Dies soll es ermöglichen, alle vorhandenen Kräfte sofort zur Problemlösung einzusetzen und den Produktionsbetrieb schnellstmöglich wieder aufnehmen zu können.

Auf die Softwareentwicklung übertragen, bedeutet dies, dass Problembehebungen im Produktivsystem immer höhere Priorität als die weitere Entwicklung von Features bekommen sollten. Ergeben sich Probleme im Produktivsystem, sollte die Entwicklungspipeline gestoppt werden, und alle verfügbaren Entwicklungskräfte sollten sich auf die Problemlösung im Produktivsystem kümmern.

Probleme sollten nie umgangen oder deren Lösung verschoben werden, um zu vermeiden, dass sich ein Problem fortsetzt und zu exponentiell steigendem Aufwand für dessen Behebung zu einem späteren Zeitpunkt führt.

3.2.3 Prinzip 6: Probleme professionell verantworten

Erstaunlicherweise kann in komplexen Systemen das Hinzufügen zusätzlicher Kontrollschritte die Wahrscheinlichkeit für zukünftige Fehler erhöhen. Entscheidungen müssen dann nämlich von Personen getroffen werden, die weit weg vom Problemraum sind und sich in Problemkontexte erst einarbeiten müssen. Dies kann vor allem in zeitkritischen Fehlersituationen problematisch werden.

Die DevOps-Philosophie besagt daher, dass Entwickler auch Verantwortung für den operativen Betrieb haben („you build it, you run it“). Dadurch liegt die Verantwortung für Qualität, Zuverlässigkeit und Entscheidungsbefugnis dort, wo auch die Arbeit erledigt wird und die meiste technische Expertise vorhanden ist. DevOps-Teams haben daher meist eine höhere Autonomie, aber auch eine höhere Verantwortung als reine Entwicklungsteams, die die Verantwortung „über die Mauer“ zum Betrieb werfen können.

■ 3.3 DevOps-geeignete Architekturen

Erfahrungen haben gezeigt, dass Architekturen DevOps fördern oder hemmen können. Nach (Kim u. a. 2017) sollten DevOps-konforme Architekturen u. a.

- risikoarme Releases ermöglichen,
- sich evolutionär von Release zu Release weiterentwickeln lassen,
- aus lose gekoppelten autonomen Komponenten bestehen,
- die von autarken Teams betreut und betrieben werden können und
- kleine Batches von Entwicklungen zulassen.

Hierzu gibt es diverse Dinge, die sowohl im Rahmen der Entwicklung als auch im Betrieb zu berücksichtigen sind. Das „Scharnier“ zwischen Entwicklung und Betrieb bilden dabei zunehmend gut über Deployment-Pipelines steuerbare Orchestrierungsplattformen (siehe auch Bild 3.1).

3.3.1 Randbedingungen für die Entwicklung

Die Entwicklung sollte dabei konsequent Continuous Integration-Prinzipien folgen. Je länger Entwickler isoliert in Branches arbeiten, desto (exponentiell) schwieriger wird es, alle Änderungen wieder zusammenzuführen. Daher führt Continuous Integration die Entwicklungs-Branches einzelner Entwickler mittels trunk-basierter Entwicklungspraktiken (siehe auch Abschnitt 6.2.3) und kleinen Batchgrößen idealerweise täglich zusammen. Commits, die sich nicht automatisiert deployen lassen, sollten nicht in die Versionsverwaltung übernommen werden dürfen.

Eine Deployment-Pipeline stellt sicher, dass jeglicher Code, der in die Versionsverwaltung eingeccheckt wird, automatisch gebaut und in einer produktivähnlichen Umgebung mittels automatisierter Unit-, Akzeptanz-, Integrationstests und Post-Deployment-Checks getestet wird. So lassen sich Build-, Test- oder Integrationsfehler bereits beim Einführen einer Änderung erkennen und beheben. Das Ziel ist, dass resultierender Code sich immer in einem auslieferbaren Zustand befindet.

Zur Minimierung von Releaserisiken verfolgt man meist umgebungs- oder anwendungsbasierte Releasestrategien:

- Beim Blue/Green-Deployment hat man beispielsweise zwei Produktivreleases, doch nur eines bedient Kunden. Funktioniert ein Release nicht, kann notfalls auf das funktionierende Release zurückgeschaltet werden.
- Canary-Releases funktionieren ähnlich, nur werden Features erst wenigen und dann sukzessive immer mehr Nutzern bereitgestellt. Sollten Probleme auftreten, wird wieder zurückgerollt.
- Bei Rolling-Updates werden Deployment Units inkrementell aktualisiert.
- Feature-Schalter ermöglichen das selektive Ein-/Ausschalten von Features abhängig von Last oder sonstigen Erwägungen. Features können notfalls auch wieder abgeschaltet werden, wenn diese sich nicht bewähren.

3.3.2 Nutzung von Orchestrierungsplattformen

Mehr und mehr werden Orchestrierungsplattformen wie Kubernetes, Mesos, Swarm, Nomad und weitere als Ausführungsumgebungen genutzt. Ein Grund ist, dass diese Plattformen einen resilienten Betrieb mittels Self-Healing-Techniken ermöglichen und gut aus automatisierten Deployment-Pipelines gesteuert werden können.

Insbesondere Kubernetes ist eine Open-Source-Orchestrierungsplattform, die sich mittlerweile als De-facto-Standard für die Bereitstellung, Skalierung und Verwaltung von Container-Anwendungen etabliert hat. Solche Plattformen zielen darauf ab, das automatisierte Ausbringen, Skalieren und Warten von Anwendungcontainern auf verteilten Hosts (Clustern) zu ermöglichen. Der Vorteil von Clustern ist es, dass diese unabhängig von auf ihnen laufenden Diensten skaliert werden können. Man kann also die Skalierung der Infrastruktur von der Skalierung von Anwendungsdiensten trennen und voneinander unabhängig vornehmen.

Orchestrierungsplattformen können so Flaschenhälse im DevOps-Zyklus minimieren und Infrastructure as Code für Test- und Produktivumgebungen ermöglichen sowie die auto-

matisierte Erhebung von Telemetriedaten vereinfachen. Solche Orchestrierungsplattformen werden in Kapitel 9 behandelt werden.

3.3.3 Randbedingungen im Betrieb

Damit man aus dem Betrieb belastbare und idealerweise datengestützte Schlüsse für die weitere Entwicklung ziehen kann, ist es unter anderem erforderlich, Telemetriedaten zu erheben und DevOps-Teams bereitzustellen.

Hierzu wird meist eine zentrale Telemetrie-Infrastruktur genutzt, die verteilte Events, Logs und Metriken der Geschäftslogik und Infrastruktur in einem zentralen Service konsolidiert. Cloud-native Systeme und Anwendungen sind hierzu systematisch zu loggen und an die Telemetrie-Infrastruktur anzubinden. Jedes entwickelte Feature sollte hierzu geeignet instrumentiert werden.

Damit jeder im DevOps-Team Probleme auch finden und beheben kann, müssen gegebenenfalls anlassbezogen Metriken erstellt, generiert, angezeigt und analysiert werden können. Hierzu wird üblicherweise ein Self-Service-Zugriff auf Telemetriedaten innerhalb von DevOps-Teams gewährt.

Das Kapitel 13 wird sich im Rahmen überwachbarer Architekturen mit der Erhebung solcher Telemetriedaten befassen. Um Telemetriedaten zielgerichtet analysieren zu können, sind Methoden der Statistik von Vorteil. Häufig lassen sich Probleme bei normalverteilten Ereignissen mit einfachen, aber bewährten statistischen Verfahren wie Mittelwert und Standardabweichung identifizieren. Unterliegen Ereignisse keiner Normalverteilung, kann man auf glättende Verfahren (gleitende Durchschnitte), periodische/saisonale Verfahren (Kolmogorow-Smirnow etc.) zurückgreifen.

In automatisierten Überwachungsprozessen sollten automatisierte Warnungen nur ausgelöst werden, wenn sich Metriken statistisch signifikant von ihren Normal-/Erwartungswerten unterscheiden. Dies kann im Einzelfall durchaus komplex werden. Hierfür ist solides statistisches Wissen über die Verteilung entsprechender Ereignisse und statistischer Methoden von Vorteil. Auf Basis einer solchen Datenlage sind dann u. a. hypothesengetriebene Entwicklungsmethodiken möglich.

A/B-Tests sind beispielsweise eine Testmethode zur Bewertung von Varianten eines Systems, bei der die Originalversion gegen leicht veränderte Versionen getestet wird. Ziel ist es, Nutzerreaktionen einer Gruppe mit den Nutzerreaktionen einer Kontrollgruppe zu vergleichen. In modernen UX-Praktiken werden häufig Besuchern einer Website eine von zwei Versionen einer Seite angezeigt. Mittels einer statistischen Analyse des Folgeverhaltens (Conversion Rates) kann objektiv geprüft werden, ob eine Änderung einen Effekt im Nutzerverhalten gebracht hat oder nicht. Mithilfe von Feature-Schaltern lässt sich genau steuern, wie groß der Anteil von Kunden ist, die solch eine Testversion eines Experiments erhalten.

4

Cloud-native

“Man soll nie gescheiter sein als die Eingeborenen.”

Kofi Anan, Diplomat und Generalsekretär der UNO (1996–2006)

Laut Google Trends hat der Begriff „Cloud-native“ eine eher ungewöhnliche Verbreitungsrate im Laufe der Zeit (siehe Bild 4.1). In den Geburtszeiten des Cloud Computings – also so um 2005 bis 2007 herum – wurde der Begriff erstaunlicherweise recht häufig verwendet; auch wenn es rückblickend betrachtet irgendwie selbstverständlich erscheint, in den Anfangstagen des Cloud Computings etwas als „Cloud-native“ zu bezeichnen. So selbstverständlich war das aber offenbar nicht, denn die Verwendung des Begriffs „Cloud-native“ ließ in den folgenden Jahren wieder nach. Erst seit 2015 wird der Begriff wieder häufiger verwendet und hat substanziell an Dynamik gewonnen. Wir sollten allerdings bedenken, dass Google Trends die Tendenz haben, anfällig für „industrielle Buzzwords“ zu sein.

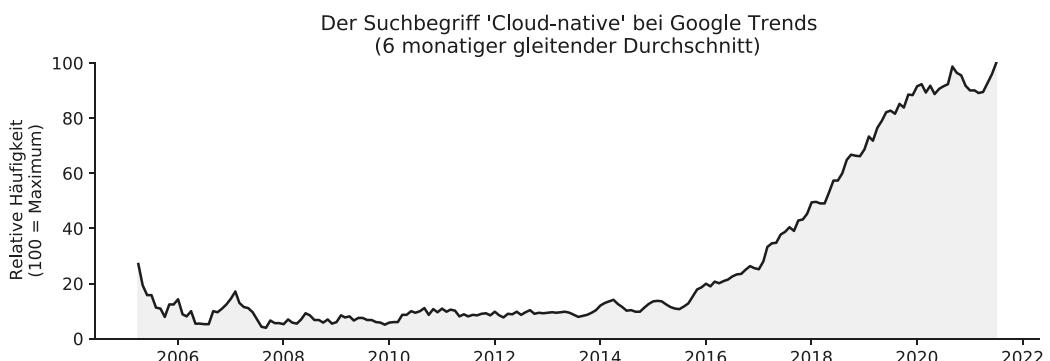


Bild 4.1 Der Begriff „Cloud-native“ in rückblickender Betrachtung

Aber auch Metastudien (Kratzke und Quint 2017), (Pahl u. a. 2019) zeigen eine sehr ähnliche Verwendung des Begriffs „Cloud-native“. Ferner legen diese Studien nahe, dass das aktuelle Verständnis des Begriffs „Cloud-native“ seinen Ursprung eher in der Forschung als in der Industrie zu haben scheint. Er wurde bereits seit 2012 systematisch in der Forschung verwendet, allerdings erst seit 2015 in der Industrie. Dort hat er allerdings seine heute bekannte Dynamik entwickelt. Dieser plötzliche Anstieg in der Industrie hat vermutlich viel mit der aktuellen Popularität von microservice- und container-basierten Ansätzen zu tun. Obwohl der Begriff also neu zu sein scheint, ist er es eigentlich nicht. Er war einfach vor 2015 nicht wirklich verbreitet und dann eher in Informatik-Forschungslaboren des Silicon Valley.

Die angesprochenen Metastudien liefern jedoch wertvolle Erkenntnisse darüber, welche zusätzlichen Eigenschaften eine Anwendung im Vergleich zu klassischen verteilten Legacy-Anwendungen erfüllen muss, um als „Cloud-native“ bezeichnet zu werden. Daher ist es an der Zeit, dass – nach fast 15 Jahren Cloud Computing – der Begriff „Cloud-native“ etwas präziser gefasst werden sollte, als der häufig zu hörende, aber letztlich wenig aussagende Definitionsansatz, dass „*Cloud-native Systeme Systeme sind, die bewusst für Cloud-basierte Laufzeitumgebungen entwickelt werden*“. Dieser Definitionsansatz ist in zahlreichen Varianten immer wieder zu hören, bewegt sich aber irgendwie auf dem Niveau, dass „*Autos für Autostraßen optimierte Fortbewegungsmittel sind*“. Das ist nicht wirklich hilfreich.

Auch wenn Literaturrecherchen letztlich keine einheitliche Definition ergeben, die zweifelsfrei klärt, was eine Cloud-native Anwendung (CNA) genau ist, können dennoch genügend Quellen identifiziert werden, aus denen sich ein Definitionsvorschlag für CNA ableiten lässt. Es gibt nämlich durchaus ein gemeinsames Verständnis über mehrere relevante Autoren und Stiftungen, wie beispielsweise der Cloud-native Computing Foundation (CNCF).



Cloud-native Computing in a Nutshell

Die CNCF definiert Cloud-native Computing als einen Ansatz für die Entwicklung und Bereitstellung von Anwendungen, der auf einer Kombination aus agilem Software-Entwicklungsprozess, DevOps-Praktiken und modernen sowie standardisierten Infrastruktur-Technologien basiert. Der Fokus liegt darauf, Anwendungen so zu gestalten und zu betreiben, dass diese für den Betrieb in der Cloud optimiert sind und die Vorteile der Cloud-Technologie systematisch nutzen.

Die wesentlichen Bausteine des Cloud-native Computing umfassen dabei:

1. **Containerisierung:** Container ermöglichen es Entwicklern, Anwendungen und ihre Abhängigkeiten in isolierte Umgebungen zu packen und sie auf jeder beliebigen Cloud-Plattform auszuführen. Container stellen sicher, dass Anwendungen unabhängig von der zugrunde liegenden Infrastruktur konsistent und zuverlässig arbeiten.
2. **Orchestrierung:** Orchestrierungstools wie Kubernetes ermöglichen es, Container-Cluster automatisch bereitzustellen, zu skalieren, zu überwachen und zu verwalten. Dadurch wird der Betrieb von Anwendungen in der Cloud effizienter und zuverlässiger.
3. **Microservices:** Cloud-native Anwendungen sind in der Regel aus kleineren, unabhängigen Services aufgebaut (Microservice-Architektur), die über Ressourcen-APIs kommunizieren. Dadurch können Anwendungen evolutionär entwickelt werden, bleiben aber flexibel und skalierbar.
4. **Infrastructure as Code (IaC):** Infrastructure as Code bedeutet, dass die gesamte Infrastruktur (z. B. Server, Netzwerke, Load Balancer) in Code definiert wird. Dadurch kann die Infrastruktur schnell und konsistent bereitgestellt und verwaltet werden.
5. **Automatisierung:** Durch die Automatisierung von Entwicklungs-, Test- und Bereitstellungsprozessen mittels Deployment-Pipelines können Teams schnellere und zuverlässigere Software-Updates bereitstellen.

■ 4.1 Definitionen in Industrie und Forschung

So nimmt die CNCF vor dem Hintergrund ihrer Mitglieder (u. a. Amazon Web Services, AT&T, CISCO, Google, IBM, Intel, Microsoft, Oracle, RedHat, SAP, VMWare) sicher einen eher industriellen Blickwinkel ein und definiert beispielsweise Cloud-native Technologien als „*Technologien, dies es Unternehmen ermöglichen, skalierbare Anwendungen in modernen, dynamischen Umgebungen wie Public, Private und Hybrid Clouds zu erstellen und auszuführen. Container, Service-Meshes, Microservices, unveränderliche Infrastrukturen (Immutable Infrastructures) und deklarative APIs sind Beispiele für diesen Ansatz*“.

Diese Techniken ermöglichen lose gekoppelte Systeme, die widerstandsfähig gegenüber Ausfällen, verwaltbar und beobachtbar sind. In Kombination mit einer robusten Automatisierung ermöglichen sie es, mit minimalem Aufwand häufige, beherrschbare, aber dennoch wirkungsvolle Änderungen an Cloud-nativen Systemen im laufenden Betrieb vornehmen zu können. Die CNCF hat dabei zum Ziel, die Akzeptanz des Cloud-nativen Paradigmas zu fördern, indem sie ein Ökosystem aus quelloffenen und herstellerneutralen Projekten unterstützt und pflegt, um Cloud-native Innovationen für jeden zugänglich zu machen.

Diese eher industriellen Gesichtspunkte werden durch akademisch orientierte Forschung durchaus gestützt. Fehling et al. schlagen vor, dass eine Cloud-native Anwendung dem **IDEAL**-Modell folgen sollte, d. h., sie sollte einen Isolierten Zustand haben, in ihrer Natur verteilt (**Distributed**) sein, mittels horizontaler Skalierung Elastisch sein, über ein Automatisiertes Managementsystem betrieben werden, und ihre Komponenten sollten Lose gekoppelt sein (Fehling u. a. 2014). Nach (Stine 2015) sind ferner gemeinsame Motivationen für Cloud-native Anwendungsarchitekturen zu berücksichtigen; so z. B. dass softwarebasierte Lösungen heutzutage wesentlich schneller in Produktion zu bringen sind (Entwicklungs geschwindigkeit), dass Systeme von Grund auf fehlerisolierend, fehlertolerant und Self-Healing konzipiert werden sollten (Sicherheit), dass zunehmend mehr horizontale (statt vertikale) Anwendungsskalierung ermöglicht werden muss (Skalierung) und schließlich eine große Vielfalt an (mobilen) Plattformen und Legacy-Systemen zu unterstützen ist (Client-Diversität).

Diese gemeinsamen Motivationen werden von verschiedenen Anwendungsarchitektur- und Infrastrukturansätzen adressiert (Balalaie, Heydarnoori, und Jamshidi 2016):

- Insbesondere Microservices betonen das Erfordernis der Dekomposition von monolithischen (Geschäfts-)Systemen in unabhängig voneinander aktualisierbaren Services (Namiot und Sneps-Sneppe 2014).
- Der Hauptmodus der Interaktion zwischen Services in einer Cloud-nativen Anwendungsarchitektur erfolgt über veröffentlichte und versionierte APIs (API-basierte Zusammenarbeit). Diese APIs sind oft HTTP-basiert und folgen oft einem REST-Stil mit JSON-Serialisierung. Es können aber auch andere Protokolle und Serialisierungsformate verwendet werden (Newman 2015).
- Einzelne Deployment-Einheiten der Architektur werden nach einer Sammlung von cloud-fokussierten Mustern wie beispielsweise dem Zwölf-Faktoren-Modell (Wiggins 2017) oder cloud-fokussierten Entwicklungs- und Architekturmustern (Fehling u. a. 2014), (Erl, Cope, und Naserpour 2015) entworfen und miteinander verbunden.

- Und schließlich werden agile Infrastrukturplattformen in einem Self-Service-Ansatz verwendet, um diese Microservices mittels in sich geschlossener Deployment Units (Container) bereitzustellen und zu betreiben. Diese Plattformen bieten zusätzliche Betriebsfunktionen auf IaaS-Infrastrukturen, wie z. B. automatische und bedarfsgerechte Skalierung von Anwendungsinstanzen, Health-Management, dynamisches Routing, Lastausgleich und Aggregation von Protokollen und Metriken (Mendonca u. a. 2021).

■ 4.2 Die Cloud-native-Definition dieses Buchs

Diese Aspekte führen zu folgendem Verständnis Cloud-nativer Anwendungen:



Eine Cloud-native Anwendung (CNA) ist ein verteiltes, beobachtbares, elastisches und auf horizontale Skalierbarkeit optimiertes Service-of-Services-System, das seinen Zustand in (einem Minimum an) zustandsbehafteten Komponenten isoliert. Die Anwendung und jede in sich geschlossene Bereitstellungseinheit dieser Anwendung wird nach Cloud-fokussierten Designmustern entworfen und auf elastischen Self-Service-Plattformen betrieben.

Natürlich ist diese Definition nur im Kontext weiterer Begriffe zu verstehen, die bereits von zahlreichen Autoren und Standardisierungsinitiativen definiert und geprägt wurden.

Beobachtbarkeit bei Softwaresystemen bezieht sich typischerweise auf Telemetriedaten, die meist in drei Aspekte unterteilt werden: Tracing (verteilte Ablaufverfolgung) ermöglicht Einblick in den Lebenszyklus von Requests in einem verteilten System. Metriken im Rahmen eines Monitorings liefern quantitative Informationen zu Prozessen, die im System ausgeführt werden. Mittels Logging (Protokollierung) lässt sich Einblick in anwendungsspezifische Nachrichten, die von Prozessen ausgegeben werden, gewinnen. Diese Beobachtbarkeit ist insbesondere für die DevOps-Prinzipien des Feedbacks essenziell (siehe Abschnitt 3.2).

Elastizität ist „*der Grad, in dem ein System in der Lage ist, sich an Workload-Änderungen anzupassen, indem es Ressourcen in einer autonomen Art und Weise provisioniert und deprovisioniert, sodass zu jedem Zeitpunkt die verfügbaren Ressourcen so gut wie möglich mit dem aktuellen Bedarf übereinstimmen*“ (Herbst, Kounev, und Reussner 2013).

Skalierbarkeit kann in strukturelle Skalierbarkeit und Lastskalierbarkeit unterschieden werden. „*Strukturelle Skalierbarkeit ist die Fähigkeit eines Systems, sich in einer gewählten Dimension ohne größere Änderungen an seiner Architektur zu erweitern. Unter Lastskalierbarkeit ist die Fähigkeit eines Systems zu verstehen, auch steigenden Datenverkehr bewältigen zu können*“ (Bondi 2000).

Service-of-Services-Systeme werden bei Cloud-nativen Anwendungen zumeist im Microservice-Architekturstil verstanden. Dieser Architekturstil versteht „*eine einzelne Anwendung als eine Suite kleiner Services, die jeweils in einem eigenen Prozess laufen und mit leichtgewichtigen Mechanismen kommunizieren. Diese Services sind um Geschäftsfunktionen herum aufgebaut*

und können unabhängig voneinander von vollautomatischen Bereitstellungsmechanismen aktualisiert werden. Es gibt nur ein Minimum an zentraler Verwaltung dieser Dienste, die in verschiedenen Programmiersprachen (polyglotte Programmierung) geschrieben sein können und unterschiedliche Datenspeichertechnologien verwenden (polyglotte Datenhaltung)" (Fowler 2014).

In sich geschlossene Bereitstellungseinheiten (Deployment Unit) sind „ein Teil der Deployment-Topologie der Anwendung zur Realisierung einer bestimmten technischen Einheit“ (Inzinger u. a. 2014). Immer häufiger wird eine Deployment Unit als „ein Standardcontainer verstanden. Das Ziel eines Standardcontainers ist es, eine Softwarekomponente und alle ihre Abhängigkeiten in einem Format zu kapseln, das selbstbeschreibend und portabel ist, sodass jede konforme Laufzeitumgebung sie ohne zusätzliche Abhängigkeiten ausführen kann, unabhängig von der zugrunde liegenden Maschine und dem Inhalt des Containers“ (OCI 2015).

Zustandsbehaftete Komponenten (meist Datenbanken) werden für „mehrere Instanzen einer skalierten Anwendungskomponente verwendet, die ihren internen Zustand synchronisieren, um ein einheitliches Verhalten zu bieten“ (Fehling u. a. 2014). Da die Skalierung zustandsbehafteter Komponenten meist aufwendiger ist als die Skalierung zustandsloser Komponenten, versucht man, Zustände in möglichst wenigen zustandsbehafteten Komponenten zu isolieren.

Unter einer elastischen Plattform versteht man eine „Middleware für die Ausführung von benutzerdefinierten Anwendungen, deren Kommunikation und Datenspeicherung über eine Self-Service-Schnittstelle mittels eines Netzwerks angeboten wird“ (Fehling u. a. 2014). Solche gut automatisierbaren Plattformen sind insbesondere für die DevOps-Prinzipien des Flow essenziell (siehe Abschnitt 3.1).

■ 4.3 Zusammenfassung und Ausblick auf Teil II bis IV

In Kapitel 2 wurde insbesondere auf die drei prägenden Service-Modelle des Cloud Computings (IaaS, PaaS, SaaS) eingegangen. Dieses Buch wird dabei insbesondere fokussieren, wie Cloud-native Anwendungen auf Basis von IaaS und PaaS entwickelt werden können (um selbst SaaS-Dienste und Anwendungen bereitzustellen zu können). Cloud Computing ist vor allem durch die Virtualisierung von Infrastruktur an seiner Basis geprägt, die zu einem Infrastructure-as-Code-Ansatz geführt hat, der in Kapitel 7 (Infrastructure as Code) in Teil II aufgegriffen wird. Insbesondere wurde dabei auch der ökonomische Effekt von Zuteilungsdauer und Ressourcengröße behandelt. Cloud Computing ist aus Kundensicht umso wirtschaftlicher, je höher die Schwankungen an Ressourcenbedarf und je kürzer die Dauer der Ressourcenzuteilung ist. Mit sehr kleinen und kurz zugeteilten Ressourcen kann man Lastkurven enger folgen und hat weniger Over Provisioning. Diese rein wirtschaftliche Erwägung hat zu einem Trend immer kleinerer und kürzer zuteilbarer Ressourcen im Cloud-native Computing geführt. Diese „Schrumpfung“ von Ressourcen wird insbesondere in Kapitel 8 (Containerisierung) sowie Kapitel 10 (Function as a Service) in Teil II aufgegriffen werden.

Kapitel 3 hat hingegen gezeigt, dass der Betrieb Cloud-nativer Anwendungen und Systeme heutzutage durch anspruchsvolle 24x7-Anforderungen geprägt ist. Man findet einfach keine Wartungs- und Aktualisierungszeitfenster mehr, die es ermöglichen, Systeme im Betrieb zu Wartungszwecken herunterzufahren, zu aktualisieren und wieder hochzufahren. Unter anderem vor diesem Hintergrund ist die DevOps-Philosophie entstanden, die besagt, dass Systeme evolutionär und in kleinen Batches entwickelt werden sollten, um risikoarme Releases zu ermöglichen. Hierzu ist es erforderlich, dass Entwickler möglichst reibungsarm arbeiten können und insbesondere das Deployment von Änderungen automatisiert erfolgen kann. Wie sich insbesondere die DevOps-Prinzipien des Flow dabei durch Deployment-Pipelines und automatisierte Betriebsumgebungen technisch unterstützen lassen, werden in Teil II das Kapitel 6 (Deployment-Pipelines) und das Kapitel 9 (Orchestration) zeigen. DevOps ist aber ein rückgekoppelter Ansatz, d. h., man sammelt Erfahrung auch während des Betriebs, um diese evolutionär in die Weiterentwicklung eines Systems einfließen lassen zu können. Damit diese Rückkopplung funktioniert, ist es erforderlich, Systeme im Betrieb beobachten zu können, um überhaupt eine Datenlage erheben zu können, die es ermöglicht, solche Rückschlüsse ziehen zu können. Wie dies realisierbar ist, wird in Teil III vor allem das Kapitel 13 (Beobachtbare Architekturen) zeigen. Für diese evolutionäre und rückgekoppelte Herangehensweise müssen Systeme aber auch architekturell vorbereitet sein. Diese architekturellen Aspekte werden in Teil III vor allem in Kapitel 12 (Microservices und Serverless Architectures) und Kapitel 14 (Domain Driven Design) behandelt werden.

Teil IV hingegen geht in Kapitel 16 darauf ein, wie man sichere Cloud-native Systeme entwickelt und auf welchen Ebenen Härtungsmaßnahmen sinnvoll erscheinen. Kapitel 17 geht im Kontext der IT-Sicherheit auf regulatorische Herausforderungen ein, die sich unter anderem im europäischen Rechtsraum aus der Datenschutz-Grundverordnung (DSGVO) bzw. der General Data Protection Regulation (GDPR) ergeben.

Insgesamt ergibt sich also – wie in Kapitel 4 gezeigt – eine äußerst komplexe Themenlage, die sich um den Begriff „Cloud-native“ in Industrie und Forschung entwickelt hat. Dieses Buch folgt dabei der in Abschnitt 4.2 vorgeschlagenen Definition des „Cloud-native“-Begriffs.

Sollte der Leser Interesse zeigen, die in Teil I behandelten Themen zu vertiefen, sei der Einstieg in die jeweilige Thematik mittels folgender Quellen empfohlen. Die Standardquelle zur Terminologie und zu den Service-Modellen des Cloud Computings IaaS, PaaS und SaaS ist und bleibt (Mell und Grance 2011). Die Darstellungen von (Weinman 2011) liefern diverse aufschlussreiche Erkenntnisse zu ökonomischen Gesetzmäßigkeiten des Cloud Computings. Dies erfolgt sowohl aus dem Blickwinkel eines Kunden als auch eines Service-Providers. Die in Abschnitt 2.3 behandelten Workloads sowie viele weitere Cloud Pattern werden umfassend in (Fehling u. a. 2014) behandelt. Das Thema DevOps wird umfassend durch (Forsgren-Velasquez u. a. 2014) aufbereitet und auch kontinuierlich im Rahmen von jährlichen DevOps-Studien fortgeschrieben. Einen Überblick über diese Studien liefert (Kim u. a. 2017). Die Begrifflichkeit Cloud-native wird mittlerweile maßgeblich durch (CNCF 2015) geprägt. Interessant mögen jedoch für den Leser auch die Überblicksstudien von (Kratzke und Quint 2017) und (Kratzke 2018) sein.

Ergänzende Materialien



<https://bit.ly/3mcJVqX>

Auf der Website zum Buch finden sich zu diesem Kapitel folgende Ergänzungsmaterialien:

- Praktische Übungen (Labs) zum Thema Workloads und zu resultierenden ökonomischen Konsequenzen; u. a. wurden die Abbildungen aus Abschnitt 2.3.1 mit einem interaktiven Tool dieses Labs generiert.
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer (Themen: Was ist Cloud Computing? Was sind Cloud-native Systeme? Crashkurs Cloud-Ökonomie, eine kurze Geschichte der Cloud, DevOps)



Teil II:

Everything as Code

5

Einleitung zu Teil II

„Software is eating the world.“

Marc Andreessen, Mitgründer von Netscape und Entwickler von Mosaic

In Teil II geht es um das „Handwerk“ der Cloud-nativen Systementwicklung, also der Cloud-nativen Programmierung. Im Vergleich zur „normalen“ Programmierung, d. h. der Anwendungsentwicklung mit einer klassischen Programmiersprache wie beispielsweise Java, Python oder C, betrachten wir hier die Programmierung primär zum Zwecke der Automatisierung von Cloud-nativen Infrastrukturelementen oder ressourceneffizienten Bereitstellung von Diensten. Der Ansatz des „Everything as Code“ hat zum Ziel, alle erforderlichen Ressourcen Cloud-nativer Systeme über alle Systemebenen (physische und virtualisierte Infrastruktur, Plattform, Applikation) automatisiert erstellen und verwalten zu können.

Kapitel 6 befasst sich dabei mit Deployment-Pipelines. Deployment-Pipelines bilden das Herzstück der Automatisierung in DevOps-Ansätze (siehe Abschnitt 3.1.3) und werden u. a. im Rahmen der Bereitstellung von Infrastruktur, dem Bau von Komponenten, deren Test und Integration zu Systemen sowie dem automatisierten Deployment von Cloud-nativen Systemen eingesetzt.

Kapitel 7 betrachtet softwaredefinierte Infrastrukturen, die unter anderem aus Deployment-Pipelines (siehe Kapitel 6) heraus vollautomatisch ausgeprägt werden können. Dieser – auch als Infrastructure as Code (IaC) – bekannte Ansatz ermöglicht es Betriebsteams, den Technologie-Stack für eine Anwendung automatisch über Software verwalten und bereitzustellen zu können, um so manuelle Prozesse zur Konfiguration physischer oder virtualisierter Hardware und Software zu minimieren oder gar ganz zu vermeiden. Dies trägt zur Minimierung von DevOps-Flaschenhälsern bei (siehe Kapitel 3).

Kapitel 8 befasst sich mit der Betriebssystem- bzw. Containervirtualisierung. Dies ist eine Methode, um mehrere Prozesse eines Betriebssystems isoliert voneinander den Kernel eines Hostsystems nutzen zu lassen, und ist daher deutlich ressourcenschonender als Hardwarevirtualisierung mittels Hypervisoren (siehe Kapitel 7). Im Cloud-nativen Umfeld haben sich Container jedoch auch als ein probates Mittel erwiesen, Komponenten Cloud-nativer Systeme in Form von Container-Images als standardisierte und Self-contained Deployment Units bereitzustellen zu können. Der eigentliche Wert von Containern liegt deswegen weniger in der Betriebssystemvirtualisierung an sich als in der Tatsache, dass Container viele Probleme des Dependency-Managements sehr pragmatisch und einheitlich lösen, indem Self-contained Deployment Units für unterschiedlichste Anwendungszwecke bereitgestellt werden können. Insbesondere die Container Runtime Environment Docker hat hier in den letzten Jahren eine wahre Renaissance der Betriebssystemvirtualisierung ausgelöst.

Kapitel 9 befasst sich mit der Orchestrierung von Cloud-nativen Anwendungen. In modernen Entwicklungsprozessen werden solche Anwendungen nicht mehr monolithisch gestaltet, sondern setzen sich vielmehr aus einer Vielzahl von lose miteinander gekoppelten Komponenten zusammen. Diese müssen zusammenarbeiten, damit eine Cloud-native Anwendung ihre Dienste bereitstellen kann. Mittels Container-Orchestrierung wird die Bereitstellung, Verwaltung, Skalierung und Vernetzung von Containern automatisiert.

Kapitel 10 betrachtet Function as a Service (FaaS). Mit FaaS-Plattformen können sehr feingranulare Anwendungsfunktionen entwickelt, ausgeführt, verwaltet und lastabhängig skaliert werden, ohne die Komplexität des Aufbaus und der Wartung der Infrastruktur zu haben, die mit der Entwicklung und dem Betrieb von Cloud-nativen Systemen häufig verbunden ist. Das Erstellen einer Anwendung nach diesem Modell ist eine Möglichkeit, eine sogenannte „Serverless“-Architektur zu erreichen. Auf derartige und weitere Cloud-native Architekturstile wird vertiefend in Teil III eingegangen werden.

6

Deployment-Pipelines

„Suche nicht nach Fehlern, suche nach Lösungen.“

Henry Ford, Gründer der Ford Motor Company und Erfinder des Fließbands

Deployment-Pipelines sind ein wesentlicher Baustein im DevOps-Ansatz, um Entwicklungszyklen schnell und agil zu halten. Ziel ist es, Code, der in ein Code Repository eingebracht wird, möglichst automatisiert zu integrieren, bauen, testen sowie gegebenenfalls in eine Umgebung (häufig Test, Staging, Production) auszubringen. Mit jedem Code Push wird so automatisiert geprüft, ob der Code in die bestehende Codebasis integriert werden kann, kompilierbar ist, alle Tests passiert und deploybar ist. Auf diese Weise können nur funktionierende Softwarezustände in funktionierende Softwarezustände überführt werden. Entwicklern wird es so technisch und ablauforganisatorisch verwehrt, Code zu erzeugen, der nicht automatisiert durch die Deployment-Pipeline verarbeitbar ist. Es gibt diverse solcher Managed oder Self-hosted Continuous Integration und Continuous Deployment CI/CD Services, die als kommerzielle oder auch als Open-Source-Software genutzt werden können.

Allen diesen Systemen ist gemein, dass eine Deployment-Pipeline aus einer Sequenz von Phasen besteht. Jede Phase kann wiederum ein oder mehrere Jobs haben. Alle Jobs innerhalb einer Phase werden parallel und isoliert voneinander ausgeführt. Eine Phase wird nur dann ausgeführt, wenn alle Jobs der vorherigen Phase erfolgreich ausgeführt werden konnten. Grundsätzlich können Pipelines beliebig aussehen, es bietet sich jedoch an, bewährten Pipeline Blueprints zu folgen:

- Build-Phase (zum Erzeugen von Executables oder Deployment Units wie beispielsweise Container)
- Test-Phase (zum Testen von Executables oder Deployment Units)
- Deploy-Phase (zum Ausbringen von Executables oder Deployment Units)

Die Prinzipien von Deployment-Pipelines und deren Zusammenspiel mit Versionskontrollsystmen werden am Beispiel von GitLab CI/CD demonstriert, da die Pipeline-Definition bei diesem selbst hostbaren System mittels sehr anschaulicher und nachvollziehbarer YAML-Dateien erfolgt. In GitLab CI/CD wird eine Pipeline per Konvention in der Datei `.gitlab-ci.yml` definiert. Die in diesem Kapitel gezeigten und am Beispiel von GitLab CI/CD gezeigten Prinzipien lassen sich aber problemlos auf andere Build- und Deployment-Systeme übertragen – auch wenn sich die Terminologie von System zu System etwas unterscheiden mag. Der Leser möge GitLab CI/CD daher als für die Vermittlung geeigneten Typräsentant und nicht als Produktempfehlung verstehen.

■ 6.1 Deployment-Pipelines as Code

Pipelines ermöglichen die kontinuierliche Integration und Bereitstellung von Software und werden zumeist aus folgenden (oder ähnlich bezeichneten) Konzepten gebildet:

- Jobs definieren, welche Einzelschritte zu tun sind. Jobs können beispielsweise Code kompilieren oder testen.
- Phasen legen fest, in welcher Abfolge Jobs ausgeführt werden sollen. Zum Beispiel muss Code erst kompiliert werden, bevor er getestet werden kann.

Jobs werden von einer Deployment-Infrastruktur ausgeführt. Mehrere Jobs in derselben Phase werden parallel ausgeführt, wenn es genügend Ausführungsressourcen der Deployment-Infrastruktur gibt, um dies zu ermöglichen. Zunehmend häufiger machen sich Deployment-Systeme selbst die noch in Kapitel 8 beschriebenen Container-Technologien zunutze und führen ihrerseits Jobs wiederum in Containern aus. Bei diesen Systemen können Jobs dann einfach als Liste von Shell-Anweisungen (also z. B. Bash-Skripte) ausgedrückt werden. Ob ein Job erfolgreich ist (und damit die Pipeline), ergibt sich aus dem Exit-Code des Prozesses.

- Ein Exit-Code von 0 bedeutet, dass ein Job erfolgreich ist und die Pipeline fortgesetzt werden kann.
- Ein Exit-Code ungleich 0 bedeutet, dass ein Job fehlerhaft ist und die weitere Bearbeitung der Pipeline abzubrechen ist.

Im Allgemeinen werden Pipelines automatisch ausgeführt und erfordern nach ihrer Erstellung keinen Eingriff. Üblicherweise werden Pipelines durch einen der folgende Trigger gestartet:

- Code Commit in ein Versionskontrollsysteem
- Zeitgesteuerte periodische Trigger (z. B. nightly-builds)
- Einzelne Jobs lassen sich jedoch auch manuell in einer Pipeline starten (z. B. Deinstallationsprozesse).

Es hat sich zudem im Sinne der Everything-as-Code-Philosophie durchgesetzt, auch Deployment-Pipelines deskriptiv mittels textbasierter Formate wie beispielsweise YAML zu beschreiben. Somit lassen sich also auch die Deployment-Pipelines selber (wie anderer Quelltext) als Code beschreiben und danach problemlos in Versionsverwaltungssystemen versionieren. Dies ermöglicht flexible und unaufwendig automatisierte Deployment-Stra tegien. Da die Pipeline-Definition zumeist Teil des Projekt-Repositorys ist, vereinfacht dies die Analyse von Abhängigkeiten und Zusammenhängen in DevOps-Szenarien erheblich.

Pipelines lassen sich beliebig komplex strukturieren. Die folgenden Pipeline-Pattern (siehe Abschnitte 6.1.1 bis 6.1.3) bilden jedoch zumeist ein gliederndes Grundgerüst. Jedes Pattern hat dabei seine eigenen Vor- und Nachteile. Abhängig von der jeweiligen Deployment-Infrastruktur können die Pattern bei Bedarf auch kombiniert werden (z. B. bei GitLab CI/CD).

6.1.1 Phasen-Pipelines

Phasen-Pipelines sind für einfache Projekte geeignet, bei denen die gesamte Konfiguration an einem leicht zu findenden Ort vorgehalten wird. Bei Phasen-Pipelines fährt die Pipeline mit der nächsten Phase fort, wenn alle Jobs einer vorhergehenden Phase erfolgreich sind. Wenn ein Job in einer Phase fehlschlägt, wird die nächste Phase nicht ausgeführt, und die Pipeline wird vorzeitig beendet.

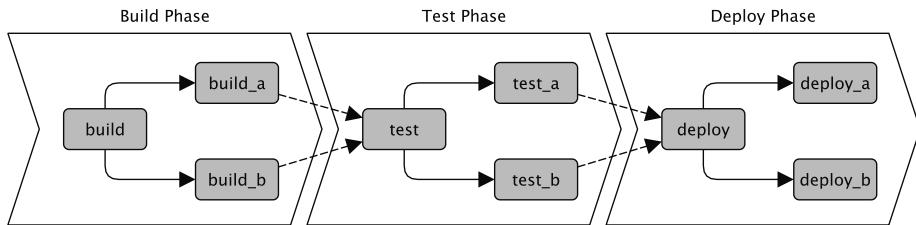


Bild 6.1 Phasen-Pipelines

Das Listing 6.1 zeigt die Pipeline aus Bild 6.1 am Beispiel der Deployment-Infrastruktur GitLab CI/CD.

Listing 6.1 Einfache Phasen-Pipeline-Definition am Beispiel von GitLab CI/CD

```

stages:      # Definition des Phasenablaufs
  - build
  - test
  - deploy

image: alpine

build_a:      # Definition eines Jobs
  stage: build # in der Build-Phase
  script:
    - echo "This job builds something."

build_b:      # Definition eines Jobs
  stage: build # in der Build-Phase
  script:
    - echo "This job builds something else."

test_a:       # Definition eines Jobs
  stage: test  # in der Test-Phase
  script:
    - echo "This job tests something."
    - echo "It will only run when all jobs in the"
    - echo "build stage are complete."

test_b:       # Definition eines Jobs
  stage: test  # in der Test-Phase
  script:
    - echo "This job tests something else."
    - echo "It will only run when all jobs in the"

```

```

- echo "build stage are complete too."
- echo "It will start at about the same time as test_a."

deploy_a:      # Definition eines Jobs
stage: deploy # in der Deploy-Phase
script:
- echo "This job deploys something."
- echo "It will only run when all jobs in the"
- echo "test stage complete."

deploy_b:      # Definition eines Jobs
stage: deploy # in der Deploy-Phase
script:
- echo "This job deploys something else."
- echo "It will only run when all jobs in the"
- echo "test stage complete."
- echo "It will start at about the same time as deploy_a."

```

6.1.2 Gerichtete Pipelines

Gerichtete Pipelines oder DAG Pipelines (DAG = Directed Acyclic Graph) sind für größere und komplexere Projekte geeignet, die effizient ausgeführt werden müssen, da ansonsten die Build- und Deployment-Zeiten zu lang dauern und ineffizient werden würden. Gerichtete Pipelines können DevOps-Prozesse beschleunigen und die Entwicklungs-Feedback-Schleife verkürzen, indem der Pipeline zusätzliche Informationen zwischen Jobabhängigkeiten mitgegeben werden.

Meist nutzt man zu Beginn in Projekten Phasen-Pipelines, die ab einer gewissen Größenordnung jedoch ineffizient werden. Die Konfiguration von Phasen-Pipelines kann mittels gerichteter Pipelines optimiert werden (siehe Bild 6.2).

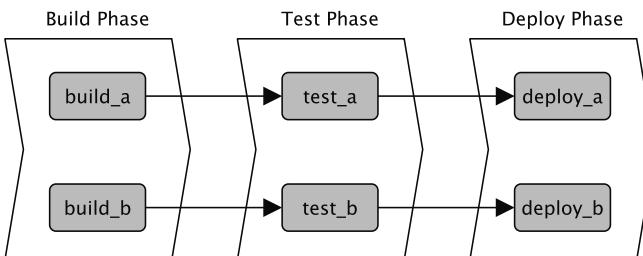


Bild 6.2
Gerichtete Pipelines

Phasen-Pipelines strukturieren nur grob die konzeptionellen Abhängigkeiten von Jobs anhand von wenigen Phasen (oft build, test, deploy). Folgt beispielsweise eine test-Phase einer build-Phase ($build \rightarrow test$), dann müssen erst alle Jobs `build_a` und `build_b` der build-Phase abgeschlossen sein, bevor die korrespondierenden Test-Jobs `test_a` und `test_b` der test-Phase gestartet werden können. Wenn nun aber der Test-Job `test_a` nur von `build_a`, aber nicht von `build_b` abhängt, könnte theoretisch bereits `test_a` gestartet werden, obwohl `build_b` noch nicht fertiggestellt ist. In einer Phasen-Pipeline würde `test_a` nicht gestartet und auf `build_b` gewartet werden. In einer gerichteten Pipeline wird `test_a` hin-

gegen gestartet. Dadurch müssen weniger Jobs zeitgleich zu Beginn einer Phase gestartet werden, und die Jobs können zeitversetzt zueinander gestartet werden. Insgesamt ergibt sich so statistisch oft eine gleichmäßige Auslastung der Deployment-Infrastruktur und Beschleunigung der Gesamtpipeline.

Wie folgender Pipeline-Auszug exemplarisch am Beispiel GitLab CI/CD zeigen soll, kann man diese feingranularen Abhängigkeiten beispielsweise mittels der `needs`-Klausel ausdrücken.

Listing 6.2 Konkrete Jobabhängigkeit am Beispiel von GitLab CI/CD

```
[...]

build_a:
  stage: build
  script:
    - echo "This job builds something quickly."

[...]

test_a:
  stage: test
  needs: [build_a] # Abhängigkeit von Job und nicht vorheriger Phase
  script:
    - echo "This test job will start as soon as build_a finishes."
    - echo "It will not wait for build_b,"
    - echo "or other jobs in the build stage, to finish."

[...]
```

6.1.3 Hierarchische Pipelines

Hierarchische Pipelines sind insbesondere für Monorepos und Projekte mit vielen unabhängig definierten Komponenten geeignet. Unter einem Monorepo versteht man eine Softwareentwicklungsstrategie, bei der Code für viele Projekte in einem einzigen großen Repository gespeichert wird.

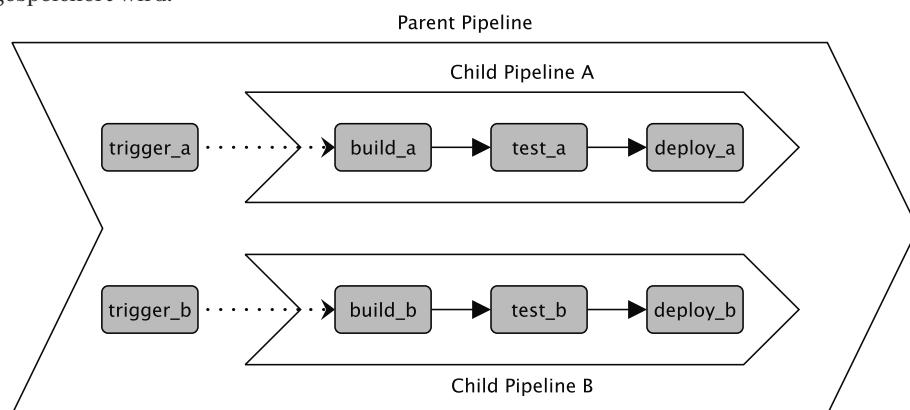


Bild 6.3 Hierarchische Pipelines

Die durch Listing 6.3 gebildete und in Bild 6.3 visualisierte Pipeline zeigt dabei exemplarisch eine Pipeline, die innerhalb eines Repositorys auf zwei weitere Teil-Pipelines A und B verweist. Die Pipelines A und B sind in diesem Beispiel als gerichtete Pipelines ausgeführt. Immer wenn die Hauptpipeline angestoßen wird, werden in diesem Beispiel auch die beiden Teilpipelines angestoßen.

Listing 6.3 Hierarchische Pipeline am Beispiel von GitLab CI/CD

```

stages:
  - triggers

trigger_a:
  stage: triggers
  trigger:
    include: a/.gitlab-ci.yml # <= Teil-Pipeline A

trigger_b:
  stage: triggers
  trigger:
    include: b/.gitlab-ci.yml # <= Teil-Pipeline B

```

Da Monorepositories üblicherweise alle Teilkomponenten eines großen Systems beinhalten, die aber dennoch durch jeweils eigenständige Teams betreut und deren Pipelines gebaut werden, bieten sich hierarchische Pipelines vor allem für den Bau von großen Systemen an. Monorepository-Strategien werden üblicherweise eingesetzt, um folgende Aspekte zu optimieren:

- **Sichtbarkeit:** Die Verwendung eines einzigen Repositorys gibt besseren Einblick in den Code und Assets für jedes Projekt. Dies hilft dabei, Abhängigkeiten über Teilkomponentengrenzen zu verwalten.
- **Kollaboration:** Ein einziges Repository zu nutzen macht die Zusammenarbeit einfacher. Das liegt daran, dass jeder auf den Code, die Dateien und die Assets zugreifen kann. So können Entwickler Assets gemeinsam nutzen und wiederverwenden.
- **Geschwindigkeit:** Die Verwendung eines einzigen Repositorys kann die Entwicklung beschleunigen. Es lassen sich zum Beispiel atomare Änderungen vornehmen (ein Code-Commit, um eine Änderung über mehrere Teilkomponenten hinweg vorzunehmen).

6.1.4 Steuerung von Pipelines

Pipelines können mittels Umgebungen und zugeordneten Umgebungsvariablen gesteuert und angepasst werden. Beispielsweise kann eine DATABASE_URL-Variablen die URL zu einer Datenbank beinhalten, die in verschiedenen Phasen und Jobs genutzt werden kann, ohne dort jedes Mal wieder neu gesetzt zu werden. Mittels solcher Variablen kann man Pipeline-Jobs auch an Umgebungen anpassen. Üblicherweise unterstützen Deployment-Systeme dabei die folgenden Arten von Variablen:

- **Explizit gesetzte Umgebungsvariablen**, die normalerweise innerhalb der Pipeline bzw. des Projekts gesetzt werden.

- **Vordefinierte Umgebungsvariablen**, die aus dem Deployment-System oder dem dahinter liegenden Versionskontrollsystem stammen. Das sind im Allgemeinen Daten wie Benutzernamen, Branch-Namen, Pipeline-, Commit-IDs, Projekt-IDs und vieles mehr. Diese Variablen können genutzt werden, um Deployments spezifisch auszuprägen.
- **Extern gesetzte Umgebungsvariablen**, die außerhalb des Repositorys gesetzt werden und insbesondere dazu dienen, Pipelines mit geheimen Zugangsdaten (Credentials) zu versorgen, die aus Sicherheitsgründen nicht in Versionskontrollsysteme eingecheckt werden sollten.

Listing 6.4 zeigt exemplarisch eine Jobdefinition eines GitLab CI/CD-Jobs, der eine Ingress-Ressource in einem Kubernetes-Cluster ausprägt (mehr zu Kubernetes in Kapitel 9 und zu Ingress-Ressourcen in Abschnitt 9.3.6). Das Beispiel entstammt einer Webtechnologie-Vorlesung des Autors, in der weit mehr als 100 Studierende individuelle Webprojekte erstellen und die Betreuer keine Lust hatten, für diese Anzahl an Projekten jeweils individuelle Anpassungen (wie beispielsweise individuelle URLs) vornehmen zu müssen. Vielmehr wird dies alles über eine automatisierte Deployment-Pipeline bewerkstelligt, die über Umgebungsvariablen gesteuert wird.

Dies erfolgt über mehrere explizit gesetzte und vordefinierte Variablen des Deployment-Systems.

- Die explizit definierte DOMAIN-Variable definiert die Domäne, unter der individuelle Projekte gehostet werden.
- Die explizit definierte INGRESS-Variable steuert, ob eine Ingress-Ressource überhaupt angelegt werden soll (und damit ein Projekt extern zugänglich gemacht werden soll).
- Die vordefinierte CI_PROJECT_ID-Variable gibt in GitLab eine eindeutige Projekt-ID an. Diese wird genutzt, um für jedes Projekt eine eindeutige URL generieren zu können.
- Ferner existiert implizit eine KUBECONFIG-Variable. Diese zeigt auf eine Kubeconfig-Datei, die ein kubectl-Command nutzt, um Zugriff auf einen externen Kubernetes-Cluster zu erhalten. Da in dieser Datei geheime Informationen (u. a. ein Zugriffstoken) stehen, wird diese nicht im Versionskontrollsystem gehalten, sondern in einer externen CI/CD-Variablen im Deployment-System gesetzt.

Listing 6.4 Ingress Deploy-Job am Beispiel von GitLab CI/CD

```
variables:                      # Setzen von Umgebungsvariablen für die Pipeline
  DOMAIN: "webtech.th-luebeck.dev"
  INGRESS: "False"

[...]

ingress:
  stage: deploy
  only:
    variables:
      - $INGRESS == "True" # Bedingte Ausführung von Jobs
  script:                      # Nutzung von Umgebungsvariablen
    - |
      HOST=webapp-$CI_PROJECT_ID.$DOMAIN \
        mo deploy/webapp-ing.yaml | kubectl apply -f -
```

Diverse Deployment-Systeme ermöglichen es ferner, solche Anpassungen mittels Umgebungen (*Environments*) zu strukturieren. Umgebungen beschreiben, wo der Code bereitgestellt wird. Solche Deployment-Systeme bieten eine Historie der Deployments in jeder Umgebung. Auf diese Weise ist es möglich, environment-spezifische Umgebungsvariablen zu nutzen. Typische Environments sind oft:

- **production** für die Produktiv- und Live-Systeme
- **staging** für Prototypen, Vorab- und Demo-Systeme
- **testing** für Test- und Systeme, die intensiven Qualitätssicherungsläufen unterworfen werden
- **development** für Deployments, die primär im Rahmen der Entwicklung oder Feature-Reviewing erforderlich werden

In GitLab CI/CD könnte eine Staging Environment beispielsweise wie in Listing 6.5 angelegt werden.

Listing 6.5 Anlegen einer Umgebung am Beispiel von GitLab CI/CD

```
deploy_staging:  
  stage: deploy  
  script:  
    - echo "Deploy to staging server"  
  environment:  
    name: staging  
    url: https://staging.example.com
```

In dieser Deployment-Umgebung stehen dann weitere Umgebungsvariablen zur Steuerung der Pipeline zur Verfügung wie beispielsweise:

- **CI_ENVIRONMENT_NAME** beinhaltet den in der Pipeline definierten Namen der Umgebung.
- **CI_ENVIRONMENT_SLUG** beinhaltet eine „bereinigte“ Version des Namens, die z. B. für die Verwendung in URL und DNS geeignet ist. Diese Variable ist garantiert eindeutig.
- **CI_ENVIRONMENT_URL** beinhaltet die URL der Umgebung, die in der Pipeline-Definition angegeben oder automatisch zugewiesen wurde.

■ 6.2 DevOps-geeignete Branching-Strategien

Moderne Versionskontrollsystme – wie beispielsweise GitHub oder GitLab – bieten leistungsfähige Werkzeuge, die es einfach machen, Branches zu erstellen und mittels Deployment-Systemen automatisiert in definierten Umgebungen auszubringen. Irgendwann müssen diese Branches jedoch wieder zusammengeführt werden. So haben z. B. Studien vom DORA Research Program gezeigt (Forsgren-Velasquez u. a. 2014), dass viele Teams auch deswegen ineffizient arbeiten, weil sie zu viele Branches pflegen oder keiner stringenten Branching-Strategie folgen. Es haben sich daher mehrere Muster etabliert, die es Teams ermöglichen, das Branching von

Versionskontrollsystmen möglichst effizient zu nutzen. Herausgestellt hat sich, dass Teams, die Branches häufiger in einen Main-Branch integrieren, auch produktiver sind. Empfohlen wird daher, dass sich DevOps-Ansätze vor allem auf eine „gesunde Mainline“ konzentrieren sollten, die mit minimalem Aufwand jederzeit in Produktion gebracht werden kann.

Einen umfassenden Überblick über Branching-Strategien und deren Stärken und Schwächen bietet (Fowler 2020). Wir werden uns hier auf nur einige wenige – dafür aber im DevOps-Kontext weit verbreitete – Strategien fokussieren, die sich u. a. wegen der stetig zunehmenden Beliebtheit des Versionskontrollsystms Git durchgesetzt haben. Dies sind z. B.

- Git-Flow,
- GitHub-Flow
- sowie die sogenannte Trunk-basierte Entwicklung.

All diesen Modellen liegt die Erkenntnis zugrunde, dass möglichst nah an einem Main-Branch gearbeitet werden sollte, Feature-Branches möglichst nur Stunden oder wenige Tage gebildet werden sollten. Sie unterscheiden sich dahingehend, dass vom Main-Branch entweder direkt (siehe Abschnitt 6.2.1 und Abschnitt 6.2.2) oder mittels Release-Branches (siehe Abschnitt 6.2.3) Systeme durch automatisierte Deployment-Pipelines in Produktion gebracht werden. Falls aus dem Main-Branch deployt wird, gibt es Strategien, die den Main-Branch durch einen vorgelagerten „Integrations-Branch“ (auch manchmal Staging genannt) schützen (siehe Abschnitt 6.2.1) oder darauf sogar verzichten (siehe Abschnitt 6.2.2). Die letzte und „radikalere“ Methode wird allerdings meist nur bei kleineren Projekten geringer Komplexität (z. B. „Webprojekte“) in Reinform angewendet, bei der die Effekte auf das Produktionssystem gut vorhersehbar sind.

6.2.1 Git-Flow

Bei diesem von Vincent Driessens beschriebenen Workflow (siehe Bild 6.4) wird dem Main-Branch ein schützender Development-Branch vorgeschaltet, in den mittler kleinere und kurzlebiger Feature-Branches eingearbeitet werden.

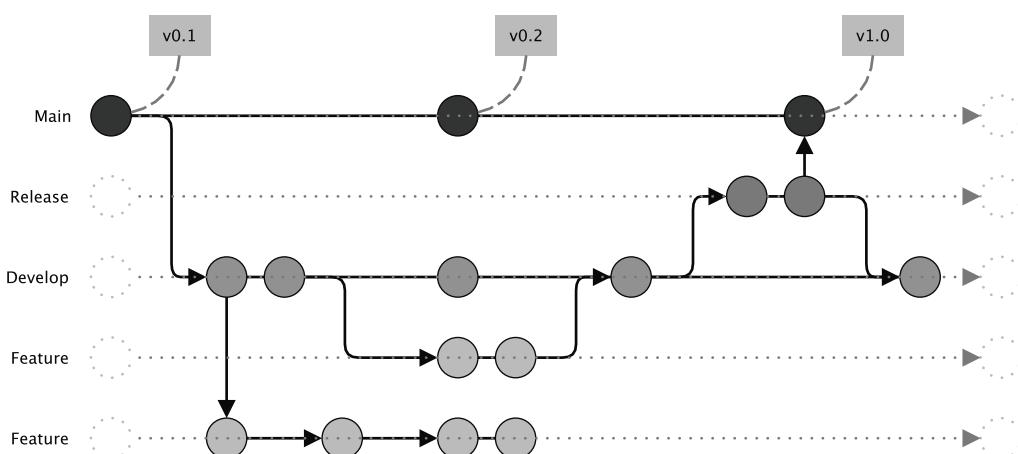


Bild 6.4 Workflow des Git-Flow

Dieser Workflow definiert ein strenges Branching-Modell, das um den Release des Projekts konzipiert wird, was insbesondere für das Management größerer Projekte und für Projekte mit einem Release-Zyklus nach Zeitplan geeignet ist. Verschiedenen Branches werden sehr spezifische Rollen zugewiesen, und es wird genau festgelegt, wann und wie die Interaktion zwischen diesen Rollen-Banches erfolgen soll.

Statt eines einzelnen Main-Banches sieht dieser Workflow zwei Branches vor, um den Verlauf des Projekts aufzuzeichnen. Der Main-Branch enthält den offiziellen Release-Verlauf, während der Develop-Branch als Integrations-Branch für Features dient. Es ist zudem üblich, alle Commits im Main-Branch mit einer Versionsnummer zu taggen.

Jedes neue Feature sollte sich in einem eigenen Branch befinden, der zum zentralen Repository gepusht werden kann. Doch statt Branches wie üblich auf Basis des Main-Banches zu erstellen, nutzen Feature-Banches den Develop-Branch als übergeordneten Branch. Wenn ein Feature fertig ist, wird es zurück in den Develop-Branch gemergt.

Ergänzend werden weitere Branches zum Vorbereiten von Releases verwendet. Sobald der Develop-Branch genügend Features für ein Release enthält oder ein vordefinierter Release-Termin ansteht, wird vom Develop-Branch ein Release-Branch abgespalten. Damit beginnt der neue Release-Zyklus. In diesem Branch sollten ab diesem Punkt keine neuen Features mehr hinzugefügt werden, sondern nur Bugfixes und ähnliche release-orientierte Änderungen. Ist er zur Auslieferung bereit, wird der Release-Branch in den Main-Branch gemergt und mit einer Versionsnummer getaggt.

6.2.2 GitHub-Flow

Der GitHub-Flow ist ein leichtgewichtiger, branch-basierter Workflow, der Teams und Projekte unterstützt, bei denen regelmäßig Bereitstellungen vorgenommen werden müssen. Im Gegensatz zu Git-Flow, der ja unter anderem einem Development- und einem Main-Branch beruht, wird beim GitHub-Flow nur auf einem Main-Branch gearbeitet. Insbesondere kleinere Projekte mit geringer Komplexität profitieren von dieser Vereinfachung. Der Ansatz gerät bei komplexeren und größeren Systemen jedoch an seine Grenzen.

Er basiert auf einem auf Pull Requests basierendem Workflow (siehe Bild 6.5):

1. Erzeugen eines Feature-Banches durch den Entwickler ausgehend vom Main-Branch.
2. Einbringen der für das Feature erforderlichen Änderungen, Streichungen und Ergänzungen mittels einer Sequenz von Commits.
3. Erzeugen eines Pull Requests.
4. Review des Pull Requests durch Verantwortliche für den Main-Branch. Dieser Schritt kann gegebenenfalls erforderliche Verbesserungen durch den Entwickler nach sich ziehen.
5. Deploy der reviewten Commits, um zu prüfen, ob erforderliche Tests bestanden werden und die Änderungen in eine Testumgebung eingespielt werden können.
6. Merge des Pull Requests in den Main-Branch, falls Schritt 5 erfolgreich bestanden wurde.

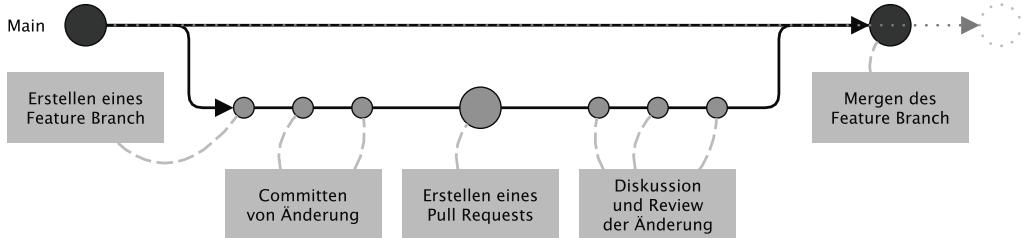


Bild 6.5 GitHub-Flow

6.2.3 Trunk-basierte Entwicklung

Diese u. a. von (Hammant 2020) beschriebene Branching-Strategie konzentriert sich darauf, alle Arbeiten auf einem Main-Branch (dem Trunk) vorzunehmen, um langlebige Branches zu vermeiden. Kleinere Teams (siehe Bild 6.6) committen direkt in den Main-Branch unter Verwendung automatisierter Main-Branch Integration (Test + Deployability-Checks), größere Teams (siehe Bild 6.6) können kurzlebiges Feature Branching verwenden.

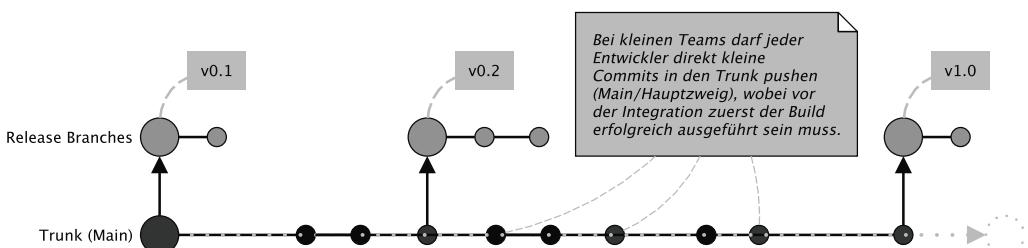
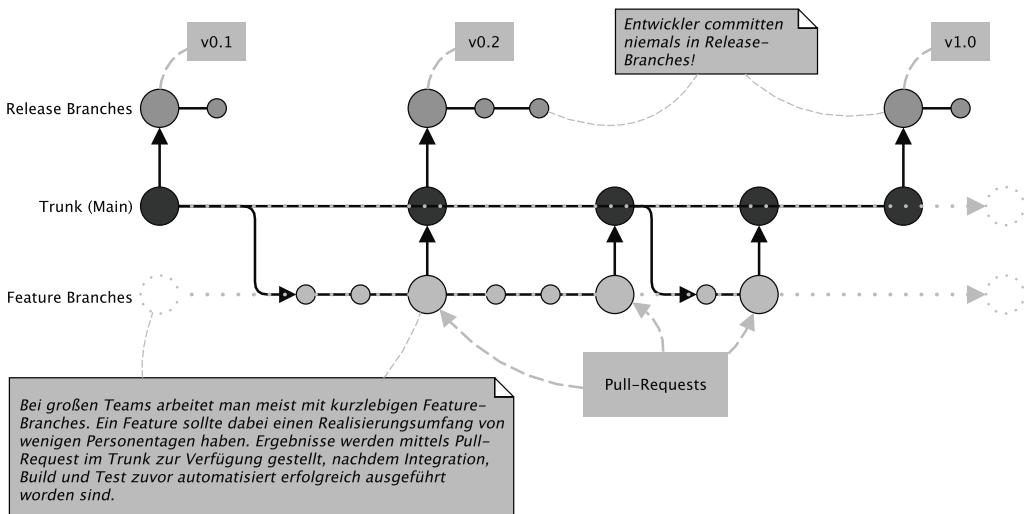


Bild 6.6 Trunk-basiertes Branching für große und kleine Teams

„Kurz“ wird dabei üblicherweise als wenige Tage interpretiert. Ausgehend vom Main-Branch werden dann Release-Branches gebildet, die mittels Deployment-Pipelines automatisiert in Produktion gebracht werden und (bis auf gegebenfalls erforderliche Hotfixes) nicht weiter fortgeführt werden. Der Main-Branch (Trunk) übernimmt somit die Rolle, die der Development-Branch im Git-Flow-Ansatz innehat. Trunk-basiertes Branching kann somit als eine Mischform von Abschnitt 6.2.1 und Abschnitt 6.2.2 verstanden werden, bei dem allerdings die Produktions-Releases aus spezifischen Branches gefahren werden.

■ 6.3 Zusammenfassung

Deployment-Pipelines werden mehr und mehr zu einem integralen Bestandteil von Versionsverwaltungssystemen, da sie dazu beitragen, Entwicklungszyklen schnell und agil zu halten (siehe auch DevOps-Prinzipien des Flow in Kapitel 3 und Abschnitt 3.1). In ein Code Repository eingebrachter Code kann mittels einer Deployment-Pipeline automatisiert integriert, gebaut, getestet und in eine Betriebs- oder Testumgebung ausgebracht werden. Sie sind daher aus der Entwicklung Cloud-nativer Anwendungen kaum noch wegzudenken. Es gibt mittlerweile eine Vielzahl an managed und self-hosted Lösungen. Einen guten Überblick liefert (Öggel und Kofler 2020).

Deployment-Pipelines können dabei wiederum als einfache Textdatei in einem Code-Repository definiert werden (Pipeline as Code). Üblicherweise werden dabei Pipeline-Phasen und innerhalb von Phasen Jobs definiert. Die Terminologie ist dabei von Produkt zu Produkt leicht unterschiedlich, die Prinzipien aber identisch. In diesem Kapitel wurde das Konzept Pipeline as Code exemplarisch am Beispiel des GitLab-Systems gezeigt. GitLab ist open-source und wird daher in vielen Unternehmen eingesetzt, jedoch auch als Managed Service bereitgestellt. Es gibt aber viele weitere und vergleichbare Lösungen, die auf ähnliche Art und Weise Deployment-Pipelines as Code definieren können. Wiederum liefert hier (Öggel und Kofler 2020) einen guten Überblick. Häufig basieren diese Lösungen im Hintergrund zur Ausführung von Jobs auf sogenannten Container-Ansätzen, die noch genauer in Kapitel 8 behandelt werden.

Damit das volle Potenzial der Automatisierung mittels Deployment-Pipelines aber auch genutzt werden kann, sollte man DevOps-geeigneten Branching-Strategien folgen. Studien vom DORA Research Program haben beispielsweise gezeigt (Forsgren-Velasquez u. a. 2014), dass viele Teams u. a. deswegen ineffizient arbeiten, weil sie zu viele Branches pflegen oder keiner stringenten Branching-Strategie folgen. Empfohlen wird daher mittlerweile, dass sich DevOps-Ansätze vor allem auf einen „gesunden Trunk“ konzentrieren sollten, der mit minimalem Aufwand jederzeit in Produktion gebracht werden kann. Auch hier liefert (Öggel und Kofler 2020) wieder einen guten Überblick. Dem Leser seien aber ergänzend der sehr interessante Überblick zu Vor- und Nachteilen unterschiedlichster Branching-Strategien von (Fowler 2020) sowie die Studien von (Forsgren-Velasquez u. a. 2014) und die Überlegungen von (Hammant 2020) zu sogenannten trunk-basierten Entwicklungstechniken empfohlen.

Ergänzende Materialien

Auf der Website zum Buch finden sich zu diesem Kapitel folgende Ergänzungsmaterialien:

- Übungen (Labs) zur praktischen Vertiefung und Nutzung von Deployment-Pipelines
- Auswahl kommerzieller und open-source verfügbarer CI/CD-Systeme (sowohl Managed Services als auch selbst-hostbare Services)
- Auswahl von weiterführenden Online-Quellen zu DevOps-Branching-Strategien
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer (Themen: DevOps-Prinzipien des Flow und des Feedbacks, Deployment-Pipelines, Auswirkung von DevOps auf Architekturen)



<https://bit.ly/3upTdUi>

7

Infrastructure as Code

„Talk is cheap. Show me the code.“

Linus Torvalds, Entwickler und Initiator des Linux-Kernels

Public Cloud Provider organisieren ihre globalen Infrastrukturen in Form „logischer“ Rechenzentren, die oft als „Regionen“ bezeichnet werden (siehe Bild 7.1).

Innerhalb von Regionen gibt es meist mehrere Verfügbarkeitszonen (oder auch nur Zonen). Diese Zonen sind einzelne Rechenzentren, die Cloud-Services für Kunden bereitstellen. Die Regionen sind voneinander isoliert, um zu verhindern, dass sich Ausfälle ausbreiten. Doch auch die Verfügbarkeitszonen (also Rechenzentren) innerhalb von Regionen werden unabhängig voneinander betrieben, sodass beim Ausfall einzelner Verfügbarkeitszonen der Betrieb weiterer Verfügbarkeitszonen fortgesetzt werden kann. Ergänzend existieren sogenannte Edge-Standorte. Edge-Standorte sind zumeist über ein Provider-Netzwerk direkt an die Regionen und Verfügbarkeitszonen angebunden, um möglichst kurze Latenzen für Nutzer von cloud-basierten Services innerhalb einer Region anbieten zu können. Logischerweise folgt die Dislozierung dieser Regionen, Zonen und Edge-Standorte dem Bedarf. Insofern lässt sich aus solchen Karten auch immer die wirtschaftliche Prosperität und der volkswirtschaftliche Digitalisierungsgrad ausmachen. Eine gewisse Unterrepräsentation der Regionen Afrika und Südamerika findet sich beispielsweise bei vielen Cloud-Providern (siehe Bild 7.1).

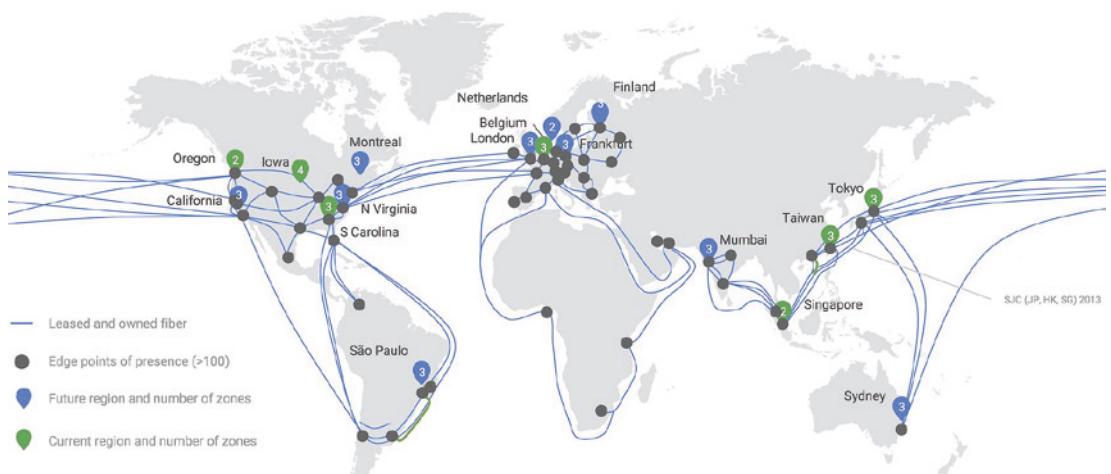


Bild 7.1 Globale Cloud-Infrastruktur am Beispiel von Google, Stand: 2020

Zumeist können Kunden innerhalb dieser Regionen Ressourcen mit webbasierten Benutzeroberflächen (Management Console) anfordern und freigeben (siehe Bild 7.2).

Bild 7.2 Web Management Console am Beispiel von Google

Insbesondere in automatisierten Prozessen ist es aber üblich, Cloud-Ressourcen auch mittels Kommandozeilentools wie beispielsweise `aws-cli` (AWS), `az` (Azure) oder `gcloud` (Google) zu verwalten. Von dieser Möglichkeit wird überwiegend in automatisierten Umgebungen wie Build- und Deployment-Pipelines Gebrauch gemacht. Viele Provider bieten darüber hinaus auch noch eine ganze Reihe für Programmiersprachen spezifische APIs (Language Bindings) zur Interaktion mit ihren Cloud-Infrastrukturen an, die u. a. von Infrastructure-as-Code-Lösungen wie Terraform genutzt werden. Die Interaktion mit Cloud-Infrastrukturen kann somit hochgradig automatisiert werden und grundsätzlich ohne manuelle Tätigkeiten erfolgen und sehr problemspezifisch ausgeprägt werden.

■ 7.1 Virtualisierung

Unter Virtualisierung versteht man grundsätzlich die Erzeugung von virtuellen Realitäten und deren Abbildung auf physische Realitäten. Im Cloud Computing werden unter Virtualisierung teilweise verschiedene Konzepte und Technologien verstanden, die der Virtualisierung von Hardware-Infrastruktur und Software-Infrastruktur dienen. Dies erfolgt zu folgenden Zwecken:

- **Multiplizität:** Erzeugung mehrerer virtueller Realitäten innerhalb einer physikalischen Realität (z. B. mehrere VMs auf einem physischen Server betreiben).
- **Entkopplung:** Bindung und Abhängigkeit zur Realität auflösen (z. B. Verschiebung einer VM von einem Virtualisierungshost zu einem anderen zur Laufzeit).
- **Isolation:** Physische Seiteneffekte zwischen virtuellen Realitäten vermeiden (z. B. beeinträchtigt eine Endlosschleife auf einer VM nicht die Applikation auf einer anderen VM).

7.1.1 Virtualisierung von Hardware-Infrastruktur

Im Cloud Computing bildet die Virtualisierung von Hardware die Grundlage der Automatisierung und die Basis des zunehmend umfangreicher und komplexer werdenden Cloud-Service-Portfolios und dient der Entkopplung der physischen Infrastruktur durch Normierung von Ressourcenkapazitäten (z. B. in Form von „S/M/L/XL“-Instanzentypen von virtuellen Maschinen) auf heterogener und wechselnder Hardware. Die Bereitstellung dieser Ressourcen kann „software-defined“ erfolgen.

Durch Hardware-Virtualisierung werden die Ressourcen eines Rechnersystems aufgeteilt und von mehreren Betriebssysteminstanzen genutzt. Die Anforderungen der Betriebssysteminstanzen werden von einer Virtualisierungssoftware (Virtual Machine Monitor, VMM) abgefangen und auf die reale vorhandene Hardware umgesetzt, die Gast- und Host-Ressourcen verwaltet. Die Virtualisierung umfasst dabei die folgenden Ressourcen:

- Prozessor in Form virtueller Rechnerkerne (vCPU)
- Hauptspeicher in Form virtuellen Speichers im Host-RAM (u. a. mittels Memory-Ballooning)
- Netzwerk in Form virtueller Netzwerkschnittstellen und virtueller Netzwerk-Infrastrukturen (VLAN) inklusive Firewallregeln
- Storage in Form virtueller Block-Devices (Block-Storage) und Dateisysteme (File-Storage)

Als die dabei relevanten Virtualisierungstechnologien sind die Para-Virtualisierung (Typ-1-Virtualisierung) und Voll-Virtualisierung (Typ-2-Virtualisierung) zu nennen (siehe Bild 7.3).

Bei der **Typ-1-Virtualisierung (Para-Virtualisierung)** läuft ein Hypervisor direkt auf der verfügbaren Hardware. Dieser entspricht somit einem auf Virtualisierung spezialisierten Betriebssystem. Das Gast-Betriebssystem muss allerdings um Treiber ergänzt werden, um mit dem Hypervisor interagieren zu können. Wegen der erforderlichen Hypervisor-Treiber können daher nicht beliebige Gast-Betriebssysteme auf Hypervisoren laufen. Für die üblichen Betriebssysteme (insbesondere Linux) im Cloud Computing-Umfeld ist dies jedoch meist gegeben. Der Hypervisor nutzt die Treiber eines Host-Betriebssystems, um auf die reale Hardware zuzugreifen.

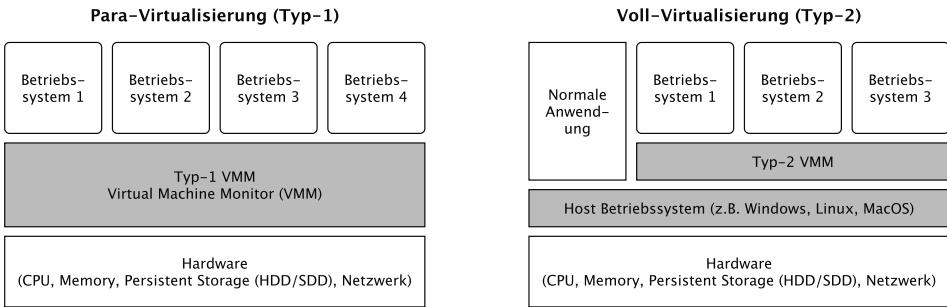


Bild 7.3 Para- und Voll-Virtualisierung

Damit müssen im Hypervisor nicht aufwendig eigene Treiber implementiert werden. Der Leistungsverlust der Para-Virtualisierung liegt dabei zumeist deutlich unter 5 %. Dieser Ansatz wird daher zumeist in Rechenzentren von Cloud-Providern gefahren.

Bei der **Typ-2-Virtualisierung (Voll-Virtualisierung)** steht jedem Gastbetriebssystem ein eigener virtueller Rechner mit virtualisierten Ressourcen (CPU, RAM, Disks, Netzwerkkarten etc.) zur Verfügung. Der VMM läuft als Anwendung unter dem Host-Betriebssystem und verteilt die Hardwareressourcen des Rechners an die VMs. Teilweise muss die VMM Hardware emulieren, die nicht für den gleichzeitigen Zugriff von mehreren Betriebssystemen ausgelegt ist (z. B. Netzwerkkarten, Grafikkarten). Der Leistungsverlust der Voll-Virtualisierung liegt daher über der Para-Virtualisierung, aber üblicherweise unter 10 %. Dieser Ansatz wird daher zumeist in DevOps-Kontexten insbesondere auf Entwicklungsrechnern für lokale Teste gefahren.

Der Vollständigkeit halber sollte noch die Virtualisierungsmethode der **Emulation** genannt werden. Durch Emulation kann die Hardware eines nicht vorhandenen oder nicht kompatiblen Rechnersystems oder können Teile eines entsprechenden Rechnersystems virtuell nachgebildet werden. Dabei wird Hardware also durch Software simuliert. Bekannt ist dieses Konzept z. B. in Form von „Gameboy“-Emulatoren für Browser oder Smartphones. Da dieser Ansatz aber durch substanziale Performance-Einbußen gekennzeichnet ist, wird er beim Betrieb von Cloud-Infrastrukturen normalerweise nicht eingesetzt. Daher wird dieser Ansatz hier nicht weiter betrachtet.

7.1.2 Virtualisierung von Software-Infrastruktur

Der Vollständigkeit halber seien ferner die beiden folgenden Software-Virtualisierungsarten genannt.

- Betriebssystem-Virtualisierung (Containerization)
- Anwendungsvirtualisierung (Runtime, z. B. die Java Virtual Machine)

Insbesondere die Betriebssystem-Virtualisierung hat bei der Standardisierung von Deployment Units Cloud-nativer Anwendungen Bedeutung erhalten und wird daher noch im Detail in Kapitel 8 behandelt werden. Da beide genannte Virtualisierungsarten auf Infrastrukturerbene eher eine untergeordnete Rolle spielen, werden sie in diesem Kapitel nur kurz behandelt.

Bei der **Betriebssystem-Virtualisierung** gibt es keinen Hypervisor. Jede Applikation läuft direkt als Prozess im Host-Betriebssystem. Diese Applikationsprozesse sind jedoch mittels Betriebssystemmechanismen isoliert und werden oft Container genannt. Die Isolation erfolgt dabei u. a. durch Namespaces (bezüglich CPU, RAM, Disk I/O), isolierte Dateisysteme und prozesseigene virtuelle Netzwerkschnittstellen. Die Start-up-Zeit solcher Container entspricht der Startdauer für einen normalen Applikationsprozess, da kein Boot des Betriebssystems erforderlich ist. Container sind im Vergleich zu virtuellen Maschinen also deutlich leichtgewichtiger und ressourcenschonender. Auch der Leistungsverlust ist in der Regel für CPU und RAM nicht messbar. Allerdings ist die Isolation weniger ausgeprägt als bei virtuellen Maschinen (siehe Bild 7.4). Das Konzept läuft in verschiedenen Betriebssystemen unter unterschiedlichen Bezeichnungen. Bei Free BSD ist es beispielsweise als Jails bekannt, in Solaris nennt sich das Konzept Zones. Im Cloud Computing ist es mit Docker insbesondere auf Linux-Systemen populär geworden, da sich mit Docker für Entwickler erstmals in sich geschlossene Deployment Units komfortabel definieren und zur Ausführung bringen ließen.

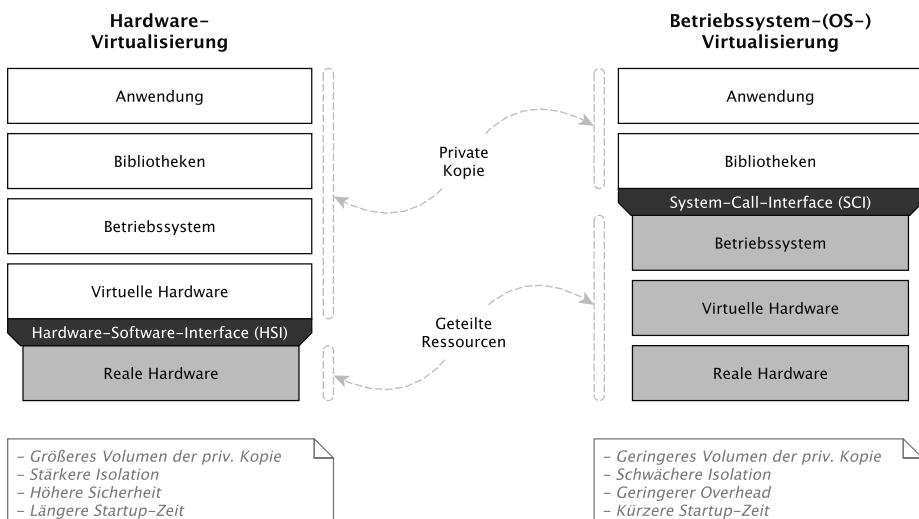


Bild 7.4 Vergleich von Hardware- und Betriebssystem-Virtualisierung

Bei der **Anwendungsvirtualisierung** wird Anwendungen eine Programmierschnittstelle und eine Laufzeitumgebung (Runtime) zur Verfügung gestellt, die eine Anwendungen (möglichst) komplett vom darunterliegenden Betriebssystem entkoppelt. Primäres Ziel ist die betriebssystemunabhängige Ausführung von Anwendungen. Das bekannteste Beispiel ist vermutlich die Java Virtual Machine. Diese Art der Virtualisierung wird aber mittlerweile von vielen modernen Programmiersprachen genutzt. Die damit einhergehenden Leistungsverluste werden billigend in Kauf genommen, da Vorteile bei der Entwicklung von Anwendungen überwiegen. Zudem können die Leistungseinbußen durch Byte-Code-Ansätze, Just-in-Time Compiler und für die Ausführung optimierte Runtime Environments mittlerweile in akzeptable Größenordnungen gebracht werden. Da Anwendungsvirtualisierung in Cloud-nativen Systemen aber meist im Rahmen des Polyglott Programming innerhalb von Containern gekapselt wird und daher nicht als eigenständiges Konzept auftritt, wird diese Virtualisierungsform nicht weiter betrachtet.

■ 7.2 Provisionierung

Dank Virtualisierung der IT-Infrastruktur ist die automatisierte Bereitstellung von Ressourcen im Verlaufe der letzten Jahrzehnte deutlich einfacher geworden. Dies gilt insbesondere für das Cloud Computing. Unter Provisionierung versteht man dabei Prozesse für die Bereitstellung und Integration von Ressourcen innerhalb einer IT-Infrastruktur. Provisionierung ist ein weit gefasster Begriff, der u. a. Richtlinien und Verfahren beim Bezug von Cloud-Diensten und -Lösungen von einem Cloud-Service-Anbieter umfasst.

Formal kann Provisionierung als die Überführung eines System-Ist-Zustands in einen System-Ziel-Zustand verstanden werden. Unter einem Systemzustand ist dabei die Gesamtheit der Software, Daten und Konfigurationen auf einem System zu verstehen. Die Überführung zwischen zwei Zuständen erfolgt durch einen Provisionierungsmechanismus, der in folgenden Schritten ausgeführt wird.

1. Ausgangszustand feststellen
2. Vorbedingungen prüfen
3. Zustandsändernde Aktionen ermitteln
4. Zustandsändernde Aktionen durchführen
5. Nachbedingungen prüfen (im Fehlerfall gegebenenfalls Zustand zurücksetzen)

Da insbesondere die Schritte 1 bis 3 für Systeme mit beliebigen Ausgangszuständen beliebig komplex werden können, hat sich bei der Provisionierung im Cloud Computing eine vereinfachte Form der Provisionierung durchgesetzt, die unter der Bezeichnung „Immutable Infrastructure“ läuft (siehe auch Abschnitt 7.2.1) und darauf basiert, dass nicht auf beliebigen, sondern auf festen und bekannten Basiszuständen (z. B. einem Basis-VM-Image) idempotente Provisionierungaktionen ausgeführt werden können. Idempotente Aktionen sind Aktionen, die dasselbe Ergebnis erzeugen, egal ob sie einmal oder mehrfach ausgeführt werden. Wiederholte Ausführungen ändern also den Zielzustand nicht. Dies macht die Sicherstellung konsistenter Systemzustände einfacher, da beim Scheitern einer Aktion einfach alle Aktionen wiederholt werden können, um einen gewünschten konsistenten System-Ziel-Zustand zu erreichen. Bei Wiederholungen muss man also nicht im Detail nachprüfen, welche Aktion nun erfolgreich und welche nicht erfolgreich war.

7.2.1 Immutable Infrastructure

In einer traditionellen veränderbaren Serverinfrastruktur werden Server kontinuierlich aktualisiert und somit verändert. Administratoren, die mit dieser Art von Infrastruktur arbeiten, können sich z. B. per SSH auf Servern einloggen, Pakete manuell aktualisieren oder zurückstufen, Konfigurationsdateien optimieren und neuen Code direkt auf Servern bereitstellen. Mit anderen Worten: Diese Server sind veränderbar und können angepasst werden, nachdem sie erstellt wurden. Eine Infrastruktur, die aus veränderlichen Servern besteht, kann selbst als veränderlich bezeichnet werden.

Eine unveränderliche Infrastruktur (Immutable Infrastructure) ist ein Infrastrukturparadigma, bei dem Server nach ihrer Bereitstellung nicht mehr verändert werden. Wenn etwas

aktualisiert, repariert oder in irgendeiner Weise verändert werden muss, werden neue Server, die aus einem gemeinsamen Image mit den entsprechenden Änderungen erstellt wurden, bereitgestellt, um die alten Server zu ersetzen. Nachdem sie validiert wurden, werden sie in Betrieb und alte Server außer Betrieb genommen. In virtualisierten Cloud-Infrastrukturen ist dies sogar einfacher.

Zu den Vorteilen einer unveränderlichen Infrastruktur gehören mehr Konsistenz und Zuverlässigkeit der Infrastruktur und vor allem ein einfacheres und besser vorhersehbares Bereitstellungsverfahren. Probleme, die in veränderlichen Infrastrukturen häufig auftreten, wie z. B. Konfigurationsabweichungen, werden gemildert oder ganz vermieden. Erforderlich sind jedoch eine umfassende Bereitstellungautomatisierung, eine schnelle Serverbereitstellung in einer Cloud-Computing-Umgebung und Lösungen für den Umgang mit zustandsabhängigen oder ephemeren Daten wie Protokollen.

Bevor Virtualisierung und Cloud Computing weit verbreitet waren, konzentrierte sich die Serverinfrastruktur primär auf physische Server. Diese physischen Server waren teuer und zeitaufwendig in der Erstellung; die Ersteinrichtung konnte Tage oder Wochen dauern, weil es u. a. so lange dauerte, neue Hardware zu bestellen, die Maschine zu konfigurieren und sie dann in einem Server-Rack zu installieren.

Da die Kosten für den Austausch eines Servers hoch waren, war es am wirtschaftlichsten, die vorhandenen Server so lange wie möglich und mit so wenigen Ausfallzeiten wie möglich zu nutzen. Das bedeutete, dass es viele Änderungen vor Ort für regelmäßige Implementierungen und Updates gab, aber auch für Ad-hoc-Korrekturen, Optimierungen und Patches. Die Folge solcher häufigen manuellen Änderungen ist, dass die Server schwer zu replizieren sind, was jeden einzelnen zu einer einzigartigen und anfälligen Komponente der gesamten Infrastruktur macht.

Das Aufkommen von Virtualisierung und On-Demand/Cloud Computing stellt hier einen Wendepunkt dar. Virtuelle Server sind preiswerter, selbst im großen Maßstab, und sie können in Minuten statt in Tagen oder Wochen erstellt und zerstört werden. Dies ermöglicht automatisierte Bereitstellungs-Workflows, z. B. durch die Verwendung von Konfigurationsmanagement-Tools oder Cloud-APIs zur schnellen, programmgesteuerten und automatischen Bereitstellung neuer Server. Die Geschwindigkeit und die geringen Kosten für die Erstellung neuer virtueller Server machen das Prinzip der Unveränderlichkeit interessant.

Das Paradigma, Server nach der Bereitstellung zu ändern, ist auch in modernen virtualisierten Infrastrukturen immer noch üblich. Im Gegensatz dazu werden unveränderliche Infrastrukturen von Anfang an so konzipiert, dass sie sich auf Technologien für die schnelle Bereitstellung von Architekturkomponenten stützen. Dies wird auch Infrastructure as Code (IaC) genannt.

7.2.2 IaC-Ansätze

Unter Infrastruktur als Code (IaC) versteht man die Verwaltung und Bereitstellung von IT-Ressourcen mittels maschinenlesbarer Definitionsdateien anstelle von physischer Hardwarekonfiguration oder interaktiven Konfigurationstools. Über IaC verwaltete IT-Infrastruktur kann sowohl physische Geräte wie Bare-Metal-Server als auch virtuelle Maschinen und zugehörige Konfigurationsressourcen umfassen.

Tabelle 7.1 Eine Auswahl gängiger Provisionierungswerkzeuge

| Tool | Methode | Provisionierungsansatz | Sprache | Anmerkungen |
|-----------|------------|------------------------|-----------|----------------|
| Ansible | Push | deklarativ + imperativ | Python | |
| CFEngine | Pull | deklarativ | C | |
| Chef | Pull | deklarativ + imperativ | Ruby | |
| Otter | Push | deklarativ + imperativ | - | nur Windows BS |
| Puppet | Pull | deklarativ | C++, Ruby | |
| SaltStack | Push, Pull | deklarativ + imperativ | Python | |
| Terraform | Pull | deklarativ | Go | |

Eine Auswahl gängiger Provisionierungswerkzeuge findet sich in Tabelle 7.1.

Bei der **deklarativen Provisionierung** definiert man den gewünschten Zustand. Das Provisionierungssystem führt dann aus, was geschehen muss, um diesen gewünschten Zustand zu erreichen. Bei der **imperativen Provisionierung** definiert man hingegen, wie die Infrastrukturressourcen geändert werden sollen. Hierzu werden bestimmte Befehle definiert, die in der richtigen Reihenfolge ausgeführt werden müssen, um im gewünschten Zustand zu enden.

Die Ansätze unterscheiden sich ferner in der Art, wie die zu provisionierenden Maschinen ihre Konfiguration beziehen. Bei **pull-basierten** Ansätzen zieht die zu konfigurierende Maschine die Konfiguration vom steuernden Provisionierungsmechanismus. Hier muss die Ausgangsbasis bereits fähig sein, dies zu tun, und auch den Provisionierungsmechanismus und dessen Konfiguration (z. B. IP-Adresse, URL-Adresse etc.) kennen. Meist ist hierzu ein spezifischer Agent des Provisionierungsmechanismus im Basiszustand des zu provisionierenden Systems erforderlich. **Push-basierte** Ansätze erfordern hier weniger Grundfähigkeiten in den Basis-Images. Der Provisionierungsmechanismus muss jedoch die anzuwendende Konfiguration auf die zu provisionierende Maschine injizieren. Gegebenenfalls sind auch hier Voraussetzungen wie etwa ein SSH-Zugang erforderlich. Diese sind aber zumeist weniger spezifisch als ein spezieller Agent des Provisionierungsmechanismus.

IaC-basierte Provisionierung bietet eine flexible Abstraktion von Ressourcen und Providern. Dieses Modell erlaubt es somit, vielfältige Ressourcen von physischer Hardware, virtuellen Maschinen und Containern bis hin zu E-Mail- und DNS-Providern abzubilden. Aufgrund dieser Flexibilität gibt es eine Reihe von Überschneidungen mit bestehenden Tools.

- **Konfigurationsmanagement-Lösungen** wie Chef oder Puppet installieren und verwalten Software auf bereits existierenden Maschinen. IaC-basierte Provisionierung konzentriert sich auf die Provisionierung von Ressourcen und damit auf die übergeordnete Abstraktion des zugrunde liegenden Rechenzentrums und der zugehörigen Dienste. Konfigurationsmanagement-Lösungen können zusammen mit IaC-basierten Provisionierungsansätzen verwendet werden, z. B. zur Initialisierung frisch provisionierter Maschinen.
- Mit **provider- bzw. infrastrukturspezifischen Tools** wie CloudFormation (AWS), Heat (OpenStack) usw. können die Details einer Infrastruktur in einer Konfigurationsdatei kodifiziert werden. Die Konfigurationsdateien erlauben es, die Infrastruktur zu erstellen, zu verändern und zu zerstören. IaC-basierte Provisionierungswerkzeuge sind von diesen Ansätzen inspiriert, setzen sie aber provider- bzw. infrastrukturagnostisch um. Dadurch lässt sich die gesamte Infrastruktur mit ihren unterstützenden Services darstellen und verwalten anstatt nur die Teilmenge, die innerhalb eines einzelnen Providers oder einer

Infrastruktur existiert. IaC-basierte Provisionierung bietet daher eine einheitliche Syntax, statt dass Nutzer für jede Plattform und jeden Dienst unabhängige und nicht interoperable Tools verwenden müssen.

- **Provider- und Infrastruktur-APIs** wie etwa Boto oder Fog werden verwendet, um einen nativen Low-Level-Zugriff auf Cloud-Anbieter und -Dienste zu ermöglichen. Einige Bibliotheken sind auf bestimmte Clouds fokussiert (z. B. Boto auf AWS), während andere (z. B. Fog) versuchen, infrastrukturagnostische APIs zu realisieren, was im Einzelfall aufgrund der geringen Standardisierung von Provider APIs nur begrenzt realisierbar ist. Die Verwendung einer Client-Bibliothek bietet nur einen Low-Level-Zugang zu den APIs, sodass die Anwendungsentwickler ihre eigenen Werkzeuge erstellen müssen, um ihre Infrastruktur aufzubauen und zu verwalten.

IaC-basierte Provisionierungstools sind nicht dafür gedacht, einen programmatischen Low-Level-Zugang zu Providern zu ermöglichen, sondern bieten stattdessen eine High-Level-Syntax zur Beschreibung, wie Cloud-Ressourcen und -Services erstellt, bereitgestellt und kombiniert werden sollen. Diese Lösungen setzen oft auf ein Plug-in-basiertes Modell zur Unterstützung von Infrastrukturen und Providern. Typische Anwendungsfälle für IaC-basierte Provisionierungslösungen sind beispielsweise:

- **Mehrschichtige Anwendungen:** Die N-Tier-Architektur ist ein verbreitetes Architekturmuster. Die häufigste 2-Tier-Architektur ist ein Pool von Webservern, die eine Datenbankebene verwenden. Zusätzliche Schichten werden für API-Server, Caching-Server, Routing-Netze usw. hinzugefügt. Dieses Muster wird verwendet, weil die Tiers unabhängig voneinander skaliert werden können. Jede Schicht kann als eine Sammlung von Ressourcen beschrieben werden inklusive Abhängigkeiten zwischen den einzelnen Schichten. Insbesondere die Abhängigkeiten machen die Provisionierung komplex, da beispielsweise sichergestellt sein muss, dass eine Datenbankschicht verfügbar ist, bevor die Webserver gestartet werden, und dass die Load Balancer die Webknoten kennen.
- **Self-Service-Cluster:** Ab einer bestimmten Unternehmensgröße wird es für ein zentrales Betriebsteam zunehmend schwieriger, eine große und wachsende Infrastruktur zu verwalten. Stattdessen wird es in agilen DevOps-Kontexten attraktiver, eine Self-Service-Infrastruktur zu erstellen, die es den Produktteams ermöglicht, ihre eigene Infrastruktur mithilfe von Werkzeugen zu verwalten, die vom zentralen Betriebsteam bereitgestellt werden (siehe Abschnitt 3.1.3). Mit IaC-basierten Provisionierungstools kann das Wissen, wie ein Service aufgebaut und skaliert werden soll, in einer Konfiguration kodifiziert werden. Solches kodifiziertes Betriebswissen kann in Form von Konfigurationen innerhalb einer Organisation geteilt werden, sodass Produktteams die Konfiguration als Blackbox nutzen und zum Betrieb ihrer Services einsetzen können.
- **Wegwerf-Umgebungen:** Es ist gängige Praxis, sowohl eine Produktions- als auch eine Staging- oder QA-Umgebung zu haben. Diese Umgebungen sind zumeist kleinere Klone von ihren Produktionspendants, werden aber zum Testen neuer Anwendungen vor der Freigabe in der Produktion verwendet. Je größer und komplexer die Produktionsumgebung wird, desto aufwendiger wird es, eine aktuelle Staging-Umgebung zu pflegen. Mittels IaC-Tools wie Terraform kann die Produktionsumgebung kodifiziert und dann geteilt werden. Diese Konfigurationen können verwendet werden, um schnell neue Umgebungen zum Testen zu erstellen, die ebenso einfach gelöscht werden können. Umgebungen lassen sich so also anlassbezogen erstellen und herunterfahren.

- **Multi-Cloud-Bereitstellung:** Infrastruktur kann als besondere Form des Hybrid-Computings über mehrere Clouds verteilt werden, um z. B. die Fehlertoleranz zu erhöhen. Eine Multi-Cloud-Bereitstellung ermöglicht eine schnellere Wiederherstellung beim Ausfall einer Region oder eines ganzen Providers. Die Realisierung von Multi-Cloud-Implementierungen kann jedoch eine große Herausforderung sein, da viele bestehende Tools für das Infrastrukturmanagement cloud-spezifisch sind. Mittels IaC-Tools lassen sich Infrastrukturen cloud-agnostisch definieren, um die Verwaltung und Orchestrierung beim Aufbau großer Multi-Cloud-Infrastrukturen zu vereinfachen.

Diese IaC-Prinzipien sollen nun an den beiden Typvertretern Vagrant (siehe Abschnitt 7.2.3) zur Provisionierung einzelner virtueller Maschinen und Terraform (siehe Abschnitt 7.2.4) zur Provisionierung komplexerer – d. h. aus mehreren VMs bestehenden – Infrastrukturen erläutert werden.

7.2.3 Provisionierung von lokalen Umgebungen

Vagrant ist ein Kommandozeilenprogramm zur Verwaltung des Lebenszyklus virtueller Maschinen und soll die grundsätzlichen Prinzipien der automatisierten Bereitstellung einzelner virtueller Maschinen veranschaulichen. Diese Art der Provisionierung wird überwiegend in Development-Workflows genutzt, um Entwicklern die Möglichkeit zu geben, Änderungen lokal vornehmen und in einer einheitlichen Testumgebung lokal austesten zu können.

Virtuelle Maschinen werden dabei mittels eines sogenannten Vagrantfiles definiert (siehe Bild 7.5). Ein Vagrantfile definiert den erforderlichen Maschinentyp und wie diese Maschine konfiguriert und bereitgestellt wird. Die Syntax von Vagrantfiles ist Ruby. Jedoch sind keine tiefegehenden Kenntnisse der Ruby-Programmiersprache erforderlich, um Änderungen am Vagrantfile vorzunehmen, da es sich meist um einfache Variablenzuweisungen handelt.

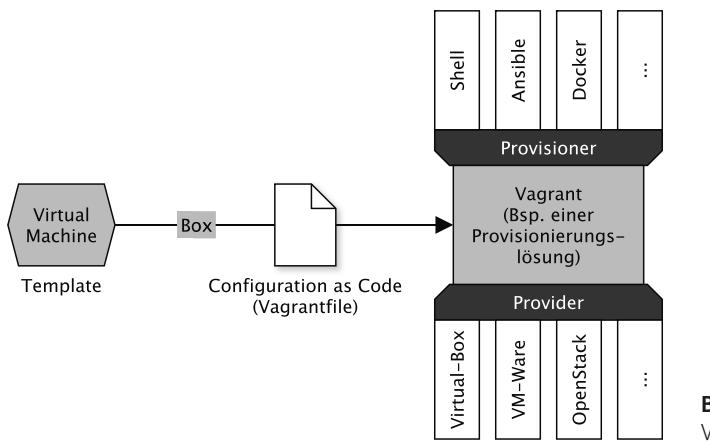


Bild 7.5
Vagrant

Boxen sind das Paketformat für Vagrant-Umgebungen. Eine Box kann auf jeder Plattform, die Vagrant unterstützt, verwendet werden, um eine identische Arbeitsumgebung zu erstellen. Boxen stellen damit einen festen und bekannten Basiszustand im Sinne einer Immutable Infrastructure dar (z. B. ein Ubuntu 20.04 Linux Image).

Provisioner ermöglichen es, automatisiert Software zu installieren und Konfigurationen zu ändern. Dies ist nützlich, da die Boxen typischerweise nicht perfekt für einen Anwendungsfall gebaut sind. Durch die Verwendung der in Vagrant eingebauten Provisionierungssysteme wird der Prozess automatisiert, sodass die Provisionierung ohne menschliche Interaktion wiederholbar ist und mit einem einzigen Vagrant up-Befehl eine vollständige Arbeitsumgebung erstellt werden kann. Vagrant bietet mehrere Optionen für die Bereitstellung der Maschine, von einfachen Shell-Skripten bis hin zu komplexeren Konfigurationsmanagementsystemen wie beispielsweise Ansible, Chef, Puppet oder Salt.

Provider erzeugen virtuelle Maschinen auf Basis einer Box üblicherweise in Typ-2-Virtualisierungsumgebungen wie etwa VirtualBox oder Hyper-V. Andere Virtualisierungsumgebungen wie vSphere oder gar private IaaS-Infrastrukturen wie OpenStack können grundsätzlich ebenfalls unterstützt werden.

Folgendes Listing 7.1 setzt beispielsweise einen NGINX-basierten Webserver als virtuelle Maschine auf.

Listing 7.1 Exemplarisches Vagrantfile

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/bionic64"

  config.vm.network "forwarded_port", guest: 80, host: 8080,
                    host_ip: "127.0.0.1"
  config.vm.synced_folder "./web", "/var/www/html"

  config.vm.provider "virtualbox" do |vb|
    vb.name = "nginx"
    vb.memory = 1024 # 1GB Memory
    vb.cpus = 2      # 2 Cpus
  end

  config.vm.provision "shell", path: "provision-nginx.sh"
end
```

Die Anpassung der Box, d. h. die erforderliche Installation des NGINX-Webservers, passiert dabei mittels dieses Shell-Skripts (Listing 7.2):

Listing 7.2 NGINX Installskript: provision-nginx.sh

```
#!/bin/bash
apt-get update
apt-get install -y nginx
```

Es ist üblich, Vagrant aufgrund seines Fokus auf Typ-2-Virtualisierung vor allem auf Entwicklungsrechnern für lokale Teste einzusetzen. Da Vagrantfiles problemlos in Versionskontrollsystmen versioniert werden können, lassen sich so Entwicklern auch problemlos vorbereitete und standardisierte Testumgebungen zur Verfügung stellen. Die verwendeten Basisimages können die gleichen sein, die auch in der Produktionsumgebung genutzt werden. Auf diese Weise lässt sich einfach eine hohe Übereinstimmung zwischen Test- und Live-Umgebungen herstellen.

7.2.4 Provisionierung von Multi-Host-Umgebungen

Terraform ist ein mächtigeres Werkzeug als Vagrant und kann nicht nur einzelne virtuelle Maschinen, sondern ganze Infrastrukturen automatisiert bereitstellen. Mittels Terraform ist es möglich, Infrastrukturen software-defined aufzubauen, zu verändern und zu versorieren. Dabei können sowohl bestehende Cloud-Service-Provider als auch individuelle In-House-Lösungen verwaltet werden. Diese Art der Provisionierung wird bei der Entwicklung Cloud-nativer Anwendungen und Dienste überwiegend dazu genutzt, Produktiv-, Test- oder Staging-Umgebungen für Deployment-Workflows automatisiert bereitstellen zu können.

Mittels Konfigurationsdateien lassen sich dabei Komponenten definieren, die für den Betrieb einer Infrastruktur, einer Plattform oder einer spezifischen Anwendung benötigt werden. Terraform folgt einem deklarativen Ansatz und generiert einen Ausführungsplan, der beschreibt, was zu tun ist, um einen definierten Zustand zu erreichen. Wird der Plan ausgeführt, wird dadurch die beschriebene Infrastruktur aufgebaut. Wenn sich die Konfiguration ändert, ist Terraform in der Lage festzustellen, was sich geändert hat, und kann inkrementelle Ausführungspläne erstellen, die zur Anpassung der Infrastruktur ausgeführt werden können.

Die von Terraform verwaltete Infrastruktur kann sowohl Low-Level-Komponenten wie Compute-Instanzen, Storage und Netzwerke als auch High-Level-Komponenten wie beispielsweise DNS-Einträge, SaaS-Funktionen usw. umfassen.

Die Infrastruktur wird mit einer High-Level-Konfigurationssyntax beschrieben. Dadurch können Infrastruktur-Blueprints versioniert und wie jeder andere Code behandelt werden. Außerdem kann die Infrastruktur gemeinsam genutzt und wiederverwendet werden. In einem Planungsschritt wird ein sogenannter **Ausführungsplan** erzeugt. Der Ausführungsplan zeigt, welche Einzelschritte aktiviert werden, wenn die Provisionierung angestoßen wird. So kann geprüft werden, ob die Infrastruktur wunschgemäß angepasst wird. Ferner wird ein **Ressourcengraph** angeforderter und anzufordernder Ressourcen aufgebaut (siehe Bild 7.6). Damit lassen sich die Erstellung und Änderung aller nicht abhängigen Ressourcen parallelisieren, um die Infrastruktur so zeiteffizient wie möglich zu erstellen. Ferner kann man so einen Einblick in die Abhängigkeiten der Infrastruktur erhalten. Durch den Abgleich von Ausführungsplan und dem Ressourcengraph lassen sich Anpassungen der Infrastruktur automatisiert umsetzen.

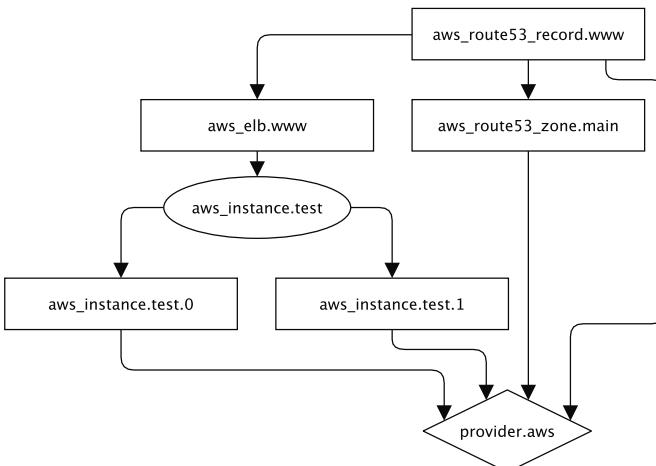


Bild 7.6
Exemplarischer
Ressourcengraph am
Beispiel AWS (Terraform)

Darüber hinaus kann Terraform auch mit **Ressourcen-Schedulern** umgehen. Dies hat Vorteile, denn insbesondere in umfangreichen Infrastrukturen wird die statische Zuordnung von Anwendungen zu Maschinen zunehmend komplexer. Effizientere dynamische Zuordnungen können durch eine ganze Reihe von Schedulern wie beispielsweise Mesos, YARN oder Kubernetes adressiert werden. Diese können verwendet werden, um Docker-Container, Hadoop, Spark und viele andere Software-Tools dynamisch zu planen (siehe auch das folgende Kapitel 9). Da Terraform nicht auf physische IaaS-Ressourcen beschränkt ist, können auch die genannten Ressourcen-Scheduler wie ressourcenbereitstellende Provider behandelt werden. Dadurch kann Terraform auf mehreren Ebenen eingesetzt werden: für die Einrichtung der physischen Infrastruktur, auf der die Scheduler laufen, sowie für die Provisionierung auf das geplante Grid. Nachteilig ist allerdings, dass nun die Trennung zwischen Infrastruktur-Provisionierung und Applikations-Scheduling (siehe Abschnitt 9.1) und Container-Orchestrierung (siehe Abschnitt 9.2) zu verschwimmen beginnt.

Der Terraform-Workflow beruht dabei auf folgenden Schritten:

- **Write:** Beschreibung eines Zielzustands mittels der domänenpezifischen Sprache HCL (HashiCorp Configuration Language)
- **Plan:** Ist-Zustand ermitteln und auf Basis des Ist-Stands notwendige Änderungen planen (entsprechend Abhängigkeiten geordnet und parallelisiert) und darstellen. Die erforderlichen Änderungen an der Infrastruktur können so gesichtet und geprüft werden.
- **Apply:** Idempotente Herstellung des Zielzustands. Der Zustand wird dabei entweder lokal (in einer .tfstate-Datei) oder in einem Remote Store (wie beispielsweise S3, HTTP, Postgres DB ...) gespeichert.

Die Beschreibung des Zielzustands erfolgt im Falle von Terraform in HCL. Mittels HCL kann eine Infrastruktur in Form mehrerer .tf-Dateien definiert werden. Variablen zur Parametrisierung einer Provisionierung können in .tfvars-Dateien hinterlegt werden (z. B. Access Credentials zu IaaS-Infrastrukturen). In diesen Dateien werden die folgenden Kurrentitäten angegeben.

Provider bilden die Schnittstelle zu Cloud-Providern wie AWS, GCE, OpenStack. Hier werden üblicherweise die Access Credentials, Zugriffsprotokolle sowie die Data Centers (Zonen) hinterlegt, wie exemplarisch in Listing 7.3 gezeigt.

Listing 7.3 Providerangaben in HCL

```
provider "google" {
  credentials = file(var.gce_credentials)
  project     = var.gce_project
  region      = var.gce_region
  zone        = var.gce_zone
}
```

Data Sources: Die Verwendung von Data Sources ermöglicht es, Informationen zu nutzen, die außerhalb von Terraform definiert sind oder von einer anderen separaten Terraform-Konfiguration definiert wurden. Diese dienen somit dem Zugriff bzw. der Bekanntmachung von Informationen, die nicht durch Terraform selber in der Infrastruktur angelegt oder abgefragt werden können, sondern extern und global definiert wurden (und daher im Rahmen der

Provisionierung nicht anpassbar sind). Diese Entität wird meist in privaten Infrastrukturen benötigt, weniger in Public-Cloud-Infrastrukturen. Oft werden hier Daten für die Nutzung von Virtualisierungsumgebungen mit geringeren Automatisierungsgraden definiert, wie etwa Netzwerknamen, Datacenter-Bezeichnungen oder IP-Ranges. Die vSphere-Virtualisierungsumgebung von VMWare ist so ein Beispiel. vSphere wird häufig in betrieblichen Kontexten und Rechenzentren kleinerer Firmen genutzt, hat im Allgemeinen aber nur einen sehr eingeschränkten Personenkreis, da die Automatisierung und Multi-Tenancy-Fähigkeit gegenüber vollwertigen IaaS-Infrastrukturen deutlich weniger stark ausgeprägt sind. In diesen Kontexten ist man auf solche Vorgabewerte häufig angewiesen. Diese können, wie in Listing 7.4 gezeigt, gesetzt werden.

Listing 7.4 Data Sources in HCL

```
data vsphere_network "private" {
  name      = "ei-vm-clients"
  datacenter_id = data.vsphere_datacenter.this.id
}

data vsphere_network "public" {
  name      = "dmz_ei_vm2"
  datacenter_id = data.vsphere_datacenter.this.id
}
```

Ressourcen sind das zentrale Element in Terraform. Jeder Ressourcenblock beschreibt ein oder mehrere Infrastrukturobjekte, wie z. B. virtuelle Netzwerke, Compute-Instanzen oder übergeordnete Komponenten wie DNS-Einträge. Wie Listing 7.5 exemplarisch zeigt, deklariert ein Ressourcenblock eine Ressource eines bestimmten Typs (`aws_instance`) mit einem bestimmten lokalen Namen (`web`). Der Name wird verwendet, um von anderen Stellen im gleichen Terraform-Modul auf diese Ressource zu verweisen, hat aber keine Bedeutung außerhalb des Geltungsbereichs dieses Moduls.

Listing 7.5 Ressourcen in HCL

```
resource "aws_instance" "web" {
  ami      = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

Provisioner können verwendet werden, um bestimmte Aktionen auf dem die Provisionierung steuernden Rechner (lokal) oder auf der zu provisionierenden Maschine auszuführen. Auf diese Weise kann man Server oder andere Infrastrukturobjekte für den Betrieb vorbereiten und konfigurieren (siehe Listing 7.6).

Listing 7.6 Provisioner in HCL

```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  [...]

  connection {
    type      = "ssh"
    user      = "ubuntu"
    private_key = file(var.bastion.sshid)
    host     = var.bastion.ip
  }

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get upgrade -y",
      "sudo apt-get install fail2ban -y",
      "sudo ufw allow http",
      "sudo ufw allow https",
      "sudo ufw allow 16443"
    ]
  }
}
```

Terraform unterscheidet sogenannte generische Provisioner, die einfach nur Dateien (z. B. Konfigurationsdateien) auf Rechner übertragen können (`file`) oder die Skripte auf entfernten Rechnern (`remote-exec`) oder dem die Provisionierung steuernden Rechner (`local-exec`) ausführen können. Darüber hinaus lassen sich auch komplexere Konfigurationsmanagementlösungen wie etwa Chef, Salt, Habitat oder Puppet anstelle von einfachen Installskripten als Provisioner nutzen.

■ 7.3 Zusammenfassung

Die Virtualisierung von Infrastrukturen ist die Grundlage des Cloud Computings und insbesondere des Service-Modells IaaS. Dabei wird in Rechenzentren, die Private oder Public IaaS-Services bereitstellen, zumeist die Typ-1-Virtualisierung (Para-Virtualisierung) genutzt. Die Typ-2-Virtualisierung (Voll-Virtualisierung) wird von Entwicklern meist nur auf Entwicklungsmaschinen genutzt, um standardisierte Test- und Entwicklungsumgebungen aufzusetzen. Auf IaaS-Ebene kommt die Betriebssystemvirtualisierung zumeist nicht zum Tragen. Diese wird erst im Rahmen der Containerisierung von Anwendungskomponenten relevant. Hierauf wird in Kapitel 8 eingegangen werden.

Da Private und Public Cloud-Infrastrukturen über APIs gut automatisierbar sind, haben sich sogenannte Infrastructure-as-Code-Ansätze, die auf dem Immutable-Infrastruktur-Paradigma (siehe Abschnitt 7.2.1) beruhen, gebildet. Unter Infrastruktur als Code (IaC) versteht man dabei die Verwaltung und Bereitstellung von IT-Ressourcen mittels maschinenlesbarer Definitionsdateien anstelle von physischer Hardwarekonfiguration oder interaktiven Konfigurationstools. Ähnlich wie bei Pipelines as Code (siehe Kapitel 6) lassen sich so Infrastrukturen mittels Code definieren und automatisiert Betriebs- und Testumgebungen ausrollen. Der Ansatz folgt dabei immer dem Prinzip, dass man ausgehend von einem Basis-Image eine virtuelle Maschine konfiguriert, die ergänzend zum Basis-Image erforderliche Softwarekomponenten inklusive Konfiguration erhält.

Das Thema Infrastructure as Code erhält im Cloud Computing leider nicht dieselbe Aufmerksamkeit, die andere Themen wie etwa Microservices (siehe Kapitel 12) erhalten. Daher ist auch weniger Literatur in diesem Bereich zu finden. Empfohlen sei dem Leser allerdings (Morris 2021), der sich sehr fokussiert dem Thema Infrastructure as Code widmet. Dieser praktische Leitfaden zeigt, wie Prinzipien, Praktiken und Muster, die durch die DevOps-Bewegung entwickelt wurden, effektiv genutzt werden können, um cloud-basierte Infrastrukturen zu verwalten. Wer einen schnelleren (allerdings produktsspezifischen) Einstieg sucht, dem sei (Hashimoto 2013) zum Aufsetzen von Umgebungen auf Entwicklungsrechnern bzw. (Brikman 2019) zum Aufsetzen und zur Skalierung von komplexeren und verteilten Produktiv- oder Testumgebungen empfohlen.

Ergänzende Materialien



<https://bit.ly/3odpoFj>

Auf der Website zum Buch finden sich zu diesem Kapitel ferner folgende Ergänzungsmaterialien:

- Praktische Übungen (Labs) zum Thema software-defined Infrastructure und IaC-basierter Provisionierung
- Übersicht inklusive Links zu IaaS-Infrastrukturen (sowohl Public als auch Private – Self-Hostable – IaaS-Infrastrukturen)
- Übersicht zu Virtualisierungsumgebungen für Entwickler
- Übersicht inklusive Links zu weiteren IaC-basierten Provisionierungslösungen
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0 Lizenz) zum Download für Dozenten und Trainer (Themen: Hintergrund Infrastructure as a Service, Infrastructure as Code, Immutable Architectures , Provisionierungswerzeuge, u. a. Vagrant und Terraform)

8

Standardisierung von Deployment Units (Container)

„Shipping code to the server is hard.“

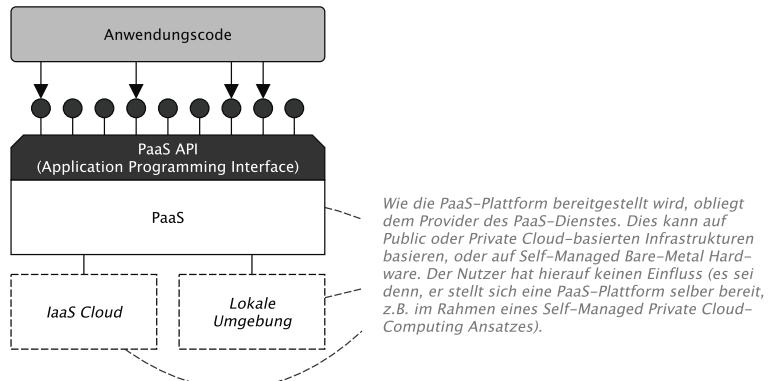
Solomon Hykes, Gründer von Docker

Wie Kapitel 7 gezeigt hat, lassen sich mittels IaaS und Provisionierungsansätzen vordefinierte Deployment-Einheiten (VM-Images) erzeugen und auf Hypervisoren als virtuelle Maschinen ausbringen. Diese Deployment-Einheiten (VM-Images) sind jedoch im Allgemeinen sehr groß (oft deutlich größer als 100 MB bis hin zu mehreren GB), und die Start-up-Zeiten dieser Maschinen sind recht lang (bis zu mehreren Minuten). Ferner wollen sich Entwickler häufig nicht mit den Details der Infrastruktur „herumschlagen“, sondern einfach nur Code zur Ausführung bringen. Aus diesem Grund sind bereits sehr früh im Cloud Computing sogenannte Platform as a Service-Angebote (PaaS) entstanden.

■ 8.1 Hintergrund (PaaS)

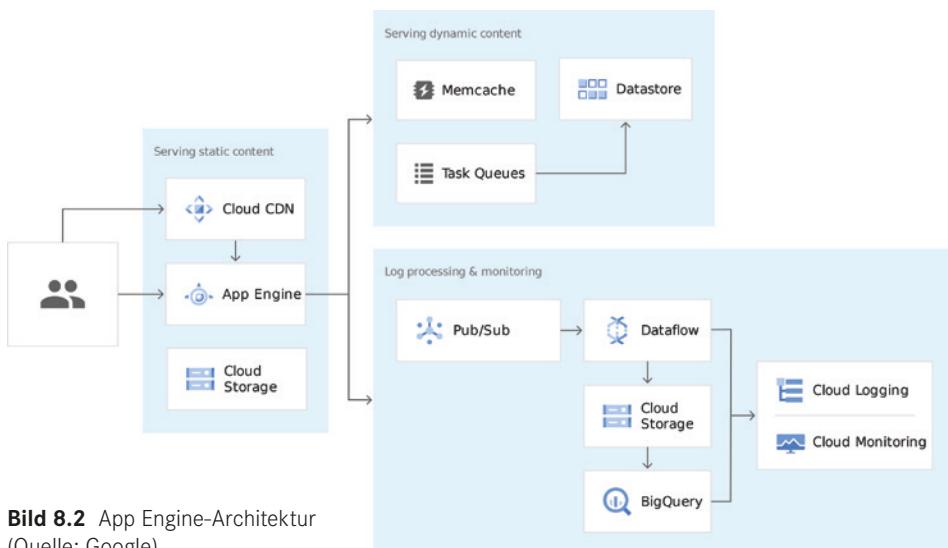
Als Platform as a Service (PaaS) bezeichnet man eine Dienstleistung, die als Cloud-Service eine Plattform für Entwickler von Anwendungen zur Verfügung stellt. Dabei kann es sich sowohl um Laufzeitumgebungen (häufig für Webanwendungen), aber auch um Entwicklungsumgebungen handeln, die mit geringem administrativem Aufwand und ohne Anschaffung der darunterliegenden Hardware und Software genutzt werden können (siehe auch Bild 2.3). Sie unterstützen wesentliche Teile des Software-Lebenszyklus von der Entwicklung, dem Test, der Auslieferung bis hin zum Betrieb der Anwendungen über das Internet.

PaaS-Dienste bieten somit eine Ad-hoc-Entwicklungs- und Betriebsplattform für den gesamten Lebenszyklus von Anwendungen. Anwendungen werden dabei per Applikationspaket oder als Quellcode deployt. Es ist zumeist kein Image für eine technische Infrastruktur notwendig. Die Anwendung sieht nur Programmier- oder Zugriffsschnittstellen einer PaaS-spezifischen Laufzeitumgebung (siehe Bild 8.1). Die PaaS-Laufzeitumgebung kann auch für eine automatische Skalierung der Anwendung mittels automatischer Provisionierung der Infrastruktur sorgen (siehe Kapitel 7). PaaS-Plattformen bieten ferner Entwicklungswerkzeuge (IDEs, Build-Systeme, Testumgebung) und Schnittstellen zur Administration und zum Monitoring von Anwendungen.

**Bild 8.1** PaaS-Plattform

PaaS-Plattformen gibt es viele, sowohl als kommerzielle Managed Services, aber auch als self-hosted Plattformen. Allein zwischen Oktober 2009 und Oktober 2010 sollen mehr als 100 PaaS-Anbieter den Markt betreten haben. Eine Übersicht aller PaaS-Plattformen kann der Leser in (Kolb 2019) nachvollziehen. Die Liste ist so umfangreich, dass hier auf eine tabellarische Aufstellung verzichtet wird. Alle PaaS-Plattformen werben aber letztlich mit dem Ziel, Kunden möglichst viele administrative Aufgaben abzunehmen, Skalierbarkeit und Hochverfügbarkeit zu ermöglichen, Fixkosten und Gesamtkosten zu senken, die Anwender flexibler zu machen und diesen eine schnelle Anwendungsentwicklung und einen baldigen Markteintritt (Time-to-Market) zu ermöglichen.

Die Google App Engine (GAE) ist beispielsweise das PaaS-Angebot von Google (siehe Bild 8.2). Wir nutzen GAE als Typrepräsentant für diese Art von Cloud-Service. Alle anderen Plattformen und Dienste folgen vom Grundsatz ähnlichen Überlegungen und technischen Einschränkungen. Bei GAE laufen Anwendungen innerhalb der Google-Infrastruktur.

**Bild 8.2** App Engine-Architektur
(Quelle: Google)

Der Betrieb der Anwendungen ist sogar innerhalb bestimmter Quoten kostenfrei. Dies macht es attraktiv, erste Experimente in der PaaS-Welt anzugehen. Für Cloud-Service-Provider ist dies wiederum eine Möglichkeit, eine erste Kundenbindung aufzubauen. Daher folgen viele Provider dem Prinzip gewisser Freikontingente. Der grundsätzlich geringere Ressourcenbedarf von PaaS im Vergleich zu IaaS-Angeboten macht dies für Provider kalkulierbar. PaaS-Plattformen unterstützen mittlerweile alle gängigen Programmiersprachen wie z. B. JavaScript, Ruby, Go .NET, Java, Python, PHP. Im Falle von GAE stehen Integrationen für alle gängigen IDEs zur Verfügung, oft bieten Provider aber auch WebIDEs an, um das Angebot noch attraktiver und friktionsfreier für den Kunden zu gestalten.

Da die Anwendungen aus Sicht des Service-Providers beliebige Workloads ausführen, können damit jedoch auch Schadcodes ausgeführt werden, die für Security Breaches oder Denial-of-Service-Angriffe auf andere oder eigene Systeme genutzt werden. Provider müssen dies schon aus rechtlichen Gründen wirksam unterbinden können. Es besteht natürlich auch aus Image-Gründen meist kein Interesse, die eigenen Dienste in der Nähe von „Hacking-Services“ zu positionieren. Daher laufen Kundenapplikationen meist in einer für den Service-Betreiber gut kontrollierbaren Sandbox, die das Verhalten der Applikation mit dem Ziel einschränkt, die Verarbeitung effizient zu halten und die eigene Infrastruktur und das Internet zu schützen. Es dürfen daher oft auch nicht alle Klassen von Standardbibliotheken genutzt werden.

GAE hat beispielsweise folgende Einschränkungen: Applikationen dürfen keine eigenen Threads öffnen und erhalten keinen Zugriff auf die Laufzeitumgebung (z. B. mittels Java Class Loader). Ferner wird die Kommunikation mit anderen Webanwendungen eingeschränkt und ist nur über – für den Provider gut kontrollierbare – Kanäle wie URL Fetch, XMPP oder E-Mail zulässig. Die zur Verfügung stehenden Kommunikationsprotokolle sind also extrem eingeschränkt. Ferner existieren zumeist Größenlimitierungen bei Anfragen und Antworten (bei GAE dürfen diese nicht größer als 1 MB sein). Als allgemeines Architekturmittel für eingehende Kommunikation haben sich daher sogenannte Web-Hooks etabliert, die bei Ereignissen (Warm-up, Messages, Cron-Jobs etc.) auch durch die PaaS-Laufzeitumgebung angestoßen werden. Ferner laufen Requests an eine GAE-Anwendung nach 60 Sekunden in einen Timeout. Und es gibt weitere diverse Einschränkungen zu Datenvolumina und Anzahl von Service-Aufrufen. Diese Arten von Einschränkungen und Limitierungen können sich dabei von PaaS-Provider zu Provider substantiell unterscheiden. Darüber hinaus ist es oft erforderlich, dass man sich bei Applikationen von speziellen providerspezifischen Bibliotheken abhängig machen muss, die einen Transfer einer Applikation von einem Provider zu einem anderen Provider zu einer teuren „One-time Exercise“ machen können.

Wesentliche Probleme bei PaaS sind seit den ersten Tagen dieses Service-Modells u. a. fehlende Standards. Daran hat sich auf der Ebene PaaS bis heute eigentlich nie nennenswert etwas getan. Dies betrifft insbesondere

- das Deployment-Format der zu hostenden Anwendungen
- und die PaaS-Runtime-Schnittstelle.

Um einen Eindruck hiervon zu gewinnen, kann der Leser einfach selber einen simplen Feldversuch mit GAE und Heroku machen. GAE und Heroku sind beides große, beliebte und qualitativ hochwertige PaaS-Provider. Wenn man aber versucht, das Getting-Started von GAE auf Heroku und umgekehrt zum Laufen zu bringen, hat man anschließend ein sehr gutes Gefühl vom Begriff „Vendor Lock-in“. Gerade bei „PaaS ist die heterogene Anbieterlandschaft ein Hindernis für die Anwendungsportabilität“ (Kolb 2018).

In den letzten Jahren zeigt sich bei PaaS-Plattformen jedoch ein erkennbarer Trend, der es Kunden ermöglicht, eigene Laufzeitumgebungen zu erweitern. Die sich dabei offensichtlich stellende Frage ist, wie PaaS-Provider dies überhaupt technisch ermöglichen können, ohne die erforderliche Kontrolle über die auszuführenden Applikationen zu verlieren und nicht an dem enorm steigenden Dependency-Management beliebiger Laufzeitumgebungen zu scheitern. Diese erweiterbaren Laufzeitumgebungen basieren technisch zumeist auf Container-basierten Ansätzen, die insbesondere durch Docker im Cloud Computing eine substanziale Bedeutung erhalten haben. PaaS entwickelt sich so zunehmend zu CaaS (Container as a Service) weiter. Ein Begriff, der zum Zeitpunkt der NIST-Definition (Mell und Grance 2011) des Cloud Computings noch nicht einmal existierte, da die Container-Technologie 2011 noch keine nennenswerte Rolle spielte.

■ 8.2 Betriebssystem-Virtualisierung

Bei Container as a Service (CaaS) handelt es sich um ein Cloud-Computing-Modell, mit dem sich container-basierte Virtualisierung als Service aus der Cloud nutzen lässt. Wie bei allen anderen Service-Modellen auch, kann CaaS sowohl als Managed Service (z. B. im Rahmen einer Public Cloud) oder als self-hosted Service (z. B. im Rahmen einer Private Cloud) bezogen werden. Nutzer von Managed Service CaaS benötigen keine eigene Infrastruktur für die Container-Virtualisierung und müssen keine eigene Container-Plattform betreiben. Sämtliche Ressourcen für die Virtualisierung wie Rechenleistung, Speicherplatz, Container-Engine und Orchestrierungssoftware stellt in diesem Fall der Cloud-Service-Provider zur Verfügung. Es ist somit möglich, CaaS als Service aus einer Public Cloud oder aus einer Private Cloud zu beziehen. Container as a Service liegt damit konzeptionell zwischen den Modellen Infrastructure as a Service (IaaS) und Platform as a Service (PaaS), besitzt aber das Potenzial, aufgrund besserer Portierbarkeit und besserer Standardisierung das PaaS-Modell langsam zu verdrängen (siehe Bild 8.3).

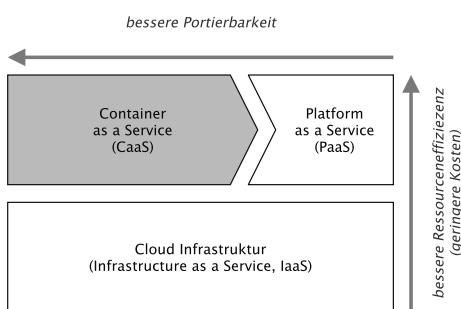


Bild 8.3
Von PaaS zu CaaS

Insbesondere für das Cloud-native Computing hat das besser portierbare CaaS vermutlich die höhere Bedeutung als das „Vendor Lock-in“-anfällige PaaS, weswegen wir uns in diesem Kapitel vor allem auf CaaS und dessen technische Hintergründe fokussieren.

Container basieren auf Betriebssystemvirtualisierung. Die Betriebssystemvirtualisierung entstammt insbesondere dem Mainframe-Umfeld und dient dazu, ein Gesamtsystem in mehrere Teilsysteme in Form lauffähiger Betriebssysteminstanzen zu partitionieren. Die Ressourcen (Prozessor, Hauptspeicher, I/O-System) werden über die Hardware/Firmware den Systeminstanzen zugeordnet. Diese Art der Virtualisierung wurde insbesondere in IBM Mainframes (zSeries) aufgrund folgender Vorteile angewendet:

- Bessere Hardwareauslastung durch Serverkonsolidierung
- Weniger Leistungsaufnahme für Rechner und Klimatisierung (Stromverbrauch)
- Flexibilität bei Aufbau einer Infrastruktur, da Systeminstanzen schnell bereitgestellt, beliebig vervielfältigbar und archivierbar sind
- Vereinfachung der Wartung u. a. durch Life-Migrationen sowie Verfügbarkeits- und Ausfallsicherheitskonzepte

Das Konzept wurde aber auch in unixoiden Betriebssystemen wie beispielsweise Sun Solaris (Containers), Linux (LXC, VServer, OpenVZ, libcontainer), FreeBSD (Jails) und zur Isolation von Laufzeitumgebungen in Form sogenannter Container adaptiert. In Containern können voneinander isolierte Anwendungen laufen, ohne dass – im Gegensatz zur Voll- oder Para-Virtualisierung (siehe Abschnitt 7.1.1) – ein zusätzliches Gastbetriebssystem erforderlich wäre. Dies macht Container deutlich ressourcenschonender und kleiner als virtuelle Maschinen (siehe Bild 8.4). Insbesondere der Leistungsverlust beim CPU/RAM-Overhead ist in der Regel gegenüber der nativen Ausführung nicht messbar.

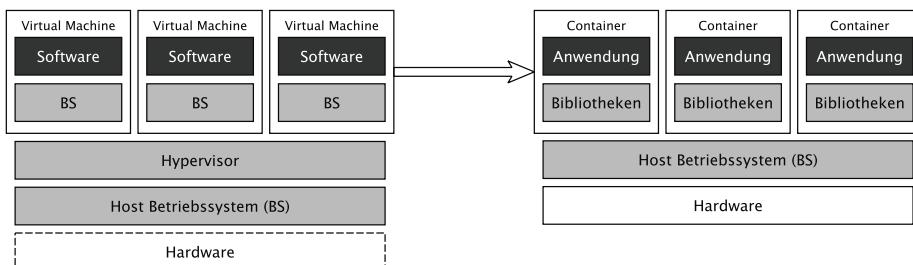


Bild 8.4 Von virtuellen Maschinen zu Containern

■ 8.3 Container Runtime Environments

Insbesondere Docker hat der Betriebssystemvirtualisierung auf Linux zu einer Renaissance verholfen. Docker umfasst eine Reihe von Produkten (Docker Daemon/Runtime Environment, Container und Images, Registry/Image Repository), die Virtualisierung auf Betriebssystemebene nutzen, um Software in sich geschlossenen Paketen ohne externe Abhängigkeiten, sogenannten Containern, bereitzustellen. Container sind voneinander isoliert und bündeln ihre eigene Software, Bibliotheken und Konfigurationsdateien. Da alle Container die Dienste eines einzigen Betriebssystemkerns nutzen, verbrauchen sie weniger Ressourcen als virtuelle

Maschinen. Ein einzelner Server oder eine virtuelle Maschine kann somit mehrere Container gleichzeitig ausführen. Man kann dadurch auch Processing-Ressourcen wesentlich feingranularer und effizient unterhalb der 1 vCPU-Schwelle bereitstellen (siehe Abschnitt 2.3.2).

Docker hat es als Toolset zum ersten Mal für eine breite Entwicklerschicht komfortabel ermöglicht, eine Anwendung und ihre Abhängigkeiten in einem Container-Image zu bündeln, das auf jedem Linux-, Windows- oder macOS-Computer ausgeführt werden kann. Unter macOS muss zwar eine virtuelle Linux-Maschine zur Ausführung der Container verwendet werden; mittels des Windows Subsystem for Linux (WSL2) lassen sich aber mittlerweile auch Linux-Kernel und somit Linux Container über die Windows Hyper-V-Funktionen auf Windows-Systemen ausführen. Docker nutzt die Ressourcen-Isolierungsfunktionen des Linux-Kernels (u. a. cgroups und Kernel-Namensräume) sowie ein union-fähiges Dateisystem (z. B. OverlayFS), um die Ausführung von Containern innerhalb einer einzelnen Linux-Instanz zu ermöglichen und so den Overhead beim Starten und Verwalten virtueller Maschinen zu minimieren.

In Containerplattformen wie etwa Kubernetes wird Docker jedoch mittlerweile zunehmend durch andere Laufzeitumgebungen ersetzt, da sich insbesondere der Docker Daemon-Ansatz in vielen Anwendungsfällen (vor allem in Build Pipelines) letztlich als hinderlich erwiesen hat. Dass dieser Technologie-Austausch im Gegensatz zu sonstigen PaaS-Plattformen funktioniert, hat sicher auch mit der erfolgreichen Standardisierung des Container-Image-Formats (OCI 2016a) und der Container-Runtime-Schnittstelle (OCI 2016b) durch die Open-Container-Initiative (OCI) zu tun. Die OCI konnte somit exakt die Komponenten standardisieren, die bei PaaS-Plattformen bisher kaum standardisiert wurden. Mittels Container-Images können Anwendungen so gepackt werden, dass sie problemlos an verschiedenen Orten und auf verschiedenen Systemen mit einer Container Runtime Engine ausgeführt werden, z. B. in einer öffentlichen oder privaten Cloud, auf Entwicklungsrechnern oder in Staging- oder Produktionsumgebungen. Das Dependency-Management vereinfacht sich dadurch aus Sicht eines Entwicklers erheblich.

Prozesse von Applikationen, die auf unterschiedlichen virtuellen Maschinen laufen, sind naturgegeben besser voneinander isoliert als Prozesse, die auf demselben Host laufen und sich auch dasselbe Betriebssystem teilen. Das ist grundsätzlich einfacher durchsetzbar. Bei der Betriebssystemvirtualisierung müssen Prozesse unterschiedlicher Applikationen mittels OS-Mechanismen „künstlich“ voneinander isoliert werden, da sie beispielsweise alle Zugriff auf dasselbe Dateisystem und dieselbe Netzwerkschnittstelle haben. Da das Cloud-native Umfeld stark von Linux geprägt ist, werden diese Isolationsmöglichkeiten am Beispiel von Mechanismen des Linux-Kernels erläutert. Die Isolation von nicht zusammengehörenden Prozessen kann dabei durch den Kernel grundsätzlich mittels Sichtbarkeit (Namespaces), mittels Rechten (Control Groups), mittels Ressourcenlimitierungen (Control Groups) und mittels Union-Filesystems erfolgen.

8.3.1 Kernel-Namespaces

Namespaces ermöglichen die Bereitstellung globaler Systemressourcen (z. B. Netzwerk, Volume Mounts) als eigene isoliert erscheinende Ressource für alle Prozesse in einem Namespace. Ein Namespace umschließt dabei eine globale Systemressource mit einer Abstraktion, die den Prozessen im Namespace den Eindruck vermittelt, dass sie über eine eigene isolierte

Instanz der globalen Ressource verfügen. Änderungen an der globalen Ressource sind nur für die Prozesse sichtbar, die Mitglieder des Namespaces sind, für andere Prozesse jedoch nicht. Mittels Namespace-Isolation kann eine Gruppe von Prozessen so voneinander getrennt werden, dass sie keine Ressourcen in anderen Gruppen „sehen“ kann. Ein PID-Namensraum bietet zum Beispiel eine separate Aufzählung von Prozesskennungen innerhalb jedes Namensraums. Die Verwendung von Namespaces erlaubt es somit, Container voneinander zu isolieren, obwohl sie dasselbe Betriebssystem teilen. Linux kann u. a. die in Tabelle 8.1 gezeigten Ressourcen in erläuterter Art in Namespaces zusammenfassen und isolieren.

Tabelle 8.1 Isolation mittels Namespaces

| Namespace | Was wird isoliert? |
|-----------|---|
| Cgroup | Cgroup root directory |
| IPC | System V Interprocess Communication, POSIX message queues |
| Network | Network devices, stacks, ports, etc. |
| Mount | Mount points |
| PID | Process IDs |
| Time | Boot and monotonic clocks |
| User | User and group IDs |
| UTS | Hostname and domain name |

8.3.2 Process Capabilities

Capabilities sind spezielle Attribute im Linux-Kernel, die Prozessen und ausführbaren Binärdateien bestimmte Berechtigungen gewähren, die normalerweise nur Prozessen des Root-Users vorbehalten sind. Das Ziel von Capabilities ist es, die Macht von „root“ in spezifische Privilegien aufzuteilen, sodass, falls ein Prozess ausgenutzt wird, der potenzielle Schaden im Vergleich zu demselben Prozess, der als root laufen würde, begrenzt ist. Capabilities können auf Prozesse und ausführbaren Dateien gesetzt werden. Ein Prozess, der durch die Ausführung einer Datei entsteht, kann die Capabilities dieser Datei erlangen.

Die in Linux implementierten Capabilities sind zahlreich, und viele wurden seit der ursprünglichen Veröffentlichung hinzugefügt. Folgende Aufzählung soll nur einen exemplarischen Eindruck geben.

- CAP_CHOWN: Änderungen an der Benutzer-ID und Gruppen-ID von Dateien vornehmen
- CAP_KILL: Umgehen von Berechtigungsprüfungen für das Senden von Signalen an Prozesse
- CAP_SYS_TIME: Setzen der System- und Echtzeit-Hardware-Uhr

Capabilities werden Sets (u. a. „permitted“, „inheritable“ und „effective“) zugewiesen und haben viele Einsatzmöglichkeiten. So können Capabilities auch dazu beitragen, die Sicherheit von Systemen zu erhöhen, indem Sie u. a. die „Superrechte“ des root-Users in beherrschbare Teilrechte aufteilen und man so den Ganz-oder-gar-nicht-Rechteansatz des privilegierten Super-Users feingranularer definieren kann.

8.3.3 Control Groups

Mittels Control Groups lassen sich in Linux Prozesse gruppieren (z. B. alle Prozesse einer Anwendung, eines Dienstes usw.) und Ressourcen-Limits (Quotas) für diese Gruppen festsetzen und durchsetzen. Control Groups (abgekürzt cgroup) sind somit eine Sammlung von Prozessen, die durch dieselben Kriterien gebunden und mit einem Satz von Parametern oder Limitierungen verknüpft sind. Diese Gruppen können hierarchisch aufgebaut sein, d. h., jede Gruppe erbt Grenzwerte von ihrer übergeordneten Gruppe. Der Kernel bietet über die cgroup-Schnittstelle Zugang zu mehreren Controllern (auch Subsysteme genannt). Zum Beispiel begrenzt der memory-Controller die Speichernutzung, cpuacct die CPU-Nutzung usw. (siehe auch Bild 8.5).

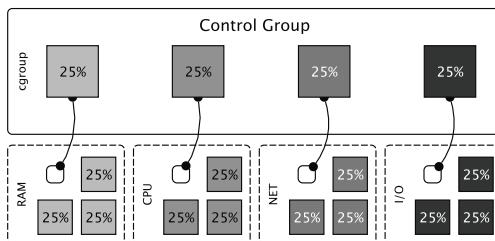


Bild 8.5
Linux Control Groups

Control Groups ermöglichen dem Kernel folgende Aspekte:

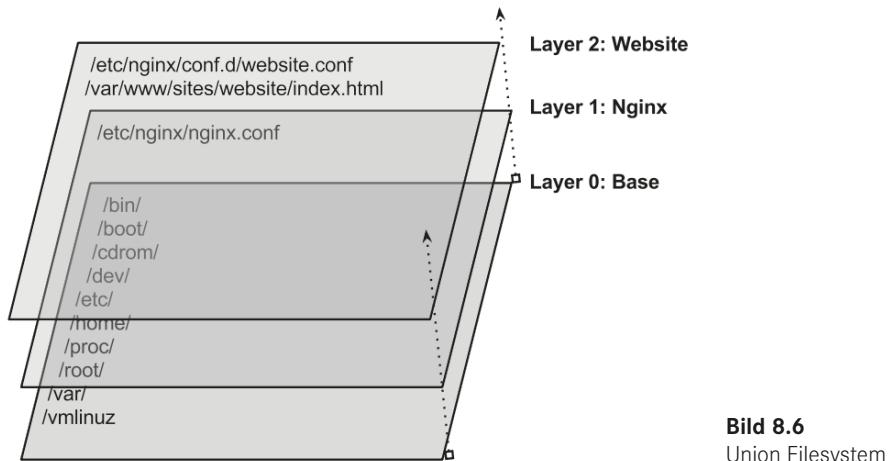
- **Ressourcenbeschränkung:** Prozessgruppen können so eingestellt werden, dass sie ein konfiguriertes Speicherlimit nicht überschreiten können.
- **Priorisierung:** Gruppen können einem festgelegten Anteil der CPU-Auslastung oder dem I/O-Durchsatz zugewiesen werden.
- **Beobachtung:** Der Kernel misst, wie viele Ressourcen bestimmte Gruppen verbrauchen. Dies kann beispielsweise für Abrechnungszwecke oder einfach nur Reporting oder Monitoring verwendet werden.
- **Kontrolle:** Der Kernel kann Prozessgruppen bei Überschreiten definierter Grenzen einfrieren (inklusive Checkpointing) und Neustarts von Prozessgruppen veranlassen.

8.3.4 Union Filesystem

Union Filesystems werden in Betriebssystemen dazu verwendet, Prozessen eigene Namensräume innerhalb von Dateisystemen zuzuweisen. Dadurch können die Dateien verschiedener Dateisysteme zu einem einzigen logischen Dateisystem vereinigt werden. Dateien, die zwar in getrennten Dateisystemen, aber im gleichen Verzeichnis liegen, werden dadurch im selben Verzeichnis angezeigt. Dabei werden den einzelnen beteiligten Layern Prioritäten zugeordnet, sodass eine eindeutige Zuordnung auch im Falle gleicher Dateinamen gewährleistet ist. Dabei überschreiben höhere Layer tiefere Layer (siehe auch Bild 8.6).

Mittels des sogenannten **Copy-on-Write** lassen sich Container vom Host isolieren. Jeder Container blendet erforderliche Dateien der Basis als In-Memory-Kopien ein und isoliert so sein Dateisystem vom (persistenteren) Host-Dateisystem. Werden diese Dateien durch den

Container geändert, erfolgt dies nur In-Memory und nicht auf dem persistenten Storage des Hosts. Auf diese Weise kann ein Container nicht das Dateisystem des Hosts ändern. Er ist „stateless“ und „vergisst“ somit geschriebene Dateien nach einem Neustart und isoliert sich so vom Dateisystem des Hosts.



8.3.5 High-Level- und Low-Level-Container-Laufzeitumgebungen

Wie wir gesehen haben, sind Container keine First-Class-Objekte im Linux-Kernel. Die gezeigten Isolationsmechanismen sind erst mit der Evolution des Linux-Kernels entstanden. Container bestehen im Wesentlichen aus mehreren zugrunde liegenden Kernel-Grundelementen: Namespaces, Control Groups und weitere LSMs (Linux Security Modules). Zusammen ermöglichen diese Kernel-Grundelemente eine sichere, isolierte und überwachte Ausführungsumgebung für Prozesse.

Aber all dies jedes Mal manuell zu tun, wenn ein neuer isolierter Prozess erstellt werden soll, wäre sehr mühsam. Aus diesem Grund sind sogenannte

- Low-Level (host-fokussiert) und
- High-Level (plattform-fokussiert)

Container Runtime Environments entstanden. Docker mag die vielleicht bekannteste und meist genutzte Container-Laufzeitumgebung sein, aber sie ist bei Weitem nicht die einzige! Grundsätzlich ist eine Container-Runtime eine Software, die die zum Ausführen von Containern erforderlichen und erläuterten Betriebssystemtechniken nutzt, ausführt und verwaltet. Diese Tools erleichtern somit die sichere Ausführung und effiziente Bereitstellung von Containern unter Verwendung der genannten Kernel-Techniken.

Low-Level OCI-Runtimes wie beispielsweise runC (Docker) fokussieren die Erstellung und Ausführung von Containern auf einem Host. Neben nativen Runtime Environments, die den containerisierten Prozess auf demselben Host-Kernel ausführen, gibt es auch noch einige Sandbox- und virtualisierte Implementierungen der OCI-Spezifikation. gVisor und Nabla sind Beispiele für Sandbox-Runtime Environments, die eine weitere Isolierung des Hosts vom containerisierten Prozess ermöglichen. Der containerisierte Prozess wird auf einer

Unikernel- oder Kernel-Proxy-Schicht ausgeführt, die dann im Namen des Containers mit dem Host-Kernel interagiert. Daher haben diese Runtimes eine reduzierte Angriffsfläche und schützen so den Host. runV oder kata sind Beispiele für virtualisierte Runtime Environments. Dies sind Implementierungen der OCI-Runtime-Spezifikation, die von einer Schnittstelle der virtuellen Maschine unterstützt werden. Sie starten eine virtuelle Maschine mit einem Standard-Linux-Kernel-Image und führen den „containerisierten“ Prozess in dieser VM aus. Sowohl sandboxed als auch virtualisierte Implementierungen präferieren somit eine bessere Isolation gegenüber der Ressourceneffizienz nativer OCI-Runtimes.

High-Level Container Runtime Interfaces (CRI) fokussieren die Erstellung und Ausführung von Containern auf geclusterten Container-Plattformen (z. B. Kubernetes). Mit diesen Plattformen wird sich das folgende Kapitel 9 befassen. Im Gegensatz zu OCI-Runtimes, die das Erstellen von Containern auf einem Host fokussieren, verfügt ein CRI daher über die erforderliche Funktionalität, um Container in dynamischen Umgebungen auszubringen. Darüber hinaus delegieren CRIs normalerweise die eigentliche Containerausführung an eine Low-Level OCI-Runtime.

■ 8.4 Bau und Bereitstellung von Container-Images

Mittels der Isolationsmechanismen des Linux-Kernels kann man also Prozesse innerhalb eines Betriebssystems auf einem Host so ablaufen lassen, dass die Prozesse den Eindruck haben, sie würden alleine und damit vollkommen isoliert auf dem Host laufen. Diese Isolationsmechanismen sind jedoch im alltäglichen Betrieb recht anspruchsvoll und damit auch fehleranfällig in der Nutzung. Mittels Container Runtime Environments lässt sich dies in den Griff bekommen. Diese Runtime Environments benötigen jedoch im Allgemeinen vorbereitete Images, die mit den gezeigten Isolationsmechanismen des Kernels zur Ausführung gebracht werden.

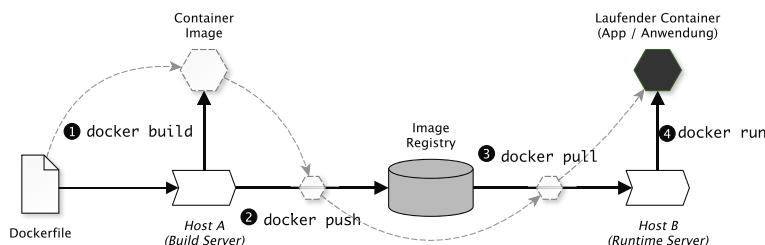


Bild 8.7 Docker-Workflow

Wir werden daher den Bau und die Bereitstellung von Komponenten mittels Container-Images erläutern. Dies erfolgt am Beispiel des Docker-Workflows (siehe Bild 8.7), da nämlich fast alle Container-Lösungen das Dockerfile-Format unterstützen. Grundsätzlich sind andere Verfahren möglich, um Container-Images für Container Runtime Environments zu bauen,

doch dieses Format ist recht gut für den Menschen lesbar und nah an der manuellen Tätigkeit, selber eine Konfiguration händisch vorzunehmen. Man hat also eine gute Beziehung zwischen der Konfigurationsaktivität und dem Resultat, was insbesondere der Nachvollziehbarkeit dienlich ist.

Docker kann Images erstellen, indem es hierfür erforderliche Anweisungen aus einem Textdokument, dem sogenannten `Dockerfile`, liest. Listing 8.1 zeigt solch ein exemplarisches `Dockerfile` zur Erstellung einer Python-basierten Webapplikation. Ein `Dockerfile` beinhaltet also alle Befehle, die ein Benutzer auf der Kommandozeile aufrufen würde, um ein Image manuell zu erstellen.

Listing 8.1 Einfaches Dockerfile

```
# Base image
FROM alpine:3.5

# Install python and pip
RUN apk add --update py2-pip

# Upgrade pip
RUN pip install --upgrade pip

# Install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# Copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# Tell the port number the container should expose
EXPOSE 5000

# Run the application
CMD ["python", "/usr/src/app/app.py"]
```

Der Befehl `docker build` erstellt aus solch einem Dockerfile und einem Kontext ein Container-Image. Der Kontext ist ein Satz von Dateien an einem bestimmten Ort. Dieser Ort kann ein Verzeichnis auf dem lokalen Dateisystem (meist das Repository selber) oder die URL eines entfernten Git-Repositorys sein. Dieses Image kann genutzt werden, um mittels des `docker run`-Befehls lokal ausgeführt zu werden. Üblicher ist es aber, den Build in Deployment-Pipelines anzustoßen, dort das Image in eine sogenannte Container Registry zu pushen und das Image von dort zu ziehen, um es in beliebigen Container Runtime Environments auszuführen.

Eine Container Registry ist ein Repository zum Speichern von Container-Images. Ein Container-Image besteht aus kontextbezogenen Dateien inklusive deren Abhängigkeiten, die alle für eine Anwendung in einem Image gekapselt werden. Nachdem ein Host ein Image in eine Registry mittels `docker push` gestellt hat, können andere Hosts es von dieser Registry mittels `docker pull` herunterladen. Dadurch kann dieselbe Anwendung inklusive ihrer Abhängigkeiten in einem in sich geschlossenen Format von einem Host zu einem anderen in Form einer standardisierten Deployment Unit transferiert und ausgeführt werden.

■ 8.5 Faktoren gut betreibbarer Container

Mittels Container-Images lassen sich also Anwendungen in Form standardisierter Deployment Units bereitstellen. Dies kann man grundsätzlich für alle Anwendungen machen. Und wenn diese tatsächlich isoliert voneinander laufen und untereinander keine Wechselwirkungen haben, ist auch kein sonderlich hohes Fehlerpotenzial gegeben. Im Falle Cloud-nativer Systeme handelt es sich jedoch zumeist um verteilte Systeme, deren Komponenten sehr wohl Wechselwirkungen untereinander sowie spezifische Besonderheiten im Betrieb haben. Diese Aspekte werden noch ausführlich in Teil III behandelt werden. Dennoch lohnt es sich, bereits an dieser Stelle Muster zu betrachten, die es einem ermöglichen, Container in Cloud-nativen Szenarien möglichst friktionsfrei einzusetzen. Hierzu ist es nicht ausreichend, nur die Abhängigkeiten in sich geschlossener Deployment Units in Form von Containern zu packen, sondern auch zu berücksichtigen, wie diese Deployment Units im Betrieb eingesetzt werden.

In diesem Zusammenhang ist die sogenannte Zwölf-Faktoren-Methodik (Wiggins 2017) zu nennen, die bei Anwendungen genutzt werden kann, die in einer beliebigen Programmiersprache geschrieben sind und eine beliebige Kombination von Backing-Diensten (Datenbank, Messaging, Queueing, Speicher-Cache usw.) verwenden. Das Besondere ist, dass diese Methodik sehr gut auf die Standardisierung von Deployment Units (Container) abgestimmt ist, weswegen wir uns diese Erfahrungen und Empfehlungen stark zusammengefasst ansehen wollen, da sie hilfreich sind, qualitativ hochwertige Komponenten zu entwickeln. Die Zwölf-Faktoren-Methodik unterstützt die Erstellung von As-a-Service-Apps, die:

- **deklarative Formate** für die Setup-Automatisierung verwenden, um Zeit und Kosten für neue Entwickler, die dem Projekt beitreten, zu minimieren;
- einen **sauberen Vertrag** mit dem zugrunde liegenden Betriebssystem haben, der maximale Portabilität zwischen Ausführungsumgebungen bietet;
- die sich für den Einsatz auf modernen **Cloud-Plattformen** eignen und damit den Bedarf an Servern und Systemadministration überflüssig machen;
- die **Divergenz zwischen Entwicklung und Produktion** minimieren und eine kontinuierliche sowie agile Bereitstellung ermöglichen;
- Anwendungen ohne signifikante Änderungen an Werkzeugen, Architektur oder Entwicklungspraktiken **horizontal skalieren** können.

Diese Methodik fasst Erfahrungen und Beobachtungen zu einer Vielzahl von Software-as-a-Service-Apps in „freier Wildbahn“ zusammen (Wiggins 2017). Dabei wird im Sinne des DevOps (siehe Kapitel 3) besonderes Augenmerk auf die Dynamik der organischen Evolution einer Anwendung im Laufe der Zeit, die Dynamik der Zusammenarbeit zwischen Entwicklern sowie der Vermeidung von „Software-Erosion“ gelegt.

8.5.1 Codebase

Die Zwölf-Faktor-Methodik betrachtet im ersten Faktor die Codebase als zentrales Instrument einer Anwendung. Eine Zwölf-Faktor-App wird immer in einem Versionsmanagementsystem als Repository (Codebase) verwaltet (z. B. git). Jede App hat dabei genau eine Codebase. Jede Komponente in einem verteilten System ist eine Anwendung und kann für sich den zwölf Faktoren entsprechen. Wenn mehrere Anwendungen denselben Code teilen, verletzt dies die zwölf Faktoren. Dies lässt sich durch Codeauslagerungen in Bibliotheken und Abhängigkeitsverwaltung lösen.

Als Deploy wird eine laufende Instanz der Anwendung bezeichnet. Es gibt zwar nur eine Codebase pro Anwendung, aber viele mögliche Deploys der Anwendung (siehe Bild 8.8). Meist gibt es Produktionssysteme, Staging-Systeme sowie lokale Entwicklungssysteme von Entwicklern für Deploys. Die Codebase ist über alle diese Deploys hinweg dieselbe, auch wenn bei jedem Deploy unterschiedliche Versionen (Branches, siehe Kapitel 6) aktiv sind. So umfasst die Staging-Umgebung meist mehr Commits als die Produktionsumgebung. Aber alle Umgebungen teilen dieselbe Codebase, was sie als verschiedene Deploys derselben Anwendung auszeichnet.

Solche Arten von Anwendungen lassen sich gut als Container-Images in Deployment-Pipelines bauen und aus diesen in unterschiedlichen Deployment-Umgebungen bereitstellen.

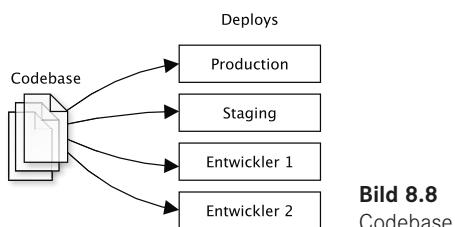


Bild 8.8
Codebase

8.5.2 Abhängigkeiten und Konfigurationen

Die Zwölf-Faktor-Methodik betrachtet in den Faktoren 2 und 3 Abhängigkeiten und Konfigurationen von Anwendungen. Die zentrale Empfehlung lautet dabei, dass Abhängigkeiten explizit deklariert und isoliert werden sollten. Die meisten Programmiersprachen bieten Dependency-Management-Systeme an, um unterstützende Bibliotheken zu verbreiten (Rubygems für Ruby, pip für Python, pub für Dart usw.).

Eine Zwölf-Faktor-App verlässt sich allerdings nie auf die Existenz von systemweiten Paketen, sondern deklariert alle Abhängigkeiten vollständig und korrekt über eine Abhängigkeitsdeklaration. Die vollständige und explizite Spezifikation der Abhängigkeiten wird gleichermaßen in Produktion und Entwicklung angewandt.

Zwölf-Faktor-Apps verlassen sich somit nicht auf die implizite Existenz irgendwelcher Systemwerkzeuge, wie z. B. Shell-Aufrufe von `ImageMagick` oder `curl`. Es gibt nämlich keine Garantie, dass diese auf allen Systemen in einer kompatiblen Version vorhanden sind. Wenn die App per Shell auf ein Systemwerkzeug zugreift, muss die App das Werkzeug mitliefern (oder lauffähig installieren).

Die Konfiguration sollte in Umgebungsvariablen abgelegt werden, um eine strikte Trennung der Konfiguration vom Code zu erzielen. Umgebungsvariablen von Deploy zu Deploy zu ändern ist einfach.

Im Gegensatz zu Konfigurationsdateien ist es unwahrscheinlich, dass Umgebungsvariablen versehentlich ins Code Repository eingecheckt werden. Ferner sind Umgebungsvariablen, anders als spezielle Konfigurationsdateien oder andere Konfigurationsmechanismen wie etwa Java Properties, sprach- und betriebssystemunabhängig.

Die Konfiguration einer Anwendung umfasst alles, was sich zwischen den Deploys ändert (Staging, Produktion, Entwicklungsumgebungen usw.). Dies umfasst u. a.

- Resource-Handles für Datenbanken und andere unterstützende Dienste
- Credentials für externe Dienste wie Amazon S3 oder Twitter
- Direkt vom Deploy abhängige Werte, wie z. B. der kanonische Hostname für den Deploy

Die Konfiguration sollte also niemals in Form von Konstanten im Code gespeichert werden, da sich die Konfiguration deutlich von Deploy zu Deploy ändert, ganz im Gegensatz zu Code. Die Definition von Konfiguration umfasst allerdings nicht die interne Anwendungskonfiguration, wie beispielsweise `config/routes.rb` in Rails. Diese Art von Konfiguration ändert sich nämlich nicht von Deploy zu Deploy und gehört daher zum Code.

8.5.3 Unterstützende Services und Port Binding

Die Zwölf-Faktor-Methodik betrachtet in den Faktoren 4 und 7 die Nutzung unterstützender Dienste und deren Bereitstellung. Ein unterstützender Service ist jeder Service, den die Anwendung über das Netzwerk im Rahmen ihrer normalen Arbeit konsumiert (z. B. MySQL, CouchDB, RabbitMQ usw.). Unterstützende Services können selbst verwaltet (z. B. selbst gehostete MySQL-Datenbank) oder auch von Dritten verwaltete Dienste sein (z. B. DB-Dienste wie AWS RDS oder auch über eine API zugängliche Dienste wie Twitter, Google Maps etc.) und viele mehr.

Der Code einer Zwölf-Faktor-App macht keinen Unterschied zwischen lokalen Services und solchen von Dritten. Für die App sind beides unterstützende Services, die über eine in der Konfiguration gespeicherte URL oder andere Lokatoren/Credentials zugreifbar sind. Jeder einzelne Service ist somit als unterstützende, aber austauschbare Ressource zu behandeln.

Ressourcen können beliebig an Deploys an- und abgehängt werden. Wenn zum Beispiel die Datenbank einer Anwendung aufgrund von Hardwareproblemen ausfällt, könnte der App-Administrator eine neue Datenbank aus einem Backup aufsetzen. Die aktuelle Produktionsdatenbank könnte abgehängt und die neue Datenbank angehängt werden – ganz ohne Codeänderung.

Deswegen sollen Services durch Port Binding exportiert werden. Webbasierte Anwendungen exportieren ihre Dienstleistung bspw. über das Protokoll HTTP, indem sich ein Anwendungsprozess an einen Port bindet und auf diesem auf HTTP-Requests wartet. Üblicherweise wird dies mittels Abhängigkeitsdeklaration implementiert. Zu der App fügt man eine Webserver-Bibliothek hinzu (z. B. Tornado für Python, Thin für Ruby oder Jetty für Java und andere JVM-basierenden Sprachen).

Dies findet vollständig im User Space statt, also im Code der Anwendung. Der Vertrag mit der Laufzeitumgebung ist also nur die Bindung an einen Port, um Requests zu bedienen. In einer lokalen Entwicklungsumgebung kann ein Entwickler so über Service-URLs wie beispielsweise `http://localhost:5000/` auf den Dienst der App zugreifen. Beim Deployment sorgt eine Routing-Schicht dafür, dass Requests von einem öffentlich sichtbaren Hostnamen zu den an die Ports gebundenen Prozessen kommen (gegebenenfalls inklusive TLS Termination). Für andere Protokolle wird analog verfahren.

Durch Port Binding kann so eine Anwendung ein unterstützender Service für eine andere App werden, indem die URL der unterstützenden App der konsumierenden App als Resource-Handle zur Verfügung gestellt wird.

8.5.4 Build-, Release- und Run-Phase

Die Zwölf-Faktor-Methodik betrachtet im Faktor 5 unterschiedliche Phasen einer Anwendung. Die Zwölf-Faktor-Methodik trennt dabei strikt zwischen Build-, Release- und Run-Phase. Es ist nicht möglich, Code-Änderungen zur Laufzeit zu machen, weil es keinen Weg gibt, diese Änderungen zurück in die Build-Phase zu schicken.

Eine Codebase wird durch drei Phasen (siehe Bild 8.9) in einen Deploy transformiert:

- Die Build-Phase ist eine Transformation, die ein Code-Repository in ein ausführbares Code-Bündel übersetzt, das man auch Build nennt. Ausgehend von einer Code-Version eines Commits werden Binaries und Assets erzeugt. Arbeitet man mit standardisierten Deployment Units (Containern), ist das Resultat der Build-Phase ein Container-Image.
- Die Release-Phase übernimmt den Build von der Build-Phase und kombiniert ihn mit der zum Deploy passenden Konfiguration. Das so erzeugte Release enthält sowohl den Build als auch die Konfiguration und kann in einer Ausführungsumgebung ausgeführt werden.
- Die Run-Phase führt anschließend ein Release der Anwendung in einer Ausführungsumgebung aus. Bei Cloud-nativen Anwendungen setzt man hier zunehmend auf Container-Plattformen als Ausführungsumgebungen. Derartige Plattformen werden wir noch genauer in Kapitel 9 behandeln.

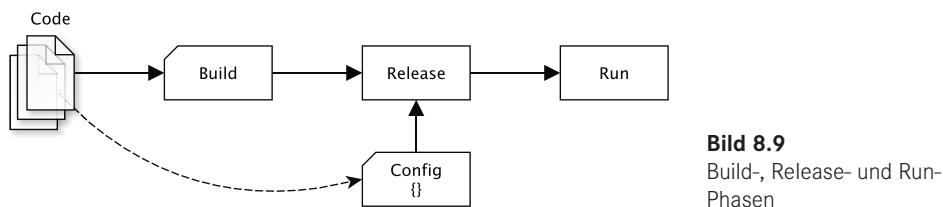


Bild 8.9
Build-, Release- und Run-
Phasen

Jedes Release sollte eine eindeutige Release-ID haben, wie zum Beispiel einen Zeitstempel des Releases (2011-04-06-20:32:17) oder eine laufende Nummer (v100). Releases werden nie gelöscht, und ein Release kann nicht verändert werden, wenn es einmal angelegt ist. Jede Änderung erzeugt einen neuen Release. Mit rollback-Kommandos eines Release-Management-Werkzeugs kann dann einfach und schnell auf ein früheres Release zurückgesetzt werden.

8.5.5 Horizontale Skalierung über Prozesse

Die Zwölf-Faktor-Methodik betrachtet in den Faktoren 6, 8 und 9 die horizontale Skalierung über Prozesse. Anwendungen werden dabei in Form eines oder mehrerer Prozesse ausgeführt. Muss die App mehr Last verarbeiten, werden einfach mehr Prozesse gestartet (horizontale Skalierung). Auf schwer zu beherrschende In-App-Parallelisierung (etwa mittels Threads) sollte verzichtet werden.

Zwölf-Faktor-Apps sind somit vom Grunddesign zustandslos und shared nothing zu konzipieren. Alle Daten werden in unterstützenden Diensten gespeichert, häufig in einer Datenbank. Der RAM oder das Dateisystem des Prozesses kann zwar als kurzfristiger Cache für die Dauer einer Transaktion verwendet werden, sollte aber als konzeptionell flüchtig angesehen werden.

Zum Beispiel kann ein Prozess eine Datei herunterladen, sie verarbeiten und die Ergebnisse in einer Datenbank speichern. Man sollte beim Applikationsdesign aber nie davon ausgehen, dass irgendetwas aus dem RAM oder im Dateisystem Zwischengespeichertes für einen künftigen Request oder Job verfügbar sein wird. Diese auf den ersten Anblick extrem einschränkende Randbedingung harmoniert aber sehr gut mit den Isolationsmechanismen der Betriebssystemvirtualisierung (siehe Abschnitt 8.3) und der Container-Philosophie im Allgemeinen.

Manchmal verletzten gebräuchliche Techniken unerwartet die zwölf Faktoren. So verlassen sich z. B. manche Websysteme auf sogenannte „Sticky Sessions“, d. h., es wird erwartet, dass alle Requests desselben Benutzers zum selben Prozess geschickt werden. Bei horizontaler Skalierung ist das meist nicht der Fall bzw. kann extrem umständlich im Betrieb bei Skalierungen werden. Sticky Sessions sind somit eine Verletzung der zwölf Faktoren. Solches Cachen kann mit prozessexternen Caching-Lösungen wie beispielsweise Memcached oder Redis gelöst werden. Man nimmt dann allerdings eine erhöhte Latenz bei der Request-Bearbeitung in Kauf (da ein weiterer Netzwerkcall erforderlich wird), gewinnt aber architekturell die Fähigkeit einer einfachen horizontalen Skalierbarkeit. Bei Cloud-nativen Systemen wird zumeist der horizontalen Skalierbarkeit ein höherer Wert beigemessen.

Die Prozesse orientieren sich konzeptionell am Unix-Prozess-Modell für Service Daemons. Mit diesem Modell können Entwickler ihre Anwendung für die Bearbeitung verschiedenster Aufgaben konzipieren, indem sie jeder Aufgabe einen Prozesstyp zuweisen. Zum Beispiel können HTTP-Requests durch einen Webprozess bedient werden und lange laufende Hintergrundarbeiten durch einen Worker-Prozess. Diese horizontal teilbare Art und Weise der Prozesse hat zur Folge, dass weitere Nebenläufigkeit einfach und zuverlässig durch Starten neuer Prozesse hinzugefügt werden kann (siehe Bild 8.10).

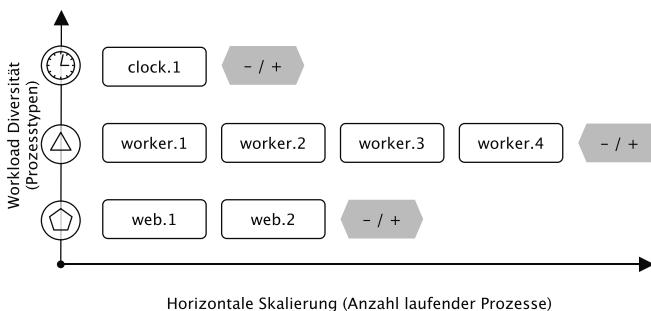


Bild 8.10
Mit dem Prozess skalieren

Allerdings sollten die Prozesse einer Zwölf-Faktor-App nie als Daemons laufen oder PID-Dateien schreiben. Stattdessen sollen sie sich auf den Prozessmanager des Betriebssystems verlassen (wie etwa systemd, den verteilten Prozessmanager einer Cloud-Plattform oder Kubernetes), um Output-Streams zu verwalten oder auf abgestürzte Prozesse zu reagieren und mit von Benutzern angestoßenen Restarts und Shutdowns umzugehen.

Dieser Faktor verbietet übrigens nicht, dass Prozesse ein internes Multithreading oder Multiplexing realisieren, z. B. mittels Threads oder mit dem Async-/Event-Modell von Werkzeugen wie EventMachine oder Node.js.

Die Prozesse einer Zwölf-Faktor-App sollten als „Wegwerf“-Komponente konzipiert werden. Dabei braucht ein Prozess idealerweise nur wenige Sekunden vom Startkommando, bis der Prozess läuft und Requests oder Jobs entgegennehmen kann. Kurze Start-up-Zeiten geben dem Release-Prozess und der Skalierung mehr Agilität und optimieren die Robustheit, weil ein Prozessmanager bei Bedarf einfacher Prozesse auf neue physikalische Maschinen verschieben kann.

Prozesse sollten ferner auf die SIGTERM-Events des Betriebssystems reagieren und sich sicher herunterfahren. Für einen Webprozess kann so ein sicheres Herunterfahren erreicht werden, indem er aufhört, an seinem Service-Port zu lauschen (und damit alle neuen Requests ablehnt), aber noch die laufenden Requests zu Ende bearbeitet und sich dann ebenfalls beendet. Für einen Job-Prozess wird ein problemloser Stopp erreicht, indem er seinen laufenden Job an die Workqueue zurückgibt.

Prozesse sollten zwar nicht mit SIGKILL beendet werden, aber die zugrunde liegende Hardware kann versagen, daher sollten Prozesse auch robust gegen plötzlichen Tod (SIGKILL) sein. Auch wenn dies viel seltener ist als ein reguläres Herunterfahren mit SIGTERM, so kommt es dennoch vor.

Schnelle Starts und problemlose Stopps optimieren die Robustheit. Dieser Faktor erleichtert schnelles elastisches Skalieren, schnelles Deployment von Code oder Konfigurationsänderungen und macht Produktionsdeployments insgesamt robuster.

8.5.6 Umgebungen, Logs und Betrieb

Die Zwölf-Faktor-Methodik betrachtet in den Faktoren 10 bis 12 Umgebungen, Logs und den Betrieb. Ziel ist die Vermeidung verschiedener unterstützender Services in Entwicklung und Produktion. Vielmehr sollten Entwicklungs-, Staging- und Produktionsumgebungen so identisch wie möglich gehalten werden, um folgende Lücken zu schließen (siehe auch Kapitel 3):

- **Zeit:** Ein Entwickler arbeitet an Code, der Tage, Wochen oder sogar Monate braucht, um in Produktion zu gehen.
- **Personal:** Entwickler schreiben Code, Operatoren deployen ihn.
- **Werkzeug:** Entwickler nutzen vielleicht einen Stack wie Nginx, SQLite und OS X – die Produktion nutzt Apache, MySQL und Linux.

Logs werden nach der 12-Faktoren-Methodik als einfacher Strom von aggregierten, nach Zeit sortierten Ereignissen gesehen und werden aus den Output-Streams aller laufenden Prozesse und unterstützenden Services konsolidiert. Logs in ihrer rohen Form sind üblicherweise ein Textformat mit einem Ereignis pro Zeile.

Eine Zwölf-Faktor-App kümmert sich nie um das Routing oder die Speicherung ihres Output-Streams, sondern schreibt den Stream von Ereignissen ungepuffert auf stdout. Bei einem lokalen Deployment siehtet ein Entwickler diesen Stream im Vordergrund seines Terminals, um das Verhalten der App zu beobachten.

Auf Staging- oder Produktionsumgebungen werden die Streams aller Prozesse von der Laufzeitumgebung erfasst, mit allen anderen Streams der Anwendung zusammengefasst und zu einem oder mehreren Zielen (z. B. mittels Log-Konsolidierungslösungen wie Fluentd) geleitet, um dort analysiert oder einfach nur archiviert zu werden. Diese Archivierungsziele (z. B. Elasticsearch) sind für die Anwendung weder sichtbar noch konfigurierbar – sie werden vollständig von der Laufzeitumgebung aus verwaltet. Wie eine solche Log-Konsolidierung zur Optimierung der Beobachtbarkeit von Cloud-native Systemen bewerkstelligt werden kann, wird noch in Kapitel 13 behandelt werden.

Letztlich sollten auch administrative Prozesse in einer Umgebung laufen, die identisch ist zu der Umgebung der üblichen lang laufenden Prozesse. Dies bedeutet, dass auch Admin-Prozesse gegen ein Release laufen und somit dieselbe Codebase und Konfiguration nutzen wie auch das Release. Administrationscode (wie beispielsweise Datenbankmigrationen, Konsole starten oder einmalig auszuführende Skripte wie `fix_bad_records.py`) muss also mit dem Anwendungscode ausgeliefert werden, um Synchronisationsprobleme zu vermeiden. Der Administrationscode sollte dieselbe Dependency-Management-Lösung verwenden wie auch die Anwendung (siehe Abschnitt 8.5.2).

■ 8.6 Zusammenfassung

Der Trend zur Containerisierung und damit zu standardisierten Deployment Units hat unter anderem mit den eher negativen PaaS-Erfahrungen und mangelnder Standardisierung von Laufzeitumgebungen für Cloud-native Anwendungen der Vergangenheit zu tun. Einen Überblick über die PaaS-Philosophie aus der Blütezeit von PaaS gibt beispielsweise (McGrath 2012). Die Einführung von (Carlson 2013) betrachtet das PaaS-Modell mehr aus der Sicht eines Entwicklers und schlüsselt die Arten von Diensten auf, die Google App Engine, Windows Azure, Heroku, Cloud Foundry und weitere Provider liefern. Es lohnt sich sehr, diese Quellen vor allem historisch und rückblickend aus dem Blickwinkel des heutigen Technologiestands zu lesen. Eine eher kritische Betrachtung des Portabilitätsproblems bei PaaS liefert (Kolb 2018).

Insofern schiebt sich mit Container as a Service (CaaS) ein Service-Modell zwischen IaaS (was häufig als zu wenig abstrakt empfunden wird) und PaaS (das mit Standardisierungsproblemen zu kämpfen hat). So wie IaaS auf der Virtualisierung von Maschinen beruht, beruht CaaS eine Ebene höher auf der Betriebssystemvirtualisierung zur Isolation von Prozessen innerhalb eines Hosts. Dies erfolgt meist mittels der Methoden des Linux-Kernels (siehe Abschnitt 8.3). Dadurch können sich mehrere Anwendungen ein und denselben Host teilen, und die Packungsdichte erhöht sich in Infrastrukturen. Die Linux-Kernelentwicklung und Systemprogrammierung wird sehr umfangreich in (Love 2007) und (Love 2010) behandelt. Ergänzend und vertiefend betrachtet etwa (Jain 2020) die Isolation von Prozessen in Linux und greift damit insbesondere die technische Grundlage von Containern auf.

Der Erfolg der Containerisierung hat auch damit zu tun, dass es mit Docker komfortabel möglich wurde, Anwendungen inklusive aller Abhängigkeiten in Form standardisierter Images bereitzustellen (siehe Abschnitt 8.4). Dies löst viele Abhängigkeitsprobleme, da Container alle Abhängigkeiten beinhalten und so nicht auf Bibliotheken oder Laufzeitumgebungen, die auf einem Host installiert sind, angewiesen sind. Insbesondere zu Docker gibt es daher eine äußerst umfangreiche Auswahl von Literatur. Der Leser wird problemlos viele geeignete und gute Quellen finden, die hier nicht alle aufgeführt werden können. Eine gute Wahl zum Einstieg sind aber oft Werke der Up&Running-Reihe von O'Reilly wie (Kane und Matthias 2018).

Dennoch sind auch die Definition und der Betrieb von Containern kein Selbstläufer. Es haben sich in den letzten Jahren diverse Best Practices entwickelt, wie Container definiert werden sollten, um effizient und friktionsfrei betreibbar zu sein. Hält man sich an diese in Abschnitt 8.5 aufgeführten Regeln, erhält man Anwendungskomponenten, die sehr gut in darauf abgestimmten und im folgenden Kapitel 9 behandelten Betriebsumgebungen ausführbar sind. Dieses als 12 Faktoren bekannt gewordene Regelwerk und sein Zusammenhang mit Containerisierung wurden nur stark zusammenfassend und nicht exakt den 12 Faktoren folgend betrachtet. Dem Leser sei daher das Studium der online verfügbaren und in zahlreiche Sprachen übersetzten Originalquelle (Wiggins 2017) durchaus ans Herz gelegt.

Ergänzende Materialien

Auf der Website zum Buch finden sich zu diesem Kapitel ferner folgende Ergänzungsmaterialien:

- Praktische Übungen (Labs) zum Thema Containerisierung inklusive Detailaspekte wie Container-Images mittels lokaler Builds erstellen, Images innerhalb von Deployment-Pipelines erstellen, Images in Registries bereitstellen sowie Image Shrinking-Techniken, um möglichst kleine Container-Images zu erzeugen
- Links auf die OCI-Standards
- Übersichten inklusive Links zu weiteren Container-Laufzeitumgebungen
- Übersichten inklusive Links zu Container-Image-Build-Tools, Image Registries (managed und self-hostable) und Vulnerability Scannern
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer (Themen: Hintergrund-Plattform as a Service, Standardisierung von Deployment Units, Betriebssystemvirtualisierung, Container, Docker, 12-Faktoren-Methodik, Container Pattern)



<https://bit.ly/3ASEYtu>

9

Container-Plattformen

„How would you design your infrastructure if you couldn't login. Never!“

Kelsey Hightower, Co-Autor von Kubernetes: Up and Running

Cloud-native Systeme werden meist nicht direkt auf der Infrastruktur betrieben, obwohl diese ja – wie Kapitel 7 zeigt – mittels IaC-basierter Provisionierung durchaus komfortabel automatisiert deployt und horizontal skaliert werden kann. Meist nutzt man vielmehr Plattformen, auf denen dann Cloud-native Systeme betrieben werden. Diese Plattformen sind für den dauerhaften Betrieb von Workloads optimiert und adressieren damit zusammenhängende Erfordernisse, die sich primär aus dem Betrieb ergeben. Solche Plattformen können entweder als Managed Service bezogen (z. B. bieten die großen Hyperscaler wie Amazon, Azure oder Google Managed Kubernetes Services an) oder selbst auf Infrastruktur betrieben werden. Hierfür gibt es diverse Plattformen wie beispielsweise Kubernetes, Mesos, Swarm oder Nomad. Welcher Weg auch beschritten wird, dieser zusätzliche Plattform-Layer adressiert ergänzend zu den Infrastruktur-Erfordernissen (siehe Kapitel 7) grundsätzlich weitere wesentliche Probleme beim Betrieb Cloud-nativer Systeme.

1. Wie kann etwa entschieden werden, auf welche Infrastrukturressourcen einer Betriebsplattform welche Anwendungen platziert werden? Was passiert mit den auf Ressourcen laufenden Anwendungen, wenn Ressourcen nicht mehr zur Verfügung stehen, z. B. weil diese im Rahmen einer Wartung heruntergefahren oder aufgrund externer Einflüsse wie einem Stromausfall plötzlich ausfallen? Diese Problemklasse wird als **Scheduling** bezeichnet und in Abschnitt 9.1 behandelt.
2. Wie können verteilte Anwendungen auf solchen Betriebsplattformen dauerhaft, automatisiert (d. h. selbstheilend) und unter Berücksichtigung ihrer „Moving Parts“ betrieben werden? Solche Moving Parts sind z. B. eine sich stetig ändernde Infrastruktur (Hosts), sich ändernde IP-Adressen von Hosts und Containern, sich ändernde Anzahl von Containern und mehr. Das Management dieser „Moving Parts“ werden wir als **Orchestrierung** bezeichnen und in Abschnitt 9.2 behandeln.

Abschnitt 9.3 wird sich auf Kubernetes als Typvertreter für solche Fragen adressierende Plattformen fokussieren. Kubernetes hat sich in diesem Bereich (ähnlich wie das Betriebssystem Linux für den Betrieb von Servern) als ein De-facto-Standard etabliert. Die am Beispiel Kubernetes erläuterten Prinzipien lassen sich grundsätzlich auf andere Plattformen übertragen. Dem Leser sollte allerdings bewusst sein, dass sich diese Plattformen in den Details von Kubernetes substanzial unterscheiden können. Dennoch macht es nach Einschätzung des Autors wenig Sinn, derartige Betriebsplattformen abstrakt (d. h. plattformagnostisch) zu betrachten, da dies dann sehr theoretisch werden würde. Alle plattformagnostischen Überlegungen werden einleitend in Abschnitt 9.1 und Abschnitt 9.2 erläutert.

■ 9.1 Scheduling

Die Grundidee solcher Betriebsplattformen ist die Idee des „Datacenter as a Computer“. Dabei fasst man mehrere Rechner zu einem sogenannten **Cluster** zusammen. Ein Cluster wirkt aus Nutzersicht wie ein einziger großer Computer. Das macht die Interaktion mit einem ansonsten extrem verteilten und dynamischen System erheblich einfacher. Solch ein „Cluster-Computer“ benötigt dann aber (wie ein normaler Rechner auch) ein Betriebssystem, das sich u. a. um die Ressourcenverwaltung kümmert. In Konsequenz müssen also viele Konzepte klassischer Betriebssysteme auf „Cluster-Betriebssysteme“ übertragen werden. Das gilt insbesondere auch für das Scheduling.

Die Grundaufgabe des Schedulings ist es, eine möglichst ressourceneffiziente und aufgabenangemessene Zuteilung von Arbeitslasten (Workloads) an Rechenressourcen, die beispielsweise mittels IaaS (siehe Kapitel 7) bereitgestellt werden können, vorzunehmen.

Unter einem **Workload** (oft auch als Job bezeichnet) versteht man dabei eine Menge von Tasks mit gemeinsamem Ausführungsziel. Da Tasks untereinander Abhängigkeiten haben können, ist die Menge an Tasks in der Regel als gerichteter azyklischer Graph (DAG) darstellbar. So könnte ein Task B beispielsweise nur auf dem Ergebnis von Task A ausführbar sein ($A \rightarrow B$). Dabei werden in solchen Graphen Tasks als Knoten und Ausführungsabhängigkeiten zwischen Tasks als Kanten dargestellt. Ein **Task** ist dabei eine atomare Rechenaufgabe inklusive Ausführungsvorschrift.

Workloads und Tasks können ausführungsrelevante Eigenschaften (**Property**) haben. Ein Workload kann z. B. Abhängigkeiten der Tasks untereinander oder einen erforderlichen Ausführungszeitpunkt haben (z. B. einmal am Ende des Monats). Tasks sind durch Eigenschaften wie ihre Ausführungsduer, Priorität und Ressourcenbedarf gekennzeichnet.

Ein **Scheduler** plant die Ausführung von Tasks auf verfügbaren Ressourcen unter Berücksichtigung der Eigenschaften und zu optimierender Scheduling-Ziele (z. B. Fairness, Durchsatz, Ressourcenauslastung). Ein Scheduler kann präemptiv sein, also Tasks unterbrechen und neu aufsetzen.

Unter **Ressourcen** verstehen wir ein Cluster von Rechnern mit jeweils eigenen CPU-, RAM-, Storage- und Netzwerkressourcen. Ein Rechner stellt seine Ressourcen temporär zur Ausführung eines oder mehrerer Tasks zur Verfügung (Slot). Die parallele Ausführung von Tasks erfolgt isoliert.

Die drei wesentlichen Aufgaben eines Cluster-Schedulers sind:

- Cluster-Awareness
- Workload-Allokation
- Workload-Ausführung

Zum Aufbau einer kontinuierlichen **Cluster-Awareness** sind die aktuell verfügbaren Ressourcen im Cluster zu verfolgen (Knoten inklusive deren verfügbaren CPU-, RAM-, Storage-Ressourcen sowie Netzwerkbandbreite). Dabei ist die Elastizität des Clusters zu berücksichtigen, da auch der Cluster aus „Moving Parts“ besteht, die durch das Hinzufügen und Entfernen von Knoten entstehen. Im Rahmen der **Workload-Allokation** muss zur Ausführung eines Workloads die passende Menge an Ressourcen entweder dauerhaft (Service) oder für einen bestimmten Zeitraum (Job) bestimmt und gebunden werden.

Während der **Workload-Ausführung** müssen Tasks des Workloads voneinander isoliert, zuverlässig ausgeführt und überwacht werden.

9.1.1 Heterogenität von Workloads

Scheduling ist grundsätzlich durch zwei Aspekte gekennzeichnet. Zum einen ist Scheduling eine NP-vollständige Optimierungsaufgabe. Es ist also kein Algorithmus bekannt, der eine optimale Lösung in polynomialer Laufzeit erzeugt. Optimale Algorithmen, die den Lösungsraum komplett durchsuchen, sind vor diesem Hintergrund daher nicht praktikabel, da deren Entscheidungszeit zu lange für große Eingabemengen ($| \text{Workloads} | \cdot | \text{Ressourcen} |$) dauert. Darüber hinaus kann die Anzahl an Workloads in DevOps-Szenarien nicht als statisch angenommen werden, da die Workloads oft einer lastbasierten horizontalen Skalierung unterliegen, sodass selbst bei optimaler Allokation der Re-Organisationsaufwand pro Workload unverhältnismäßig hoch werden kann.

Ferner sind in Clustern betriebene Workloads üblicherweise sehr heterogen. In einer Studie von (Reiss u. a. 2012) wurden in Google-Rechenzentren über mehrere Monate die ausgeführten Workloads in mehreren Plattformen mit mehr als 10 000 Knoten untersucht. Danach gibt es zwar überwiegend viele kleine, ressourcenarme und kurze Workloads. Aber eben nicht nur! So hatten die meisten Workloads zwar nur einen Task, aber es gibt auch Workloads, die aus 10 000 und mehr Tasks bestanden. Die Ausführungszeit von Workloads bewegte sich dabei in einem Spektrum, das sich vom einstelligen Sekundenbereich am unteren Ende bis hin zu mehr als 40 Tagen (längere Workloads wurden in dieser Studie nicht betrachtet) am oberen Ende bewegte. Und im Spektrum dazwischen sind alle Mischformen von Taskanzahl und Dauer exponentiell abnehmend vertreten. Ähnlich sieht es hinsichtlich CPU- und Hauptspeicherressourcen der untersuchten Workloads aus. Charakteristische Unterschiede bei Workloads sind somit nach (Reiss u. a. 2012) u. a.:

- **Dauer:** Sekunden, Minuten, Stunden, Tage, unendlich
- **Terminierung:** Sofort, später, zu einem definierten Zeitpunkt
- **Zweck:** Datenverarbeitung, Request-Handling
- **Verbrauch:** CPU-, RAM-, (S/H)DD-, netzwerkdominant
- **Zustand:** Zustandsbehaftet und zustandslos

Nach (Reiss u. a. 2012) sind mindestens zwei Haupt-Workload-Kategorien zu unterscheiden.

- **Batch-Jobs:** Bei diesen Workloads liegt die Ausführungszeit üblicherweise im Minuten- bis Stundenbereich. Diese Workloads sind zumeist zustandsbehaftet und haben eine eher niedrige Priorität und sind gut unterbrechbar. Es ist nicht ungewöhnlich, dass diese Workloads häufig bis zu einem bestimmten Zeitpunkt abgeschlossen sein müssen.
- **Service-Jobs:** Diese Art von Workloads sollen auf unbestimmte Zeit unterbrechungsfrei laufen, haben üblicherweise eine hohe Priorität und können daher nicht so ohne Weiteres unterbrochen werden. Sie sind teilweise zustandslos.

Üblicherweise berücksichtigen diese beiden Workload-Arten daher auch alle einschlägigen Plattformen. Vor diesem komplexen und heterogenen Hintergrund nutzen allerdings unterschiedliche Plattformen unterschiedlichste Scheduling-Algorithmen mit verschiedenen Scheduling-Architekturen.

9.1.2 Scheduling-Algorithmen

Unter der Berücksichtigung von Awareness-Daten wie beispielsweise:

- **Ressource:** Welche Ressourcen stehen bereit, welchen Bedarf hat der Task?
- **Lokalität:** Wo sind die Daten, die ein Task benötigt?
- **Randbedingungen:** Welche Ausführungszeiten müssen garantiert werden?
- **Kosten:** Welche Betriebskosten sind einzuhalten?
- **Priorität:** Welche Priorität hat der Task?
- **Fehlertoleranz:** Wahrscheinlichkeit eines Ausfalls? Sind beispielsweise Racks für Wartung markiert?
- **Erfahrungswerte:** Wie hat sich ein Task in der Vergangenheit verhalten?

leitet der Cluster-Scheduler mittels Scheduling-Algorithmen Placement Decisions ab. Diese beinhalten Slot-Reservierungen und Slot-Stornierungen (im Fehlerfall, Optimierungsfall oder bei Constraint-Verletzungen) von Ressourcen für Workloads auf der verfügbaren und einsatzbereiten Infrastruktur. Scheduling-Algorithmen können dabei unterschiedliche Scheduling-Ziele verfolgen:

- **Fairness:** Kein Task soll unverhältnismäßig lange warten müssen, während ein anderer bevorzugt wird.
- **Maximaler Durchsatz:** Bearbeitung so vieler Tasks pro Zeiteinheit wie möglich.
- **Minimale Wartezeit:** Möglichst geringe Zeit von der Übermittlung bis zum Ausführungssstart eines Tasks.
- **Ressourcenauslastung:** Möglichst hohe Auslastung der verfügbaren Ressourcen.
- **Zuverlässigkeit:** Ein Task wird garantiert ausgeführt.
- **Minimierung der End-to-End-Ausführungszeit** z. B. durch Datenlokalität.

9.1.2.1 Einfache Scheduling-Algorithmen

Einfache Scheduling-Algorithmen optimieren das Scheduling von Tasks oft in genau einer Dimension (z. B. CPU-Auslastung) bzw. wenigen Dimensionen (CPU + RAM). Populäre Algorithmen wie Binpack folgen dabei oft einer sehr einfachen Fit-First-Strategie, die eine hohe Auslastung von Ressourcen realisiert. Der ebenso einfache Spread-Algorithmus verfolgt hingegen eine Round-Robin-Strategie, die eher in einer gleichmäßigen Auslastung von Ressourcen resultiert (siehe Bild 9.1). Systeme wie Docker Swarm nutzen derartige Algorithmen, die – obwohl sehr einfach – dennoch für viele Anwendungsfälle absolut ausreichend sind.



Bild 9.1
Einfache Scheduling-Algorithmen

9.1.2.2 Multidimensionale Scheduling-Algorithmen

Multidimensionale Scheduling-Algorithmen wie etwa der Dominant Resource Fairness-Algorithmus (DRF, siehe Bild 9.2) optimieren das Scheduling von Tasks dabei nach mehreren Kriterien. DRF strebt beispielsweise eine möglichst faire Aufteilung der Ressourcen über verschiedene Nutzer (Kunden, Projekte etc.) an (Ghodsi u. a. 2011). Dabei wird davon ausgegangen, dass jeder Nutzer üblicherweise eine dominante Ressource hat, die sein System primär benötigt und daher besonders intensiv nutzt. Es gibt sowohl Processing-, Memory-, Storage- als auch netzwerklastige Anwendungen. Allerdings sind Anwendungen, die zeitgleich sowohl Processing-, Memory-, Storage- oder netzwerklastig sind, nicht die Regel. Es macht also oft keinen Sinn, Storage genauso zu gewichten wie etwa Memory. Vielmehr kann man sich an der dominanten Ressource orientieren, die durch Beobachtung ermittelt werden kann. Die Ressourcenallokation erfolgt dann derart, dass jeder Nutzer (von N Nutzern) mindestens $1/N$ aller Ressourcen seiner dominanten Ressourcen bekommt. Der DRF-Scheduling-Algorithmus ist also darauf ausgelegt, die dominanten Ressourcen pro Nutzer zu maximieren. Die Fairness kann zusätzlich auch noch gewichtet werden. Wenn die Anwendungen eines Teams beispielsweise doppelt so wichtig wären, würde dieses Team auch doppelt so viele Ressourcen bekommen. Insbesondere das Mesos-System nutzt diese Art von Scheduling-Algorithmen (Hindman u. a. 2011).

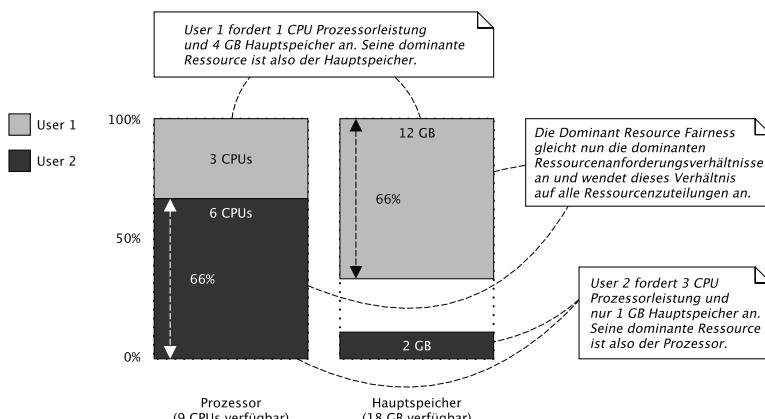


Bild 9.2 Multidimensionale Scheduling-Algorithmen (hier: Dominant Resource Fairness, DRF)

9.1.2.3 Kapazitätsbasierte Scheduling-Algorithmen

Kapazitätsbasierte Scheduling-Algorithmen werden meist für jobbasierte Systeme verwendet, z. B. um intensive Rechenprozesse mit großen Datenmengen (Big Data, Petabyte-Bereich) auf Computerclustern durchzuführen. Üblicherweise werden dabei Jobqueues definiert und zu jeder Queue eine Kapazitätszusage in Ressourcenanteilen vom Cluster definiert. Als fair wird dabei ein Ressourcenanteil verstanden, der höher als eine definierte minimale Kapazitätszusage ist (siehe Bild 9.3). Derartige Scheduling-Algorithmen sorgen dafür, dass diese Minimal-Randbedingung stets sichergestellt ist. Damit ein Cluster dafür nicht statisch partitioniert werden muss, ist eine Überbuchung von Ressourcen erlaubt. Wird durch eine Überbuchung jedoch eine Kapazitätszusage gefährdet oder überschritten, werden die überbuchten Ressourcen entzogen. Hierfür ist somit ein präemptiver Scheduler notwendig,

der notfalls Jobs unterbrechen und später fortsetzen oder erneut starten kann. Der YARN-Scheduler (Vavilapalli u. a. 2013) des Hadoop-Systems (White 2015) arbeitet beispielsweise nach diesem Prinzip.

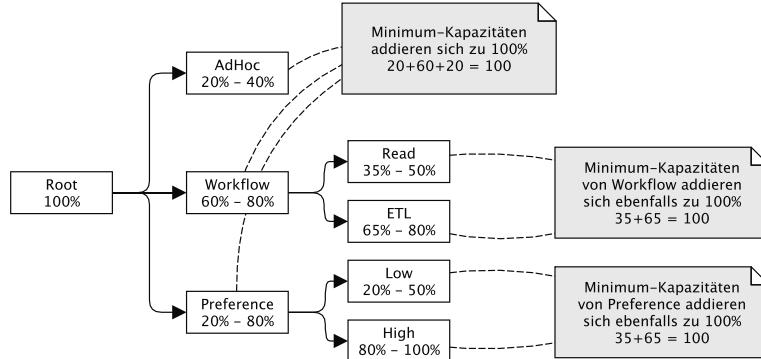


Bild 9.3 Kapazitätsbasierte Scheduling-Algorithmen

9.1.3 Scheduling-Architekturen

Welche Scheduling-Ziele nun im Einzelnen zu verfolgen sind, ist natürlich von den zu betrachtenden Anwendungsfällen abhängig und kann nicht pauschal beantwortet werden. Mittels unterschiedlicher Scheduling Architekturen lassen sich auch unterschiedliche Scheduling-Algorithmen für unterschiedliche Anwendungsfälle realisieren.

Grundsätzlich folgen alle Scheduling-Architekturen dem in Bild 9.4 gezeigten Prinzip. Jede Scheduler-Architektur muss Anforderungen hinsichtlich Performance (z. B. geringe Queueing-Time, Decision-Time, Ausführungslatenz), Hochverfügbarkeit und Fehlertoleranz sowie Skalierbarkeit bezüglich Anzahl an Workloads und verfügbaren Ressourcen erfüllen.

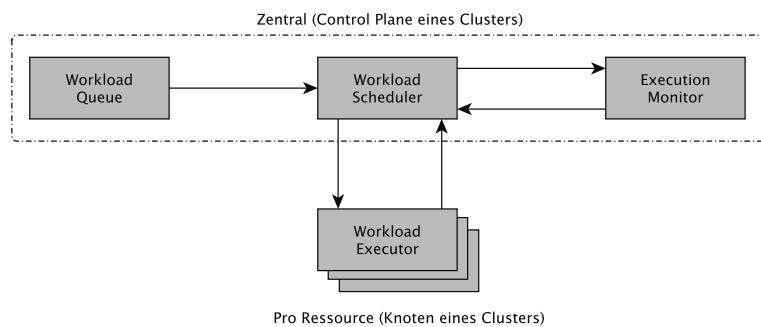


Bild 9.4 Konzeptionelle Scheduling-Architektur

Zentral für den Cluster empfängt eine **Workload-Queue** eingehende Workloads, die zur Ausführung gebracht werden sollen. Ein **Workload-Scheduler** plant die Workloads ein und steuert die Ausführung gegebenenfalls voneinander abhängiger Tasks innerhalb eines Workloads. Der **Execution-Monitor** überwacht sowohl die Taskausführung als auch die zur Verfügung stehenden Ressourcen des Clusters.

Auf jedem Host eines Clusters läuft ferner ein **Executor** im Sinne einer Runtime Environment (siehe auch Abschnitt 8.3.5), der Tasks ausführen kann und dem Workload Scheduler und Execution-Monitor Awareness-Informationen zur Verfügung stellt.

Die Ansätze unterscheiden sich allerdings in der Art und Weise, wie der Workload Scheduler zu seinen Allokationsentscheidungen kommt. Hier können mindestens die folgenden Scheduler-Architekturansätze unterschieden werden (siehe Bild 9.5).

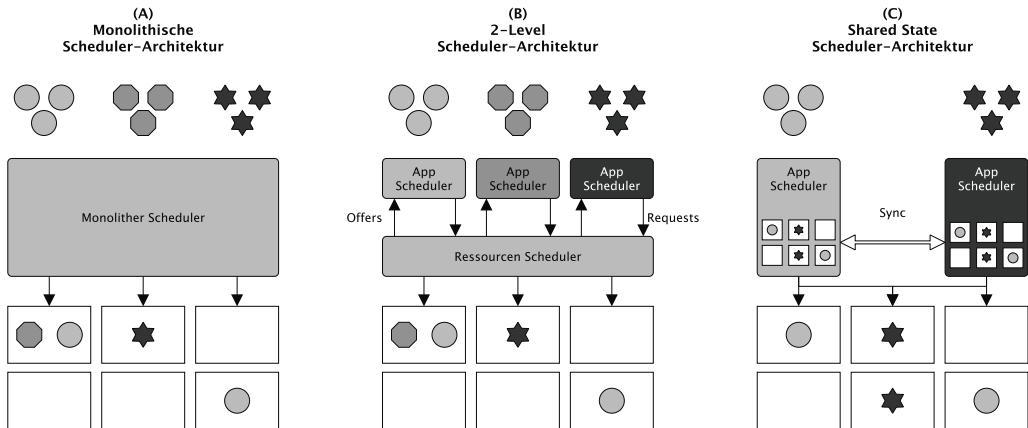


Bild 9.5 Verschiedene Scheduler-Architekturen

9.1.3.1 Monolithischer Scheduler

Monolithische Scheduler bestehen aus einem einzigen Scheduling-Agenten, der alle Anfragen bearbeitet, und werden daher häufig im High-Performance-Computing eingesetzt, da hier Workloads oft homogenen Charakter haben (also processing-intensiv sind). Ein monolithischer Scheduler wendet in der Regel eine einzige Scheduling-Algorithmen-Implementierung für alle eingehenden Workloads an, sodass es schwierig ist, je nach Workload-Typ unterschiedliche Scheduling-Logiken zu verwenden.

Monolithische Scheduler werden jedoch auch (u. a. durch Google) erfolgreich zum Scheduling heterogener Workloads verwendet. Das Google Borg-System gilt beispielsweise als konzeptioneller Ahne (Verma u. a. 2015) des mittlerweile populären Kubernetes-Systems. Auch Kubernetes, der YARN Scheduler von Hadoop oder das Swarm-System von Docker arbeiten nach dem in Bild 9.5 A gezeigten Prinzip.

Die Vorteile monolithischer Scheduler liegen in der konzeptionell einfach umsetzbaren Möglichkeit globaler Optimierungsstrategien. Damit einher gehen jedoch Probleme beim Scheduling heterogener Workloads, die ein potenzielles Skalierbarkeits-Bottleneck bilden können. Dieses kann man durch komplexe und umfangreiche Implementierung von Schedulern kompensieren, oder man toleriert einfach die geringere Effizienz des homogenen Schedulings.

9.1.3.2 2-Level-Scheduler

Ein zweistufiger Scheduler besteht hingegen aus einem zentralen Ressourcen-Scheduler und weiteren Workload-Schedulern für spezifische Workload-Arten (siehe Bild 9.5 B). Dabei wird die Ressourcenzuweisung für jeden Workload-Scheduler dynamisch angepasst, indem ein zentraler Ressourcen-Scheduler entscheidet, wie viele Ressourcen jeder Workload-Scheduler erhält.

Dieser Ansatz wird etwa in Mesos verwendet (Ghodsi u. a. 2011), (Hindman u. a. 2011). Da eine bestimmte Ressource vom zentralen Scheduler jeweils nur einem Workload-Scheduler angeboten wird, werden cluster-weite Ressourcenüberbuchungen konzeptionell vermieden. Die Ressourcenzuteilung an die Workload-Scheduler erfolgt dabei oft nach dem Prinzip der dominanten Ressourcenfairness (siehe Abschnitt 9.1.2.2). Da jeweils nur ein Workload-Scheduler die Eignung einer Ressource prüft, wird die Gleichzeitigkeitssteuerung als pessimistisch bezeichnet. Diese pessimistische Strategie ist weniger fehleranfällig, aber langsamer als eine optimistische Gleichzeitigkeitssteuerung, die eine Ressource vielen Sub-Schedulern gleichzeitig anbietet.

Der zentrale Ressourcen-Scheduler kennt alle verfügbaren Ressourcen und darf diese allozieren. Er nimmt Ressourcenanfragen (Requests) von den Workload-Schedulern entgegen und unterbreitet entsprechend einer Scheduling Policy Ressourcenangebote (Offers). Der Workload-Scheduler nimmt Workloads entgegen, „übersetzt“ diese in Ressourcenanfragen und wählt workload-spezifisch die passenden Ressourcenangebote aus. Offers sind eine zeitlich beschränkte Allokation von Ressourcen, die explizit angenommen werden muss. Im Sinne der Fairness kann ein prozentualer Anteil der Ressourcen pro Workload-Scheduler durch den zentralen Ressourcen-Scheduler garantiert werden.

Die Vorteile solcher 2-Level-Scheduler liegen in der mit dem Mesos-System nachgewiesenen Skalierbarkeit auf Tausenden von Knoten (z. B. Twitter, Airbnb, Apple Siri) und einer flexiblen Architektur für heterogene Scheduling-Logiken. Anders als bei monolithischen Schedulern sind Workload-Scheduler übergreifende Logiken allerdings nur schwer zu realisieren.

9.1.3.3 Shared-State Scheduler

Bei dem in Bild 9.5 C gezeigten Shared State-Ansatz gibt es ausschließlich workload-spezifische Scheduler.

Die Workload-Scheduler synchronisieren kontinuierlich den aktuellen Zustand des Clusters (Workload-Allokationen und verfügbare Ressourcen). Jeder Workload-Scheduler entscheidet über die Platzierung von Tasks auf Basis des ihm bekannten aktuellen Cluster-Zustands. Hier wird also eine optimistische Gleichzeitigkeitssteuerung genutzt. Ressourcenüberbuchungen werden somit zugelassen, ein zentraler Koordinierungsdienst erkennt lediglich Scheduling-Konflikte und löst diese auf, indem er Zustandsänderungen nur für einen der beteiligten Workload-Scheduler erlaubt und anderen Workload-Schedulern einen Fehler meldet.

Die Vorteile solcher Shared-State Scheduler liegen also in dem tendenziell geringeren Kommunikations-Overhead. Nachteilig ist allerdings, dass das komplette Scheduling pro Workload-Scheduler entwickelt werden muss. Es sind ferner keine globalen Scheduling-Ziele (z. B. Fairness) möglich. Ferner ist die Skalierbarkeit für große Cluster noch unklar, da noch nicht umfangreich in der Praxis erprobt und insbesondere die Auswirkungen bei einer hohen Anzahl an Konflikten noch ungeklärt sind.

Das bekannteste dieser Systeme dürfte das Omega-System von Google sein (Schwarzkopf u. a. 2013). Omega gewährt jedem Scheduler vollen Zugriff auf den gesamten Cluster, sodass alle Scheduler in einem freien Wettbewerb miteinander stehen. Es gibt keine zentrale Ressourcenzuteilung, da alle Entscheidungen zur Ressourcenzuteilung in den Schedulern getroffen werden. Es gibt auch keine zentrale Policy-Enforcement-Engine. Durch die Unterstützung unabhängiger Scheduler-Implementierungen und die Offenlegung des gesamten Zuweisungsstatus der Scheduler kann Omega auf viele Scheduler skalieren und mit verschiedenen Arbeitslasten mit eigenen Scheduling-Richtlinien arbeiten.

■ 9.2 Orchestrierung

Scheduling befasst sich also damit, Betriebskomponenten (also Workloads) an Ressourcen einer bestehenden Infrastruktur für deren Ausführung zuzuweisen - also der Workload-Allokation. Dabei können die unterschiedlichen Betriebskomponenten als unabhängig voneinander betrachtet werden. Diese Unabhängigkeit ist im Betrieb bei verteilten Systemen allerdings naturgemäß nicht gegeben. Man wird z. B. Frontend-, Backend-Service und Datenbank-Services einer Webapplikation schwerlich im Betrieb als voneinander isoliert zu betrachtende Komponenten verstehen können. Der Ausfall eines Backend-Service mag gegebenenfalls keine Auswirkungen auf einen dahinter liegenden Datenbank-Service haben, sehr wohl aber auf den davor liegenden Frontend-Service. Nur für das Scheduling können alle drei Komponenten einheitlich betrachtet werden, da für alle Komponenten schlicht und ergreifend erforderliche Ressourcen zu allokieren sind.

Aus einer Betriebsperspektive sieht das allerdings anders aus. Daher versteht man unter Orchestrierung vor allem die automatisierte Konfiguration, Verwaltung und Koordinierung von Anwendungen und Services, die üblicherweise in mehrere Betriebskomponenten aufgeteilt sind und auf mehreren Knoten einer geclusterten Plattform (wie z. B. Kubernetes) laufen. Hier sind gegenüber dem Scheduling weitere Abstraktionen zur Ausführung von Anwendungen mit ihren Services in einem (großen) Cluster erforderlich. Orchestrierung ist ferner keine statische, einmalige (oder seltene) Aktivität wie die Provisionierung oder das Scheduling, sondern eine dynamische, kontinuierliche Aktivität mit dem Ziel, alle Standard-Betriebsprozeduren einer Anwendung im Betrieb zu automatisieren. Es geht also bei der Orchestrierung um die Sicherstellung einer dauerhaften und verlässlichen Workload-Ausführung, die durch externe Einflussfaktoren gestört werden kann.

9.2.1 Definition von Betriebszuständen

Orchestrierung beruht dabei auf der Formulierung von gewünschten Betriebszuständen und Abhängigkeiten zwischen den Betriebskomponenten einer Anwendung mittels eines **Blueprints**. Solch ein Blueprint beschreibt somit die Betriebskomponenten (zumeist Container), deren Betriebsanforderungen (z. B. Bedarf an RAM, CPU-Anteilen, Anzahl an Prozessen, die einen Service bereitstellen sollen, usw.) sowie die angebotenen und benötigten Schnittstellen. Der Cluster-Orchestrator übersetzt diese dann in zahlreiche Steuerungsaktivitäten wie z. B.:

- Scheduling von Containern mit applikationsspezifischen Randbedingungen inklusive Re-Scheduling von Containern im Fehlerfall (Auto-Healing) oder zur Ressourcen-Optimierung (siehe Abschnitt 9.1)
- Aufbau von notwendigen Netzwerkverbindungen zwischen Containern
- Bereitstellung von persistenten Speichern für zustandsbehaftete Container
- (Auto-)Skalierung von Containern
- Container-Logistik: Verwaltung, Bereitstellung und Caching von Container-Images
- Package-Management: Verwaltung und Bereitstellung von Applikationen

- Management von Services: Service Discovery, Naming, Load Balancing
- Automatismen für Rollout-Workflows wie z. B. Canary Rollout
- Monitoring und Diagnose von Containern und Services
- und weitere

Damit bilden Cluster-Orchestrator eine zentrale Schnittstelle zwischen Entwicklung und Betrieb und bilden eine wesentliche Grundlage für DevOps (siehe Kapitel 3), da sie durch zahlreiche regelkreisbasierte Automatismen zu den Prinzipien des Flows (siehe Abschnitt 3.1) beitragen.

9.2.2 Regelkreis: Desired versus Current State

In der Robotik und Automation versteht man unter einem Regelkreis eine kontinuierliche Feedback-Schleife, die den Zustand eines Systems regelt. Das klassische Beispiel für einen Regelkreis ist ein Raumthermostat. An einem Thermostaten kann man die gewünschte Raumtemperatur einstellen (*Desired State*). Die tatsächliche Raumtemperatur ist der aktuelle Zustand (*Current State*) und kann vom Thermostaten gemessen werden. Der Thermostat agiert nun mittels der beiden Heizkörperventil-Steuerungsbefehle *öffnen* und *schließen*, um den aktuellen Zustand (Raum-Ist-Temperatur) näher an den gewünschten Zustand (Raum-Wunschtemperatur) zu bringen. Liegt die Wunschtemperatur über der aktuellen Raumtemperatur, wird das Ventil geschlossen, liegt sie unterhalb der aktuellen Raumtemperatur, wird das Ventil entsprechend geöffnet.

Dieses erstaunlich einfache Prinzip kann auch beim Betrieb von Anwendungen innerhalb von Clustern angewendet werden. Regelkreise sind dabei äußerst robust gegen Änderungen des Desired und Current State, da es für Regelkreise unerheblich ist, ob der Soll-Zustand oder der Ist-Zustand geändert wurde. Regelkreise reagieren nur auf die Differenz, egal ob diese durch eine Operator-Entscheidung (Änderung des Desired State) oder einen externen Einfluss wie den Ausfall einer Systemkomponente (Änderung des Current State) entstanden ist. Existiert eine Abweichung, wird ein entsprechender Steuerungsbefehl gegeben mit dem Ziel, die Abweichung zu reduzieren.

Diese Regelkreise werden in Orchestrationsplattformen durch Controller realisiert, die den Zustand des Clusters und seiner Ressourcen beobachten und bei Bedarf Änderungen vornehmen oder anfordern. Jeder Controller versucht dabei den aktuellen Zustand des Clusters und seiner Ressourcen näher an den gewünschten Zustand heranzuführen. Ein Controller verfolgt dabei mindestens einen Ressourcentyp, der mittels eines Blueprints definierbar ist. Innerhalb der Blueprints wird der gewünschte Zustand angegeben. Der Controller für diese Ressource ist dafür verantwortlich, dass sich der aktuelle Zustand diesem gewünschten Zustand annähert.

Ein Job-Controller ist ein gutes Beispiel für einen solchen Controller. Ein Job ist im Allgemeinen eine Ressource zum Anlegen eines einmaligen, meist aufwendigen, d. h. lang laufenden, Berechnungsauftrags (z. B. eine Sentiment-Analyse aller Produktratings des letzten Monats), die im Rahmen eines Containers ausgeführt wird. Wenn der Job-Controller eine neue Aufgabe sieht, wendet er sich an den Scheduler, um einen entsprechenden Container ausführen zu lassen. Der gewünschte Zustand ist, dass dieser Auftrag erfolgreich abgeschlossen ist.

(der Prozess also mit dem Exit-Code 0 terminiert). Sobald die Arbeit für einen Job erledigt ist (der Controller also die Terminierung mit Exit-Code 0 erkennt), aktualisiert der Job-Controller dieses Job-Objekt und markiert es als „Fertig“. Sollte der Job während der Bearbeitung scheitern (z. B. weil Daten zur Berechnung temporär nicht abrufbar sind), endet der Job in einem Zustand ungleich „Fertig“ (also einem Exit-Code ungleich 0). Dann wird der Controller den Job erneut durch den Scheduler starten lassen, da der Zielzustand „Fertig“ nicht erreicht wurde. Dem Controller ist dabei egal, was die Ursache des Scheiterns war (Netzwerkprobleme, Stromausfall, reguläre Wartung). Auch der Job muss sich nicht darum kümmern, dass er im Falle eines Scheiterns Aufgaben wiederholt ausführen muss. Dies macht die Entwicklung von Applikationslogik natürlich einfacher, da man sich weniger um die komplexe Behandlung von Fehlerzuständen kümmern muss.

Die Regelkreise von Controllern sorgen somit für eine inhärente Self-Healing-Fähigkeit von Orchestrierungsplattformen. Insbesondere Kubernetes macht von dem in Bild 9.6 gezeigten Prinzip regen Gebrauch.

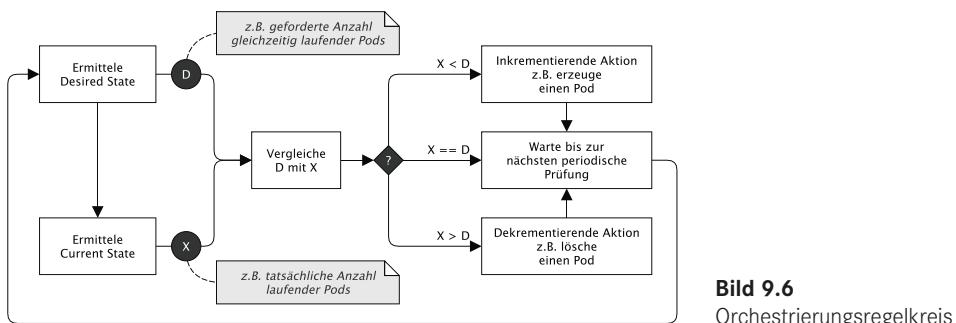


Bild 9.6
Orchestrierungsregelkreis

■ 9.3 Inside Kubernetes

Kubernetes ist ein Cluster-Orchestrator für OCI-konforme Container, der eine Reihe an Kern-Abstraktionen für den Betrieb von Anwendungen in Container-Clustern einführt. Bei Kubernetes handelt es sich um ein Open-Source-Projekt, das von Google initiiert wurde. Google hat damit die jahrelange Erfahrung im Betrieb großer Cluster der Öffentlichkeit zugänglich gemacht, will damit aber natürlich auch Synergien mit dem eigenen Cloud-Geschäft fördern. Kubernetes skaliert bis zu mehrere Tausend Knoten große Cluster, basiert auf einer monolithischen Scheduler-Architektur (siehe Abschnitt 9.1.3.1) und folgt einem regelkreisbasiertem Orchestrierungsansatz (siehe Abschnitt 9.2.2). Blueprints verteilter Anwendungen werden in Kubernetes mittels YAML- oder JSON-Dateien (sogenannte Manifeste) definiert.

Zur Standardisierung der Cluster-Orchestrierung wurde die Cloud Native Computing Foundation (<https://cncf.io>) gegründet. Kubernetes ist das Kernprojekt der CNCF. Die Veranschaulichung entsprechender Orchestrierungskonzepte erfolgt daher am Beispiel von Kubernetes als Typvertreter für solche Orchestrierungsplattformen. Daneben gibt es jedoch weitere

Plattformen wie etwa Mesos, Swarm oder Nomad, die grundsätzlich denselben Konzepten folgen und daher nicht unerwähnt bleiben sollten. Mehr dazu findet der Leser auf der Webseite zu diesem Buch.

Die Orchestrierungsfunktionalitäten basieren dabei u. a. auf Teilfunktionalitäten im Bereich des Schedulings sowie der Netzwerk- und Storage-Virtualisierung:

- **Cluster-Scheduler:** Kubernetes nutzt einen monolithischen Standard-Scheduler (siehe Abschnitt 9.1.3.1). Der Scheduler bestimmt anhand von Constraints und verfügbaren Ressourcen, welche Knoten des Clusters gültige Placements für einen Workload in der Scheduling-Warteschlange sind. Der Scheduler ordnet dann jeden gültigen Knoten anhand seiner Eignung und bindet den Workload an den geeignetsten Knoten. Es können jedoch weitere Scheduler implementiert und neben dem Standard-Scheduler ausgeführt werden. Zu deployende Workloads können dann angeben, mit welchem Scheduler Ressourcen allokiert werden sollen. Auf das Standard-Scheduling wird u. a. in Abschnitt 9.3.3 eingegangen werden.
- **Container Storage Interface (CSI):** Hierbei handelt es sich um eine Schnittstelle, mit der beliebige Block- und File-Storage-Systeme für containerisierte Workloads auf Container-Orchestrierungssystemen wie Kubernetes zugänglich gemacht werden können. Mithilfe von CSI können Storage-Systeme von Drittanbietern mittels Plug-ins in Kubernetes im Sinne einer Storage-Virtualisierung eingebunden werden. Gängige Anbindungen umfassen dabei u. a. Cloud-Storage-Services wie GCE Volumes oder AWS Block-Store, aber NFS- oder iSCSI-basierte Systeme sowie Storage-Cluster wie Ceph oder GlusterFS. Über diese können sogenannte Persistent Volumes für statusbehaftete Container bereitgestellt werden. Insbesondere auf Funktionalitäten zur Anforderung von persistenten Volumes wird u. a. in Abschnitt 9.3.8 eingegangen werden.
- **Container Network Interface (CNI):** Das CNI besteht aus einer Spezifikation und Bibliotheken zum Schreiben von Plug-ins für die Konfiguration von Netzwerkschnittstellen in Linux-Containern. Über CNI können gängige Overlay Networks zur Netzwerkvirtualisierung mittels Plug-ins in Kubernetes für die Container-to-Container-Kommunikation genutzt werden. Mittels dieser Overlay Networks lassen sich virtualisierte Netzwerkfunktionalitäten wie beispielsweise transparente Container-to-Container-Verschlüsselung (weave) oder Netzwerkisolation (Calico) nutzen. Insbesondere auf Funktionalitäten zur Netzwerkisolation wird u. a. in Abschnitt 9.3.9.3 eingegangen werden.

9.3.1 Kubernetes-Architektur

Kubernetes (K8s) ist ein Cluster-System zur Automatisierung der Bereitstellung, Skalierung und Verwaltung von containerisierten Anwendungen. Ein Kubernetes-Cluster besteht dabei aus Master Nodes (Control Plane) und Worker Nodes zur Ausführung von containerisierten Anwendungen. Mehrere Container werden dabei zu sogenannten **Pods** zusammengefasst. Ein Pod kann auch nur aus einem Container bestehen. Alle Container eines Pods werden auf demselben Knoten des Clusters zur Ausführung gebracht. Pods sind in Kubernetes die kleinste schedulbare Workload-Einheit. Als Nutzer dieses Clusters interagiert man normalerweise mit dem Kommandozeilenprogramm kubectl. Zur Ausführung von Anwendungen erforderliche Ressourcen werden in sogenannten Manifestdateien (im YAML- oder JSON-Format) definiert.

Auf einem oder mehreren **Master Nodes** läuft die verteilte **Control Plane** des Clusters, die aus folgenden Komponenten besteht (siehe auch Bild 9.7):

- Dem **Scheduler**, der zusammengehörige Container (im Sprachgebrauch von Kubernetes Pods genannt) Worker Nodes zur Ausführung zuweist.
- **Controller-Managern**, sie überwachen und steuern die Ressourcen von Anwendungen (siehe Abschnitt 9.2.2).
- **Cloud-Managern**, sie kapseln die Anbindungen an Cloud-Infrastrukturen (insbesondere die Anbindung an Load Balancer zum Exponieren von Services).
- Mittels eines **verteilten und konsistenten Key-Value-Stores** (ETCD) werden Clusterzustände innerhalb der Control Plane hochverfügbar und konsistent verwaltet.
- Ein **API-Server** stellt eine Schnittstelle zur Steuerung und Verwaltung des Clusters bereit.
- Weitere **Plug-ins**, wie beispielsweise **Network- und Storage-Plug-ins** dienen der Bereitstellung von Netzwerk- und Persistenz-Funktionalität.

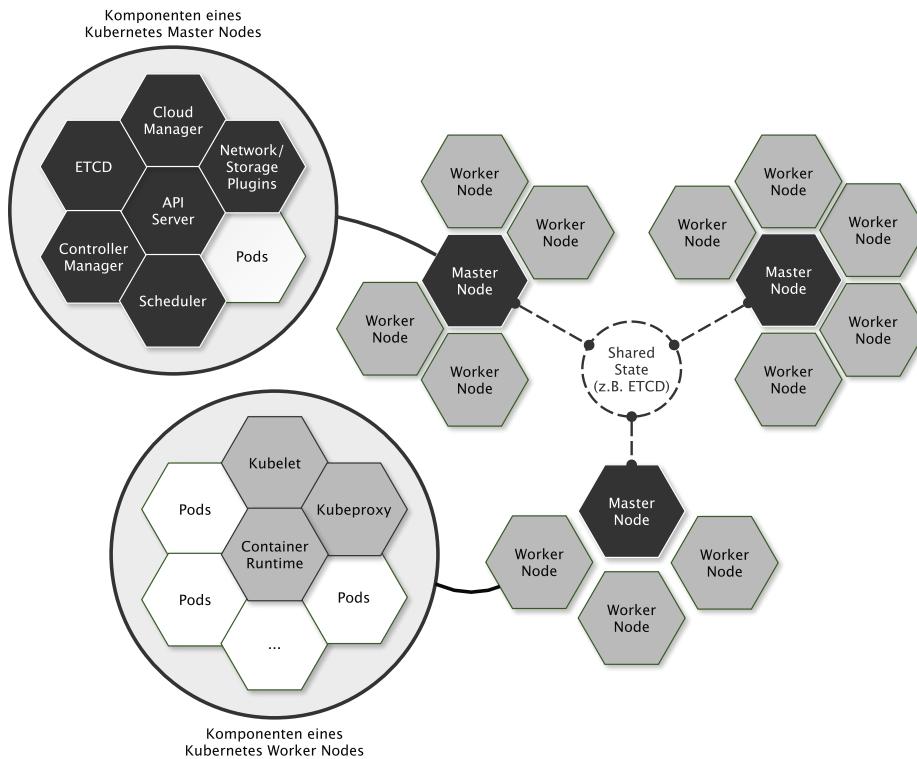


Bild 9.7 Kubernetes-Architektur

Die Komponenten der Control Plane treffen globale Entscheidungen für den Cluster (z. B. Scheduling) und erkennen Cluster-Ereignisse, um darauf zu reagieren (z. B. Neustart von abgestürzten Pods). Alle genannten Komponenten der Control Plane können grundsätzlich auf jedem Knoten im Cluster ausgeführt werden. Um Wechselwirkungen mit Benutzer-Workloads zu vermeiden, werden jedoch meist alle Komponenten der Control Plane auf dedizierten

Master Nodes platziert. Grundsätzlich können auch Workloads (Pods) auf Master Nodes ausgeführt werden. Jedoch sind Cluster aus den genannten Gründen der Workload-Isolation meist so konfiguriert, dass diese nur **Worker Nodes** zur Ausführung zugewiesen werden, die primär der Ausführung von Workloads dienen. Die Interaktion mit der Control Plane erfolgt dabei über folgende Komponenten:

- Das **Kubelet** ist der primäre auf jedem Knoten laufende Node-Agent, der u. a. den Knoten beim **API-Server** registriert.
- Ein **Kube-Proxy** wird als Netzwerk-Proxy auf jedem Knoten ausgeführt, stellt die Kubernetes-API auf jedem Knoten bereit und ist u. a. für Weiterleitung von TCP-, UDP- und SCTP-Traffic sowie Load Balancing für Services verantwortlich.
- Um Container in Pods auszuführen, verwendet Kubernetes eine **Container Runtime Environment**, wie beispielsweise Docker, CRI-O oder containerd.

9.3.2 Verwaltete Ressourcen und Basis-Blueprint

Anwendungen werden in Kubernetes mittels Blueprints beschrieben, die in Form von Manifest-Dateien im YAML- oder JSON-Format ausgedrückt werden. Die Manifeste steuern dabei die Erzeugung der in Bild 9.8 gezeigten Ressourcen, die zum Betrieb einer Anwendung in Kubernetes durch Controller erforderlich sind.

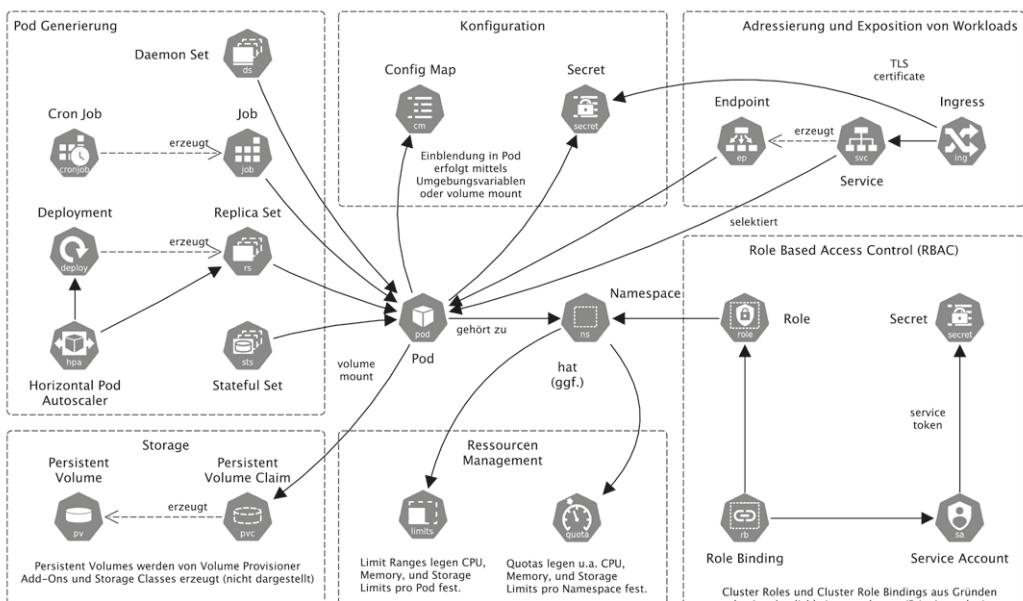


Bild 9.8 Die wichtigsten Kubernetes-Ressourcen und ihre Zusammenhänge

Bild 9.9 zeigt dabei ein Grundgerüst eines Blueprints, das die Grundlage vieler Workloads in Kubernetes bildet. Dabei wird ein horizontal skalierbarer Workload über ein **Deployment** angegeben. Ein **Replication Controller** stellt dabei sicher, dass eine spezifizierte Anzahl an Instanzen eines **Pods** im Cluster laufen. Dies wird im Rahmen eines Regelkreises periodisch geprüft (siehe Bild 9.6), gegebenenfalls werden zusätzliche Pods gestartet oder überzählige Pods heruntergefahren. Um eine variable Menge an Pods unter einem einheitlichen Namen adressieren zu können, dienen **Services**. Diese fassen mehrere zusammengehörige Pods unter einem DNS-Namen zusammen und verteilen Requests mittels eines einfachen Load Balancings über alle Pods. Solche Services können mittels **Ingress**-Ressourcen-Cluster extern bereitgestellt werden. Benötigen Pods einen persistenten Speicher, können diese mittels **Persistent Volume Claims (PVC)** dynamisch von einem Storage-System (**Storage Class**) angefordert und an einen Pod gebunden werden. Umgebungsspezifische Konfigurationen (**Config Maps**) bzw. sensible Zugangsdaten (**Secrets**) können Pods als Umgebungsvariablen oder Konfigurationsdateien zur Verfügung gestellt werden, ohne dass diese sensiblen bzw. sich häufig ändernden Daten in Versionskontrollsystemen oder Image-Registries hinterlegt werden müssten. **Config Maps** und **Secrets** ermöglichen also die Trennung von Code und Konfiguration, wie sie z. B. von der 12-Faktoren-Methodik empfohlen wird (siehe Abschnitt 8.5.2).

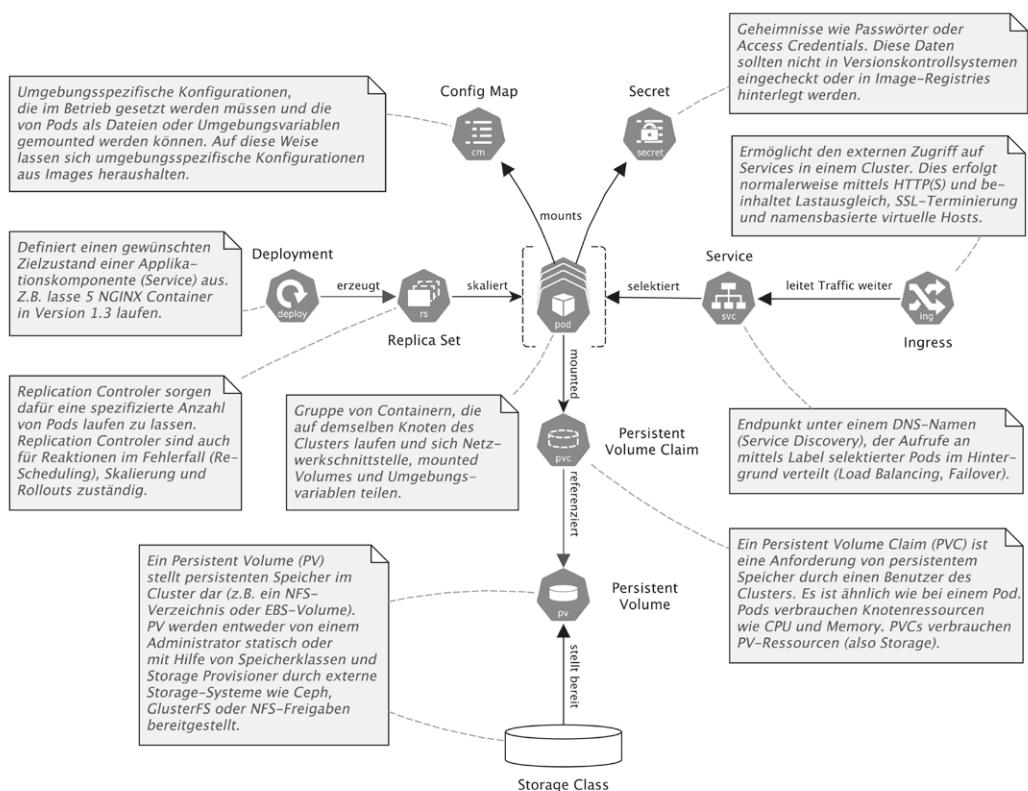


Bild 9.9 Kubernetes-Basis-Blueprint eines Workloads

Darüber hinaus definiert Kubernetes mehrere Dutzend solcher Ressourcen. Die wichtigsten davon und ihre Zusammenhänge sind in Bild 9.8 dargestellt. In den folgenden Abschnitten wird auf diese auch in anderen Plattformen wie etwa Mesos/Marathon oder Swarm unter anderem Namen bekannte Konzepte im Einzelnen eingegangen.

9.3.3 Schedulbare Workloads

Wir haben bereits in Abschnitt 9.1.1 gesehen, dass man zumindest zwei Workload-Kategorien unterscheiden muss, nämlich

- dauerhaft laufende Service- und
- meist einmalig angestoßene, terminierende, aber oft lang laufende und jobartige Prozesse. Kubernetes deckt exakt diese beiden Workload-Arten mit sogenannten Deployments (Service-Prozesse) und Jobs (jobartige Prozesse) ab.

Ergänzend sind aber auch noch sogenannte *Daemon-Sets* und *Stateful-Sets* vorgesehen. Demzufolge kann man mittels Kubernetes folgende in Tabelle 9.1 aufgeführte Workloads betreiben.

Tabelle 9.1 Workloads

| Workload | Einsatzzweck |
|--------------|--|
| Deployment | Kontinuierlich laufende Serviceprozesse zur Erbringung einer Dienstleistung |
| Job | Einmalig angestoßene terminierende Prozesse mit einem Ergebnis |
| Cron Job | Periodisch angestoßene Jobs (siehe dort) |
| Daemon-Set | Hintergrundprozesse, die auf allen Knoten eines Clusters laufen müssen. |
| Stateful-Set | Anwendungen, die stabile, eindeutige Netzwerkbezeichner, eine stabile, beständige Speicherung oder eine geordnete Bereitstellung und Skalierung erfordern. Zumeist sind dies verteilte Datenbanksysteme. |

9.3.3.1 Deployments

Ein Deployment ermöglicht deklaratives Ausbringen und Aktualisieren von Applikationskomponenten (Pods). Sobald die Pods erstellt wurden, überwacht ein zugeordneter **Replica-Set-(RS-)Controller** diese Instanzen kontinuierlich. Sollte z. B. der Knoten, der eine Instanz ausführt, ausfallen oder der Pod gelöscht werden bzw. abstürzen, ersetzt der *Replica-Set*-Controller den Pod durch eine neue Instanz auf einem anderen Node im Cluster, den der Scheduler bestimmt.

Ein *Replica-Set*-Controller überwacht mittels eines Regelkreises den Ist-Zustand des Deployments mit dem im Manifest definierten Soll-Zustand. *Replica-Sets* werden ihrerseits wieder von **Deployment**-Controllern gesteuert. Mittels dieser Trennung der Verantwortlichkeiten zwischen *Deployments* und *Replica-Sets* sind auch Rolling-Updates möglich, z. B. wenn ein Pod auf eine neue Version aktualisiert werden soll. Bild 9.10 zeigt exemplarisch, wie ein Deployment von fünf Nginx-Instanzen (Version 1.14.2) zu einem Deployment von drei Nginx-Instanzen (Version 1.16.1) aktualisiert wird.

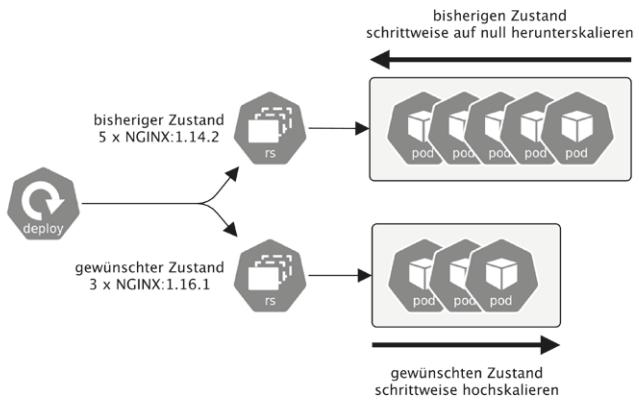


Bild 9.10
Deployment

Der *Deployment Controller* liegt hierzu für die Phase des Übergangs von Version 1.14.2 zu 1.16.1 zwei Replica-Sets an (eines für Version 1.14.2 und eines für Version 1.16.1). Dabei wird schrittweise das *Replica-Set* für Version 1.16.1 auf die Wunschanzahl hochskaliert, während das bestehende *Replica-Set* für Version 1.14.2 schrittweise auf null herunterskaliert und anschließend gelöscht wird.

Für den Nutzer des Clusters stellt sich dieses Update wie folgt im Deployment-Manifest dar.

Listing 9.1 Beispiel eines Deployment-Manifests (nginx-deployment.yaml)

```
apiVersion: apps/v1
kind: Deployment
metadata: { name: nginx-deploy }
spec:
  replicas: 5 # <= Änderung auf 3
  selector: { matchLabels: { app: nginx } }
  template:
    metadata: { labels: { app: nginx } }
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2 # <= Änderung zu nginx:1.16.1
          ports: [{ containerPort: 80 }]
```

Die Anwendung dieses Manifests mittels

```
> kubectl apply -f nginx-deployment.yaml
```

erzeugt dann automatisch die erforderlichen Maßnahmen, wie das Anlegen von *Replica-Set*-Controllern sowie das Starten der Regelkreise, deren Ausführung im Einnehmen des definierten bzw. geänderten Sollzustands in Listing 9.1 endet.

9.3.3.2 (Cron-)Jobs

Jobs sind meist länger laufende Aufgaben (z. B. das Trainieren eines neuronalen Netzes), die einmalig ausgeführt werden. Ein *Job-Controller* erzeugt hierzu ein oder mehrere Pods und trägt Sorge dafür, dass eine vorgegebene Anzahl von diesen erfolgreich terminiert. Es können mehrere Pods parallel ausgeführt werden. Wird ein *Job* gelöscht, werden auch alle von ihm erzeugten Pods gelöscht. Listing 9.2 zeigt exemplarisch einen *Job* zur Berechnung der Zahl Pi mit einer Genauigkeit bis zur 2000. Nachkommastelle.

Listing 9.2 Beispiel eines einmaligen Jobs (job.yaml)

```
apiVersion: batch/v1
kind: Job
metadata: { name: pi }
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
  restartPolicy: Never # <= Container in Jobs nicht neu starten
```

Cron-Jobs sind periodisch auszuführende Aufgaben (z. B. Erstellung einer Monatsabrechnung). *Cron-Jobs* sind dabei dem Cron-Daemon unixoider Betriebssysteme nachempfunden. Ein Cron-Daemon dient der zeitbasierten Ausführung von Prozessen in Unix und Unix-artigen Betriebssystemen wie Linux, BSD oder macOS, um wiederkehrende Aufgaben zu automatisieren. Die Periodizität von Kubernetes-*Jobs* wird daher auch nach dem gleichen Crontab-Format angegeben, das auch von Linux/Unix-Cron-Jobs bekannt ist. Ein Überblick über diese Syntax findet sich in Listing 9.3. Ein *Cron-Job* erzeugt zu den so festgelegten Zeitpunkten *Jobs*, die wie beschrieben ausgeführt werden. Listing 9.4 zeigt exemplarisch einen *Job*, der alle fünf Minuten aufgerufen wird.

Listing 9.3 Cron-Syntax

```
* * * * *
| | | |
| | | +--- Wochentag (0-7, Sonntag ist 0 oder 7)
| | +----- Monat (1-12)
| +----- Tag (1-31)
+----- Stunde (0-23)
----- Minute (0-59)

*           = Ausführung immer
*/n         = Ausführung alle n ...
n,x,y     = Ausführung um/am n, x und y

Beispiele:
0      5  *  * *  Immer um 5 Uhr morgens (Cluster-Zeit)
*/10    *  *  * *  Alle zehn Minuten
0    8,17  *  * *  Immer um 8 Uhr und 17 Uhr      (Cluster-Zeit)
59   23 31 12 *  Immer am 31.12. um 23:59 Uhr (Cluster-Zeit)
```

Listing 9.4 Beispiel eines Cron-Jobs (cronjob.yaml)

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/5 * * * *" # <= Alle fünf Minuten
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
  restartPolicy: OnFailure
```

Wie die Implementierung der Workloads genau aussieht, woher diese ihre Daten beziehen und wo sie ihre Berechnungsergebnisse ablegen bzw. bereitstellen, obliegt wie bei allen durch Kubernetes verwalteten Workloads der Pod-Spezifikation (`template`) und deren Container-Spezifikationen (`containers`).

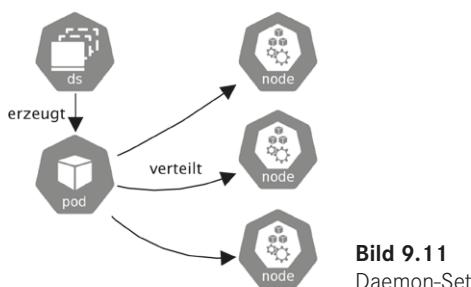
Zu beachten ist allerdings, dass Kubernetes grundsätzlich von kontinuierlich laufenden Prozessen (Services) ausgeht und daher auch bei durch *Job-Controllern* erzeugten Pods von einer `restartPolicy` von `Always` ausgeht. Dies ist bei *Jobs* (egal ob einmalig oder zeitgesteuert) meist nicht gewünscht. Die `restartPolicy` ist bei Jobs daher meist auf `Never` oder `OnFailure` zu setzen.

Wie bei Deployments auch, lassen sich *Jobs* mittels `kubectl` ausbringen und anpassen.

```
> kubectl apply -f cronjob.yaml
```

9.3.3.3 Daemon-Sets

Ein **Daemon-Set** sorgt dafür, dass alle (oder einige) Knoten eine Kopie eines Pods ausführen (siehe Bild 9.11). Workloads, die diese Art des Schedulings erfordern, erbringen häufig



cluster-interne Zusatzdienste (ähnlich wie Hintergrundprozesse in Betriebssystemen, z. B. einem Drucker-Spooler). Immer wenn ein Knoten zum Cluster hinzugefügt wird, lässt ein **Daemon-Controller** automatisch einen Pod auf diesem hinzufügen. Werden Knoten aus dem Cluster entfernt, werden vom *Daemon-Controller* überwachte Pods automatisch gelöscht. Wird ein *Daemon-Set* gelöscht, werden auch die von ihm erstellten Pods gelöscht.

Ein typischer Verwendungszweck für *Daemon-Sets* ist beispielsweise die Log-Konsolidierung auf jedem Knoten (siehe auch Abschnitt 13.1).

Mittels *Daemon-Sets* lassen sich also Workloads betreiben, die auf jedem Knoten des Clusters ausgeführt werden müssen und typischerweise Hintergrund- oder Überwachungsaufgaben dauerhaft ausführen. Grundsätzlich lassen sich *Daemon-Sets* genauso definieren wie auch *Deployments* (siehe Listing 9.1). Für

- `kind` muss jedoch `DaemonSet` gesetzt
- und auf die Angabe von `replicas` muss in der Spezifikation verzichtet werden, da der Controller dafür Sorge trägt, dass pro Node genau ein Pod ausgeführt wird und nach diesem Konzept nicht mehrere Pods pro Node auszuführen sind. Sollten Pods auf einem Knoten ausfallen, werden diese automatisiert neu gestartet.

9.3.3.4 Stateful-Sets

Deployments und *Daemon-Sets* sind Workload-Arten, die im Allgemeinen sehr gut horizontal skalierbar sind. Das bedeutet, die Menge an Pods, die von den entsprechenden Controllern überwacht wird, kann problemlos erhöht oder reduziert werden. Dies hängt damit zusammen, dass diese Arten von Workloads zumeist „stateless“ sind, d. h., beim Wiederanfahren keinen Zustand benötigen, damit sie ihre Leistung wieder erbringen können (bzw. der erforderliche Zustand kann aus einer aus dem Prozess ausgelagerten Zustandskomponente einfach bezogen werden).

In Abschnitt 9.1.1 haben wir aber bereits gesehen, dass Workloads sich auch hinsichtlich ihrer Zustandsbehaftung unterscheiden (Reiss u. a. 2012). Während *Deployments* und *Daemon-Sets* also gut horizontal skalierbare (und damit stateless konzipierte), dauerhaft laufende Prozesse gemäß Abschnitt 8.5.5 abdecken, sich Jobs auf einmalige bzw. wiederkehrende Prozesse fokussieren, gibt es gemäß (Reiss u. a. 2012) und Abschnitt 9.1.1 durchaus Workloads, die sich dieser einfachen horizontalen Skalierungsform aufgrund ihrer „Statefulness“ entziehen.

Kubernetes sieht hierfür sogenannte **Stateful-Sets** vor, die zur Verwaltung von Stateful-Anwendungen vorgesehen sind und diverse Best Practices zum Betrieb zustandsbehafteter verteilter Systeme konzeptionell in der Ressource *Stateful-Set* kapseln (siehe Bild 9.12). Ein *Stateful-Set* verwaltet – wie ein *Deployment* auch – die Bereitstellung und Skalierung eines Satzes von Pods und bietet allerdings Garantien für die Reihenfolge und Einzigartigkeit dieser Pods. Wie ein *Deployment* verwaltet auch ein *Stateful-Set* Pods, die auf einer identischen Container-Spezifikation beruhen. Im Gegensatz zu einem *Deployment* verwaltet ein *Stateful-Set* für jeden seiner Pods jedoch eine unveränderliche Identität, die auch Restarts des Pods überdauert. Diese Pods werden anhand derselben Spezifikation erstellt, sind aber nicht austauschbar. Jeder hat Pod erhält eine dauerhafte Kennung, die er bei jeder Neuplanung beibehält. *Deployments* erzeugen hingegen zufällige Kennungen für Pods (da diese aufgrund der Annahme der horizontalen Skalierbarkeit austauschbar und flüchtig sind).

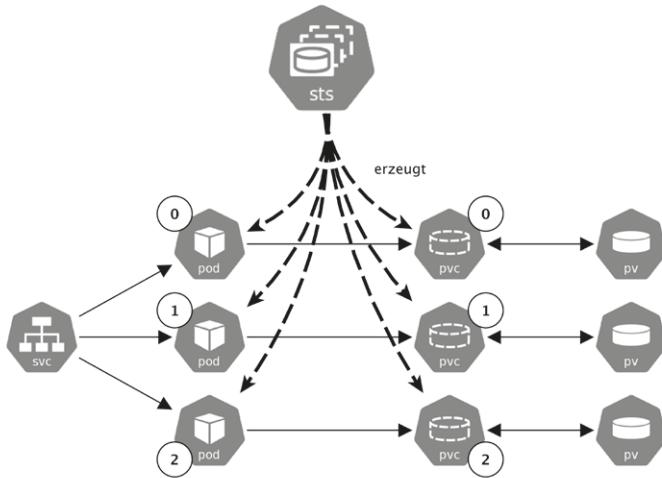


Bild 9.12
Stateful-Set

Da Stateful-Anwendungen oft die Nutzung persistenter Volumes erforderlich machen, kann ein *Stateful-Set* mittels einem Volume Claim-Template zu jedem Pod ein ihm zugeordnetes Volume erzeugen, das auch bei einem Restart des Pods aufgrund seiner eindeutigen Kennung diesem wieder zugeordnet werden kann. Auf diese Weise lässt sich ein persistierter Zustand auf einem Volume auch über Restarts hinweg erhalten und wieder eindeutig einem neu zu startenden Pod zuordnen. In Abschnitt 9.3.8 werden wir noch genauer auf Persistenz mittels Volumes in Kubernetes eingehen.

Stateful-Sets sind somit wertvoll für den Betrieb von Anwendungen, die einen oder mehrere der folgenden Punkte erfordern.

- Beständige und eindeutige Netzwerkbezeichner (DNS-Namen von Pods)
- Beständige Speicherung (Volumes)
- Geordnete Bereitstellung und Skalierung von Komponenten (Pods) in der aufsteigenden Ordnung von Ordinalnummern von Pods

Auf diese Weise ergibt sich eine stabile Adressierung und Zuordnung von Pods zu DNS-Namen und Volumes (auch über Reschedulings/Neustart von Pods hinweg). *Stateful-Sets* erfordern allerdings einen Headless Service (siehe Abschnitt 9.3.6), der für die Netzwerkidentität der Pods verantwortlich ist. Dieser Service wird allerdings nicht durch den Stateful-Set-Controller selber erstellt, sondern muss durch den Nutzer erstellt werden.

Listing 9.5 zeigt die Wirkungsweise von *Stateful-Sets* am Beispiel eines ETCD-Clusters. ETCD ist ein verteilter, auf dem RAFT-Algorithmus (Ongaro und Ousterhout 2014) beruhender Key-Value Store, der einen ausfallsicheren und streng konsistenten Zustand in einem Verbund mehrerer Nodes herstellen und erhalten kann. Hierbei ist sicherzustellen, dass die Knoten sich untereinander im Verbund kennen. Die Knoten 1 bis 4 (etcd-1, etcd-2, etcd-3, etcd-4) müssen sich daher beim Knoten 0 (etcd-0) registrieren. Soll der Cluster heruntergefahren werden, muss der jeweilige Knoten sich im Verbund abmelden (also etcd-4 bspw. bei etcd-3, usw.). Fällt im Verbund ein Knoten aus (z. B. etcd-3), so gibt es dann nur noch die Knoten etcd-0, -1, -2, und -4 (aber nicht mehr etcd-3). Wird nun ein neuer Pod gestartet, erhält dieser die fehlende Ordinalnummer. Man kann diesem auch das entsprechende Volume des

vorherigen Knotens zuordnen (nämlich vol-3). Auf diese Weise ergibt sich über die DNS-Namen und Volume-Nummerierung eine stabile Adressierung und Persistenz, auch wenn Pods durchaus auf anderen physischen Knoten gestartet werden und andere IP-Adressen zugewiesen bekommen.

Listing 9.5 Beispiel eines Stateful-Sets (statefulset.yaml)

```

apiVersion: v1
kind: Service
metadata: { name: etcd }
spec:
  clusterIP: None # <= Headless Service
  selector: { app: etcd }

---
apiVersion: apps/v1
kind: StatefulSet
metadata: { name: etcd }
spec:
  serviceName: etcd
  replicas: 5
  selector:
    matchLabels: { app: etcd }
  template:
    metadata:
      name: etcd
      labels: { app: etcd }
    spec:
      containers:
        - name: etcd
          image: etcd:3.2.24
          volumeMounts: [{ name: datadir, mountPath: /var/run/etcd }]
  volumeClaimTemplates: # <= Template zur Erzeugung eines Volumes pro Pod
    - metadata: { name: datadir }
      spec:
        accessModes: ["ReadWriteOnce"]
        resources: { requests: { storage: 1Gi } }

```

Die Stabilität wird auch in Skalierungssituationen erhalten. Wird der Cluster hochskaliert, werden weitere Pods mit der nächsten der bislang höchsten vergebenen Ordinalnummer fortnumeriert. Dabei wird nur ein Pod zur Zeit hochgefahren. Wird der Cluster herunterskaliert, wird zuerst der Pod mit der bislang höchsten vergebenen Ordinalnummer heruntergefahren, dann der nächste mit der dann höchsten Ordinalnummer und so weiter. Auch hier wird nur ein Pod zur Zeit heruntergefahren. Auf diese Weise ergibt sich auch bei Skalierungen immer eine stabile Situation für $n - 1$ betroffene Pods. Einzig der gerade in Skalierung befindliche Pod ist in einem undefinierten Zustand. Die Skalierung ist zugunsten der Stabilität allerdings im Allgemeinen deutlich langsamer als bei Deployments. Aus diesem Grund sollten *Stateful-Sets* auch nur für zustandsbehaftete Anwendungen (wie bspw. Datenbanken) verwendet werden, bei denen man sicherstellen muss, dass diese nicht aus zustandserhaltenen Situationen heraußskaliert werden.

9.3.4 Scheduling Constraints

Zur Platzierung von Workloads benötigt der Scheduler zusätzliche Informationen zum Ressourcenbedarf, damit geeignete Ressourcen möglichst effizient zugewiesen werden können. Ergänzend kann das Scheduling mittels Randbedingungen eingeschränkt werden.

9.3.4.1 Angabe des Ressourcenbedarfs mittels Requests und Limits

Der Ressourcenbedarf zur Ausführung eines Workloads kann dabei mittels **Requests** und **Limits** in jeder Container-Spezifikation angegeben werden. Mittels *Requests* und *Limits* steuert Kubernetes Ressourcen wie CPU und Speicher.

- **Requests** sind die Ressourcenmenge, die ein Container garantiert bekommt. Der Scheduler plant einen Pod nur für einen Knoten ein, der ihm diese Ressource auch sicher bereitstellen kann.
- **Limits** stellen hingegen sicher, dass ein Container niemals einen bestimmten Wert überschreitet. *Limits* können niemals niedriger sein als der *Request*. Überschreitet ein Container sein Limit, wird er von der Container Runtime Engine automatisch terminiert.

Requests und *Limits* werden pro Container in den Container-Specs angegeben (siehe Listing 9.6) und entsprechen den Minimal- und Maximalangaben, die vom kapazitätsbasierten Scheduling (siehe Abschnitt 9.1.2.3) bekannt sind. Da Pods immer als Gruppe geplant werden, müssen *Requests* und *Limits* für alle Container eines Pods addiert werden, um einen Gesamtwert für den Pod zu erhalten. Um zu steuern, wie viele Ressourcen insgesamt vergeben werden können, können pro Namespace Quotas festgelegt werden (siehe auch Abschnitt 9.3.9.2).

Listing 9.6 Beispiel eines Deployment-Manifests mit Ressourcenbedarf

```
apiVersion: apps/v1
kind: Deployment
metadata: { name: nginx-deploy }
spec:
  replicas: 5
  selector: { matchLabels: { app: nginx } }
  template:
    metadata: { labels: { app: nginx } }
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports: [{ containerPort: 80 }]
          resources:
            requests: { cpu: "100m", memory: "100M" } # <= Ressourcenanforderung
            limits: { cpu: "1000m", memory: "1G" } # <= Maximum
```

Werden keine Ressourcenangaben in einem Workload-Manifest angegeben, plant der Scheduler nach dem „Best Effort“-Prinzip. Dies kann jedoch bedeuten, dass ein Workload zu wenige Ressourcen zur Ausführung erhält oder ein Workload andere Workloads auf einem Knoten dominiert.

Requests und *Limits* für CPU-Ressourcen werden in CPU-Einheiten (Millicores) gemessen. Eine CPU in Kubernetes entspricht 1 vCPU / Core für Cloud-Anbieter und 1 Hyperthread auf Bare-Metal-Prozessoren. Bruchteile von Anforderungen sind zulässig, d. h., ein Container mit einem Request von 0.5 erhält garantiert halb so viel CPU wie einer, der 1 CPU anfordert. Der Ausdruck 0.1 entspricht dem Ausdruck 100 m, der üblicherweise als „einhundert Millicpu“ (oder Millicores) gelesen wird. Eine Anforderung mit einem Dezimalpunkt wie 0.1 wird von der API in 100 m konvertiert. Eine Genauigkeit von weniger als 1 m ist nicht zulässig. Die CPU wird immer als absolute Menge angefordert, niemals als relative Menge. Das bedeutet, 0.1 ist die gleiche CPU-Menge auf einem Single-Core-, Dual-Core- oder 48-Core-Computer (die Taktrate der Prozessoren spielt dabei keine Rolle).

Speicher-*Requests* und -*Limits* werden hingegen in Bytes gemessen. Eine Speichermenge kann als einfache Ganzzahl oder als Festkommazahl mit einem der folgenden Suffixe ausgedrückt werden:

- E: (Exa-Byte)
- P: (Peta-Byte)
- T: (Tera-Byte)
- G: (Giga-Byte)
- M: (Mega-Byte)
- K: (Kilo-Byte)

Es können auch die Zweierpotenzäquivalente verwendet werden: Ei, Pi, Ti , Gi, Mi, Ki.

9.3.4.2 Knoten-Selektoren

Der Scheduler kann ferner mittels *Constraints* und *Affinities* Randbedingungen für das Scheduling mitgeteilt bekommen. Damit lässt sich beispielsweise einschränken bzw. präferieren, auf welcher Art von Knoten ein Workload ausgeführt werden soll.

Mittels Node-Labels lassen sich Nodes kennzeichnen, um so dem Scheduler zu ermöglichen, bestimmte Workloads nur bestimmten Nodes zuweisen zu können.

Listing 9.7 Labeling von Nodes

```
# Nodes lassen sich mittels Key-Value Paaren labeln,
# z. B. so:
kubectl label nodes node-1 disktype:ssd
kubectl label nodes node-2 disktype:ssd
kubectl label nodes node-3 disktype:hdd
kubectl label nodes node-4 disktype:hdd
```

Dies kann z. B. sinnvoll bei Clustern sein, deren Knoten nicht vollkommen homogen aufgebaut sind. Im Beispiel aus Listing 9.7 sind exemplarisch Nodes mit schnellen SSD-Platten und (aus Kostengründen) Nodes mit langsameren HDD-Platten konfiguriert. Aus Geschwindigkeitsgründen könnte gewollt sein, dass ein Webserver eines Content Deliver Networks nur auf Nodes mit SSD-Platten ausgeführt wird, um geringe Auslieferungszeiten von Storage bis Browser sicherstellen zu können. Solche Scheduling-Randbedingungen lassen sich in Pod-Spezifikationen in Form von Node-Selektoren realisieren, wie Listing 9.8 exemplarisch zeigt.

Listing 9.8 Beispiel eines Node-Selektors

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
  nodeSelector: { disktype: ssd } # <= harte Scheduling Randbedingung
```

9.3.4.3 Knotenaffinitäten

Node-Selektoren ermöglichen es dem Scheduler, die Ausführung von Pods auf Knoten mit bestimmten UND-verknüpften Labeln hart zu beschränken. Diese Einschränkung ist jedoch in vielen Fällen zu hart und einschränkend und wurde daher durch ein Affinitätskonzept ergänzt. Dieses erweitert die ausdrückbaren Randbedingungen erheblich:

- Affinitäten sind ausdrucks voller, da sie neben exakten (UND-verknüpften) Übereinstimmungen mehr Übereinstimmungsregeln bieten.
- Es lässt sich angeben, dass Affinitäten zwar zu präferieren, aber keine harte Anforderung sind. Wenn der Scheduler Affinitäten nicht erfüllen kann, wird der Pod in diesen „weichen“ Fällen dennoch geplant.
- Es können Affinitäten formuliert werden, die sich auch auf andere Pods (und deren Platzierung im Cluster) beziehen. So lassen sich beispielsweise Regeln festlegen, die vermeiden, dass Pods auf identische Nodes platziert werden (z. B. um Ausfallwahrscheinlichkeiten zu minimieren).

Es gibt zwei Arten von Affinitäten. Die *Knotenaffinität* ähnelt Node-Selektoren (jedoch mit den ersten beiden der oben aufgeführten Vorteile). Die *Inter-Pod-(Anti)-Affinität* bezieht sich auf Pods und wertet Pod-Labels und aktuelle Pod-Platzierungen aus.

Es gibt zwei Arten von *Knotenaffinitäten*:

- `requiredDuringSchedulingIgnoredDuringExecution` (harte Randbedingung, ähnlich Node-Selektor)
- `preferredDuringSchedulingIgnoredDuringExecution` (weiche Randbedingung)

Der Teil `RequiredDuringSchedulingIgnoredDuringExecution` bedeutet, dass der Pod nicht umgeplant wird, wenn sich zur Laufzeit Node-Labels ändern. Der Kubernetes-Scheduler unterstützt aktuell keine Labeländerungen zur Laufzeit.

Solche *Knotenaffinitäten* lassen sich in Workload-Manifesten innerhalb der Pod-Spezifikationen angeben. Das in Listing 9.9 gezeigte Beispiel präferiert eigene On-Premise Knoten (gelabelt als `provider=on-premise`). Dieser Workload würde also bevorzugt im eigenen Rechenzentrum platziert werden, es sei denn, dort findet der Scheduler keinen Platz mehr. Nur dann würden gegebenenfalls teurere Public Cloud-Maschinen genutzt werden (gelabelt bspw. als `provider=aws`, `provider=gce` oder `provider=azure`).

Listing 9.9 Beispiel einer Knotenaffinität

```

apiVersion: v1
kind: Pod
metadata: { name: pod-with-constant-workload }
spec:
  containers:
    - name: constant-workload
      image: crunch:2.0
  affinity: # <= weiche Knotenaffinität
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: provider
                operator: In
                values: [ "on-premise" ]

```

9.3.4.4 Pod-(Anti-)Affinitäten

Ergänzend zu den Knotenaffinitäten kann man Randbedingungen für das Scheduling auf Basis von Pod-Labels (und nicht anhand von Knotenlabels) definieren. Solche (Anti-)Affinitätsregeln haben die Form: Dieser Pod sollte/darf (nicht) auf Knoten X ausgeführt werden, wenn auf Knoten X bereits ein Pod ausgeführt wird, der Regel Y erfüllt.

Konzeptionell werden die Knoten X über einen von Kubernetes vergebenen Topologieschlüssel seiner Knoten vergeben, wie etwa Knoten, Rack, Cloud-Provider-Zone, Cloud-Provider-Region und mehr. Das Beispiel in Listing 9.10 schränkt das Scheduling von Redis Pods (In-Memory Cache) beispielsweise so ein, dass nie zwei Pods auf demselben Knoten platziert werden. Dies kann aus Gründen der gleichmäßigen Verteilung von Netzwerklasten und Ausfallsicherheit durchaus Sinn machen.

Listing 9.10 Beispiel einer Pod-Anti-Affinität

```

apiVersion: apps/v1
kind: Deployment
metadata: { name: redis-cache }
spec:
  selector:
    matchLabels: { app: store }
  replicas: 3
  template:
    metadata:
      labels: { app: store }
  spec:
    containers:
      - name: redis-server
        image: redis:3.2-alpine
    affinity: # <= Harte Pod-Anti-Affinität
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:

```

```

- key: app
  operator: In
  values: ["store"]
topologyKey: "kubernetes.io/hostname"

```

9.3.5 Automatische Skalierung von Workloads

Deployment-Workloads können mittels der `replicas`-Angabe in der Workload-Spezifikation eines Manifests einfach hoch- und heruntergesetzt werden. Die Skalierung erfolgt dann durch Aktualisierung und Anwendung eines *Deployment*-Manifests (siehe Abschnitt 9.3.3.1) oder mittels des Command Line Interfaces `kubectl`.

```
kubectl scale deployment my-service --replicas 4
```

Beide Varianten (Manifest oder mittels `kubectl scale`) sind jedoch Aktionen, die zumeist eine menschliche Interaktion inklusive kognitiven Entscheidungsprozess erfordern. Um Lastsituationen mit einem Workload möglichst eng und agil folgen zu können, wäre es besser, diese Skalierungsentscheidung durch einen Regelkreis auf Basis der aktuellen Lastsituation automatisiert erfolgen zu lassen. Dies wird in Kubernetes als horizontale Pod-Autoskalierung bezeichnet.

Die horizontale Pod-Autoskalierung ist als vollautomatischer Regelkreis konzipiert, der periodisch die Ressourcenauslastung eines Workloads anhand der in einem *Horizontal Pod Autoscaler*-Manifest angegebenen Metriken abgreift und mit einem Zielwert abgleicht und auf Basis dessen die Anzahl von Pods eines Deployments herauf- oder heruntersetzt (siehe Bild 9.13). Der Regelkreis nutzt die Ressourcenmetriken (insbesondere CPU und Hauptspeicher) von der Kubernetes-Metrikkomponente. Grundsätzlich können auch selbst definierte Metriken zur Autoskalierung herangezogen werden. Diese Option wird hier allerdings nicht weiter betrachtet.

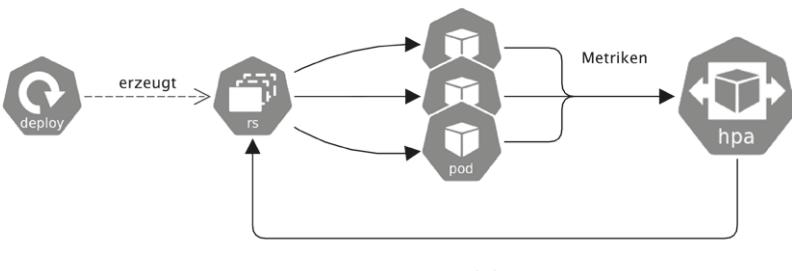


Bild 9.13 Horizontal Pod Autoscaler (HPA)

Der Regelkreis der horizontalen Pod-Autoskalierung arbeitet dabei auf dem Verhältnis zwischen dem gewünschten Wert einer Metrik (d^M , desired state) und dem aktuellen Wert einer Metrik (c^M , Current State) und verknüpft dieses mit der aktuellen Anzahl an Anzahl an Pods r eines Deployments, um die neue Anzahl an Pods r' für die nächste Iteration des Regelkreises zu bestimmen. Auch bei der Autoskalierung handelt es sich also um einen Orchestrierungs-

regelkreis gemäß Abschnitt 9.2.2. Zur Berechnung des aktuellen Werts der Metrik wird der Mittelwert der gegebenen Metrik über alle Pods C_i^M des Deployments berechnet.

$$r' = [r \frac{C^M}{d^M}] = [r \frac{\frac{1}{r} \sum_{n=1}^r C_n^M}{d^M}] = [\frac{1}{d^M} \sum_{n=1}^r C_n^M]$$

Wenn der aktuelle Wert der Prozessorauslastung beispielsweise bei 200m liegt und der gewünschte Wert auf 100m festgelegt wurde, wird die Anzahl der Replikate verdoppelt, da $\frac{200}{100} = 2$. Wenn der aktuelle Wert stattdessen 50m ist, wird die Anzahl der Replikate halbiert, da $\frac{50}{100} = \frac{1}{2}$. Die Skalierung wird übersprungen, wenn das Verhältnis nahe genug an 1 innerhalb einer global konfigurierbaren Toleranz liegt. Diese ist standardmäßig auf 10 % eingestellt.

Zusätzlich gibt es einen speziellen Befehl `kubectl autoscale` zur einfachen Erstellung eines **Horizontal Pod Autoscalers (HPA)**. Mittels

```
kubectl autoscale deploy my-service --min=2 --max=5 --cpu-percent=80
```

lässt sich ein Autoskalierer für das Deployment `my-service` erstellen. Die Ziel-CPU-Auslastung (Desired State) ist dabei auf 80 % (d. h. 800 m) festgelegt, und die Anzahl an Pods wird zwischen zwei und maximal fünf Pods durch den Regelkreis skaliert werden.

Horizontale Autoskalierung ist vor allem für *Deployment*-Workloads vorgesehen und probat, wenn die Skalierung mittels einfacher Ressourcenmetriken wie CPU- oder Hauptspeicherauslastung erfolgen kann und eine Skalierung bis auf null Pods (Scale-to-Zero) nicht gewünscht oder erforderlich ist. Häufig ist eine Skalierung jedoch auf Basis von Requests oder zu verarbeitenden Ereignissen im Sinne einer ereignisgesteuerten Skalierung erforderlich. Auch eine Scale-to-Zero-Fähigkeit ist für viele Anwendungsfälle aus wirtschaftlichen Erwägungen eine interessante Option. Für diese Art von Skalierungserfordernissen sind jedoch oft FaaS-basierte Ansätze, die ergänzend zu Orchestrierungsplattformen eingesetzt werden können, besser geeignet. FaaS-basierte Ansätze und ereignisbasierte Autoskalierung werden daher noch genauer in Kapitel 10 betrachtet werden. Diese Arten der Skalierung gehören nicht zum Standardumfang von Kubernetes.

9.3.6 Exponieren von Workloads als interne und externe Services

Kubernetes ermöglicht es, eine Menge von Pods als Netzwerkdienst unter einem DNS-Namen bekannt zu machen. Dies ermöglicht ein DNS-basiertes Service-Discovery. Hierzu werden die Pods mittels Selektoren bestimmt und in Form eines **Services** bereitgestellt (siehe Listing 9.11). Dies kann cluster-intern oder cluster-extern erfolgen. Kubernetes gibt Pods ihre eigenen IP-Adressen und *Services* einen eindeutigen DNS-Namen. Pods, die über einen *Service* angesprochen werden, unterliegen automatisch einem Load Balancing über die IP-Adressen der selektierten Pods.

Listing 9.11 Beispiel einer Service-Ressource

```

apiVersion: v1
kind: Service
metadata: { name: http-service }
spec:
  selector:          # <= Pod-Selektion
    labels: { app: nginx } # <= zu selektierende Labels
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 8080
    - name: https
      protocol: TCP
      port: 443
      targetPort: 8443
---
apiVersion: apps/v1
kind: Deployment
metadata: { name: nginx-deployment }
spec:
  replicas: 3
  selector:
    matchLabels: { app: nginx }
  template:
    metadata:
      labels: { app: nginx } # <= Label-Zuweisung an Pod
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 8080
            - containerPort: 8443

```

Services werden standardmäßig nur cluster-intern bereitgestellt. Dies erfolgt mittels des Service-Typs `ClusterIP`. Services können aber auch cluster-extern exponiert werden (siehe Bild 9.14):

- Der Service-Typ `LoadBalancer` bindet einen Dienst an eine Public IP eines Cloud-Providers.
- Der Service-Typ `NodePort` macht einen Service-Port auf allen IP-Adressen der Cluster-Knoten bekannt.
- Mittels `ExternalName` können ergänzend auch externe Dienste in den Cluster eingebettet und wie cluster-interne Dienste genutzt werden.

Die Exponierung mittels Load Balancern funktioniert meist nur in Public IaaS-provisionierten Clustern. Insbesondere auf self-managed Clustern können Services oft nur mittels HTTP-basiertem Ingress exponiert werden. Dies hat den Hintergrund, dass Load Balancer eine Public-IP-Adresse für jeden exponierten Service erfordern, die beim Public Cloud-Service-Provider über den Cloud-Manager angefordert werden. Insbesondere Public IP-V4-Adressen sind jedoch mittlerweile recht knappe Ressource.

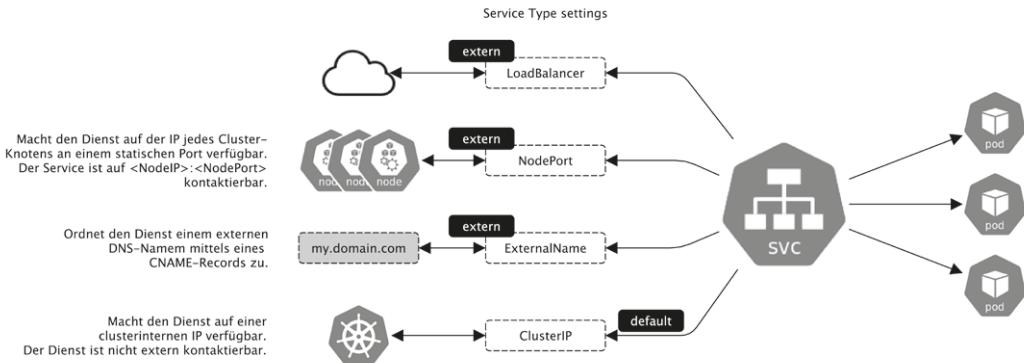
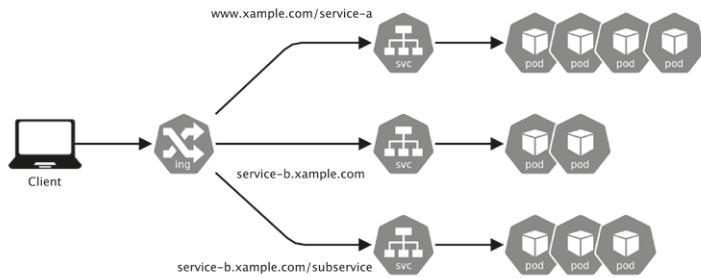


Bild 9.14 Service Types

Daher werden insbesondere HTTP-basierte Services (Webservices) meist über ein sogenanntes **Ingress** exponiert (siehe Bild 9.15). Ein *Ingress* macht externe HTTP- und HTTPS-Routen für Services innerhalb des Clusters verfügbar. Grundsätzlich können *Ingresses* so konfiguriert werden, dass eine Public-IP-Adresse für alle auf einem Cluster betriebenen Webservices geteilt werden kann. Anders als service-basierte NodePort- oder LoadBalancer-Exponierungen (die pro Exponierung jeweils eine eindeutige IP-Adresse und Portnummer benötigen), braucht ein Ingress nur eine einzige IP-Adresse und eine Portnummer, um eine Vielzahl an Diensten zu exponieren. Damit ist die Exponierung von Services mittels Ingress-Ressourcen hinsichtlich der knappen Ressource von Public IP-Adressen (nur IP-V4) deutlich schonender als die Bereitstellung über LoadBalancer. *Ingress* laufen ferner auf Anwendungsschicht des Netzwerkstapels (HTTP) und bieten daher Funktionen wie beispielsweise cookie-gestützte Sitzungsaffinitäten und können ferner mittels TLS-Zertifikaten abgesichert werden (HTTPS).

Bild 9.15
Ingress

Ein *Ingress* kann so konfiguriert werden, dass für *Services* extern erreichbare URLs (inklusive Load Balancing, SSL/TLS-Terminierung und namenbasiertes virtuelles Hosting) angeboten werden können. Das Routing wird hierzu durch Regeln gesteuert, die in einem Ingress-Manifest definiert werden (siehe auch Listing 9.12). Allerdings funktionieren *Ingress*-Ressourcen meist nicht mit Services, die andere Protokolle als HTTP(S) verwenden, da sie auf einer HTTP-basierten Reverse Proxy-Lösung wie z. B. NGINX basieren. Für andere Protokolle als HTTP muss daher normalerweise ein Service vom Typ NodePort oder LoadBalancer verwendet werden.

Listing 9.12 Beispiel einer Ingress-Ressource

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata: { name: example-ingress }
spec:
  rules:
    - host: www.xample.com # <= öffentlicher DNS-Name
      http:
        paths:
          - path: /service-a # <= Route http://www.xample.com/service-a
            pathType: Prefix
            backend: # <= zu exponierender Service
              service:
                name: service-a
                port: { number: 8080 }

```

Auch wenn dies auf den ersten Blick sehr einschränkend klingt, sind insbesondere REST-basierte APIs und gRPC-basierte Protokolle eine verbreitete Art und Weise, Schnittstellen zu Cloud-nativen Services bereitzustellen. Sowohl REST als auch gRPC basiert auf dem HTTP-Protokoll. Von diesem Setting wird oft nur bei Datenbank- oder Messaging-Systemen abgewichen, die allerdings oft nur cluster-intern genutzt und nicht als Public-Service exponiert werden. Insofern hat sich die *Ingress*-Ressource zu einem probaten und einfach gut konfigurierbaren „Eingangstor“ für öffentliche Service-APIs etabliert.

9.3.7 Health Checking

Neben der Ressourcenzuweisung an Workloads ist für einen dauerhaften und verlässlichen Betrieb eines Workloads auch erforderlich, dessen „Gesundheit“ kontinuierlich zu überwachen. Viele Prozesse, die über einen längeren Zeitraum ausgeführt werden, enden gegebenenfalls in einem fehlerhaften Zustand (z. B. in einem seltenen Deadlock), stürzen ab und können nur durch einen Neustart wiederhergestellt werden. Zur Detektion und automatisierten Behebung solcher problembehafteten Workload-Zustände bietet Kubernetes sogenannte **Probes** an.

Mittels sogenannter *Liveness Probes* lässt sich ein sogenannter Heart Beat für Workloads realisieren, den die Plattform kontinuierlich prüft. Liveness Probes können pro Container in der Container-Spezifikation eines Workload-Manifests definiert werden. Definiert werden können Command-, HTTP- oder TCP-basierte Probes.

Diese Probes werden in definierten Zeitintervallen geprüft. Ist die Prüfung mehrmals (definierbar über einen Schwellwert) erfolglos, wird der Container in einem solchen Fehlerfall über die Container Runtime Environment automatisch neu gestartet.

Listing 9.13 zeigt einen command-basierten *Probe* innerhalb einer Container-Spezifikation. Hierzu wird das Command im Container ausgeführt. Ein Exit-Code von 0 wird als „Alive“ interpretiert, alles andere als „Failure“. Die Überprüfung erfolgt alle zehn Sekunden.

Listing 9.13 Command-basierter Liveness Probe (Auszug)

```

livenessProbe:
  exec:
    command:
      - cat
      - ./tmp/healthy
  failureThreshold: 1
  periodSeconds: 10

```

Listing 9.14 zeigt einen HTTP-basierten *Probe* innerhalb einer Container-Spezifikation. Bei HTTP-Probes werden Return-Codes zwischen 200 und 399 als „Alive“ interpretiert, alles andere als „Failure“. Die Überprüfung erfolgt alle zehn Sekunden. Der Container wird als fehlerhaft neu gestartet, wenn mehr als drei „Failure“-Zustände direkt hintereinander erkannt werden.

Listing 9.14 HTTP-basierter Liveness Probe (Auszug)

```

livenessProbe:
  httpGet:
    path: /healthz
    port: 80
  failureThreshold: 3
  periodSeconds: 10

```

Listing 9.15 zeigt einen TCP-basierten Probe auf Port 3306 (z. B. ein MySQL-Server). Ein erfolgreicher Aufbau wird als „Alive“ interpretiert, alles andere als „Failure“. Die Überprüfung erfolgt alle zehn Sekunden. Der Container wird als fehlerhaft neu gestartet, wenn mehr als drei „Failure“-Zustände direkt hintereinander erkannt werden.

Listing 9.15 TCP-basierter Liveness Probe (Auszug)

```

livenessProbe:
  tcpSocket:
    port: 3306
  failureThreshold: 3
  periodSeconds: 10

```

Manchmal können Prozesse auch Requests nur vorübergehend nicht bedienen. So könnte eine Anwendung möglicherweise beim Start große Datenmengen oder Konfigurationsdateien laden oder nach dem Start von externen Diensten abhängig sein. In solchen Fällen sollte der Prozess nicht beendet werden, allerdings auch keine neuen Requests bekommen. Kubernetes bietet hierfür sogenannte **Readiness Probes** an, die solche Situationen zu erkennen und zu mildern helfen. Ein Pod mit Containern, die mittels *Readiness Probes* melden, dass sie nicht bereit sind, empfängt beispielsweise keinen Datenverkehr mehr über Kubernetes Services. *Readiness Probes* können grundsätzlich wie **Liveness Probes** als Command-, TCP- oder HTTP-basierte Probes definiert werden (siehe Listing 9.16).

Listing 9.16 Command-basierter Readiness Probe (Auszug)

```
readinessProbe:
  exec:
    command:
      - cat
      - /tmp/ready
  failureThreshold: 1
  periodSeconds: 10
```

Insbesondere bei sogenannten Legacy-Applikationen kommt ein weiteres Problem in Cloud-nativen Szenarien hinzu. Legacy-Applikationen haben oft sehr lange Startzeiten (manchmal im Bereich von Minuten). In solchen Fällen kann es schwierig sein, Parameter für einen *Liveness Probe* einzurichten. Hierzu kann ein **Start-up Probe** eingerichtet werden. Dieser wird erst geprüft und hat üblicherweise einen höheren Fehlerschwellenwert und eine längere Prüfperiode, um auch ungewöhnlich lange Startzeiten abdecken zu können. Der in Listing 9.17 gezeigte *Start-up Probe* deckt Start-up-Phasen von bis zu fünf Minuten ab, dabei wird alle 30 Sekunden geprüft, ob die Datenbank erreichbar ist. Es werden während des Starts zehn aufeinanderfolgende Fehler akzeptiert. Erst dann würde die Readiness- und Liveness-Prüfung vorgenommen werden.

Listing 9.17 TCP-basierter Start-up Probe (Auszug)

```
startupProbe:
  tcpSocket:
    port: 3306
  failureThreshold: 10
  periodSeconds: 30
```

Tabelle 9.2 zeigt die Parameter, mit denen das Timing und die Schwellwerte der Fehlererkennung in Probes definiert werden können.

Tabelle 9.2 Konfigurationsparameter von Probes

| Parameter | Erläuterung |
|---------------------|---|
| initialDelaySeconds | Anzahl der Sekunden nach dem Start des Containers, bevor Liveness- oder Readiness-Tests gestartet werden. |
| periodSeconds | Zeitlicher Abstand (in Sekunden), mit dem ein Probe gecheckt wird. |
| timeoutSeconds | Anzahl der Sekunden, nach denen ein Probe antworten muss. |
| successThreshold | Minimale Anzahl aufeinanderfolgender Erfolge, um ein Probe nach einem Fehler als erfolgreich anzusehen. |
| failThreshold | Anzahl an Wiederholungsversuchen, bevor ein Probe als fehlerhaft interpretiert wird. |

9.3.8 Persistenz

Das Verwalten von persistentem Speicher ist ein anderes Problem als das Verwalten von flüchtigen Recheninstanzen. Das Persistenz-Subsystem von Kubernetes stellt daher eine API für Benutzer und Administratoren bereit, die Details zur Bereitstellung von Speicher von der Art seiner Verwendung abstrahiert. Dies erfolgt mittels zweier API-Ressourcen namens *Persistent Volume* und *Persistent Volume Claim*.

- Ein **Persistent Volume (PV)** repräsentiert persistenten Speicher (Festplatte) im Cluster, der von einem Administrator bereitgestellt oder mithilfe von Speicherklassen dynamisch erzeugt wurde. Es ist wie ein Knoten einer normalen statischen Ressource des Clusters. *PVs* haben einen Lebenszyklus, der von einem Pod unabhängig ist und die Details der Implementierung des Speichers erfasst, sei es NFS, iSCSI oder ein cloud-provider-spezifisches Speichersystem.
- Ein **Persistent Volume Claim (PVC)** ist eine Anforderung von persistentem Speicher (Festplatte) durch einen Benutzer. Es ist analog einem Pod, der CPU- und Memory-Ressourcen eines Knotens benötigt. *PVCs* erzeugen PV-Ressourcen. Mittels *PVCs* können bestimmte Quality of Service-, Größen- und Zugriffsmodi von persistentem Speicher angefordert werden, z. B. können sie *ReadWriteOnce*, *ReadOnlyMany* oder *ReadWriteMany* bereitgestellt werden.

Nutzer können persistenten Speicher mittels *PVCs* anfordern. Hierzu muss eine sogenannte **Storage-Klasse** angegeben werden, aus der der persistente Speicher bedient werden soll. Auf die Angabe einer expliziten Storage-Klasse kann verzichtet werden, wenn es eine Default Storage-Klasse im Cluster gibt. Pods greifen auf den so angeforderten persistenten Speicher zu, indem sie den *PVC* als Volume im Pod mounten. *PVCs* müssen sich hierzu im selben Namespace wie der Pod befinden. Ein der Storage-Klasse zugeordneter **Volume Provisioner** im Cluster erzeugt anhand des *PVC* im Namespace des Pods ein *Persistent Volume*. Das *Persistent Volume* wird dann auf dem Host und im Pod bereitgestellt (siehe Bild 9.16 und Listing 9.18).

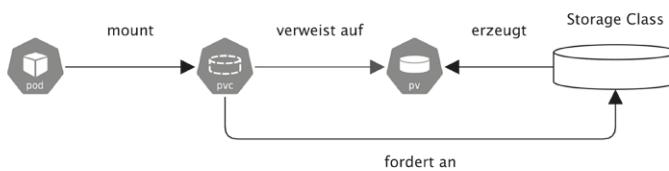


Bild 9.16

Bereitstellung persistenten Speichers

Listing 9.18 PVC-basiertes Mounten von persistentem Speicher

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata: { name: myclaim }
spec:
  accessModes:[ "ReadWriteOnce" ]
  volumeMode: Filesystem
  resources: { requests: { storage: 8Gi } }

---
apiVersion: v1
kind: Pod
metadata: { name: mypod }
  
```

```

spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts: [{ mountPath: "/var/www/html", name: mypd }]
  volumes:
    - name: mypd
      persistentVolumeClaim: { claimName: myclaim }

```

9.3.9 Isolation von Workloads

In Kubernetes lassen sich sogenannte *Namespaces* wie folgt zur Isolation nutzen. *Namespaces* können dabei sowohl

- zur Isolation von Nutzern in Mehrbenutzer-Szenarien (siehe Abschnitt 9.3.9.1),
- zur Isolation von Workloads (siehe Abschnitt 9.3.9.2)
- als auch zur Isolation des Netzwerkverkehrs (siehe Abschnitt 9.3.9.3)

genutzt werden und sind damit u. a. ein wirkungsvolles Mittel für Mehrbenutzer-Szenarien (Multi-Tenancy).

9.3.9.1 Namespaces und Role-based Access Model (Multi-Tenancy)

Jeder angelegte *Namespace* erhält automatisch einen Default-Service-Account mit einem generierten Access-Token zur Authentifizierung. Es können pro Namespace beliebig weitere Service-Accounts angelegt werden, die im Rahmen eines Systems rollenbasierter Zugriffsregeln zur Zuweisung von Rechten genutzt werden können. Mittels Service-Accounts und Zugriffsregeln (Role-based Access Model, RBAC) können innerhalb eines Namespace komplexe Rechtesysteme realisiert werden. Dabei werden zwei Arten von Rollen unterschieden:

- **Roles** weisen Rechte in einem *Namespace* zu (für den diese Rolle erzeugt wurde).
- **Cluster Roles** weisen Rechte namespace-übergreifend auf Cluster-Ebene zu. Das Anlegen von *Cluster Roles* erfolgt analog zu Roles.

Das Zusammenspiel zwischen *Role*, *Service Account*, *Role Binding* und *Secrets* (Token) zeigt Bild 9.17.

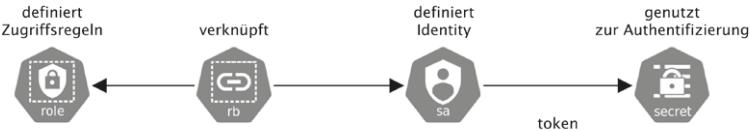


Bild 9.17 Rollenbasiertes Zugriffsmodell

Das Listing 9.19 zeigt exemplarisch, wie sich Leserechte und volle Zugriffsrechte zuweisen lassen, die etwa mittels des `kubectl create rolebinding`-Commands einem Service-Account zugewiesen werden können. Das Rollenmodell basiert in Kubernetes auf Ressourcen, die in der Cluster-API nach dem REST-Prinzip definiert und in Form von Gruppen (`apiGroups`) gegliedert sind. Eine Rolle fasst dabei diverse Aktionen auf Ressourcen ("get", "list", "watch", "create", "update", "patch", "delete") innerhalb einer API-Gruppe zusammen.

Listing 9.19 Definition von Rollen mittels des RBAC-Modells

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-read-role
rules:
- apiGroups: ["pods"]
  resources: ["*"]
  verbs: ["get", "watch", "list"] # <= Leserechte
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: rw-role
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: [*] # <= Alle Rechte

```

Demzufolge ist das Rollenmodell von Kubernetes einerseits sehr feingranular und wird damit schnell komplex und unübersichtlich. So müssen beispielsweise die üblichen Schreibrechte erst durch Zusammenfassung von "create"-, "update"-, "patch"- und "delete"-Aktionen sowie Leserechte durch Zusammenfassen von "get"-, "list"- und "watch"-Aktionen gebildet werden. Andererseits lassen sich damit sehr detaillierte und feingranulare Rollenmodelle erzeugen, die es ermöglichen, typische Schreibrechte in Erzeugungs-, Aktualisierungs- und Löschrechte aufzugliedern.

9.3.9.2 Quotas und Limit Ranges

Während sich mit dem RBAC-Modell also Rechtesysteme in Namespaces definieren lassen, lassen sich auch pro Namespaces Ressourcenkontingente festlegen, um zu vermeiden, dass insbesondere in Mehrbenutzerszenarien ein Namespace den Ressourcenbedarf anderer Namespaces dominiert. Mit **Resource Quotas** lässt sich daher der Ressourcenverbrauch pro Namespace beschränken. Um zu vermeiden, dass dabei einzelne Pods oder Container alle verfügbaren Ressourcen eines Namespace monopolisieren, kann zusätzlich die Ressourcenzuweisung pro Container limitiert bzw. mit Default-Werten belegt werden (**Limit Range**).

Listing 9.20 legt für einen Namespace fest, dass alle Container aller Pods eines Namespaces zusammen maximal zwei CPUs und 2 Gigabyte des Clusters erhalten. Jeder Container muss dabei mindestens 0.1 CPUs und darf maximal eine CPU verbrauchen. Wird keine Angabe gemacht, wird einem Container als Default 0.2 CPUs zugewiesen.

Listing 9.20 Festlegen von Quotas

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources

```

```

spec:
  hard:
    limits.cpu: "2"
    limits.memory: 2Gi
---
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limits
spec:
  limits:
  - type: Container
    max: { cpu: "1000m" }
    min: { cpu: "100m" }
    default: { cpu: "200m" }

```

Tabelle 9.3 zeigt einen (unvollständigen) Überblick, welche Ressourcen auf diese Art Ressourcenkontingenzen innerhalb eines *Namespaces* unterworfen werden können.

Tabelle 9.3 Auswahl definierbarer Ressourcenkontingente

| Ressource | Art | Erläuterung |
|------------------------|---------|---|
| cpu | Compute | Maximum der Summe aller CPU-Requests. |
| memory | Compute | Maximum der Summe aller Memory-Requests. |
| storage | Storage | Maximum der Summe aller Storage-Requests. |
| persistentvolumeclaims | Objekte | Gesamtzahl aller <i>Persistent Volume Claims</i> , die im Namespace zeitgleich existieren können. |
| configmaps | Objekte | Gesamtzahl aller <i>Config Maps</i> , die im Namespace zeitgleich existieren können. |
| secrets | Objekte | Gesamtzahl aller <i>Secrets</i> , die im Namespace zeitgleich existieren können. |
| pods | Objekte | Gesamtzahl aller <i>Pods</i> , die im Namespace zeitgleich existieren können. |
| jobs | Objekte | Gesamtzahl aller <i>Jobs</i> , die im Namespace zeitgleich existieren können. |
| services | Objekte | Gesamtzahl aller <i>Services</i> , die im Namespace zeitgleich existieren können. |

9.3.9.3 Network Policies

Mittels des RBAC-Modells lassen sich Rechtesysteme insbesondere für Multi-Tenancy-Szenarien definieren und somit vor allem Nutzer eines Clusters voneinander isolieren. Mittels Ressourcenkontingenzen lässt sich der Ressourcenbedarf pro Namespace beschränken. Auf diese Weise kann eine Monopolisierung von Ressourcen durch einen oder wenige Namespaces vermieden werden. Dennoch können noch beliebige Workloads innerhalb eines Clusters miteinander kommunizieren. Dazu müssen Workloads letztlich nur vom Cluster schematisch

vergebene DNS-Namen voneinander kennen. Um diese Kommunikationsfähigkeit zwischen oder sogar innerhalb von Namespaces voneinander isolieren zu können, ist die Steuerung des Verkehrsflusses durch den Cluster erforderlich.

Kubernetes sieht hierzu **Network Policies** als anwendungszentriertes Konstrukt vor, mit dem sich festlegen lässt, wie ein Pod mit verschiedenen Netzwerk-„Entitäten“ kommunizieren darf. *Network Policies* werden durch ein Netzwerk-Plug-in implementiert. Um „Network Policies“ nutzen zu können, muss der Cluster eine geeignete Overlay-Netzwerklösung verwenden, die *Network Policies* unterstützt. Das Erstellen von *Network Policy*-Ressourcen ohne einen Controller, der sie implementiert, ist zwar möglich, hat aber keine Auswirkungen. Die Entitäten, mit denen ein Pod kommunizieren kann, können dabei durch erlaubte Pods und Namespaces eingeschränkt werden. Ein Pod kann dabei allerdings niemals den Zugriff auf sich selbst blockieren. Erlaubte Pods und Namespaces lassen sich mittels Selektoren definieren, um festzulegen, welcher Datenverkehr zu und von dem/den Pod(s) erlaubt ist. Ergänzend können Richtlinien auch auf Basis von IP-Blöcken (CIDR-Bereiche) definiert werden.

Standardmäßig sind Pods nicht isoliert und akzeptieren daher Datenverkehr von jeder Quelle. Pods werden nur dann isoliert, wenn sich eine *Network Policy* auf sie bezieht. Sobald eine *Network Policy* in einem Namespace einen bestimmten Pod auswählt, lehnt dieser Pod alle Verbindungen ab, die nicht durch eine *Network Policy* zugelassen sind. Andere Pods im Namespace, die von keiner *Network Policy* ausgewählt werden, nehmen weiterhin den gesamten Datenverkehr an.

Network Policies sind additiv und stehen daher nicht im Widerspruch zueinander. Wenn eine oder mehrere Richtlinien einen Pod auswählen, ist der Pod auf das beschränkt, was durch die Vereinigung der Ein-/Ausgangsregeln dieser Richtlinien erlaubt ist. Die Reihenfolge der Auswertung hat also keinen Einfluss auf das Ergebnis der Richtlinie. Damit ein Netzwerkfluss zwischen zwei Pods zulässig ist, müssen sowohl die Egress-Richtlinie auf dem Quell-Pod als auch die Ingress-Richtlinie auf dem Ziel-Pod den Verkehr zulassen. Wenn entweder die Egress-Richtlinie auf dem Quell-Pod oder die Ingress-Richtlinie auf dem Ziel-Pod den Datenverkehr verweigert, wird der Datenverkehr blockiert.

Die in Listing 9.21 definierte Netzwerkrichtlinie erlaubt beispielsweise Eingehende, aber namespace-interne Kommunikation zwischen Pods des Namespaces und eingehende Kommunikation von Pods im „nginx-ingress“-Namespace. Die ausgehende Kommunikation (Egress) wird nicht eingeschränkt.

Listing 9.21 Definition von Netzwerkrichtlinien

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-and-internal
spec:
  podSelector: {} # <= Selektiert alle Pods
  policyTypes: [Ingress]
  ingress:
    - from:
        - { namespaceSelector: matchLabels: { name: nginx-ingress } }
        - { namespaceSelector: matchLabels: { name: mynamespace } }
```

■ 9.4 Zusammenfassung

Unter **Scheduling** versteht man die Allokation von Ressourcen zur Ausführung von Workloads. Um ein Problembewusstsein für die erwartbare Heterogenität zu schedulender Workloads zu entwickeln, sei die Studie von (Reiss u. a. 2012) empfohlen. Da Scheduling grundsätzlich ein NP-vollständiges Problem und daher keine optimale Lösung dafür bekannt ist, haben sich diverse Scheduling-Algorithmen in Container-Plattformen entwickelt.

- **Einfache Scheduling-Algorithmen** (siehe Abschnitt 9.1.2.1) optimieren das Scheduling von Workloads oft in genau einer Dimension (z. B. CPU-Auslastung) bzw. wenigen Dimensionen (CPU + Hauptspeicher). Oft wird dabei einer einfachen Fit-First-Strategie (Bin Pack, hohe Auslastung von Ressourcen) oder einer Round-Robin-Strategie (Spread, gleichmäßige Auslastung von Ressourcen) gefolgt.
- **Multidimensionale Scheduling-Algorithmen** (siehe Abschnitt 9.1.2.2) optimieren das Scheduling von Tasks dabei nach mehreren Kriterien. Das Konzept der Dominant Resource Fairness (DRF) strebt eine möglichst faire Aufteilung der Ressourcen über verschiedene Nutzer (Kunden, Projekte etc.) an (Ghodsi u. a. 2011).
- **Kapazitätsbasierte Scheduling-Algorithmen** (siehe Abschnitt 9.1.2.3) werden hingegen meist für jobbasierte Systeme verwendet, z. B. um intensive Rechenprozesse mit großen Datenmengen (Big Data) durchzuführen. Dabei wird Kapazitätszusage in Ressourcenanteilen vom Cluster definiert (Vavilapalli u. a. 2013).

Diese Scheduling-Algorithmen werden in unterschiedlichen Scheduler-Architekturen eingesetzt.

- **Monolithische Scheduler** (siehe Abschnitt 9.1.3.1) bestehen aus einem einzigen Scheduling-Agenten. Ein monolithischer Scheduler wendet in der Regel eine einzige Scheduling-Algorithmus-Implementierung für alle eingehenden Workloads an und eignet sich daher eher für homogene Workloads (Verma u. a. 2015).
- **Zweistufige Scheduler** (siehe Abschnitt 9.1.3.2) bestehen hingegen aus einem zentralen Ressourcen-Scheduler und weiteren Workload-Schedulern für spezifische Workload-Arten. Dabei wird die Ressourcenzuweisung für jeden Workload-Scheduler dynamisch angepasst (Hindman u. a. 2011).
- Bei **Shared-State-Schedulern** (siehe Abschnitt 9.1.3.3) gibt es ausschließlich workload-spezifische Scheduler, die kontinuierlich den aktuellen Zustand des Clusters untereinander synchronisieren. Ressourcenüberbuchungen werden als zugelassen und Schedulingkonflikte gegebenenfalls nachträglich aufgelöst (Schwarzkopf u. a. 2013).

Während sich das Scheduling also damit befasst, Workloads an Ressourcen einer bestehenden Infrastruktur für deren Ausführung zuzuweisen, versteht man unter **Orchestrierung** (siehe Abschnitt 9.2) vor allem die automatisierte Konfiguration, Verwaltung und Koordinierung von Anwendungen und Services. Orchestrierung ist somit keine einmalige (oder seltene) Aktivität wie das Scheduling, sondern eine dynamische, kontinuierliche überwachende Aktivität mit dem Ziel, die Sicherstellung einer dauerhaften und verlässlichen Workload-Ausführung, die durch externe Einflussfaktoren möglicherweise gestört werden kann.

- Orchestrierung beruht dabei auf der Formulierung von gewünschten Betriebszuständen und Abhängigkeiten zwischen den Betriebskomponenten einer Anwendung mittels sogenannter **Blueprints**, die von einem Orchestrator in Steuerungsaktivitäten umgesetzt werden.
- Diese Steuerungsaktivitäten werden dabei durch einen **Regelkreis** ausgelöst (siehe Abschnitt 9.2.2), der kontinuierlich den **Desired State** mit dem **Current State** von Betriebskomponenten im Sinne einer kontinuierlichen Feedback-Schleife vergleicht. Regelkreise sind dabei erstaunlich robust gegen Änderungen des Desired und Current State. Existiert eine Abweichung, wird ein entsprechender Steuerungsbefehl gegeben mit dem Ziel, die Abweichung zu reduzieren.

Die am weitesten verbreitete so arbeitende Orchestrierungsplattform ist Kubernetes (siehe Abschnitt 9.3), das unter anderem auf den jahrzehntelangen Erfahrungen von Google beim Betrieb von Container-Management-Plattformen beruht. Der Leser sei hier explizit auf die Arbeiten von (Burns u. a. 2016), (Verma u. a. 2015) und (Schwarzkopf u. a. 2013) verwiesen, die viele konzeptionelle Überlegungen zur Architektur und zu erforderlichem Featureumfang von Orchestrierungsplattformen begründen. Wie wir am Beispiel von Kubernetes gesehen haben, basieren diese Lösungen zumeist auf

- einer horizontal skalierbaren und **geclusterten Plattformarchitektur** (siehe Abschnitt 9.3.1).
- Anwendungen bestehen aus mehreren Workloads, deren Zusammenwirken und Ressourcenfordernisse in **Blueprints** definiert werden (siehe Abschnitt 9.3.2).
- Als schedulbare **Workload-Kategorien** (siehe Abschnitt 9.3.3) sind zumindest kontinuierlich laufende Services (*Deployments*) und einmalig bzw. periodisch angestoßene *Jobs* vorgesehen. Kubernetes sieht zudem noch weitere Workloads vor, die zumeist für Hintergrundaufgaben auf jedem Knoten des Clusters genau einmal ausgeführt werden (*Daemon-Sets*), und Anwendungen, die stabile und eindeutige Netzwerkbezeichner oder eine geordnete Bereitstellung und Skalierung erfordern (*Stateful-Sets*).
- Die Platzierung von Workloads innerhalb von Clustern lässt sich mit **Scheduling Constraints** beeinflussen (siehe Abschnitt 9.3.4).
- Horizontal skalierbare Workloads lassen sich ferner einer **automatischen Skalierung** unterwerfen, die es ermöglicht, Lasten elastisch folgen zu können (siehe Abschnitt 9.3.5).
- Damit service-basierte Komponenten Cloud-nativer Anwendungen miteinander interagieren können, ist es erforderlich, dass diese unter **DNS-Namen** bekannt gemacht werden können (siehe Abschnitt 9.3.6). Dies kann im Falle von Kubernetes cluster-intern mittels *Services* und cluster-extern im Falle von Webservices mittels *Ingress*-Ressourcen realisiert werden.
- Zur Realisierung eines **Self-Healing-Betriebs** überwachen diese Plattformen normalerweise die „Gesundheit“ von Workloads im Rahmen eines Health Checkings (siehe Abschnitt 9.3.7). Sind Workloads nicht mehr erreichbar oder reagieren nicht mehr wie spezifiziert, werden diese automatisiert gestoppt und neu gestartet.
- Für **statusbehaftete Komponenten** Cloud-nativer Anwendungen ermöglicht es Kubernetes, logische **Volumes** über Storage Provisioner anzufordern und diese in Workloads zu mounten (siehe Abschnitt 9.3.8).
- Werden Orchestrierungsplattformen in **Mehrbenutzerszenarien (Multi-Tenancy)** eingesetzt, so müssen Workloads voneinander **isoliert** werden können (siehe Abschnitt 9.3.9).

Hierbei kann u. a. die Sichtbarkeit mittels Namensräumen (*Namespaces*) eingeschränkt werden. Namensräume können ferner Ressourcenkontingente zugewiesen werden (*Quotas*). Mittels Netzwerkrichtlinien (*Network Policies*) kann auch zulässiger Netzwerkeverkehr innerhalb und zwischen Namensräumen eingeschränkt werden.

Ähnlich wie zu Docker (siehe Kapitel 8) gibt es zu Kubernetes eine äußerst umfangreiche Auswahl von Literatur. Der Abschnitt 9.3 sollte hier nur als sehr zusammenfassender Überblick verstanden werden. Eine gute Wahl für einen vertieften Einstieg ist aber sicher (Luksa 2018), der einen umfassenden Überblick gibt. Wer sich insbesondere mit der Verknüpfung von Deployment-Pipelines und Kubernetes im Kontext von DevOps (siehe auch Kapitel 3) befassen möchte, sei ergänzend auf (Aroundel und Domingus 2019) verwiesen.

Ergänzende Materialien

Auf der Website zum Buch finden sich zu diesem Kapitel ferner folgende Ergänzungsmaterialien:

- Praktische Übungen (Labs) zum Thema Container-Orchestrierung mit Kubernetes, aber auch Docker Swarm (u. a. Anbindung an Deployment-Pipelines, horizontale Skalierungstechniken, Self-Healing-Fähigkeiten und mehr).
- Übersichten inklusive Links zu weiteren Orchestrierungsplattformen
- Übersichten inklusive Links zu verschiedenen Kubernetes-Distributionen
- Übersichten inklusive Links zu hilfreichen Tools im täglichen Umgang mit Kubernetes
- Übersichten inklusive Links zu Storage- und Netzwerk-Lösungen für Kubernetes
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer (Themen: Platform as a Service (PaaS) , Container-Orchestrierung, Scheduling, Orchestrierung mittels Blueprints, Kubernetes)



<https://bit.ly/3DaAqj1>

10

Function as a Service

„No server is easier to manage than no server.“

Werner Vogels, CTO und Vicepresident von Amazon Web Services

Function as a Service (FaaS) ist eine weitere Option für die Bereitstellung von Anwendungen in der Cloud. FaaS vereinfacht die Bereitstellung von Anwendungen in der Cloud, die sich gut in viele feingranulare Funktionen gliedern lassen. Beim sogenannten Serverless Computing wird Geschäftslogik in Form von Funktionen auf einer FaaS-Plattform bereitgestellt und betrieben. Diese Plattform führt die Funktionen bei Bedarf aus, ohne dass aus Kundensicht hierfür Infrastruktur provisioniert oder gewartet werden muss. Der infrastrukturelle Ops-Aspekt, der IaaS (siehe Kapitel 7) deutlich prägt und sowohl bei PaaS (siehe Kapitel 8) und der Container-Orchestrierung/CaaS (siehe Kapitel 9) nicht gänzlich wegabstrahiert wird, entfällt bei FaaS aus Dev-Perspektive vollkommen. Im Zentrum steht die Funktion und weniger die Ausführungsumgebung der Funktion.

FaaS-Plattformen ermöglichen eine ereignis- bzw. request-basierte elastische Ressourcenzuweisung an auszuführende Funktionen. Muss ein System 100 gleichzeitige Requests bearbeiten, werden diese Requests an 100 (oder mehr) Kopien einer Funktion geleitet. Wenn die Nachfrage auf zwei gleichzeitige Anfragen sinkt, werden die nicht benötigten Funktionsinstanzen terminiert. Erfolgen über einen längeren Zeitraum gar keine Anfragen, wird die Anzahl an Funktionsinstanzen auf null herunterskaliert. Geht ein Request ein, wird notfalls eine Funktionsinstanz neu gestartet, falls diese zum Zeitpunkt des Request-Eingangs nicht laufen sollte (siehe auch Bild 10.1). Dadurch ist die Anwendung dennoch erreichbar, obwohl eine Anwendung nicht kontinuierlich durch laufende Funktionsinstanzen hinterlegt sein mag. Dies wird als **Scale-to-Zero**-Fähigkeit bezeichnet.

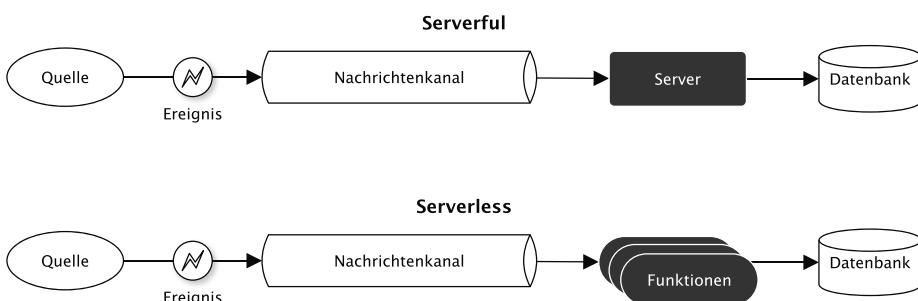


Bild 10.1 Serverful und Serverless

Container-Orchestrationsplattformen, wie beispielsweise Kubernetes, skalieren hingegen Workloads nie bis auf null (siehe auch Abschnitt 9.3.5), da dies als ein „unhealthy“ Workload-Zustand interpretiert werden würde. Auch wenn man mit Containern gut unter die 1 vCPU-Schwelle bei der Ressourcenzuteilung kommen kann, ist bei einem Service mindestens immer eine Ausführungsinstanz für kontinuierlich verfügbare Services erforderlich. Oberhalb von dieser einen Ausführungsinstanz lassen sich z. B. mittels Autoskalierung Lastkurven eng folgen. Aber letztlich muss jederzeit eine Ausführungsinstanz laufen, egal wie selten ein Request auch eingehen mag. Andernfalls ist ein Service nicht erreichbar. Diese letzte Ausführungsinstanz ist somit immer ein Constant Workload gemäß Abschnitt 2.3 und stellt damit den unwirtschaftlichsten Teil dar. Mittels Scale-to-Zero lässt sich auch dieser letzte (unwirtschaftlichste) Anteil komplett dem Pay-as-you-go-Kostenmodell unterwerfen, da auf konzeptionelle Always-on-Komponenten nun vollends verzichtet werden kann (bzw. dessen Betrieb dem Service-Provider überlassen wird).

Kunden des Service-Modells FaaS zahlen somit tatsächlich nur für die Ressourcen, die Funktionen auch tatsächlich nutzen und nicht anteilig für Ressourcen die Always-on-Charakter haben und somit gemäß der Cloud-Ökonomie als besonders teuer einzustufen sind. Service-Provider berücksichtigen natürlich ihre Kosten für den Betrieb einer FaaS-Plattform in ihrem FaaS-Kostenmodell, dennoch haben Studien von (Villamizar u. a. 2017) gezeigt, dass sich immer noch substanziale Betriebsersparnisse von bis zu 75 % aus Kundensicht ergeben können.

Dieses manchmal als Serverless Computing bezeichnete Cloud-Computing-Service-Modell ermöglicht Cloud-Providern, Computing-Ressourcen also nach Bedarf zuzuweisen, ohne dass sich Nutzer von Serverless Computing-Plattformen um die Dimensionierung der erforderlichen Infrastruktur kümmern müssen. Beim Serverless Computing werden die Ressourcen somit nicht dauerhaft im flüchtigen Speicher gehalten, sondern Processing-Ressourcen werden request-basiert bereitgestellt. Wenn eine Anwendung nicht in Gebrauch ist, werden einer Anwendung somit auch keine Rechenressourcen zugewiesen. Die Preisgestaltung basiert somit auf der tatsächlichen Menge der von einer Anwendung genutzten Ressourcen (und nicht der durch den Nutzer angeforderten Ressourcen). Serverless Computing erfordert natürlich weiterhin physische oder virtuelle Infrastruktur und Computing-Ressourcen. Entwickler von serverlosen Anwendungen müssen sich jedoch nicht mit der Kapazitätsplanung, Konfiguration, Verwaltung, Wartung, dem Betrieb oder der Skalierung von Containern, VMs oder physischen Servern beschäftigen. Dies liegt in Verantwortung des Serverless Computing-Providers.

Serverless Computing kann somit den Prozess der Bereitstellung von Code in die Produktion vereinfachen. Serverless Code kann in Verbindung mit anderen Architekturstilen, wie beispielsweise Microservices oder service-orientierten Architekturen, verwendet werden. Alternativ können Anwendungen allerdings auch so geschrieben werden, dass sie rein serverlos sind. Dies sollte allerdings nicht mit dezentralen Peer-to-Peer-(P2P-)Computing-Ansätzen verwechselt werden, die auch manchmal als serverlos bezeichnet werden, da sie konzeptionell von keinem logischen Zentralserver abhängen.

■ 10.1 FaaS-Plattformen

Serverless-Anbieter bieten Laufzeitumgebungen an, die oft auch als Function-as-a-Service-(FaaS-)Plattformen bezeichnet werden. Diese führen Anwendungslogik aus, speichern aber keine Daten, sind also zustandslos konzipiert. Der Einsatz solcher Plattformen bringt folgende Vorteile.

- **Kosten:** Serverless kann kosteneffizienter sein als das Mieten oder Kaufen einer festen Anzahl von Servern, da dies in der Regel mit erheblichen Perioden der Unterauslastung oder Leerlaufzeiten verbunden ist. Es kann auch kosteneffizienter sein als die Nutzung von Autoskalierungsfunktionen von IaaS-Infrastrukturen oder Container-Orchestrierungsplattformen, da nur die für die Ausführung des Codes zugewiesene Zeit und der zugewiesene Speicher in Rechnung gestellt werden, ohne damit verbundene Gebühren für Leerlaufzeiten. Studien konnten signifikante Kostenreduktionen nachweisen (Villamizar u. a. 2017), doch wie immer hängt dies stark vom Anwendungsfall ab (siehe Abschnitt 2.3). Insbesondere Anwendungsfälle, die durch gar keine Grundlast über längere Zeiträume charakterisiert sind, profitieren von diesem Modell stärker als Anwendungsfälle, die eine vielleicht niedrige, aber dennoch kontinuierlich vorhandene Grundlast haben.
- **Elastizität und Skalierbarkeit:** Eine serverlose Architektur erübrigt es für Entwickler und Betreiber, Richtlinien oder Systeme zur automatischen Skalierung einzurichten und abzustimmen zu müssen, da der Cloud-Anbieter für die Skalierung der Kapazität verantwortlich ist. Da Cloud-native Systeme sowohl nach unten als auch nach oben skalieren, werden diese Systeme als elastisch bezeichnet.
- **Produktivität:** Bei FaaS sind die Codeeinheiten einfache ereignisgesteuerte Funktionen. Das bedeutet, dass sich der Programmierer in der Regel nicht um Multithreading oder die direkte Bearbeitung von HTTP-Anfragen in seinem Code kümmern muss, was die Aufgabe der Back-End-Softwareentwicklung vereinfacht und die Entwicklerproduktivität in vielen Fällen erhöhen kann.

Allerdings sind auch Nachteile mit dieser Form des Utility Computings hinsichtlich Zustand, limitierten Ausführungs dauern und mit Kaltstarts zusammenhängenden Latenzen verbunden.

- **Zustandslosigkeit:** Jeder Zustand einer FaaS-Funktion, der persistent sein soll, muss außerhalb der FaaS-Funktionsinstanz externalisiert werden. Für FaaS-Funktionen, die eine rein funktionale Transformation ihrer Eingabe zu ihrer Ausgabe darstellen, ist dies kein Problem. Auch das Konzept der „Twelve-Factor-App“ (siehe Abschnitt 8.5) hat genau die gleiche Einschränkung. Zustandsorientierte Funktionen müssen daher einen Zustand in einer Datenbank, einen anwendungsübergreifenden Cache (wie Redis) oder einen Netzwerkknoten-/Objektspeicher (wie S3) speichern, um einen Zustand über Requests hinweg zur Verfügung zu stellen. Diese Zustandslosigkeit wird von vielen Entwicklern als hinderlich empfunden, die in „klassischen“ Umgebungen ihre Entwicklungserfahrungen gesammelt haben.
- **Ausführungs dauern:** FaaS-Funktionen sind typischerweise darin begrenzt, wie lange die Verarbeitung eines Requests dauern darf. Typische Grenzen bei Managed Service Providern sind mehrere Minuten (z. B. fünf bis zehn Minuten). Bestimmte Klassen von langlebigen Tasks eignen sich also nicht für FaaS-Funktionen. Möglicherweise ist es er-

forderlich, mehrere verschiedene koordinierte FaaS-Funktionen zu erstellen, während man dies in einer herkömmlichen Umgebung als eine einzige langlebige Aufgabe lösen würde, die sowohl die Koordination als auch die Ausführung übernimmt.

- **Kaltstartbedingte Latenzen:** Es dauert einige Zeit, bis eine FaaS-Plattform eine Instanz einer Funktion vor jedem Ereignis initialisiert. Diese Startlatenz kann selbst für eine bestimmte Funktion erheblich variieren und ist abhängig von einer Vielzahl von Faktoren. Latenzen können von wenigen Millisekunden bis zu mehreren Sekunden reichen. Die Initialisierung einer Funktion ist entweder ein „Warmstart“ – die Wiederverwendung einer Instanz und ihres Host-Containers von einem vorherigen Ereignis – oder ein „Kaltstart“ – der die Erstellung einer neuen Container-Instanz beinhaltet, die den Host-Prozess der Funktion startet.

Ergänzend kommt hinzu, dass das „Ökosystem“, in dem FaaS-Funktionen ausgeführt werden müssen, noch durch weitere Limitierungen charakterisiert ist, wie beispielsweise durch Storage-Services, die für einen derartigen feingranularen Betrieb von Services eigentlich nicht ausgelegt sind. Der Zugriff auf einen Object-Storage erhöht etwa substanziell die Latenzen einer Funktion. Insgesamt fehlen noch autoskalierende Storage-Services mit niedrigen Latenzen. Auch die Koordination feingranularer Kontrollflüsse ist problematisch. Aufgrund des sogenannten **Double-Spending-Problems** sollten sich Funktionen z. B. nicht synchron gegenseitig aufrufen, da dies zu mehrfachen Kosten führen kann. Auf das Double-Spending-Problem werden wir noch in Kapitel 12 und insbesondere in Abschnitt 12.6 eingehen, da sich hieraus eine spezifische Interpretation der Microservice-Architektur gebildet hat, die manchmal als Serverless-Architektur bezeichnet wird.

Infolgedessen werden solche FaaS-Plattformen überwiegend in den in Bild 10.2 gezeigten Anwendungsfällen eingesetzt (Jonas u. a. 2019), die alle durch eine gewisse Einfachheit charakterisiert sind.

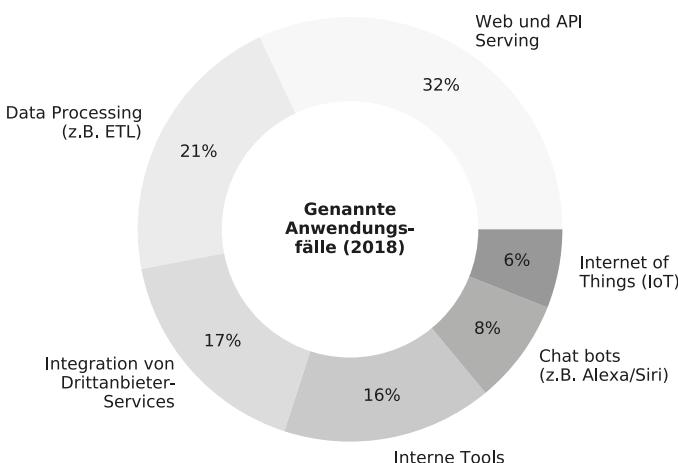


Bild 10.2
FaaS-Anwendungsfälle

10.1.1 Das FaaS-Programmiermodell

Das FaaS-Programmiermodell ist event-driven und zustandslos. Ereignisse können aus unterschiedlichsten Quellen stammen, z. B. aus Datastores, Message-Queues, Mobile-Apps und Webanwendungen, Sensoren, Chat Bots oder zeitgesteuerten Tasks (CRON). Es wird dabei ein polyglotter Sprachansatz durch eine Vielzahl von Ausführungsumgebungen unterstützt, u. a. meist Go, Java, JS, PHP, Python, Ruby, Swift und .NET. Mittels Container-Images lassen sich häufig weitere problemspezifische Sprachen ergänzen.

Das grundsätzliche Prinzip wird am Beispiel der Open-Source-FaaS-Plattform OpenWhisk (Sciabarrà 2019) erläutert, dem konzeptionell aber alle FaaS-Plattformen folgen (siehe Bild 10.3); wobei die Terminologie wie etwa *Ereignisquellen*, *Trigger*, *Regeln* und *Funktionen* durchaus vielfältig in den unterschiedlichen Realisierungen abgewandelt werden. Bis auf diese terminologischen Unterschiede folgen aber alle FaaS-Plattformen dem in Bild 10.3 gezeigten Prinzip.

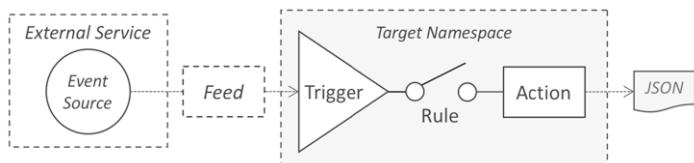


Bild 10.3
FaaS-Programmiermodell
(Quelle: OpenWhisk)

- **Ereignisquelle:** Hierbei handelt es sich um Services, die Ereignisse generieren. Dies können Object-Stores oder Datenbanken, Message-Queues, Mobile-Apps und Webanwendungen, Sensoren, Chat Bots, zeitgesteuerten Tasks und viele mehr sein. Welche Ereignisquellen eine FaaS-Plattform unterstützt, ist häufig produkt- oder provider-spezifisch. Immer weisen Ereignisse dabei häufig auf Datenänderungen hin oder beinhalten die zu verarbeitenden Daten selbst.
- **Trigger** sind benannte Kanäle für Klassen von Ereignissen, die von *Ereignisquellen* gesendet werden. Sie bilden ein durchaus heterogenes Umfeld externer Services auf ein einheitliches Event-System der FaaS-Plattform ab.
- **Regeln** werden verwendet, um einen Trigger einer **Funktion** zuzuordnen. Durch diese Zuordnung wird bei einem Auslöseereignis die zugeordnete Aktion aufgerufen. In der *Funktion* erfolgt die eigentliche Verarbeitung der Daten des Ereignisses. Sind die Daten nicht selbst im Ereignis enthalten, so wird die Funktion die erforderlichen Daten beispielsweise anhand eines im Ereignis codierten Uniform Resource Locators (URL) laden.
- **Funktionen** sind somit zustandslose und meist sehr kleine Programmlogiken, die auf einer FaaS-Plattform ausgeführt werden. *Funktionen* kapseln daher Anwendungslogik, die als Reaktion auf *Ereignisse* ausgeführt werden soll. *Funktionen* können zumeist manuell über eine REST-API der FaaS-Plattform, ein Command Line Interface (FaaS-CLI) oder über *Trigger* automatisiert aufgerufen werden.

10.1.2 Zu berücksichtigende Randbedingungen

Die Limitierung von Funktionen führen zu von vornherein zu berücksichtigenden Randbedingungen. Der Effekt des Einsatzes von FaaS kann überraschend sein, da er auch erhebliche architekturelle Auswirkungen haben kann, wie Abschnitt 12.6 noch zeigen wird. Es empfiehlt sich daher, einige Best Practices zu beherzigen:

- **Single Purpose-Funktionen:** Funktionen sollten konsequent dem Single-Responsibility-Prinzip folgen und nur eine einzige Verantwortung haben. Dies macht Funktionen und deren Wechselwirkungen einfacher zu verstehen, zu testen und zu debuggen.
- **Kein Funktions-Chaining:** Funktionen sollten lose gekoppelt sein und daher insbesondere nicht dem Anti-Pattern folgen, sich gegenseitig synchron aufzurufen. Funktionen sollten vielmehr Nachrichten in Message-Queues oder in Datastores pushen, um andere Funktionen zu triggern.
- **Leichtgewichtigkeit:** Jede Funktion sollte nur eine Aufgabe erfüllen und von einer minimalen Anzahl an externen Bibliotheken abhängen. Unnötiger Code macht Funktionen gegebenenfalls unnötig groß mit negativen Effekten auf die Startzeit. Man sollte z. B. vermeiden, die riesige PyTorch-Bibliothek zu verwenden, nur weil man eine einzige Funktion daraus benötigt.
- **Zustandslosigkeit:** Funktionen sollten keine Daten persistent speichern, die Funktionsaufrufe überdauern. Funktionen laufen normalerweise in isolierten Umgebungen und teilen nichts mit anderen Funktionsinstanzen (auch nicht mit Instanzen derselben Funktion).
- **Trennung von Einstiegspunkt und Funktionslogik:** Funktionen haben einen Einstiegspunkt, der vom Funktions-Framework aufgerufen wird. Der framework-spezifische Kontext wird im Allgemeinen zusammen mit dem Aufrufkontext an diesen Einstiegspunkt übergeben. Wird die Funktion z. B. über ein HTTP-API-Gateway aufgerufen, enthält der Kontext HTTP-spezifische Details. Diese Einstiegspunktdetails sollten vom Rest des Codes getrennt werden. Dies verbessert die Verwaltbarkeit, Testbarkeit und Portabilität von Funktionen über FaaS-Plattformen hinweg.
- **Vermeidung von lang laufenden Funktionen:** Die meisten FaaS-Angebote haben eine Obergrenze für die Ausführungszeit pro Funktion. Langfristige Funktionen können daher zu Problemen wie längeren Ladezeiten und Zeitüberschreitungen führen. Wenn möglich, sollten daher große Funktionen in kleinere Funktionen refaktoriert werden, die lose gekoppelt über Ereignisse zusammenarbeiten.
- **Funktionsübergreifende Kommunikation mittels Streaming:** Anstatt Informationen untereinander weiterzugeben (enge Kopplung), sollten Funktionen eine Warteschlange verwenden (lose Kopplung), in die die Nachrichten gesendet werden. Andere Funktionen können basierend auf den Ereignissen (wie beispielsweise Element hinzugefügt, entfernt oder aktualisiert) in dieser Warteschlange ausgelöst und ausgeführt werden. Den Themen loser und enger Kopplung wird sich noch im Detail das Kapitel 12 und dem Streaming insbesondere der Abschnitt 12.2.4 und der Abschnitt 12.4.2 widmen.

10.1.3 Veranschaulichung des FaaS-Programmiermodells

Es gibt eine Vielzahl an Managed Service oder selbst-hostbaren FaaS-Plattformen. Ein vollständiger Überblick kann daher nicht gegeben werden, und dieser wäre wegen der Dynamik in diesem Bereich auch schnell veraltet. Auf der Website zu diesem Buch werden aber einige dieser FaaS-Plattformen aufgeführt und kontinuierlich fortgeschrieben.

Das FaaS-Programmiermodell soll daher am Beispiel der Open-Source-FaaS-Plattform OpenWhisk im Sinne eines Typvertreters mittels der Programmiersprache Python veranschaulicht werden. Auch wenn FaaS-Plattformen (zumindest bislang) keinem OCI-ähnlichen Standard folgen, lassen sich die hier vermittelten Prinzipien durchaus auf andere FaaS-Plattformen wie etwa OpenFaas, FnProject oder kommerzielle Managed FaaS-Services wie AWS Lambda oder Google Cloud Functions übertragen.

Funktionen in OpenWhisk haben unabhängig von der Ereignisquelle immer das gleiche Format (siehe auch Listing 10.1).

- Funktionen empfangen Events in Form eines event-Parameters. Der Einstiegspunkt der Verarbeitung muss in einer Funktion namens `main()` liegen, kann von dort aber weitere Funktionen aufrufen (und durchaus komplexe Abhängigkeiten haben, wenn die Funktion als Aktionspaket bereitgestellt wird). Dieser Parameter enthält alle Eingabeparameter der Funktion in Form eines Dictionary.
- Die `main()` liefert eine Antwort in Form eines in JSON-konvertierbaren Dictionary als Antwort auf einen Request.

Listing 10.1 Eine einfache FaaS-Funktion (OpenWhisk, Python)

```
# File: greeting.py
from typing import Dict, Any

def main(event: Dict[str, Any]) -> Dict[str, Any]:
    name = event.get("name", "Gast")
    return {
        "message": f"Hello, { name }!"
    }
```

Diese Funktion erwartet einen Eingabeparameter mit dem Namen `name`. Wenn kein Wert für `name` angegeben wird, wird der Standardwert „Gast“ verwendet. Die Funktion gibt dann eine Antwort mit einer Begrüßungsnachricht zurück, die den Namen enthält.

Um diese Funktion in OpenWhisk zu deployen, muss sie als Aktion registriert werden, bspw. mittels der OpenWhisk-Befehlszeilenschnittstelle `wsk` (CLI):

```
> wsk action create greeting greeting.py --kind python:3.10
```

Dieser Befehl erstellt eine Aktion mit dem Namen `greeting` und verwendet den Funktionscode aus der Datei „`greeting.py`“. Der Parameter `--kind python:3.10` gibt an, dass die Aktion mit der Laufzeitumgebung Python 3.10 ausgeführt werden soll.

Die so eingerichtete Aktion existiert nach einem Deploy isoliert in der Plattform und kann mittels **wsk** ausgeführt werden:

```
> wsk action invoke greeting --result --param name "John"

# Antwortrückgabe:
{
  "message": "Hallo, John!"}
```

Damit eine Aktion auf externe Ereignisse reagieren kann, müssen Trigger eingerichtet werden, die externe Ereignisse von Triggern mittels Regeln mit Aktionen verknüpfen. Mehrere unterschiedliche Trigger können so eine Aktion auslösen. Ein Trigger kann aber auch mehrere Aktionen auslösen.

| | |
|---|--|
| <pre>> wsk trigger create mytrigger \ --feed /whisk.system/web/httppost \ --param url "https://my.api.org/greet"</pre> | # Name des Triggers # Feed-Provider (HTTP-Trigger) # Öffentlich zugängliche URL |
|---|--|

Dieser Trigger kann dann mittels einer Regel mit der auszuführenden Aktion verknüpft werden.

```
> wsk rule create myrule mytrigger greeting
```

Sobald der Trigger eingerichtet ist, können HTTP-Anfragen an die URL gesendet werden, die dann über die Kette Trigger – Regel – Aktion die Funktion (in unserem Fall Listing 10.1) mit den entsprechenden Parametern ausführen.

OpenWhisk unterstützt viele verschiedene Arten weiterer Trigger, um Aktionen auszulösen. Dies sind insbesondere:

- **Zeitbasierte Trigger:** OpenWhisk bietet einen eingebauten Alarm-Feed (`/whisk.system/alarms/alarm`), der zeitbasierte Trigger ermöglicht. Man kann Cron-Ausdrücke verwenden, um Aktionen basierend auf einem bestimmten Zeitplan auszulösen, z. B. alle fünf Minuten, stündlich, täglich usw.
- **Feed-basierte Trigger:** OpenWhisk unterstützt eine Vielzahl von Feeds, die als Trigger dienen können. Einige Beispiele sind:
 - **HTTP-Feed** (`/whisk.system/web/httppost`): Erlaubt es, Aktionen durch eingehende HTTP-Anfragen auszulösen. Dies entspricht dem oben gezeigten Beispiel.
 - **CouchDB-Feed** (`/whisk.system/couchdb/changes`): Dieser Trigger ermöglicht das Auslösen von Aktionen basierend auf Änderungen in einer CouchDB-Datenbank. Dabei kann festgelegt werden, welche Arten von Änderungen (z. B. Insert, Update, Delete) den Trigger auslösen sollen.
 - **Kafka-Feed** (`/whisk.system/kafka/changes`): Dieser Trigger ermöglicht es, Aktionen basierend auf Nachrichten aus einem Apache Kafka-Cluster auszulösen. Der Trigger kann mit einem Kafka-Topic verknüpft werden und Aktionen auslösen, wenn neue Nachrichten in das Topic geschrieben werden.

Nachfolgendes Beispiel zeigt, wie man einen zeitbasierten Trigger einrichten kann, um bspw. Aktionen periodisch jeden Morgen um 08:00 Uhr auszulösen.

```
> wsk trigger create daily --feed /whisk.system/alarms/alarm --param cron "*0 8 * *"
> wsk rule create greatDaily daily greeting
```

Trigger, Regeln oder Aktionen lassen sich mittels der in OpenWhisk-Funktion CLI `wsk` sowohl auflisten als auch löschen:

```
# List
> wsk trigger list          # Auflisten aller Trigger
> wsk rule list            # Auflisten aller Regeln
> wsk action list          # Auflisten aller Aktionen

# Delete
> wsk trigger delete <trigger-name> # Löschen eines Triggers
> wsk rule delete <rule-name>       # Löschen einer Regel
> wsk action delete <action-name>  # Löschen einer Aktion (Funktion)
```

OpenWhisk ist eine leistungsstarke FaaS-Plattform, um Funktionen effizient und skalierbar auszuführen. Um OpenWhisk im Detail zu nutzen und auch komplexere Funktionen in Form von Aktionspaketen bereitstellen zu können, die durchaus komplexe Abhängigkeiten bspw. zu Deep-Learning-Bibliotheken wie TensorFlow oder PyTorch haben können, wird die offizielle OpenWhisk-Dokumentation oder -Literatur (Scialabà 2019) empfohlen. Die Dokumentation bietet umfassende Informationen zum Einsatz von OpenWhisk und behandelt Themen wie Funktionen, Trigger, Aktionspakete (und vieles mehr) deutlich detaillierter.

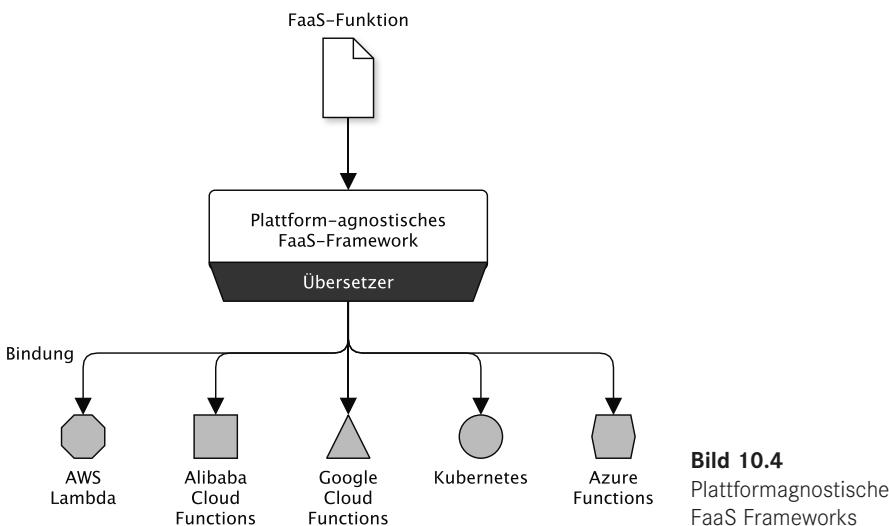
■ 10.2 Plattformagnostische FaaS-Frameworks

Wie bereits erwähnt, ist die FaaS-Plattform-Landschaft durch viele Produkte und Managed Service-Provider gekennzeichnet. Ein Konsolidierungsprozess wird hier voraussichtlich erst noch einsetzen. Funktionen, die auf der *Google Functions*-Plattform laufen, sind beispielsweise nicht 1:1 auf *AWS Lambda* übertragbar. Grundsätzlich herrscht hier eine ähnliche Situation vor wie in der Anfangsphase des Cloud Computings bei PaaS-Anbietern (siehe Abschnitt 8.1). Wesentliche Probleme bei PaaS waren und sind u. a. fehlende Standards. Dies betrifft insbesondere

- das Deployment-Format der zu hostenden Anwendungen
- und die PaaS-Runtime-Schnittstelle.

Dieses Problem ist auch bei FaaS zu einem gewissen Grad zu beobachten. Die Ausgangslage ist aber grundsätzlich besser. Während man mit PaaS die Standardisierung sehr heterogener Workloads abdecken muss, ist FaaS sehr stark auf die Ausführung zustandsloser Funktionen fokussiert. Dies ist eine wesentlich klarer abgrenzbare Domäne mit geringer Komplexität, die grundsätzlich gut standardisierbar ist. Wie immer haben jedoch kommerzielle Service-Provider der ersten Stunde daran kein Interesse, da man durch Standardisierung aufgebauten Kundenbindungspotenziale reduziert. Da sich aber das Deployment-Format bei FaaS gut durch Container standardisieren lässt (und dies im Wesentlichen auch von vielen FaaS-Plattformen mittlerweile so genutzt wird), bleibt nur noch die Standardisierung der Runtime-Schnittstelle. Hier kann man auf die De-facto Standardisierung über eine weit verbreitete Open-Source-Plattform wie etwa *OpenWhisk* setzen, zumal *OpenWhisk* auch als kommerzieller Managed FaaS-Service namens *Cloud Functions* von IBM angeboten wird. Die Herausforderer haben – anders als die etablierten Service-Provider – eher das Interesse zur Standardisierung, da Herausforderer die Kundenbindung der großen Hyperscaler (wie AWS, Azure und Google) reduzieren wollen. IBM hat zudem jahrzehntelange Erfahrung in der Zusammenarbeit mit Standardisierungsgremien. Die Standardisierung von FaaS-Services steht aber dennoch erst am Anfang. Die Ausgangslage ist aber eine bessere als bei PaaS.

Ein anderer Ansatz, die Portabilität von Funktionen zu steigern, ist die Nutzung plattform-agnostischer Frameworks. Agnostische Frameworks definieren Funktionen FaaS-plattform-unabhängig und „übersetzen“ diese agnostischen Funktionen für spezifische FaaS-Plattformen oder FaaS-Provider (siehe Bild 10.4).



Das umfangreichste Framework aus diesem Bereich ist vermutlich das *Serverless Framework* (Collins, Radkakrishnan und Fine 2015). Damit lassen sich Funktionen FaaS-plattform-agnostisch (jedoch natürlich *serverless framework*-spezifisch), wie in Listing 10.2 gezeigt, definieren. Somit geht man eine Abhängigkeit zu einem Framework, aber immerhin nicht mehr zu einer spezifischen FaaS-Plattform ein.

Listing 10.2 Agnostisch definierte FaaS-Funktion (handler.py)

```
import json, datetime

def endpoint(params):
    current_time = datetime.datetime.now().time()
    name = params.get("name", "stranger")
    return {
        "message": f"Hello { name }. The current time is { current_time }"
    }
```

Die Bindung an einen Provider erfolgt dann mittels einer in Listing 10.3 exemplarisch gezeigten Plattformbindung.

Listing 10.3 Bindung an einen FaaS-Provider

```
service: hello-service

provider:
    name: openwhisk          # Durch andere Provider ersetzbar
    runtime: python            # Durch andere Umgebungen ersetzbar

plugins: ["serverless-openwhisk"] # Durch weitere Plug-ins ergänzbar

functions:
    currentTime:
        handler: handler.endpoint
        events:[{ http: { path: "hello", method: "get" }}]
```

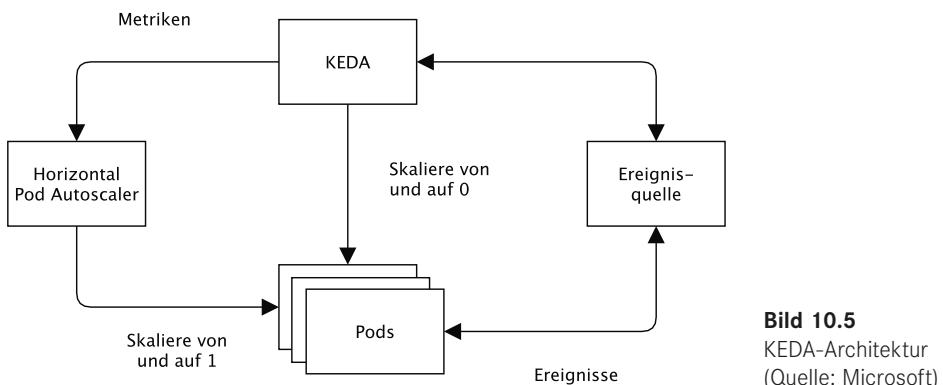
Im gezeigten Beispiel wird eine Funktion für *OpenWhisk* deployed. Die Plattformen sind aber austauschbar. Nach eigenen Angaben unterstützt *Serverless Framework* zum Zeitpunkt des Schreibens dieses Buchs unter anderem die Provider AWS Lambda, Azure Lambda, Google Cloud Functions, OpenWhisk und Kubeless sowie die Laufzeitumgebungen .NET, Go, Java, Node JS, Python, Ruby, Rust und Swift. Man muss aber auch sagen, dass die Unterstützung von AWS durchaus die umfangreichste ist, was einen höheren Grad der Portabilität suggeriert, als tatsächlich mit solchen Ansätzen aktuell erreicht werden kann. Auch hier können sich inhärente Abhängigkeiten zu Providern einschleichen, wenn man sich nicht bewusst auf die portablen FaaS-Plattform-Funktionalitäten beschränkt.

■ 10.3 Ereignisbasierte Autoskalierung

Kubernetes sieht mit dem Horizontal Pod Autoscaling-Konzept (siehe Abschnitt 9.3.5) eine ressourcenbasierte Autoskalierung von Pods vor. Häufig ist jedoch eine rein ressourcenbasierte Kennzahl wie beispielsweise 66 % Prozessorauslastung nicht hilfreich für Entscheidungen, ob nun hoch- und herunterskaliert werden soll. 66 % Prozessorauslastung kann viel oder ganz normal sein. Das hängt ganz vom Workload ab. Man stelle sich eine Warteschlange vor, in der zu zwei Dritteln der Zeit immer maximal ein Task enthalten ist, der allerdings sehr berechnungsintensiv ist. Dies würde zu einer Auslastung von ca. 66 % führen. Das Hinzuschalten eines weiteren Pods würde hier rein gar nichts bringen, da keine Parallelverarbeitung stattfinden kann. In solchen Fällen ist es besser, Skalierungsentscheidungen z. B. auf Basis einer Warteschlangenlänge vorzunehmen. Steigt die Warteschlangenlänge über z. B. drei Einträge, wird ein weiterer Pod hinzugeschaltet, ansonsten nicht. Sinkt sie unter drei Einträge, kann ein Pod abgeschaltet werden. Diese Prüfung und damit die Skalierungsanpassung erfolgen periodisch.

Bei der ereignisbasierten Skalierung wird also die Ressourcenauslastung, die in vielen Fällen nicht wirklich hilfreich für Skalierungsentscheidungen auf Pod-Ebene ist, nicht berücksichtigt, sondern man versucht, andere Größen zur Skalierung heranzuziehen.

KEDA (siehe Bild 10.5) ist ein ereignisgesteuerter Autoscaler für Kubernetes, der genau für diesen Einsatzzweck vorgesehen ist, und er beherrscht dabei auch Scale-to-Zero. Damit lässt sich das im vorherigen Abschnitt 10.1 erläuterte Prinzip ereignisgetriggerner Funktionen zumindest innerhalb von Kubernetes auch durch einen Autoskalierer nachbauen. KEDA wird im Übrigen innerhalb von Azure Functions eingesetzt und wurde im Rahmen dessen als Open-Source-Komponente von Microsoft veröffentlicht (Microsoft 2014).



Mit KEDA lässt sich die Skalierung von Workloads in Kubernetes basierend auf der Anzahl der zu verarbeitenden Ereignisse steuern. KEDA ist als leichtgewichtige Komponente konzipiert, die ergänzend zu jedem Kubernetes-Cluster hinzugefügt werden kann und den Horizontal Pod Autoscaler (siehe Abschnitt 9.3.5) ergänzt, um Workloads skalieren zu können, die einer ereignisgesteuerten Autoskalierung unterworfen werden sollen. KEDA übernimmt dabei zwei Funktionen innerhalb von Kubernetes:

- Ein **Agent** aktiviert und deaktiviert Kubernetes-Deployments, um bei keinen vorliegenden Ereignissen auf null skalieren zu können.
- KEDA fungiert ferner als **Metrik-Server** für Kubernetes, der dem Horizontal Pod Autoscaler umfangreiche Ereignisdaten wie beispielsweise Warteschlangenlängen zur Verfügung stellt, um die eigentliche Skalierung vornehmen zu können.

KEDA unterstützt dabei eine breite Palette von Skalierern, die sowohl erkennen können, ob ein Deployment aktiviert oder deaktiviert werden soll und Metriken vielfältiger Ereignisquellen auswerten kann (siehe Tabelle 10.1).

Tabelle 10.1 Überblick von KEDA-Ereignisquellen (nicht vollständig)

| Kategorie | Skalierer (Ereignisquellen) |
|--------------------------|---|
| Streaming | ActiveMQ, Kafka, Liiklus, NATS, RabbitMQ |
| NoSQL-Datenbanken | InfluxDB, MongoDB, Redis |
| SQL-Datenbanken | MSSQL, MySQL, PostgreSQL |
| Metrikdatenbanken | OpenStack Metric, Prometheus |
| Zeitgesteuert | CRON |
| Ressourcenbasiert | CPU, Memory |
| Managed AWS Services | CloudWatch, Kinesis, SQS |
| Managed Azure Services | Blob Storage, Event Hubs, Log Analytics, Monitor, Pipelines, Service Bus, Storage Queue |
| Managed Google Services | Cloud Pub/Sub |
| Weitere Managed Services | Huawei Cloudeye, IBM MQ |
| Sonstige | External, External Push |

Zudem sind weitere externe Skalierer ergänzbar, um auch sehr spezifische Skalierungsanforderungen abilden zu können. Mittels Customer Defined Resources (CRDs) können Ereignisquellen (Trigger) Kubernetes Workloads zugeordnet werden.

- **ScaledObjects** repräsentieren dabei das gewünschte Mapping zwischen einer Ereignisquelle (z. B. Rabbit MQ) und einem Kubernetes *Deployment* (siehe Abschnitt 9.3.3.1) oder einem *StatefulSet* (siehe Abschnitt 9.3.3.4).
- **ScaledJobs** repräsentieren das Mapping zwischen Ereignisquelle und einem Kubernetes-*Job* (siehe Abschnitt 9.3.3.2).

DaemonSets sind nicht durch KEDA skalierbar, da sie per Definition auf jedem Host des Clusters laufen sollen (siehe Abschnitt 9.3.3.3) und damit keinen sinnvoll zu skalierenden Workload aus dem Blickwinkel der ereignisgesteuerten Autoskalierung darstellen.

KEDA Skalierer agieren dabei auf einer Metrik. Wird der Zielwert einer Metrik (z. B. Länge von Einträgen in einer Warteschlange) unterschritten, wird dies als Ereignis zum Herunterskalieren gewertet, wird der Zielwert überschritten, wird dies als Ereignis zum Hochskalieren gewertet. Ergänzend können auch noch „Dämpfungen“ und Prüfintervalle definiert werden, um zu vermeiden, dass zu „hektisch“ skaliert wird. Anders als der *Horizontal Pod Autoscaler* von Kubernetes kann KEDA Workloads auch komplett deaktivieren und so bei keinen vorlie-

genden Ereignissen auch auf null skalieren (**Scale-to-Zero**). Muss ein *ScaledObject* allerdings erst aktiviert werden, erhöht dies natürlich die Start-up-Latenz substanziell. Insofern sind auch bei KEDA die typischen FaaS-Latenzschwankungen zu beobachten.

Listing 10.4 zeigt ein Beispiel eines autoskalierten To-do-Workloads. Für diesen Workload werden zwei Trigger registriert.

1. Ein periodischer Cron-Trigger, der einmal zwischen 00:30 Uhr und 00:45 Uhr den `todos-scaler`-Workload startet, um mindestens einmal pro Tag eingehende To-dos zu bearbeiten.
2. Ferner ein streaming-basierter Trigger, der ereignisbasiert auf Basis von aufgelaufenen Einträgen einen Stream skaliert. Im Stream `todos` laufen To-dos auf und sollen bei mehr als fünf To-dos durch den `todos-workload` bearbeitet werden (oder einmal in der Nacht gemäß Trigger 1).

Für die Skalierung wird in Listing 10.4 zudem eine obere (`maxReplicaCount`) oder untere Grenze (`minReplicaCount`) gesetzt. Ferner sind eine Abklingzeit (`cooldownPeriod`) und Abfrageintervall (`pollingInterval`) jeweils in Sekunden definiert. Die Abklingzeit ist dabei die Zeitspanne, die nach dem letzten aktivierten Trigger gewartet wird, bevor die Skalierung wieder auf 0 zurückgeht. Das Abfrageintervall ist das Intervall, in dem die Trigger für Skalierungsentscheidungen überprüft werden.

Listing 10.4 Beispiel einer KEDA-Skalierungsregel

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: todos-scaler
spec:
  scaleTargetRef:
    name: todos-workload      # Zu skalierender Workload
  cooldownPeriod: 200          # Dämpfung
  maxReplicaCount: 25          # Obere Skalierungsgrenze
  minReplicaCount: 0           # Untere Skalierungsgrenze
  triggers:
    - type: cron               # [1] Periodischer Trigger
      metadata:
        timezone: Europe/Berlin
        start: 30 * * * *
        end: 45 * * * *
        desiredReplicas: 1
    - type: redis-streams       # [2] Trigger (Warteschlangenlänge)
      metadata:
        stream: todos
        consumerGroup: process-todos
        pendingEntriesCount: 5
```

Wie man am Beispiel von KEDA sehen kann, ist es also durchaus möglich, Scale-to-Zero-Skalierungen auch ohne FaaS-Plattform zu realisieren. Hierdurch lässt sich auch die fehlende Standardisierung einer über alle Plattformen und Anbieter einheitliche FaaS-Runtime-Schnittstelle kompensieren, da eine solche Schnittstelle nicht erforderlich ist, da nur ganz normale Kubernetes Workloads skaliert werden.

■ 10.4 Zusammenfassung

Container-Orchestrierungsplattformen – wie beispielsweise Kubernetes – skalieren Workloads normalerweise nie bis auf null (siehe auch Abschnitt 9.3.5), da dies als ein „unhealthy“ Workload-Zustand interpretiert werden würde. Daher ist immer eine letzte Ausführungsinstanz erforderlich, die stets ein Constant Workload gemäß Abschnitt 2.3 beinhaltet, der grundsätzlich die unwirtschaftlichste Last für Cloud Computing darstellt. Mittels der Scale-to-Zero-Fähigkeit von FaaS-Plattformen lässt sich auch dieser letzte (unwirtschaftlichste) Anteil komplett dem Pay-as-you-go-Kostenmodell unterwerfen, da auf konzeptionelle Always-on-Komponenten verzichtet werden kann. Studien von (Villamizar u. a. 2017) haben z. B. gezeigt, dass sich substantielle Betriebsersparnisse hierdurch von bis zu 75 % aus Kundensicht ergeben können.

Das FaaS-Programmiermodell (siehe Abschnitt 10.1) ist dabei ereignisgesteuert, zustandslos und beruht auf Konzepten wie *Ereignisquellen*, *Trigger*, *Regeln* und *Funktionen*. Ereignisse können dabei aus unterschiedlichsten Quellen wie Streaming-Systemen, Datastores oder auch sonstigen (managed) Cloud-Services stammen. Der Einsatz dieses Modells bringt nach (Jonas u. a. 2019) oft Kostenvorteile im Betrieb, eine inhärente Elastizität und Skalierbarkeit sowie eine gesteigerte Entwicklungsproduktivität. Allerdings sind auch Nachteile mit dieser Form des Utility Computings hinsichtlich Zustandslosigkeit, limitierten Ausführungsduern und mit Kaltstarts zusammenhängenden Latenzschwankungen verbunden (Jonas u. a. 2019).

Die aktuelle FaaS-Plattform-Landschaft ist durch viele Produkte sowie Managed Service-Provider gekennzeichnet und u. a. sehr stark vom AWS Lambda Service geprägt. Wesentliche Portabilitätsprobleme bei FaaS entstehen u. a. durch eine noch fehlende Standardisierung des Funktions-Interface (siehe Abschnitt 10.2). Durch selbst gehostete FaaS-Plattformen kann man jedoch eine zu starke Abhängigkeit von Service-Providern vermeiden, indem man auf Plattformen setzt, die sowohl self-hosted als auch als managed Service genutzt werden können. Hier sei insbesondere auf OpenWhisk (Sciabarrà 2019) verwiesen. Self-hosted FaaS-Plattformen erfordern natürlich, dass man dennoch Infrastruktur für den Betrieb der FaaS-Plattform bereitstellen muss, was den Vorteil des „*Serverless*“-Konzepts natürlich reduziert. Ein anderer Ansatz, die Portabilität von FaaS-Funktionen sicherzustellen, ist die Nutzung plattformagnoischer Frameworks (siehe Abschnitt 10.2). Agnostische Frameworks definieren Funktionen FaaS-plattformunabhängig und „übersetzen“ diese agnostischen Funktionen für spezifische FaaS-Plattformen oder FaaS-Provider. Hier ist insbesondere (Collins, Radkakrishnan und Fine 2015) zu nennen.

Setzt man eh eine Orchestrierungsplattform wie Kubernetes ein, können leichtgewichtige Ansätze wie eine ereignisgesteuerte Skalierung einen zu FaaS vergleichbaren Effekt bringen (siehe Abschnitt 10.3). Bei der ereignisgesteuerten Skalierung nutzt man Metriken zur Skalierung, die sich nicht notwendig am Ressourcenverbrauch orientieren. Mit KEDA (Microsoft 2014) lässt sich so die Skalierung von Workloads in Kubernetes basierend auf der Anzahl der zu verarbeitenden Ereignisse steuern. KEDA ist als leichtgewichtige Komponente konzipiert, die ergänzend zu jedem Kubernetes-Cluster hinzugefügt werden kann, um Workloads skalieren zu können, die einer ereignisgesteuerten Autoskalierung unterworfen werden sollen. Dies müssen im Übrigen keine Funktionen gemäß dem FaaS-Programmiermodell sein. Anwendungsbezogene Einstiege in diese Thematik geben beispielsweise (Zambrano 2018) oder (Katzer 2020) zumeist an Beispielen des AWS Lambda Service.

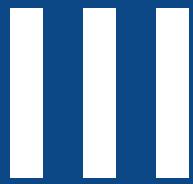
Ergänzende Materialien

Auf der Website zum Buch finden sich zu diesem Kapitel ferner folgende Ergänzungsmaterialien:

- Praktische Übungen (Labs) zu den FaaS-Plattformen Google Cloud Functions, Kubeless und OpenWhisk
- Übersichten inklusive Links zu FaaS-Plattformen, die als Managed Service (Public Cloud) angeboten werden
- Übersichten inklusive Links zu FaaS-Plattformen und ereignisbasierten Autoskalierern, die self-hosted betreibbar sind (Private Cloud)
- Übersichten inklusive Links zu plattformagnostischen FaaS-Frameworks
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer (Themen: Serverless Computing, Function as a Service (FaaS) , FaaS-Programmiermodell, FaaS-Plattformen)



<https://bit.ly/2WmWg2V>



Teil III: Cloud-native Architekturen

11

Einleitung zu Teil III

„Der Unterschied zwischen guter und schlechter Architektur ist die Zeit, die man dafür aufwendet.“

Sir David Alan Chipperfield, britischer Architekt minimalistischer Bauten

Während sich Teil II vor allem mit den technischen Bausteinen Cloud-nativer Systeme und Anwendungen befasst hat, wird sich Teil III nun primär mit der architekturellen Ebene von Cloud-nativen Systemen befassen und sich der Frage widmen, wie man die technischen Bausteine des Teil II zielführend „verschaltet“. Es werden dabei Architekturansätze und Methodiken behandelt, wie Cloud-native Systeme und Anwendungen gebaut werden können, die gemäß den in Kapitel 2 gezeigten ökonomischen Gesetzmäßigkeiten kosteneffektiv betrieben und evolutionär weiterentwickelt werden können.

Kapitel 12 betrachtet hierzu vor allem die beiden das Cloud-native Umfeld stark prägenden Architekturstile Microservice- und Serverless-Architekturen. Vor dem Hintergrund der bereits in Kapitel 3 genannten DevOps-Prinzipien geht dieses Kapitel dabei insbesondere darauf ein, wie unabhängig deploybare und aktualisierbare Komponenten von Cloud-nativen Anwendungen in sich evolutionär entwickelnden Architekturen integriert, fehlertolerant betrieben und skaliert werden können. Dies beinhaltet nicht nur technische, sondern auch organisatorische Überlegungen. Es werden dabei datenbankbasierte, request-response-basierte und ereignisbasierte Integrationsstile genauso behandelt wie Fehlertoleranzkonzepte (Architectural Safety) und Skalierungskonzepte für Komponenten Cloud-nativer Systeme und Anwendungen.

In **Kapitel 13** geht es hingegen stärker um den Betrieb und die Beobachtbarkeit von Cloud-nativen Anwendungen zur Laufzeit. Insbesondere beleuchtet dieses Kapitel, wie Cloud-native Anwendungen systematisch mittels Instrumentierung beobachtet werden können, um Rückschlüsse über Optimierungspotenziale aus dem Betrieb sammeln zu können. Hierzu ist eine Konsolidierung von Telemetriedaten erforderlich, die mittels existierender Technologiestacks realisierbar ist. Die hierzu erforderliche Instrumentierung von Systemen kann dabei manuell mittels Logging, Monitoring und Tracing-Lösungen erfolgen. Es ist aber auch möglich, die Instrumentierung Cloud-nativer Anwendungen automatisiert mittels Service-Mesh-Lösungen vorzunehmen. Service-Meshs ermöglichen darüber hinaus auch das Management und die Analyse von Verkehrstopologien innerhalb von Cloud-nativen Anwendungen, was diverse evolutionäre Roll-out-Strategien wie Canary-Releases und Blue/Green-Deployments vereinfacht.

Kapitel 14 bildet den Abschluss des gewählten Bottom-up-Ansatzes dieses Buchs. Dieses Kapitel befasst sich damit, wie sich Cloud-native Anwendungen systematisch entwerfen lassen. Dabei wird zusammenfassend die stark von Fachlichkeit geprägte Methodik des Domain

Driven Designs betrachtet, und insbesondere werden die Tätigkeiten des strategischen und taktischen Designs zu Microservices in Bezug gesetzt. Die Methodik des Domain Driven Design stammt zwar aus einer Zeit, als der Begriff „Cloud-native“ noch nicht einmal existierte. Dennoch ist das Architektur-Konzept des Bounded Contexts dieser Methodik erstaunlich passend zum eher technisch verstandenen Konzept des Microservice aus Kapitel 12. Die Methodik des Domain Driven Designs harmoniert sehr gut mit dem sich aus DevOps-Erfordernissen (siehe Kapitel 3) ergebenden evolutionären Architekturansatz von Microservices (siehe Kapitel 12). Das Kapitel gibt einen Überblick, wie man methodisch entlang einer strategischen Domänenanalyse mittels Subdomains, Bounded Contexts und Context Mapping zu einem taktischen Design für Geschäftslogik und Architekturen für unabhängig deploybare und aktualisierbare Komponenten Cloud-nativer Anwendungen und Services kommt.

In den Schlussbemerkungen in **Kapitel 18** werden noch einmal alle in diesem Buch behandelten Pattern und Best Practices zur Entwicklung Cloud-nativer Anwendungen und Dienste tabellarisch aufgeführt. Dadurch finden sich relevante Stellen gegebenenfalls etwas zielgerichteter, als das rein über das Inhaltsverzeichnis möglich wäre.

12

Microservice und Serverless-Architekturen

„You build it, you run it.“

Werner Vogels, CTO und Vicepresident von Amazon Web Services

Eine Komponente ist eine Einheit von Software, die unabhängig austauschbar und erweiterbar ist. Komponenten können daher als prozessfremd (Out-of-Process-Komponenten) oder prozesseigen als Bibliotheken (In-Process-Komponenten) konzipiert sein. Im Gegensatz zu Services sind Bibliotheken Komponenten, die in ein Programm eingebunden und über In-Memory-Funktionsaufrufe aufgerufen werden. Services sind hingegen prozessfremde Komponenten, die über einen Mechanismus wie z. B. eine Webservice-Anfrage oder andere Interprozess-Kommunikationsmechanismen wie Remote-Prozeduraufrufe kommunizieren.

Cloud-native Systeme sind üblicherweise Service-of-Services-Systeme und beruhen daher substanzial auf solchen Service-Interaktionen. Bei der Bearbeitung eines Service Requests ist es daher nicht ungewöhnlich, dass ein Service weitere nachgelagerte Services abfragt. Service Requests wandern somit sinnbildlich erst einmal die interagierenden Services entlang „flussaufwärts“ und die Antworten (Responses) fließen dann wieder „flussabwärts“ zum Requestor (User).

Hierfür haben sich die Begrifflichkeiten Upstream- und Downstream-Services etabliert (siehe auch Bild 12.1). Obwohl diese Terminologie oft verwendet wird, werden diese Begrifflichkeiten manchmal unterschiedlich genutzt. Das hängt damit zusammen, was als Bezugspunkt gewählt wird. Oft (leider nicht immer) kann man sich den Bezugspunkt aus dem Kontext erschließen. Wir betrachten diese Begrifflichkeiten in diesem Buch immer aus dem Blickwinkel des Consuming-Service. Consuming-Services fragen Upstream-Services an, die dann eine Providing-Service-Rolle haben. Der Unterschied zwischen Consuming- und Providing-Service-Rolle bezieht sich jeweils nur auf zwei interagierende Services. Es kann durchaus sein, dass ein Consuming-Service für einen anderen Downstream-Service die Rolle eines Providing-Service einnimmt, von diesem also upstream liegt.

Die Verwendung von Services zur Dekomposition von Systemen hat Vorteile. So ermöglicht sie u. a. eine explizitere Komponentenschnittstelle, da viele Programmiersprachen über keinen guten Mechanismus zur Definition einer expliziten öffentlichen Schnittstelle verfügen. Oft sind es nur Dokumentation und Disziplin, die verhindern, dass Clients die Kapselung einer Komponente aufbrechen, was zu einer zu engen Kopplung zwischen Komponenten führen kann. Services machen es einfacher, dies zu vermeiden, indem sie explizite Interprozess-Kommunikationsmechanismen technisch erforderlich machen.

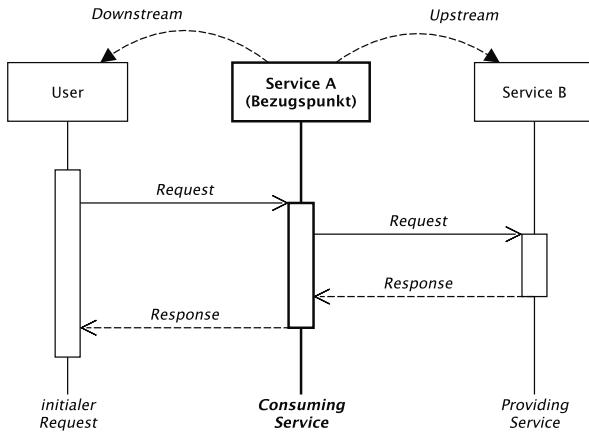


Bild 12.1
Upstream- und Downstream-Services

Die Dekomposition mittels Services hat jedoch auch Nachteile. Remote-Aufrufe sind meist teurer als prozessinterne Aufrufe, und daher müssen Remote-APIs grobgranularer sein, was oft umständlicher in der Anwendung ist. Wenn die Zuweisung von Verantwortlichkeiten zwischen Komponenten anzupassen ist, sind solche Verhaltensänderungen schwieriger zu bewerkstelligen, da Prozessgrenzen überschritten werden müssen.

Dennoch haben sich die Dekompositionen mittels Services bei Cloud-nativen Systemen in Form eines Microservice genannten Architekturstils durchgesetzt, der in diesem Kapitel genauer betrachtet werden soll. Im Rahmen von FaaS (siehe Kapitel 10) wurden sogar noch feingranularere Komponenten (Funktionen) eingeführt. Funktionen sind somit besonders kleine Microservices, die auf null Instanzen skaliert werden können (Scale-to-Zero). Mit diesen Funktionen – manchmal auch Nanoservices genannt – gehen auch architekturelle Auswirkungen einher, die manchmal als Serverless Architectures bezeichnet werden und in Abschnitt 12.6 behandelt und deren Besonderheiten in Bezug zu den Microservice-Überlegungen dieses Kapitels gesetzt werden sollen.

■ 12.1 Eigenschaften von Microservices

Folgt man Martin Fowler (siehe auch Bild 12.2), so ist der Microservice-Architekturstil ein Ansatz zur Entwicklung einer einzelnen Anwendung als eine Reihe kleiner Services, die jeweils in einem eigenen Prozess ausgeführt werden und mit leichtgewichtigen Mechanismen, oft einer HTTP-Ressourcen-API, kommunizieren. Diese Services sind um Geschäftsfunktionen herum aufgebaut und können unabhängig voneinander durch vollautomatische Bereitstellungsmechanismen aktualisiert ausgebracht werden. Es gibt nur ein Minimum an zentraler Verwaltung dieser Services, die in verschiedenen Programmiersprachen (polyglotte Programmierung) geschrieben sein können und unterschiedliche Datenspeichertechnologien verwenden (polyglotte Persistenz).

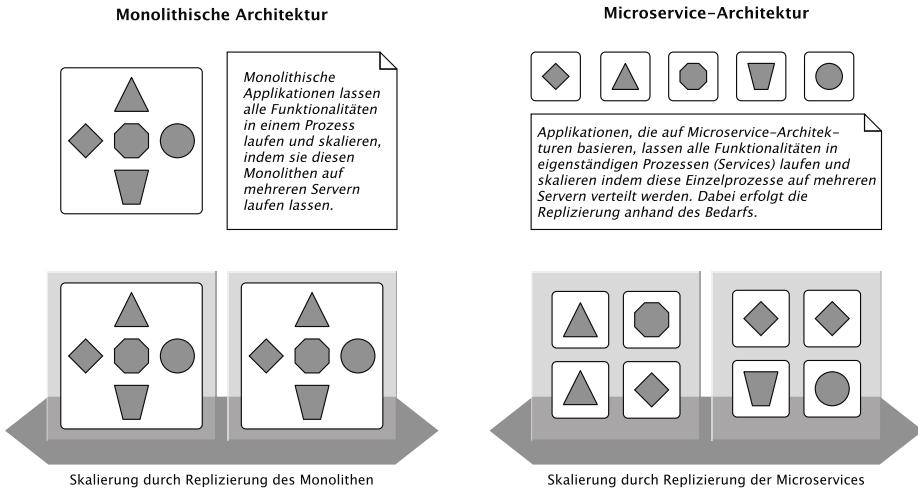


Bild 12.2 Monolithische und microservice-basierte Anwendungen

Besonders zu betonen ist dabei die Forderung der **unabhängigen Aktualisierbarkeit** der Einzelkomponenten solcher Architekturen. Erst eine unabhängige Aktualisierbarkeit ermöglicht den reibungsarmen 24x7-Betrieb großer und komplexer Systeme (siehe das DevOps-Problem aus Kapitel 3). Obwohl auch serviceorientierte Architekturen der Dekomposition mittels Services folgen, sind diese hingegen oft als Deployment-Monolithen entworfen, deren Komponenten nicht notwendig unabhängig voneinander aktualisierbar sind.

Auch ist die Dekomposition bei Microservice-Ansätzen häufig eine andere. Die Dekomposition erfolgt oft in Form von Services, die um die Geschäftsfähigkeiten herum organisiert sind. Solche Services nehmen eine Broad-Stack-Implementierung der Software für diesen Geschäftsbereich, einschließlich Benutzeroberfläche, persistente Speicherung und alle externen Kollaborationen vor. Folglich sind die Teams meist deutlich funktionsübergreifender hinsichtlich der gesamten Bandbreite an Fähigkeiten, die für die Entwicklung erforderlich sind: u. a. User-Experience, Datenbank- und Projektmanagement.

Obwohl „Microservice“ ein beliebter Name für diesen Architekturstil geworden ist, führt der Name zu einer manchmal etwas unglücklichen Fokussierung auf die Größe eines Services und zu Diskussionen darüber, wie groß oder klein „Mikro“ nun eigentlich ist. Dies ist oft nicht zielführend. Wesentlicher ist meist die Erkenntnis der unabhängigen Aktualisierbarkeit von Komponenten. Diese unabhängige Aktualisierbarkeit ist natürlich mit kleineren Komponenten meist einfacher zu realisieren. Doch ist die Größe nicht der alleinig ausschlaggebende Faktor. Daher kann man eine Reihe von Größen von Services im Microservice-Umfeld entdecken. Die größten Teams folgen Amazons Vorstellung des sogenannten Zwei-Pizza-Team (d. h., das gesamte Team wird von zwei *amerikanischen* Pizzen satt), was in etwa einem Dutzend Personen entspricht. Auf der kleineren Größenskala sind jedoch auch Teams von drei bis vier Personen bekannt, die in etwa genauso viele Services betreuen. Insofern gibt es auch bei Microservices kleinere und größere Services. Teamgrößen von mehr als einem Dutzend Personen pro Team sind jedoch eher ungewöhnlich und auch eher bei außergewöhnlich komplexen (wenn auch klar abgrenzbaren) Services zu finden.

Microservices unterscheiden sich ferner hinsichtlich ihres Entwicklungsmodells. Die meisten Anwendungsentwicklungen basieren oft auf einem Projektmodell, bei dem das Ziel darin besteht, eine Software zu liefern, die mit Projektende als fertiggestellt gilt. Nach der Fertigstellung wird die Software an eine Wartungsorganisation übergeben, und das Projektteam, das sie erstellt hat, wird oft aufgelöst oder für neue Aufgaben neu zusammengesetzt. Befürworter von Microservices bevorzugen stattdessen die Vorstellung, dass ein Team ein Produkt über dessen gesamte Lebensdauer hinweg im Sinne einer **Service Ownership** besitzen sollte. Dies bringt die Entwickler in täglichen Kontakt damit, wie sich ihre Software in der Produktion verhält, und erhöht den Kontakt zu den Anwendern, da sie zumindest einen Teil der Supportlast übernehmen müssen. Die **Produktmentalität** knüpft an die Verknüpfung mit den Geschäftsfunktionen an. Anstatt die Software als eine Reihe von Funktionen zu betrachten, die fertiggestellt werden müssen, gibt es eine fortlaufende Beziehung, bei der die Frage lautet, wie die Software ihre Benutzer dabei unterstützen kann, die Geschäftsfähigkeit zu verbessern.

Microservice-Architekturen folgen somit einer **evolutionären Design-Philosophie**, die den unabhängigen Austausch und die Erweiterbarkeit von Komponenten als Leitgedanken hat. Diese Betonung der Austauschbarkeit ist dabei letztlich nur ein Spezialfall eines allgemeineren Prinzips des modularen Designs. Letztlich versucht man Dinge, die sich gleichzeitig ändern, im selben Modul zu behalten, um nur den geänderten Service neu bauen und deployen zu müssen. Bei monolithischen Ansätzen erfordert jede Änderung einen vollständigen Build und ein Deployment der gesamten Anwendung. Diese unabhängige Aktualisierbarkeit kann Release-Prozesse damit vereinfachen und beschleunigen. Der Nachteil ist, dass man berücksichtigen muss, dass Änderungen an einem Service dessen Konsumenten beeinträchtigen können.

Wenn Services **lose gekoppelt** sind, sollte eine Änderung an einem Service keine Änderung an einem anderen erfordern. Der springende Punkt des Microservice-Ansatzes besteht darin, Änderungen an einem Service vorzunehmen und ihn bereitzustellen zu können, ohne dass ein anderer Teil des Systems geändert werden muss. Welche Art von Dingen verursacht eine enge Kopplung? Ein klassischer Fehler besteht darin, einen Integrationsstil zu nutzen, der einen Service eng an einen anderen bindet und dazu führt, dass Änderungen innerhalb eines Services Änderungen für die aufrufenden Services erforderlich machen. Ein lose gekoppelter Service weiß daher idealerweise so wenig wie möglich über die Services, mit denen er zusammenarbeitet. Dies bedeutet auch, dass die Anzahl der verschiedenen Arten von Requests zwischen zwei Services begrenzt werden sollten, da eine „gesprächige Kommunikation“ nicht nur zu potenziellen Leistungsproblemen, sondern auch zu tendenziell engeren Kopplungen führen kann.

Die Begrifflichkeiten lose Kopplung und hohe Kohäsion werden insbesondere in der Objekt-orientierten Analyse und Design (OOAD) genutzt, um „gute“ (also vor allem flexibel wieder-verwendbare) Domänenmodelle zu entwickeln. Doch diese Begriffe sind nicht der OOAD vorbehalten und können natürlich auch für das Design von Services herangezogen werden. Insbesondere in der objektorientierten Programmierung beschreibt Kohäsion, wie gut eine Programmeinheit eine logische Aufgabe oder Einheit abbildet. In einem System mit starker Kohäsion ist jede Komponente (Service) idealerweise für nur genau eine wohldefinierte Aufgabe zuständig (**Single-Responsibility-Prinzip**). Nur zusammengehörige Aufgaben sollten auch zusammen deployt werden müssen. Davon unabhängige Aufgaben sollten hingegen

unabhängig deployt werden können. Um dies zu erreichen, muss man allerdings zusammengehörige Aufgaben in Deployment Units voneinander isolieren. Warum? Wenn wir das Verhalten zusammengehöriger Aufgaben ändern möchten, möchten wir es an einem Ort ändern und diese Änderung so schnell wie möglich freigeben können. Wenn man hierfür erforderliche Änderungen an vielen verschiedenen Orten in den Quelltext einbringen müsste, wären viele verschiedene Services (möglicherweise gleichzeitig) freizugeben, um diese Änderung ausbringen zu können. Das gleichzeitige Bereitstellen vieler Services ist hingegen meist riskant. Das Bestreben von DevOps (siehe Kapitel 3) sind jedoch schnelle und nicht riskante Deployments. Architekturen mit einer hohen Kohäsion und loser Kopplung sind also eine Voraussetzung für agile DevOps-Ansätze (viele kleine, schnelle, nicht riskante Updates).

In diesem Zusammenhang wird auch häufig das „Gesetz von Conway“ genannt. Diese nach dem Informatiker Melvin Conway benannte Beobachtung besagt, dass die Strukturen von Systemen durch die Kommunikationsstrukturen der sie umsetzenden Organisationen vorbestimmt sind.

„Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication structure.“ (Conway 1968)

Da demnach Organisationsstrukturen offenbar Architekturen und damit Eigenschaften von Systemen beeinflussen können, müsste man demzufolge Organisations- und Teamstrukturen schaffen, die einer Architektur mit hoher Kohäsion und loser Kopplung förderlich sind, um damit das Bestreben von DevOps (siehe Kapitel 3) schneller und nicht risikanter Deployments zu unterstützen. Beweise zur Stützung des Conway-Gesetzes wurden u. a. vom MIT und der Harvard Business School veröffentlicht. Hier wurden Softwareprodukte von lose gekoppelten Organisationen (z. B. Open-Source-Communitys) mit Produkten von eng gekoppelten Organisationen (z. B. straff geführten Unternehmen wie Microsoft) verglichen. Das Resultat war, dass Software lose gekoppelter Organisationen meist wesentlich modularer war als Softwareprodukte eng gekoppelter Organisationen. Auch interne Studien von Microsoft zu Windows Vista (das in der Einführungsphase durch massive Qualitätsprobleme aufgefallen war) führten zu ähnlichen Erkenntnissen. Unternehmen wie Netflix und Amazon haben tatsächlich ihre internen Strukturen – z. B. mit dem Zwei-Pizza-Prinzip – an den gewünschten Qualitätseigenschaften von Microservices ausgerichtet.

Das gemeinsame Element ist dabei zumeist das Konzept der Service Ownership. Im Allgemeinen bedeutet Service Ownership, dass das Team, das einen Service initial entwickelt hat, auch für Änderungen an diesem Service und dessen Betrieb verantwortlich ist. Das Team sollte sich frei fühlen, den Code nach Belieben umzustrukturieren, solange diese Änderung die konsumierenden Services nicht beeinträchtigt. Für viele Teams erstreckt sich Service Ownership auf alle Aspekte des Service, d. h. von der Anforderungserhebung über das Erstellen, Bereitstellen und Warten der Anwendung. Dieses Ownership-Modell führt zu einer erhöhten Autonomie von Teams und Liefergeschwindigkeit. Wenn ein Team für die Bereitstellung und Wartung der Anwendung verantwortlich ist, besteht ein Anreiz für die Erstellung von Services, die einfach bereitzustellen sind. Es ist also für ein Team nicht möglich „etwas über die Mauer zu werfen“ und dann zu verschwinden. Es gibt ja niemanden, dem man es zuwerfen kann! Beziehungsweise man wirft es sich selber zu. Dieses Modell überträgt die Entscheidungen an den Personenkreis, der am besten in der Lage ist, technische Entscheidungen fundiert zu treffen. Dadurch erhält das Team mehr Macht und Autonomie, ist aber auch für seine Arbeit verantwortlich.

■ 12.2 Integrationsmuster für Microservices

Es gibt eine ganze Reihe von technischen Optionen, wie Microservices untereinander kommunizieren können: SOAP, XML-RPC, REST ... Protokollpuffer (gRPC). Man sollte daher erst einmal darüber nachdenken, was man von einer Technologie erwartet, die im Kontext von Microservices und DevOps eingesetzt werden soll.

- Eine Leitlinie ist sicher die **Vermeidung von Breaking-Changes**. Ab und an sind Änderungen in einem Service erforderlich, die auch Änderungen in Consuming-Services erforderlich machen. Wir sollten also grundsätzlich Technologien bevorzugen, die sicherstellen, dass dies so selten wie möglich geschieht. Wenn ein Microservice beispielsweise einem gesendeten Datenelement neue Felder hinzufügt, sollten bestehende Verbraucher nicht betroffen sein. Dies macht beispielsweise Ansätze, die von client- wie serverseitig identisch generierten Stubs abhängig sind (z. B. RPC-basierte Lösungen), problematisch.
- Eine weitere Richtlinie ist die Verwendung möglichst **technologieagnostischer APIs**. Insbesondere in der Cloud-nativen Domäne kommen kontinuierlich neue Tools, Frameworks und Produkte heraus, u. a. weil einige Bereiche noch einen Konsolidierungsprozess durchlaufen müssen (z. B. die Standardisierung von FaaS). Es ist also nicht unwahrscheinlich, dass ein Microservice in naher Zukunft im Kontext eines alternativen Technologie-Stacks funktionieren muss. Für die Kommunikation zwischen Microservices bedeutet dies, dass APIs möglichst technologieunabhängig oder auf Basis von in der Vergangenheit bewährten Protokollen – wie etwa HTTP – gestaltet werden sollten. Ferner sollten Integrationstechnologien vermieden werden, die bestimmen, mit welchem Technologie-Stack Microservices implementiert werden müssen (z. B. Legacy Enterprise Service-Busses).
- Häufig genutzte Services sind oft durch eine **einfache Integration aus dem Blickwinkel von Consuming-Services** gekennzeichnet. Im Idealfall sollten Clients dabei die Freiheit bei der Auswahl ihrer Integrationstechnologie haben. Andererseits kann auch eine Client-Library für den Microservice die Akzeptanz durch eine schnellere Lernkurve und Entwicklungsgeschwindigkeit bei der Integration als Upstream-Service erhöhen. Man sollte jedoch bedenken, dass solche Client-Libraries möglicherweise nicht immer mit anderen Zielen deckungsgleich sind, die man ebenfalls erreichen möchte. Komfortable Client-Bibliotheken können zwar die Integration erleichtern, führen jedoch auch meist zu einer höheren Kopplung.
- Bei der Integration von Services sollte man **Implementierungsdetails kapseln**. Consuming-Services sollten nicht an interne Implementierung von Upstream-Services gebunden sein. Dies führt meist zu einer erhöhten Kopplung. Ein Microservice sollte also so entwickelt sein, dass interne Änderungen an Upstream-Services nicht erfordern, dass auch Consuming-Services geändert werden müssen. Daher sollten in Microservices grundsätzlich Technologien vermieden werden, die es erforderlich machen, interne Repräsentation freizulegen (um z. B. Performance zu gewinnen).

Wir können vor diesem Hintergrund mehrere hinsichtlich ihres Kopplungsgrades abnehmende Integrationsstile ausmachen, die im Folgenden detaillierter vor dem Hintergrund der gerade genannten Leitlinien betrachtet werden sollen:

- Datenbasierte Integration
- RPC-basierte Interprozesskommunikation
- HTTP-/REST-basierte Integration inklusive API-Versioning
- Ereignisbasierte Integration

12.2.1 Datenbankbasierte Integration

Die häufigste Form der Integration „in freier Wildbahn“ ist vermutlich die datenbankbasierte Integration auf Basis einer Datenkopplung (siehe Bild 12.3). Wenn Services Informationen von einem Stateful-Service benötigen, müssen sie nur eine Datenbank abfragen. Und wenn sie Daten ändern wollen, aktualisieren oder erstellen sie Datensätze in einer Datenbank. Dies ist wahrscheinlich die schnellste und beliebteste Form der Integration. Obwohl dies ein weit verbreitetes Muster ist, ist es durchaus mit Schwierigkeiten hinsichtlich unabhängiger Aktualisierbarkeit von Services behaftet.

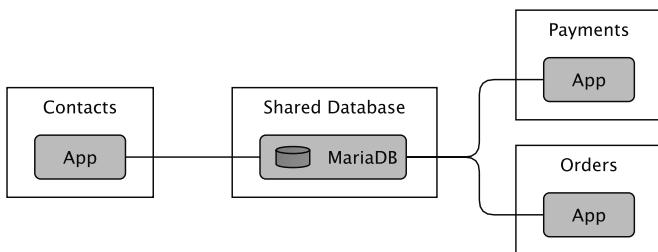


Bild 12.3
Shared-Database-Pattern

- Man zwingt Consuming-Services dazu, sich an interne Implementierungsdetails zu binden. Die Datenstrukturen der Datenbank werden vollständig mit allen anderen Parteien geteilt, die Zugriff auf die Datenbank haben. Das Ändern des Datenbankschemas kann Änderungen in Consuming-Services nach sich ziehen. Dies erhöht die Kopplung und damit entsprechend erforderliche Abstimmungen von Änderungen zwischen Services.
- Consuming-Services sind nach diesem Pattern an eine bestimmte Datenbanktechnologie gebunden. Infolgedessen könnten Consuming-Services beispielsweise datenbankspezifische Treiber verwenden, die im Falle einer Datenbankänderung problematisch werden können (z. B. MariaDB anstelle von Postgres). Implementierungsdetails sollten vor Consuming-Services grundsätzlich verborgen werden, um ein gewisses Maß an Autonomie in Bezug darauf zu ermöglichen, wie Stateful-Services ihre Interna im Laufe der Zeit ändern können.

Daher haben sich im Cloud-native Umfeld weitere Integrationsmuster entwickelt, die die Autonomie zwischen Consuming- und Providing-Services besser sicherstellen können.

12.2.2 (g)RPC-basierte Interprozesskommunikation

Remote Procedure Calls (RPC) sind eine Technik zur Realisierung von Interprozesskommunikation, die den Aufruf von Funktionen in anderen Adressräumen ermöglicht. Im Normal-

fall werden die aufgerufenen Funktionen auf einem anderen Computer als das aufrufende Programm ausgeführt. Es existieren mehrere Implementierungen dieser Technik, die in der Regel untereinander nicht kompatibel sind. Die am weitesten verbreitete Variante ist das ONC RPC (Open Network Computing Remote Procedure Call), das vielfach auch als Sun RPC bezeichnet wird. ONC RPC wurde ursprünglich durch Sun Microsystems für das Network File System (NFS) entwickelt.

RPC basiert auf dem Client-Server-Modell. Die Kommunikation beginnt, indem der Client eine Anfrage (Call) an einen ihm bekannten Server schickt und auf die Antwort wartet. In der Anfrage gibt der Client an, welche Funktion mit welchen Parametern ausgeführt werden soll. Der Server bearbeitet die Anfrage und schickt die Antwort an den Client zurück. Nach Empfang der Nachricht kann der Client seine Verarbeitung fortführen.

Um eine entfernte Funktion aufzurufen, muss eine Nachricht vom Client-Prozess zum Server-Prozess versendet werden. In dieser müssen der Name der Funktion (oder eine ID) und die zugehörigen Parameterwerte enthalten sein. Die Suche nach einem entsprechenden Server kann durch Broadcast (in einem lokalen Netz) realisiert werden oder durch Inanspruchnahme eines Verzeichnisdienstes (Registry). Die erforderlichen serverseitigen (`register`) und clientseitigen (`lookup + binding + error recovery`) Vorgänge werden normalerweise in sogenannten Client-/Server-Stubs gekapselt, die aus Interface Definitions generiert werden.

Aus Sicht eines client-seitigen Entwicklers sehen Remote Procedure Calls wie normale Local Procedure Calls (LPC) im eigenen Prozessraum aus. Da RPCs aber Netzwerkkommunikation beinhalten, können diese – anders als LPC im eigenen Prozessraum – natürlich scheitern. Weitere diesem RPC-Modell folgende Technologien sind etwa CORBA, RMI (Java), XML-RPC oder JSON-RPC.

Im Cloud-native Umfeld ist ferner gRPC populär. gRPC (gRPC Remote Procedure Call) ist ein universelles RPC-Protokoll zum Aufruf von Remote Procedures in verteilten Systemen. Es basiert auf HTTP/2 und Protocol Buffers und ist ein Projekt der Cloud-native Computing Foundation. Mittels einer IDL wird die Schnittstelle unabhängig von einer konkreten Programmiersprache spezifiziert. Mittels des Protocol-Buffer-Compilers `protoc` (und eines Plug-ins für gRPC) kann aus dieser Beschreibung Server- und Client-Code, sogenannte Stubs, generiert werden.

Da gRPC auf HTTP (und somit auf Anwendungsebene definiert ist) und nicht auf UDP (wie RPC) beruht, ist es einfacher zu handhaben (beispielsweise mittels Kubernetes Ingress Rules, siehe Kapitel 9). Die Kopplung zwischen Services ist somit im Vergleich zum „normalen“ RPC als geringer einzuschätzen. Requestor und Service müssen nicht in derselben Sprache geschrieben sein wie etwa bei Java RMI. Wie in vielen RPC-Systemen basiert aber auch gRPC auf der Idee, einen Service zu definieren und die Methoden anzugeben, die mit ihren Parametern und Rückgabetypen remote aufgerufen werden können. Auf der Serverseite implementiert ein Service diese Schnittstelle und führt einen gRPC-Server-Prozess aus, um Client-Requests zu verarbeiten (siehe Bild 12.4). Auf der Clientseite verfügt der Client über einen Stub, der dieselben Methoden wie der Server-Prozess bereitstellt. gRPC-Clients und -Server können in einer Vielzahl von Laufzeitumgebungen ausgeführt werden und miteinander kommunizieren. Es werden unter anderem die Programmiersprachen C, C++, C#, Dart, Go, Java, Kotlin, Node.js, Objective-C, PHP, Python und Ruby unterstützt; wobei diese Liste nicht als vollständig verstanden werden sollte.

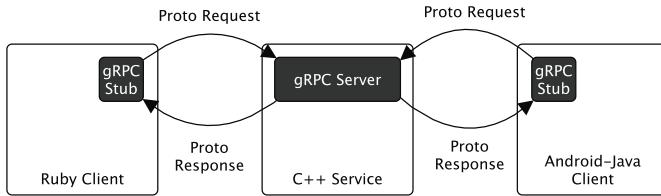


Bild 12.4
gRPC

Standardmäßig verwendet gRPC Protocol Buffers zum Serialisieren strukturierter Daten, obwohl es auch mit anderen Datenformaten wie etwa JSON verwendet werden kann. Hierzu müssen die zu serialisierenden Daten und Servicemethoden, wie in Listing 12.1 exemplarisch gezeigt, definiert werden. Protocol Buffer-Daten sind als Nachrichten strukturiert, wobei jede Nachricht aus einer Liste von Feldern (Key-Value Paare) besteht, die den Namen, den Datentyp und die Serialisierungsposition innerhalb der Nachricht angeben. Mittels des Protocol Buffer Compiler protoc lassen sich Protokolldefinitionen in Datenzugriffsklassen unterstützter Programmiersprachen generieren.

Listing 12.1 Exemplarische gRPC Interface-Definition (.proto-Datei)

```

// The greeter service definition.
service Greeter {
    // Sends a greeting
    rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
    string name = 1;
}

// The response message containing the greetings
message HelloReply {
    string message = 1;
}

```

Mittels gRPC lassen sich dabei vier Arten von Servicemethoden definieren, mit denen sich auch Streaming sowohl auf Client- als auch auf Server-Seite realisieren lässt.

- Bei **Unary RPCs** sendet der Client einen einzelnen Request an einen Service und erhält eine einzelne Response als Antwort. Dies erfolgt wie bei einem ganz normalen synchronen Funktionsaufruf.
- Bei **Server-Streaming-RPCs** sendet der Client einen Request an einen Service und erhält als Antwort einen Stream von Nachrichten. Der Client liest aus dem Rückgabe-Stream, bis keine Nachrichten mehr vorhanden sind.
- Bei **Client-Streaming-RPCs** sendet der Client einen Stream von Nachrichten an den Service. Sobald der Client die Nachrichten fertig geschrieben hat, wartet er darauf, dass der Server sie liest und seine Response zurückgibt.
- Bei **Bidirectional Streaming-RPCs** senden beide Seiten Streams von Nachrichten. Die beiden Streams arbeiten unabhängig voneinander, sodass Clients und Server in beliebiger Reihenfolge lesen und schreiben können.

Da gRPC ein auf HTTP/2-basierendes Binärprotokoll ist (HTTP/1 ist ein Textprotokoll), ist gRPC daher bis zu einer Größenordnung schneller als übliche HTTP-basierte Request-Response-Verfahren. Allerdings erhöht auch gRPC die Kopplung zwischen Services (durch Stubs und Service-Definition-Files, siehe z. B. Listing 12.1). Muss man .proto-Dateien in gRPC-Dateien anfassen, muss man meist sowohl Consuming- wie auch Providing-Service aktualisieren. Mit automatisierten Deployment-Pipelines kann man diesen Aufwand zwar „verstecken“, allerdings erhöht dies dennoch die Kopplung zwischen Services.

Synchrone – also blockierende – RPC-Aufrufe sind die engste Annäherung an die Abstraktion eines lokalen Prozedurauftrags, den das RPC-Modell ja grundsätzlich anstrebt. Andererseits sind Netzwerkzugriffe von Natur aus I/O-Prozesse und damit um Größenordnungen langsamer als lokale Prozeduraufträge. Daher ist es in vielen Szenarien üblich, RPCs asynchron zu starten, ohne den aktuellen Thread zu blockieren. Die gRPC-Programmier-API ist in den meisten Sprachen daher sowohl synchron als auch asynchron (d. h. mittels Futures und await/async) nutzbar.

Auf der begleitenden Website zum Buch findet der Leser u. a. Labs, die gRPC praktisch vertiefen.

12.2.3 Representational State Transfer (REST)

REST ist ein von Roy Fielding propagiertes Paradigma für die Softwarearchitektur von verteilten Systemen, insbesondere für Webservices (Fielding 2000). REST hat das Ziel, einen request-response-basierten Architekturstil zu schaffen, der die bereits existierende Infrastruktur des WWW synergetisch, aber dennoch konzeptionell fundiert nutzt. REST fordert dabei insbesondere eine einheitliche Schnittstelle zu Ressourcen im Netz. Der Zweck von REST liegt schwerpunktmäßig auf der Maschine-zu-Maschine-Kommunikation und stellt eine einfache Alternative zu ähnlichen Verfahren oder dem verwandten RPC-Verfahren dar. Anders als bei vielen verwandten Architekturen kodiert REST jedoch keine Methodeninformation im Uniform Resource Identifier (URI), da der URI Ort und Namen der Ressource angibt, nicht aber die Funktionalität, die der Webdienst zu der Ressource anbietet. Der Vorteil von REST liegt darin, dass im WWW bereits ein Großteil der für REST nötigen Infrastruktur (z. B. Web- und Application-Server, HTTP-fähige Clients usw.) vorhanden ist und viele Webdienste per se REST-konform sind. Eine Ressource kann dabei über verschiedene Medientypen repräsentiert werden.

REST folgt dabei den folgenden Prinzipien:

1. **Client-Server:** Dabei stellt der Server (Provider) einen Dienst bereit, der bei Bedarf vom Client (Requestor) angefragt werden kann. Der Hauptvorteil, den diese Anforderung bringt, ist die einfache Skalierbarkeit der Server, da diese unabhängig vom Client agieren. Dies ermöglicht u. a. eine unterschiedlich schnelle Entwicklung der beiden Komponenten.
2. **Caching:** HTTP Caching kann zur Performance-Optimierung genutzt werden. Mittels Caching kann es allerdings passieren, dass Clients auf veraltete Cache-Daten zurückgreifen.
3. **Zustandslosigkeit (Stateless):** Jede REST-Nachricht enthält alle Informationen, die für den Server bzw. Client notwendig sind, um die Nachricht zu verstehen. Weder der Server noch die Anwendung soll Zustandsinformationen zwischen zwei Nachrichten speichern.

Jeder Request eines Clients an den Server ist in sich geschlossen und muss daher sämtliche Informationen über den Anwendungszustand beinhalten, die vom Server für die Verarbeitung der Anfrage benötigt werden. Diese Eigenschaft begünstigt insbesondere die horizontale Skalierbarkeit eines Services.

4. **Mehrschichtige Systeme:** Die Systeme sollen mehrschichtig aufgebaut sein. Dadurch reicht es, dem Anwender lediglich eine Schnittstelle anzubieten. Dahinter liegende Ebenen können verborgen bleiben und somit die Architektur insgesamt vereinfacht werden. Dies ermöglicht eine bessere horizontale Skalierbarkeit der Server sowie eine mögliche Abschottung mittels Firewalls. Durch Cache-Speicher an den Grenzen (z. B. vom Server zum Web) kann die Effizienz der Anfragen erhöht werden.
5. **Einheitliche Schnittstelle:** REST-basierte Services sollen eine einheitliche Schnittstelle anbieten, die auf selbstbeschreibenden Nachrichten, adressierbaren Ressourcen, Repräsentationen von Ressourcen sowie dem Prinzip des *Hypermedia as the Engine of Application State (HATEOAS)* beruht.

Die Einheitlichkeit und einfache Nutzbarkeit der Schnittstelle ist dabei mittels der folgenden Prinzipien zu gewährleisten.

REST-Nachrichten sollen **selbstbeschreibend** sein. Dazu zählt u. a. die Verwendung von Standardmethoden (beispielsweise HTTP-Verben, siehe auch Tabelle 12.1). Über diese Standardmethoden lassen sich Ressourcen manipulieren. Jede Ressource eines Services sollte ferner über einen Uniform Resource Identifier (URI) identifiziert werden können. Jeder REST-konforme Service ist über einen Uniform Resource Locator (URL) adressierbar. Dieser Locator standardisiert den Zugriffsweg zum Angebot eines Webservices für eine Vielzahl von Anwendungen (Clients). Eine derartig konsistente Adressierbarkeit erleichtert es, einen Webservice im Rahmen eines Service-of-Service-Ansatzes zu nutzen. Ressourcen können dabei unterschiedliche Darstellungsformen (Repräsentationen) haben. Ein REST-konformer Service kann verschiedene Repräsentationen einer Ressource ausliefern (häufig HTML, JSON oder XML) oder auch die Beschreibung oder Dokumentation des Services selbst. Die Veränderung einer Ressource (also deren Status) soll nur über eine Repräsentation erfolgen.

Beim **HATEOAS-Prinzip** navigiert der Client einer REST-Schnittstelle ausschließlich über URLs, die vom Service bereitgestellt werden (siehe Listing 12.2). Abhängig von der gewählten Repräsentation geschieht die Bereitstellung der URIs über Hypermedia, also z. B. in Form von „href“- und „src“-Attributen bei HTML-Dokumenten, oder in für die jeweilige Schnittstelle definierten und dokumentierten JSON- bzw. XML-Attribut-/Elementen. Die Links zu Folgeoperationen stehen demnach in der Antwort und müssen dem Client nicht implizit bekannt sein. Die Links können daher auch in einer neuen Service-Version geändert werden, ohne den Client anpassen zu müssen. HATEOAS-konforme REST-Services sind aufgrund dieser Eigenschaft formal ein endlicher Automat, dessen Zustandsveränderungen durch die Navigation mittels der bereitgestellten URIs erfolgen. HATEOAS ermöglicht dadurch lose Kopplung von Services und gewährleistet, dass die Schnittstelle auf Provider-Seite verändert werden kann, ohne dass diese Anpassung auf Client-Seite implementierungstechnisch nachvollzogen werden muss. Im Gegensatz dazu kommuniziert beispielsweise ein SOAP-basierter Webservice über ein fixiertes Interface. Für eine Änderung des Services muss hier eine neue Schnittstelle bereitgestellt und in Form einer Schnittstellenbeschreibung (ein WSDL-Dokument) definiert werden.

Listing 12.2 Beispiel einer REST-Ressource inklusive HATEOAS-konformer Folgeoperationen (Links)

```

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: ...

{
  "account": {                                     # Zustand
    "account_id": "123abc",
    "balance": {
      "currency": "EUR",
      "value": 100.0
    },
    "links": {                                     # Links auf Folgeoperationen
      "deposit": "/accounts/123abc/deposit",      # Einzahlungsoperation
      "withdraw": "/accounts/123abc/withdraw",     # Abhebe-Operation
      "transfer": "/accounts/123abc/transfer",     # Überweisungsoperation
      "close": "/accounts/123abc/close"           # Konto-Schließungsoperation
    }
  }
}

```

CRUD umfasst die vier grundlegenden atomaren Operationen

- Create (Erzeugen)
- Read (Lesen)
- Update (Aktualisierung)
- Delete (Löschen)

persistenter Speicher und wird in vielen Systemen als Muster für persistierende Komponenten herangezogen. CRUD kann häufig mittels HTTP-Verben, wie in Tabelle 12.1 gezeigt, in REST-konforme Service-APIs übersetzt werden. Erläuternd sind die korrespondierenden SQL-92-Statements relationaler Datenbanken angegeben.

Tabelle 12.1 REST-Entsprechung von CRUD-Operationen

| CRUD | SQL-92 | HTTP (REST) |
|--------|--------|----------------|
| Create | INSERT | PUT oder POST |
| Read | SELECT | GET |
| Update | UPDATE | PATCH oder PUT |
| Delete | DELETE | DELETE |

Auf der Website zum Buch finden sich Labs, die die REST-basierte Integration praktisch vertiefen.

12.2.4 Ereignisbasierte Integration (asynchron)

Anstelle einer oft auf synchroner Request-Response basierenden Integration kehrt die ereignisbasierte Integration die Dinge um. Consuming-Services initiieren keine Requests mehr, um dann auf Responses zu warten. Consuming-Services werden vielmehr zu Event-Emitting-Services, die Ereignisse (Events) erzeugen, um mitzuteilen, dass etwas geschehen ist, und erwarten, dass andere Services wissen, wie diese Events zu verarbeiten sind. Event-Emitting-Services sagen niemandem, was zu tun ist. Daher sind ereignisbasierte Systeme von Natur aus asynchron und lose gekoppelt. Asynchrone Architekturen sind meist auch gleichmäßiger verteilt. Die Geschäftslogik ist also nicht zentralisiert, sondern wird gleichmäßig durch mehrere Services realisiert.

Die ereignisbasierte Integration resultiert somit meist in stärkerer Entkopplung von Services. Genau dies strebt man in Microservice-Architekturen an. Da ein Event erzeugender Service nicht weiß, welche anderen Services darauf reagieren, bedeutet dies auch, dass Ereignissen problemlos neue Abonnenten hinzugefügt werden können, was vor allem die horizontale Skalierung vereinfacht (siehe auch Bild 12.5). Daher erfreuen sich Messaging-Systeme wie Kafka oder NATS (aber auch Datenbanken wie Redis) steigender Beliebtheit im Cloud-native Computing, da diese Systeme oft eine pragmatische Grundlage für asynchrone und entkoppelte Architekturen legen können.

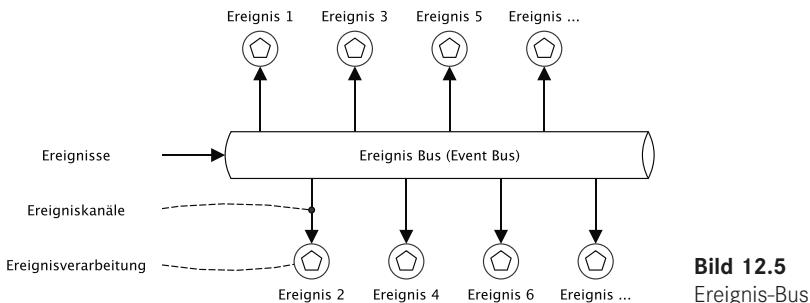


Bild 12.5

Ereignis-Bus

Große Anwendungen bestehen stets aus mehreren Komponenten und sind daher abhängig von deren Reaktivität. Systeme müssen

- stets antwortbereit (responsive),
- widerstandsfähig (resilient),
- elastisch (elastic) und
- nachrichtenorientiert (message-driven)

sein. Derartige Systeme werden dann als **reaktive Systeme** bezeichnet (Kuhn, Hanafee, und Allen 2017). Werden die genannten vier reaktiven Qualitäten in der Architektur einer jeden Ebene des Gesamtsystems berücksichtigt, sind reaktive Systeme miteinander komponierbar.

Die **Responsivität** (zeitgerechte Antwort) ist die Grundlage für Funktion, Benutzbarkeit und Zuverlässigkeit eines Systems, da Fehler in verteilten Systemen nur durch die Abwesenheit einer Antwort sicher festgestellt werden können. Ohne vereinbarte Timeouts ist die Erkennung und Behandlung von Fehlern nicht möglich.

Die **Widerstandsfähigkeit** von Systemen ist dabei durch Replizieren der Funktionalität, Isolation von Komponenten sowie Delegieren von Verantwortung erreichbar. Der Ausfall eines Teilsystems soll auf dieses begrenzt bleiben, um andere Teilsysteme nicht in ihrer Funktion zu beeinträchtigen. Die Wiederherstellung des Normalzustandes wird einer übergeordneten Komponente übertragen, die die geforderte Verfügbarkeit sicherstellt.

Elastische Systeme bleiben auch unter sich ändernden Lastbedingungen antwortbereit. Bei Laständerungen werden automatisch die Replizierungsfaktoren und damit die genutzten Ressourcen angepasst. Dazu darf das System keine Engpässe aufweisen, die den Gesamt-durchsatz vor Erreichen der geplanten Maximalauslegung einschränken. Ideal ist eine Architektur, die keine fixen Engpässe aufweist. In diesem Fall kann das System in unabhängige Komponenten zerlegt und diese können unabhängig auf beliebig viele Ressourcen verteilt werden. Reaktive Systeme unterstützen die Erfassung ihrer Auslastung zur Laufzeit, um automatisch regelnd eingreifen zu können. Dank ihrer Elastizität können sie auf Speziallösungen verzichten und mit handelsüblichen Komponenten implementiert werden.

Nachrichtenorientierte Systeme verwenden asynchrone Nachrichtenübermittlung zwischen Komponenten zur Sicherstellung von deren Entkopplung und Isolation. Die explizite Verwendung von Nachrichtenübermittlung führt zu ortsunabhängigen Komponenten und erlaubt die transparente Skalierung. Ortsunabhängigkeit bedeutet hier, dass Code und Semantik des Programms nicht davon abhängen, ob dessen Teile auf demselben Computer oder verteilt über ein Netzwerk ausgeführt werden. Die Überwachung von Nachrichtenpuffern ermöglicht kontinuierlichen Einblick in das Laufzeitverhalten des Systems. Nichtblockierende nachrichtenorientierte Systeme erlauben eine effiziente Verwendung von Ressourcen, da Komponenten beim Ausbleiben von Nachrichten vollständig inaktiv bleiben können.

Verschiedenste **Reaktive Erweiterungen von Programmiersprachen** (oft als Rx abgekürzt) erfahren daher im Rahmen ereignisbasierter reaktiver Architekturen steigende Beliebtheit (Bainomugisha u. a. 2013), da es diese Bibliotheken ermöglichen, die Details der asynchronen Eventabwicklung zu vereinfachen. So lassen sich etwa mehrere Requests asynchron starten. Anschließend beobachtet man das Eintreffen von Antwortevents der Downstream-Services, die in einem Observable auftauchen, und reagiert einfach nur noch mittels einer Sequenz von Transformationen (Operator-Chain) darauf. Rx-Bibliotheken in Programmiersprachen sind daher ein hilfreicher Mechanismus, mit dem die Ergebnisse mehrerer Events zusammengeführt und Operationen auf diesem Event-Stream ausgeführt werden können (siehe Bild 12.6). Im Kern kehrt Rx traditionelle Kontrollflüsse um. Anstatt nach Daten zu fragen (und auf die Antwort zu warten), beobachtet man mittels Observables das Ergebnis einer Operation und reagiert, wenn sich etwas ändert. Bei Rx-Implementierungen können aus der funktionalen Programmierung bekannte Funktionen wie beispielsweise `filter()`, `map()` und `reduce()` (und viele mehr) auf solchen Observables ausgeführt werden.

Auf der Website zum Buch finden sich Labs, die die ereignisbasierte Kopplung praktisch vertiefen. Der Leser findet ferner Links zum ReactiveX-Programmiermodell und eine Übersicht unterstützter Programmiersprachen.

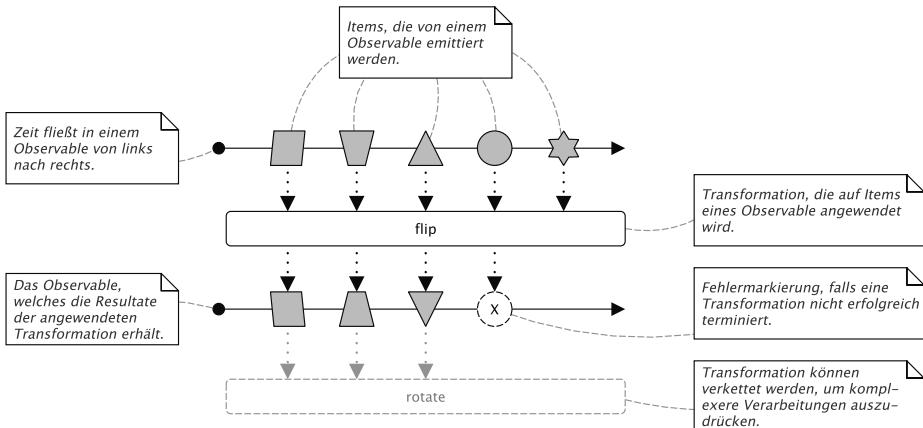


Bild 12.6 Reaktive Verarbeitung von Ereignissen (Rx)

12.2.5 API-Versioning

Die API eines Services sollte möglichst stabil sein. Im Unternehmensumfeld möchte man Änderungen insbesondere an Legacy-Client-Anwendungen vermeiden. Betreiber einer öffentlichen Web-API müssen bei Updates ebenfalls vorsichtig vorgehen. Wenn z. B. Twitter seine REST-API ändert würde, wären Millionen von Client-Installationen zu aktualisieren. Dennoch kann man Änderungen an APIs in evolutionären Systemen nicht ausschließen und nie ganz vermeiden. In diesem Fall kann man sich des sogenannten API-Versionings bedienen, das die Kompatibilität zwischen Clients und serverseitigen API-Versionsständen optimieren soll. Ganz allgemein sind zwei Services S1 und S2 zueinander kompatibel, falls sie sich gegenseitig austauschen lassen, ohne dass Consuming-Services dies bemerken würden. Kompatibilität kann jedoch auch als ein historischer Zusammenhang verstanden werden. Wenn z. B. der Dienst S2 durch Änderung von S1 entstand, spricht man von Abwärts- und Vorwärtskompatibilität (siehe auch Bild 12.7). Angenommen, es wurde ein Consuming-Service C1 für S1 und später ein weiterer Consuming-Service C2 für S2 entwickelt. Wenn C2 auch mit S1 funktioniert, dann ist C2 **abwärtskompatibel**, und S1 ist **vorwärtskompatibel**. Wenn umgekehrt C1 mit S2 funktioniert, dann ist C1 aufwärts- und S2 abwärtskompatibel (siehe auch Bild 12.7).

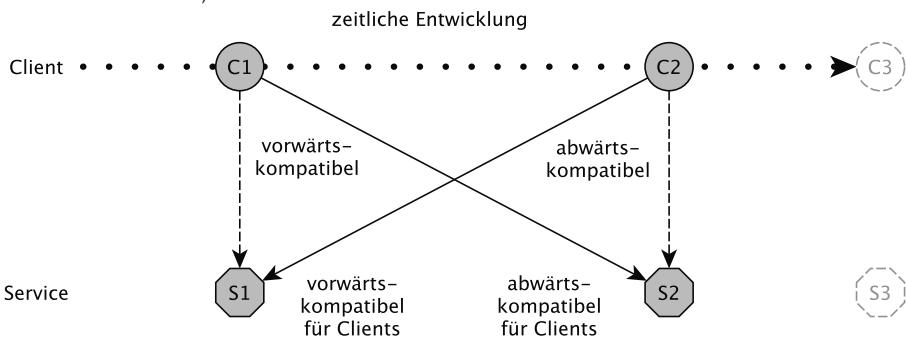


Bild 12.7 Vorwärts- und Abwärtskompatibilität

Nachrichten mit Datenformaten wie XML, JSON oder Protocol Buffers lassen sich zum Beispiel um neue optionale Felder erweitern. Weil die Felder optional sind, muss ein älterer Consuming-Service sie nicht nutzen, wenn er eine Nachricht an einen Server schickt. Die API des Servers ist in diesem Fall **abwärtskompatibel**.

Eventuell antwortet der Providing-Service dem Consuming-Service mit einer Nachricht und verwendet dabei Felder, die der Consuming-Service nicht kennt. Falls der Consuming-Service diese Nachricht trotzdem akzeptiert, ist er **vorwärtskompatibel**. Der Client ignoriert die unbekannten Felder, sollte sie aber nicht aus dem erhaltenen Dokument entfernen, falls weitere dokumentenzentrierte Interaktionen mit dem Server folgen.

Eine API sollte grundsätzlich stabil sein. Falls trotzdem Änderungen zum Umsetzen neuer Anforderungen notwendig sind, ist es besser, die Änderungen auf die beschriebene abwärts- und vorwärtskompatible Art und Weise durchzuführen. Ist eine inkompatible Änderung unausweichlich, wird meist zur Veranschaulichung ein neuer Versionsidentifikator im Rahmen eines API-Versionings eingeführt. Bekannt ist hier beispielsweise das Semantic Versioning mit dem Versionierungsschema **MAJOR.MINOR.PATCH** (siehe auch Bild 12.8):

- Die MAJOR-Version wird bei inkompatiblen API-Änderungen inkrementiert. Man spricht dann auch von einem **Breaking-Change**.
- Die MINOR-Version kommt zum Einsatz, wenn die API um zusätzliche Funktionen erweitert wird, aber insgesamt abwärtskompatibel bleibt. Dies wird auch als **Feature Release** bezeichnet.
- Die PATCH-Version wird ausschließlich für abwärtskompatible Fehlerkorrekturen verwendet. Man spricht in diesem Zusammenhang auch von einem **Fix**.

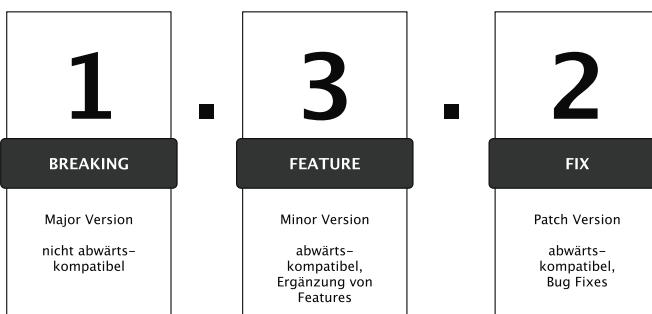


Bild 12.8
Semantic Versioning

Selbst bei sehr kleinen inkompatiblen Änderungen ist also immer die MAJOR-Version hochzuzählen. Semantic Versioning zeigt generell nicht an, wie umfangreich eine Änderung ist oder wie hoch der Migrationsaufwand für einen Client bei einem Umstieg wäre. Man kann auch nicht voraussetzen, dass ein Consuming-Service, der für Version 1.2.3 entwickelt wurde, auch mit der Version 1.1.9 läuft, da die Versionen 1.1.x ja dennoch weniger Funktionsumfang bieten als die Version 1.2.x.

Insbesondere bei so versionierten REST-APIs ist der Version-in-der-URL-Ansatz weit verbreitet und leicht nachvollziehbar. Allerdings ist dieser Ansatz streng genommen nicht REST-konform, da sich ein Pfadbestandteil nicht auf die Ressource, sondern die API bezieht:

- <https://my.service.io/v1.0.1/ressource>
- <https://my.service.io/v1.2.0/ressource>

Ein Versionsidentifikator in der URL macht für eine REST-API streng genommen keinen Sinn. Zweck einer URL – beziehungsweise eines URI – ist die eindeutige Identifikation einer Ressource. Wird mit dem Versionsidentifikator tatsächlich eine versionierte Entität identifiziert, ist dies REST-konform. Was dann allerdings gemäß REST versioniert wird, ist die Ressource und nicht die API!

Der Version-in-der-URL-Ansatz passt jedoch zu APIs mit Remote Procedure Calls (RPC). Eine versionierte API kann beispielsweise bei mehreren Backend-Servern umgesetzt werden, die jeweils eine API-Version unterstützen. Ein API-Gateway, das zwischen Client und Server steht, könnte anhand der URLs die Requests zum jeweiligen Backend-Server weiterleiten. Der Versionsidentifikator lässt sich auch als Query-Parameter, Urlform-encoded-Parameter oder applikationsspezifischer Header übergeben. API-Gateways unterstützen in der Regel alle genannten Alternativen (siehe auch Abschnitt 12.6.2).

Bei API-Änderungen ist insbesondere Vorsicht bei rein semantischen Änderungen der API geboten. Problematisch sind immer semantische Veränderungen, die nicht auch an eine Syntaxänderung gekoppelt sind. Angenommen, das Feld „Amount“ wird zur Angabe von Beträgen in Euro verwendet. Nach einer Änderung der API könnte ein neues Währungsfeld hinzugefügt werden, sodass die API flexibler wird, denn nun ließen sich auch mehrere Währungen verwenden. Ältere Clients, die das neue Währungsfeld nicht kennen, gehen jedoch weiterhin davon aus, dass im „Amount“-Feld ausschließlich Beträge in Euro angegeben sind. Eventuelle Integrationsfehler könnten unbemerkt bleiben, weil ältere Clients noch immer das „Amount“-Feld vorfinden und die Änderungen daher nicht syntaktisch erkennen können.

Semantische Veränderungen sollten daher auch immer syntaktisch signalisiert werden, beispielsweise durch ein zweites Amount-Feld. Das ursprüngliche Feld mit Euro-Beträgen bleibt für die älteren Clients erhalten und wird in der Dokumentation als veraltet („deprecated“) markiert.

Man sieht, dass API-Versioning in den Details komplex werden kann. Grundsätzlich sollte man daher API-Versioning als ein **Mittel der letzten Wahl** ansehen. Wenn API-Versioning vermieden werden kann, sollte man es vermeiden. Insbesondere wenn Consuming-Service und eine Upstream-Service-API vom selben Team entwickelt und betrieben werden, besteht kein wirklicher Grund, die API zu versionieren. Die Komplexität, die Versionierung mit sich bringt, kann man in diesem Fall vermeiden. Schwierig wird es, falls Consuming-Services (oder sonstige Clients) existieren, deren Aktualisierung man nicht beeinflussen kann oder möchte. Als Betreiber einer öffentlichen API kennt man häufig gar nicht alle Consuming-Services (insbesondere nicht bei Public APIs) oder kann zumindest nicht erwarten, dass sie zu einem bestimmten Zeitpunkt aktualisiert werden. Typischerweise werden daher APIs in der Produktentwicklung versioniert, weil die Entwicklung über einen längeren Zeitraum erfolgt und weil je nach Art des Produktes viele Installationen existieren können. Dennoch sollte eine API möglichst stabil sein. Falls Änderungen trotzdem notwendig sind, dann sollten sie abwärts- und vorwärtskompatibel umgesetzt werden. Die Einführung eines neuen Versionsidentifikators bei einer Service-API sollte eine Ausnahme sein, die aber manchmal notwendig ist, um Consuming-Services auf inkompatible Änderungen hinzuweisen.

■ 12.3 Architekturelle Sicherheit

Je mehr ein Service davon abhängt, dass ein anderer aktiv ist, desto stärker wirkt sich die Service-Verfügbarkeit des einen auf die Fähigkeit des anderen aus, seine Arbeit zu erledigen. Unter den Resilience Patterns ist Isolation ein zentrales Leitmotiv. Häufig ist es möglich, mittels einer losen und gepufferten Integration Services so zu koppeln, dass ein Upstream-Service durchaus offline geschaltet werden kann, ohne dass ein Downstream-Service von geplanten oder ungeplanten Ausfällen betroffen wird.

Eine Folge der Verwendung von Services als Komponenten ist, dass Anwendungen so konzipiert werden müssen, dass sie den **Ausfall von Services tolerieren** können. Jeder Service-Request könnte aufgrund der Nichtverfügbarkeit eines Services fehlschlagen, und der Client muss darauf so fehlerisolierend wie möglich reagieren. Dies ist ein Nachteil im Vergleich zu einem monolithischen Design, da es zusätzliche Komplexität einführt, um damit umzugehen. Die Folge ist, dass man beim Microservice-Design systematisch Service-Ausfälle zu berücksichtigen hat. Da Services jederzeit ausfallen können, ist es wichtig, die Ausfälle schnell zu erkennen und – wenn möglich – den Service automatisch wiederherzustellen. Bei Microservice-Anwendungen wird daher viel Wert auf die Echtzeitüberwachung der Anwendung gelegt, wobei sowohl architektonische Elemente (wie viele Anfragen pro Sekunde erhält die Datenbank) als auch geschäftsrelevante Metriken (z. B. wie viele Bestellungen pro Minute eingehen) überprüft werden.

In diesem Zusammenhang haben sich mehrere Stabilitätsmuster etabliert, die dabei helfen, die Erkennung von Fehlern zu beschleunigen und die Ausbreitung von Fehlern durch Isolation einzudämmen. Diese Arten von Mustern können teilweise über sogenannte Service-Meshs nachträglich in die Service-to-Service-Kommunikation bestehender Anwendungen „injiziert“ werden (mehr dazu in Kapitel 13, insbesondere Abschnitt 13.3). Übliche Stabilitätsmuster sind dabei u. a. die nachfolgend erläuterten.

12.3.1 Circuit-Breaker

Das Circuit-Breaker-Muster dient in der Softwareentwicklung dazu, wiederkehrende Verbindungsfehler zu einem Upstream-Service zu entdecken und den Zugriff zu der Ressource für eine vorgegebene Zeit zu blockieren. Circuit-Breaker können ferner dazu verwendet werden, Funktionalität der Anwendung zeitweise zu deaktivieren. Das Konzept leitet sich von der elektrischen Sicherung ab und sollte insbesondere Upstream-Services vor Überlastsituationen schützen. Die Grundidee ist dabei, einen geschützten Funktionsaufruf über einen Proxy laufen zu lassen, der auf Fehler überwacht (siehe Bild 12.9). Sobald die Fehler einen bestimmten Schwellenwert erreichen, löst die Sicherung aus, und alle weiteren Anrufe kehren mit einem Fehler zurück, ohne dass der Request an den eigentlichen Upstream-Service geht, um diesen insbesondere vor Überlast in Restart-Situationen zu schützen. Erst nach einer Recovery-Periode wird versucht, den Upstream-Service wieder anzusprechen.

Für jeden Upstream-Service wird meist eine eigene Sicherung vorgesehen, sodass der Zugriff auf andere Services weiterhin möglich ist, auch wenn der Zugriff auf einen bestimmten Service ausgefallen ist. Zudem wird eine Sicherung meist mit einem Bulkhead (siehe Abschnitt 12.3.2) kombiniert, damit für jeden Service ein eigener Thread-Pool zur Verfügung steht.

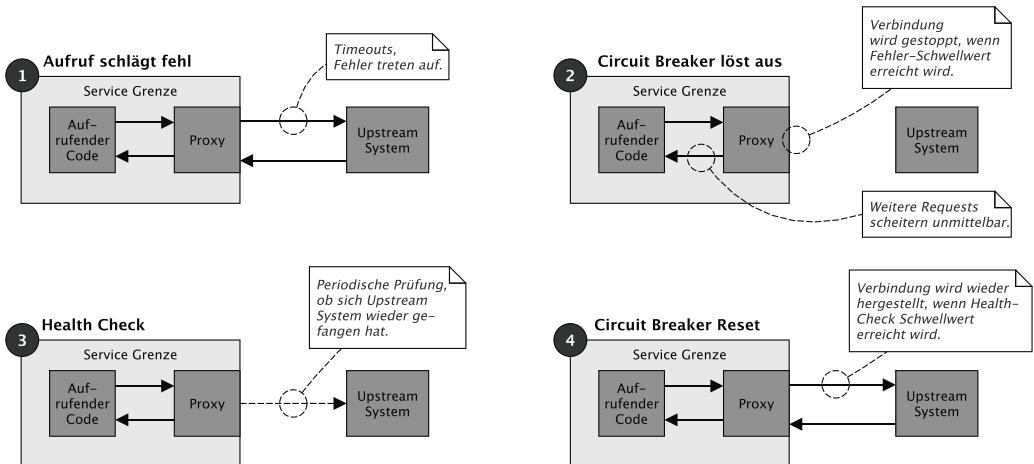
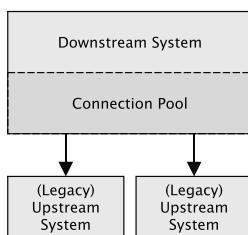


Bild 12.9 Circuit-Breaker-Stabilitätsmuster

12.3.2 Bulkhead

Ein Bulkhead ist ein weiteres Stabilitätsmuster in der Softwareentwicklung. Der Name leitet sich vom Schott (englisch: bulkhead) eines Schiffes ab. Beim Bulkhead wird ein Softwaresystem in mehrere Teilsysteme unterteilt. Falls eines der Teilsysteme von einem Service überlastet wird, schlägt dies nicht auf die anderen Teilsysteme durch, sodass diese weiterhin zur Verfügung stehen. Bild 12.10 zeigt dies am Beispiel von Connection-Pools und Upstream-Services. Im Sinne eines Bulkhead Musters sollten Connection-Pools nicht über mehrere Upstream-Services gepoolt werden, sondern es sollte pro Upstream-Service jeweils ein eigener Connection-Pool verwendet werden, um etwaige Nichterreichbarkeiten von Upstream-Services isolieren zu können. Fällt nämlich ein Upstream-Service aus, könnte ein gemeinsamer Connection-Pool gegebenenfalls von einem fehlerhaften Service gebunden werden und so auch Verbindungen zu eigentlich voll funktionsfähigen Upstream-Services verhindern.

Gemeinsamer Connection Pool



Bulkhead

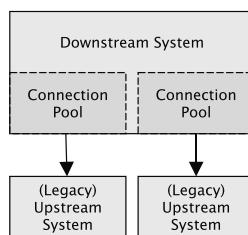


Bild 12.10
Bulkhead-Stabilitätsmuster

12.3.3 Idempotente API-Operationen

Man bezeichnet eine Operation op als idempotent, wenn $op(op(x)) = op(x)$ für alle x gilt. So ist z. B. die Betragsfunktion $|x|$ idempotent, denn $\|x\| = |x|$ gilt für alle x in \mathbb{R} . Beim REST-Paradigma (siehe Abschnitt 12.2.3) müssen beispielsweise die Methoden GET, HEAD, PUT und DELETE laut HTTP-Spezifikation idempotent sein. Das bedeutet, dass das mehrfache Absenden der gleichen Anforderung sich nicht anders auswirkt als ein einzelner Aufruf. Idempotente Operationen haben den Vorteil, dass bei einer Wiederholung von Requests oder Messages – wie nach einem Restart eines Services – Resultate erfolgreich verarbeiteter Operationen unverändert bleiben und Resultate nicht erfolgreicher Operationen wiederhergestellt werden. Diese Eigenschaft macht es Recovery-Operationen natürlich deutlich einfacher. Eine idempotente Operation einer Schnittstelle stellt also im besten Fall verlorene Daten wieder her und hat im schlimmsten Fall einfach nur keinen Effekt, aber es werden niemals Daten durch alte Daten überschrieben!

■ 12.4 Skalierung von Microservices

Die Skalierbarkeit einer Anwendung kann u. a. an der Anzahl der Requests oder Ereignisse gemessen werden, die gleichzeitig verarbeitet werden können. Der Punkt, an dem eine Anwendung zusätzliche Requests oder Ereignisse nicht mehr effektiv verarbeiten kann, bezeichnet man als die Grenze ihrer Skalierbarkeit. Diese Grenze wird erreicht, wenn mindestens eine kritische Hardwareressource aufgebraucht ist und daher mehr Ressourcen für die Verarbeitung benötigt werden (CPU, Memory, Disk, Network etc.). Dies kann grundsätzlich auf zwei Arten erfolgen.

- Unter **horizontaler Skalierung** versteht man die Skalierung durch Hinzufügen weiterer Maschinen zu einem Ressourcenpool (Scaling out),
- während **vertikale Skalierung** die Skalierung durch Hinzufügen von mehr Leistung (z. B. CPU, RAM) zu einer vorhandenen Maschine bezeichnet (Scaling up).

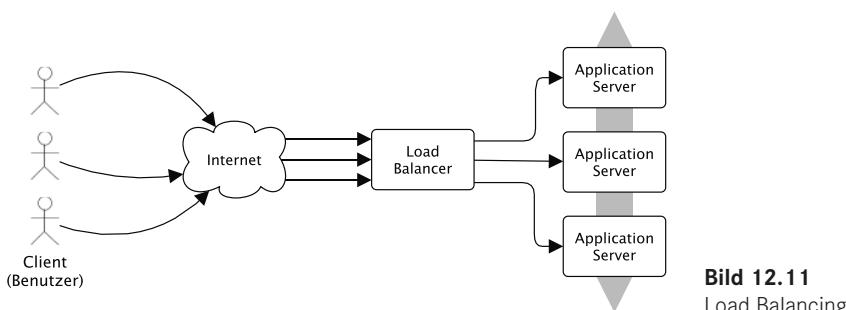
Einer der grundlegenden Unterschiede zwischen den beiden Skalierungsformen besteht darin, dass für die horizontale Skalierung Programmlogik so in kleinere Teile zerlegt werden muss, damit diese Logik auf mehreren Maschinen parallel ausgeführt werden kann. In vielerlei Hinsicht ist die vertikale Skalierung somit aus einer Software-Entwicklungsperspektive einfacher, da sich die Logik nicht wirklich ändern muss. Man führt lediglich denselben Code auf Rechnern mit höherer Leistung und mehr Ressourcen aus. Vertikale Skalierung ist aber limitiert (insbesondere bei Big Data, geografischer Distribution, Vermeidung von Single Point of Failures etc.), spätestens wenn Datenumfänge verarbeitet werden müssen, die die physische Größe einer auch großzügig dimensionierten Einzelmaschine übersteigen. Dies ist insbesondere bei Big Data-Problemen der Fall.

Microservice-basierte Architekturen setzen dabei primär auf **horizontale Skalierung**. Kubernetes unterstützt diese horizontale Skalierung durch HPA (siehe Abschnitt 9.3.5) oder auch durch ereignisbasierte Autoskalierung (siehe Abschnitt 10.3). Aufgrund dessen haben

sich mehrere auf horizontale Skalierbarkeit fokussierte Skalierungspattern in Microservice-Architekturen etabliert. Diese sollen nachfolgend betrachtet werden.

12.4.1 Load Balancing

Unter Lastausgleich (engl. Load Balancing) versteht man die effiziente Verteilung des eingehenden Netzwerkverkehrs auf eine Gruppe von Back-End-Servern. Ein Load-Balancer fungiert dabei als eine Art „Verkehrspolizist“, der Requests an mehrere Server verteilt, um deren Geschwindigkeit und Auslastung zu optimieren (siehe Bild 12.11). Fallen Server aus, leitet der Load-Balancer den Datenverkehr an die verbleibenden Server um und kann so Ausfälle einzelner Server nach außen hin transparent kompensieren. Werden neue Server hinzugefügt, beginnt der Load-Balancer automatisch, eingehende Requests auch an diese zu senden, und verteilt so die Last.



Die Aufgaben eines Load-Balancers umfassen somit

- die effiziente Verteilung von Requests an mehrere Server mit dem Ziel, die Last über alle Server auszugleichen (Lastausgleich),
- die Optimierung der Gesamtverfügbarkeit und Zuverlässigkeit, indem Anforderungen nur an Server gesendet werden, die verfügbar sind,
- und das Ermöglichen einer horizontalen Skalierung, indem Server nach Bedarf hinzugefügt oder entfernt werden können.

Das Service-Konzept von Kubernetes (siehe Abschnitt 9.3.6) ermöglicht exakt diese Funktionen für Pods.

12.4.2 Messaging

Im Rahmen der ereignisbasierten Integration von Services haben sich im Wesentlichen zwei Nachrichtenaustauschpattern etabliert, die von nahezu allen Messaging-Systemen unterstützt werden.

Beim **Queueing** werden Nachrichten von Erzeugern (Producer) in der Warteschlange gespeichert, bis sie verarbeitet und gelöscht werden (siehe Bild 12.12). Jede Nachricht wird von

einem einzelnen Verbraucher (Consumer) immer nur einmal verarbeitet. Mithilfe von Nachrichtenwarteschlangen können umfangreiche Verarbeitungsprozesse entkoppelt, Aufgaben gepuffert oder im Stapelbetrieb verarbeitet und hohe Workloads entschärft werden. Laufen Warteschlangen voll, können zusätzliche Ressourcen zugeschaltet werden, um die Warteschlangen wieder abzubauen. Laufen Warteschlangen leer, können Ressourcen abgeschaltet werden, um den Ressourcenverbrauch zu minimieren. Das Queueing-Pattern eignet sich also hervorragend für einen Lastausgleich bei ereignisbasierten Integrationen von Services.

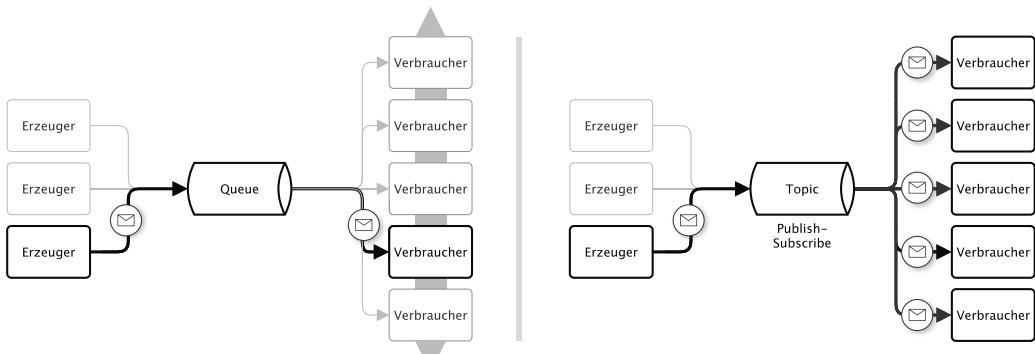


Bild 12.12 Messaging-Pattern

Solche Nachrichtenwarteschlangen ermöglichen es somit unterschiedlichen Teilen eines Systems, asynchron miteinander zu kommunizieren und Vorgänge gepuffert zu verarbeiten. Viele Erzeuger und Verbraucher können die Warteschlange nutzen, aber jede Nachricht wird nur einmal von einem einzelnen Verbraucher verarbeitet. Aus diesem Grund wird dieses Messaging-Muster häufig auch als 1-zu-1- oder Punkt-zu-Punkt-Kommunikation bezeichnet. Nachrichtenwarteschlangen können auch persistiert werden, sodass selbst bei Ausfällen Ereignisabläufe wieder ab- und in Verbraucher eingespielt werden können.

Beim **Publish/Subscribe**-Pattern wird hingegen jede veröffentlichte Nachricht einem Topic zugeordnet und sofort von allen Abonnenten des Topics empfangen (siehe Abschnitt 12.12). Auch Pub/Sub-Messaging kann verwendet werden, um ereignisgesteuerte Architekturen zu ermöglichen und so Services voneinander zu entkoppeln. Auf diese Weise sind Kenntnisbeziehungen zwischen Services nicht erforderlich, sondern nur zu einem zentralen Pub/Sub-Service. Da allerdings alle Verbraucher (Subscriber) eines Topics dieselben Nachrichten verarbeiten, ist ein Lastausgleich über mehrere Service-Instanzen nur eingeschränkt möglich. Das Pub/Sub-Modell eignet sich vielmehr, um Resultate von Erzeugern (Publishern) zu serialisieren und an interessierte Verbraucher (Subscriber) weiterzuleiten. Sollen serialisierte Resultate wieder einem Lastausgleich unterworfen werden, wäre eine Option, alle Nachrichten eines Topics von einem Verbraucher (Subscriber) in eine Queue weiterzuleiten. Queueing und Pub/Sub können also durchaus miteinander kombiniert werden.

12.4.3 Skalierung zustandsbehafteter Komponenten

Die horizontale Skalierung zustandsloser (stateless) Services ist relativ einfach, da man keinen Zustand zwischen den Service-Instanzen synchronisieren muss. Insofern sollten Microservices – wenn möglich – zustandslos konzipiert werden. Auch der Fokus der horizontalen Skalierbarkeit mittels Prozessen, der bereits in Abschnitt 8.5.5 behandelten 12-Faktor-Methodologie, geht exakt in diese Richtung.

Dennoch wird man natürlich in den meisten Anwendungen nicht vermeiden können, an irgendeiner Stelle einen Zustand speichern zu müssen. Nahezu alle Systeme beinhalten daher normalerweise Datenbanken oder datenbankähnliche Komponenten wie beispielsweise Dateisysteme. Doch auch solche extrem zustandsbehafteten Komponenten (Stateful-Services) müssen skalierbar sein. Verschiedene Arten von Datenbanken bieten unterschiedliche Formen der Skalierung. Insbesondere NoSQL-Datenbanken haben hier die Möglichkeiten enorm erweitert. All diesen Überlegungen liegt letztlich das CAP-Theorem von (Fox und Brewer 1999) und (Gilbert und Lynch 2002) zugrunde. Dieses besagt im Kern, dass man von den Eigenschaften Strict Consistency (Konsistenz), Availability (Verfügbarkeit) und Partition Tolerance (Partitionstoleranz) nur zwei zeitgleich in einem verteilten System sicherstellen kann. Stateful-Services können somit entweder CA-, CP- oder AP-Systeme zur Zustandshaltung nutzen (siehe Bild 12.13).

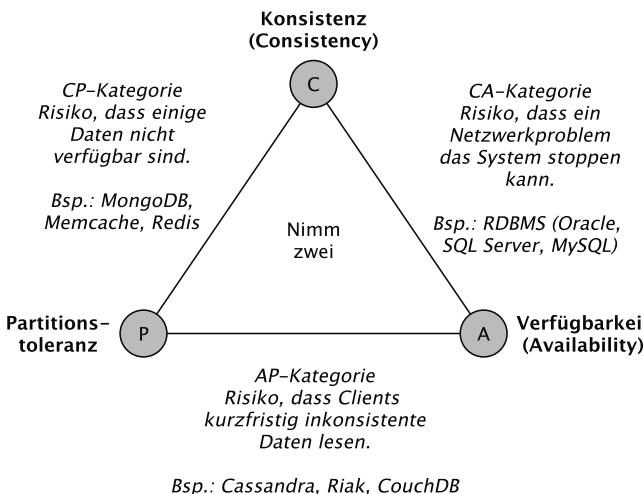


Bild 12.13

CAP-Theorem

In vielen Anwendungskontexten ist dabei die Verfügbarkeit gesetzt, und man muss dann entweder die strikte Konsistenz (also BASE statt ACID) oder die Partitionstoleranz aufgeben. BASE steht dabei für Basically Available, Soft State und Eventual Consistency. Anstatt sofortige Konsistenz zu erzwingen (wie es etwa das ACID-Modell relationaler Datenbanksysteme vorsieht), stellen BASE-basierte NoSQL-Datenbanken die Verfügbarkeit von Daten sicher, indem sie diese über die Knoten des Datenbank-Clusters verteilen und replizieren. Aufgrund der fehlenden sofortigen Konsistenz können sich Datenwerte somit aus Sicht eines Datenbanknutzers im Laufe der Zeit (z. B. während zwei direkt aufeinanderfolgenden Datenbankabfragen) ändern. Das BASE-Modell bricht mit dem Konzept einer Datenbank, die ihre eigene Konsistenz erzwingt, und delegiert diese Verantwortung an die Entwickler.

Die Tatsache, dass BASE keine sofortige Konsistenz erzwingt, bedeutet allerdings nicht, dass diese nie erreicht wird. BASE-Datenbanken erlauben allerdings bereits Reads, solange die Datenbank noch nicht sicherstellen kann, dass neue bzw. geänderte Daten bereits auf allen Knoten der verteilten Datenbank erfolgreich hinterlegt wurden.

Dabei spielt natürlich eine große Rolle, ob es mehr Read- oder mehr Write-Requests gibt. Meist überwiegen Read-intensive Anwendungsfälle, und Writes können besser an Einzelknoten kanalisiert werden. Infolgedessen versucht man mit entsprechenden Pattern, dieses Ungleichgewicht geeignet zu berücksichtigen.

12.4.3.1 Scaling for Reads

Viele Dienste werden überwiegend gelesen. Selbst in relationalen Datenbanksystemen (z. B. MySQL, Postgres) können Daten von einem Primärknoten auf ein oder mehrere Replikate kopiert werden. Dies geschieht häufig, um sicherzustellen, dass eine Kopie der Daten sicher aufbewahrt wird. Man kann dies jedoch auch zum Load Balancing von Lesevorgängen verwenden (siehe Bild 12.14). Alle Schreibvorgänge können hierfür an einen einzelnen Primärknoten gehen, aber Lesevorgänge kann man über mehrere Lesereplikate verteilen. Die Replikation von der Primärdatenbank auf die Replikate erfolgt irgendwann nach dem Schreiben. Dies bedeutet, dass bei dieser Technik beim Lesen manchmal veraltete Daten angezeigt werden, bis die Replikation abgeschlossen ist. Dieses Verhalten wird als eventual consistent bezeichnet. Wenn solche vorübergehenden Inkonsistenzen in Anwendungsfällen folgenlos bleiben (und dies ist häufig der Fall), ist dies eine recht einfache und übliche Methode, um Stateful-Services zu skalieren.

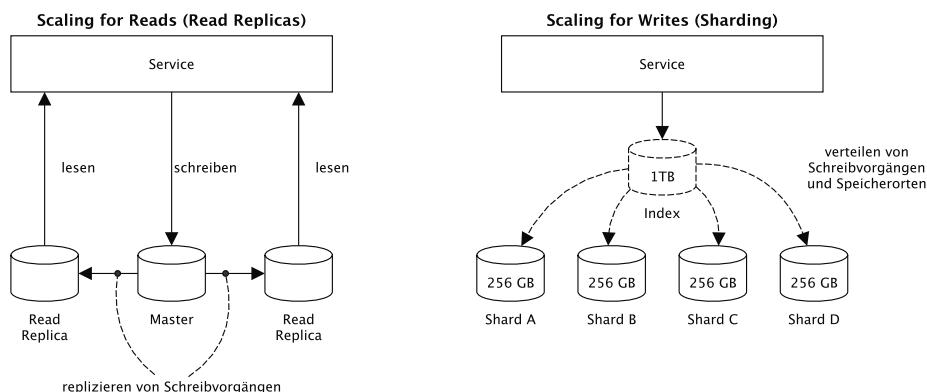


Bild 12.14 Skalierung zum Zwecke der Lese- und Schreiboptimierung

12.4.3.2 Scaling for Writes (Sharding)

Sharding ist eine Methode der Datenbankpartitionierung. Dabei wird ein Datenbestand in mehrere Teile aufgeteilt und jeweils von einer eigenen Serverinstanz verwaltet (siehe Bild 12.14). Mittels Sharding können somit auch umfangreiche Datenmengen verwaltet werden, welche die Kapazitäten eines einzelnen Servers sprengen würden. Da die einzelnen Teile von einer eigenen Serverinstanz verwaltet werden, werden nicht nur die Daten selbst, sondern auch die dafür benötigte Rechenleistung aufgeteilt. Die Methode der Aufteilung, die sogenannte Partitionsstrategie, spielt dabei eine entscheidende Rolle.

Der größte Nachteil des Shardings ist natürlich, dass ein Zugriff über andere Kriterien als das Aufteilungskriterium unverhältnismäßig aufwendig ist. Abfragen über andere Kriterien oder Joins müssen auf mehrere Server aufgeteilt werden. Das Datenmodell und die Zugriffspfade müssen so entworfen werden, dass derartige Zugriffe nur selten oder gar nicht vorkommen, sonst werden die Vorteile des Shardings zunichtegemacht. Diese Notwendigkeit macht Sharding zu einer attraktiven Methode vor allem für NoSQL-Systeme, bei denen typischerweise Joins ohnehin nicht unterstützt werden und auch Zugriffe über Sekundärschlüssel eher eine Ausnahme sind. Somit lässt sich Sharding für diese Systeme vergleichsweise einfach implementieren.

Bei vielen NoSQL-Systemen wird zudem Sharding und Replikation verknüpft, etwa bei Cassandra und Riak. Dabei wird z. B. in einer logischen ringförmigen Anordnung der Server jeder Shard auch gleichzeitig auf einen oder mehrere nachfolgende Server im Ring repliziert. Somit hält jeder Server jeweils eine Kopie mehrerer Shards. Durch diese Anordnung und Synchronisierungsmechanismen entstehen sehr flexible und fehlertolerante Systeme, welche auch mit sehr großen Datenmengen befüllt werden können.

12.4.3.3 Command Query Responsibility Segregation (CQRS)

In vielen Fällen sind unterschiedliche Modelle für unterschiedliche Anforderungen eines Systems erforderlich. So gibt es beispielsweise optimierte Datenmodelle für Online-Transaktionsverarbeitung (OLTP), Online-Analytical Processing (OLAP) und Suche. Ein weiterer Grund ist die in Microservice-Architekturen häufig anzutreffende polyglotte Persistenz, bei der mehrere Datenbanken für unterschiedliche Anforderungen an den Datenzugriff verwendet werden. Ein komplexeres Cloud-natives System könnte zum Beispiel eine Dokumentendatenbank (Document Store) als operative Datenbank, einen Column-Store für Analysen/Berichte und eine Suchmaschine für die Implementierung von Suchfunktionen verwenden.

Diesen Umstand kann man sich in Microservice-Architekturen mittels des sogenannten CQRS Pattern zunutze machen. Dieses Pattern wird noch im Detail im kommenden Kapitel 14 und Abschnitt 14.3.2.3 behandelt werden. Stark zusammengefasst werden auch beim CQRS-Ansatz die Zuständigkeiten der Modelle des Systems nach ihrem Schreib-Lese-Charakter getrennt. Ein Command kann nur auf dem stark konsistenten Befehlsausführungsmodell operieren. Eine Query kann keinen der Systemzustände direkt verändern – weder die Lesemodelle noch das Befehlsausführungsmodell. CQRS basiert somit auf einem einzigen Modell zur Ausführung von Operationen, die den Zustand des Systems verändern (Systembefehle bzw. Commands). Dieses Command Execution-Modell ist das einzige Modell, das stark konsistente Daten im Sinne einer Single Source of Truth repräsentiert.

Das System kann aus dieser Single Source of Truth allerdings beliebig viele Modelle mittels Projektionen erzeugen, die für die Darstellung von Daten für Benutzer oder andere Systeme erforderlich sind. Diese Modelle sind read-only. Keine der Operationen des Systems kann die Daten der gelesenen Modelle direkt ändern. CQRS kann auf Command-Seite sehr einfach im Rahmen ereignisgesteuerter Integrationen verwendet werden. Mehr zu CQRS aber folgt in Abschnitt 14.3.2.3.

12.4.4 Caching

Die Leistungsfähigkeit von microservice-basierten Systemen lässt sich auch durch die Vermeidung unnötiger Wiederberechnungen steigern. Dies wird im Allgemeinen als Caching bezeichnet. Unter einem Cache versteht man dabei zumeist einen schnellen Pufferspeicher, der (wiederholte) Zugriffe auf ein langsames Hintergrundmedium oder aufwendige Neuberechnungen zu vermeiden hilft. Daten, die bereits einmal geladen, generiert oder berechnet wurden, verbleiben im Cache, sodass sie bei späterem Bedarf schneller aus diesem abgerufen werden können. Auch können Daten, die vermutlich bald benötigt werden, vorab vom Hintergrundmedium abgerufen und vorerst im Cache bereitgestellt werden (read-ahead).

Beim **clientseitigen Caching** speichert der Client das zwischengespeicherte Ergebnis (siehe Bild 12.15 A). Der Client kann entscheiden, wann (und ob) er eine neue Kopie abruft. Im Idealfall bietet der Upstream-Service Hinweise, die dem Client helfen zu verstehen, was mit der Antwort zu tun ist, wann und ob ein neuer Request gestellt werden muss. Das clientseitige Caching kann dazu beitragen, Netzwerkrequests drastisch zu reduzieren und so wirkungsvoll die Last auf Upstream-Services zu verringern.

Beim **serverseitigen Caching** übernimmt, wie in Bild 12.15 B gezeigt, der Server die Caching-Verantwortung und verwendet möglicherweise ein System wie Redis oder Memcache oder einen einfachen In-Memory-Cache. Beim serverseitigen Caching ist für Clients das Caching transparent. Mit einem Cache innerhalb einer Servicegrenze kann es einfacher sein, Dinge wie eine Cache-Invalidierung durchzuführen oder Cache-Treffer zu verfolgen und zu optimieren. In einer Situation, in der mehrere Arten von Clients existieren, ist ein serverseitiger Cache oft der schnellste Weg, um die Leistung des Gesamtsystems zu verbessern.

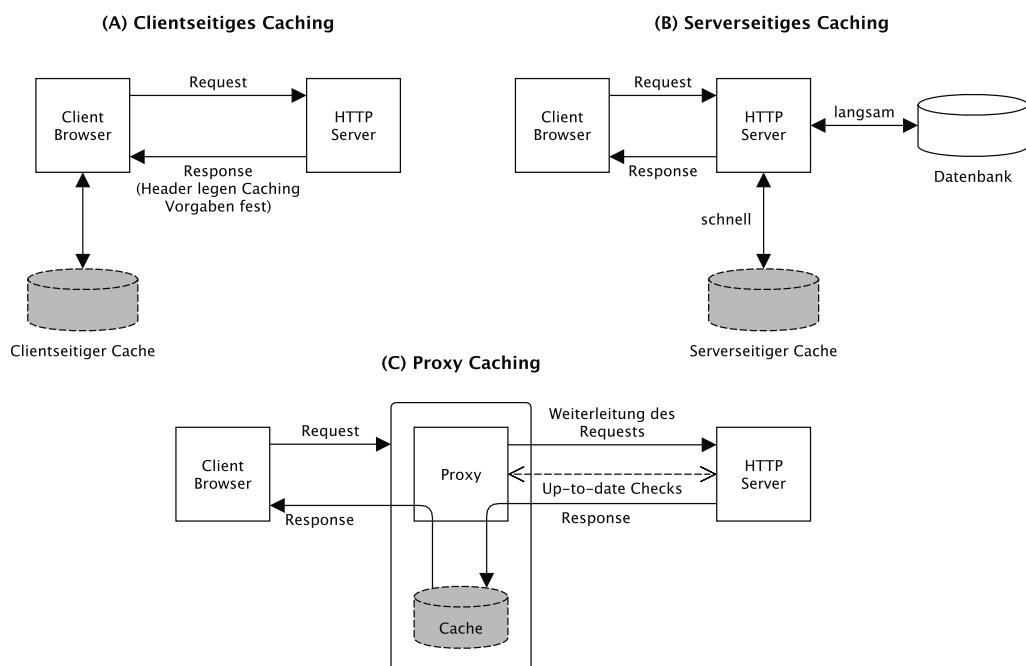


Bild 12.15 Client-, server- und proxy-basiertes Caching

Beim **Proxy-Caching** wird hingegen ein Proxy zwischen dem Client und dem Server platziert (siehe Bild 12.15 C). Ein gutes Beispiel hierfür ist die Verwendung eines Reverse-Proxy- oder Content-Delivery-Netzwerks (CDN). Beim Proxy-Caching ist das Caching sowohl für den Client als auch für den Server transparent. Proxy-Caching ist oft eine sehr einfache Möglichkeit, einem vorhandenen System Caching nachträglich hinzuzufügen. Wenn der Proxy so konzipiert ist, dass er generischen Datenverkehr zwischenspeichert, kann er auch mehr als einen Service zwischenspeichern (z. B. mittels Reverse-Proxys wie Squid). Ein Proxy zwischen Client und Server führt allerdings erst einmal zu zusätzlichen Netzwerk-Hops und damit Latenzen. Meist führt dies jedoch selten zu Problemen, da die Leistungsoptimierungen, die sich aus dem Caching selbst ergeben, die zusätzlichen Netzwerklatenzen häufig überwiegen.

■ 12.5 Prinzipien zur Entwicklung von Microservices

Microservice-Architekturen ermöglichen somit agile und evolutionäre Entwicklungen von komplexen Systemen. Insbesondere in der Cloud-basierten Geschäftswelt sind erfolgreiche Produkte oft microservice-basiert. Nicht alle microservice-basierten Produkte sind aber auch erfolgreich. Microservices sind kein Garant für Erfolg. Nach (Newman 2015) sollte man grundsätzlich die nachfolgend erläuterten Prinzipien beherzigen.

12.5.1 Prinzip 1: Bilde Modelle um Geschäftskonzepte

Die Erfahrung hat gezeigt, dass Schnittstellen, die aus fachlichen Kontexten (d. h. domain-driven) abgeleitet werden, stabiler sind als solche, die um technische Konzepte herum gebaut werden. Durch die Modellierung der Systemdomäne bildet man nicht nur stabilere Schnittstellen, sondern stellt auch sicher, dass Änderungen in Geschäftsprozessen besser abbildungbar sind. Die Methodik der Bounded Contexts des Domain Driven Designs (DDD) bietet sich an, um potenzielle Domänengrenzen zu definieren. Diese Methodik wird noch im Detail in Kapitel 14 behandelt werden.

12.5.2 Prinzip 2: Erschaffe eine Kultur der Automatisierung

Ein einzelner Microservice ist einfach, häufig schon fast trivial. Ein gesamtes System von Microservices ist jedoch meist komplex. Ein wesentlicher Grund ist die im Resultat steigende Anzahl von Wechselwirkungen des Gesamtsystems, bedingt durch wesentlich mehr klein-teilige Interaktionen zwischen vielen kleinen Microservices. Die Einführung einer Kultur der Automatisierung ist hier ein wichtiger Schritt, um diese Komplexitätsverschiebung beherrschen zu können (siehe auch Kapitel 3).

Automatisierte Tests stellen beispielsweise sicher, dass Services auch nach Updates in komplexeren Wechselwirkungsketten stets wie erwartet funktionieren. Automatisierte Deployments ermöglichen schnelles Feedback zur Produktionsqualität jedes Check-ins. Die Definition von Environments (z. B. Test, Staging, Development) ermöglicht, unterschiedliche Ausbaustufen eines Systems mit einer einheitlichen Methode bereitzustellen und zu testen (bzw. in Produktion bringen) zu können. Container-Images (siehe Kapitel 8) können die Bereitstellung beschleunigen, indem die Erstellung von Deployment Units, deren Konfiguration und deren Betrieb standardisiert und mittels Orchestrierungsplattformen automatisiert werden (siehe Kapitel 9).

12.5.3 Prinzip 3: Blende interne Implementierungsdetails aus

Damit ein Service unabhängig von anderen Diensten entwickelt werden kann, ist es wichtig, dass dessen Implementierungsdetails verborgen werden, um implizite Abhängigkeiten zu vermeiden. Services sollten auch ihre Datenbanken verbergen, um zu vermeiden, dass eine der häufigsten Arten der engen Kopplung (Datenkopplung, siehe Abschnitt 12.2.1) vermieden wird. Stattdessen sollten eher Konzepte wie event-basierte Ansätze verwendet werden (siehe Abschnitt 12.2.4), um Daten über mehrere Services hinweg bereitzustellen. Technologieunabhängige APIs ermöglichen Flexibilität hinsichtlich des Einsatzes verschiedener Technologie-Stacks. Insbesondere mittels REST kann die Trennung von internen und externen Implementierungsdetails methodisch sichergestellt werden (siehe Abschnitt 12.2.3).

12.5.4 Prinzip 4: Dezentralisiere

Um die erforderliche Autonomie von Microservice-Teams zu maximieren, sind Self-Service-Lösungen zu nutzen, um Software bei Bedarf bereitzustellen und die Entwicklung und das Testen für Teams so reibungsarm wie möglich zu halten (siehe Kapitel 3). Teams sollten Servicebesitzer und für die vorgenommenen Änderungen sowie den Betrieb von Services gleichermaßen verantwortlich sein. Teams sollten an der Organisation ausgerichtet werden, um sicherzustellen, dass das Gesetz von Conway funktioniert. Wenn übergreifende Richtlinien (z. B. im Rahmen von Architekturen) erforderlich sind, sollten diese in einem Shared-Governance-Modell entwickelt werden. Da Ansätze wie Enterprise Service Busses oder Orchestrierungssysteme leicht zur Zentralisierung von Geschäftslogik und „dummen“ Diensten führen können, sollten diese vermieden werden. Vielmehr sollten *Prefer-Choreography-over-Orchestration-* und *Dumb-Middleware-with-Smart-Endpoints*-Ansätze genutzt werden, um die zugehörige Logik und Daten innerhalb von Servicegrenzen zu halten, um eine hohe Servicekohäsion zu gewährleisten.

12.5.5 Prinzip 5: Definiere unabhängig aktualisierbare Einheiten

Bei Breaking-Changes sollten versionierte Endpunkte nebeneinander verwendet werden (siehe Abschnitt 12.2.5), damit konsumierende Services erst im Laufe der Zeit angepasst werden können. Dies ermöglicht es, die Geschwindigkeit und die Veröffentlichung neuer

Funktionen zu optimieren. Bei Verwendung der RPC-basierten Integration sollte eine eng gebundene Client- oder Server-Stub-Generierung vermieden werden (wie sie beispielsweise Java RMI fördert).

Die Verwendung eines *One-Service-per-Container*-Modells reduziert Auswirkungen, die die Aktualisierung eines Services auf andere Services haben könnte. Man sollte ferner die Verwendung von Blue/Green- oder Canary-Release-Techniken erwägen, um Deployments von Releases zu trennen. Es sollte ferner die Norm werden, Änderungen an einem einzelnen Service vorzunehmen und ihn für die Produktion freizugeben, ohne dass andere Services hierfür gesperrt werden müssen. Downstream-Service-Teams sollten selber entscheiden können, wann ihre Services aktualisiert werden müssen. Dies sollte nicht durch die Upstream-Service-Teams erzwungen werden können.

12.5.6 Prinzip 6: Isoliere Fehler

Eine Microservice-Architektur kann widerstandsfähiger sein als ein monolithisches System, aber nur, wenn Fehler als Teil des Systemdesigns von Grund auf berücksichtigt werden. Daraus sollten insbesondere Remote Procedure Calls nicht wie Local Calls behandelt werden, da dadurch andere Arten von Fehlermodi (u. a. Netzwerkfehler) ausgeblendet werden. Timeouts und Circuit-Breaker-Stabilitätsmuster sollten entsprechend eingestellt und verwendet werden, um die Auswirkungen einer fehlerhaften Komponente zu begrenzen. Teilweise lassen sich diese Mechanismen mittels Service-Meshs sogar nachträglich in Anwendungen injizieren (siehe Abschnitt 13.3). Es sollten ferner die Auswirkungen analysiert werden, die sich für den Nutzer ergeben, wenn sich nur ein Teil des Systems fehlerhaft verhält. Im Falle von Netzwerkpartitionen sollte die Frage beantwortet werden können, ob in solchen Situationen die Verfügbarkeit oder Konsistenz geopfert werden kann.

12.5.7 Prinzip 7: Baue gut beobachtbare Services

Bei Microservices reicht es nicht, das Verhalten einer einzelnen Serviceinstanz oder den Status einer einzelnen Maschine zu beobachten, um festzustellen, ob das System als Ganzes ordnungsgemäß funktioniert. Es ist vielmehr eine integrierte Sicht auf eine Cloud-native Anwendung, ihre Services sowie die Plattform und Infrastruktur, auf der diese Services laufen, erforderlich. Dabei sollten

- Logs
- Metriken
- und Traces

für eine integrierte Sicht erfasst und in einem zentralen System zum Zwecke der Beobachtbarkeit (Observability) konsolidiert werden. Oft ist dies auch aufgrund regulatorischer Anforderungen erforderlich. Wie dies im Detail realisierbar ist, wird noch in Kapitel 13 erläutert werden.

■ 12.6 Serverless-Architekturen

Neben Microservice-Architekturen werden im Cloud-native Umfeld zunehmend häufiger sogenannte Serverless-Architekturen genannt. Dies ist ein Architekturansatz, der letztlich den Microservice-Architekturansatz aufgreift und Besonderheiten berücksichtigt, die sich aus dem FaaS-Programmiermodell (siehe Abschnitt 10.1) ergeben.

Wie das Kapitel 10 gezeigt hat, sind FaaS-Plattformen lediglich Ereignisverarbeitungssysteme (siehe auch Bild 12.16). Ereignisse können z. B. über HTTP gesendet oder von weiteren Ereignisquellen (aus Cloud-Infrastrukturen) empfangen werden. Die Plattform bestimmt dann, welche Funktionen für ein Ereignis registriert sind, sendet das Ereignis an die Funktionsinstanz und wartet auf eine Antwort, um diese Antwort an den Requestor weiterzuleiten (im Falle von request-response-basiertem Triggering).

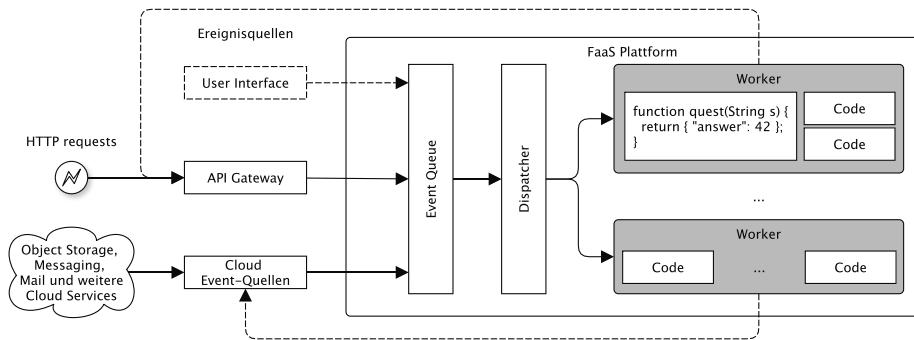


Bild 12.16 FaaS-Plattformen sind Ereignisverarbeitungssysteme.

Mit dem FaaS-Programmiermodell (siehe Abschnitt 10.1) gehen aber architekturelle Implikationen einher, die sich aus diesem Programmieransatz sehr feingranularer und ereignissteuerter Funktionen begründen. Folgerichtig werden in Abgrenzung zu Microservices diese Funktionskomponenten manchmal auch als Nanoservice bezeichnet. Die architekturellen Besonderheiten werden als Serverless-Architektur bezeichnet. Serverless-Architekturen sind letztlich nichts als eine Spezialform von Microservices, die allerdings Besonderheiten des FaaS-Programmiermodells berücksichtigen.

Unter Serverless Computing versteht man somit ein Cloud-Computing-Modell, bei dem Cloud-Provider Ressourcen ereignisgesteuert zuweisen. Entwickler von Serverless-Anwendungen müssen sich daher nicht mit Kapazitätsplanung, Konfiguration, Verwaltung, Wartung, Betrieb oder Skalierung von Containern, VMs oder physischen Servern befassen (daher auch der Name Serverless!). Serverless Computing ist zustandslos konzipiert, d. h., es werden keine Zustände in Funktionen zwischen Aufrufen gespeichert. Wenn Funktionen keine Ereignisse verarbeiten müssen, werden diesen auch keine Ressourcen (Prozessor, Hauptspeicher) zugewiesen. Die Preisgestaltung basiert auf dem Pay-as-you-go-Prinzip (siehe Kapitel 2) basierend auf der Menge tatsächlich verbrauchter Ressourcen (Prozessor, Hauptspeicher), die zur Verarbeitung von Ereignissen erforderlich sind.

Serverless Computing sollte allerdings nicht mit dezentralen Processing-Modellen wie Peer-to-Peer (P2P) oder Volunteer Computing (VC) verwechselt werden. Auch mittels Serverless Computing bereitgestellte Services bleiben (aus Downstream-Service-Sicht) konzeptionell Zentraldienste, die dem Client-Server-Modell folgen, auch wenn diese intern technisch dezentral und verteilt realisiert sein mögen.

12.6.1 Architekturelle Konsequenzen von Serverless-Limitierungen

Neben den in Kapitel 10 bereits genannten Einschränkungen und Besonderheiten des FaaS-Programmiermodells hinsichtlich Zustandslosigkeit von Funktionen, limitierten Ausführungs dauern und Latenzschwankungen aufgrund gegebenenfalls erforderlicher Kaltstarts von Funktionscontainern gibt es weitere, nicht ganz so offensichtliche Limitierungen, die jedoch nennenswerten Einfluss auf Architekturen haben.

Eine solche Limitierung ist das in Bild 12.17 gezeigte **Double-Spending-Problem**. Dieses Problem tritt auf, wenn eine Funktion f synchron eine andere Funktion g aufruft. In diesem Fall wird dem Verbraucher die Ausführung von f und g in Rechnung gestellt – obwohl nur g Ressourcen verbraucht, weil f auf das Ergebnis von g wartet.

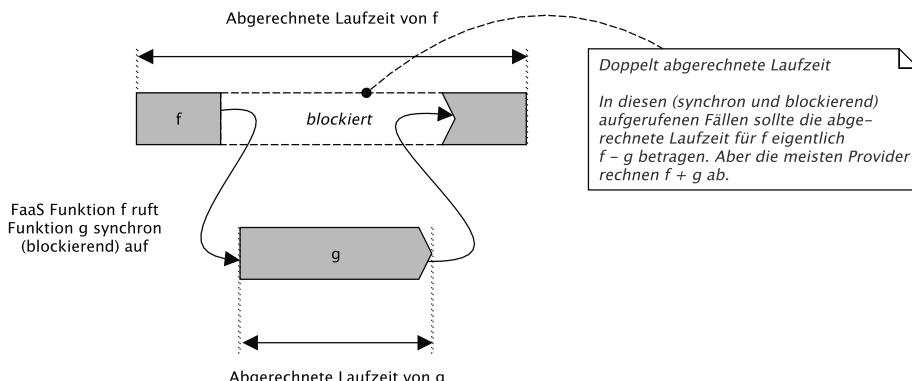


Bild 12.17 Double-Spending-Problem

Um dieses Problem der doppelten Ausgaben zu vermeiden, delegieren viele Serverless-Architekturen die Komposition von Funktionen (also die Steuerung des Kontrollflusses) an Clientanwendungen und Edge-Geräte außerhalb des Bereichs von FaaS-Plattformen. Dieses Kompositionsproblem führt somit zu neuen, stärker verteilten und dezentralisierten Formen von Cloud-native Architekturen.

Insbesondere bei Serverless-Architekturen findet man daher im Vergleich zu herkömmlichen E-Commerce-Anwendungen häufiger ein Redesign von Anwendungsarchitekturen, das oft nachhaltig bis zum Kontrollfluss durchschlägt. Serverless-Architekturen gehen dabei häufiger weiter als reine Microservice-Architekturen und integrieren noch wesentlich konsequenter Backend-Services von Drittanbietern wie Authentifizierung oder Datenbankdienste. Diese Backend-Services werden auch als **Backend as a Service (BaaS)** bezeichnet.

Die Entwicklung von Funktionen für FaaS-Plattformen wird häufig auf sehr problemspezifische, sicherheitsrelevante oder rechenintensive Funktionen beschränkt. Funktionen, die klassisch auf einem zentralen Anwendungsserver bereitgestellt würden, werden in Serverless-Architekturen als viele isolierte Funktionen bereitgestellt. Die Integration all dieser isolierten und autarken Kleinstservices als sinnvolle Endbenutzerfunktionalität wird an Endgeräte delegiert (sehr häufig in Form von nativen mobilen Anwendungen oder progressiven Webanwendungen).

Es ergibt sich damit der in Bild 12.18 gezeigte „Serverless-Effekt“, der durch folgende Punkte charakterisiert ist:

1. Querschnittslogik – wie beispielsweise Authentifizierung oder Speicherung – wird an externe Dienste von Drittanbietern delegiert.
2. Die Komposition und Kontrollflusssteuerung von Services wird auf Endbenutzer-Clients oder Edge-Geräte verlagert. Dieses Erfordernis kann durch API-Gateways (siehe auch Abschnitt 12.6.2) teilweise etwas reduziert werden.
3. Dies bedeutet, dass selbst die Downstream-Service-Orchestrierung nicht mehr vom Service-Provider selbst, sondern vom Service-Consumer über bereitgestellte Anwendungen durchgeführt wird. Solche Client- oder Edge-Geräte verbinden Services von Drittanbietern häufig direkt.
4. Endpunkte für anwendungsspezifische Funktionen werden über API-Gateways bereitgestellt. Dabei werden meist HTTP- und REST-basierte bzw. REST-ähnliche Kommunikationsprotokolle bevorzugt.

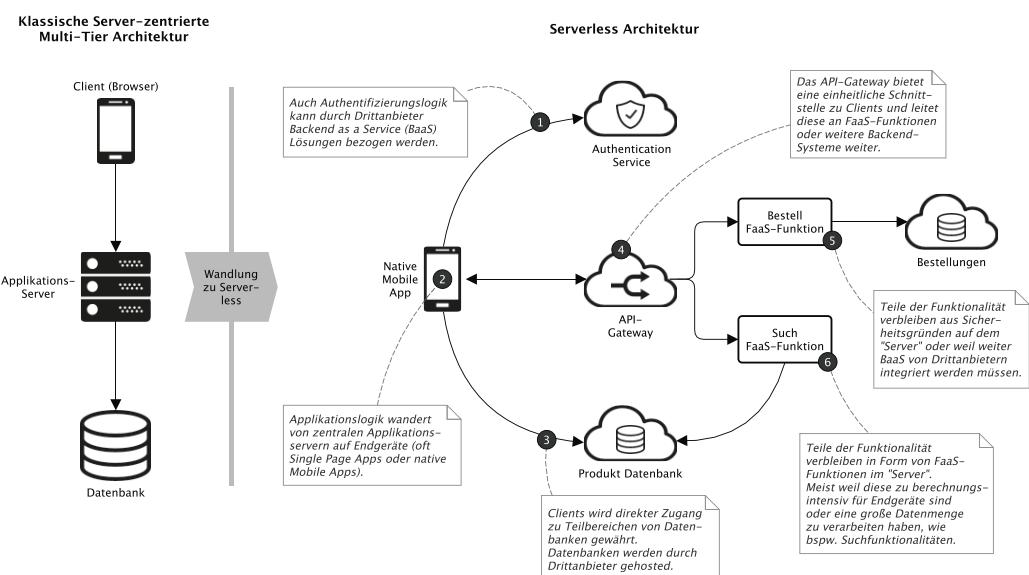


Bild 12.18 Der Serverless-Effekt auf Architekturen

In Serverless-Architekturen werden mittels FaaS-Plattformen meist nur dann anwendungsspezifische Funktionen bereitgestellt, wenn eine der folgenden Bedingungen erfüllt ist (siehe Bild 12.18, Teilbilder 5 und 6):

- Die Funktionalität ist sicherheitsrelevant und muss vom Service-Provider in einer kontrollierten Laufzeitumgebung ausgeführt werden.
- Die Funktionalität ist für Consumer-Clients oder Edge-Geräte zu processing- oder datenintensiv.
- Die Funktionalität ist so domänen-, problem- oder anwendungsspezifisch, dass einfach kein externer Drittanbieter-Service existiert.

Diese Endbenutzer-Choreografie hat zwei interessante Auswirkungen:

- Der Service-Consumer stellt jetzt die für die Steuerung des Service-Kontrollflusses erforderlichen Ressourcen bereit.
- Da die Servicekomposition außerhalb des Bereichs der FaaS-Plattform erfolgt, werden Probleme wie das Double-Spending-Problem vermieden.

12.6.2 Das API-Gateway-Pattern

API-Gateways werden mehr und mehr zu einem fundamentalen Bestandteil einer Cloud-nativen Architektur. Sie bilden einen zentralen request-response-basierten (siehe Abschnitt 12.2.3) Zugangspunkt zu den Backend-Services. Je mehr solcher Backend-Services entstehen, desto eher entsteht der Wunsch, dass Frontend-Services nicht mit Dutzenden oder Hunderten von Einzelservices interagieren müssen, sondern einen zentralen Zugangspunkt erhalten. Insbesondere das FaaS-Programmiermodell erzeugt enorm viele solcher Kleinstservices und damit den Bedarf nach einer Konsolidierung.

Durch die Einführung eines API-Gateways, das eine Grenze zu den Backend-Systemen bildet (und daher auch ab und an „Edge-Service“ genannt wird) und als zentraler Zugangspunkt dient, können diverse Orchestrationsnachteile in Serverless-Architekturen (aber auch Micro-service-Architekturen) kompensiert werden.

Zu den Nachteilen von Service-Architekturen sind u. a. viele Kommunikationsverbindungen für Clients zu nennen. Insgesamt steigt die Komplexität eines Gesamtsystems mit der Anzahl bereitgestellter APIs. Hieraus ergeben sich folgende Anforderungen:

- **Querschnittliche Lösungen** sollten nicht für jeden Backend-Service individuell umgesetzt werden müssen, sondern einheitlich über alle Services implementiert werden. Dazu zählt beispielsweise eine erste Authentifizierung, SSL-Terminierung oder das Handling von (Security-)Headern, aber auch der Schutz von APIs vor Überlastung und Missbrauch mittels Rate Limiting.
- Man möchte die Nutzung von APIs mittels **Analyse- und Überwachungstools** überwachen, um zu bestimmen für welche Zwecke bereitgestellte APIs verwendet werden.
- Wenn **APIs monetarisiert** werden sollen, muss eine Verbindung zu einem Fakturierungssystem hergestellt werden.
- In **Service-of-Services** Architekturen können zur Beantwortung einzelner Requests Dutzende von Upstream-Services abgefragt werden müssen. Dabei werden im Laufe der Zeit neue API-Services hinzukommen oder entfernt werden. Dennoch sollten all diese **Änderungen** nicht bis zum Client durchschlagen, der alle Services stets am gewohnten Platz finden sollte.

Weitere zu berücksichtigende Punkte sind u. a.:

- **Same-Origin-Policy:** Damit die Clients mit den verschiedenen Backend-Services kommunizieren dürfen, muss bei jedem Backend eine Cross-Origin-Resource-Sharing-(CORS-)Ausnahme definiert werden.
- Da Backend-Services öffentliche Endpunkte haben, sind auch interne Schnittstellen von den Clients erreichbar. Meist ist dies nicht gewollt, da dies die Angriffsfläche eines Systems erhöht.

All diese Punkte adressiert das sogenannte API-Gateway-Pattern (siehe Bild 12.19). Der Sinn und Zweck dieses Patterns ist es, die Kommunikation von Clients zu den Upstream-Services immer über das API-Gateway laufen zu lassen, um die Kommunikationskomplexität für Clients zu reduzieren und einen Punkt zu haben, an dem querschnittliche Belange konfiguriert werden können. Die Clients müssen folglich auch nur noch die Adresse des API-Gateways kennen – das Gateway leitet die Anfragen dann an den spezifischen Service weiter. Somit ist eine Hauptaufgabe eines API-Gateways das Reverse Proxying. An dieser zentralen Stelle lassen sich querschnittliche Belange umsetzen, sodass diese nicht von jedem Service separat bereitgestellt werden müssen, wie etwa einheitliche CORS-Regeln (Same-Origin-Policy) oder Authentifizierung von Nutzern.

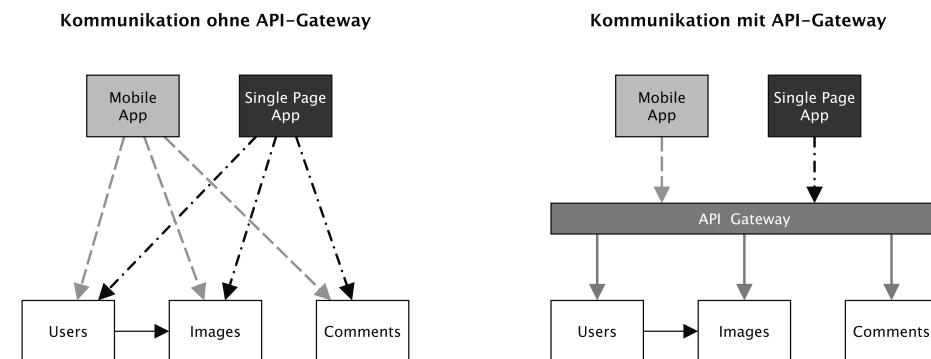


Bild 12.19 API-Gateway-Pattern

Insbesondere das Reverse Proxying eines API-Gateways ist von zentraler Bedeutung, da verschiedene Endpunkte unter einem Namen zusammengefasst werden können. Ohne Reverse Proxying müssten die in Bild 12.19 gezeigten drei beispielhaften Services *Users*, *Comments* und *Images* unter drei verschiedenen Adressen erreicht werden. Gibt es Clients, die man nicht kontrollieren kann (z. B. eine ausgerollte Mobile-App ohne Update-Zwang), ist man in den möglichen Änderungen sehr beschränkt, die man an den Endpunkten vornehmen kann.

Die zwei Kernargumente für ein Gateway sind somit die Einfachheit für Clients und die Kapselung der Service-Architektur. Hat man mehr als einen Endpunkt, ist es oft hilfreich, diese zu bündeln. Die Verwaltung von „Endpunktlisten“ in Clients sollte vermieden werden, da sie schnell unübersichtlich werden können. Da Architekturen sich gemäß der DevOps-Philosophie evolutionär entwickeln, werden sich im Laufe der Zeit auch die Endpunkte immer wieder ändern. API-Gateways vermeiden, dass Clients Änderungen direkt ausgesetzt werden, und ermöglichen es, APIs kompatibel zu halten, selbst wenn sich die Service-Landschaft sich immer wieder verändert.

Hierbei sind die folgenden Randbedingungen zu berücksichtigen:

- Ein API-Gateway ist ein konzeptioneller Single Point of Failure, der in die Architektur eingebbracht wird.
- Damit ergibt sich eine Reihe von Anforderungen an die Verfügbarkeit, Belastbarkeit und an die Fähigkeiten des Gateways.
- Ein Gateway muss mindestens so verfügbar sein wie die Höchstanforderung an einen beliebigen Service dahinter.
- Es muss auch alle Kommunikationstechnologien unterstützen, welche die aufgerufenen Services benötigen (beispielsweise HTTP/2 oder Websockets).

Einige API-Gateway-Produkte versprechen dabei unzählige Features mit entsprechender Auswirkung auf die Komplexität. Ein Austausch eines API-Gateways ist meist problemlos in Produktion möglich und für die Clients völlig transparent. Deshalb lohnt es sich, die exakten Anforderungen abzuwarten und komplexere Lösungen erst bei Bedarf einzuführen. Auch sollte man es vermeiden, Geschäftslogik oder „Features“ aus den Services in das Gateway zu verschieben. Das führt oft zu unübersichtlichen und schwer zu wartenden Systemen mit vielen Abhängigkeiten.

Einfache API-Gateways lassen sich auch mit Reverse Proxys wie NGINX realisieren. NGINX ist der Standard-Ingress-Controller in Kubernetes. Anders ausgedrückt. Ein Kubernetes Ingress kann oft als API-Gateway schon ausreichen (siehe Abschnitt 9.3.6). Auf der Website zum Buch findet sich eine Übersicht weiterer Open-Source-API-Gateway-Produkte unterschiedlichster Hersteller.

12.6.3 Abgrenzung zu Microservices

Es lässt sich bei Serverless-Architekturen beobachten, dass diese Art von Architektur dezentraler und verteilter ist, von Drittanbietern unabhängig bereitgestellte Dienste „offensiver“ nutzt und daher im Vergleich zu Microservice-Architekturen wesentlich dezentraleren und „mesh-artigen“ Charakter hat.

Serverless-Architekturen sind somit durchaus microservice-basierte Architekturentwürfe, die allerdings wesentlich konsequenter BaaS-Dienste (Backend as a Service) von Drittanbietern einbinden. Benutzerdefinierter Code wird dabei auf ein Minimum reduziert und meist auf den Logic-Tier beschränkt und in verwalteten, kurzlebigen Containern auf einer FaaS-Plattform (Functions as a Service) ausgeführt. Dadurch entfällt bei solchen Serverless-Architekturen ein Großteil der Notwendigkeit für Always-on-Komponenten. Serverless-Architekturen können so von erheblich reduzierten Betriebskosten, Komplexität und schnelleren Entwicklungszyklen (Time-to-Market) profitieren. Dies wird allerdings durch eine erhöhte Abhängigkeit von Cloud-Providern und noch vergleichsweise unreifen unterstützenden BaaS-Diensten erkauft.

Serverless- und Microservice-Architekturen widersprechen sich nicht. Sie können gemeinsam auftreten und sind kompatibel, da sie viele Gemeinsamkeiten teilen (etwa die Präferenz für Zustandslosigkeit aufgrund des Fokus auf horizontaler Skalierbarkeit). Microservices eignen sich für lang laufende, komplexere Dienste mit hohem Ressourcen- und Managementbedarf. Demgegenüber führen Serverless-Architekturen Funktionen nur bei Bedarf aus und eignen sich damit insbesondere für die ereignisgesteuerten Abläufe. Serverless konzipierte Services können in Microservice-Architekturen als eigenständige Services auftreten und andersherum.

■ 12.7 Zusammenfassung

Irgendwie hat es (vermutlich) die Video-Streaming-Plattform Netflix geschafft, Microservices für die Fachwelt zu „romantisieren“. Angeblich sollen über 700 Microservices für die Plattform ausgeführt werden. Aufgrund des Erfolgs von Netflix waren viele Unternehmen bereit, diesen Architekturansatz zu adaptieren. Was häufig übersehen wurde und wird, ist allerdings die Tatsache, dass Microservice-Erfolgsgeschichten hauptsächlich von produktbasierten Unternehmen stammen, die aufgrund dieses Fokus sehr viele Iterationen und Modelle durchlaufen haben, bevor ein erfolgreiches und marktreifes Produkt entstanden ist.

Microservices beruhen auf dem Prinzip der unabhängigen Austauschbarkeit und losen Kopp lung von Einzelkomponenten, die in autonomen Teams im Sinne des DevOps entwickelt und betrieben werden. Damit hat dieser Architekturansatz (soll er funktionieren) nicht nur eine technische, sondern eben auch immer eine arbeitsorganisationale sowie arbeitskulturelle Implikation, die nicht außer Acht gelassen werden sollte.

- Microservices sind Lösungen für komplexe Probleme. Ohne komplexes Problem gibt es keinen wirklichen Grund für Microservices.
- Auch wenn es keine nennenswerten Skalierungserfordernisse gibt, bringen Microservices häufig wenig Mehrwert.
- Ferner ist die Teamgröße für erfolgreiche Microservice-Ansätze entscheidend. Wenn nicht genügend (große) Teams existieren, sollte man Microservices besser vermeiden. Auch wenn man gewohnt ist, in Projekt- und nicht in Produktteams zu denken, läuft man schnell Gefahr, Microservices nicht im Sinne der Sache zu verwenden.

Unabhängig von diesen Worten der Warnung, sind Microservice-Architekturen durchaus erstrebenswert, denn man kann sehr viel beim Entwurf robuster, reaktiver und verteilter Systeme lernen. Dem Leser sei daher bei Interesse die folgende vertiefende Literatur von in diesem Kapitel behandelten Themen empfohlen.

Zum Thema Microservices findet sich eine Vielzahl an Literatur. Eine aus Sicht des Autors wirklich zu empfehlende Quelle ist aber vermutlich (Newman 2015). Newman bereitet hier viel praktische Erfahrung aus seiner Funktion als IT-Architekt im Kontext von Microservices auf überschaubar vielen Seiten auf. Wer sich in diesem Zusammenhang auch für die Implikationen der Aufbau- und Ablauforganisation von Unternehmen auf die Architekturen von IT-Systemen interessiert, sei hier begleitend der zwar betagte, aber dennoch noch gültige Artikel „How Committees Invent“ von (Conway 1968) ergänzend empfohlen. Es ist gegebenenfalls auch ein Abgleich mit (Kim u. a. 2017) empfohlen (obwohl bereits in Kapitel 3 erwähnt).

REST ist zwar ein sehr populärer und grundsätzlich einfacher Ansatz zur Entwicklung von APIs im Microservice Umfeld. Dennoch folgt nicht jede als REST-konform postulierte API auch immer (allen) REST-Prinzipien. Hier sei daher sowohl die Originalquelle von (Fielding 2000) als auch das REST Maturity Model von (Richardson, Amundsen, und Ruby 2013) zur Lektüre empfohlen. Hinsichtlich der Versionierung von APIs sei ferner der Verweis auf die Originalquelle des Semantic Versioning (Preston-Werner 2013) erlaubt. Auch sollten hier die im Kontext der ereignisbasierten Integration zunehmend populärer werdenden reaktiven Ansätze (Kuhn, Hanafee, und Allen 2017) nicht vergessen werden. Wer hier eine gute Einführung in Python (der Programmiersprache, die zumeist in den begleitenden Labs genutzt wird) sucht, sei auf (Picard 2018) verwiesen.

Die Skalierung von Stateful-Services und verteilten Datenbanksystemen ist ein Thema für sich und mit einer ganz eigenen innewohnenden Komplexität. In Microservice-Architekturen versucht man diese Komplexität daher gerne in entsprechend für diese Komplexität gebauten Systemen zu isolieren. Meist sind dies NoSQL-Datenbanken. Daher sei hier auf die grundsätzliche CAP-Problematik und damit natürlich auf (Fox und Brewer 1999) sowie (Gilbert und Lynch 2002) verwiesen. Wer einen eher anwendungsbezogenen Einstieg in diese Thematik sucht, wird aber vermutlich eher mit Werken wie (Redmond und Wilson 2012) glücklich werden.

Serverless-Architekturen binden konsequenter als Microservice-Architekturen BaaS-Dienste (Backend as a Service) von Drittanbietern ein. Benutzerdefinierter Code wird auf ein Minimum reduziert, meist auf den Logic-Tier beschränkt und auf FaaS-Plattformen (Functions as a Service) ausgeführt. Dadurch entfällt ein Großteil der Notwendigkeit für Always-on-Komponenten, und Serverless-Architekturen können so oft von erheblich reduzierten Betriebskosten profitieren. Dies wird allerdings durch eine erhöhte Abhängigkeit von Cloud-Providern und noch vergleichsweise unreifen BaaS-Diensten erkauft. Für einen kritischen und reflektiven Blick auf den Serverless-Architekturansatz wird auf die Studien von (Baldini u. a. 2017), (Kratzke 2018), (Jonas u. a. 2019) und (Yussupov u. a. 2019) verwiesen. Eher anwendungsbezogene Einstiege in diese Thematik geben etwa (Zambrano 2018) oder (Katzer 2020). Doch auch hier wird der Leser nach kurzer Recherche viele geeignete Quellen finden.

Ergänzende Materialien

Auf der Website zum Buch finden sich zu diesem Kapitel ferner folgende Ergänzungsmaterialien:

- Praktische Übungen (Labs) zu Serviceintegrationen in Microservice-Architekturen (u. a. gRPC, REST, Messaging) und dem FaaS-Programmiermodell
- Übersichten inklusive Links zu gRPC-/REST-Bibliotheken für Python
- Übersichten inklusive Links zu Open-Source-Messaging-Systemen
- Übersichten inklusive Links zum ReactiveX-Programmiermodell (Liste unterstützender Programmiersprachen)
- Übersichten inklusive Links zu Open-Source-API-Gateways
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer (Themen: FaaS, Microservices, Serverless)



<https://bit.ly/2XYyCtM>

13

Beobachtbare Architekturen

„Vertraue, aber prüfe nach.“
(Russisch: „Dowerjai, no prowerjai“)

Lenin, Kommunist und Gründer der Sowjetunion

Es wurde bereits im Kapitel 3 erwähnt, dass komplexe Systeme nicht mehr vollständig von einer einzelnen Person überschau- und verstehbar sind. Für komplexe und durch umfangreiche Service-Interaktionen geprägte Microservice-Architekturen gilt dies im Besonderen. Daher sollten Annahmen, die beim Design getroffen wurden, kontinuierlich überprüft werden. Hierzu sind Feedback-Schleifen im Produktivsystem erforderlich, mittels derer man kontinuierlich und automatisiert Einblick in das dynamische Systemverhalten erhält. Die Beobachtbarkeit von Softwaresystemen realisiert man dabei üblicherweise mittels Telemetriedaten, die häufig in drei Aspekte unterteilt werden:

- Mittels Protokollierung (**Logging**) lässt sich Einblick in anwendungsspezifische Nachrichten, die von Prozessen ausgegeben werden, gewinnen.
- Metriken im Rahmen eines **Monitorings** liefern quantitative Informationen zu Prozessen, die im System ausgeführt werden.
- Eine verteilte Ablaufverfolgung (**Distributed Tracing**) ermöglicht hingegen Einblick in das dynamische Verhalten und den Lebenszyklus von Requests entlang von Systemkomponenten, sodass Fehler und Latenzen bestimmbar sind.

Diese Aspekte sind eng miteinander verbunden. Metriken können verwendet werden, um beispielsweise eine Teilmenge von Traces mit schlechter Performance zu lokalisieren. Mit diesen Traces verknüpfte Logs können dazu beitragen, die Hauptursache für dieses Verhalten zu ermitteln. Anschließend können basierend auf dieser Analyse neue Metriken konfiguriert werden, um dieses Problem auch in anderen Systemteilen zu erkennen und zu beheben.

■ 13.1 Konsolidierung von Telemetriedaten

Um einen solchen kontinuierlichen und systematischen Einblick in das dynamische Systemverhalten zu erhalten, ist jedoch eine Infrastruktur zur Erfassung von Telemetriedaten erforderlich, die die oben genannten Logging-, Metrik- und Tracing-Daten in einem zentralen Service zum Zwecke der Beobachtbarkeit und Analysierbarkeit des Gesamtsystems konsolidiert. Hierzu müssen

- Services systematisch geloggt,
- Monitoringdaten erzeugt
- und Requests entlang von Systemkomponenten getracet werden

können. Dies kann durch manuelle Instrumentierung (siehe Abschnitt 13.2) oder automatisierte Instrumentierung (siehe Abschnitt 13.3) erfolgen. Diese durch Instrumentierungen erfassten Daten sollten jedoch im Sinne einer Telemetrikconsolidierung einfach analysiert werden können. DevOps-Teams erhalten daher zumeist einen (gegebenenfalls eingeschränkten) Self-Service-Zugriff auf derart konsolidierte Telemetriedaten. Mittels dieser konsolidierten Telemetrie-Infrastruktur können so DevOps-Teams auch anlassbezogene und unvorhergesehene Probleme analysieren, isolieren, beheben und idealerweise zukünftig vermeiden. Hierzu müssen allerdings auch Auswertungen und Visualisierungen eventuell anlassbezogen erstellt und analysiert werden können, um auch neue und bislang unbekannte Probleme datenbasiert analysieren zu können.

Das Prinzip funktioniert dabei immer wie in Bild 13.1 gezeigt. Services (aber auch Jobs) werden zur Überwachung mittels einer **Instrumenting Library** instrumentiert. Abhängig von der Art der Telemetriedaten (Logging, Monitoring, Tracing) werden diese an eine **Zeitreihen-Datenbank** entweder aktiv übertragen (*push*) oder von dieser periodisch abgefragt (*pull*). Im Falle von kontinuierlich laufenden Services können zu überwachende Services abhängig vom verwendeten Monitoring-Stack mittels einer *Service Discovery* automatisiert über die Operating-Plattform (z. B. Kubernetes, siehe Kapitel 9) ermittelt werden.

Im Falle von Jobs (oder auch Functions, siehe Kapitel 10) müssen Telemetriedaten hingegen an einen kontinuierlich laufenden Forwarder (manchmal auch Push-Gateway genannt) aktiv übertragen werden, der dann die Telemetriedaten an die *Zeitreihen-Datenbank* weiterleitet. Welcher Weg gegangen wird, ist immer etwas vom Monitoring-Stack und von der Art der Telemetriedaten abhängig. Tabelle 13.1 gibt hier einen entsprechenden Überblick über gebräuchliche Varianten. Auf der Website zum Buch findet sich ein Überblick weiterer Produkte.

Die *Zeitreihen-Datenbank* kann üblicherweise mittels eines *User Interfaces* interaktiv abgefragt werden, um anlassbezogen Probleme zu identifizieren bzw. einen stark verdichteten und visualisierten Systemüberblick zu erhalten. Dabei können für unterschiedliche Zielgruppen und Fragestellungen unterschiedliche Visualisierungen definiert werden, die durchaus geeignet sind, komplexe Zusammenhänge anschaulich zu visualisieren (siehe Bild 13.2). Ergänzend kann ein *Alert-Manager* für den Systemzustand als kritisch identifizierte Zeitreihen kontinuierlich überwachen und automatisiert Warnungen erzeugen, wenn Zeitreihen beispielsweise aus definierten Schwellwertbereichen ausbrechen und damit ungewöhnliches Verhalten zeigen. Dies kann auf Probleme hinweisen, die über das User Interface gegebenenfalls anlassbezogen genauer analysiert werden müssen.

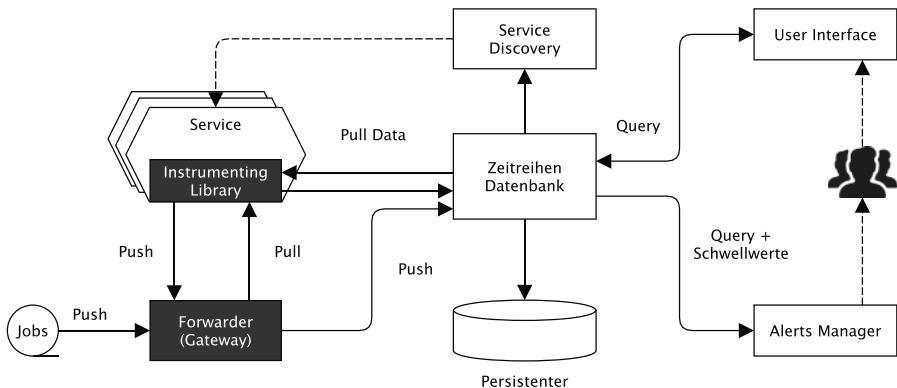


Bild 13.1 Konsolidierung von Telemetriedaten

Tabelle 13.1 Konsolidierungsansätze von Telemetriedaten

| Workload | Daten | Richtung | Forwarder | Anmerkungen und typische Produkte |
|----------|---------|----------|----------------|--|
| Service | Logs | Push | Ja | z. B. Fluentd + Elasticsearch + Kibana |
| Service | Metrics | Pull | Nein | z. B. Prometheus + Grafana |
| Service | Traces | Push | Tracing Server | z. B. Jaeger oder APM |
| Job | Logs | Push | Ja | z. B. Fluentd + Elasticsearch + Kibana |
| Job | Metrics | Push | Ja | z. B. Prometheus + Grafana |
| Job | Traces | | | ehler unüblich |



Bild 13.2 Visualisierung von Telemetriedaten am Beispiel von Kibana (Quelle: elastic.co)

Wie Services für einen solchen Stack von Observability-Produkten (Bibliotheken zur Instrumentierung, Zeitreihen-Datenbank, Benutzeroberfläche) zur Instrumentierung genutzt werden können, werden wir im Folgenden betrachten. Diese Instrumentierung kann mittels einer manuellen Instrumentierung im Sinne einer Whitebox-Überwachung (siehe Abschnitt 13.2) oder auch automatisiert im Sinne einer Blackbox-Überwachung (siehe Abschnitt 13.3) erfolgen. Beide Varianten haben ihre Vor- und Nachteile.

■ 13.2 Instrumentierung von Systemen

Wir werden uns in diesem Abschnitt mit der manuellen Instrumentierung von Software zur Erfassung von Ereignissen im Verlaufe der Prozessausführung (Logging, siehe Abschnitt 13.2.1), dem Erfassen von quantitativen Metriken im zeitlichen Verlauf (Monitoring, siehe Abschnitt 13.2.2) sowie der Verfolgung von Requests entlang von Services (Tracing, siehe Abschnitt 13.2.3) befassen. Wir beginnen dabei mit der geläufigsten Instrumentierung, dem sogenannten Logging, das viele Entwickler kaum noch bewusst als Instrumentierung wahrnehmen, da es so allgegenwärtig ist.

13.2.1 Logging

Mittels Logging (Protokollierung) lässt sich Einblick in anwendungsspezifische Nachrichten, die von Prozessen ausgegeben werden, gewinnen. Das Problem in Cloud-nativen Systemen ist, dass sich Logs über zahlreiche Services verteilen und Service für Service analysiert werden müssen. Bequemer wäre es, wenn diese Logs zentral in einer durchsuchbaren Datenbank für (Fehler-)Analysen hinterlegt wären. Daher haben sich Tools zur Log-Aggregation und zentralisierten Log-Analyse etabliert.

Insbesondere das Logging mittels `print()`-Statements auf der Standardausgabe (`stdout`) ist vermutlich so alt wie die Programmierung selbst und daher für viele Entwickler so allgegenwärtig in der Softwareentwicklung, dass es oft nicht mal mehr bewusst als eine Whitebox-Instrumentierung zur Beobachtbarkeit von Systemen empfunden wird. Logs machen dennoch das Verhalten einer laufenden App sichtbar. Oft werden Logs in Dateien auf Platte geschrieben (Logdatei). Logs in ihrer rohen Form sind üblicherweise ein Textformat mit einem Ereignis pro Zeile.

In Cloud-nativen Systemen werden Logs von Services aber auch häufig als Stream von Ereignissen ungepuffert auf `stdout` geschrieben (siehe auch Abschnitt 8.5.6). Log-Router (wie Logstash, Fluentd, usw.) fassen die `stdout`-Streams aller Prozesse eines Hosts zusammen und leiten diese an Zeitreihen-Datenbanken zur Archivierung weiter. Diese Archivierungsziele sind für die App weder sichtbar noch konfigurierbar – sie werden vollständig von der Laufzeitumgebung aus verwaltet. Mittels dieser Zeitreihen-Datenbanken (z. B. Elasticsearch) kann das Verhalten einer App flexibel und automatisiert beobachtet werden. Logs werden dadurch zu einem gut analysierbaren Stream von aggregierten, nach Zeit sortierten Ereig-

nissen, die aus den Output Streams aller laufenden Prozesse und Services zusammengefasst werden. Dies schließt ein:

- Bestimmte Ereignisse in der Vergangenheit zu finden
- Umfangreiche grafische Darstellungen (wie Requests pro Minute)
- Aktives Alarmieren aufgrund benutzerdefinierter Heuristiken (wie ein Alarm, falls die Anzahl von Fehlern pro Minute eine gewisse Grenze überschreitet)

Da das Loggen aller Ereignisse die für das Logging verfügbaren Ressourcen innerhalb kurzer Zeit aufbrauchen kann und die Auffindbarkeit bestimmter Ereignisse erschweren würde, werden meist die in Tabelle 13.2 genannten Dringlichkeitsstufen (Log Level) genutzt, mittels derer das Logging von Events z. B. in Production-, Staging-, Test-Environments ein- oder ausgeschaltet werden kann.

Tabelle 13.2 Gebräuchliche Log-Level

| Log Level | Beschreibung |
|-----------|---|
| Fatal | Fehler, der zur Terminierung einer Anwendung führt. |
| Error | Laufzeitfehler, welcher die Funktion der Anwendung behindert, oder unerwarteter Programmfehler. |
| Warning | Aufruf einer veralteten Schnittstelle, fehlerhafter Aufruf einer Schnittstelle, Benutzerfehler oder ungünstiger Programmzustand. |
| Info | Laufzeitinformationen wie der Start und Stopp der Anwendung, Benutzeranmeldungen und -abmeldungen sowie durchgeführte Geschäftstransaktionen. |
| Debug | Informationen zum Programmablauf. Wird im Normalfall nur in der Entwicklung oder zur Nachvollziehung eines Fehlers verwendet. |
| Trace | Detaillierte Verfolgung des Programmablaufs, insbesondere zur Nachvollziehung eines Programmierfehlers. |

Üblicherweise werden nur Ereignisse bis zu einem definierten Log-Level protokolliert. Die Log-Ebene Info bedeutet also, dass Ereignisse der Kategorien Fatal, Error, Warning und Info protokolliert würden, aber nicht die Ebenen Debug und Trace.

Zum Logging stehen unterschiedliche Programmbibliotheken für verschiedene Softwareumgebungen zur Verfügung. Die Standardbibliothek von Python enthält beispielsweise bereits ein eingebautes und flexibles Logging-Modul, mit dem verschiedene Logging-Konfigurationen für unterschiedliche Logging-Erfordernisse erstellt werden können. Diese ist für Unified Logging absolut ausreichend. Viele andere Sprachen bieten ähnliche Logging-Bibliotheken.

Diese Logging-Bibliotheken bieten Funktionen, die es Entwicklern ermöglichen, Logs für verschiedene Zielorte (Stdout, Log-Dateien, Remote-Server etc.) anzulegen. Hierzu kann man zumeist verschiedene Handler anlegen. Protokoll-Events werden an die entsprechenden Handler weitergeleitet und entsprechend verarbeitet. Da im Cloud-native Umfeld zumeist direkt auf stdout geloggt werden soll, ist Logging aus Entwicklersicht sogar besonders einfach, da beispielsweise das Dateihandling komplett wegfällt. Folgende Logging-Best-Practices werden oft empfohlen und lassen sich etwa in Python mit dem gezeigten Listing 13.1 sehr einfach umsetzen:

1. Logge auf stdout (das Unified Logging-System und die Plattform machen den Rest, siehe auch Abschnitt 8.5.6)
2. Definiere den Log-Level über eine Umgebungsvariable
3. Logge immer auch den Service-Namen, in dem Ereignis ausgelöst wurde
4. Logge den Zeitpunkt, den Level und das Event
5. Logge Zeitpunkte im ISO 8601-Format inklusive Zeitzone

Listing 13.1 Logging-Instrumentierung am Beispiel von Python

```
import os, logging

logging.basicConfig(
    level = os.getenv('LOGLEVEL', logging.WARNING),           #1
    format = '%(asctime)s | %(name)s | %(levelname)s | %(message)s', #4
    datefmt = '%Y-%m-%dT%H:%M:%S%z'                           #5
)

log = logging.getLogger('service-name')                         #3

def process_request():
    log.debug('This debug message will not be logged (default setting)')
    log.info('This info message will not be logged (default setting)')
    log.warning('This warning message will be logged (default setting)')
    log.error('This error message will be logged (default setting)')
    log.critical('This critical message will be logged (default setting)')
```

Auf der Website zum Buch finden sich Labs, die das Thema Logging und Log-Konsolidierung praktisch vertiefen.

13.2.2 Monitoring

Unter Monitoring versteht man die Überwachung von Vorgängen und die quantitative Erfassung von Kennzahlen. Es ist ein Überbegriff für alle Arten von systematischen Erfassungen, Messungen oder Beobachtungen eines Vorgangs oder Prozesses mittels technischer Hilfsmittel und Beobachtungssysteme. Eine Funktion des Monitorings besteht insbesondere darin, bei einem beobachteten Ablauf oder Prozess festzustellen, ob dieser den gewünschten Verlauf nimmt und bestimmte Schwellwerte eingehalten werden, um bei Abweichungen steuernd eingreifen zu können. Im Cloud-native Umfeld versteht man unter Monitoring insbesondere die Erfassung von Metriken in Form von Zeitreihen, um vor allem quantitative Informationen zu Prozessen und Vorgängen eines Cloud-nativen Systems erheben und analysieren zu können. Metrikbasierte Monitoring-Lösungen zeichnen Echtzeitmetriken üblicherweise mittels Zeitreihen-Datenbanken auf, die von Anwendungen und Controllern abgefragt werden und u. a. Echtzeit-Warnmeldungen ermöglichen. Monitoring-Lösungen bestehen meist aus mehreren Tools (siehe auch Bild 13.1):

- Exporter, die normalerweise auf überwachten Hosts ausgeführt werden, um lokale Host- und Service-Metriken an einen Zentralspeicher zu exportieren.

- Zeitreihen-Datenbank zur zentralisierten Speicherung von Metriken.
- Alert-Manager, der bei einer Schwellwertüberschreitung Benachrichtigungen verschicken kann.
- Nutzeroberfläche (UI) zum Darstellen von Dashboards.
- Querying-System, das zum Erstellen von Dashboards und Warnungen verwendet wird.

Exporter werden regelmäßig von Monitoring-Lösungen wie beispielsweise Prometheus abgefragt. Hierbei wird mittels HTTP meist der Standard-URL-Pfad `/metrics` abgefragt. Es werden jedoch auch etablierte Überwachungs- und Verwaltungsprotokolle wie SNMP, JMX oder CollectD unterstützt. Jede der Datenquellen liefert die aktuellen Werte der Metriken. Eine zentrale Monitoring-Lösung aggregiert dann Daten über die Datenquellen hinweg. Einige Lösungen (wie etwa Prometheus) verfügen hierzu über eine automatische Service-Discovery, um Ressourcen, die als Datenquellen verwendet werden sollen, automatisch ermitteln zu können. Diese gesammelten Daten werden in einer Datenbank zur Zeitreihenanalyse gespeichert. Dabei kann man zwei Arten des Monitorings unterscheiden.

- **Blackbox-Monitoring:** Standard-Systemmetriken sind hingegen ohne Instrumentierung erfassbar. Dies umfasst insbesondere die Überwachung von physischen oder virtuellen Maschinen oder Containern hinsichtlich allgemeiner Systemmetriken wie Speicherplatz, CPU-Auslastung, Speicherauslastung, Load Averages usw.. Diese Form des Monitorings ist im Allgemeinen durch Betriebssysteme gut unterstützt.
- **Whitebox-Monitoring:** Instrumentierung, um service-spezifische Metriken erfassen zu können.

Im Cloud-nativen Umfeld werden Services allerdings nicht nur mittels Blackbox-Monitoring, sondern ergänzend auch mittels Whitebox-Monitoring hinsichtlich service-spezifischer Metriken überwacht.

Hierzu müssen die überwachten Anwendungen, wie exemplarisch in Listing 13.2 gezeigt, instrumentiert werden, um deren interne Metriken als Exporter selbst bereitzustellen. Hierzu stehen unterschiedliche Metrik-Exporter-Programmbibliotheken für verschiedene Softwareumgebungen zur Verfügung. Es lassen sich somit sowohl service-spezifische Metriken als auch Blackbox-Standardmetriken exportieren, wie die Auslastung des Arbeitsspeichers, CPU, Festplatte oder Netzwerk.

Listing 13.2 Metrikinstrumentierung am Beispiel von Prometheus

```
from prometheus_client import start_http_server, Summary
import random, time

# Metrik: Bearbeitungsdauer von Requests zu verfolgen
REQUEST_TIME = Summary('req_processing_seconds', 'Time spent processing request')

@REQUEST_TIME.time()          #Funktion zur Metrikerhebung annotieren
def process_request(t):
    time.sleep(t)

if __name__ == '__main__':
    start_http_server(8000)  #HTTP-Server zur Metrikbereitstellung
    while True:
        process_request(random.random())
```

Da Batch-Jobs keinen Always-on-Charakter haben, können diese – anders als kontinuierlich verfügbare Services – nicht periodisch von Monitoring-Systemen abgefragt werden. Über sogenannte Push-Gateways (siehe Bild 13.1) können jedoch auch kurzlebige Prozesse und Batch-Jobs Metriken einem Monitoring-System zugänglich machen.

Die Instrumentierung ändert sich dann etwas, und der zu instrumentierende Job muss die Adresse des für ihn verantwortlichen Push-Gateways kennen (siehe Listing 13.3).

Listing 13.3 Metrikinstrumentierung am Beispiel von Prometheus

```
from prometheus_client import Gauge, push_to_gateway
g = Gauge('job_last_success_unixtime', 'Last time a batch job successfully finished')
g.set_to_current_time()
push_to_gateway('localhost:9091', job='batchA') #Push-Gateway
```

13.2.2.1 Metrikarten

Unterschiedliche Metrik-Exporter-Programmbibliotheken stellen zumeist die folgenden Arten von Metriken zur Verfügung.

- **Zähler (Counter):** Ein Zähler ist eine kumulative Metrik, die einen einzelnen monoton ansteigenden Zähler darstellt, dessen Wert nur erhöht oder beim Neustart auf null zurückgesetzt werden kann. Zähler sind geeignet, um die Anzahl eingegangener Requests, erledigter Aufgaben oder aufgetretener Fehler zu überwachen. Die Zähler-Instrumentierung erfolgt wie in Listing 13.4 gezeigt.
- **Messung (Gauge):** Eine Messung ist eine Metrik, die einen einzelnen numerischen Wert darstellt, der willkürlich hinauf- und hinuntergehen kann. Messungen werden normalerweise für Messwerte wie Temperaturen oder die aktuelle Speichernutzung verwendet, aber auch für Zählungen, die steigen und fallen können wie die Anzahl gleichzeitiger Requests. Listing 13.4 zeigt exemplarisch, wie solche Messinstrumentierungen erfolgen können.
- **Verteilungen (Histogramme):** Ein Histogramm erfasst Beobachtungen (oft Messgrößen wie Zeitdauern oder Größen) und zählt sie in konfigurierbaren Bereichen. Histogramme stellen die Verteilung von Beobachtungen ausführlicher dar. Häufig reichen auch einfache Quantile, um Verteilungen einschätzen zu können. Listing 13.4 zeigt, wie solche Instrumentierungen zur Erfassung von Messdaten-Verteilungen erfolgen können.

Listing 13.4 Instrumentierung mittels Counter, Gauge und Histogramm

```
from prometheus_client import Counter, Gauge, Histogram
c = Counter('request_execptions', 'Occured exceptions')      # Zähler-Metrik
g = Gauge('inprogress_requests', 'Current requests')        # Messung
h = Histogram('request_latency_second', 'Request Latency')  # Histogramm
@c.count_exceptions()                                         # Annotation einer
@g.track_in_progress()                                         # Funktion mit
@h.time()                                                       # definierten Metriken
def process_request():
    run_complex_process()
```

```
# Messungen können auch mittels Callbacks bestimmt werden
queue = []
g.set_function(lambda: len(queue)) # Messung mittels Callback
```

13.2.2.2 Empfehlungen für die Metrikinstrumentierung

In Cloud-nativen Systemen unterscheidet man dabei üblicherweise kontinuierlich verfügbare Online-Systeme und Batch-Systeme, die leicht unterschiedliche Erfordernisse bei der Metrikerhebung haben. In beiden Fällen sollte die Metrikinstrumentierung, ähnlich wie das verbreitetere Logging, bei Komponenten Cloud-nativer Systeme ein integraler Bestandteil des Codes sein. Grundsätzlich sollte die Instrumentierung von Requests dabei einheitlich erfolgen. So ist es beispielsweise Empfehlenswert, Requests am Ende ihrer Bearbeitung zu zählen, da dies dann mit den Fehler- und Latenzstatistiken übereinstimmt und in der Regel einfacher zu codieren ist.

Ein **Online-System** ist ein System, bei dem ein Nutzer oder ein anderes System eine sofortige Reaktion erwartet. Die meisten Datenbank- und Webservices fallen in diese Kategorie. Typische Schlüsselmetriken solcher Komponenten sind oft:

- Anzahl der beantworteten Requests
- Anzahl aktuell laufender Requests
- Anzahl aufgetretener Fehler
- Latenz

Batch-Systeme zeichnen sich hingegen dadurch aus, dass sie nicht kontinuierlich ausgeführt werden, was für die Metrikerhebung meist ein Push-Gateway (siehe Bild 13.1) erforderlich macht. Typische Schlüsselmetriken von Batch-Jobs sind:

- der Zeitpunkt der letzten erfolgreichen Ausführung
- Dauer der Teilschritte eines Jobs
- Gesamtausführungszeit eines Jobs
- Zeitpunkt des letzten Abschlusses eines Jobs (erfolgreich oder fehlgeschlagen)
- Gesamtzahl verarbeiteter Datensätze

13.2.3 Tracing

Tracing-Systeme ermöglichen die Rückverfolgung von Service-Interaktionen im laufenden Betrieb und helfen beim Erfassen von Zeitreichendaten (Traces), die u. a. zur Behebung von Latenzproblemen in Servicearchitekturen hilfreich sein können. Oberflächen von Tracing-Systemen zeigen meist Abhängigkeitsdiagramme an, um das Gesamtverhalten einschließlich Fehlerpfaden oder Aufrufabfolgen von Services einfacher in Analysen erfassen zu können.

Jedes Tracing-System benötigt eine Möglichkeit, den Kausalzusammenhang zwischen Aktivitäten in vielen unterschiedlichen Services (Prozessen) zu ermitteln. Das Problem hierbei ist, dass diese Interaktionen extrem heterogen sind. Services können über RPC-Frameworks, Messaging-Systeme, direkte HTTP-Aufrufe, UDP-Pakete oder gänzlich anders miteinander interagieren. Tracing-Systeme können daher grundsätzlich Kausalzusammenhänge anhand zweier Strategien bestimmen:

- **Blackbox-Tracing**-Systeme stellen Zusammenhänge zwischen Services her, indem deren Netzwerkverkehr beobachtet und daraus Kausalzusammenhänge ermittelt werden (Chow u. a. 2014). Der Vorteil ist, dass Services nicht instrumentiert werden müssen. Allerdings muss man einen Kompromiss zwischen der Beobachtungsdauer und der Qualität abgeleiteter Traces eingehen. Resultierende Traces sind ferner mit Unsicherheit behaftet.
- **Annotationsbasierte** (Whitebox)-Tracing Systeme verfolgen hingegen einen Whitebox-Ansatz und sind daher aufwendiger im Betrieb, da zu tracende Services im Code instrumentiert werden müssen. Hierzu muss die Codierung der Services „angefasst“ werden, um Aufrufe zwischen Systemen für das Tracing-System zu kennzeichnen. Allerdings lassen sich so Kausalzusammenhänge zweifelsfrei und unmittelbar ermitteln. Im Cloud-native Umfeld hat sich daher dieser Ansatz durchgesetzt, der insbesondere durch die X-Trace- (Fonseca u. a. 2007) und Dapper-Paper (Sigelman u. a. 2010) terminologisch geprägt wurde.

Wir werden uns daher primär mit dem Whitebox-Ansatz befassen, da sich bei Cloud-native Systemen entsprechende Tracing-Informationen mittels HTTP-Headern entlang von Service-Aufrufen weiterreichen lassen. Diese Art der Interaktion ist in Cloud-native Systemen einer der häufigsten. Unter den Begriffen Trace, Span und Span-Kontext versteht man dabei die in Bild 13.3 gezeigten Zusammenhänge:

- **Trace:** Ein Trace beschreibt eine Transaktion, die sich durch ein verteiltes System bewegt.
- **Span:** Eine benannte, zeitgesteuerte Operation, die einen Teil des Workflows der verteilten Transaktionsverarbeitung darstellt. Spans können mit Daten annotiert werden (Log), die zur Verfolgung und Auswertung von Transaktionen erforderlich oder hilfreich sein können.
- **Span-Kontext:** Ein Space-Kontext sind Trace-Informationen, die die verteilte Transaktion zwischen Services begleiten. Der Span-Kontext enthält die Trace-ID, die Span-ID und alle anderen Daten, die das Tracing-System zur Verfolgung von Transaktionen in nachgeschalteten Services benötigt.

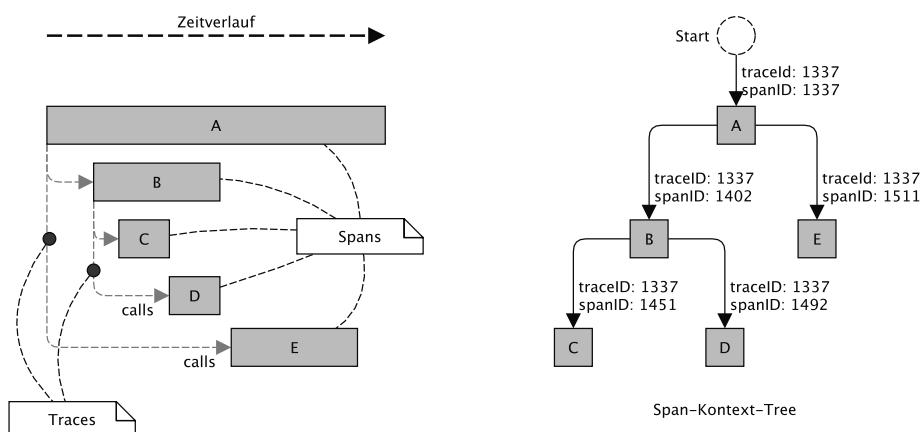


Bild 13.3 Tracing: Trace, Span und Kontext

Code muss beim Whitebox Tracing analog zum Logging und Monitoring „instrumentiert“ werden, um Trace-Daten an Tracing-Systeme zu melden. Diese Systeme bieten zumeist auch eine grafische Oberfläche, die es einem ermöglicht, die Vielzahl an Traces mittels Querys zu filtern und sich grafisch für Analysen aufzubereiten zu lassen (siehe Bild 13.4). Hier gibt es sowohl kommerzielle managed Services oder self-hosted Lösungen. Die Website zum Buch gibt einen Überblick über entsprechende Systeme.

Damit dies aber funktioniert, muss normalerweise die Konfiguration und Einbettung von Tracing-Code unter Nutzung von Instrumentierungsbibliotheken für spezifische Tracing-Systeme erfolgen. Da die Instrumentierung dann leicht tracing-system-spezifisch werden kann, haben sich aber auch **Tracing-Standards** wie beispielsweise die *OpenTracing-API* entwickelt, um zu vermeiden, dass eine Instrumentierung auch über Servicegrenzen hinweg reibungslos funktioniert. Die Instrumentierung werden wir daher am Beispiel dieses Standards veranschaulichen.

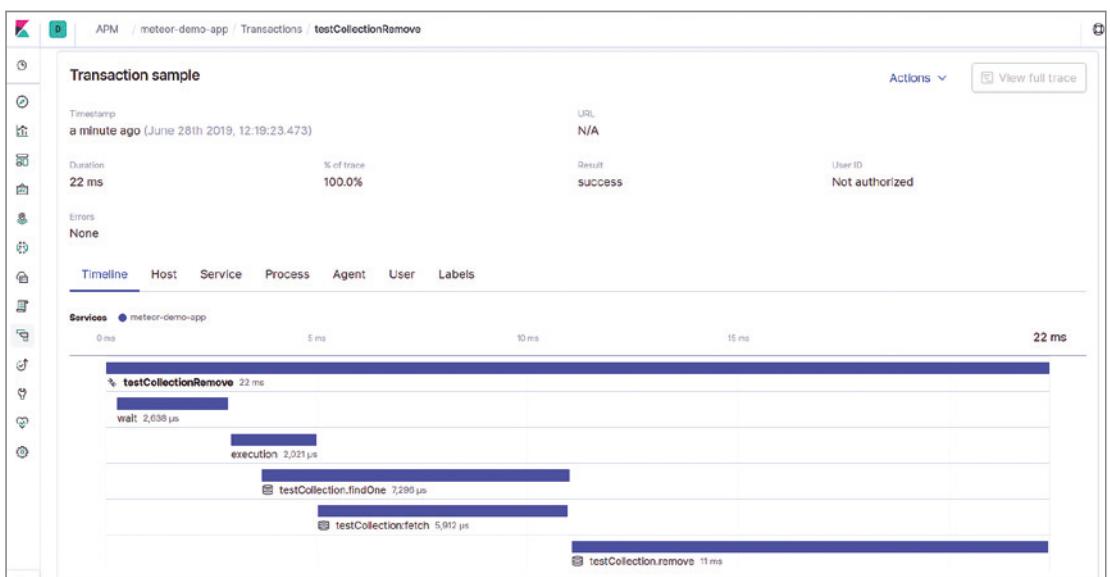


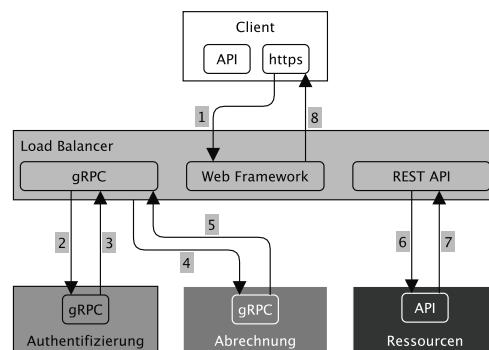
Bild 13.4 Tracing am Beispiel Elastic APM (Quelle: elastic.co)

13.2.3.1 Empfehlungen für die Instrumentierung

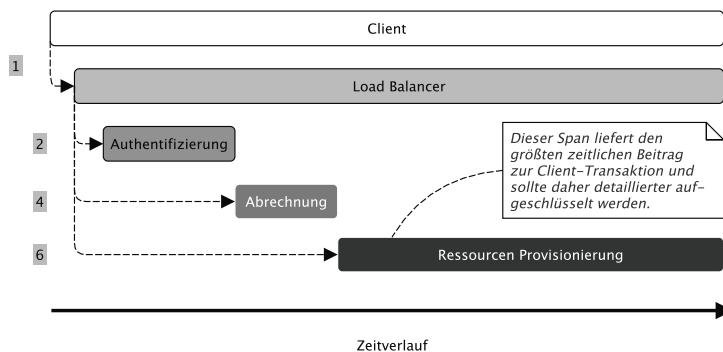
Die beiden grundlegenden Aspekte bei der Tracing-Instrumentierung sind Spans und die Beziehungen zwischen diesen Spans. Spans (siehe Bild 13.3) sind dabei logische Arbeitseinheiten in einem verteilten System, die einen Namen, eine Startzeit und eine Dauer haben. In einer Ablaufverfolgung sind Spans mit der verteilten Systemkomponente verbunden, die sie erzeugt hat. Ein Span kann auf weitere Spans verweisen, die in einem kausalen Transaktionszusammenhang (Aufrufzusammenhang) stehen. Diese Verbindungen zwischen Spans helfen, die Semantik des laufenden Systems sowie den kritischen Pfad für latenzempfindliche (verteilt) Transaktionen zu bestimmen.

Grundsätzlich ist es wünschenswert, Spans und deren Relationen zueinander für alle Komponenten eines Systems erfassen zu können. Dies kann jedoch aufwendig sein, insbesondere dann, wenn Komponenten in einem System vorhanden sind, die überhaupt nicht für Tracing vorbereitet sind. Daher ist es oft ratsam, mit dem Tracing bei Komponenten zu beginnen, die unter eigener Entwicklung stehen (wobei man also Zugriff auf den Quellcode hat) und von denen bekannt ist, dass sie eine vielfältige Interaktion mit anderen Komponenten haben. Dies sind in Cloud-nativen Systemen oft gRPC-Schichten oder Web-Frameworks für REST-basierte APIs. Solche Komponenten bilden oft eine gute Grundlage für die Instrumentierung und decken erfahrungsgemäß eine beträchtliche Anzahl relevanter Transaktionspfade ab. Dabei sollten die Code-Komponenten priorisiert instrumentiert werden, die entlang eines kritischen Pfads einer hochwertigen Transaktion liegen. Durch die Priorisierung anhand der Spans in einer Transaktion, die auf dem kritischen Pfad liegen und die meiste Zeit verbrauchen, besteht die größte Chance für eine messbare Optimierung. Das Hinzufügen einer detaillierten Instrumentierung zu einem Span, der nur einen vernachlässigbaren Teil der gesamten Transaktionszeit ausmacht, wird wahrscheinlich keinen bedeutenden Gewinn für das Verständnis der End-to-End-Latenz bringen.

(A) Verteiltes System (Beispiel)



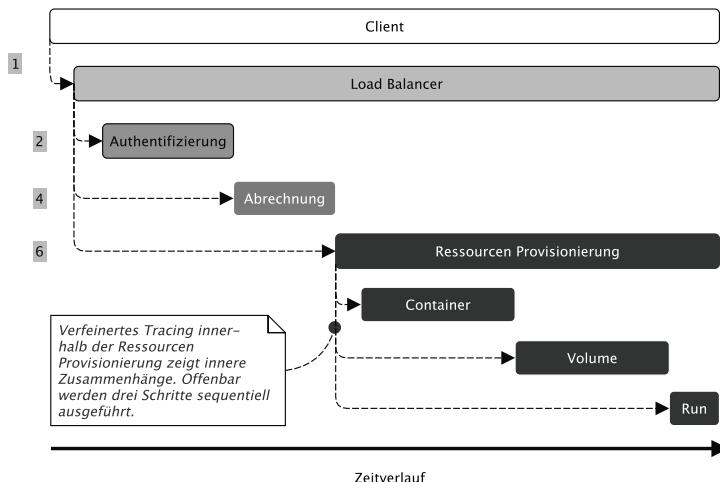
(B) Trace einer Transaktion innerhalb dieses Systems

**Bild 13.5** Tracing-Beispiel

Die Tracing-Instrumentierung ist meist aufwendiger als die Logging- (siehe Abschnitt 13.2.1) oder Monitoring-Instrumentierung (siehe Abschnitt 13.2.2). Der größte Nutzen aus diesem Aufwand entsteht dann, wenn eine Codeabdeckung erreicht wird, die es ermöglicht, für eine gewisse Anzahl hochwertiger Transaktionen End-to-End-Traces generieren zu können. Es ist dabei hilfreich, die Instrumentierung zu visualisieren, um Bereiche zu identifizieren, die weitere Sichtbarkeit durch Instrumentierung benötigen. Diese iterative Art der Instrumentierung veranschaulichen Bild 13.5 und Bild 13.6.

Mit einem konzeptionellen Verständnis (siehe Bild 13.5 B) eines Transaktionsflusses lassen sich Komponenten zielgerichtet instrumentieren. Im gezeigten Beispiel von Bild 13.5 A sollte mit dem gRPC-Service begonnen werden, da dieser die meisten Ein- und Ausgänge hat. Das Web-Framework zu instrumentieren macht ebenfalls Sinn, da es dieses ermöglicht, (erste grobe) End-to-End-Traces zu erstellen.

(A) Verfeinertes End-to-End Tracing einer Transaktion



(B) Optimierung der Transaktion

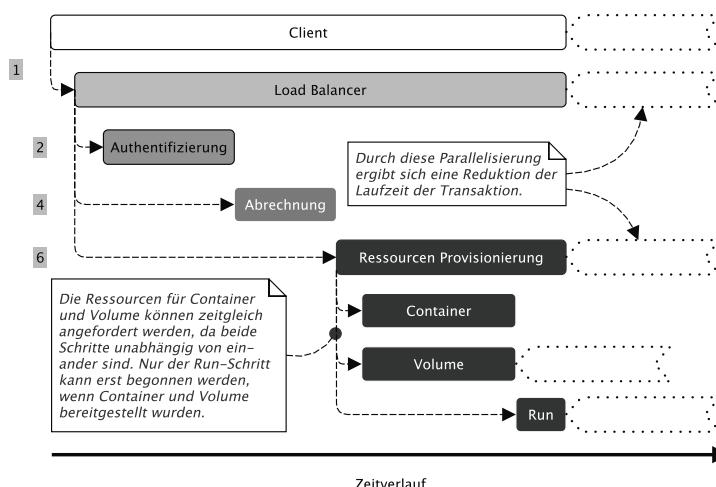


Bild 13.6 End-to-End-Tracing

Auf Basis eines solchen ersten End-to-End-Tracings lässt sich bewerten, wo weitere Tracing-Instrumentierungen perspektivisch den größten Nutzen bringen könnten. Bild 13.5 B zeigt, dass der größte Zeitbedarf die Zuweisung von Ressourcen benötigt. Dies ist somit ein guter Ansatzpunkt, um den Zuweisungsprozess genauer zu untersuchen und dessen beteiligte Komponenten zu instrumentieren. Sobald ergänzend die Ressourcen-Provisionierung instrumentiert wird, kann auch der innere Prozess aufgeschlüsselt werden.

Anhand von Bild 13.6 A lässt sich erkennen, dass die Zeit während der Ressourcen-Provisionierung aufgrund sequenziell ablaufender Prozesse entsteht, deren Laufzeit sich addiert. Durch Optimierungen am Code könnte man Ressourcen gegebenenfalls parallel statt seriell bereitstellen und so zu einem Tracing-Bild wie in Bild 13.6 B gelangen.

Nach einer solchen Optimierung kann auf die oberste Ebene der Ablaufverfolgung nach weiteren grobgranularen Bereichen mit großen Zeitdauern gesucht werden. In diesen kann analog eine feingranularere Instrumentierung eingebracht werden. Erbringen weitere feingranularere Instrumentierungen wenig weiteren Einblick, sollte der Aufwand hingegen in die Instrumentierung einer anderen Transaktion fließen.

13.2.3.2 Tracing-Instrumentierung und Erzeugung von Spans

Für eine solche Tracing-Instrumentierung muss man zumeist eine Tracing-Bibliothek initialisieren. Listing 13.5 zeigt dies am Beispiel der Jaeger-Bibliothek. Üblicherweise wird eine Sampling-Rate definiert, die angibt, wie viel Prozent aller Requests tatsächlich mit Tracing erfasst werden sollen. Insbesondere in Produktivsystemen großer Dimension ist ein 100%-Tracing nur in Ausnahmefällen sinnvoll. Das gezeigte Beispiel definiert beispielsweise ein 25%-Tracing, d. h., nur ein von vier Requests soll im Mittel erfasst werden. Welcher Wert hier genau zu wählen ist, hängt stark vom Anwendungsfall ab. Achten Sie darauf, dass eine Tracing-Rate einen statistisch aussagekräftigen Anteil abdecken sollte. Ferner werden in der Config zumeist Dinge wie das Tracing-Backend angegeben (in Listing 13.5 nicht gezeigt). Wie dies zu erfolgen hat, ist stark von der Tracing-Lösung abhängig, daher wird an dieser Stelle dann auf die entsprechende Dokumentation der Tracing-Lösung bzw. die begleitenden Labs zu diesem Buch verwiesen.

Listing 13.5 Tracing-Instrumentierung

```
from jaeger_client import Config

def init_tracer(service):
    config = Config(config={
        'sampler': {
            'type': 'const',
            'param': 0.25,           # Sampling Rate
        }
    }, service_name=service)
    return config.initialize_tracer()

tracer = init_tracer('hello-world')      # Erzeugung einer Tracer-Instanz
```

Mittels eines tracer-Objekts (Tracer-Instanz) können nun Spans im Verlaufe der Programm-ausführung erzeugt werden. Die hierfür in Listing 13.6 erläuterten Grundfunktionen der OpenTracing-API finden sich so in analoger Form auch in anderen Tracing-Bibliotheken:

- Eine Tracer-Instanz wird verwendet, um neue Spans mittels der `start_span`-Funktion zu erzeugen.
- Jedem Span wird ein Operationsname gegeben, in diesem Fall `say-hello`.
- Die Start- und End-Zeitstempel des Spans werden automatisch von der Tracer-Implementierung erfasst.

Listing 13.6 Annotation von Spans mit Tags und Logs

```
def say_hello(hello_to):
    with tracer.start_span("say-hello") as span: # Span-Erzeugung
        span.set_tag("hello-to", hello_to) # Tag-Annotation
        hello_str = f"Hello, { hello_to }"
        print(hello_str)
        span.log_kv({"event": "helloed"}) # Log Annotation
```

Spans können für aussagekräftige Analysen sowohl mit Tags als auch Logs angereichert werden, um Inhalte einer Transaktion an einem Span zu annotieren. Tags und Logs können bei der Auswertung unter anderem zum Filtern relevanter Teilmengen herangezogen werden. Ein Tag ist ein Schlüssel-Wert-Paar, das bestimmte Metadaten bereitstellt. Ein Log ähnelt einem regulären Log (siehe Abschnitt 13.2.1) und enthält u. a. einen Zeitstempel sowie eine Nachricht aus dem Span, in dem es protokolliert wurde. Tags sind insbesondere dazu gedacht, Attribute eines Spans zu beschreiben, die für die gesamte Dauer des Spans gelten. Wenn ein Span z. B. eine HTTP-Anfrage darstellt, dann kann die URL der Anfrage als Tag aufgezeichnet werden. Wenn der Server hingegen mit einer Umleitungs-URL geantwortet hat, kann dies als Ereignis mit einem eindeutigen Zeitstempel in Form eines Logs protokolliert werden.

13.2.3.3 Serverseitiges Tracing und Extraktion von Span-Kontexten

Das Ziel der serverseitigen Ablaufverfolgung ist es, die Lebensdauer eines Requests an einen Server zu verfolgen und diese Instrumentierung mit einer bereits vorhandenen Ablaufverfolgung eines Downstream-Services oder Clients zu verbinden. Die Ablaufverfolgung einer Serveranfrage folgt dabei zumeist folgendem Schema:

- Ein Server empfängt ein Request.
- Es wird der Trace-Kontext aus dem Interprozess-Transportprotokoll (z. B. HTTP) extrahiert.
- Ein neuer Span wird erzeugt und mit dem Trace-Kontext verknüpft.
- Der aktuelle Trace-Status wird gespeichert.
- Der Server beendet die Request-Bearbeitung und sendet das Response.
- Beenden des erzeugten Spans.

Da dieser Workflow u. a. von der Request-Verarbeitung eines Frameworks abhängen kann, muss der Workflow gegebenenfalls framework-spezifisch ausgeprägt werden. Dies kann durch Filter, Middleware, einen konfigurierbaren Stack oder andere Mechanismen wie etwa Annotationen erfolgen. Um in verteilten Systemen eine Verfolgung über Prozessgrenzen hinweg durchführen zu können, müssen Services in der Lage sein, die Verfolgung von Spans fortzusetzen, die von Downstream-Services oder Clients injiziert wurden. Der OpenTracing-Standard ermöglicht dies durch Inject- und Extract-Methoden, die den Kontext eines Spans

in ein Trägerprotokoll kodieren können. Wenn es einen instrumentierten Request bereits auf der Downstream-Service- bzw. Client-Seite gab, kann dieser Span-Kontext extrahiert und mit neuen Spans verknüpft werden (siehe Bild 13.3 und Listing 13.7). Webservices verwenden beispielsweise HTTP-Header als Träger für diesen Kontext.

Listing 13.7 Extraktion eines Span-Kontexts von einem Downstream-Service oder Client

```
ctx = tracer.extract(opentracing.Format.HTTP_HEADERS, request.headers) # Extraktion
with tracer.start_span("say-hello", child_of=ctx) as span: # Relation
    hello_str = f"Hello, { hello_to }"
    print(hello_str)
```

13.2.3.4 Clientseitiges Tracing und Weiterreichen von Span-Kontexten

Das Aktivieren der clientseitigen Ablaufverfolgung ist für Komponenten erforderlich, die in der Lage sind, selber Requests abzusenden und nicht nur zu bearbeiten. Hierzu muss ein Span in den Header eines Requests injiziert werden, das dann an den Upstream-Service als Request weiterwandert. Genau wie bei der serverseitigen Ablaufverfolgung müsste man hierzu wissen, wie man die Art und Weise ändert, wie Requests gesendet und empfangen werden, um sicherzustellen, dass ein Trace während einer verteilten Request-Bearbeitung entlang mehrerer Servicekomponenten durchgängig sichtbar ist.

Insbesondere Microservices (siehe Kapitel 12) agieren oft sowohl als Client als auch als Server innerhalb eines größeren verteilten Systems, und ihre ausgehenden Client-Requests an Upstream-Services sollten mit dem Request verknüpft werden, die der Service zu diesem Zeitpunkt bearbeitet. Wenn es eine aktive Ablaufverfolgung gibt, startet man einen Span für die Client-Anforderung mit dem aktiven Span als Elternteil. Andernfalls hat der gestartete Span keinen übergeordneten Span.

Listing 13.8 zeigt dies für das Micro-Webframework Flask am Beispiel einer einzigen Route. Bei der Bearbeitung eines Requests wird ein weiterer Request an einen Upstream-Service gesendet. Man sieht daran, dass diese Art der Instrumentierung zum „Durchschleifen“ von Tracing-Daten in HTTP-Headern bereits deutlich aufwendiger ist, da sie sowohl Upstream- als auch Downstream-Tracing-Erfordernisse berücksichtigen muss.

Listing 13.8 Weitergabe eines Span-Kontexts an einen Upstream-Service

```
from flask import Flask, request
app = Flask(__name__)

[...]

@app.route('/resource')
def do_some_service_computing():
    ctx = tracer.extract(opentracing.Format.HTTP_HEADERS, request.headers) #Extraktion
    with tracer.start_span("say-hello", child_of=ctx) as span: #Relation
        headers = { 'user-agent': 'my-service/0.0.1' }
        tracer.inject(span.opentracing.Format.HTTP_HEADERS, headers) #Injektion
        resp = requests.get('https://upstream.svc.com', headers=headers) #Weitergabe
        if resp.statuscode == 200:
            span.set_tag('result': 'success') #Tagging
```

```

    r.log_kv({ 'value' : resp.json() }) #Logging
else:
    span.set_tag('result': 'failure') #Tagging
    resp.log_kv({ 'error' : resp.statuscode, 'message' : r.text }) #Logging
    abort(resp.statuscode, 'Failure in upstream service')

    result = process_further(resp.json())
    return result

```

Solche Art von Aufwänden kann man sich für eine reine Instrumentierung sicher in voller Breite und Tiefe über alle Systemteile eines verteilten Systems meist nicht leisten. Vor allem weil eine solche Instrumentierung häufig überwiegend manuell vorgenommen werden muss. Wesentlich angenehmer wäre es, wenn man zur Herstellung einer Observability bestehende Systeme nachträglich und im Sinne einer Blackbox Observability instrumentieren könnte. Darum soll es im folgenden Abschnitt 13.3 gehen.

■ 13.3 Automatisierte Instrumentierung

Die Instrumentierung – insbesondere beim Tracing von Service-Requests entlang von Service-Ketten – kann durchaus aufwendig sein. Das hat auch u. a. mit der verteilten Microservice-Systemen innenwohnenden Komplexität zu tun. Komplexität verschwindet letztlich nie. Man kann Komplexität aber verschieben, verstecken, kapseln oder isolieren, um sie besser handhaben zu können.

Im Rahmen von Microservice-Architekturen haben sich dafür sogenannte Service-Meshs etabliert, die eine Netzwerkabstraktion für containerbasierte Applikationen und Services erzeugen, um deren Management zu vereinfachen und Aspekte wie Traffic-Management, Zuverlässigkeit, Beobachtbarkeit und Sicherheit innerhalb von microservice-basierten Architekturen zu vereinfachen und querschnittlich zu lösen. Die Entwicklung und der Betrieb eines einzelnen Microservice sind zwar einfach, manchmal fast schon trivial.

Die Gesamtkomplexität eines Problems verschwindet aber nicht. Wenn die einzelnen Komponenten eines Systems an Komplexität einbüßen, muss die Komplexität des Grundproblems aber „irgendwohin“. Die Komplexität wandert bei Microservice-Architekturen letztlich in eine komplexere Service-to-Service-Interaktion. Der Betrieb komplexer Service-of-Service-Systeme bringt somit weitere Herausforderungen mit sich.

- Inhärente Komplexität
- Geschäftslogik ist über voneinander unabhängige Microservices verteilt.
- Sicherheit (komplexere Zugriffskontrolle und vergrößerte Attack Surfaces)
- Verteiltes Logging, Traceability

Service-Meshs lösen diese Herausforderungen außerhalb einer Applikation in einer explizit dafür vorgesehenen Schicht.

13.3.1 Eigenschaften von Service-Meshs

Anwendungen und Dienste benötigen häufig zusammengehörige Funktionen, wie z. B. Überwachung, Protokollierung und Konfiguration. Solche kohärenten Peripherieaufgaben können als separate Komponenten oder Dienste implementiert werden. Werden diese Aufgaben in einem eigenen Prozess oder Container bereitgestellt, um Plattformdienste sprachübergreifend bereitzustellen, spricht man von einem Sidecar. Proxys in Service-Meshs werden somit häufig als Sidecars in Containern bereitgestellt, die jeder Serviceinstanz zur Seite gestellt werden.

Ein Service-Mesh besteht – vereinfacht gesprochen – aus einer Kontrollebene (Control Plane), die Proxys steuert (siehe Bild 13.7). Die Proxys sind Serviceinstanzen zugeordnet (Sidecar Pattern) und spannen eine Datenschicht (Data Plane) für Services auf. Die Instrumentierung einer Cloud-native Applikation mittels eines Service-Meshs erfolgt also nicht (wie in Abschnitt 13.2) sprachspezifisch innerhalb einzelner Komponenten der Applikation, sondern außerhalb der Komponenten einer Applikation über beigestellte Proxy-Container.

Wesentliche Komponenten und deren Verantwortlichkeiten in einem Service-Mesh sind demnach:

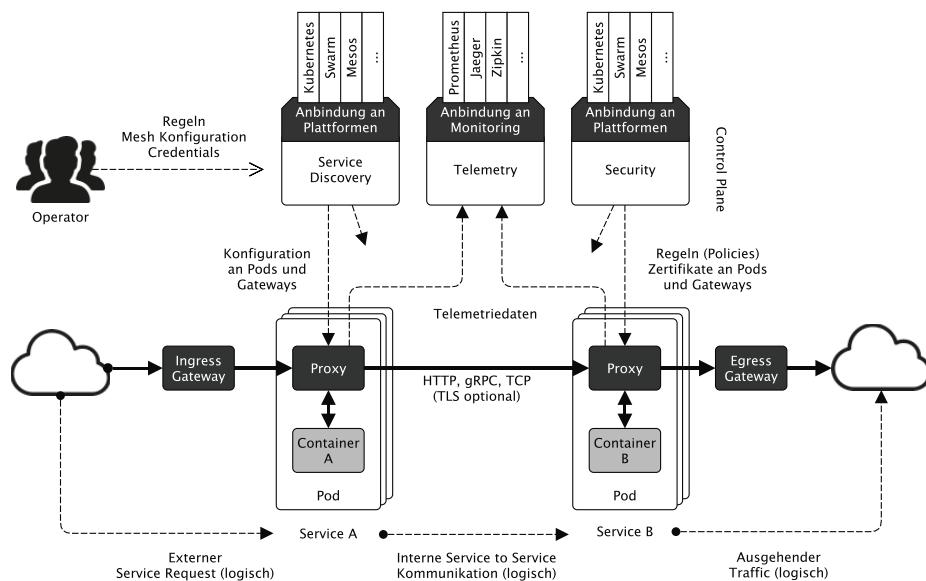


Bild 13.7 Service-Mesh

- **Proxy:** Jeder Service kommuniziert innerhalb eines Service-Meshs über einen dedizierten Proxy (Sidecar Pattern). Die gesamte Kommunikation im Mesh erfolgt somit über diese Proxys. Proxys können so das Kommunikationsverhalten anreichern und Telemetriedaten (siehe Abschnitt 13.2) sammeln sowie Funktionen im Rahmen des Traffic-Managements übernehmen. Die Services zur Seite gestellten Proxys lassen sich von Orchestrierungs-Frameworks verwalten, um z. B. bei jedem Deployment automatisch platziert zu werden. Die Komponenten einer Cloud-native Applikation werden so automatisiert instrumentiert.

- **Control Plane:** Ein Service-Mesh wird über eine Control Plane verwaltet. Hier lassen sich Routen und Regeln für das Traffic-Management vornehmen. Im Rahmen des Resilienz-Managements lassen sich ferner Regeln zu Timeouts, Retries etc. konfigurieren. Über die Control Plane erfolgt ferner die Service Discovery, die Durchsetzung von Access- und Policy-Richtlinien, sowie die Konfiguration der Erhebung von an Proxys erhebbaren Telemetrydaten.
- **Data Plane:** Auf der Data Plane erfolgt die Service-to-Service-Kommunikation. Dies umfasst auch die eingehende (Ingress) und ausgehende (Egress) Kommunikation des Service-Meshs. Alle Kommunikation innerhalb der Data Plane wird transparent für die verwalteten Services über die Proxys umgeleitet.

Dabei stellen Service-Meshs vielfältige Zuverlässigkeit- und Sicherheitsmechanismen bereit, die nicht im Servicecode realisiert werden müssen. Dies umfasst unter anderem Resilienz-Patterns (siehe auch Abschnitt 12.3) wie beispielsweise:

- Circuit-Breaker
- Timeouts + Retries
- Latenzbasiertes Load Balancing
- Health-aware Service Discovery

Am Markt sind dabei nicht notwendig zueinander kompatible Service-Mesh-Lösungen entstanden. Mit dem Service-Mesh-Interface (SMI) etabliert sich aber immerhin ein Standard für die Container-Plattform Kubernetes (CNCF 2019). Man muss allerdings konstatieren, dass die Standardisierung in diesem Bereich noch nicht so weit fortgeschritten ist wie in anderen Bereichen des Cloud-nativen Computings. SMI definiert einen Standard, der von Lösungen unterschiedlicher Anbieter implementiert werden kann. Dies ermöglicht sowohl eine Standardisierung für Endbenutzer als auch Innovationen durch Anbieter von Service-Mesh-Technologie. SMI ermöglicht Flexibilität und Interoperabilität und deckt die gängigsten Service-Mesh-Funktionen ab. Dabei wird der folgende grundlegende Funktionsumfang von Service-Meshes definiert:

- **Traffic Policies** legen fest, wie Richtlinien zu Identität und Transportverschlüsselung zwischen Services definiert und durchgesetzt werden können.
- Die **Traffic Telemetry** umfasst die Erfassung quantitativer Messdaten wie Fehlerrate und Latenz zwischen Services.
- Das **Traffic-Management** leitet Verkehr zwischen Services innerhalb eines Service-Meshs.

Üblicherweise werden diese Funktionalitäten in Form von Feature-Umfängen zu

- Traffic-Management,
- Resilienz,
- Sicherheit (Security)
- sowie Management und Analyse von Verkehrstopologien

von Service-Mesh-Lösungen angeboten, auf die in den folgenden Abschnitten detaillierter eingegangen werden soll. Dabei orientieren wir uns so weit es geht am SMI-Standard (CNCF 2019). Wo dies nicht geht, veranschaulichen wir Prinzipien am Beispiel von *Istio*, einer der zum Zeitpunkt des Erscheinens dieses Buchs umfangreichsten Service-Mesh-Lösung.

Betreibt man eine Cloud-native Applikation auf der Orchestrierungsplattform Kubernetes, so reicht es mit *istio* aus, den Namespace dieser Anwendung entsprechend zu labeln. Wird beispielsweise das Label `istio-injection=enabled` mittels

```
kubectl label ns my-cloud-native-app istio-injection=enabled
```

für einen Namespace gesetzt, wird allen neuen Pods, die in diesem Namespace erstellt werden, automatisch ein Proxy als Sidecar hinzugefügt und so das Service-Mesh aufgespannt. Die Instrumentierung erfolgt also vollkommen außerhalb der Container-Images der Applikation. Zusätzliche Funktionalitäten eines Service-Meshs im Bereich des Traffic-Managements, der Resilienz und der Sicherheit können nun mittels zusätzlicher Ressourcen eingebracht und aktiviert werden.

13.3.2 Traffic-Management

Mittels des Service-Mesh-Interface-(SMI-)Standards können unter anderem Zugriffsrichtlinien für Komponenten von Anwendungen definiert werden. Die Zugriffskontrolle ist bei SMI dabei additiv. Datenverkehr wird standardmäßig verweigert, es sei denn, es werden erlaubende Zugriffsregeln definiert. Ein Traffic Target ordnet dabei eine Reihe von Traffic Definitions (Regeln) einer Dienstidentität zu, die wiederum einer Gruppe von Pods zugeordnet ist. Der Zugriff wird über referenzierte Traffic Specs und eine Liste von Quelldienstidentitäten gesteuert. Jeder Pod, der versucht, eine Verbindung herzustellen und nicht in der definierten Liste der Quellen enthalten ist, wird abgelehnt. Der Zugriff wird basierend auf der Dienstidentität (Kubernetes-Service-Accounts) gesteuert. Regeln sind Traffic Definitions, die definieren, wie zulässiger Verkehr für bestimmte Protokolle aussehen darf. Ein gültiges Traffic Target muss ein Ziel, mindestens eine Regel und mindestens eine Quelle angeben.

Listing 13.9 zeigt einen häufigen Anwendungsfall, den Zugriff auf Metriken so einzuschränken, dass diese nur von Prometheus abgefragt werden dürfen.

Listing 13.9 Zugriffssteuerung mittels Service-Meshs

```
kind: TCPRoute
metadata:
  name: tcp-traffic
spec:
  matches:
    ports: [8080]
---
kind: HTTPRouteGroup
metadata:
  name: http-traffic
spec:
  matches:
    - name: metrics
      pathRegex: "/metrics"
      methods: ["GET"]
    - name: everything
```

```

pathRegex: ".*"
methods: ["*"]

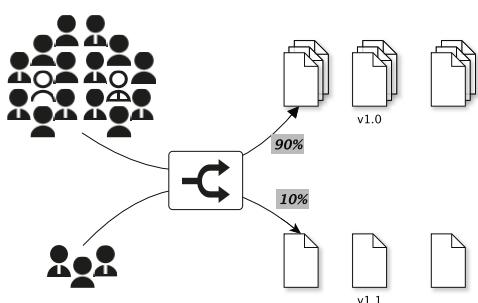
---
kind: TrafficTarget
metadata:
  name: path-specific
  namespace: default
spec:
  sources:          #Quellen
    - kind: ServiceAccount
      name: prometheus
      namespace: default
  destination:      #Ziele
    kind: ServiceAccount
    name: service-a
    namespace: default
  rules:             #Traffic zwischen Quellen und Zielen
    - kind: TCPRoute
      name: tcp-traffic
    - kind: HTTPRouteGroup
      name: http-traffic
      matches: ["metrics"]

```

Mittels des SMI-Standards kann man also Datenverkehr protokollspezifisch selektieren. Es wird zusammen mit der Zugriffskontrolle und anderen Richtlinien verwendet, um zu definieren, was mit bestimmten Arten von Verkehr geschehen soll, wenn dieser durch das Netz fließt. Auf diese Weise lässt sich Datenverkehr protokollspezifisch definieren und beispielsweise Regelungen wie einem Traffic-Split unterwerfen.

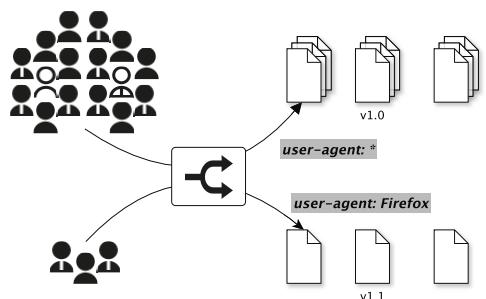
Mittels Traffic-Splits lässt sich im einfachsten Fall der prozentuale Anteil des Datenverkehrs zwischen verschiedenen Services schrittweise steuern. Hierdurch kann man in einen Service eingehenden Datenverkehr auf verschiedene Ziele aufteilen (siehe Bild 13.8 A). Traffic-Splits kann man beispielsweise verwenden, um Canary-Versionen für neue Softwareversionen zu orchestrieren – also Traffic erst einmal nur zu einem kleinen Anteil auf eine neue Version zu lenken, um Erfahrungen im Betrieb einer neuen Version zu gewinnen.

(A) Traffic-Split mittels Zufall



z.B. für Canary-Releases, kontrollierte Trafficsteigerung auf einen neuen Service

(B) Traffic-Split mittels Request-Introspektion



z.B. um Nutzer spezifischer Webbrowser auf für diese Webbrowser optimierte Seiten zu lenken

Bild 13.8 Traffic-Management

Listing 13.10 Traffic-Splits für Canary-Releases nutzen

```

kind: TrafficSplit
metadata:
  name: canary
spec:
  service: website          #Traffic-Split für Service definieren
  backends:
    - service: website-v1.0   #90% gehen an v1.0 des Website-Service
      weight: 90
    - service: website-v1.1   #10% gehen an v1.1 des Website-Service
      weight: 10
  
```

Die Gewichtung des Verkehrs zwischen verschiedenen Services kann auch für A/B-Test-szenarien genutzt werden. Hierzu kann ein Traffic-Split etwa mittels HTTP-Header-Filtern kombiniert werden, um einen bestimmten Benutzeranteil an ein Backend weiterzuleiten, während alle anderen Benutzer (Kontrollgruppe), die nicht zu diesem Segment gehören, an das Standard-Backend weitergeleitet werden. Hierzu können HTTP-Header-Filter definiert werden, mittels derer Traffic-Splits anhand von HTTP-Routen vorgenommen werden können.

Listing 13.11 Traffic-Splits für A/B-Tests nutzen

```

kind: HTTPRouteGroup
metadata:
  name: ab-test
matches:
  - name: firefox-users
    headers:
      - user-agent: ".*Firefox.*"  #Selektiere Traffic von Firefox-Browsern
    ---
kind: TrafficSplit
metadata: { name: ab-test }
spec:
  service: website
  matches:
    - kind: HTTPRouteGroup
      name: ab-test
    backends:
      - service: website-v1.1      #Firefox-Browser landen immer hier
        weight: 100
      - service: website-v1.0      #und niemals hier
        weight: 0
  
```

Das in Listing 13.11 gezeigte Beispiel realisiert den in Bild 13.8 B gezeigten Fall. Hier wird eine Route definiert, um Nutzer anhand ihres Browsers (User-Agent) selektieren zu können. Im gezeigten Beispiel werden alle Firefox-Nutzer selektiert und auf die Version V1.1 des Services gelenkt. Analog lassen sich so natürlich auch andere Header-Informationen auswerten, um z. B. den Traffic einer Nutzergruppe, die einem Betatest zugestimmt hat, auf besondere Services zu lenken.

13.3.3 Resilienz

Die folgenden Fähigkeiten im Bereich der Resilienz und der Sicherheit werden (noch) nicht vom SMI-Standard abgedeckt. Daher werden derartige Service-Mesh-Fähigkeiten am Beispiel von *Istio* erläutert. Die Prinzipien sind aber auf andere Service-Meshs durchaus übertragbar und werden absehbar auch irgendwann in den SMI-Standard einfließen.

Virtual Services sind ein wichtiger Baustein der Traffic-Routing-Funktionalität von *Istio*. Mit einem virtuellen Service lässt sich konfigurieren, wie Requests an einen Service innerhalb eines Istio-Service-Meshs weitergeleitet werden. Jeder virtuelle Service besteht aus einem Satz von Routing-Regeln, die der Reihe nach ausgewertet werden, sodass *Istio* jeden Request an den virtuellen Service mit einem bestimmten realen Service innerhalb des Meshs abgleichen kann. Virtuelle Services entkoppeln Consuming-Services von Providing-Services durch einen Routing-Mittler. Mittels virtuellen Services hat man so umfangreiche Möglichkeiten, verschiedene Regeln für das Routing von Datenverkehr in einem Mesh festzulegen. Damit lassen sich unter anderem Canary- und A/B-Testszenarien – wie bereits am SMI-Standard gezeigt – abbilden.

Ergänzend lassen sich aber auch sogenannte Resilienz-Pattern durch virtual Services definieren. Unter Resilienz versteht man die Widerstandsfähigkeit von Systemen gegenüber Teilsystemausfällen. Resilienz lässt sich in Systemen unter anderem durch Wahl geeigneter Einstellungen von

- Timeouts
- Wiederholungen
- und Circuit-Breaker
- erreichen.

Ein **Timeout** ist die Zeitspanne, die ein Proxy auf Antworten von einem bestimmten Service warten sollte, um sicherzustellen, dass Requests innerhalb eines vorhersehbaren Zeitrahmens erfolgreich sind oder fehlschlagen. Timeout für HTTP-Anfragen sind in *Istio* standardmäßig deaktiviert. Für einige Anwendungen und Services ist der Standard-Timeout des Betriebssystems jedoch möglicherweise nicht geeignet. Ein zu langer Timeout könnte beispielsweise zu einer übermäßigen Latenz in Fehlersituationen führen, während ein zu kurzes Timeout dazu führen könnte, dass Requests unnötig fehlschlagen, während man auf die Rückkehr einer Operation wartet, an der mehrere Services beteiligt sind.

Um optimale Timeout-Einstellungen zu finden und zu verwenden, lassen sich mit *Istio* Timeouts auf einfache Weise auf Ebene virtueller Services dynamisch pro Service anpassen, wie Listing 13.12 zeigt.

Listing 13.12 Timeouts in Istio setzen

```
kind: VirtualService
metadata: { name: ratings }
spec:
  hosts: [ratings]
  http:
    - route:
        - destination: { host: ratings, subset: v1 }
        timeout: 10s #Timeout setzen
```

Wiederholungen (Retrys) geben an, wie oft ein Proxy maximal versucht, eine Verbindung zu einem Service herzustellen, wenn der erste Aufruf fehlschlägt. Wiederholungen können die Verfügbarkeit von Services und damit die Widerstandsfähigkeit des Gesamtsystems gegenüber Fehlerzuständen verbessern. Wiederholungen stellen sicher, dass Requests nicht dauerhaft aufgrund von vorübergehenden Problemen wie einem überlasteten Service oder Netzwerk fehlschlagen. Das Intervall zwischen den Wiederholungsversuchen ist variabel und wird von *Istio* automatisch festgelegt, um zu verhindern, dass der aufgerufene Dienst mit Anfragen überlastet wird. Das Standard-Wiederholungsverhalten für Requests sind etwa zwei Wiederholungsversuche.

Wie bei Timeouts kann aber auch das Wiederholungsverhalten servicespezifisch konfiguriert werden. Listing 13.13 konfiguriert maximal drei Wiederholungsversuche, um nach einem anfänglichen Aufruffehler eine Verbindung herzustellen, jeweils mit einem Timeout von zwei Sekunden.

Listing 13.13 Retrys in Istio setzen

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata: { name: ratings }
spec:
  hosts: [ratings]
  http:
    - route:
        - destination: { host: ratings, subset: v1 }
        retries:          # Retry
          attempts: 3      # Verhalten
          perTryTimeout: 2s # definieren
```

Circuit-Breaker sind ein weiterer Mechanismus zur Optimierung der Resilienz in servicebasierten Anwendungen. Mittels eines Circuit-Breakers werden Request-Limits für Services festgelegt, z. B. die Anzahl der gleichzeitigen Verbindungen oder wie oft Aufrufe an einen Service fehlschlagen dürfen. Sobald dieses Limit erreicht ist, wird der Circuit-Breaker „ausgelöst“ und stoppt weitere Verbindungen. Die Verwendung eines Circuit-Breaker-Musters ermöglicht einen schnellen Ausfall von Services (Fail early), anstatt dass Clients versuchen, eine Verbindung zu überlasteten oder ausgefallenen Services herzustellen.

Wie bei Timeouts und Wiederholungen können auch Circuit-Breaker-Settings service-spezifisch konfiguriert werden. Listing 13.14 begrenzt exemplarisch die Anzahl der gleichzeitigen Verbindungen auf 100 Verbindungen.

Listing 13.14 Circuit-Breaker in Istio setzen

```
kind: DestinationRule
metadata: { name: reviews }
spec:
  host: reviews
  subsets:
    - name: v1
      labels:
```

```

version: v1
trafficPolicy:          # Circuit-Breaker
connectionPool:         # Pattern
  tcp: { maxConnections: 100 } # definieren

```

Alle drei die Resilienz steigernden Mechanismen (Timeouts, Retries, Circuit-Breaker) müssen dabei nicht in der Applikation selber gesetzt werden, sondern können von außen nachträglich „injiziert“ und konfiguriert werden.

13.3.4 Sicherheit

Die Zerlegung einer monolithischen Anwendung in atomare Services bietet verschiedene Vorteile, darunter eine bessere Agilität, eine bessere Skalierbarkeit und eine bessere Wiederverwendbarkeit von Services. Allerdings haben Microservices auch besondere Sicherheitsanforderungen:

- Um sich gegen Man-in-the-Middle-Angriffe zu schützen, benötigt man eine Verschlüsselung des Datenverkehrs.
 - Um eine flexible Service-Zugriffskontrolle zu ermöglichen, benötigt man feinkörnige Zugriffsrichtlinien.
 - Um festzustellen, wer was zu welchem Zeitpunkt getan hat, benötigt man Auditing-Tools.
- Auch solche Sicherheitserfordernisse adressieren Service-Meshs oft mittels Authentifizierung, Autorisierung, Verschlüsselung der Service-Communication, TLS-Endpunkt-Termination und transparentes Zertifikat-Handling (TLS).

Service-Meshs setzen Kommunikationsregeln dabei flexibler auf Basis von Serviceidentitäten (logisch) anstatt Network Controls (physisch) durch. Die Sicherheitsfunktionen von Istio bieten beispielsweise eine transparente TLS-Verschlüsselung und Tools für Authentifizierung, Autorisierung und Audit (AAA). Die Leitgedanken dabei sind:

- **Security by Default:** Es sollen keine Änderungen am Anwendungscode und an der Infrastruktur erforderlich werden.
- **Defense in Depth:** Integration mit bestehenden Sicherheitssystemen zur Bereitstellung mehrerer Verteidigungsebenen.
- **Zero-Trust Networking:** Aufbau von Sicherheitslösungen auf vertrauensunwürdigen Netzwerken.

Zero-Trust-Netzwerke beschreiben einen Ansatz für das Design und die Implementierung von IT-Netzwerken. Das Hauptkonzept hinter Zero Trust ist, dass vernetzten Geräten, wie z. B. Laptops, standardmäßig nicht vertraut werden sollte, selbst wenn sie mit einem verwalteten Unternehmensnetzwerk wie dem Firmen-LAN verbunden sind und selbst wenn sie zuvor verifiziert wurden. Viele Unternehmensnetzwerke bestehen aus vielen miteinander verbundenen Segmenten, cloud-basierten Diensten und Infrastrukturen sowie Verbindungen zu entfernten und mobilen Umgebungen und zunehmend auch Verbindungen zu nichtkonventioneller IT wie IoT-Geräten.

Der traditionelle Ansatz, Geräten innerhalb eines fiktiven Unternehmensperimeters oder Geräten, die über ein Virtual Private Network (VPN) damit verbunden sind, zu vertrauen, macht in solch hochgradig diversifizierten und verteilten Umgebungen zunehmend weniger Sinn.

Hier wird vielmehr auf eine gegenseitige (mutual) Authentifizierung oder Zwei-Wege-Authentifizierung gesetzt, die sich in einem Authentifizierungsprotokoll gleichzeitig gegenseitig authentifizieren. Gegenseitige Authentifizierung ist in einigen Protokollen (beispielsweise SSH) vorgegeben und in anderen (z. B. TLS) optional. Standardmäßig weist das TLS-Protokoll dem Client nur die Identität des Servers mit X.509-Zertifikaten nach, und die Authentifizierung des Clients gegenüber dem Server wird der Anwendungsschicht überlassen. TLS bietet jedoch auch eine Client-zu-Server-Authentifizierung unter Verwendung der clientseitigen X.509-Authentifizierung an. Da dies die Bereitstellung der Zertifikate für die Clients erfordert und weniger benutzerfreundlich ist, wird sie nur selten in Endbenutzeranwendungen eingesetzt.

Die gegenseitige TLS-Authentifizierung (mTLS) ist in Business-to-Business-(B2B-)Anwendungen wesentlich weiter verbreitet, da hier die Sicherheitsanforderungen im Vergleich zu Endkundenumgebungen in der Regel viel höher sind. Die gegenseitige Authentifizierung ermöglicht das Zero-Trust Networking, da sie die Kommunikations- und Informationsintegrität gewährleistet, indem u. a. folgende Angriffe verhindert werden.

- Von **Man-in-the-Middle-Angriffen** (MITM) spricht man, wenn eine dritte Partei eine Nachricht abhören oder abfangen möchte und dabei manchmal die beabsichtigte Nachricht für den Empfänger verändert.
- Ein **Replay-Angriff** ähnelt einem MITM-Angriff, bei dem ältere Nachrichten aus dem Kontext gerissen wiedergegeben werden, um den Server zu täuschen.
- **Spoofing-Angriffe** beruhen auf der Verwendung falscher Daten, um sich als ein anderer Benutzer auszugeben, um Zugriff auf einen Server zu erhalten oder als jemand anderes identifiziert zu werden.

In Cloud-nativen Systemen wird mTLS zunehmend häufiger durch Service-Meshs transparent bereitgestellt. *Istio* realisiert dies etwa durch mehrere Komponenten der Control Plane, die die sichere Konfiguration der Data Plane automatisiert sicherstellt:

- Eine **Zertifizierungsstelle (CA)** für die Schlüssel- und Zertifikatsverwaltung (Mutual TLS)
- Ein **Konfigurations-API-Server** verteilt Authentication Policys, Authorisation Policys und Secure-Naming-Informationen an die Proxys.
- Sidecar- und Perimeter-Proxys arbeiten als **Policy Enforcement Points (PEPs)**, um die Kommunikation zwischen Clients und Servern zu sichern.
- Ferner gibt es *Istio*-spezifische Envoy-Proxy-Erweiterungen zur Verwaltung von Telemetrie und Auditingfunktionalitäten.

Die Authentifizierung für Services kann mithilfe von Peer- und Request-Authentifizierungsrichtlinien festgelegt werden. Diese Authentication Policys werden im *Istio*-Konfigurationsspeicher gespeichert. Der *Istio*-Controller überwacht den Konfigurationsspeicher und sendet diese Policys asynchron an die Ziel-Endpunkte. Sobald der Proxy eine Policy empfängt, wird diese sofort auf diesem Pod wirksam. Dabei werden zwei Arten der Authentifizierung unterschieden:

- Peer-Authentifizierung wird für die Service-to-Service-Kommunikation verwendet, um den Client zu verifizieren, der die Verbindung herstellt. Hierbei wird mTLS für die Transport-Authentifizierung verwendet.
- Request-Authentifizierung wird für die Endbenutzer-Authentifizierung verwendet, um den an die Anfrage angehängten Berechtigungsnachweis zu verifizieren. Dabei werden zumeist die Authentifizierung mit JSON Web Token-(JWT-)Validierung und die Verwendung eines benutzerdefinierter Authentifizierungsanbieters (OAUTH2) ermöglicht.

Mittels Peer-Authentifizierungsrichtlinien kann durchgesetzt werden, welche Kommunikationskanäle mTLS unterworfen werden und welche nicht. Dies erfolgt bei *Istio* mittels der folgenden Modi:

- **PERMISSIVE**: Workloads akzeptieren sowohl mTLS als auch Klartextverkehr.
- **STRICT**: Workloads akzeptieren nur mTLS-Verkehr.
- **DISABLE**: mTLS ist deaktiviert.

Die Richtlinie in Listing 13.15 sorgt dafür, dass Kommunikation mit Pods des Review-Service verschlüsselt erfolgt.

Listing 13.15 Authentication Policy in Istio setzen

```
kind: PeerAuthentication
metadata: { name: example-peer-policy }
spec:
  selector:
    matchLabels: { app: reviews }
  mtls:
    mode: STRICT
```

Autorisierungsfunktionen bieten hingegen eine mesh-, namespace- und workload-weite Zugriffskontrolle für Workloads in einem Mesh. Jeder Proxy führt hierzu eine Autorisierungs-Engine aus, die Requests zur Laufzeit autorisiert. Wenn ein Request beim Proxy eingeht, wertet die Autorisierungs-Engine den Anfragekontext anhand der aktuellen Authorisation Policy aus und gibt das Autorisierungsergebnis (ALLOW oder DENY) zurück.

Autorisierungsrichtlinien können dabei als ALLOW- oder DENY-Regeln formuliert werden. DENY-Regeln haben Vorrang vor den ALLOW-Regeln. Beziehen sich mehrere Authorisation Policy's auf denselben Workload, wendet Istio diese additiv an. Mittels Selektoren kann die Anwendung von Authorisation Policy's auf bestimmte Workloads eingeschränkt werden. Selektoren enthalten hierzu (Kubernetes-üblich) eine Liste von {Schlüssel: Wert}-Paaren, wobei der Schlüssel der Name des Labels ist. Authorisation Policy's ohne Selektoren gelten für alle Workloads im gleichen Namensraum der Authorisation Policy.

Listing 13.16 verweigert beispielsweise Requests, wenn die Quelle nicht aus dem Namensraum `foo` stammt. Auf diese Weise lassen sich etwa Namensräume in Multi-Tenancy-Kubernetes-Clustern voneinander isolieren und Netzwerk-Traffic von einem Namensraum in einen anderen Namensraum unterbinden. Eine Netzwerkséparation lässt sich allerdings auch mit den *Network Policy's* von Kubernetes realisieren (jedoch nicht mit allen Overlay-Networks, siehe Abschnitt 9.3.9.3).

Listing 13.16 Authorisation Policy in Istio setzen

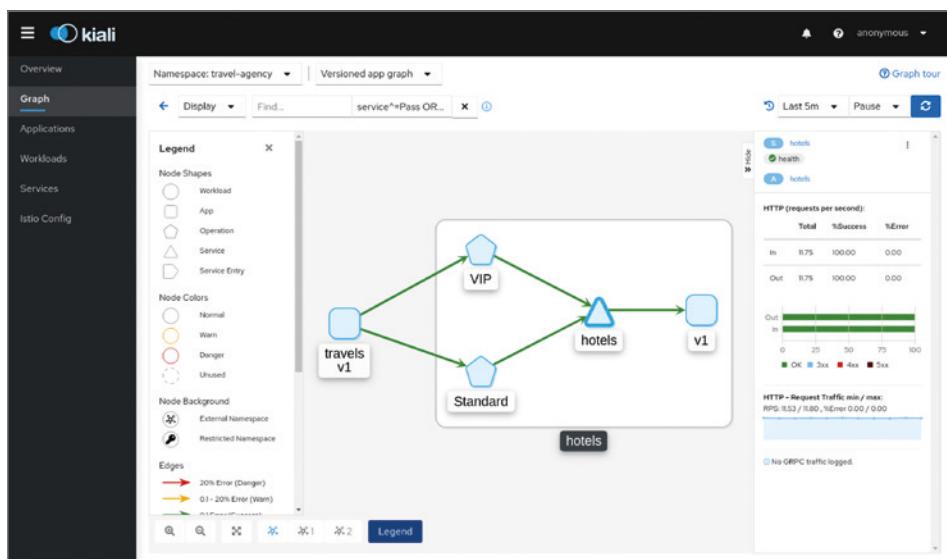
```

kind: AuthorizationPolicy
metadata:
  name: httpbin-deny
  namespace: foo
spec:
  selector:          # Ziel-Pods
    matchLabels:
      app: httpbin
      version: v1
  action: DENY        # Policy
  rules:             # Quellen
  - from:
    - source:
      notNamespaces: ["foo"]

```

13.3.5 Management und Analyse von Verkehrstopologien

Oft bieten Service-Mesh-Lösungen auch eine Art Management-Konsole zur Bedienung und Konfiguration eines Service-Meshs. Kiali ist etwa eine webbasierte Management-Konsole für *Istio*-basierte Service-Meshs und bietet Dashboards, Beobachtungsmöglichkeiten sowie Konfigurations- und Validierungsfunktionen, um die in den vorhergehenden Abschnitten gezeigten Möglichkeiten mittels einer webbasierten Oberfläche konfigurieren und analysieren zu können.

**Bild 13.9** Visualisierung von Verkehrstopologien (Quelle: Kiali)

Derartige Managementlösungen können auch Verkehrstopologien Cloud-nativer Anwendungen ableiten und visualisieren. Hierzu werden die Verkehrsströme entlang der Service-Mesh-Proxys ausgewertet. Da aller Verkehr innerhalb des Service-Meshs über diese Proxys abgewickelt wird, können diese Zugangspunkte genutzt werden, um Verkehrstopologien auch für unbekannte Microservice-Architekturen mit Hunderten oder gar Tausenden Microservices abzuleiten. Kiali kann filterbare Verkehrstopologien darstellen und detaillierte Metriken und Auswertungen entlang dieser Verkehrstopologien annotieren (siehe Bild 13.9), was das Auffinden von Problemen in komplexen Service-Topologien großer Cloud-nativer Anwendungen erheblich vereinfachen kann.

■ 13.4 Zusammenfassung

Die Anforderung an die Beobachtbarkeit von Cloud-nativen Anwendungen ergibt sich u. a. aus dem DevOps-Kontext und der evolutionären Entwicklung. In einem solchen Kontext ist man gezwungen, Änderungen „am schlagenden Herzen“ vorzunehmen. Weil Cloud-native Anwendungen komplexe Systeme sind, können nicht alle Effekte von Änderungen vorhergesehen werden. Die Wirkungsweise und das Systemverhalten müssen also kontinuierlich beobachtbar sein, damit ungewünschte Effekte schnell erkannt und behoben sowie zukünftig vermieden werden können.

Dieses Kapitel hat gezeigt, dass man hierzu „Messfühler“ in Cloud-nativen Anwendungen platzieren kann, die über einzelne Aspekte des Systemzustands Aufschluss geben, jedoch für ein ganzheitliches Bild konsolidiert und aufbereitet werden sollten (siehe Abschnitt 13.1). Leser, die sich für den etwas mehr metrikfokussierten Grafana/Prometheus-Ansatz interessieren, seien hiermit auf (Bastos und Araújo 2019) verwiesen. Der in diesem Handbuch eher verfolgte ELK-Stack wird im Detail in (Shukla und Kumar 2019) behandelt. Mit beiden Lösungs-Stacks kann man letztlich ein „konsolidiertes Lagebild“ von Systemen aufbauen.

Die konsolidierten Daten müssen jedoch in Systemen abgegriffen werden. Wie wir gesehen haben, kann dies in einem Whitebox-Ansatz erfolgen, indem man Programmcode mittels entsprechenden Logging-, Metrik- oder Tracing-Bibliotheken instrumentiert (siehe Abschnitt 13.2). Leser, die sich insbesondere für diesen Whitebox-Ansatz interessieren, seien unter anderem auf (Parker u. a. 2020), aber auch (Julian 2017) verwiesen.

Abschnitt 13.3 hat aber auch gezeigt, dass eine Blackbox-Instrumentierung automatisiert mittels Service-Meshs erfolgen kann. Mittels Service-Meshs kann man die Resilienz und Sicherheit von Cloud-nativen Systemen konfigurieren sowie ein Traffic-Management realisieren, das sowohl Canary- als auch A/B-Teste ermöglicht. Ferner lassen sich mit solchen Lösungen auch komplexe Verkehrstopologien definieren, überwachen, analysieren und visualisieren. Erste Standardisierungen erfolgen zumindest für Kubernetes durch die SMI-Spezifikation (CNCF 2019). Dennoch ist die Landschaft noch sehr heterogen, und die einzelnen Lösungen sind in ihren Details nicht zueinander kompatibel. *Istio* ist sicher eine Lösung, die aktuell einen der umfangreichsten Funktionsumfänge zu bieten hat (Calcote und Butcher 2019). Einen umfassenderen Überblick über die aktuelle Service-Mesh-Landschaft bietet (Khatri und Khatri 2020).

Für eine ganzheitliche Beobachtbarkeit sollten aber sowohl die Whitebox- als auch die Blackbox-Instrumentierung sich gegenseitig ergänzend eingesetzt werden.

Ergänzende Materialien



<https://bit.ly/3kS53n2>

Auf der Website zum Buch finden sich zu diesem Kapitel ferner folgende Ergänzungsmaterialien:

- Praktische Übungen (Labs) zu den Themen Beobachtbarkeit, Traffic-Management, Visualisierung von Verkehrstopologien mittels Service-Meshs sowie Tracing, Logging und Log-Konsolidierung
- Übersichten inklusive Links zu Zeitreihen-Datenbanken
- Übersichten inklusive Links zu Instrumentierungsbibliotheken für Logging, Monitoring und Tracing
- Übersichten inklusive Links zu Service-Mesh-Produkten
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer (Themen: Beobachtbarkeit von Systemen, Konsolidierung von Telemetriedaten, Logs, Metriken, Tracing; Service-Meshs: Traffic-Management, Verkehrstopologien, Resilienz, Sicherheit)

14

Domain-driven Design

„The model is not the diagram.“

Eric Evans, Autor von „Domain-driven Design: Tackling Complexity in the Heart of Software“

Gemäß Kapitel 12 sollten Services in Cloud-nativen Systemen u. a. folgenden Prinzipien entsprechen und folgende Eigenschaften haben:

- Die Dekomposition des Gesamtsystems sollte mittels Services erfolgen.
- Services sollten dabei unabhängig voneinander austausch- und aktualisierbar sein.
- Um dies zu gewährleisten, sollte eine lose Kopplung von Services angestrebt werden.
- Innerhalb von Services sollte eine hohe Kohäsion (Single-Responsibility-Prinzip) herrschen.
- Das Gesamtsystem folgt einem evolutionären Design und einer evolutionären Entwicklung.
- Hierbei ist Conway's Law zu berücksichtigen. Services sollten demnach als lang lebende Produkte und nicht als zeitlich befristete Projekte verstanden wissen (You build it, you run it).

Noch mehr als die Programmierung einzelner Services ist die Entwicklung einer tragfähigen Architektur eine Kunst und beruht zu einem großen Teil auf Erfahrung. Im Zusammenhang mit Microservice-Architekturen wird aber immer wieder eine Methodik erwähnt, die (mit einer höheren Wahrscheinlichkeit) Architekturen erzeugt, die gut auf die in Kapitel 12 genannten Erfordernisse abgestimmt sind.

Dieses Kapitel widmet sich daher der Methodik des Domain-driven Design (DDD), die sehr gut mit Microservices und den genannten Erfordernissen harmoniert. DDD ist allerdings weder auf Cloud-native Systeme oder Microservice-Architekturen festgelegt und kann auch in anderen Kontexten der Entwicklung tragfähiger Software-Architekturen eingesetzt werden. Domain-driven Design (DDD) ist eine Methodik zur Modellierung komplexer Software. Der Begriff „Domain-driven Design“ wurde 2003 von Eric Evans in seinem gleichnamigen Buch (Evans 2003) geprägt – also deutlich bevor es den Begriff Cloud-native und entsprechende Systeme überhaupt gab.

Domain-driven Design basiert auf folgenden zwei Annahmen:

- Der Schwerpunkt des Softwaredesigns liegt auf der Fachlichkeit und der Fachlogik.
- Der Entwurf komplexer fachlicher Zusammenhänge sollte auf einem Modell der Anwendungsdomäne, dem Domänenmodell basieren.

DDD orientiert sich an agiler Softwareentwicklung, ist aber SW-entwicklungsprozess-agnostisch. Eine iterative Softwareentwicklung und eine enge Zusammenarbeit zwischen Entwicklern und Domänen-Experten sind jedoch erforderlich.

Domain-driven Design (DDD) strebt dabei nicht nach einem perfekten, sondern nach einem effektiven Design. Effektives Design erfüllt die Anforderungen einer Firma, einer Organisation oder einer sonstigen Institution bis zu einem Grad, der notwendig ist, damit man sich mittels digitalisierter Prozesse von Mitbewerbern oder in derselben Domäne tätigen Institutionen positiv abheben kann. Effektives Design zwingt Organisationen dazu zu erkennen, welche Bereiche man fokussieren sollte. Nur in diesen Bereichen wird effektives Design eingesetzt, um ein domänenkonformes Softwaremodell zu entwickeln und fortzuschreiben. Andere nicht domänenspezifische Bereiche versucht man durch Standardsoftware oder mittels externer Services abzudecken.

Die folgenden Darstellungen orientieren sich dabei insbesondere an (Vernon 2017) und (Khononov 2019). Weil DDD eine besonders beliebte Methodik ist, insbesondere micro-service-basierte Architekturen zu entwickeln, sei an dieser Stelle dem Leser dennoch der Hinweis gestattet, dass es sich bei DDD um eine Methodik und nicht mehr handelt. Keinesfalls ein Gesetz! *Methodiken* helfen einem dabei, komplexe Probleme zu bewältigen, indem sie bewährte Handlungsweisen systematisieren. Man kann jedoch auch ohne Methodiken dieselben Probleme bewältigen. Insbesondere die ersten „Problemlöser“ haben selten geeignete Methodiken an der Hand und müssen diese erst mühsam selber erarbeiten.

Es lassen sich also durchaus gute Cloud-native Anwendungen ohne Kenntnis von DDD entwickeln. Der Leser kann natürlich auch nur einzelne DDD-Bausteine nutzen und diese mit anderen Methodiken und Vorgehensweisen kombinieren. Im Weiteren werden hierzu die DDD ausmachenden Bausteine näher betrachtet:

- Fokus auf Fachlichkeit (siehe Abschnitt 14.1)
- Strategisches Design (siehe Abschnitt 14.2)
- Taktisches Design (siehe Abschnitt 14.3)

Der Leser kann diese Bausteine auch gerne hinsichtlich seines individuellen Mehrwerts für einen ganz spezifischen Kontext oder spezifisches Problem bewerten. Oft ist eine erste vertiefte Auseinandersetzung mit dem Thema DDD nämlich in solch besonders drängenden Teilbereichen besonders zielführend.

■ 14.1 Fachlichkeit

Die DDD-Philosophie legt einen besonderen Fokus auf die Fachlichkeit – und beschränkt dies nicht nur auf die Anforderung erhebenden Schritte von Entwicklungsmethodiken. Nach dieser Philosophie dient Software zur Umsetzung von Geschäftsanforderungen (und nicht umgekehrt). Begründet wird dies damit, dass Fachlichkeit meist langlebiger als kurzfristige „IT-Moden“ ist. Ein Fokus auf Fachlichkeit bringt somit Stabilität in Softwarearchitekturen und macht damit auch komplexe Systeme beherrschbarer. Diese Fachlichkeit ist bei DDD der gemeinsame Nenner aller am Entwicklungsprozess Beteiligten. Somit begleitet fachliches Domänenwissen alle Stufen der Softwareentwicklung und nicht nur während der Anforderungserhebung und der Test- und Nachweisführung. Softwareentwicklung startet und endet somit immer mit dem Wissen von Domänenexperten.

Bei DDD geht es im Kern somit um

- eine gemeinsame von Fachlichkeit geprägte Sprache (bzw. gemeinsames Verständnis), die von allen Beteiligten verstanden wird,
- die Optimierung der Zusammenarbeit entlang einer durchgehenden Fachlichkeit
- mittels Methoden und Verfahren,
- die die Änderung von Denkmustern bei allen Beteiligten fördert.

DDD unterscheidet hierzu im Wesentlichen die zwei Ebenen des strategischen (siehe Abschnitt 14.2) und des taktischen Designs (siehe Abschnitt 14.3), die dabei helfen, sich einen Problemraum zu erschließen und daraus eine passende Lösung innerhalb eines Lösungsraums abzuleiten (siehe Bild 14.1).

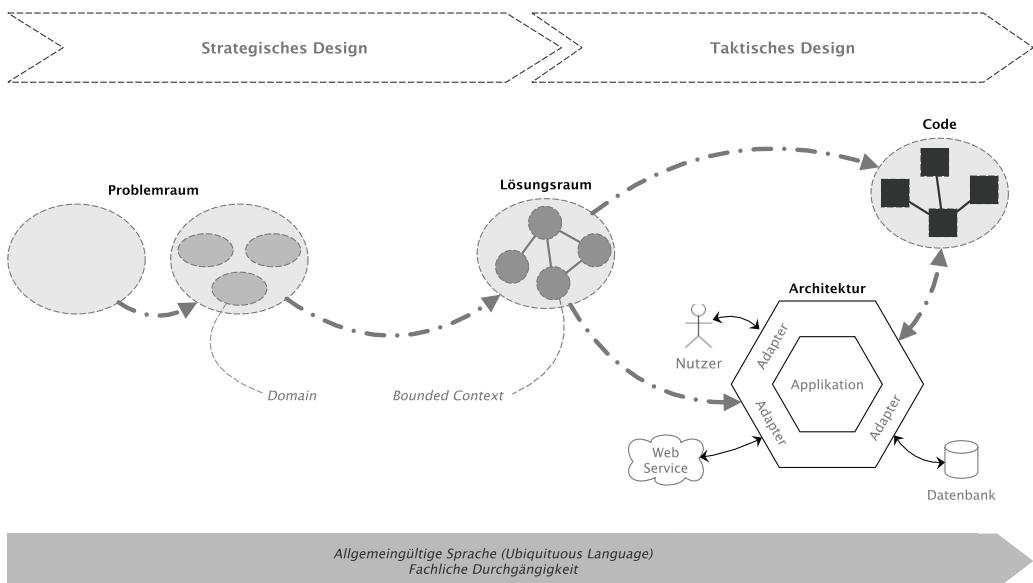


Bild 14.1 Vom Problemraum zur Lösung mittels DDD

Mittels des **strategischen Designs** analysiert man dabei, was wichtig für die Zielerreichung (das Geschäft) ist. Hierbei versucht man die Fachexpertise von Domain Matter Experts mittels einer gemeinsamen Sprache (einer sogenannten **Ubiquitous Language**) einzubeziehen. Der Problemraum wird dabei in einem ersten Schritt mittels **Subdomains** gegliedert und diese hinsichtlich ihrer Art in Kern-, Unterstützungs- und generische Subdomänen klassifiziert. Innerhalb von Kern- und Teilen von Unterstützungssubdomänen werden feingranularere **Bounded Contexts** definiert und mittels **Context Mapping** zueinander in Beziehung gesetzt. Diese Bounded Contexts bilden den Übergang vom strategischen zum taktischen Design und dienen der Ausarbeitung des Domänenmodells in den besonders wertschöpfenden Bereichen. Auf die detaillierte Ausarbeitung anderer Bereiche wird verzichtet.

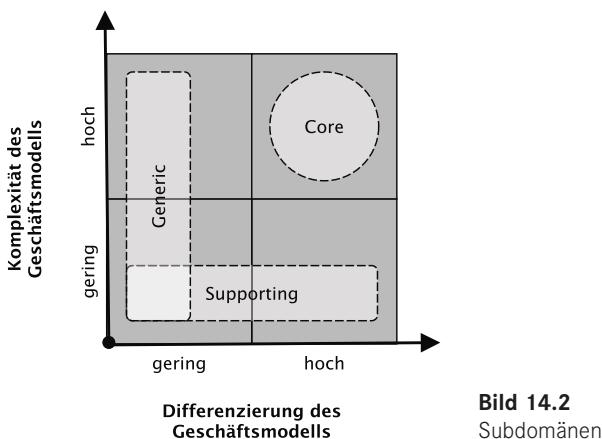
Oft ergeben sich im Rahmen der Softwareentwicklung **Microservices** aus einem Bounded Context. Innerhalb eines Bounded Context wird dabei eine gemeinsame Sprache (eine Ubiquitous Language) definiert und für den weiteren Softwareentwicklungsprozess inner-

halb eines Microservices verwendet. Die Arbeiten im Inneren eines Bounded Contexts umfassen u. a. die Definition von Entitys und Value-Objekten, die in sogenannten Aggregaten in **Domänenmodellen** zusammengefasst werden. Service-Interaktionen über die Grenzen von Bounded Contexts werden mittels **Domain-Events** modelliert, um Schnittstellen zwischen Services definieren zu können.

DDD beschreibt somit den in Bild 14.1 gezeigten Prozess und unterstützende Verfahren und Techniken, die einer Philosophie der Ausrichtung der Softwareentwicklung an der Fachlichkeit folgen. Durch diese methodische Herangehensweise korrespondieren Fachlichkeit und Software auch für Domänenexperten nachvollziehbar. Hierdurch kann jederzeit beantwortet werden, in welcher Softwarekomponente welche Fachlichkeit umgesetzt wird. Derartige entwickelte Softwarelösungen skalieren mit der Systemgröße. Die Integrationskomplexität einer fachlichen Änderung wächst somit nicht mit der Größe eines Systems. Somit bleiben auch wachsende und sich evolutionär entwickelnde Systeme wartbar. Die Zusammenarbeit von Fachexperten und Softwareentwicklung ist dabei von einer technologiefreien Kommunikation geprägt und vereint unterschiedliche Fähigkeiten von Fachexperten und Entwicklung.

■ 14.2 Strategisches Design

Unter einer Domäne versteht man im Allgemeinen einen bestimmten Tätigkeits- oder Wissensbereich. Das Domänenkonzept ist sehr breit und abstrakt. Um es konkreter und greifbarer zu machen, wird es bei DDD in kleinere Teile aufgeteilt, die Subdomänen genannt und zumeist in drei Kategorien unterteilt werden (siehe Bild 14.2).



Alle Subdomänen, unabhängig von der Kategorie, sind wichtig für Gesamtlösungen. Fehlt eine Domäne, wird die Lösung unvollständig sein. Die genannten Domänen erfordern jedoch einen unterschiedlichen Aufwand und können auch unterschiedliche Anforderungen an Qualität und Vollständigkeit haben.

Ein Geschäftsfeld (Business Domain) definiert den gesamten Tätigkeitsbereich einer Organisation oder eines Unternehmens. Im Allgemeinen ist es die Dienstleistung, die Einrichtungen Kunden anbieten. Ein Beispiel:

- DHL => Paketdienst
- Starbucks => Kaffee
- Karstadt => Einzelhandel

Ein Unternehmen kann dabei in mehreren Geschäftsbereichen tätig sein. Zum Beispiel bietet Amazon sowohl Einzelhandels- als auch Cloud-Computing-Dienste an. Unternehmen wandeln gegebenenfalls ihre Geschäftsbereiche auch im Verlaufe der Zeit. Nokia war beispielsweise im Laufe seiner Geschichte in so unterschiedlichen Bereichen wie Holzverarbeitung, Gummiherstellung, Telekommunikation und Mobiltelefonen tätig.

14.2.1 Subdomänen

Um die Ziele seiner Geschäftsdomäne zu erreichen, muss ein Unternehmen in mehreren Subdomänen operieren. Eine Subdomäne ist ein feingranularer Bereich der Geschäftstaktivität. Alle Subdomänen zusammen ergeben die Geschäftsdomäne des Unternehmens. Oft korrelieren Subdomänen mit den Abteilungen oder anderen Organisationseinheiten des Unternehmens. Zum Beispiel könnte ein Online-Handelsshop Subdomänen wie Katalogmanagement, Werbung, Buchhaltung, Support, Kundenbeziehungen, Lieferantenbeziehungen und andere enthalten.

Oft werden die folgenden Subdomänen unterschieden:

- Kerndomäne (Core Subdomain)
- Unterstützungsdomäne (Supporting Subdomain)
- Generische Subdomäne (Generic Subdomain)

14.2.1.1 Kerndomäne (Core Subdomain)

Eine Kerndomäne unterscheidet ein Unternehmen von seinen Konurrenten. Dabei kann es sich um die Erfindung neuer Produkte oder Dienstleistungen handeln oder um die Optimierung bestehender Prozesse und damit um die Reduzierung von Kosten. Eine einfach zu implementierende Kerndomäne kann nur einen kurzebigen Wettbewerbsvorteil bieten. Daher sind Kern-Subdomänen naturgemäß komplex. Üblicherweise gibt es hohe Eintrittsbarrieren, die es Wettbewerbern schwermachen, Lösungen des Unternehmens gleichwertig zu kopieren oder zu imitieren.

Üblicherweise investiert man im DDD System- und Softwareentwicklungsaufwände primär in die Kerndomäne. Kern-Subdomains müssen meist intern implementiert werden. Sie können nicht gekauft oder übernommen werden; andernfalls wären Wettbewerber einfach in der Lage, dasselbe zu tun. Es wäre auch unklug, die Implementierung einer Kern-Subdomäne auszulagern.

14.2.1.2 Unterstützende Subdomäne (Supporting Subdomain)

Im Gegensatz zu den Kern-Subdomains bieten unterstützende Subdomains keinen Wettbewerbsvorteil für Unternehmen. Sie unterstützen lediglich das Kerngeschäft. Zu den Kern-Subdomänen eines Online-Werbeunternehmens gehören beispielsweise die Anpassung der Anzeigen an die Besucher, die Optimierung der Wirksamkeit der Anzeigen und die Minimierung der Kosten für die Werbeplätze.

Hierfür muss das Unternehmen u. a. seine kreativen Materialien katalogisieren. Die Art und Weise, wie das Unternehmen seine Kreativmaterialien verwaltet, hat jedoch keinen nennenswerten Einfluss auf seine Gewinne. In diesem Bereich gibt es kaum Potenziale, um sich am Markt abzuheben. Kunden erwarten einfach, dass solche Dinge laufen.

Der auffälligste Unterschied zwischen Kern- und unterstützenden Subdomänen ist die Komplexität der Geschäftslogik. Unterstützende Subdomänen sind meist einfach. Ihre Geschäftslogik basiert häufig auf einfachen ETL-Operationen (siehe Abschnitt 14.3.1.1) und sogenannten CRUD-Schnittstellen (siehe Abschnitt 14.3.1.2). Im Gegensatz zu den Kern-Subdomänen ändern sich die unterstützenden Subdomänen auch nicht oft. Sie bieten keinen Wettbewerbsvorteil für das Unternehmen, und daher gibt es weniger geschäftlichen Wert, sie im Laufe der Zeit weiterzuentwickeln.

Mangels Wettbewerbsvorteils ist es somit sinnvoll, unterstützende Subdomänen – wenn möglich – nicht selbst zu implementieren. Anders als bei generischen Subdomänen gibt es jedoch häufig keine vorgefertigten Lösungen, sodass auch unterstützende Subdomänen oft selbst implementiert werden müssen. In diesem Bereich sind jedoch oft einfache Anwendungsentwicklungs-Frameworks ausreichend. Diese Einfachheit der Geschäftslogik macht unterstützende Subdomänen zu einem naheliegenden Kandidaten für das Outsourcing.

14.2.1.3 Generische Subdomänen (Generic Subdomain)

Auch generische Subdomains sind in der Regel komplex und schwer zu implementieren. Allerdings bieten generische Subdomains keinen Wettbewerbsvorteil für das Unternehmen. Hier gibt es bereits bewährte und weitverbreitete Implementierungen, die von vielen Unternehmen genutzt werden. Generische Subdomänen sind somit Geschäftsaktivitäten, die alle Unternehmen auf sehr ähnliche Weise durchführen. Üblicherweise versucht man für generische Subdomänen Standardsoftware und -systeme zu verwenden. Ein typisches Beispiel wäre die Identitätsverwaltung mittels Microsofts Active Directory. Oft finden sich hier auch technische Standards wie LDAP.

14.2.1.4 Anmerkungen am Beispiel einer Fallstudie

Ein und dieselbe Subdomäne kann durchaus in verschiedene Kategorien fallen, je nachdem, in welcher Geschäftsdomäne ein Unternehmen oder eine Organisation tätig ist. Für ein Unternehmen, das sich beispielsweise auf Identitätsmanagement spezialisiert hat, ist Identitätsmanagement durchaus eine Kerndomäne (z. B. die Microsoft-Abteilung, die das Produkt Active Directory verantwortet). Für ein Unternehmen, das sich hingegen auf Customer-Relationship-Management spezialisiert hat, ist das Identitätsmanagement jedoch ein allgemeiner (technologisch austauschbarer) Teilbereich.

Um eine Kategorisierung von Subdomänen für ein Unternehmen oder eine Organisation vorzunehmen, sollte man daher aus einer unternehmensspezifischen Sicht die in Tabelle 14.1

aufgeführten Kriterien ermitteln, um zu beurteilen, ob eine Domäne eine Kern-, unterstützende oder generische Subdomäne ist. Diese Einschätzung ist natürlich sehr individuell. Der Datenbankanbieter Oracle wird eine Subdomäne Datenhaltung hinsichtlich seiner Marktbedeutung vermutlich anders einschätzen als der Lieferdienst Lieferando.

Tabelle 14.1 Kategorien von Subdomänen im Domain-driven Design

| | Core | Supporting | Generic |
|---|------|------------|-------------|
| Alleinstellungsmerkmal/Geschäftskritisch | ja | nein | nein |
| Unternehmensspezifisch | ja | ja | nein |
| Technische Komplexität (Einstiegshürde) | hoch | niedrig | mittel/hoch |
| Technischer Wandel | hoch | niedrig | niedrig |
| Existieren Lösungen/Produkte/Services? | nein | teilweise | ja |
| In-House-Entwicklung sinnvoll | ja | möglich | nein |
| Out-Sourcing der Entwicklung ratsam | nein | möglich | nein |
| Existieren COTS-Produkte/Managed Services | nein | teilweise | ja |

Diese Zusammenhänge sollen noch einmal am Beispiel eines EMR-Systems (Electronic Medical Records) für kleinere Kliniken veranschaulicht werden (siehe Bild 14.3), welches (Khononov 2019) entlehnt ist. Für diesen Fall einer elektronischen Patientenakte seien die folgenden Subdomänen identifiziert worden:

- **Patientenakten** für die Verwaltung der Patientendaten und -akten (persönliche Informationen, Anamnese etc.).
- **Labor** für die Bestellung von Labortests und die Verwaltung der Testergebnisse.
- **Terminplanung** für die Planung von Terminen.
- **Dateiablage** für die Speicherung und Verwaltung von Dateien, die den Patientenakten beigefügt sind (z. B. verschiedene Dokumente, Röntgenbilder, gescannte Papierdokumente).
- **Identitätsmanagement**, um sicherzustellen, dass die richtigen Personen Zugriff auf die richtigen Informationen haben.

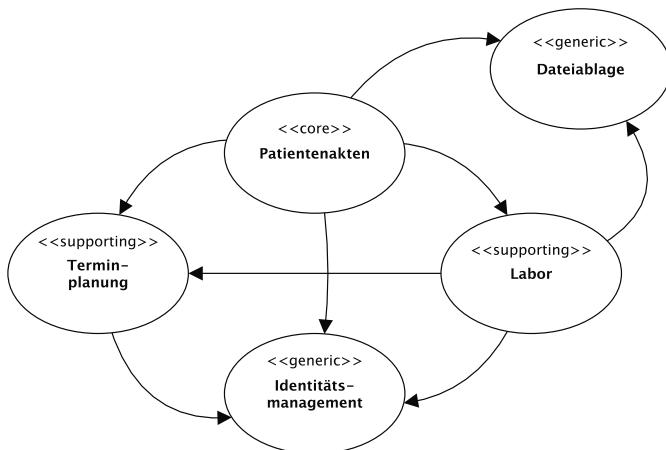


Bild 14.3
Fallstudie einer elektronischen Patientenakte

Wie würde man nun vermutlich diese Subdomänen klassifizieren? Die offensichtlichsten Domänen sind die Dateiablage und das Identitätsmanagement. Beides sind eindeutig generische Subdomänen. Aber was ist mit den anderen? Das hängt u. a. davon ab, was dieses bestimmte EMR-System von anderen Systemen auf dem Markt abheben soll.

Bei einem EMR-System kann man ziemlich sicher davon ausgehen, dass *Patientenakten* eine Kerndomäne ist. Wenn allerdings ein EMR-System ebenfalls zum Ziel hat, Kliniken durch eine innovative Terminplanung effizienter zu machen, dann wäre *Terminplanung* gegebenenfalls auch als Kerndomäne zu werten. Andernfalls ist es eine unterstützende Subdomäne, die durch eine bestehende Terminplanungs-Engine gut abdeckbar wäre. Allerdings wird man sich hier dann kaum von Wettbewerbern abheben können. Ähnlich liegt der Fall bei der Subdomäne *Labor*: Sollte ein wesentlicher Teil des Business Case eine nahtlose Integration zwischen Patientenakten und Labor sein, dann ist das Labor höchstwahrscheinlich eine Kerndomäne. Andernfalls ist es eine unterstützende Subdomäne.

14.2.2 Ubiquitous Language

Manchmal wird eine Domäne auch als Problemraum bezeichnet. Das kommt daher, dass eine Domäne fachliche Probleme definiert, die eine Software lösen soll. (Vernon 2017) unterscheidet daher einen Problemraum und einen Lösungsraum. Der Problemraum konzentriert sich auf die fachlichen Probleme, die wir zu lösen versuchen. Die genannten Subdomänen gehören in diesen Raum.

Der Lösungsraum konzentriert sich darauf, wie die Probleme technisch gelöst werden sollen. Ein Lösungsraum ist konkreter, technischer und enthält mehr Details. Die Schwierigkeit ist, wie man die Beziehungen zwischen dem fachlichen Problemraum und den softwaretechnischen Lösungsraum erhält.

Übliche Software-Entwicklungsmethodiken basieren meist implizit auf den folgenden Übersetzungen:

- Domänenwissen → Analysemodell
- Analysemodell → Anforderungen
- Anforderungen → Systementwurf
- Systementwurf → Quellcode

Bei jedem dieser Übergänge können Übersetzungsfehler passieren. In vielen Softwareentwicklungsmodellen gibt es somit Tätigkeiten, in denen Domänenwissen in eine ingenieursfreundliche Form „übersetzt“ wird. Diese Artefakte werden meist als Analysemodell bezeichnet und umfassen eine Beschreibung der Anforderungen an das System und nicht ein Verständnis der dahinterliegenden Geschäftsdomäne.

Für den Wissensaustausch von Fachdomäne zur Entwicklung ist dies jedoch nach (Vernon 2017) gefährlich. Bei jeder Übersetzung gehen Informationen verloren; im Falle der Softwareentwicklung geht Domänenwissen, das für die Lösung von Geschäftsproblemen unerlässlich ist, auf dem Weg zu den Softwareingenieuren verloren.

Seit der Softwarekrise in den 1960ern haben viele Untersuchungen das Scheitern von Softwareprojekten untersucht. Fast alle haben gezeigt, dass u. a. Kommunikation für den Projekterfolg unerlässlich ist. Fast alle Softwareentwicklungsmodelle versuchen daher,

Domänenwissen von Domänenexperten an die Ingenieure weiterzugeben. Meist geschieht dies durch Vermittlerrollen, die Wissen von der Fachdomäne an die Entwicklung „durchreichen“ und übersetzen: System-/Business-Analysten, Product Owner, Projektmanager und mehr. Das impliziert aber nicht automatisch eine effektive, d. h. auf Verständnis beruhende, Kommunikation.

DDD versucht vor diesem Hintergrund, das Wissen von Domänenexperten zu Softwareingenieuren mittels einer allgegenwärtigen – und gemeinsam entwickelten – Sprache zu bringen, der sogenannten Ubiquitous Language.

14.2.2.1 Eine gemeinsame Sprache als Schlüssel zu einem gemeinsamen Verständnis

Der Begriff „ubiquitär“ bedeutet allumfassend, überall vorhanden, allgegenwärtig. DDD verwendet die Begrifflichkeit Ubiquitous Language im Sinne einer im Projekt allgegenwärtigen (und von allen gleichermaßen) verstandenen Sprache. Alle relevanten Konzepte müssen in dieser Sprache beschrieben werden können. Die Sprache wird evolutionär weiterentwickelt und ist in allen Phasen der Entwicklung präsent, auch im Code, z. B. in Form von Paket-, Klassen-, Variablen- oder Parameterbezeichnungen.

Die Ubiquitous Language ist eines der wichtigsten Konzepte in DDD, da sie die Grundlage für gemeinsames und durchgängiges Verständnis bildet und somit einen guten Indikator für Durchdringung und Beschreibungsstabilität einer Domäne darstellt. Das Problem dabei ist, dass es meist aufwendig ist, die Klärung aller fachlichen Begriffe vorzunehmen. Unklare Begriffe deuten jedoch immer auf tieferliegende verdeckte (gegebenenfalls unerkannt unverstandene) Konzepte hin. Der Sinn einer Ubiquitous Language liegt also vor allem darin, implizite Annahmen explizit zu machen, um eine eindeutige Bedeutung innerhalb der Ubiquitous Language festlegen zu können.

Ziel ist es dabei nicht, Domänenexperten etwas über Softwareentwicklung (z. B. Singletons und abstrakte Fabriken) beizubringen. Der Zweck der allgegenwärtigen Sprache ist es, das Verständnis und die mentalen Modelle der Geschäftsdomäne der Domänenexperten in leicht verständliche Begriffe für die Entwicklung zu fassen, die in allen Phasen der Softwareentwicklung Anwendung finden soll (siehe Bild 14.4).

Formulierungen mittels Begrifflichkeiten der Ubiquitous Language sollten also eher in dieser Form

„Eine Werbekampagne kann verschiedene kreative Materialien anzeigen.“

als in dieser Form

„Der Anzeigen-iframe zeigt eine HTML-Datei an.“

formuliert werden. Denn viele Domänenexperten mögen vielleicht noch den Begriff *HTML* kennen, ob er ihnen jedoch etwas fachlich im Problemraum sagt, ist oft fraglich. Spätestens bei der Begrifflichkeit *iframe* wird man viele Fachexperten verloren haben.

Doch selbst zwischen Fachexperten gibt es die üblichen Probleme von mehrdeutigen und synonymen Begriffen, die eine Ubiquitous Language durchaus rechtfertigen können.

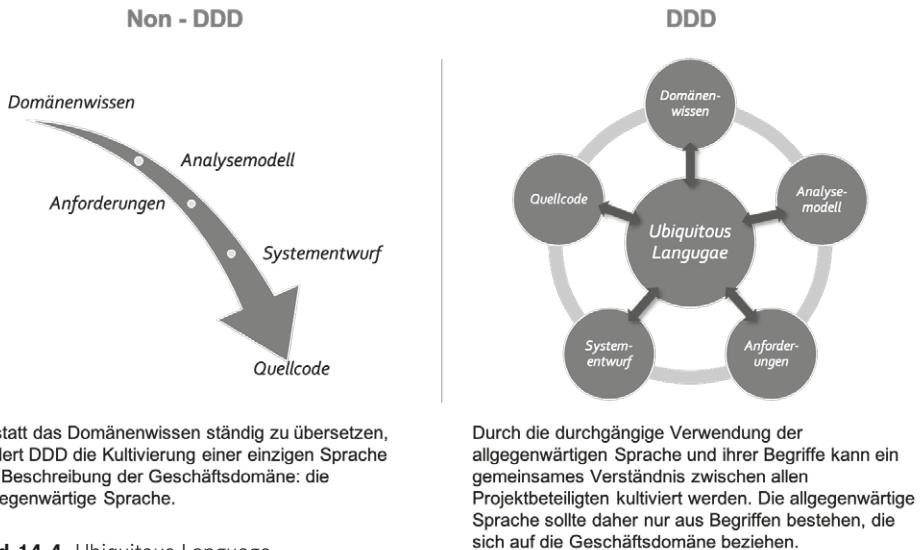


Bild 14.4 Ubiquitous Language

14.2.2.2 Mehrdeutige und synonome Begriffe

Lassen Sie uns dies an einem Begriff verdeutlichen, der bewusst nicht der deutschen Sprache entnommen wurde, um auch etwas mehr „sprachlichen Abstand“ zum Problem gewinnen zu können.

Der englische Begriff „Policy“ hat z. B. mehrere Bedeutungen: Er kann eine gesetzliche Vorschrift oder einen Versicherungsvertrag bedeuten. Die genaue Bedeutung ergibt sich meist aus dem Kontext. Problematisch sind Kontexte, in denen etwa Versicherungsverträge aufgrund gesetzlicher Vorschriften erforderlich sind. Welches Konzept meint Policy nun? Bei Versicherungsgesellschaften haben wir eine Domäne, in der beide Konzepte zeitgleich vorkommen.

Ubiquitäre Sprache verlangt für solche Fälle eine einzige Bedeutung für jeden Begriff, und daher sollte Policy explizit mit den beiden Begriffen „Regulierungsvorschrift“ oder „Versicherungsvertrag“ modelliert werden, aber nie Policy genannt werden.

In DDD können ferner zwei Begriffe nicht austauschbar verwendet werden. Zum Beispiel verwenden viele Systeme den Begriff „Benutzer“. Der Begriff „Benutzer“ wird im normalen Sprachgebrauch jedoch häufig austauschbar verwendet: z. B. „Benutzer“, „Besucher“, „Administrator“ oder „Konto“.

Synonome Begriffe bezeichnen meist unterschiedliche Feinkonzepte. So beziehen sich die Begriffe „Besucher“ und „Konto“ technisch gesehen auf die Benutzer des Systems; in den meisten Systemen stellen jedoch nicht registrierte und registrierte Benutzer unterschiedliche Rollen dar und haben unterschiedliche Verhaltensweisen. Daten von „Besuchern“ werden meist zu Analysezwecken verwendet, während „Konten“ das System und seine Funktionen tatsächlich nutzen.

Es ist vorzuziehen, jeden Begriff explizit und in seinem spezifischen Kontext zu verwenden. Ein Verständnis der Unterschiede zwischen den verwendeten Begriffen ermöglicht die Erstellung einfacherer und klarerer Modelle und Implementierungen der Entitäten der Geschäftsdomäne.

14.2.3 Bounded Contexts

Wie wir gesehen haben, ist es für den Erfolg eines Projekts wichtig, eine Ubiquitous Language zu entwickeln, die klar und konsistent sein und die mentalen Modelle der Domänenexperten widerspiegeln muss. Es ist aber nicht ungewöhnlich, dass mentale Modelle von Domänenexperten selbst nur in einem spezifischen Kontext konsistent sind. Erweitert man den Kontext, entstehen Inkonsistenzen mit anderen Kontexten.

Insbesondere Enterprise-Architecture-Management-Ansätze haben oft den Anspruch, eine globale Ubiquitous Language für ein ganzes Unternehmen zu entwickeln. Häufig scheitern diese Ansätze jedoch. Warum dies so ist, soll anhand einer Fallstudie zu Marketing- und Vertriebskontexten deutlich werden.

Angenommen, Sie arbeiten für ein Telemarketing-Unternehmen (siehe Bild 14.5). Die Marketingabteilung des Unternehmens generiert sogenannte „Leads“ durch Online-Anzeigen. Die Vertriebsabteilung ist dafür zuständig, potenzielle Kunden zum Kauf der Produkte oder Dienstleistungen zu bewegen. Beide Abteilungen haben jedoch ein unterschiedliches Verständnis eines Leads.

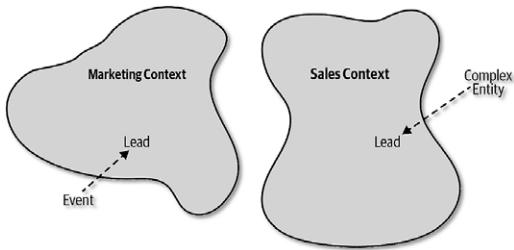


Bild 14.5 Fallstudie Telemarketing

- **Marketing-Abteilung:** Im Marketing stellt ein Lead eine Benachrichtigung dar, dass jemand an einem der Produkte interessiert ist. Das Ereignis des Erhalts der Kontaktdata des potenziellen Kunden wird als Lead bezeichnet. Ein Kunde könnte z. B. seine E-Mail-Adresse hinterlassen haben, weil er sich aufgrund einer Online-Marketing-Aktion für ein Auto interessiert.
- **Vertriebsabteilung:** Im Vertrieb repräsentiert ein Lead den gesamten Lebenszyklus eines Verkaufsprozesses. Er ist kein bloßes Ereignis, sondern ein lang andauernder Prozess. Zum Beispiel könnte der Kunde eine Probefahrt in einem Autohaus vereinbart haben, lässt sich dann hinsichtlich Finanzierungsmöglichkeiten informieren, konfiguriert an einem weiteren Termin schließlich Details eines Neuwagens, klärt dann irgendwann Zulassungsdaten und holt es nach Lieferung im Autohaus ab und vereinbart dann auch irgendwann seine erste Inspektion.

Die Lösung zum Umgang mit solchen Problemen ist letztlich trivial. Die DDD-Methodik hat gar nicht zum Ziel, eine einzige allumfassende Ubiquitous Language zu entwickeln, sondern mehrere kleinere. Man teilt hierzu die allgegenwärtige Sprache in mehrere kleinere Sprachen auf und ordnet dann jede Sprache dem expliziten Kontext zu, in dem sie angewendet werden kann – ihrem begrenzten Kontext (Bounded Context).

In unserem Beispiel können wir zwei begrenzte Kontexte identifizieren: Marketing und Sales. Der Begriff „Lead“ existiert in beiden Bounded Contexts (siehe Bild 14.6). Solange ein Lead in jedem begrenzten Kontext nur eine einzige Bedeutung hat, bleiben die Ubiquitous Languages zu Marketing und Vertrieb konsistent und folgt den mentalen Modellen der Domänenexperten. Würde man nicht so vorgehen, wäre es erforderlich, ein einziges riesiges Modell zu konzipieren und abzustimmen.

**Bild 14.6**

Begriffsdefinition nur innerhalb von
Bounded Contexts (Quelle: Khononov)

Die Größe eines Bounded Contexts ist für sich genommen kein entscheidender Faktor. Modelle sollten nie per se groß oder klein sein. Modelle müssen klar sein und einem Zweck folgen. Je umfassender die Grenze einer Ubiquitous Language ist, desto schwieriger wird es, sie konsistent zu halten. Daher hängt die Entscheidung, wie groß ein Bounded Context sein soll, von der spezifischen Problemdomäne ab.

- Es kann von Vorteil für die Modellierung sein, eine große Ubiquitous Language in kleinere, besser handhabbare Problemdomänen aufzuteilen.
- Das Streben nach kleinen Bounded Contexts wird aber auch mehr Integrationsaufwände erzeugen.

Letztlich nutzt man Bounded Contexts bei der evolutionären Entwicklung von Cloud-nativen Anwendungen dazu, pro Bounded Context einen Microservice einer Cloud-nativen Anwendung abzuleiten. Je kleiner die Kontexte also werden, desto mehr zu integrierende Microservices wird man erhalten. Man muss also einen guten Mittelwert zwischen möglichst unaufwendig definierbaren Ubiquitous Languages (=> viele kleine Kontexte) und möglichst wenigen zu integrierenden Softwarekomponenten (=> wenige große Kontexte) finden.

Abschnitt 14.2.1 hat gezeigt, dass eine Geschäftsdomäne aus mehreren Subdomains besteht. Abschnitt 14.2.3 befasst sich nun mit der Zerlegung einer Geschäftsdomäne in eine Reihe von feingranulareren Problemdomänen (Bounded Contexts). Beide Methoden erscheinen irgendwie redundant. Doch ist das so?

Um das Geschäft eines Unternehmens zu verstehen, muss seine Geschäftsdomäne analysiert werden. Gemäß DDD beinhaltet die Analysephase die Identifizierung der verschiedenen Arten von Subdomains (Core, Supporting, Generic). Identifikation ist hier das Schlüsselwort. Die Subdomänen sind bereits vorhanden, sie sind ein Teil des Geschäfts. **Durch Analyse entdeckt man Subdomänen, die bereits vorhanden sind. Subdomänen werden nicht entwickelt!** Diese Subdomains helfen dabei, Schwerpunkte zu setzen und begrenzte Mittel primär in den Core-Subdomains einzusetzen, z. B. zur Entwicklung einer Ubiquitous Language in einem Bounded Context.

Bounded Contexts hingegen werden entworfen. Die Wahl der Grenzen von Modellen ist eine strategische Designscheidung. Mittels Bounded Contexts wird entschieden, wie die Geschäftsdomäne in kleinere, überschaubare Problemdomänen unterteilt wird.

14.2.4 Context Mapping

Modelle in verschiedenen Bounded Contexts können unabhängig voneinander entwickelt und implementiert werden. Bounded Contexts sind aber nicht isoliert, da ein System nicht aus vollkommen unabhängigen Komponenten aufgebaut werden kann. Die Komponenten müssen miteinander interagieren, um die übergreifenden Systemziele zu erreichen. Dies gilt natürlich auch für Bounded Contexts. Obwohl sich ihre Implementierungen unabhängig voneinander entwickeln können sollen, müssen sie miteinander integriert werden. Infolgedessen wird es immer Berührungspunkte in Form von Schnittstellen geben, die als Verträge bezeichnet werden.

Zwei Bounded Contexts verwenden per Definition unterschiedliche Ubiquitous Languages. Welche Sprache soll für die Integration also verwendet werden? Hierfür sieht DDD Vertrags-Pattern zur Handhabung von Beziehungen und Integrationen zwischen Bounded Contexts vor. Diese Pattern werden meist in drei Gruppen unterteilt, die jeweils eine Art der Teamzusammenarbeit repräsentieren:

- Partnerschaftliche Kooperation
- Customer-Supplier
- Separate Ways

All diese Pattern machen letztlich die Anwesenheit oder Abwesenheit von Machtgefällen zwischen unabhängig voneinander agierenden Teams explizit und berücksichtigen dabei auch die Besonderheiten von „Conway’s Law“ (siehe Kapitel 12). Conway’s Law besagte ja, dass nicht nur fachliche und technische Anforderungen, sondern auch Organisationsstrukturen (also z. B. Abteilungsgrenzen, Standorte usw.) in Unternehmen beeinflussen, wie softwaretechnische Lösungen aussehen werden. Es kann insbesondere im Betrieb zu Problemen führen, diesen für Softwareingenieure oft als „lästig“ empfundenen – aber eben existenten – Aspekt zu ignorieren. Vielmehr sollte er bei der evolutionären Entwicklung von Architekturen frühzeitig bedacht und explizit gemacht werden.

14.2.4.1 Partnerschaftliche Kooperationsmuster (Partners und Shared-Kernel)

Kooperationsmuster beziehen sich auf Bounded Contexts, die von Teams mit gut etablierter Kommunikation implementiert werden. Diese Anforderung ist automatisch für Bounded Contexts erfüllt, die von demselben Team implementiert werden. Diese Pattern sind aber auch für Teams mit abhängigen Zielen geeignet, bei denen der Erfolg eines Teams von dem eines anderen Teams abhängt und umgekehrt. Auch hier ist das Hauptkriterium die Qualität der Kommunikation und Zusammenarbeit der Teams.

Im **Partnerschaftsmodell** wird die Integration zwischen Bounded Context ad hoc – also anlassbezogen – koordiniert. Ein Team kann ein zweites über eine Änderung in der API benachrichtigen, und das zweite Team wird kooperieren und erforderliche Anpassung unmittelbar vornehmen. Die Koordination der Integration erfolgt hier in beide Richtungen. Kein Team diktiert die Sprache, die für die Definition der Verträge verwendet wird. Die Teams können Unterschiede gemeinsam bewerten und die am besten geeignete Lösung wählen. Außerdem kooperieren beide Seiten bei der Lösung von Integrationsproblemen. Keines der Teams ist daran interessiert, das andere zu blockieren.

Eine derartige Kooperation wird wie in Bild 14.7, Teilbild A.1, dargestellt. Gut eingespielte Praktiken der Zusammenarbeit, ein hohes Maß an Engagement und häufige Synchronisationen zwischen den Teams sind für dieses Integrationsmuster erforderlich. Dieses Muster ist meist nicht für geografisch verteilte Teams geeignet, da es zu Synchronisations- und Kommunikationsproblemen vor allem in sich stark unterscheidenden Zeitzonen führen kann.

Das Modell des **gemeinsamen Kernels (Shared-Kernel)** ist eine formalere Art, einen Vertrag zwischen mehreren Bounded Contexts ohne Machtgefälle zu definieren. Anstelle von Ad-hoc-Integrationen wird hier der Vertrag explizit in einer komplizierten Bibliothek – dem gemeinsamen Kernel – definiert. Gemeinsam genutzte und entwickelte Librarys definieren somit die Integrationsmethoden und die Sprache, die von beiden Bounded Contexts verwendet werden.

Shared-Kernel widersprechen in gewisser Weise einem Kernprinzip von Bounded Contexts: Nur ein Team soll einen Bounded Context besitzen. Der gemeinsam genutzte Shared-Kernel ist allerdings im gemeinsamen Besitz mehrerer Teams. Der Schlüssel zur Implementierung des Shared-Kernel-Musters besteht darin, den Umfang des Shared-Kernels klein zu halten und nur auf den Integrationsvertrag zu beschränken.

Diese Art der Kooperation wird wie in Bild 14.7, Teilbild A.2, dargestellt. Der Shared-Kernel wird von mehreren Bounded Contexts sowohl referenziert als auch besessen. Jedem Team steht es frei, den Shared-Kernel zu ändern. Eine Änderung des Vertrags kann jedoch den Build des anderen Teams unterbrechen. Daher erfordert auch dieses Muster ein hohes Maß an Synchronisation zwischen den Teams.

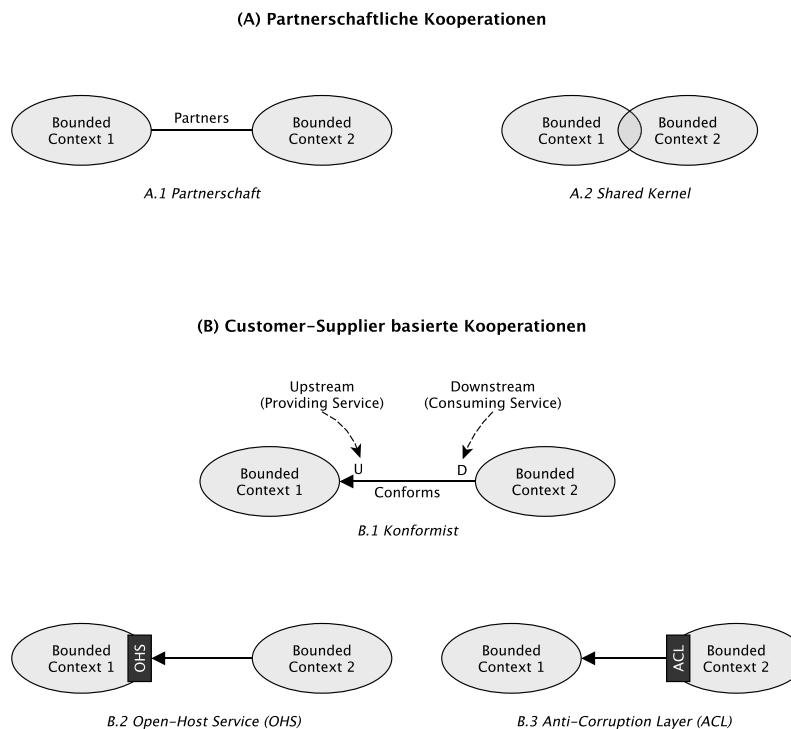


Bild 14.7 Mögliche Beziehungen zwischen Bounded Contexts

Shared-Kernel können eine disziplinierende Voraussetzung für die Integration von Bounded Contexts sein, die vom gleichen Team betrieben und implementiert werden. In einem solchen Fall können Ad-hoc-Integrationen der Bounded Contexts die Grenzen der Kontexte im Laufe der Zeit „verwaschen“. Ein Shared-Kernel kann hier für die explizite Definition des Integrationsvertrags verwendet werden. In diesen Szenarien wird auch das Prinzip des Kontextbesitzes durch ein Team nicht verletzt – beide Bounded Contexts werden ja vom selben Team implementiert.

14.2.4.2 Customer-Supplier-Kooperation

Die zweite Gruppe von Kooperationsmustern sind die sogenannten Customer-Supplier-Pattern. Anders als im Fall der Kooperation können beide Teams (Upstream und Downstream) unabhängig voneinander erfolgreich sein (siehe Bild 14.7 B). Daher gibt es in vielen Fällen ein Machtungleichgewicht: Entweder das vorgelagerte oder das nachgelagerte Team kann den Integrationsvertrag diktieren.

DDD sieht die folgenden drei Muster vor, solche Machtungleichheiten zu adressieren.

Beim **Konformist-Pattern** ist das Kräfteverhältnis zugunsten des Upstream-Teams verschoben, das keine wirkliche Motivation hat, die Bedürfnisse seiner Kunden zu unterstützen. Stattdessen stellt es nur den Integrationsvertrag zur Verfügung, der nach seinem eigenen Modell definiert ist – „*Take it or leave it*“.

Solche Machtungleichgewichte existieren häufig bei unternehmensexternen Service-Providern, können aber auch durch interne Organisationspolitik verursacht werden. Wenn das Downstream-Team das Modell des Upstream-Teams akzeptieren kann, wird die Beziehung zwischen den Bounded Contexts als konformistisch bezeichnet und die Beziehung wie in Bild 14.7, Teilbild B.1, dargestellt.

Das Downstream-Team ist also konform mit dem Modell des Upstream-Teams. Die Entscheidung des Downstream-Teams, einen Teil seiner Autonomie aufzugeben, kann durchaus sinnvoll sein. Zum Beispiel kann der vom Upstream-Team offengelegte Vertrag ein branchenübliches und gut etabliertes Modell sein.

Ist das Downstream-Team hingegen nicht bereit, das Modell des Upstream-Teams vorbehaltlos zu akzeptieren, kann ein **Anti-Corruption-Layer** angewendet werden. Wie im Fall des konformistischen Musters ist das Machtgleichgewicht in dieser Beziehung immer noch zugunsten des Upstream-Services verschoben. In diesem Fall ist der Downstream Bounded Context jedoch nicht bereit, sich bedingungslos anzupassen. Stattdessen wird das Modell des Upstream Bounded Contexts über eine Antikorruptionsschicht in ein Modell übersetzt, das auf die Bedürfnisse des eigenen Bounded Contexts zugeschnitten ist. Diese Beziehung wird wie in Bild 14.7, Teilbild B.3, dargestellt.

Das Anti-Corruption-Layer-Pattern adressiert Szenarien, in denen es nicht wünschenswert oder den Aufwand wert ist, dem Modell des Downstream-Teams vollständig zu entsprechen. Dies kann mehrere Gründe haben:

- Wenn der nachgelagerte Bounded Context eine Core-Subdomäne enthält – also sehr spezifisch ist. Das Modell einer Core-Subdomäne erfordert besondere Aufmerksamkeit, und die Einhaltung des Modells des Suppliers könnte die Modellierung der eigenen Problemdomäne behindern.

- Wenn das vorgelagerte Modell schlecht oder unpassend und gegebenenfalls über Jahrzehnte gewachsen ist. Dies ist häufig bei der Integration von Altsystemen der Fall.
- Wenn sich der Vertrag des Suppliers häufig ändert und der Consumer sein Modell vor solchen häufigen Änderungen schützen möchte. Mit einer Antikorruptionsschicht wirken sich die Änderungen im Modell des Lieferanten nur auf den Übersetzungsmechanismus aus.
- Wenn das Modell des Suppliers sehr groß ist und nur Teile dieses Modells benötigt werden. Während das Machtungleichgewicht beim Anti-Corruption-Layer-Pattern also auf Downstream-Seite gelöst wird, kann es auch auf Upstream-Seite gelöst werden. Das sogenannte **Open-Host-Service-Pattern** adressiert den Fall, dass die Macht in Richtung der Consuming-Services verzerrt ist. Der Supplier ist also daran interessiert, seine Consumer zu schützen und den bestmöglichen Service zu bieten. Um die Verbraucher vor Änderungen an seiner Implementierung zu schützen, entkoppelt der Upstream-Supplier sein Implementierungsmodell von der öffentlichen Schnittstelle. Diese Entkopplung ermöglicht es dem Supplier, sein Implementierungsmodell und sein öffentliches Modell mit unterschiedlicher Geschwindigkeit weiterzuentwickeln.

Eine derartige Beziehung wird wie in Bild 14.7, Teilbild B.2, dargestellt. Die öffentliche Schnittstelle des Anbieters ist also nicht dazu gedacht, seiner eigenen Ubiquitous Language zu entsprechen, sondern soll für die Consumer ein bequemes Protokoll bereitstellen, das in einer integrationsorientierten Sprache ausgedrückt ist. Daher wird das öffentliche Protokoll als „veröffentlichte Sprache“ (Published Language) bezeichnet. In gewissem Sinne ist das Open-Host-Service-Muster eine Umkehrung des Musters der Antikorruptionsschicht: Anstelle des Consumers implementiert der Supplier die Übersetzung seines internen Modells für seine Kunden.

Es ist von entscheidender Bedeutung, dass die veröffentlichte Schnittstelle gut dokumentiert und für die Consumer bequem ist. In diesem Zusammenhang sei u. a. die OpenAPI-Spezifikation (OpenAPI 2018) (vormals Swagger Specification) genannt, die einen Standard zur Beschreibung von REST-konformen Programmierschnittstellen (API) definiert und anstrebt, ein offenes und herstellerneutrales Beschreibungsformat für insbesondere REST-basierte API-Services bereitzustellen. Lösungen dieser Kategorie werden daher gerne insbesondere bei der Dokumentation von REST-basierten APIs eingesetzt, da sie es dem Upstream-Provider ermöglichen, die veröffentlichte Sprache effizient zu dokumentieren.

14.2.4.3 Separate Ways

Die letzte Möglichkeit der Zusammenarbeit ist, gar nicht zusammenzuarbeiten. Streng genommen kann man dies nicht wirklich als Kooperationsmuster bezeichnen. Dieses Muster kann dennoch aus verschiedenen Gründen auftreten. Es kann in Fällen, in denen die Teams nicht bereit oder in der Lage sind zusammenzuarbeiten, angewendet werden. Dies ist meist auf eine oder mehrere der folgenden Ursachen zurückzuführen.

1. **Probleme mit der Kommunikation:** Ein häufiger Grund für die Vermeidung von Zusammenarbeit sind Kommunikationsschwierigkeiten, die durch die Größe der Organisation oder interne politische Probleme verursacht werden. In diesen Fällen kann es kostengünstiger sein, Funktionalität in mehreren Bounded Contexts zu duplizieren.
2. **Generic Subdomains (Librarys):** Wenn die fragliche Subdomäne generisch ist, kann es, wenn die generische Lösung einfach zu integrieren ist, kostengünstiger sein, sie in jedem der Bounded Contexts lokal zu integrieren. Ein Beispiel ist ein Logging-Framework.

Es wird meist wenig Sinn machen, es als Service bereitzustellen. Die Duplizierung der Funktionalität in mehreren Bounded Contexts ist letztlich meist weniger kostspielig als die Zusammenarbeit.

3. **Modellunterschiede:** Wenn Modelle von Bounded Contexts so unterschiedlich sind, dass eine konforme Beziehung nicht möglich ist und die Implementierung einer Antikorruptionsschicht teurer wäre als die Duplizierung der Funktionalität. Auch in einem solchen Fall ist es für die Teams oft kostengünstiger, getrennte Wege zu gehen.

Es kann also gute (und auch weniger gute, siehe Punkt 1) Gründe geben, getrennte Wege zu gehen. Das Separate-Ways-Pattern sollte jedoch bei der Integration von Core-Subdomains auf alle Fälle vermieden werden. Eine doppelte Implementierung solcher Subdomains würde die Strategie des Unternehmens, diese möglichst effektiv und optimiert zu implementieren, konterkarieren.

14.2.4.4 Context Maps als Landkarte von Machtverhältnissen

Context Maps sind eine visuelle Top-Level-Darstellung aller Bounded Contexts eines Systems. Diese erstaunlich einfache visuelle Notation der gezeigten Pattern (siehe etwa Bild 14.8) gibt bereits wertvolle strategische Einblicke auf mehreren Ebenen:

- **High-Level-Design:** Eine Context Map bietet einen Überblick über die Komponenten und Modelle des Systems.
- Es werden insbesondere **Kommunikationsmuster** zwischen den Teams dargestellt und welche Teams nach welchen Strategien miteinander zusammenarbeiten. Insbesondere werden Machtgefälle zwischen Teams explizit.
- **Organisatorische Fragen:** Context Maps liefern somit nicht nur Einblick in das technische Systemdesign, sondern auch in organisatorische Fragen und zeigen gegebenenfalls bereits frühzeitig Probleme auf. Was bedeutet es beispielsweise, wenn Downstream-Consumer eines bestimmten Upstream-Teams alle auf Antikorruptionsschichten zurückgreifen oder wenn sich alle Separate Ways auf dasselbe Team konzentrieren?

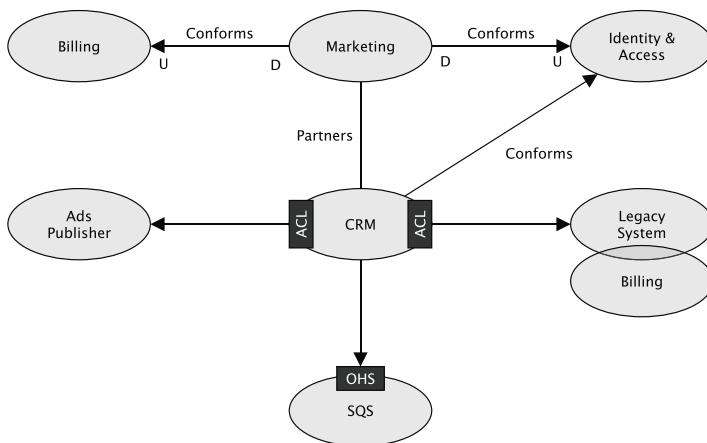


Bild 14.8
Exemplarische
Context Map

■ 14.3 Taktisches Design

Bislang haben wir uns vor allem mit strategischen Designentscheidungen befasst; insbesondere den Prinzipien für die Unterteilung von Geschäftsdomänen in Komponenten und die Modellierung der Interaktionen zwischen ihnen. Im Tactical Design geht es nun um die Implementierung dieser Komponenten.

Einige der hier besprochenen Pattern würden ein ganzes Buch rechtfertigen (Fowler 2002; Davis 2019; Scholl, Swanson und Jausovec 2019). Daher werden nur die wichtigsten und oft genutzten Pattern angesprochen und auf einem rein Überblick gebenden Niveau aufbereitet, mit dem Ziel, für das Tactical Design hilfreiche Entwurfsmuster zu erwähnen und auf deren Unterschiede einzugehen,

1. wie Geschäftslogik implementiert werden kann,
2. welche geeigneten Architekturmuster es für unterschiedliche Einsatzgebiete gibt
3. und wie Interaktionen zwischen Bounded Contexts in Cloud-native Anwendungen und Diensten gehandhabt werden können.

14.3.1 Oft genutzte Pattern für Geschäftslogik

Wie in Abschnitt 14.2.1 deutlich wurde, sind nicht alle Subdomänen gleich geschaffen. Verschiedene Subdomänen haben unterschiedliche Ebenen der strategischen Bedeutung und Komplexität. Nachfolgend werden daher vier verschiedene Arten der Implementierung von Geschäftslogik für Core und Supporting Subdomains herausgegriffen, die sich jeweils für einen anderen Grad an Komplexität in der Geschäftsdomäne eignen.

- Das ETL-Pattern in Abschnitt 14.3.1.1 zum Laden von Daten aus Supporting Subdomains.
- Das Active Record-Pattern in Abschnitt 14.3.1.2 zur Abbildung komplexerer Datenstrukturen in Supporting Subdomains.
- Das Domain Model-Pattern in Abschnitt 14.3.1.3 zur Abbildung der Zusammenhänge in Core Subdomains.
- Das Event-Sourcing-Pattern in Abschnitt 14.3.1.4 zur Handhabung von zeitlichen Ereignisabfolgen in Core Subdomains.

14.3.1.1 Das ETL-Pattern (primär Supporting Subdomains)

Das **Extract-Transform-Load-(ETL-)Muster** eignet sich gut für die einfachsten Problemdomänen, in denen die Geschäftslogik ETL-Operationen (Extract-Transform-Load) ähnelt. Wie Bild 14.9 zeigt, extrahiert jede Operation Daten aus einer Quelle, wendet dann eine Transformationslogik an, um sie in eine andere Form umzuwandeln, und lädt das Ergebnis in den Zielspeicher, aus dem es dann seiner eigentlichen Verarbeitung zugeführt wird.



Bild 14.9
Das Extract-Transform-Load-(ETL-)Pattern

Die Geschäftslogik wird durch einfache Prozeduren realisiert. Jede Prozedur wird als einfaches, unkompliziertes prozedurales Skript implementiert. Sie kann eine dünne Abstraktionschicht für die Integration mit Speichermechanismen verwenden, ist aber auch frei, direkt auf die Datenbanken zuzugreifen.

Die einzige Anforderung, die Prozeduren erfüllen müssen, ist das transaktionale Verhalten. Jede Operation sollte entweder erfolgreich sein oder fehlschlagen. Wenn ein Prozess aus irgendeinem Grund fehlschlägt, sollte das System in einem konsistenten Zustand bleiben – daher der Name des Musters, Transaktionsskript.

14.3.1.2 Das Active Record-Pattern (primär Supporting Subdomains)

Wie das ETL-Pattern, unterstützt auch dieses Muster Fälle, in denen die Geschäftslogik einfach ist. Hier kann die Geschäftslogik jedoch auf komplexeren Datenstrukturen operieren. Zum Beispiel können anstelle flacher Datensätze kompliziertere Objektbäume und Hierarchien – wie in Bild 14.10 gezeigt – behandelt werden.

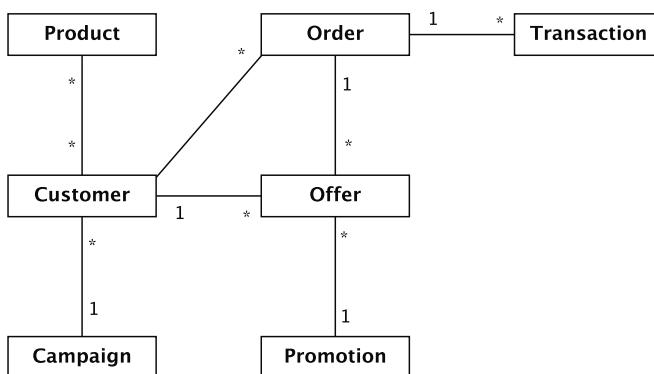


Bild 14.10
Beispiel eines
Active Record

Der Betrieb auf solchen Datenstrukturen über ein ETL-Skript würde zu viel sich wiederholendem Code führen. Daher verwendet dieses Muster dedizierte Objekte, um komplexe Datenstrukturen zu repräsentieren: aktive Datensätze. Neben der Datenstruktur implementieren diese Objekte auch Datenzugriffsmethoden zum Erstellen, Lesen, Aktualisieren und Löschen von Datensätzen – die sogenannten CRUD-Operationen. Folglich sind die aktiven Datensatzobjekte von einem objektrelationalen Mapping (ORM) oder einem anderen Datenzugriffs-Framework abhängig. Der Name des Musters leitet sich von der Tatsache ab, dass jedes Record „aktiv“ ist und die erforderliche Datenzugriffslogik implementiert. Anders als beim Transaction Script-Pattern erfolgt der Zugriff nicht direkt auf einer Datenbank, sondern über die Active Records, beispielsweise wie in Listing 14.1 veranschaulicht. Mehrere transaktional zusammenhängende Zugriffe auf Active Records können dabei durchaus in Transaktionen eingebettet werden.

Listing 14.1 Nutzung von Active Records

```

public void createUser(String name, String email) {
    try {
        DB.beginTransaction();

        User usr = new User(name, email); // Active Record-Code
        usr.save(); // Active Record-Code

        DB.commit()
    } catch {
        DB.rollback();
    }
}

```

14.3.1.3 Das Domain Model-Pattern (primär Core Subdomains)

Nach der Definition von Martin Fowler ist ein **Domänenmodell** ein Objektmodell einer Domäne, das sowohl Verhalten als auch Zustand (Daten) enthält. Die taktischen Pattern (Evans 2003) von DDD liefern u. a. die folgenden Bausteine für solche Objektmodelle:

- Value Object
- Aggregate
- Domain-Event

Alle diese Muster haben eines gemeinsam. Die Geschäftslogik steht an erster Stelle und vor allem in direktem Bezug zur Ubiquitous Language. Dieses Muster ermöglicht es dem Code, die Ubiquitous Language zu „sprechen“ und den mentalen Modellen der Domänenexperten konzeptionell sehr eng zu folgen.

Value Objects sind dabei Objekte, die durch ihre Werte eindeutig identifiziert werden können und sich nicht ändern (Immutables). Ein triviales Beispiel für ein Value Object ist das String-Objekt in Java. Es ist unveränderlich, und alle seine Operationen führen zu einer neuen Instanz eines Strings.

Ein **Aggregat** ist eine Entität, die eine Hierarchie von Objekten darstellt. Im Gegensatz zu einem Wertobjekt kann eine Entität nicht allein durch ihren Wert identifiziert werden. Das heißt, jedes Aggregat benötigt ein ID-Feld, um identifiziert zu werden. Aggregate sind (anders als Value Objects) Mutables, d. h., sie können ihren Zustand ändern.

DDD sieht vor, dass der Entwurf eines Systems von seiner Fachlichkeit getrieben sein sollte. Aggregate bilden da natürlich keine Ausnahme. Ein Aggregat ist daher selten ein flacher Datensatz, meist ist ein Aggregat eine Hierarchie zusammengehöriger Objekte (siehe Bild 14.10).

Daher ist es entscheidend, die Konsistenz von Aggregatzuständen zu schützen. Das Aggregatmuster zieht dazu eine klare Grenze zwischen dem Aggregat und seinem äußeren Bereich (siehe Bild 14.11). Nur die Geschäftslogik des Aggregats darf seinen Zustand verändern. Alle Prozesse oder Objekte außerhalb des Aggregats dürfen nur seinen Zustand lesen oder seine öffentlichen Methoden ausführen. Die öffentlichen Methoden des Aggregats sind für die Validierung der Eingabe und die Durchsetzung aller Geschäftsregeln und Invarianten verantwortlich. Diese strenge Abgrenzung stellt auch sicher, dass die gesamte Geschäftslogik in Bezug auf das Aggregat an einer Stelle implementiert wird – im Aggregat selbst.

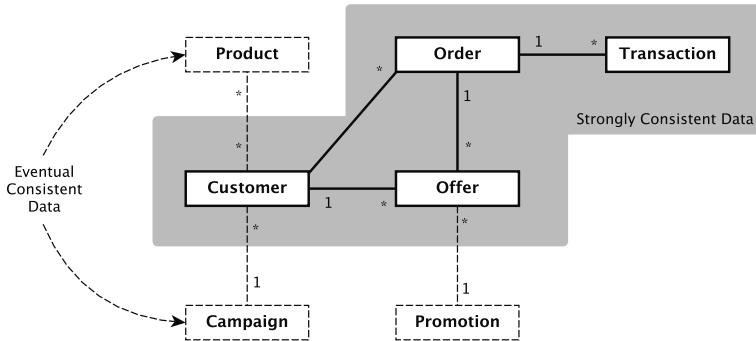


Bild 14.11 Beispiel einer Aggregatgrenze

Da der Zustand von Aggregaten nur durch ihre eigene Geschäftslogik geändert werden kann, fungiert die Aggregatgrenze in DDD auch immer als eine Transaktionsgrenze. Alle Änderungen am Zustand des Aggregats sollten transaktional als eine atomare Operation übertragen werden. Eine Transaktion darf sich also nur auf ein Aggregat beziehen.

Da alle in einem Aggregat enthaltenen Objekte dieselbe Transaktionsgrenze haben, können allerdings Leistungs- und Skalierbarkeitsprobleme auftreten, wenn Aggregate zu groß werden. Man kann dies nutzen, um Grenzen von Aggregaten anhand eines Strict- und Eventual-Consistency-Kriteriums zu schneiden (siehe Bild 14.11). Die Konsistenz der Daten kann somit eine praktische Heuristik für den Entwurf der Aggregatgrenzen sein. Nur die Informationen, die für die Implementierung der Geschäftslogik des Aggregats streng konsistent sein müssen, sollten sich innerhalb der Grenzen des Aggregats befinden. Objekte, die *eventual consistent* sein können, sollten nicht zum Aggregat gehören und nur über ihre ID referenziert werden. Da ein Aggregat meist eine Hierarchie von Objekten darstellt, sollte aus Gründen des Zustandsschutzes von Aggregaten nur eines von ihnen als öffentliche Schnittstelle des Aggregats bestimmt werden – die Wurzel des Aggregats (Aggregate Root).

Im Allgemeinen werden vier Regeln genannt (Vernon 2017), die berücksichtigt werden sollten, „gute“ Aggregate zu definieren:

1. Schütze fachliche Invarianten innerhalb von Aggregatengrenzen (z. B. mittels Aggregate Roots).
2. Entwirf kleine Aggregate (d. h., Aggregate so klein wie möglich halten, indem nur Objekte aufgenommen werden, die von der Geschäftsdomäne in einem stark konsistenten Zustand benötigt werden).
3. Referenziere andere Aggregate nur über ihre Identität.
4. Aktualisiere andere Aggregate unter Verwendung von Eventual Consistency.

Ein **Domain-Event** ist eine Nachricht, die ein bedeutendes Ereignis der Domäne beschreibt, das in der Geschäftsdomäne aufgetreten ist, wie z. B.: Auftrag bezahlt, Lager wieder aufgefüllt oder Werbekampagne veröffentlicht. Solche Events sind ein Teil der öffentlichen Schnittstelle eines Aggregats. Ein Aggregat veröffentlicht über sein Public Interface solche Ereignisse in Form von Domain-Events.

Domain-Events sind ein Teil der öffentlichen Schnittstelle eines Aggregats. Ein Aggregat veröffentlicht seine Ereignisse. Wie Bild 14.12 zeigt, können andere Prozesse, Aggregate

oder sogar externe Systeme die Domänenereignisse abonnieren und als Reaktion darauf ihre eigene Logik im Rahmen von Pub/Sub- oder Queueing-Pattern ausführen, die bereits in Abschnitt 12.2.4 und Abschnitt 12.4.2 behandelt wurden.

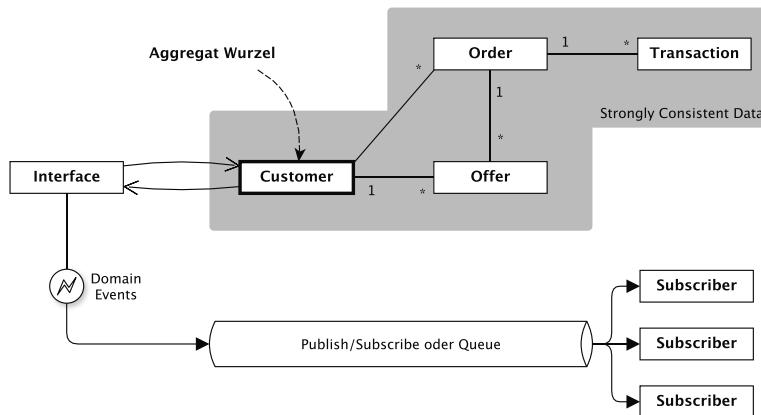


Bild 14.12 Kommunikation mit Aggregaten nur über ihre Wurzel

14.3.1.4 Das Event-Sourcing-Pattern (primär Core Subdomains)

Das **Event-Sourcing-Pattern** verwendet Domain-Events von Aggregaten des Domain Models (siehe Abschnitt 14.3.1.3), um die Dimension der Zeit in das Domänenmodell einzuführen. Jede Änderung des Systemzustands sollte als Domänenereignis ausgedrückt und aufgezeichnet werden.

Die Datenbank, die die Domänenereignisse des Systems speichert, ist der einzige stark konsistente Speicher – die „Single Source of Truth“ des Systems. Jede Operation an einem Aggregat, das von Ereignissen gespeist wird, folgt diesem Ablauf (siehe auch Bild 14.13):

- Laden der Domänenereignisse des Aggregats.
- Erstellung der Zustandsdarstellung.
- Ausführen der Geschäftslogik und Erzeugen neuer Domain-Events.
- Übertragen der neuen Domain-Events in den Event Store.

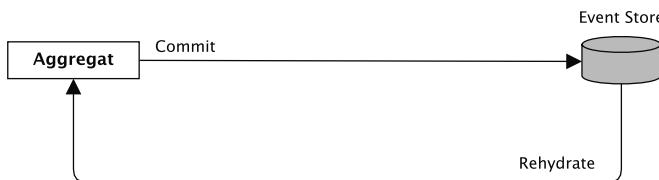


Bild 14.13
Event-Sourcing

Event-Sourcing wird z. B. in der Finanzindustrie zur Darstellung von Änderungen in einem Ledger verwendet. Ein Ledger ist ein Append-Only-Log, das den Verlauf von Transaktionen protokolliert. Ein aktueller Zustand (z. B. der Kontostand) kann immer durch ein Replaying der Ledger-Datensätze abgeleitet werden. Event-Sourcing ist somit besonders praktisch für Systeme, die Geld oder monetäre Transaktionen verwalten. Es erlaubt, die Entscheidungen, die das System getroffen hat, und den Geldfluss zwischen Konten leicht nachzuvollziehen.

Im Vergleich zu einer zustandsbasierten Darstellung erfordert das Event-Sourcing-Modell mehr Aufwand bei der Modellierung. Allerdings ist dieses Muster insbesondere in den folgenden Szenarien erwägenswert.

- **Replaying Time Machine:** Da Domain-Events verwendet werden können, um den aktuellen Zustand eines Aggregats wiederherzustellen, können sie auch für die Wiederherstellung aller vergangenen Zustände des Aggregats verwendet werden.
- **Zustands- und Verhaltensanalyse:** Event-Sourcing bietet tiefe Einblicke in den Zustand und das Verhalten des Systems. Außerdem ermöglicht das flexible Modell die Umwandlung der Ereignisse in verschiedene Zustandsdarstellungen – auch solche, die ursprünglich nicht geplant waren.
- **Auditierbarkeit:** Die persistierten Domain-Events stellen ein stark konsistentes Audit-Protokoll von allem dar, was mit den Zuständen der Aggregate passiert ist. Gesetze verpflichten einige Geschäftsdomänen, solche Audit-Protokolle zu implementieren (z. B. Finanzindustrie). Event-Sourcing bietet dies von Haus aus.

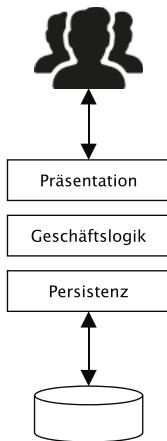
14.3.2 Oft genutzte Pattern für die Architektur

Während sich Abschnitt 14.3.1 insbesondere mit geeigneten Pattern befasst hat, um Programmlogik für Core und Supporting Domains zu entwerfen, wird sich dieser Teil damit befassen, wie diese Programmlogik mit weiteren Systemkomponenten im Rahmen einer Systemarchitektur wechselwirken sollte, um effektiv in ein Gesamtsystem eingebettet werden zu können.

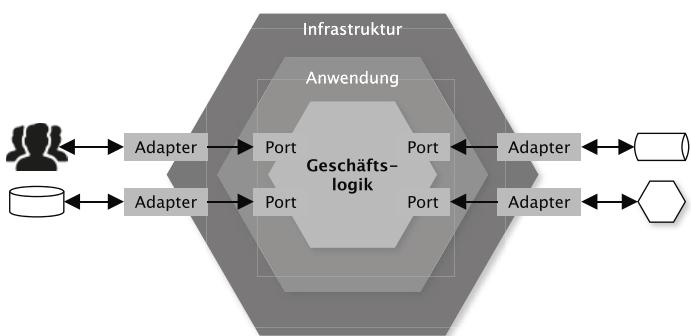
Hierfür haben sich diverse Architekturmuster entwickelt (siehe Bild 14.14). Welche Architekturmuster hierfür im Einzelnen verwendet werden, ist eine wichtige taktische Designentscheidung. Das richtige Muster unterstützt die Umsetzung der funktionalen und nicht-funktionalen Anforderungen von Systemen. Wir werden uns drei häufig anzutreffende Architekturmuster für Cloud-native Anwendungen und geeignete Anwendungsfälle ansehen. Der Leser sollte aber beachten, dass es eine Vielzahl weiterer Muster (Fowler 2002) gibt und hier nur oft genutzte Muster betrachtet werden.

1. Layered Architecture (Abschnitt 14.3.2.1)
2. Ports & Adapters (Abschnitt 14.3.2.2)
3. CQRS (Command Query Responsibility Segregation, Abschnitt 14.3.2.3)

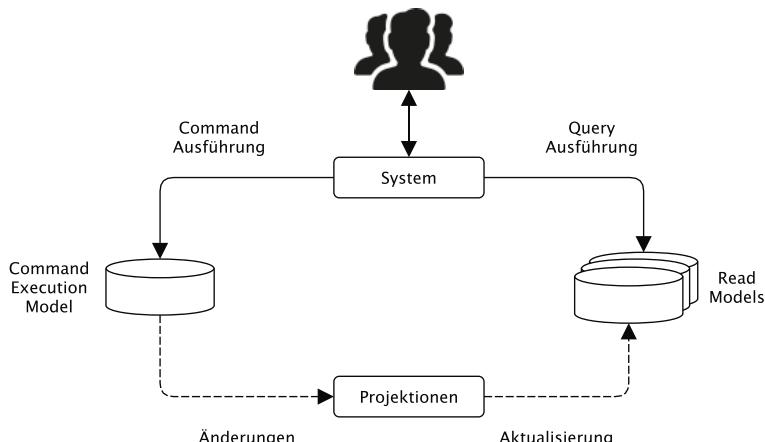
(A) Layered Architecture



(B) Ports & Adapter (hexagonale) Architektur



(C) Command Query Responsibility Segregation (CQRS)

**Bild 14.14** Architektur-Pattern

14.3.2.1 Die Ebenen-Architektur

Bei der **Ebenen-Architektur** (manchmal auch 3-Tier-Architekturmuster genannt) handelt es sich um DAS Architekturmuster, das vermutlich jedem Leser bekannt ist.

Die Präsentationsebene definiert und stellt dabei die Benutzeroberfläche dar. Die Logikebene implementiert die Geschäftslogik, und die Persistenzebene ermöglicht Zugriff auf Persistenzmechanismen wie beispielsweise Datenbanken oder andere Infrastrukturkomponenten wie File-Storage-Systeme und Ähnliches.

Durch die Abhängigkeit zwischen der Geschäftslogik und den Datenzugriffsschichten passt dieses Architekturmuster sehr gut zu Systemen, deren Geschäftslogik mit dem Active Record-Pattern (siehe Abschnitt 14.3.1.2) implementiert wurde. Das Pattern ist daher häufig in Bounded Contexts für **Supporting Subdomains** anzutreffen.

Das Ebenen-Modell impliziert aber, dass die Interaktion mit dem System ausschließlich von Benutzern über die Präsentationsebene im Sinne eines Human-Machine-Interface ausgeht. Grundsätzlich könnte man auch die Machine-to-Machine-Kommunikation über diesen Layer abwickeln, doch wird dies meist so konzeptionell nicht vorgesehen. Daher wurde die Ebenen-Architektur weiterentwickelt, um vielfältigere Interaktionen zwischen Logiken vorzusehen.

14.3.2.2 Das Ports & Adapter-Pattern

Das Ports & Adapter-Pattern (auch hexagonale Architektur genannt) wurde von Alistair Cockburn vorgeschlagen (Cockburn 2005) und ist der Ebenen-Architektur insofern ähnlich, als dass die Codebasis des Systems ebenfalls nach technologischen Gesichtspunkten zerlegt wird. Das Ports & Adapter-Pattern unterscheidet sich jedoch hinsichtlich:

- **Terminologie:** Es werden die Benutzeroberfläche des Systems, der Code für den Datenzugriff und alle anderen infrastrukturellen Belange einfach als „Infrastruktur“ bezeichnet. Dadurch wird insbesondere die Machine-to-Machine-Interaktion konzeptionell besser berücksichtigt.
- **Abhängigkeiten:** Anstatt von technologischen Belangen dominiert (UI-Technologien und Persistenz-Technologien) zu sein, nimmt die Geschäftslogik in der Ports & Adapters-Architektur die zentrale Rolle ein. Sie ist nicht direkt von einer der Infrastrukturkomponenten des Systems abhängig.
- **Anwendungsschicht:** Die Anwendungsschicht implementiert eine Fassade für die öffentlichen Schnittstellen des Systems. Sie beschreibt alle vom System angebotenen Operationen und choreografiert die Geschäftslogik des Systems, um diese auszuführen.

Wie Bild 14.14 B und die von vielen Autoren gewählte hexagonale Darstellung dieses Patterns visualisieren soll, ist das Ziel der Ports & Adapter-Architektur, die Geschäftslogik des Systems von den Infrastrukturkomponenten zu entkoppeln. Anstatt die Infrastrukturkomponenten direkt zu referenzieren und aufzurufen, definiert die Geschäftslogikschicht Ports, die von der Infrastrukturschicht implementiert werden müssen. Die Infrastrukturschicht implementiert Adapter der Ports für die Arbeit mit verschiedenen Technologien. Die Anwendungsschicht liefert die Adapter für die Ports der Geschäftslogik oft mittels Dependency Injection.

Die Entkopplung der Geschäftslogik von allen technologischen Belangen macht das Ports & Adapter-Pattern damit vor allem für Geschäftslogiken, die auf dem Domain-Model-Pattern (siehe Abschnitt 14.3.1.3) implementiert werden, interessant. Das Pattern ist daher häufig in Bounded Contexts für **Core Subdomains** anzutreffen.

14.3.2.3 Das CQRS-Pattern

Die von Greg Young vorgeschlagene Command Query Responsibility Segregation (CQRS) ist ein Entwurfsmuster zur Anbindung von Stateful-Services wie Datenbanken (Millett und Tune 2015). Das Pattern sieht eine Aufteilung des Objektmodells in zwei Objektmodelle hinsichtlich der Zugriffe auf eine Datenbank vor:

- ein Objektmodell für schreibende Zugriffe
- mehrere optimierte Objektmodelle für lesende Zugriffe

Im Gegensatz zu dem eher üblichen Create-Read-Update-Delete-(CRUD-)Modell, mit dem Zugriffe auf eine Datenbank mit einem gemeinsamen Objektmodell strukturiert werden (siehe

Active Record-Pattern in Abschnitt 14.3.1.2), gibt es bei CQRS mehrere Objektmodelle, ein Objektmodell für das Schreiben und mehrere für spezifische Fragestellungen optimierte Objektmodelle für das Lesen. In der CQRS-Architektur (siehe Bild 14.14 C) sind die Zuständigkeiten der Modelle des Systems also nach ihrem Schreib-Lese-Charakter getrennt. *Commands* dürfen nur auf dem stark konsistenten Befehlsausführungsmodell operieren. *Queries* können keine der Systemzustände direkt verändern – weder die Lesemodelle noch das Befehlsausführungsmodell.

Wie Bild 14.14 C zeigt, basiert CQRS somit nur auf einem einzigen Modell zur Ausführung von Operationen, die den Zustand des Systems verändern (*Commands*). Gemäß dem CQRS-Pattern ist dieses Command Execution-Modell das einzige Modell, das stark konsistente Daten repräsentiert (Single Source of Truth).

Es können aber beliebig viele Modelle mittels Projektionen erzeugt werden, die für die Darstellung von Daten für Benutzer oder andere Systeme erforderlich sind. Diese Modelle sind Read-only und mittels *Queries* abfragbar. Keine der Operationen des Systems kann die Daten der gelesenen Modelle direkt ändern.

In vielen anderen „klassischen“ Kontexten werden ebenfalls unterschiedliche Modelle für unterschiedliche Anforderungen des Systems genutzt – insbesondere dann, wenn die Datens Mengen oder Ereignisfrequenzen substanzielle Umfänge annehmen. So gibt es beispielsweise in vielen (auch monolithischen) Unternehmensarchitekturen optimierte Datenmodelle für Online-Transaktionsverarbeitung (OLTP), Online-Analytical Processing (OLAP) und Suche.

Dieser Ansatz harmoniert auch gut mit der in Microservice-Architekturen vorzufindenden **polyglotten Persistenz**, bei der mehrere Datenbanken für unterschiedliche Anforderungen an den Datenzugriff verwendet werden. Ein einzelnes System könnte zum Beispiel eine Dokumentendatenbank (Document Store) als operative Datenbank, einen Column Store für Analysen/Berichte und eine Suchmaschine für die Implementierung von Suchfunktionen verwenden.

CQRS eignet sich insbesondere für ereignisgesteuerte Domänenmodelle (siehe Abschnitt 14.3.1.4). Reines Event-Sourcing lässt es nicht zu, Datensätze basierend auf den Zuständen der Aggregate abzufragen, aber CQRS kann dies ergänzend ermöglichen, indem die Zustände in abfragbare Datenbanken projiziert werden. CQRS ist daher häufig in Bounded Contexts mit Event-Sourcing von **Core Subdomains** anzutreffen.

Projektionen lassen sich grundsätzlich synchron oder asynchron erzeugen. Sychrone Projektionen sind dabei in Fehlerfällen einfacher zu handhaben, können Read-Modelle aber nicht in „Echtzeit“ bereitstellen. Häufig werden synchrone Projektionen periodisch (z. B. einmal in der Nacht, einmal pro Woche etc.) per Batch-Jobs generiert. Asynchrone Projektionen können Read-Models bei Eintreten eines Events aktualisieren. Die Handhabung in Fehlerfällen ist aber schwieriger, da oft nicht nachvollzogen werden kann, welche Aktualisierungen nun erfolgreich in welche Read-Models projiziert werden konnten. Dann bleibt unklar, ob die Read-Models noch zueinander konsistent sind.

Synchre Projektionen wie in Bild 14.15 A funktionieren grundsätzlich nach folgendem Verfahren:

- Die Projektions-Engine fragt das Command Execution-Modell nach Datensätzen ab, die nach dem letzten verarbeiteten Checkpoint (oft ein Zeitstempel) hinzugefügt oder aktualisiert wurden.

- Die Projektions-Engine verwendet die aktualisierten Daten, um die Read-Models des Systems neu zu generieren bzw. zu aktualisieren.
- Die Projektions-Engine speichert den Checkpoint des zuletzt verarbeiteten Datensatzes. Dieser Wert wird bei der nächsten Iteration verwendet, um Datensätze abzurufen, die nach dem letzten verarbeiteten Datensatz hinzugefügt oder geändert wurden.

Bei **asynchronen Projektionen** veröffentlicht das *Command Execution Model*, wie in Bild 14.15 B, alle bestätigten Änderungen mittels eines Pub/Sub-Messaging-Bus. Die Projektions-Engines des Systems können die veröffentlichten Nachrichten abonnieren und sie zur Projektion der gelesenen Modelle verwenden.

Trotz der offensichtlichen Skalierungs- und Leistungsvorteile der asynchronen Projektionsmethode ist sie anfälliger für die Fallacies of Distributed Computing. Wenn die Nachrichten nicht in der richtigen Reihenfolge verarbeitet oder dupliziert werden, werden inkonsistente Daten in die gelesenen Modelle projiziert.

Die asynchrone Methode macht es auch schwieriger, neue Projektionen hinzuzufügen oder bestehende zu regenerieren. Aus diesen Gründen ist es ratsam, immer als Basis eine synchrone Projektion zu nutzen und nur bei Bedarf eine zusätzliche asynchrone Projektion darauf zu implementieren, also die Methoden aus Bild 14.15 A und Bild 14.15 B zu kombinieren.

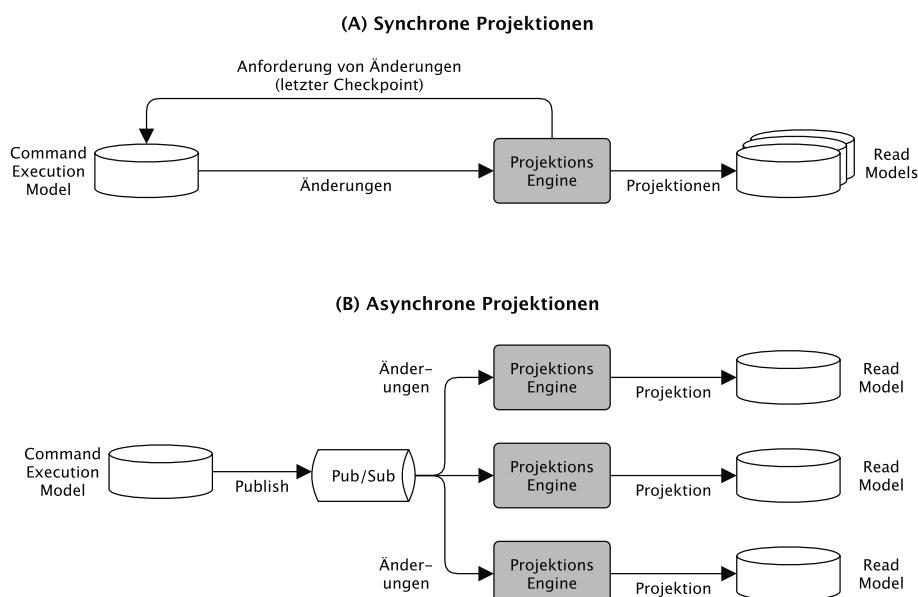


Bild 14.15 Sychrone und asynchrone Projektionen

■ 14.4 Zusammenfassung

Zusammenfassend betrachtet, beschreibt Domain-driven Design (DDD) einen Prozess und unterstützende Methodiken, die einer Philosophie der Ausrichtung der Softwareentwicklung an der Fachlichkeit folgen. DDD ist dabei vorgehensmodellagnostisch und kann mit unterschiedlichen (agilen) Vorgehensmodellen verwendet werden. Dabei durchläuft man strategische und taktische Entwurfsschritte, die dabei – ausgehend von einem Problemraum – methodisch unterstützen, eine Lösung im Lösungsraum zu definieren.

Auf Ebene des strategischen Designs (siehe Abschnitt 14.2) werden dabei in einem ersten Schritt Subdomänen identifiziert. Entwicklungs- und Verfeinerungsaufwände werden dabei primär in Core und teilweise in Supporting Subdomains investiert. Insbesondere die Schritte im strategischen Design helfen dabei, den Problemraum zu strukturieren und Aufwände in die Bereiche zu lenken, die Unternehmen und Organisationen am ehesten helfen, sich positiv von Wettbewerbern abzuheben. In diesen Bereichen werden sogenannte Bounded Contexts definiert, die letztlich den technischen Lösungsraum strukturieren und etwa auf Microservice (siehe Kapitel 12) abgebildet werden können.

Bounded Contexts bilden somit den Übergang vom strategischen zum taktischen Design. Modelle in verschiedenen Bounded Contexts können dabei grundsätzlich unabhängig voneinander entwickelt und implementiert werden. Bounded Contexts sind zwar unabhängig voneinander entwickelbar, aber nicht vollkommen isoliert voneinander. Auch Bounded Contexts müssen miteinander interagieren. Infolgedessen gibt es immer Berührungspunkte zwischen Bounded Contexts, die als Verträge bezeichnet werden und in Form von Context Maps visualisiert werden können (siehe Abschnitt 14.2.4 und Abschnitt 14.2.4.4). Dabei kann es Machtgefälle zwischen Bounded Contexts geben, die in Form von Kooperationen (siehe Abschnitt 14.2.4.1) oder Customer-Supplier-Abhängigkeiten (siehe Abschnitt 14.2.4.2) festgelegt werden sollten. Diese Context Maps bilden den Übergang vom Problemraum (strategisches Design) zum Lösungsraum (taktisches Design).

Im Bereich des taktischen Designs (siehe Abschnitt 14.3) werden Bounded Contexts dann softwaretechnisch weiter verfeinert. Hier greift man üblicherweise auf bekannte Architekturmuster (wie z. B. Layered Architecture, Ports & Adapters oder CQRS, siehe auch Abschnitt 14.3.2) sowie Entwurfsmuster (wie Domain Model, Event-Sourcing, Extract Transform Load, Active Record; siehe auch Abschnitt 14.3.1) zurück. Diese sind nicht DDD-spezifisch.

Die Referenz zur DDD-Methodik ist und bleibt sicher (Evans 2003). Einen wirklich guten, erfreulich kurzen und kompakten Überblick gibt (Khononov 2019). Aber wie bei den Themen Container, Orchestrierung und Microservices gibt es viele weitere Literatur, und der Leser wird sicher für seinen Lesegeschmack passende vertiefende Quellen finden. Eine komplette Auflistung guter und geeigneter Literatur würden den Rahmen hier sprengen.

Eine lange Liste geeigneter Architekturmuster findet sich in (Fowler 2002), sicher aus einer Zeit, bevor der Begriff Cloud-native überhaupt in seinem heutigen Verständnis geprägt wurde. Eher DDD-bezogene und den Cloud-nativen Trend aufnehmende Architekturmuster werden in (Millett und Tune 2015) behandelt.

Ergänzende Materialien

Auf der Website zum Buch finden sich zu diesem Kapitel ferner folgende Ergänzungsmaterialien:

- Tabellarische Aufstellung aller Patterns und Best Practices, die in diesem Buch zur Erstellung Cloud-nativer Anwendungen und Dienste behandelt wurden.
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer (Themen: Domain-driven Design; effektives Software-Design; strategisches Design: Subdomains, Bounded Contexts und Ubiquitous Language, Context Maps; taktisches Design: Pattern für Geschäftslogiken und für Architekturen)



<https://bit.ly/3ijmACJ>

IV

Teil IV: Sichere Cloud-native Anwendungen

15

Einleitung zu Teil IV

„Sicher ist, dass nichts sicher ist. Selbst das nicht.“

Joachim Ringelnatz, deutscher Schriftsteller, Kabarettist und Maler

IT-Sicherheit ist ein kritischer Faktor für alle Arten von Anwendungen, insbesondere, wenn diese dazu genutzt werden, sensible Daten zu verarbeiten. Traditionell wurde IT-Sicherheit oft als separate Funktion betrachtet. Dieser Ansatz hat jedoch seine Grenzen und kann dazu führen, dass IT-Sicherheit als isoliertes Problem betrachtet wird, das von anderen Aspekten getrennt ist. Mittlerweile hat es sich mehr durchgesetzt, IT-Sicherheit als integrierte Funktion zu betrachten. Dies bedeutet, dass IT-Sicherheit von Anfang an in die Planung und Entwicklung von IT-Systemen und Anwendungen einbezogen werden sollte, anstatt als nachträglicher Gedanke oder Zusatz zu fungieren. Das gilt in besonderem Maße für Cloud-native Anwendungen, die naturgemäß wesentliche größere Angriffsflächen als klassische Anwendungen bieten.

Bisher wurden solche Sicherheitsaspekte daher im Verlaufe dieses Buchs an geeigneten Stellen integriert behandelt, z. B. in den Abschnitten 8.3 (Container Runtime Environments), 9.3.9 (Isolation von Workloads), 12.3 (Architekturelle Sicherheit), 12.5.7 (Baue gut beobachtbare Services), 12.6.2 (API Gateway Pattern), 13.1 (Konsolidierung von Telemetriedaten) und 13.3.4 (Sicherheit durch Service Meshs). Das Problem bei solch integrierten Darstellungen ist aber, dass so Sicherheit zwar überall eine Rolle spielt und das Bewusstsein für Sicherheitsrisiken und deren Handhabung stärkt. Sicherheit an sich ist dann aber für den Leser in einem Buch nicht mehr an einer zentralen Stelle gebündelt; was ebenfalls nicht ideal ist.

Dieser Teil fokussiert daher vor allem die Fragestellung, was man bei der Entwicklung „sicherer“ Cloud-nativer Anwendungen zu berücksichtigen hat. IT-Sicherheit sollte dabei aber nie als isoliertes Problem, sondern als integrierte Funktion betrachtet werden. Um sich dabei in diesem Teil nicht zu wiederholen, wird an geeigneten Stellen auf Abschnitte verwiesen, die solche Sicherheitsaspekte gegebenenfalls bereits behandelt haben. Der Leser findet dennoch in diesem Teil Überlegungen zur Entwicklung Cloud-nativer Systeme, die sich vor allem aus dem Blickwinkel der IT-Sicherheit und des Datenschutzes ergeben.

Kapitel 16 befasst sich primär mit der technischen Sicherheit Cloud-nativer Systeme vor allem im Sinne einer Härtung von Systemen entlang von sogenannten Angriffsvektoren. Diese Systemhärtung betrachtet vor allem die Ebene virtueller Maschinen in Cloud-Infrastrukturen und die Ebene von Containern in Orchestrierungsplattformen.

Kapitel 17 fokussiert eher regulatorische Aspekte, die sich vor allem aus dem deutschen bzw. dem europäischen Rechtsraum (also vor allem aus der Datenschutz-Grundverordnung, DSGVO), aber auch aus internationalen Regularien zum Cloud Computing ergeben, denen

vor allem die Hyperscaler aus den USA unterworfen sind und damit auch deren Kunden betreffen. Um beide Aspekte ranken sich gewisse Mythen, die oft dazu herangezogen werden, Cloud-native Lösungen zu früh zu verwerfen. Diese „Mythen“ sollen in diesem Kapitel daher etwas versachlicht werden.

16

Härtung Cloud-nativer Anwendungen

„Haltet die Bösen immer voneinander getrennt. Die Sicherheit der Welt hängt davon ab.“

Theodor Fontane, deutscher Schriftsteller und bedeutender Vertreter des Realismus

Bild 16.1 zeigt ein Modell des Lebenszyklus von Cyber-Angriffen, welches in diesem Kapitel als lenkende Gedankenstütze dienen soll, wirkungsvolle Gegenmaßnahmen zur Härtung von Systemen zu erörtern. Nach diesem Modell durchläuft ein Angreifer verschiedene Phasen. Es beginnt mit der anfänglichen Erkundung und Kompromittierung der Zugangsmittel (oft über Mobiltelefone oder Desktop-PCs von Mitarbeitern, die in der Regel keine Sicherheitsexperten sind). Diese Schritte werden häufig durch Social-Engineering-Methoden und Phishing-Angriffe unterstützt. Ziel ist es, in die Nähe des System of Interest zu gelangen. Diese Social-Engineering Schritte werden in diesem Kapitel jedoch nicht behandelt, da technische Lösungen hier oft nicht in der Lage sind, den schwächsten Punkt der Sicherheit – den Menschen – einzudämmen. Der Leser sei auf entsprechende Forschungsarbeiten wie (Aleroud u. a. 2017) und (Heartfield u. a. 2015) verwiesen.

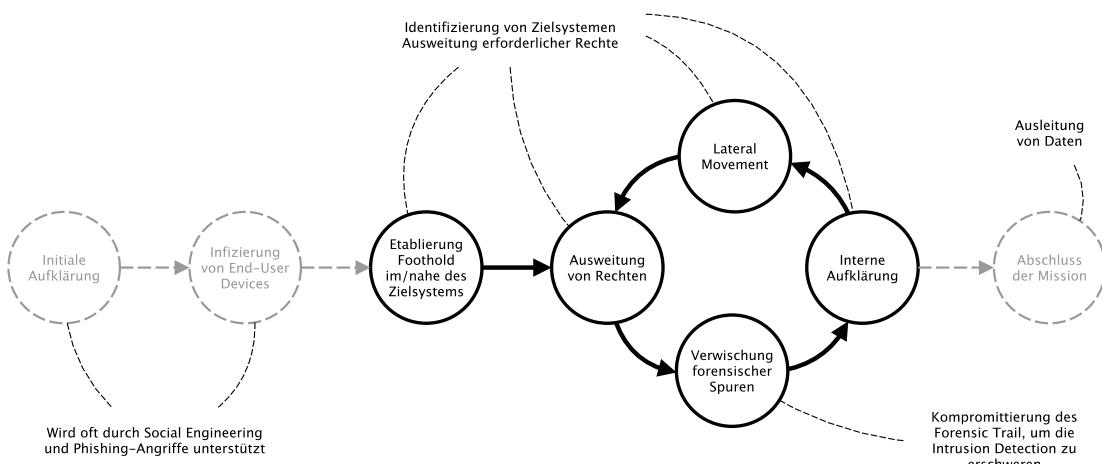


Bild 16.1 Modell des Lebenszyklus von Cyber-Angriffen



Wichtige Hinweise zur Umsetzung und Anwendbarkeit

Die in diesem Kapitel erwähnten Tools dienen lediglich als repräsentative Beispiele ihrer Art und stellen keine abschließende oder umfassende Aufzählung oder Empfehlung dar. Die Auswahl der verwendeten Tools basiert nicht zwangsläufig darauf, dass es sich hierbei um die neuesten oder für jeden Anwendungsfall geeignetsten oder am häufigsten genutzten Tools handelt. Vielmehr wurden sie ausgewählt, weil sie Cloud-native Sicherheitsaspekte effizient und prägnant veranschaulichen können.

Auf der Ebene der Container-Sicherheit und der Orchestrierungs-Plattformen wurde der Fokus dabei bewusst auf Kubernetes-basierte Ökosysteme gelegt, da es sich bei Kubernetes zum Zeitpunkt der Veröffentlichung dieser Ausgabe um den De-facto-Standard der Container-Orchestrierung handelt.

Es ist wichtig zu betonen, dass aus dieser Darstellung keine direkte Empfehlung abgeleitet werden sollte, die gezeigten Anwendungen und Vorgehensweisen unverändert zu übernehmen. Es sollte dabei immer der problemspezifische Anwendungsfall berücksichtigt werden. Für umfassende und spezifische Empfehlungen und Richtlinien wird dringend geraten, anerkannte und spezialisierte Quellen wie bspw. (Martin u. a. 2021) oder Best Practices wie etwa (NSA 2022) zu konsultieren.

Für uns sind die folgenden Schritte Etablierung eines Foothold → Ausweitung von Rechten → Verwischung von Spuren → interne Aufklärung → Lateral Movement dieses Modells von primärem Interesse. Ziel eines Angreifers ist es normalerweise immer, seine Privilegien zu erweitern, um Zugriff auf ein Zielsystem zu erlangen. Dies kann durch Fehlkonfigurationen des Systems an sich oder durch Nutzung von sogenannten Exploits geschehen. Da dies in der Regel Spuren auf dem System hinterlässt, die auf eine Sicherheitslücke hinweisen können, ist der Angreifer bestrebt, die forensischen Spuren zu verfälschen. Angreifer setzen daher forensische Gegenmaßnahmen ein, um ihre Anwesenheit auf Systemen zu verbergen und Analysen zu behindern (z. B. durch das Löschen von Logdaten oder einzelnen Ereignissen wie Log-ins). Mit einem kaum erkennbaren Fußabdruck wird die interne Erkundung des Netzwerks des Zielsystems durchgeführt, um die laterale Erkundung des Zielsystems zu erreichen und weitere aussichtsreiche Systeme zu infiltrieren. Dieses „Lateral Movement“ kann ein komplexer und langwieriger Prozess sein, der Wochen dauern kann. Daher sind infiltrierte Rechner für Angreifer wertvoll und werden in der Regel so lange wie möglich genutzt, auch nach Abschluss der Mission.

Mit dieser Modellbildung der Vorgehensweise ist es aber möglich, sich Gedanken über Gegenmaßnahmen zu machen, die diesen Cyber Attack Lifecycle stören oder erschweren. Dabei denkt man in sogenannten Angriffsvektoren, die möglichen „Lateral Movements“ von Angreifern in Systemen entsprechen. Ziel ist es, diese Bewegungsmöglichkeiten für einen Angreifer so zu erschweren, dass ein erfolgreicher Angriff nur zu unverhältnismäßig hohen Kosten und Risiken durchführbar ist.

Unter einem Angriffsvektor versteht man dabei einen bestimmten Weg oder Mechanismus, den ein Angreifer nutzen kann, um in ein Computersystem oder eine Anwendung einzudringen und unautorisierte Aktivitäten durchzuführen. Angriffsvektoren können verschiedene Formen annehmen, z. B. die Ausnutzung von Schwachstellen in der Architektur von Cloud-

Diensten, unzureichende Konfiguration von Netzwerkprotokollen oder Phishing-Angriffe auf Benutzerkonten. Bild 16.2 zeigt vier beispielhafte Angriffsvektoren, die in der Regel nicht in Reinform, sondern in Kombinationen ihrer Teilschritte angewendet werden. Dabei wird davon ausgegangen, dass eine Cloud-native Anwendung in containerisierter Form auf einer Container-Orchestrierungsplattform betrieben wird, die aus mehreren physischen oder virtuellen Maschinen besteht, und dass die Container-Orchestrierungsplattform ebenso wie die physische Infrastruktur einer Log-Konsolidierung im Rahmen eines Observability-Konzepts unterliegen. Die Cloud-native Anwendungen folgen dem 12-Faktoren-Prinzip, d. h., sie werden automatisiert aus einem Repository über eine Deployment-Pipeline bereitgestellt. In diesem für Cloud-native Anwendungen typischen Setting lassen sich vier zentrale Angriffsvektoren identifizieren.

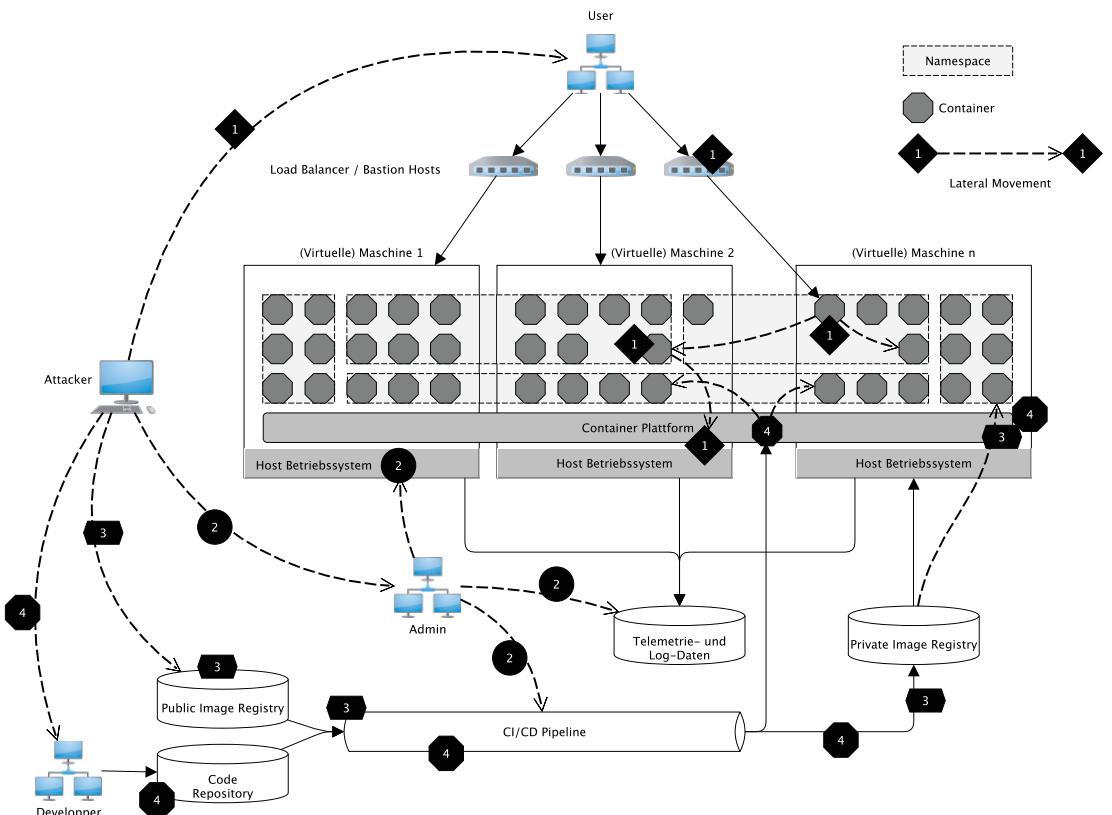


Bild 16.2 Beispielhafte Angriffsvektoren auf eine Cloud-native Anwendung (ohne Anspruch auf Vollständigkeit)

1. Beim ersten Angriffsvektor kann es einem Angreifer gelungen sein, einen normalen Benutzer zu infizieren oder sich als normaler Benutzer auszugeben. Dieser Personenkreis hat in der Regel sehr eingeschränkte Rechte. Mittels sogenannter Exploits, die über die öffentliche Schnittstelle ausgenutzt werden können, kann es einem Angreifer unter Umständen aber sogar gelingen, einen Foothold in einem Container zu erlangen. Ausgehend von diesem Container können weitere Container im gemeinsamen Namespace oder sogar

fremde Namespaces der Containerplattform infiltriert werden. Dies kann insbesondere auf Plattformen problematisch werden, die im Multi-Tenancy-Betrieb genutzt werden. Gelingt dem Angreifer ein sogenannter Container Breakout auf einem dieser Container, kann er sogar Zugriff auf eine oder mehrere physische oder virtuelle Maschinen der Infrastruktur erlangen.

2. Beim zweiten Angriffsvektor könnte es einem Angreifer beispielsweise durch Social Engineering oder Phishing-Angriffe gelingen, die Kontrolle über den Computer eines Administrators zu erlangen und sich einen wertvollen Vorsprung auf einem System in der Nähe des Ziels zu verschaffen. Administratoren haben in der Regel sehr weitreichende Rechte. Über diesen Rechner wäre dann eine laterale Bewegung auf nahezu jeden Rechner in der Infrastruktur möglich. Ein solcher Angriff könnte die Container-Plattform, das Deployment-System sowie das Observability-System selbst kompromittieren.
3. Der dritte Angriffsvektor nutzt Public Image Registries aus und greift indirekt über das meist hochautomatisierte Deployment-System an. Gelingt es einem Angreifer beispielsweise, ein Rootkit oder ähnliche Malware bzw. ausnutzbare Exploits in einem weit verbreiteten Image (z. B. einem Python-Basisimage) oder einer Bibliothek (z. B. einer Logging-Bibliothek) zu platzieren, so verbreitet sich eine solche „Infektion“ entlang aller Komponenten, die diese Abhängigkeiten nutzen. Diese Schwachstellen können so in internen Image-Registries (denen man oft vertraut, weil man die Kontrolle darüber hat) abgelegt werden und werden so im Rahmen normaler Deployments mit ausgerollt. Die so platzierten Schwachstellen können dann mit den Mitteln des Angriffsvektors 1 ausgenutzt werden.
4. Beim vierten Angriffsvektor könnte es einem Angreifer beispielsweise durch Social Engineering oder Phishing-Attacken gelingen, die Kontrolle über den Rechner eines Entwicklers zu erlangen und damit ein Code-Repository zu kompromittieren. Auf diese Weise könnten Abhängigkeiten in Projekten umgangen, Exploits in die Codebasis eingebaut oder bspw. über Daemonsets beliebige Container auf allen Infrastrukturhosts einer Kubernetes-Installation ausgerollt werden. Dies ermöglicht weitere Exploits über die Angriffsvektoren 1 und 3.

Angreifer können diese Vektoren nutzen, um auf vertrauliche Daten zuzugreifen, Schadsoftware zu installieren, Denial-of-Service-Angriffe zu starten oder andere Arten von Cyber-Angriffen durchzuführen. Cloud-native Anwendungen stellen dabei besondere Anforderungen an die Sicherheitsarchitektur, da verschiedene Komponenten der Cloud-Infrastruktur auf unterschiedlichen Ebenen miteinander interagieren und Angriffsvektoren in jeder dieser Komponenten und auf jeder dieser Ebenen liegen können. Eine effektive Sicherheitsstrategie erfordert daher eine umfassende Analyse der möglichen Angriffsvektoren und ein kontinuierliches Monitoring, um schnell auf Bedrohungen reagieren zu können.

Die Angriffsvektoren 1 und 2 können dabei durch die sogenannte Härtung virtueller Infrastrukturen (vgl. Abschnitt 16.1) deutlich erschwert werden. Die Härtung containerisierter Workloads (vgl. Abschnitt 16.2) zielt hingegen auf die Abwehr der Angriffsvektoren 3 und 4 (Schutz der Supply Chain) sowie die Minimierung von „Ausbruchsmöglichkeiten“ entlang des Angriffsvektors 1 ab. Wir orientieren uns dabei in den folgenden Abschnitten auch an den Kubernetes Hardening Guides der NSA (National Security Agency, US-Behörde) (NSA 2022).

Das Zusammenspiel aller in Abschnitt 16.1 und Abschnitt 16.2 gezeigten Maßnahmen zeigt Bild 16.3.

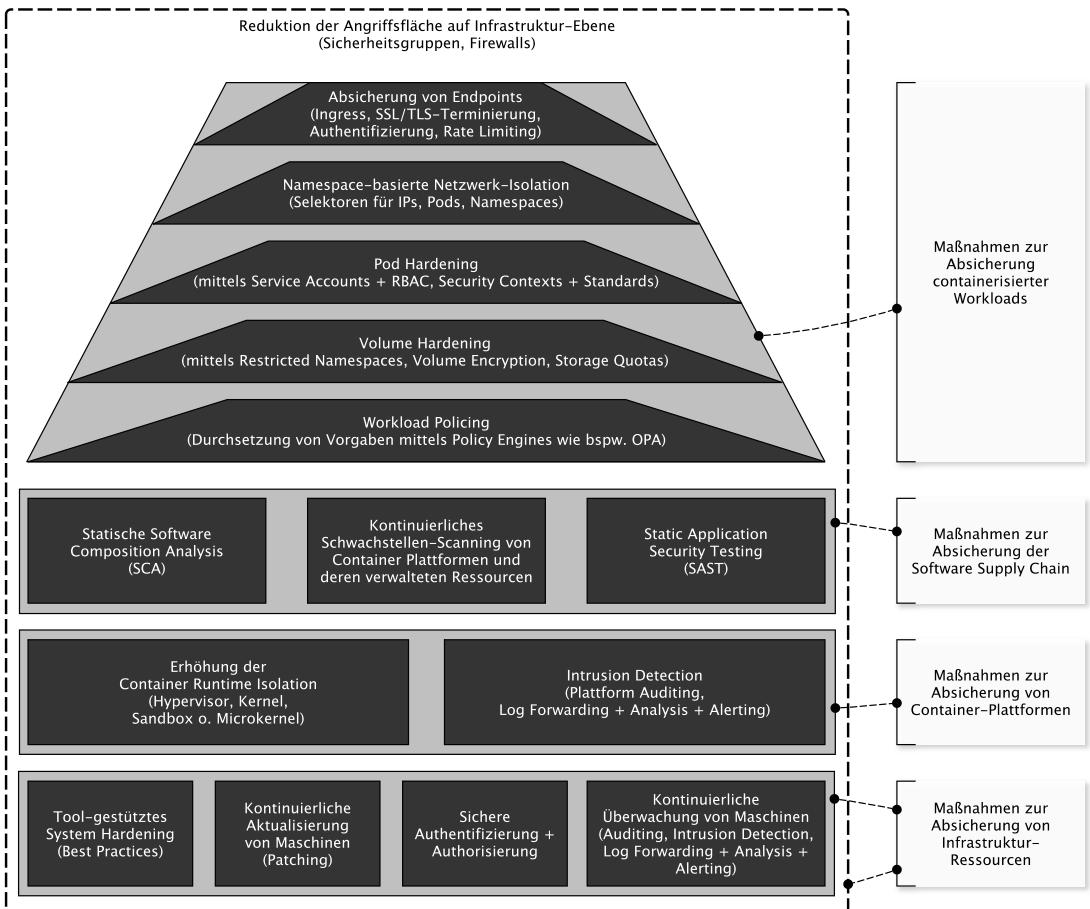


Bild 16.3 Maßnahmen und deren Ebenen zur Absicherung von Cloud-nativen Workloads

■ 16.1 Härtung (virtueller) Infrastrukturen

Es gibt eine Vielzahl von Best Practices, die der Absicherung von virtuellen Maschinen in Public-Cloud-IaaS-Infrastrukturen dienen. Es bietet sich daher an, die Härtung virtueller Maschinen systematisch und einheitlich durchzuführen. Idealerweise erfolgt dies gleich mit der automatisierten Infrastructure as Code-Provisionierung (vgl. Abschnitt 7.2). Hier sind einige der wichtigsten Punkte, die dabei zu bedenken sind:

- Es sollte ein methodisches **System Hardening** implementiert werden. In der Praxis haben sich dabei folgende Ziele herauskristallisiert: Minimierung der möglichen Angriffsmechanismen durch Reduzierung von Schwachstellen, Beschränkung der einem Angreifer zur Verfügung stehenden Werkzeuge und Minimierung der ihm zur Verfügung stehenden Rechte, um die Wahrscheinlichkeit der Entdeckung eines erfolgreichen Angriffs zu erhöhen.

- Es sollte ferner eine regelmäßige **Aktualisierung von virtuellen Maschinen** vorgesehen werden, um Sicherheitslücken möglichst schnell zu schließen und Patches zu installieren. Dieser Patching-Prozess sollte automatisiert ablaufen, um sicherzustellen, dass bekannte Lücken schnell geschlossen werden und nur kurz von Angreifern ausgenutzt werden können.
- Der Einsatz von starken **Authentifizierungs- und Autorisierungsmechanismen** soll sicherstellen, dass nur autorisierte Personen auf virtuelle Maschinen zugreifen können. Dabei muss der Datenverkehr zwischen Administratoren und virtuellen Maschinen verschlüsselt werden.
- Es sollte zudem eine **kontinuierliche Überwachung virtueller Maschinen** mittels automatisierter Überwachungstools erfolgen, um Bedrohungen, Angriffe oder Sicherheitsverletzungen schnell erkennen und darauf reagieren zu können. Dies kann analog der Konsolidierung von Telemetriedaten erfolgen (wie wir schon in *Abschnitt 13.1* gesehen haben).
- Schließlich sollte man **Sicherheitsgruppen** (bzw. analoge Konzepte wie bspw. Firewalls) konsequent nutzen, um die Netzwerkverbindungen für virtuelle Maschinen zu steuern. Dabei sollte nur der Verkehr von bekannten und zugelassenen IP-Adressen oder IP-Ranges auf definierte Ports zugelassen werden. Gegebenenfalls kann auf den virtuellen Maschinen eine Host-spezifische Firewall eingerichtet werden, die als zweite Verteidigungslinie dienen kann.

Da die meisten virtuellen Maschinen in Cloud-nativen Kontexten meist auf Linux-Systemen basieren, beschränken wir uns aus Gründen der Praktikabilität auf diese und erläutern die konkreten Sicherungsmechanismen am Beispiel der Ubuntu-Distribution. Die Prinzipien sind aber natürlich auf andere Betriebssysteme oder Distributionen vom Grundsatz übertragbar.

16.1.1 Tool-gestütztes System Hardening

Unter Härt(en) (engl. Hardening) versteht man die Erhöhung der Sicherheit eines Systems, indem nur dedizierte Software verwendet wird, die für den Betrieb des Systems notwendig ist. Das System soll dadurch besser vor Angriffen geschützt werden, indem es eine kleinere Angriffsfläche bietet. Das National Institute of Standards and Technology definiert System Hardening beispielsweise wie folgt: „Ein Prozess, der dazu dient, eine Angriffsmöglichkeit zu eliminieren, indem Schwachstellen behoben und unnötige Dienste abgeschaltet werden.“

Grundsätzlich empfiehlt sich der Einsatz von Linux Hardening-Lösungen wie bspw. Lynis, die dabei helfen können, das System weiter und anhand von Best Practices zu härten. Lynis ist hier wie immer nur als Referenztool zu verstehen, welches die prinzipielle Wirkungsweise solcher Tools veranschaulichen soll.

Listing 16.1 Installation der Linux Hardening-Lösung Lynis

```
sudo apt-get install lynis
sudo lynis audit system
```

Derartige Tools liefern einen Überblick, welche Maßnahmen gegebenenfalls zu ergreifen sind. Lynis gibt bspw. Empfehlungen und einen Hardening-Index an, der einen Eindruck liefert, welchen Härtungsgrad ein System hat und wie dieser verbessert werden kann.

Listing 16.2 Beispiel einer Lynis-Auswertung (nur Auszüge)

```
-[ Lynis 3.0.7 Results ]-

Great, no warnings

Suggestions (51):
-----
* Install libpam-tmpdir to set $TMP and $TMPDIR for PAM sessions [DEB-0280]
  https://cisofy.com/lynis/controls/DEB-0280/

* Install apt-listbugs to display a list of critical bugs prior to each APT
  installation. [DEB-0810]
  https://cisofy.com/lynis/controls/DEB-0810/

* Install fail2ban to automatically ban hosts that commit multiple authentication
  errors. [DEB-0880]
  https://cisofy.com/lynis/controls/DEB-0880/

[...]

Lynis security scan details:

Hardening index : 62 [#####
] Tests performed : 255 Plugins enabled : 1

Components:
- Firewall [V]
- Malware scanner [X]

Scan mode:
Normal [V] Forensics [ ] Integration [ ] Pentest [ ]

Lynis modules:
- Compliance status [?]
- Security audit [V]
- Vulnerability scan [V]

Files:
- Test and debug information : /var/log/lynis.log
- Report data : /var/log/lynis-report.dat
```

Dieses Vorgehen funktioniert sehr gut für Infrastruktur-Komponenten, auf denen bekannte und standardisierte Softwarepakete installiert und betrieben werden. Selbst entwickelte Software sollte hingegen lieber in Form von Containern bereitgestellt werden, und deren Images im Rahmen von automatisierten Deployment-Pipelines sollten geprüft und gehärtet werden (vgl. auch Abschnitt 6.1 in Teil II sowie Abschnitt 16.2 in diesem Kapitel).

16.1.2 Kontinuierliche Aktualisierung von virtuellen Maschinen

Die kontinuierliche Aktualisierung von virtuellen Maschinen dient aus dem Blickwinkel der IT-Sicherheit dazu, bekannte Exploits und Schwachstellen in Softwarepaketen schnellstmöglich zu schließen, um Angreifern wenig Zeit zu lassen, Schwachstellen ausnutzen zu können. Dies kann grundsätzlich auf zwei Arten erfolgen.

1. Man kann bspw. im Rahmen einer IaC-Provisionierung (vgl. *Abschnitt 7.2*) neue Maschinen-Images immer dann bauen, wenn neue Sicherheitspatches ausgespielt werden. Alternativ kann man Images auch periodisch neu bauen, indem man bspw. Cron-gesteuerte Deployment-Pipelines (vgl. *Abschnitt 6.1*) dafür nutzt. Dies hat den Vorteil, dass dies dem Konzept von Immutable Architectures (vgl. *Abschnitt 7.2.1*) eher entspricht und die Konfigurationsdrift minimiert.
2. Da aber fast alle Linux-Distributionen auf Paketverwaltungssysteme setzen und viele auch automatische Updatemöglichkeiten bieten, kann man natürlich auch diese nutzen. Dies entspricht streng genommen nicht mehr vollständig dem Prinzip von Immutable Architectures (vgl. *Abschnitt 7.2.1*), ist aber relativ leicht zu realisieren und ist nach Erfahrung des Autors Systemadmins meist näher.

Die theoretisch denkbare dritte Variante der manuellen Aktualisierung wird hier als weder praktikabel noch erstrebenswert verworfen und daher nicht weiter behandelt.

Wie einfach der zweite Ansatz auf Linux-Systemen umgesetzt werden kann, soll im Folgenden am Beispiel des sogenannten Unattended Upgrading (unüberwachte Aktualisierung) gezeigt werden. Unattended Upgrades sind ein Mechanismus zur Automatisierung von Systemaktualisierungen auf Ubuntu-Linux-Systemen. Um Unattended Upgrades einzurichten, muss ein entsprechendes Paket installiert und konfiguriert werden. Idealerweise erfolgen die Installation und Konfiguration automatisiert im Rahmen des IaC Provisioning (vgl. *Abschnitt 7.2*).

Listing 16.3 Installation des Unattended Upgradings

```
sudo apt-get install unattended-upgrades
```

Welche Pakete automatisch aktualisiert werden sollen, kann man in entsprechenden Konfigurationsdateien festlegen. Listing 16.4 zeigt bspw., wie auf einem Ubuntu-System alle verfügbaren Updates berücksichtigt werden.

Listing 16.4 Konfigurationsdatei /etc/apt/apt.conf.d/50unattended-upgrades

```
Unattended-Upgrade::Allowed-Origins {
    "${distro_id}:${distro_codename}";
    "${distro_id}:${distro_codename}-security";
    "${distro_id}ESMApps:${distro_codename}-apps-security";
    "${distro_id}ESM:${distro_codename}-infra-security";
    "${distro_id}:${distro_codename}-updates";
};
```

Weitere Optionen, die insbesondere die Aktualisierungsfrequenz betreffen, lassen sich ebenso definieren. Listing 16.5 zeigt bspw., dass Paketlisten einmal täglich zu aktualisieren,

herunterladbare Updates automatisch herunterzuladen, ungenutzte Pakete nach sieben Tagen zu entfernen und automatisch Updates durchzuführen sind.

Listing 16.5 Konfigurationsdatei /etc/apt/apt.conf.d/20auto-upgrades

```
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Download-Upgradeable-Packages "1";
APT::Periodic::AutocleanInterval "7";
APT::Periodic::Unattended-Upgrade "1";
```

Der hier gezeigte Ansatz sollte für Infrastruktur genutzt werden, um bspw. den Linux-Kernel, Firewalls, die Container Runtime Environment usw. automatisiert aktuell zu halten.

16.1.3 Sichere Authentifizierung mittels SSH

Um den Fernzugriff auf virtuelle Maschinen zu ermöglichen, empfiehlt sich die Verwendung von SSH. SSH steht für „Secure Shell“ und ist ein Netzwerkprotokoll, das für sichere Netzwerkverbindungen verwendet wird, insbesondere für den sicheren Fernzugriff auf Systeme. SSH ermöglicht die sichere und verschlüsselte Übertragung von Daten über ein unsicheres Netzwerk wie das Internet. Mit SSH können sich Administratoren und Benutzer sicher auf entfernten Systemen einloggen und Befehle ausführen. SSH kann auch zur Übertragung von Dateien über eine sichere Verbindung (SFTP) und zur Verwaltung öffentlicher Schlüssel (SSH-Keys) sowie zum verschlüsselten Tunneling von Verbindungen verwendet werden.

SSH verwendet hierzu ein asymmetrisches Verschlüsselungsverfahren, um die Sicherheit der übertragenen Daten zu gewährleisten. Für den Verbindungsauflauf nutzt der Client einen öffentlichen und einen privaten Schlüssel. Der Server hält ebenfalls eine öffentliche und eine private Schlüsselkomponente vor. Die öffentlichen Schlüssel beider Parteien werden ausgetauscht, und der Client verwendet den öffentlichen Schlüssel des Servers, um eine verschlüsselte Verbindung aufzubauen. Die über diese Verbindung übertragenen Daten sind verschlüsselt und können nur von der Gegenstelle entschlüsselt werden, die über den privaten Schlüssel verfügt. Dies gilt entsprechend den NSA-Empfehlungen (NSA 2022) als ein verlässlicher Schutz.

Zur Einrichtung von SSH muss ein entsprechendes Paket installiert und konfiguriert werden. Idealerweise erfolgen auch hier die Installation und Konfiguration des SSH-Servers im Rahmen des IaC-Provisioning der virtuellen Maschine automatisiert (vgl. *Abschnitt 7.2*).

Listing 16.6 Installation der Secure Shell

```
sudo apt-get install openssh-server
```

Um den SSH-Zugriff von anderen Systemen aus zu ermöglichen, muss sichergestellt werden, dass der SSH-Port in der Security Group oder Firewall geöffnet ist. Standardmäßig verwendet SSH den Port 22.

Lynis empfiehlt im Übrigen, folgende Härtungsmaßnahmen in der SSH-Konfiguration vorzunehmen (siehe Listing 16.7). Dies umfasst insbesondere die Reduktion gleichzeitiger

SSH-Sessions, die Reduktion der maximalen Anzahl zulässiger Autorisierungsversuche und die Deaktivierung des Forwardings. Die Maßnahmen dienen dazu, mittels Skripten automatisierte Log-in-Rateversuche zu erschweren und im Worst Case das Lateral Movement von einer kompromittierten Maschine zu verhindern.

Listing 16.7 SSH-Konfiguration /etc/ssh/sshd_config (nur Anpassung zu den Server Default-Settings)

```
MaxSessions 2
ClientAliveCountMax 2
PubkeyAuthentication yes
MaxAuthTries 3
AllowAgentForwarding no
AllowTcpForwarding no
X11Forwarding no
TCPKeepAlive no
```

Um einen SSH-Schlüssel auf einer Maschine (Client) zu erstellen und auf dem Server zur Authentifizierung zu hinterlegen, erstellt man einen SSH-Schlüssel unter Nutzung des Befehls `ssh-keygen` im Terminal. Üblicherweise wird der öffentliche Schlüssel im Verzeichnis `~/.ssh/id_rsa.pub` gespeichert und der private Schlüssel im Verzeichnis `~/.ssh/id_rsa`. Der private Schlüssel verbleibt auf dem Rechner des Nutzers (Administrators). Der öffentliche Schlüssel kann auf dem Server mit folgendem Befehl hinterlegt werden und dient dazu, einen Nutzer über seinen privaten Schlüssel zu authentifizieren.

Listing 16.8 Hinterlegen eines SSH Keys auf einem Server

```
ssh-copy-id user_name@ip_address
```

Bei diesem Vorgang muss sich der Nutzer nur einmalig mit seinem Passwort authentifizieren. Der öffentliche Schlüssel wird dabei auf dem Server in der Datei `/home/user_name/.ssh/authorized_keys` angehängt und kann ab sofort zur Authentifizierung mittels des privaten SSH-Schlüssels genutzt werden, ohne ein Passwort eingeben zu müssen.

Listing 16.9 Einloggen auf einer VM von einem Client mittels SSH

```
ssh user_name@ip_address
```

Durch die Verwendung von SSH-Schlüsseln kann so der Zugriff auf entfernte virtuelle Maschinen sowohl sicherer als auch bequemer gestaltet werden, da auf die Eingabe von Passwörtern verzichtet werden kann. Erfahrungsgemäß ist diese Art des Log-ins für erstmalige SSH-Nutzer allerdings anfangs etwas gewöhnungsbedürftig und wird auch nicht als sicherer empfunden, da man ja das „geliebte“ Passwort nicht nutzt. Faktisch ist die Authentifizierung mittels eines privaten SSH-Schlüssels jedoch sicherer und vor allem als Best Practice anzusehen, da ein Angreifer erst einmal (physischen) Zugriff auf den Rechner eines Nutzers erlangen muss. Der Aufwand für Angreifer erhöht sich damit signifikant und lässt viele Angriffsvektoren, die auf Passwortraten oder Phishing basieren, unwirtschaftlich werden.

16.1.4 Kontinuierliche Überwachung virtueller Maschinen

Die kontinuierliche Überwachung virtueller Maschinen dient insbesondere dazu, den inneren Kreis des Cyber-Attack-Lifecycles (vgl. Bild 16.1) zu erschweren, indem mittels

- verhaltensbasierter Intrusion Detection (Auditing),
- signaturbasierter Intrusion Detection (Rootkit Detection)
- und mittels Log Forwarding

kontinuierlich das System hinsichtlich ungewöhnlicher Zugriffe, Veränderungen überprüft wird und diese Ergebnisse kontinuierlich an ein externes Analysesystem weitergereicht werden, um es Angreifern zu erschweren, unerkannt sogenannte Footholds in Systemen über längere Zeiträume darin aufrechterhalten zu können. Man geht also davon aus, dass der äußere Perimeter auf irgendeine Art durchbrochen wurde, und versucht, diese „Breaches“ nachträglich zu erkennen.

Das Ziel von Intrusion Detection ist es, Bedrohungen für die IT-Sicherheit frühzeitig zu erkennen und geeignete Maßnahmen zu ergreifen, um Schäden zu minimieren. Host-basierte Intrusion Detection wird auf einzelnen Systemen installiert und überwacht das System auf ungewöhnliche Aktivitäten, während netzwerk basierte Intrusion Detection den Datenverkehr auf Netzwerkebene analysiert. Wir konzentrieren uns im Weiteren auf die Host-basierte Intrusion Detection. Idealerweise erfolgen die Installation und Konfiguration von Intrusion Detection und Log Forwarding-Lösungen automatisiert im Rahmen eines IaC Provisioning (vgl. Abschnitt 7.2).

16.1.4.1 Verhaltensbasierte Intrusion Detection mittels Auditing

Verhaltensbasierte Intrusion Detection analysiert das Verhalten von Benutzern und Systemen, um unbekannte Angriffe zu erkennen. Hierbei werden beispielsweise Anomalien im Netzwerkverkehr, ungewöhnliche Benutzeraktivitäten oder unerwartete Systemereignisse überwacht. Hierzu werden oft sogenannte Auditsysteme eingesetzt.

Ein Auditsystem ist ein System, das dazu dient, Informationen über die Aktivitäten auf einem Computersystem zu sammeln, zu speichern und zu überwachen. Es erfasst Informationen darüber, wer auf das System zugreift, wann und welche Aktionen ausgeführt werden. Dadurch kann es helfen, die Sicherheit des Systems zu erhöhen, indem es unautorisierte Zugriffe oder Aktivitäten aufspürt und Benutzeraktivitäten protokolliert, um mögliche Sicherheitsverletzungen zu untersuchen.

Ein Auditsystem kann in der Regel so konfiguriert werden, dass es bestimmte Ereignisse oder Aktivitäten aufzeichnet, wie z. B. Änderungen an Systemdateien, Netzwerkverbindungen oder Anwendungen, die ausgeführt werden. Die aufgezeichneten Daten können dann verwendet werden, um Probleme zu diagnostizieren oder zu beheben, Sicherheitsbedrohungen zu erkennen und den Compliance-Status des Systems zu überprüfen.

Um sicherzustellen, dass ein Auditsystem zuverlässige Daten liefert, muss es in der Regel so konfiguriert werden, dass es nur aufzeichnet, was notwendig ist, und dass die aufgezeichneten Daten vor Manipulation oder Löschung geschützt sind. Darüber hinaus sollten die aufgezeichneten Daten regelmäßig überprüft werden, um potenzielle Sicherheitsbedrohungen zu erkennen und zu beheben.

Der Linux Audit Daemon (auditd) ist eine Userspace-Komponente des Linux-Auditing-Systems, die für das Sammeln und Schreiben von sogenannten Audit-Protokolldateien verantwortlich ist. Diese Protokolle können bei der Überwachung von Sicherheitsverletzungen oder Sicherheitsvorfällen von entscheidender Bedeutung sein. Systemadministratoren können diese Protokolle verwenden, um ungewöhnliche Aktivitäten zu erkennen und zurückzuverfolgen, wie das System kompromittiert wurde. Diese nachträgliche Analyse kann helfen, auf den Vorfall zu reagieren und die Sicherheit zu erhöhen, um ähnliche Vorfälle in Zukunft zu vermeiden.

Listing 16.10 Installation von auditd

```
sudo apt-get install auditd
sudo systemctl enable auditd
```

Systemereignisse wie Log-in-Vorgänge und neuralgische Vorgänge, die bspw. Root-Rechte erfordern, werden dann in Log-Dateien (bei auditd in der Datei `/var/log/audit/audit.log`) protokolliert. Diese Audit-Logs sollte einem Log-Forwarding (vgl. Abschnitt 16.1.4.3) unterworfen werden, um damit die Verwischung forensischer Spuren zu erschweren bzw. unverfälschte Logs außerhalb gegebenenfalls kompromittierter Systeme analysieren zu können.

16.1.4.2 Signaturbasierte Intrusion Detection

Die signaturbasierte Intrusion Detection verwendet hingegen – anders als die verhaltensbasierte Intrusion Detection – vordefinierte Regeln oder Muster, um bekannte Angriffe zu identifizieren.

Rootkits sind Softwaretools, die nach einem Einbruch in ein Softwaresystem auf dem kompromittierten System installiert werden, um zukünftige Anmeldevorgänge (Log-ins) des Eindringlings zu verbergen und Prozesse und Dateien zu verstecken. Um sich gegen Rootkits verteidigen zu können, muss daher erkannt werden, wenn Systeme verändert wurden. Dies kann ein Hinweis darauf sein, dass ein Rootkit installiert wurde oder andere Systemmanipulationen stattgefunden haben.

rkhunter (Rootkit Hunter) ist solch ein Tool, das nach Rootkits, Backdoors und möglichen lokalen Exploits sucht. rkhunter ist hier als Typvertreter für eine ganze Klasse von Werkzeugen zur Detektion von Systemmanipulationen zu verstehen. Die Wirkungs- und Arbeitsweise aller dieser Tools ist ähnlich und folgt dem nachfolgend gezeigten Prinzip.

rkhunter vergleicht vorhandene Dateien anhand von MD5-Hashes mit kompromittierten Dateien, sucht nach von Rootkits angelegten Ordner, falschen Dateirechten, versteckten Dateien, verdächtigen Zeichenketten in Kernelmodulen und führt eine Reihe weiterer Tests durch.

Listing 16.11 Installation des Rootkit-Detektors rkhunter sowie Aufbau der Hash-Datenbank

```
sudo apt-get install rkhunter
sudo rkhunter --propupd
```

Bevor rkhunter verwendet werden kann, muss eine Hash-Datenbank von Systemdateien erstellt werden. Mit dem Parameter `--propupd` werden Hash-Werte für Dateien aus dem aktuellen Systemzustand erzeugt. Ein System kann dann mittels

```
sudo rkhunter -c -sk
```

geprüft werden. Bei einem solchen Scan werden dann die gefundenen Werte mit den Hashwerten der Datenbank verglichen. Alle nicht identischen Hashwerte zeigen Änderungen einer Datei an. Zu beachten ist, dass dies nur funktioniert, wenn das System zum Zeitpunkt des Updates mit --propupd noch nicht kompromittiert war. Es empfiehlt sich daher, diesen Schritt am Ende eines automatisierten Deployments durchzuführen, um sicherzustellen, dass die Hashwerte auch dem letzten beabsichtigten Systemzustand entsprechen. Ferner sollte nach jeder Aktualisierung ebenfalls die Hash-Datenbank aktualisiert werden, andernfalls sind Fehlalarme zu erwarten.

Die in Listing 16.12 exemplarisch gezeigten Log-Dateien sollten ebenfalls wie die Audit-Logs einem Log-Forwarding unterworfen werden (vgl. Abschnitt 16.1.4.3).

Listing 16.12 Gekürzter Beispiel-Log eines rkhunter-Scans /var/log/rkhunter.log

```
[ Rootkit Hunter version 1.4.6 ]

Checking system commands...

Performing 'strings' command checks
  Checking 'strings' command [ OK ]

Performing 'shared libraries' checks
  Checking for preloading variables [ None found ]
  Checking for preloaded libraries [ None found ]
  Checking LD_LIBRARY_PATH variable [ Not found ]

Performing file properties checks
  Checking for prerequisites [ OK ]
  /usr/sbin/adduser [ OK ]
  /usr/sbin/chroot [ OK ]
  /usr/sbin/cron [ OK ]
  /usr/sbin/depmod [ OK ]
...
Checking for rootkits...

Performing check of known rootkit files and directories
  55808 Trojan - Variant A [ Not found ]
  ADM Worm [ Not found ]
  AjaKit Rootkit [ Not found ]
  Adore Rootkit [ Not found ]
  BeastKit Rootkit [ Not found ]
  beX2 Rootkit [ Not found ]
  BOBKit Rootkit [ Not found ]
...
System checks summary
=====

Rootkit checks...
  Rootkits checked : 498
  Possible rootkits: 0
```

```

Applications checks...
  All checks skipped

The system checks took: 48 seconds

All results have been written to the log file: /var/log/rkhunter.log

No warnings were found while checking the system.

```

16.1.4.3 Log Forwarding

Die in Abschnitt 16.1.4.1 und Abschnitt 16.1.4.2 genannten Systeme zur Intrusion Detection protokollieren ihre Ergebnisse normalerweise in Protokollen (gegebenenfalls auch in externen Zeitreihendatenbanken). Zur Auswertung und Automatisierung des Alertings bietet es sich daher an, diese Logdateien einer Log-Konsolidierung zu unterwerfen. Dies folgt den bereits in gezeigten Prinzipien und dient letztlich der Observability eines Systems.

Listing 16.13 zeigt am Beispiel von Filebeat, wie einfach das Forwarding von Log-Dateien an eine zentrale ElasticSearch-Datenbank zum Zwecke der Log-Konsolidierung auf einem Host installiert werden kann. FileBeat ist dabei wie alle Tools in diesem Teil als Typvertreter zu verstehen. Es gibt andere Log-Forwarding Tools, die ähnlich funktionieren und gleichermaßen geeignet sind.

Listing 16.13 Installation von Filebeat

```

sudo apt-get install filebeat
sudo systemctl enable filebeat
sudo systemctl start filebeat

```

Die Konfiguration von Filebeat erfolgt wie in Listing 16.14 gezeigt. Letztlich müssen nur die Adresse der zentralen Elasticsearch-Datenbank (inklusive Access Credentials) und die Verzeichnisse und Log-Dateien, die dem Log Forwarding unterworfen werden sollen, angegeben werden. Anschließend werden die Logs mit jeder Aktualisierung automatisch vom Host an einen zentralen Logstore weitergeleitet und damit auch potenziellen Manipulationen von Angreifern entzogen.

Listing 16.14 Filebeat-Konfiguration /etc/filebeat/filebeat.yml (Auszug wesentlicher Parameter)

```

# ----- Elasticsearch Output -----
output.elasticsearch:
  # Array of hosts to connect to.
  hosts: ["name.of.central.logstore:9200"] # Muss angepasst werden

  # Protocol - either `http` (default) or `https`.
  #protocol: "https"

  # Authentication credentials - either API key or username/password.
  #api_key: "id:api_key"
  #username: "elastic"

```

```

#password: "changeme"

...
# ===== Filebeat inputs =====
filebeat.inputs:

# filestream is an input for collecting log messages from files.
- type: filestream
  enabled: true
  id: my-filestream-id # Unique ID among all inputs, an ID is required.

# Paths that should be crawled and fetched. Glob based paths.
paths:
  - /var/log/*.log
  - /var/log/**/*.*.log
# Diese Verzeichnisliste kann erweitert werden

```

FileBeat oder vglb. Tools ermöglichen ein Log-Forwarding auf Infrastrukturebene. Es bietet sich an, ein solches Log-Forwarding ebenfalls auf Ebene der Container-Plattform vorzunehmen. Hierzu gibt es Helmcharts, die Filebeat ähnlich einfach in Kubernetes deployen, um alle Container-Logs einer Log-Konsolidierung zu unterwerfen. Dabei werden Container-Logs automatisch um hilfreiche Metadaten von Kubernetes angereichert, wie bspw. Informationen zum Namespace, Host, Pod und Labels, um Logs von Interesse effizient und pragmatisch filtern zu können.

16.1.5 Einsatz von Sicherheitsgruppen und Firewalls

Grundsätzlich sollte darauf geachtet werden, dass nur notwendige Dienste auf einem System laufen. Jeder unnötig laufende Dienst öffnet Ports auf einem System, die grundsätzlich die Angriffsfläche für Angriffsvektor 1 erhöhen.

Cloud-Anbieter und Cloud-Infrastrukturen stellen hierfür sogenannte Sicherheitsgruppen oder ähnlich benannte Konzepte (Firewall, Security Group ...) zur Verfügung. Eine Sicherheitsgruppe (Security Group) ist eine virtuelle Firewall, die den Netzwerkverkehr für virtuelle Maschinen kontrolliert. Eine Sicherheitsgruppe erlaubt oder blockiert den ein- und ausgehenden Datenverkehr auf Basis konfigurierter Regeln. Sicherheitsgruppen können für eine oder mehrere Instanzen virtueller Maschinen gelten und Regeln enthalten, die den Zugriff auf die Instanzen steuern. Beispielsweise kann man Regeln erstellen, die eingehenden Datenverkehr über Port 80 (HTTP) von allen IP-Adressen zulassen, um den Zugriff auf eine Webanwendung zu ermöglichen. Der Zugriff auf Port 22 (SSH) wird jedoch nur erlaubt, wenn sich der Ursprung in einer anderen (vertrauenswürdigen) Sicherheitsgruppe befindet, deren Maschinen alle keinen direkten Zugriff aus dem Internet erlauben.

Sicherheitsgruppen bieten somit eine sehr feingranulare Kontrolle über den Netzwerkverkehr und ermöglichen es, den Zugriff auf Protokolle, Ports und IP-Adressbereiche auf Infrastrukturebene zu beschränken, um unberechtigte Zugriffe zu verhindern oder zu erschweren.

Dadurch kann die Kontrolle des Netzwerkverkehrs außerhalb der virtuellen Maschinen zentral definierbar (und anpassbar) erfolgen und muss nicht host-basiert und damit verteilt erfolgen. Ergänzend kann dennoch der Einsatz von host-basierten Firewalls als zweite Verteidigungslinie erwogen werden. Dies ist insbesondere dann zu empfehlen, wenn man keine oder wenig Gestaltungsspielraum über die Sicherheitsgruppen der IaaS-Infrastruktur hat, wie das gegebenenfalls beim Einsatz von IaaS-Lösungen in einem Private Cloud-Kontext oder einer reinen Virtualisierungslösung wie vSphere der Fall sein kann.

Hierfür gibt es verschiedene Firewall-Lösungen, die für Ubuntu und weitere Linux-Distributionen üblicherweise empfohlen werden.

- Uncomplicated Firewall (UFW) – UFW ist eine einfache Firewall-Option, die in Ubuntu bereits vorinstalliert ist. Sie kann über die Kommandozeile konfiguriert werden und ist einfach zu verwenden.
- Firewalld – Firewalld ist eine Firewall-Option, die von Red Hat entwickelt wurde und auch in Ubuntu verfügbar ist. Sie bietet ebenfalls eine einfache Möglichkeit, komplexe Firewall-Regeln zu erstellen und zu verwalten.
- iptables – iptables ist eine leistungsfähige Firewall-Option, die jedoch komplexer zu konfigurieren ist als UFW und Firewalld. Wenn bereits Erfahrung mit iptables vorhanden ist, kann es eine gute Option sein, komplexe Probleme umzusetzen.

Listing 16.15 Installation und Aktivierung der Uncomplicated Firewall und fail2ban

```
sudo apt-get install ufw fail2ban
sudo ufw allow ssh
sudo systemctl enable ufw
sudo systemctl start ufw
```

Insbesondere der Umgang mit UFW ist (im Gegensatz zu iptables) recht intuitiv. Standardmäßig ist UFW so konfiguriert, dass alle eingehenden Verbindungen abgewiesen und alle ausgehenden Verbindungen zugelassen werden. Auf diese Weise kann per Default niemand eine Verbindung herstellen, während jede Anwendung innerhalb des Servers mit der Außenwelt kommunizieren kann.

Mit UFW kann die Kommunikation hinsichtlich spezifischer Ports oder Portbereiche und IP-Adressen sowie Subnetze oder Netzwerkschnittstellen sehr flexibel zugelassen oder abgelehnt werden, wie die Beispiele in Listing 16.16 zeigen.

Listing 16.16 Beispiele für die Einrichtung von UFW-Firewall-Regeln

```
# Erlaubt Zugriff von überall auf HTTPS
sudo ufw allow https

# Erlaubt Zugriff von überall auf TCP und UDP Port Ranges
sudo ufw allow 6000:6007/tcp
sudo ufw allow 6000:6007/udp

# Erlaubt Zugriff aus einem Subnet auf alle Ports
sudo ufw allow from 203.0.113.4
```

```
# Erlaubt Zugriff aus einem Subnet auf Port 22 (SSH)
sudo ufw allow from 203.0.113.0/24 to any port 22

# Erlaubt Zugriff auf Port 3306 (MySQL) nur über die Netzwerkschnittstelle eth1
sudo ufw allow in on eth1 to any port 3306

# Ablehnen von Verbindungen von einer spezifischen IP-Adresse
# Vielleicht Quelle eines DoS-Angriffs
sudo ufw deny from 203.0.113.4
```

Diese Art von Regelungen gelten dann für den Host, auf dem diese eingerichtet wurden. Mittels der eingangs erwähnten Sicherheitsgruppen können solche Regeln auch außerhalb des Hosts durch die Netzwerkinfrastruktur durchgesetzt werden, was die Konfiguration der Hosts natürlich vereinfacht.

Idealerweise erfolgen die Installation und Konfiguration lokaler Firewalls wie UFW automatisiert im Rahmen des IaC Provisioning (vgl. *Abschnitt 7.2*). Auch Sicherheitsgruppen in IaaS-Infrastrukturen lassen sich im Rahmen eines IaC Provisioning automatisiert ausbringen.

■ 16.2 Härtung containerisierter Workloads

Für eine umfassende Darstellung zur Härtung containerisierter Workloads insbesondere im Kubernetes-Umfeld wird grundsätzlich auf weiterführende Literatur wie bspw. (Martin u. a. 2021) oder den Kubernetes Hardening Guide der NSA (NSA 2022) verwiesen. Die folgenden Darstellungen folgen jedoch einem ähnlichen Ansatz wie (Martin u. a. 2021) und berücksichtigen die Empfehlungen von (NSA 2022) bei der Analyse und Ableitung der „wesentlichen“ Maßnahmen zur Erhöhung und Härtung containerisierter Workloads. Der Kubernetes Hardening Guide (NSA 2022) benennt bspw. Strategien, um häufige Fehlkonfigurationen zu vermeiden und empfohlene Härtungsmaßnahmen beim Einsatz von Kubernetes umzusetzen. Dies umfasst unter anderem:

- Scannen von Containern und Pods auf Schwachstellen oder Fehlkonfigurationen.
- Ausführung von Containern und Pods mit den geringstmöglichen Rechten.
- Netzwerk trennung (Netzwerkisolation), um den Schaden einer Kompromittierung zu begrenzen und Lateral Movement zu erschweren.
- Nutzung von Firewalls bzw. Sicherheitsgruppen, um nicht benötigte Netzwerkverbindungen zu begrenzen.
- Nutzung von Verschlüsselung, um Vertraulichkeit zu schützen.
- Nutzung einer starken Authentifizierung und Autorisierung, um den Zugriff von Benutzern und Administratoren zu beschränken und die Angriffssoberfläche zu begrenzen bzw. zu reduzieren.

- Erfassung und Überwachung von Audit-Protokollen, damit Administratoren automatisiert über potenziell böswillige Aktivitäten proaktiv gewarnt werden können.
- Regelmäßig und automatisierte Überprüfung aller Kubernetes-Einstellungen mittels Schwachstellen-Scans, um sicherzustellen, dass Risiken angemessen berücksichtigt und Sicherheits-Patches angewendet werden.

Das dabei betrachtete Threat Model ergibt sich wie auch im vorherigen Kapitel aus Bild 16.2. Und auch in diesem Kapitel ist es unser Bestreben, den Cyber Attack Lifecycle eines Angreifers aus Bild 16.1 möglichst an vielen Stellen zu stören und zu erschweren.

Wir werden dabei dem Verlauf des Angriffsvektors 1 aus Bild 16.2 folgen und uns fragen, wie wir „von außen“ kommend Layer für Layer härten können, um Penetrationen und Ausbrüche zu erschweren.

- Abschnitt 16.2.1 befasst sich hierzu mit der Frage, wie man den Zugang zum System auf Ebene des Ingress absichern kann.
- Abschnitt 16.2.2 erläutert, wie man die Isolation von Workloads mittels Namespaces erreichen und den Datenverkehr zwischen von Ingress erreichbaren Pods so einschränken kann, dass ein Lateral Movement so erschwert wird, damit gekaperte Pods nicht beliebigen Zugang zu anderen Ressourcen und Workloads des Clusters erhalten.
- Abschnitt 16.2.3 geht darauf ein, wie man Kubernetes Workloads und Pods konfigurieren sollte, damit ein Ausbrauch aus der Isolationsumgebung erschwert wird.
- Ergänzend betrachten wir in Abschnitt 16.2.4, Möglichkeiten besser isolierende Container Runtime Environments zu nutzen, die bspw. nicht nur auf „relativ schwachen“ Isolationsmöglichkeiten des Betriebssystem-Kernels beruhen, sondern auch stärkere Isolationsmöglichkeiten der Systemvirtualisierung nutzen.
- Abschnitt 16.2.5 betrachtet Möglichkeiten zur Isolation von Storage, um Data Breaches und die Ausleitung von Daten zu erschweren.
- Abschnitt 16.2.6 betrachtet „Policies“ als Mechanismen, die verwendet werden, um das Verhalten und die Einschränkungen von Ressourcen in einem Kubernetes-Cluster zu steuern. Sie ermöglichen es Administratoren, Regeln und Richtlinien festzulegen, um die Sicherheit des Clusters zu gewährleisten.
- Abschnitt 16.2.7 zeigt, dass eine umfassende Intrusion Detection-Strategie in Kubernetes in der Regel eine Kombination mehrerer Ansätze umfasst, die spezialisierte Tools und Plattformen erfordern, die oft nicht zum Standardumfang von Kubernetes gehören.
- Abschnitt 16.2.8 widmet sich der Sicherung der Software Supply Chain. Diese Maßnahmen haben zum Ziel, Schwachstellen in Abhängigkeiten oder Konfigurationen systematisch zu erkennen und kontinuierlich zu überprüfen.

16.2.1 Absicherung von Public Endpoints mittels Ingresses

Ein Kubernetes Ingress ist (wie bereits in *Abschnitt 9.3.6* erläutert) eine Kubernetes-Ressource, die den Zugriff auf Dienste innerhalb eines Kubernetes-Clusters verwaltet. Sie fungiert als Eingangstor (engl. „Ingress“) für den eingehenden Datenverkehr und ermöglicht es, den Verkehr auf verschiedene Dienste basierend auf den Regeln und Konfigurationen zu verteilen.

(vgl. auch *Kapitel 9.3.6*). Eine Ingress-Ressource definiert hierzu eine Reihe von Regeln, die den Datenverkehr auf Basis von Hostnamen, Pfaden oder anderen Kriterien an verschiedene Dienste innerhalb des Clusters weiterleiten. Sie ermöglicht die externe Erreichbarkeit von Diensten und bietet eine zentrale Anlaufstelle für den eingehenden Datenverkehr und damit natürlich auch für Härtungsmaßnahmen.

Die Ingress-Ressource selbst ist nur eine Konfiguration und nicht direkt verantwortlich für die tatsächliche Lastverteilung oder Terminierung des Datenverkehrs. Der Ingress-Controller, der in der Kubernetes-Clusterumgebung läuft, ist für die Umsetzung der Ingress-Regeln verantwortlich. Der Ingress-Controller liest die Ingress-Ressourcen und implementiert die definierten Regeln, indem er beispielsweise Lastausgleich, SSL/TLS-Terminierung, Authentifizierung und andere Funktionen bereitstellt. Es gibt verschiedene Ingress-Controller-Implementierungen wie zum Beispiel Nginx Ingress Controller, Traefik, HAProxy Ingress und viele mehr.

Insgesamt ermöglicht die Verwendung von Kubernetes Ingress die einfache Konfiguration und Verwaltung des externen Zugriffs auf Dienste innerhalb des Clusters und bietet eine Vielzahl von Funktionen zur Lastverteilung, Sicherheit und Routensteuerung des eingehenden Datenverkehrs an. Auf ein paar soll hier exemplarisch am Beispiel des Standard Ingress Controllers Nginx eingegangen werden.

Kubernetes Ingress-Ressourcen bieten verschiedene Möglichkeiten, um containerisierte Workloads zu schützen. Grundsätzlich fungiert eine Ingress-Ressource technisch gesehen als Reverse Proxy und leitet eingehenden Datenverkehr an verschiedene interne Services weiter, sodass diese nicht direkt dem Internet ausgesetzt sind. Die gängigsten Schutzmechanismen, die mit Ingress-Ressourcen ergänzend implementiert werden können, sind folgende:

- **SSL/TLS-Terminierung:** Eine Ingress-Ressource kann SSL/TLS-Terminierung unterstützen, indem sie den eingehenden verschlüsselten Datenverkehr entschlüsselt und dann an die Backends weiterleitet. Dies ermöglicht die Verschlüsselung des Datenverkehrs zwischen dem Client und dem Ingress-Controller.
- **Authentifizierung:** Durch die Konfiguration von Ingress-Ressourcen können verschiedene Authentifizierungs- und Autorisierungsmethoden implementiert werden. Beispielsweise kann eine Ingress-Ressource die Verwendung von Benutzername/Passwort, JWT (JSON Web Token) oder OAuth 2.0 ermöglichen, um den Zugriff auf die Workloads zu beschränken.
- **Rate Limiting:** Mit Ingress-Ressourcen können auch Rate-Limiting-Mechanismen implementiert werden, um den eingehenden Datenverkehr auf eine bestimmte Anzahl von Anfragen einer IP pro Sekunde zu begrenzen. Auch dies kann helfen, DDoS-Angriffe zu verhindern und die Verfügbarkeit der Workloads aufrechtzuerhalten.

Diese Schutzmechanismen können je nach den spezifischen Anforderungen und der verwendeten Ingress-Controller-Implementierung variieren. Nachfolgende Absätze haben daher eher illustrativen Charakter und zeigen, wie diese Mechanismen an Endpunkten zumeist außerhalb von Anwendungen definiert und durchgesetzt werden können.

SSL/TLS-Terminierung

Unter SSL/TLS-Terminierung versteht man den Prozess, bei dem der eingehende verschlüsselte Datenverkehr auf einem Proxy oder Load Balancer entschlüsselt wird, bevor er an den

internen Server weitergeleitet wird. Im Falle von Kubernetes übernimmt der Ingress die Rolle des Proxys. Normalerweise kommuniziert ein Client (z. B. ein Webbrowser) mit einem Server über eine verschlüsselte Verbindung mittels SSL/TLS, um die Vertraulichkeit und Integrität der übertragenen Daten zu gewährleisten. Diese Verschlüsselung findet auf der Anwendungsschicht statt, wobei der SSL/TLS-Handshake stattfindet, um eine sichere Verbindung herzustellen.

Bei der SSL/TLS-Terminierung übernimmt ein Proxy oder Load Balancer die Aufgabe, den SSL/TLS-Handshake mit dem Client durchzuführen und den Datenverkehr zu entschlüsseln. Der verschlüsselte Datenstrom wird dabei aufgebrochen, und der Proxy oder Load Balancer fungiert als Vermittler zwischen dem Client und dem internen Server. Das wirkt auf den ersten Blick unsicher, da die interne Kommunikation im Cluster ja nun nicht mehr verschlüsselt erfolgt, dennoch überwiegen die Vorteile SSL/TLS-Terminierung vor allem aus folgenden Gründen:

- Flexibilität bei der Zertifikatsverwaltung: Da der Proxy oder Load Balancer die SSL/TLS-Verbindung mit dem Client herstellt, kann er auch die Verwaltung der Zertifikate übernehmen. Dadurch können Zertifikate zentral verwaltet und erneuert werden, ohne dass jeder interne Server einzelne Zertifikate benötigt.
- Einfache Integration von Sicherheitsmaßnahmen: Durch die Entschlüsselung des Datenverkehrs auf dem Proxy oder Load Balancer können verschiedene Sicherheitsmaßnahmen wie WAF (Web Application Firewall), Intrusion Detection/Prevention-Systeme und andere Sicherheitsmechanismen leichter implementiert werden.

Um SSL/TLS-Terminierung in Kubernetes nutzen zu können, muss ein TLS-Secret in einem Kubernetes-Namespace installiert und mit einer Ingress-Ressource verknüpft werden. Hierzu muss als Erstes ein Secret-Objekt erstellt werden, das die erforderlichen Zertifikatsdateien enthält. Listing 16.17 zeigt ein Beispiel, wie ein TLS-Secret im Cluster hinterlegt werden kann (wir gehen dabei davon aus, dass ein gültiges HTTPS-Zertifikat vorliegt):

Listing 16.17 Anlegen eines Zertifikats zur Absicherung des HTTP-Datenverkehrs

```
kubectl create secret tls my-tls-secret \
--key <path/to/private_key> \
--cert <path/to/certificate>
```

Dieses Zertifikat kann dann an eine Ingress-Ressource, wie in Listing 16.18 gezeigt, gebunden werden und wird dann zur Verschlüsselung des HTTP-Datenverkehrs genutzt (HTTPS).

Listing 16.18 Angabe eines Zertifikats in einer Ingress-Ressource (HTTPS)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    cert-manager.io/issuer: letsencrypt-issuer # Optional, nur erforderlich, wenn
                                                # mit cert-manager automatisiert
                                                # Zertifikate erstellt und erneuert
                                                # werden sollen
```

```

spec:
  tls:                                     # Angabe des Zertifikats (HTTPS)
    - hosts: ["my.fancy.host.name.com"]
      secretName: my-tls-secret
  rules:
    - host: my.fancy.host.name.com
      http:
        paths:
          - path: /
            pathType: Prefix
        backend:
          service:
            name: my-service
            port: { number: 80 }

```

Alternativ zu einem vorhandenen Zertifikat kann man sich auch Zertifikate von einer Zertifizierungsinstanz automatisch ausstellen lassen (bspw. von Let's Encrypt). Dies kann mit dem Add-on Cert-Manager realisiert werden. Hierzu muss dieses Add-on allerdings im Cluster installiert und ein Issuer im Namespace eingerichtet werden. Dies kann bspw. wie in Listing 16.19 gezeigt geschehen.

Listing 16.19 Beispiel eines Let's Encrypt Issuers

```

apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: letsencrypt-issuer
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: notification@my.fancy.company.com
    privateKeySecretRef:
      name: issuer-account-key
    solvers:
      - http01:
          ingress:
            class: nginx

```

Authentifizierung mittels OAuth2

OAuth2 ist ein offenes Protokoll, das verwendet wird, um den Zugriff auf Ressourcen in Webanwendungen oder APIs sicher zu autorisieren. Es ermöglicht Benutzern, einer Anwendung Zugriff auf ihre geschützten Ressourcen zu gewähren, ohne dass die Anwendung Benutzernamen und Passwörter speichern oder weitergeben muss, was die Angriffsfläche reduziert und dem Prinzip der Datensparsamkeit folgt (siehe auch Abschnitt 17.2). Das OAuth2-Protokoll besteht aus mehreren Komponenten, darunter der Resource Owner (Benutzer), der Authorization Server (Autorsierungsserver), der Client (Anwendung) und der Resource Server (Ressourcenserver). Der Ablauf des Protokolls beinhaltet die Ausstellung von Zugriffstoken durch den Autorisierungsserver an den Client, die der Client dann verwendet, um auf die geschützten Ressourcen zuzugreifen.

Der OAuth2-Standard bietet verschiedene Autorisierungsflüsse, je nachdem, wie die Anwendung den Zugriff auf die geschützten Ressourcen benötigt. Zu den gängigen Flüssen gehören der „Authorization Code Grant“, der „Implicit Grant“, der „Client Credentials Grant“ und der „Resource Owner Password Credentials Grant“. Jeder Fluss hat seine eigenen Anwendungsfälle und Sicherheitsaspekte. OAuth2 wird in vielen Webanwendungen und APIs eingesetzt, um den Zugriff auf geschützte Ressourcen sicher zu ermöglichen, ohne dass die Anwendung direkten Zugriff auf Benutzerdaten haben muss. Es ist ein weit verbreitetes Protokoll und eine bewährte Methode zur sicheren Autorisierung in einer Vielzahl von Anwendungen.

Das Listing 16.20 zeigt ein Beispiel eines „Client Credentials Grant“-Autorisierungsflusses, bei dem ein Client (Anwendung) Anmeldeinformationen verwendet, um Zugriff auf Ressourcen zu erhalten, ohne die Identität eines Benutzers darzustellen. Solch ein „Client Credentials Grant“ kann in Kubernetes mittels einer Ingress-Ressource und entsprechenden Annotations eingerichtet werden:

1. Hierzu muss sichergestellt sein, dass ein Autorisierungsservice existiert, der den „Client Credentials Grant“ unterstützt. Der Autorisierungsserver sollte über die erforderlichen Endpunkte verfügen, um Zugriffstoken auf der Grundlage von Client-Anmeldeinformationen auszustellen.
2. Mittels einer Ingress-Ressource in einer Kubernetes-Umgebung kann eingehender Datenverkehr auf den entsprechenden Endpunkt eines Servers umgeleitet werden. Die genaue Konfiguration der Ingress-Ressource hängt dabei von der genutzten Kubernetes-Plattform und dem verwendeten Ingress-System ab.

Folgendes Beispiel zeigt illustrativ eine mögliche Ingress-Konfiguration.

Listing 16.20 Beispiel einer mittels OAuth abgesicherten Ingress-Ressource

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: oauth-secured-ingress
  annotations:
    nginx.ingress.kubernetes.io/auth-type: "oauth2"
    nginx.ingress.kubernetes.io/auth-method: "client_certificate"
    nginx.ingress.kubernetes.io/auth-url: "https://your-auth-server.com/token"
    nginx.ingress.kubernetes.io/auth-signin: "https://your-auth-server.com/login"
spec:
  rules:
    - http:
        paths:
          - path: /your-resource-path
            backend:
              serviceName: your-service
              servicePort: 80
```

Rate Limiting

Unter Rate Limiting versteht man einen Schutzmechanismus, der darauf abzielt, die Anzahl der Anfragen oder Aktionen von Benutzern oder Clients innerhalb eines bestimmten Zeitraums zu begrenzen. Es wird eingesetzt, um die Ressourcenverwendung zu steuern,

um Überlastung, Missbrauch und unverhältnismäßigen Datenverkehr auf einem System zu verhindern. Mittels Rate Limiting können folgende Ziele erreicht werden:

- **Verhinderung von Überlastung:** Rate Limiting hilft dabei, die Last auf einem System zu verteilen, indem es die Anzahl der Anfragen oder Aktionen pro Sekunde, Minute oder Stunde begrenzt. Dadurch wird verhindert, dass das System überlastet und die Performance für alle Benutzer beeinträchtigt wird.
- **Schutz vor DDoS-Angriffen:** Rate Limiting kann auch dazu beitragen, Distributed Denial-of-Service-(DDoS-)Angriffe abzuwehren. Durch Begrenzung der Anzahl der Anfragen pro Sekunde oder pro IP-Adresse kann der Datenverkehr von einem Angreifer reduziert oder blockiert werden, wodurch das Zielsystem vor Überlastung geschützt wird.
- **Gewährleistung einer gerechten Ressourcenverteilung:** Durch das Rate Limiting wird sichergestellt, dass Ressourcen gerecht zwischen Benutzern oder Clients verteilt werden und nicht durch einzelne Nutzer monopolisiert werden können. Jeder Benutzer oder Client erhält eine faire Chance, auf das System zuzugreifen, anstatt dass einzelne Benutzer oder Clients alle verfügbaren Ressourcen beanspruchen (Ressourcenmonopolisierung).

Rate Limiting muss allerdings sorgfältig konfiguriert und dimensioniert werden, um sicherzustellen, dass es angemessen auf die Anforderungen des Systems und der Benutzer abgestimmt ist. Zu restriktive Limitierungen können die Benutzbarkeit beeinträchtigen, während zu großzügige Limitierungen den Schutzapekt verringern können. Eine gute Balance ist entscheidend, um die Vorteile des Rate Limiting zu nutzen und gleichzeitig ein positives Nutzungserlebnis zu gewährleisten.

Man kann Rate Limiting sehr einfach in einer Kubernetes Ingress-Ressource bspw. mittels der `nginx.ingress.kubernetes.io/limit-rps` Annotation konfigurieren. Diese Annotation legt die maximale Anzahl von Anfragen pro Sekunde (Requests per Second, RPS) fest, die für den Zugriff auf den Ingress-Endpunkt erlaubt sind. Listing 16.21 zeigt ein Beispiel, wie Rate Limiting in einer Ingress-Ressource aktiviert werden kann:

Listing 16.21 Beispiel einer Rate-limitierten Ingress-Ressource

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: rate-limited-ingress
  annotations:
    nginx.ingress.kubernetes.io/limit-rps: "10" # Maximal 10 Anfragen pro Sekunde
spec:
  rules:
    - host: example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: my-service
                port: { number: 80 }
```

16.2.2 Namespace-basierte Netzwerkisolation

In Abschnitt 9.3.9 wurde bereits gezeigt, wie man mithilfe von Namespaces, dem Role Based Access Model, Quotas, Limit Ranges und Network Policies die Workloads verschiedener Nutzer voneinander trennen kann, um verschiedene Tenants voneinander zu isolieren. Diese Inhalte sollen daher an dieser Stelle nicht wiederholt werden.

Die Netzwerkisolation ist dabei vielleicht der neuralgischste Punkt. Standardmäßig erlauben Kubernetes-Cluster die uneingeschränkte Kommunikation zwischen Pods innerhalb des Clusters, was ein Sicherheitsrisiko darstellen kann, insbesondere in mandantenfähigen Umgebungen, in denen mehrere Anwendungen und Teams koexistieren. Dies öffnet offensichtlich Tür und Tor für vielfältige Lateral Movements innerhalb des Clusters, sofern es einem Angreifer – auf welche Art auch immer – gelingt, einen Foothold in irgendeinem Pod zu erlangen. Auf die Möglichkeiten zur Netzwerkisolation soll daher an dieser Stelle noch einmal detaillierter eingegangen werden.

Eine NetworkPolicy ist ein Kubernetes-Objekt, mit dem Zugriffsrichtlinien erstellt werden können, um die Kommunikation zwischen Pods und externen Einheiten in einem Namensraum auf der Grundlage verschiedener Faktoren wie IP-Adressen, Ports, Protokolle und Labels einzuschränken. Ein Ingress-Abschnitt definiert hierfür Regeln für eingehenden Verkehr, während ein Egress-Abschnitt Regeln für ausgehenden Verkehr definiert. Dabei werden Selektoren verwendet. Pod-Selektoren (`podSelector`) wählen Pods anhand ihrer Labels aus, Namespace-Selektoren (`namespaceSelector`) wählen Pods in bestimmten Namespaces aus, und mittels IP-Blöcken (`ipBlock`) können IP-Adressblöcke spezifiziert werden, denen der Zugriff auf Pods erlaubt oder verweigert wird.

Ingress-Datenverkehr bezieht sich auf eingehenden Netzwerkverkehr, der an einen Pod oder eine Gruppe von Pods im Kubernetes-Cluster gerichtet ist. Wenn beispielsweise ein Benutzer außerhalb des Clusters eine Anfrage an einen Pod innerhalb des Clusters sendet, wird dieser Datenverkehr als Ingress-Verkehr zu diesem Pod betrachtet. Egress-Datenverkehr hingegen bezieht sich auf ausgehenden Netzwerkverkehr von einem Pod oder einer Gruppe von Pods im Kubernetes-Cluster. Wenn beispielsweise ein Pod im Cluster eine Anfrage an einen externen Dienst oder Endpunkt außerhalb des Clusters sendet, wird dieser Verkehr als Egress-Datenverkehr von diesem Pod betrachtet.

Mittels NetworkPolicies können in Kubernetes Richtlinien erstellt werden, die festlegen, wie Pods innerhalb eines bestimmten Namespaces miteinander und mit externen Entitäten kommunizieren können. NetworkPolicy-Regeln können auf verschiedenen Faktoren wie IP-Adressen, Ports, Protokollen und Labels basieren, sodass Datenverkehr je nach Sicherheitsanforderungen flexibel auf bestimmte Pods oder Gruppen von Pods beschränkt werden kann.

Pod-Selektoren

Eine NetworkPolicy kann hierzu mittels des Felds `podSelector` Pods auf der Grundlage ihrer Labels auswählen und bestimmt so, für welche Pods diese Richtlinie gilt. Mehrere Richtlinien wirken additiv. Listing 16.22 zeigt ein Beispiel einer NetworkPolicy für Backend-Pods, die über das Label `name: backend` selektiert werden. Der Ingress-Abschnitt erlaubt für diese Backend Pods eingehenden Verkehr von Frontend-Pods auf Port 8080, der Egress-Abschnitt definiert Regeln für ausgehenden Verkehr an Datenbank-Pods auf Port 5432.

Listing 16.22 Network Policy auf Basis von Pod-Selektoren innerhalb eines Namespace

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-network-policy
spec:
  podSelector:
    matchLabels: { name: backend }
  policyTypes: ["Ingress", "Egress"]
  ingress:
  - from:
    - podSelector:
      matchLabels: { name: frontend }
    ports:
    - port: 8080
      protocol: TCP
  egress:
  - to:
    - podSelector:
      matchLabels: { name: database }
    ports:
    - port: 5432
      protocol: TCP

```

Namespace-Selektoren

Listing 16.23 zeigt das Beispiel einer NetworkPolicy, die den internen Datenverkehr innerhalb des Namespaces ermöglicht, indem alle Pods im Namespace ausgewählt werden. Es wird kein spezifisches Quell-Pod-Label angegeben, was bedeutet, dass alle Pods erlaubt sind, Verbindungen zu anderen Pods im selben Namespace herzustellen.

Für den Egress-Verkehr wird keine spezifische Regel definiert, was bedeutet, dass alle ausgehenden Verbindungen erlaubt sind. Dadurch können die Pods Verbindungen zu externen Ressourcen herstellen.

Für den Ingress-Verkehr wird die Quelle auf den bestimmten Ingress-Namespace beschränkt. Dieser muss dem tatsächlichen Namen des Namespace des Kubernetes-Clusters entsprechen, in dem die Ingress Pods laufen (vgl. Abschnitt 9.3.6). Gegebenenfalls muss das Manifest entsprechend der Umgebung angepasst werden. Dadurch wird sichergestellt, dass externer Datenverkehr nur über einen Ingress in den Namespace zugelassen wird.

Listing 16.23 Network Policy (nur NS-interner und aus dem Ingress-NS kommender Datenverkehr)

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-internal-namespace-external-ingress
spec:
  podSelector: {}          # Wählt alle Pods im Namespace aus
  policyTypes:
  - Ingress
  - Egress

```

```

ingress:
- from:
  - namespaceSelector:
    matchLabels:
      name: ingress # Erlaubt Verkehr aus Ingress Namespace (ggf. anders benannt)
  - podSelector: {} # Erlaubt Datenverkehr innerhalb des Namespace
egress: []

```

IP-Blöcke

Ferner ist es möglich, auch IP-Adressblöcken den Zugriff auf Pods zu erlauben oder zu verweigern; zum Beispiel nur aus einem vertrauenswürdigen Admin-Netz. Das Feld `ipBlock` kann hierzu in NetworkPolicy's verwendet werden, um einen CIDR-Block oder eine einzelne IP-Adresse zu definieren. Es kann analog zu `namespaceSelector` und `podSelector` in Ingress- oder Egress-Regeln verwendet werden.

Listing 16.24 zeigt die Nutzung, wie das `ipBlock`-Feld verwendet wird, um den CIDR-Block `192.168.0.0/16` anzugeben. Der Eingangsbereich lässt eingehenden Datenverkehr zum Pod an Port 8080 unter Verwendung des TCP-Protokolls nur zu, wenn die Quell-IP-Adresse innerhalb dieses CIDR-Blocks liegt.

Listing 16.24 NetworkPolicy (Zugriff von einem IP-Adressblock)

```

...
ingress:
- from:
  - ipBlock: { cidr: 192.168.0.0/16 }
ports:
- port: 8080
protocol: TCP
...

```



Tipp

IP-Blöcke sollten in NetworkPolicy's nur für IP-Adressen verwendet werden, die nicht durch das Kubernetes-Cluster selber verwaltet werden. Innerhalb des Clusters sollte die Selektion von Pods mittels Namespace- und Pod-Selektoren erfolgen.



CIDR (Classless Inter-Domain Routing)

Ein CIDR-Block ist eine Methode zur Darstellung und Aufteilung von IP-Adressbereichen. Es handelt sich um eine Notation zur Identifizierung und Gruppierung von IP-Adressen und Netzwerken.

Ein CIDR-Block besteht aus einer IP-Adresse und einem Netzwerkpräfix, die durch einen Schrägstrich ("/") getrennt sind. Die IP-Adresse gibt den Beginn des Adressbereichs an, während die Netzwerkpräfixlänge die Anzahl der in der Adresse enthaltenen führenden Bits angibt, die den Netzwerkteil der Adresse darstellen. Der restliche Teil der Adresse stellt den Host-Bereich dar.

Zum Beispiel repräsentiert der CIDR-Block „192.168.0.0/24“ das Netzwerk mit der IP-Adresse „192.168.0.0“ und einer Netzwerk-Präfixlänge von 24 Bits. Das bedeutet, dass die ersten 24 Bits der Adresse den Netzwerkteil darstellen, während die restlichen 8 Bits für die Hosts innerhalb des Netzwerks reserviert sind.

CIDR-Blöcke werden häufig verwendet, um IP-Adressbereiche in Subnetze aufzuteilen und Routing-Informationen effizient zu verteilen. Sie ermöglichen eine flexible und skalierbare Verwaltung von IP-Adressen und erleichtern die Konfiguration von Netzwerken und Firewalls durch die Verwendung präziser Netzwerkbereiche.

Um eine spezifische IP-Adresse in einer NetworkPolicy zu definieren, kann bspw.

```
cidr: 10.50.1.162/32
```

angegeben werden. Ein mit der Netzwerkpräfixlänge „/32“ definierter CIDR-Block bedeutet, dass nur die spezifische IP-Adresse vor dem Präfix erlaubt ist.

16.2.3 Pod Hardening

Die Härtung von Workloads in Kubernetes ist ein wesentlicher Aspekt, um die Sicherheit und Integrität von Anwendungen und Ressourcen in Kubernetes-Clustern zu gewährleisten. Dies beinhaltet verschiedene Maßnahmen, um potenzielle Schwachstellen zu minimieren und sicherheitsrelevante Best Practices umzusetzen. Zu den wichtigen Themen der Workload-Härtung gehören die Verwendung von Service-Accounts zur Authentifizierung und Zugriffskontrolle, die Limitierung von Ressourcen mittels Quotas, die Anwendung von Security Contexts zur Isolation und Sicherheit von Containern sowie die Durchsetzung von Sicherheitsanforderungen mittels Policy's. Diese Mechanismen spielen eine entscheidende Rolle, um die Sicherheit von Workloads in Kubernetes zu stärken, den Lebenszyklus von Cyber-Angriffen (vgl. Bild 16.1) zu stören und dabei potenzielle Angriffsvektoren (insbesondere Privilege Escalation, Lateral Movement) aus Bild 16.2 zu entschärfen.

Sichere Handhabung von Service-Accounts

In Kubernetes ist ein ServiceAccount ein Objekt, das eine Identität für Pods und andere Ressourcen im Cluster bereitstellt. Ein ServiceAccount wird verwendet, um Pods zu authentifizieren und ihnen Zugriffsrechte auf Ressourcen im Kubernetes-Cluster zu gewähren.

Listing 16.25 Beispiel eines Service-Accounts in Kubernetes

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account

# Alternativ kann der Service-Account auch mittels kubectl
# innerhalb eines Namespaces angelegt werden.
# > kubectl create serviceaccount my-service-account
```

Listing 16.26 zeigt dabei, wie ein ServiceAccount angelegt werden kann. Und Listing 16.27 zeigt, wie ein solcher ServiceAccount mit einem Pod (oder sonstigem Workload wie Deployment, DaemonSet, Job, StatefulSet etc.) verknüpft werden kann.

Listing 16.26 Beispiel der Zuordnung eines Service-Accounts an einen Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-service-account
spec:
  serviceAccountName: my-service-account
  containers:
  - name: container
    image: image:tag
  ...
```

Ein ServiceAccount besteht dabei aus zwei Hauptkomponenten:

- **Token:** Beim Erstellen eines ServiceAccounts wird automatisch ein Token generiert, das dem ServiceAccount zugeordnet wird. Dieses Token wird verwendet, um den ServiceAccount gegenüber der Kubernetes-API zu authentifizieren. Jeder Pod, dem ein ServiceAccount zugewiesen ist, erhält eine Kopie dieses Tokens, um sich als der zugehörige ServiceAccount zu authentifizieren.
- **Identität und Rollen:** Ein ServiceAccount hat eine eindeutige Identität im Kubernetes-Cluster. Auf der Grundlage dieser Identität können dann über die Kubernetes-RBAC-(Role-Based Access Control-)Mechanismen Rollen und Berechtigungen zugewiesen werden. Eine Rolle definiert, welche Aktionen ein ServiceAccount auf bestimmte Ressourcen ausführen kann, während eine Rollenbindung die Zuweisung einer Rolle zu einem bestimmten ServiceAccount oder einer bestimmten Gruppe von ServiceAccounts darstellt (vgl. *Abschnitt 9.3.9.1*).

Durch die Verwendung von ServiceAccounts in Kubernetes können Pods somit identifiziert und authentifiziert werden, um damit auf bestimmte Ressourcen im Cluster zuzugreifen. Dies ermöglicht einerseits eine feinere Zugriffskontrolle, eine Trennung von Verantwortlichkeiten, und eine bessere Sicherheit in Kubernetes-Clustern andererseits ermöglicht dies auch, dass kompromittierte Pods Zugriff auf Teile der Kubernetes Control-Plane erlangen können. Die häufigsten Berechtigungen, die ein Service-Account haben kann, sind:

- **Lesen von Ressourcen:** Ein ServiceAccount kann Lesezugriff auf Ressourcen im Cluster haben, wie bspw. Pods, Services, ConfigMaps oder gar Secrets.
- **Schreiben oder Aktualisieren von Ressourcen:** Ein ServiceAccount kann Schreibzugriff haben, um Ressourcen zu erstellen oder zu aktualisieren, z. B. Pods oder Deployments.
- **Löschen von Ressourcen:** Ein ServiceAccount kann die Berechtigung haben, Ressourcen zu löschen, z. B. das Löschen von Pods oder Services.
- **Zugriff auf die Kubernetes-API:** Ein ServiceAccount kann Zugriff auf die Kubernetes-API haben, um beispielsweise Informationen über den Cluster oder andere Ressourcen abzurufen.

Alle Pods erhalten damit die Rechte im Kubernetes-System, die dem Pod zugewiesenen ServiceAccount mittels RoleBindings (RBAC) zugewiesen wurden (vgl. *Abschnitt 9.3.9.1*). Pods können also nicht nur Workloads ausführen, sondern auch das Kubernetes-System programmatisch nutzen und bspw. Pods mit privilegiertem Zugriff starten (und damit einem Lateral Movement eines Angreifers dienen) oder eine Vielzahl von Pods starten, um bspw. eine verbrauchsorientierte Denial-of-Service-Attacke zu starten.

Bei der Verwendung von ServiceAccounts sollten daher folgende Aspekte im Hinblick auf Härtung und IT-Sicherheit berücksichtigt werden:

- **Prinzip des geringsten Privilegs:** Es sollten nur die Berechtigungen gewährt werden, die zum Ausführen einer Aufgabe benötigt werden. Vermeiden Sie die Vergabe von zu weitreichenden Berechtigungen, um das Risiko von Missbrauch oder unbefugtem Zugriff zu verringern. Verfolgen Sie das Prinzip des geringsten Privilegs und gewähren Sie nur die minimale Anzahl von Berechtigungen, die für den ordnungsgemäßen Betrieb der Anwendung erforderlich sind.
- **Separation of Duties:** Man sollte es vermeiden, ServiceAccounts mit unterschiedlichen Verantwortlichkeiten oder Zugriffsbereichen dieselben Berechtigungen zu gewähren. Durch eine Trennung der Verantwortlichkeiten und Zugriffsrechten zwischen verschiedenen ServiceAccounts lässt sich das Risiko von Missbrauch oder unbefugtem Zugriff reduzieren.
- **Berechtigungen nur innerhalb von Namespaces zuweisen:** Bei der Zuweisung von Berechtigungen an ServiceAccounts sollten nur die erforderlichen Rollen und Berechtigungen zugewiesen und die Verwendung von Cluster-Rollen oder globalen Berechtigungen nur in begründbaren Ausnahmefällen zugelassen werden.
- **Periodisches Überprüfen der Berechtigungen:** Regelmäßig sollten die Berechtigungen der ServiceAccounts überprüft und auf ihre Aktualität hin überprüft werden. Dabei sollte geprüft werden, ob alle gewährten Berechtigungen immer noch erforderlich sind und ob es Anzeichen für unbefugte oder missbräuchliche Verwendung gibt.
- **Schützen von Tokens:** ServiceAccounts nutzen Zugriffstoken, um sich gegenüber der Kubernetes-API zu authentifizieren. Es ist daher wichtig, diese Tokens zu schützen und sicherzustellen, dass sie nicht unbeabsichtigt preisgegeben werden. Speichern Sie Tokens daher nur in den dafür vorgesehenen Kubernetes-Konzepten wie Secrets. Geben Sie Tokens von ServiceAccounts, die Pods zugewiesen werden, in keinem Fall nach außen weiter.
- **Monitoring und Protokollierung:** Überwachen Sie die Aktivitäten der ServiceAccounts und protokollieren Sie deren Aktionen. Durch eine effektive Überwachung können verdächtige Aktivitäten erkannt und ungewöhnliches Verhalten identifiziert werden.



Tip

Einem Pod, der keine Interaktion mit dem Kubernetes-System machen muss, sollte niemals ein ServiceAccount zugewiesen werden (oder nur ein ServiceAccount ohne Rechte).

Limitierung von Ressourcen

Unter der Monopolisierung von Ressourcen versteht man die Situation, in der Nutzer eine bestimmte Ressource oder eine Gruppe von Ressourcen derart nutzen, als wäre diese Ressource

exklusiv und müsste nicht mit anderen geteilt werden. Dies kann zu unfairen Ressourcenverteilungen und Einschränkungen für andere Nutzer führen. So könnten z. B. einige Pods nicht ausgeführt werden, weil andere Pods bereits den kompletten Prozessor oder Hauptspeicher belegen. Auch wenn eine Ressourcenmonopolisierung meist nicht zu einer Datenausleitung führt, kann eine solche für verbrauchsorientierte Denial-of-Service-Angriffe genutzt werden. Eine Ressourcenmonopolisierung in Kubernetes kann auftreten, wenn bestimmte Pods, Deployments oder Namespaces übermäßige Ressourcen für sich beanspruchen und dadurch anderen Anwendungen im Cluster benötigte Ressourcen nicht mehr (ausreichend) zur Verfügung stehen. Um die Ressourcenmonopolisierung in Kubernetes zu verhindern, gibt es mehrere Ansätze:

- **Quotas:** Kubernetes bietet die Möglichkeit, Ressourcenquoten auf Namespace-Ebene festzulegen. Durch die Festlegung von Grenzwerten für CPU, Speicher und andere Ressourcen können die verfügbaren Ressourcen fair auf die Anwendungen im Cluster verteilt werden. Ressourcenquoten helfen dabei, dass keine einzelne Anwendung die gesamten Ressourcen monopolisiert (vgl. *Abschnitt 9.3.9.2*).
- **LimitRange:** Mit LimitRange können spezifische Ressourcengrenzwerte für Pods oder Namespaces festgelegt werden. Dadurch wird verhindert, dass einzelne Pods oder Namespaces übermäßige Ressourcen für sich beanspruchen und die Verfügbarkeit für andere Anwendungen beeinträchtigen können (vgl. *Abschnitt 9.3.9.2*).



Was sind Denial-of-Service-Angriffe?

Denial-of-Service-(DoS-)Angriffe sind bösartige Handlungen, bei denen das Ziel darin besteht, die Verfügbarkeit eines Computersystems, eines Dienstes oder einer Anwendung zu beeinträchtigen oder vollständig zu blockieren. Bei einem DoS-Angriff wird versucht, verfügbare Ressourcen zu erschöpfen. Ziel ist es dabei, dass das System nicht mehr in der Lage ist, legitimen Benutzern den Zugriff oder die Nutzung zu ermöglichen.

Es werden verschiedene Arten von DoS-Angriffen unterschieden:

- **Flooding-Angriffe:** Bei dieser Art von Angriff wird das Zielsystem mit einer großen Anzahl von Anfragen, Datenpaketen oder Verbindungen überflutet. Beispiele sind SYN Flood-Angriffe, bei denen ein Angreifer eine große Anzahl von SYN-Anfragen an ein System sendet, um die Verbindungstabelle des Zielsystems zu überlasten und es für legitime Verbindungen unerreichbar zu machen.
- **Verbrauchsorientierte Angriffe:** Bei diesen Angriffen wird das Zielsystem gezielt mit Anfragen oder Daten belastet, die eine hohe Verarbeitungsleistung oder Ressourcen wie CPU, Speicher oder Netzwerkanwendung erfordern. Dadurch werden die Ressourcen des Zielsystems erschöpft, und es kann den legitimen Benutzern keinen Service mehr bieten.
- **Distributed Denial-of-Service-(DDoS)-Angriffe:** Bei einem DDoS-Angriff wird das Zielsystem von einer Vielzahl von Computerressourcen angegriffen, die Teil eines Botnetzes oder eines Netzwerks von kompromittierten Computern sind. Dies ermöglicht es den Angreifern, einen viel größeren Angriffsverkehr zu erzeugen und das Ziel mit einem koordinierten Angriff zu überwältigen.

DoS-Angriffe können schwerwiegende Auswirkungen haben, wie z. B. finanzielle Verluste, Rufschädigung, Beeinträchtigung der Produktivität und Beeinträchtigung der Dienstqualität für legitime Benutzer.

Security Context

Ein Security Context in Kubernetes ist eine Konfigurationsoption, mit der Sicherheitsattribute für Pods oder Container festgelegt werden können. Der Security Context ermöglicht die Kontrolle verschiedener Aspekte der Sicherheit, wie z. B. Benutzer- und Gruppenidentitäten, Berechtigungen, Zugriffsrechte und andere Sicherheitsrichtlinien. Der Security Context kann sowohl auf Pod- als auch Container-Ebene in Kubernetes konfiguriert werden:

- **Pod-Ebene:** Der Security Context auf Pod-Ebene wird auf alle Container in einem Pod angewendet. Hier können Attribute wie der Benutzer, die Gruppe und die Berechtigungen für alle Container im Pod festgelegt werden.
- **Container-Ebene:** Hier wird der Security Context auf Container-Ebene spezifisch für jeweils einen einzelnen Container innerhalb eines Pods konfiguriert. Dadurch können Container innerhalb desselben Pods unterschiedliche Sicherheitsattribute haben.

Grundsätzlich ist es möglich, sowohl einen Security Context auf Pod-Ebene als auch auf Container-Ebene zu definieren. In diesem Fall wird der Container-Ebene Security Context die höhere Priorität zugewiesen.

Die gängigsten Attribute, die im Security Context definiert werden können, sind:

- **runAsUser:** Legt den Benutzer (UID) fest, unter dem der Prozess im Container ausgeführt werden soll.
- **runAsGroup:** Legt die Gruppe (GID) fest, unter der der Prozess im Container ausgeführt werden soll.
- **fsGroup:** Legt die Gruppe (GID) für das Dateisystem im Container fest, um den Zugriff auf gemeinsame Dateisysteme zu regeln.
- **privileged:** Legt fest, ob der Container privilegierte Zugriffsrechte auf dem Hostsystem haben soll.
- **readOnlyRootFilesystem:** Legt fest, ob das Dateisystem des Containers als schreibgeschützt gemountet werden soll.
- **capabilities:** Steuert die Fähigkeiten des Containers und ermöglicht es, bestimmte Berechtigungen einzuschränken (siehe auch *Abschnitt 8.2.2*).
- **seLinuxOptions:** Ermöglicht die Konfiguration von SELinux-Optionen für den Container.
- **appArmorProfile:** Legt das AppArmor-Profil für den Container fest.
- **seccompProfile:** Legt das Seccomp-Profil für den Container fest.

Der Security Context ist eine wichtige Funktion, um die Sicherheit von Pods und Containern in Kubernetes zu gewährleisten und insbesondere Container Breakouts und damit Lateral Movements zu erschweren. Durch die Verwendung des Security Context können Zugriffsrechte, Identitäten und andere sicherheitsrelevante Aspekte gesteuert werden, um die Angriffsfläche für Lateral Movements zu verringern.

Listing 16.27 zeigt ein Beispiel einer Pod-Definition, in der der Security Context verwendet wird, um die Fähigkeiten (Capabilities) auf Dateioperationen einzuschränken und den Pod nicht als Root-Benutzer auszuführen.

- Der securityContext auf Pod-Ebene wird verwendet, um den Pod nicht als Root-Benutzer auszuführen. Die Option `runAsNonRoot: true` stellt sicher, dass der Pod als nicht privilegierter Benutzer ausgeführt wird.
- Der securityContext auf Container-Ebene wird verwendet, um die Fähigkeiten einzuschränken. Die Option `Capabilities` ermöglicht das Hinzufügen oder Entfernen von bestimmten Fähigkeiten.
- In diesem Fall werden die Fähigkeiten zuerst alle entfernt (`drop: ALL`), und dann werden selektiv einige Fähigkeiten hinzugefügt (`add: CHOWN` und `add: DAC_OVERRIDE`). Das bedeutet, dass der Container keine Fähigkeiten standardmäßig besitzt, außer der Fähigkeit, Besitzerinformationen zu ändern (`CHOWN`) und DAC (Discretionary Access Control) zu überschreiben. Weitere gewünschte Fähigkeiten können in der Liste unter `add` hinzugefügt werden.

Listing 16.27 Beispiel einer Security Context-Definition für einen Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  securityContext:      # Pod-Level
    runAsNonRoot: true
  containers:
    - name: my-container
      image: my-image
      securityContext:      # Container-Level
        capabilities:
          drop:["ALL"]
          add: ["CHOWN", "DAC_OVERRIDE"] # Weitere Fähigkeiten hier hinzufügen
```



Was ist SELinux?

SELinux (Security-Enhanced Linux) ist ein Sicherheitsmechanismus und ein Sicherheitsmodul, das in Linux-Kernels implementiert ist. Es wurde ursprünglich von der National Security Agency (NSA) entwickelt und später als Open-Source-Projekt veröffentlicht. SELinux erweitert die herkömmlichen Linux-Zugriffskontrollmechanismen durch die Implementierung von Mandatory Access Controls (MAC) auf dem Linux-Kernel.

Im Gegensatz zu den standardmäßigen Linux-Zugriffskontrollmechanismen wie den Discretionary Access Controls (DAC), bei denen die Zugriffsrechte von Dateien und Ressourcen vom Eigentümer und der Gruppe verwaltet werden, verwendet SELinux eine feinere Körnung der Zugriffskontrolle. Es erzwingt ein separates Sicherheitsrichtliniensystem, das Zugriffsberechtigungen für Ressourcen basierend auf vordefinierten Sicherheitskontexten vergibt.



Was ist AppArmor?

AppArmor (Application Armor) ist ein Sicherheitsmodul für den Linux-Kernel, das Mandatory Access Control (MAC) verwendet, um die Zugriffskontrolle für Prozesse auf dem System zu erhöhen. Es wurde von Novell entwickelt und ist als Open-Source-Software verfügbar. AppArmor ermöglicht die Definition von Sicherheitsrichtlinien auf der Basis von Profilen für einzelne Anwendungen und Dienste.

Mit AppArmor lassen sich granulare Zugriffskontrollen auf Prozessebene festlegen, indem sie die Aktionen und Berechtigungen einschränken, die von einer bestimmten Anwendung ausgeführt werden dürfen. AppArmor arbeitet auf Basis von Richtlinien, die durch Profile definiert werden, die spezifische Anwendungen oder Dienste repräsentieren.



Was ist Seccomp?

Seccomp steht für „Secure Computing Mode“ und ist ein Sicherheitsmechanismus für den Linux-Kernel. Es ermöglicht die Einschränkung der Systemaufrufe (Syscalls), die von einer Anwendung ausgeführt werden können. Seccomp wurde entwickelt, um die Angriffsfläche von Anwendungen zu reduzieren und die Auswirkungen von Sicherheitsverletzungen zu minimieren.

Mit Seccomp kann die Liste der erlaubten Systemaufrufe eingeschränkt werden, die eine Anwendung verwenden kann. Dies wird erreicht, indem ein „Syscall-Filter“ definiert wird, der festlegt, welche Systemaufrufe erlaubt oder verboten sind. Alle anderen Systemaufrufe werden abgelehnt und führen zu einem Fehler.

Dies bietet folgende Vorteile:

- **Reduzierung der Angriffsfläche:** Durch die Beschränkung der erlaubten Systemaufrufe wird die Angriffsfläche einer Anwendung verringert. Wenn eine Anwendung durch einen Angriff kompromittiert wird, sind weniger potenzielle Angriffsvektoren verfügbar.
- **Schutz vor Zero-Day-Angriffen:** Da viele bekannte Angriffe auf Sicherheitslücken in Systemaufrufen basieren, kann die Beschränkung der Systemaufrufe in einer Anwendung dazu beitragen, Zero-Day-Angriffe zu erschweren.
- **Fehlerisolierung:** Wenn eine Anwendung aufgrund eines Fehlers oder einer Schwachstelle abstürzt, kann Seccomp dazu beitragen, den Schaden aufgrund der eingeschränkten Berechtigungen zu begrenzen.

Es gibt zwei Hauptmodi für Seccomp. Im **Strict Mode** werden nur vordefinierte Systemaufrufe (z. B. read, write, exit) und bestimmte für die Anwendung notwendige Systemaufrufe erlaubt. Alle anderen Systemaufrufe führen zu einem Fehler. Im **Filter Mode** kann ein benutzerdefinierter Syscall-Filter definiert werden, um spezifische Systemaufrufe zu erlauben oder zu verbieten.



Was ist ein Zero-Day Exploit?

Ein Zero-Day Exploit bezieht sich auf eine Art von Sicherheitsangriff, der eine Schwachstelle in einer Software oder einem System ausnutzt, die den betroffenen Entwicklern oder Anbietern zuvor unbekannt ist. Der Begriff „Zero-Day“ bezieht sich auf den Umstand, dass die Schwachstelle noch nicht bekannt oder behoben wurde und daher den Angreifenden einen Vorteil verschafft und durch folgende Eigenschaften charakterisiert sind:

- **Unbekannte Schwachstelle:** Ein Zero-Day Exploit nutzt eine Sicherheitslücke aus, die den Software-Entwicklern oder Systembetreibern zum Zeitpunkt des Angriffs nicht bekannt ist. Daher haben sie „null Tage“ Zeit gehabt, um die Schwachstelle zu patchen oder Abwehrmaßnahmen zu ergreifen.
- **Aktiver Angriff:** Ein Zero-Day Exploit wird in der Regel von Angreifenden eingesetzt, um in Systeme einzudringen, Daten zu stehlen, Schadcode einzuschleusen oder andere böswillige Aktivitäten durchzuführen. Da die Schwachstelle unbekannt ist, sind die Verteidigungsmechanismen oft nicht darauf vorbereitet.
- **Zeitlicher Vorteil:** Der Angreifer kann den Zero-Day Exploit nutzen, um in ein System einzudringen, bevor die Software-Entwickler oder Systemadministratoren über die Schwachstelle informiert sind und entsprechende Gegenmaßnahmen ergreifen können.

Zero-Day Exploits sind besonders gefährlich, da sie die üblichen Schutzmechanismen umgehen können, die auf bekannte Schwachstellen abzielen. Die Auswirkungen eines erfolgreichen Zero-Day Exploits können schwerwiegend sein und reichen von Datenverlust und -manipulation bis hin zur Übernahme von Systemen oder Netzwerken. Da Zero-Day Exploits von ihrer Natur her unbekannt sind, ist es wichtig, dass veröffentlichte Sicherheitsupdates möglichst schnell eingespielt werden, um den zeitlichen Vorteil für den Angreifer zu minimieren (siehe auch Abschnitt 16.1.2 und Abschnitt 16.2.8).

Durchsetzung von Sicherheitsanforderungen mittels Security Policies

Obwohl Security Contexts ein wirkungsvolles Mittel sind, Pods sicher zu betreiben, ist die Nutzung von Security Contexts in Kubernetes nicht vorgeschrieben. Das heißt, Workloads können mit, ohne oder mit ungeeigneten Security Contexts in Kubernetes betrieben werden. Kubernetes ermöglicht zwar das Setzen von Security Contexts, kann aber die Nutzung nicht standardmäßig erzwingen und damit durchsetzen. Dadurch ist Sicherheit auf den „Goodwill“ der Nutzer bzw. Entwickler angewiesen. Erfahrungsgemäß ist das selten ein zielführender Ansatz.

Zur Durchsetzung von Sicherheitsanforderungen mittels Security Contexts waren bis Kubernetes 1.21 daher sogenannte Pod Security Policies (PSPs) erforderlich, mit der man Sicherheitsrichtlinien für Pods in einem Kubernetes-Cluster definieren kann. Mittels solcher PSPs lassen sich bestimmte Sicherheitsanforderungen und -beschränkungen für Pods definieren und durchsetzen. Dadurch wird die Sicherheit des Clusters erhöht, indem unsichere oder nicht konforme Pods gar nicht erst zur Ausführung gebracht werden können.

Ab Kubernetes 1.21 sind Pod Security Policies allerdings als „deprecated“ markiert und mit Version 1.25 ausgelaufen. Der offizielle Nachfolger von Pod Security Policies (PSPs) in

Kubernetes sind „Pod Security Standards“ (PSS), weswegen wir uns auf PSS fokussieren. Pod Security Standards sind eine Teilmenge von PSPs und konzentrieren sich vor allem auf die Durchsetzung einer sichereren Pod-Konfiguration und definieren eine Reihe von Best Practices und Sicherheitsanforderungen, die für Pods gelten sollten. Sie bieten einen konsistenten Ansatz für die Implementierung sicherer Pod-Konfigurationen und tragen dazu bei, sicherheitsrelevante Aspekte zu adressieren, ohne dabei zu restriktiv zu sein. Man geht hier einen Mittelweg zwischen Pragmatismus, Beherrschbarkeit für Entwickler sowie Security Best Practices, allerdings unter Verzicht der Möglichkeit, feingranulare (und fehleranfällige) Anpassungen vornehmen zu können.

Die Einführung von Pod Security-Standards ist Teil des Kubernetes-Sicherheitsprojekts SIG-Auth. Ziel des Projekts ist es, eine einheitliche Sicherheitsarchitektur für Kubernetes bereitzustellen, die die Sicherheitsrichtlinien und -kontrollen verbessert. Im Gegensatz zu PSPs, die als eigene API-Ressource existieren, sind Pod Security-Standards in Kubernetes nativ integriert und als Teil der Admission Control implementiert. Dies bedeutet, dass die Pod-Konfiguration bereits bei der Erstellung oder Aktualisierung von Pods auf Sicherheitskonformität überprüft wird.

Dabei werden verschiedene Isolationsebenen für Pods definiert. Mithilfe dieser lässt sich festlegen, wie das Verhalten von Pods auf klare und konsistente Weise eingeschränkt werden soll. Kubernetes bietet hierzu ab Version 1.25 einen integrierten Pod Security Admission Controller zur Durchsetzung der Pod-Sicherheitsstandards. Pod-Sicherheitseinschränkungen werden bei der Erstellung von Pods auf Namespace-Ebene angewendet. Der Pod Security Admission Controller stellt Anforderungen an den Security Contexts eines Pods und andere damit zusammenhängende Felder entsprechend drei definierter Stufen:

- **Privileged:** Die Privileged-Richtlinie ist absichtlich offen und völlig uneingeschränkt. Dieser Richtlinientyp ist in der Regel für Arbeitslasten auf System- und Infrastrukturebene gedacht, die von privilegierten, vertrauenswürdigen Benutzern verwaltet werden.
- **Baseline:** Die Baseline-Richtlinie zielt darauf ab, den Betrieb gängiger containerisierter Arbeitslasten zu erleichtern und gleichzeitig bekannte Privilegieneskalationen zu verhindern. Diese Richtlinie richtet sich an Anwendungsbetreiber und Entwickler von nicht kritischen Anwendungen. Da privilegierte Pods die meisten Sicherheitsmechanismen deaktivieren, sind privilegierte Pods in dieser Richtlinie verboten. Ebenso ist der privilegierte Zugriff auf den Host in der Basisrichtlinie nicht erlaubt. Dies schließt bspw. explizit das Mounten von HostPath-Volumes oder das Nutzen von HostPorts aus.
- **Restricted:** Die Restricted-Richtlinie setzt die aktuellen Best Practices für die Pod-Härtung durch, allerdings auf Kosten einer gewissen Kompatibilität. Das kann bspw. viele Helm-Charts aus öffentlichen Repositorys betreffen, die sich unter dieser Richtlinie gegebenenfalls nicht immer betreiben lassen. Sie richtet sich an Betreiber und Entwickler von sicherheitskritischen Anwendungen oder ist für Benutzer gedacht, deren Ursprung und Handlungsweise nicht eingeschätzt und nicht wirkungsvoll sanktioniert werden kann. Also für Nutzergruppen, die potenziell bösartige Absichten verfolgen und bereitgestellte Infrastruktur für eigene Zwecke missbrauchen könnten. Diese Richtlinie umfasst daher zusätzlich zu den Einschränkungen der Baseline-Richtlinie unter anderem, dass Volumes nicht mehr direkt, sondern nur noch mittels PersistentVolumeClaims gemountet werden können, Container nicht mehr als Root User laufen dürfen und alle zulässigen Betriebssystem-Capabilities entzogen werden (einzig und allein Ports unterhalb von 1024 dürfen eingerichtet werden, NET_BIND_SERVICE).

Diese Richtlinien können für Namespaces mittels Labels aktiviert werden und gelten damit auf allen Pods in einem Nameservice. Kubernetes definiert hierzu eine Reihe von Labels, die festlegen, welche der vordefinierten Sicherheitsrichtlinien für einen Namespace angewendet werden sollen. Das gewählte Label legt fest, welche Maßnahmen die Kontrollebene ergreift, wenn ein Verstoß festgestellt wird. Dabei werden die folgenden Modi unterschieden:

- **Enforce:** Verstöße gegen die Richtlinie führen zur Ablehnung des Pods.
- **Audit:** Verstöße gegen die Richtlinie führen dazu, dass diese im Audit-Protokoll protokolliert werden. Die Pods werden aber ausgeführt.
- **Warn:** Verstöße gegen die Richtlinie führen zu einer Warnung für den Benutzer. Die Pods werden aber ausgeführt.

Für einen Namespace kann man dabei einen oder alle Modi konfigurieren oder sogar verschiedene Stufen für verschiedene Modi festlegen. Für jeden Modus gibt es zwei Bezeichnungen, die die verwendete Richtlinie bestimmen. Listing 16.28 zeigt illustrativ die Sicherheitskonfiguration für einen Namespace, die

- alle Pods blockiert, die die Anforderungen der Baseline-Richtlinie der Kubernetes-Version 1.26 nicht erfüllen,
- entsprechende Verstöße protokolliert, die die Anforderungen der Restricted-Richtlinie der Kubernetes-Version 1.27 nicht erfüllen,
- sowie eine benutzerseitige Warnung für Pods generiert, die die Anforderungen der Restricted-Richtlinie der neuesten Kubernetes-Version nicht erfüllen (um z. B. Nutzer frühzeitig über neue Best Practices und Entwicklungen zu informieren).

Listing 16.28 Beispiel eines Namespace mit mehreren Sicherheitsrichtlinien

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
  labels:
    # Setze Baseline Policy der Kubernetes-Version 1.26 durch
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/enforce-version: v1.26

    # Logge Verstöße gegen die Restricted Policy der Kubernetes-Version 1.26
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/audit-version: v1.26

    # Warne bei Verstößen gegen die Restricted Policy der neuesten Kubernetes-Version
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: latest
```

Das Labeling kann – wie Listing 16.29 zeigt – natürlich auch mittels kubectl und damit bspw. automatisiert in Deployment-Pipelines erfolgen.

Listing 16.29 Labeling von Namespaces mittels kubectl zur Durchsetzung von Sicherheitsrichtlinien

```
# Selektiert alle Namespaces, die keine Sicherheitsrichtlinie haben.
# Hier gibt es ggf. Nachbesserungsbedarf.
#
kubectl get namespaces --selector='!pod-security.kubernetes.io/enforce'

# Setzt die Restricted Richtlinie in Version 1.27 auf einem Namespace durch
kubectl label --overwrite ns my-namespace \
    pod-security.kubernetes.io/enforce=restricted \
    pod-security.kubernetes.io/enforce-version=v1.27
```

16.2.4 Erhöhung der Container Runtime Isolation

Die Betriebssystemvirtualisierung ist im Gegensatz zur Maschinenvirtualisierung grundsätzlich schwächer hinsichtlich der Isolation von Workloads, da man bei diesem Ansatz ja den Betriebssystemkernel gemeinsam nutzt. Dies bietet Angreifern naturgegeben eine entsprechend größere und anfälliger Angriffssoberfläche. Es gibt verschiedene Tools und Technologien, die helfen können, den Grad der Isolation containerisierter Workloads zu erhöhen. Diese unterteilt man zumeist in:

- **Hypervisor-basierte Containerisierung:** Bei der Verwendung von Hypervisor-basierten Containerisierungslösungen wird die Virtualisierungsgrenze durch den Hypervisor und die zugrunde liegende Hardware durchgesetzt. Beispiele für solche Lösungen sind die Verwendung des Windows-Hypervisors Hyper-V in Kombination mit containerfähigen Betriebssystemen oder Container-Engines, die jede Anwendung in einem leichten Hypervisor bereitstellen, um die Sicherheit zu erhöhen.
- **Kernel-basierte Lösungen:** Die bereits in Abschnitt 16.2.3 erwähnte Lösung „seccomp“ kann verwendet werden, um die Systemaufruf-Fähigkeiten eines Containers zu begrenzen und somit die Angriffsfläche des Kernels zu reduzieren. Seccomp kann durch die Anwendung einer entsprechenden Pod Policy erzwungen werden.
- **Anwendungssandboxes:** Einige Container-Engine-Lösungen bieten die Möglichkeit, eine zusätzliche Isolationsschicht zwischen der containerisierten Anwendung und dem Host-Kernel hinzuzufügen. Diese Isolationsgrenze zwingt die Anwendung dazu, in einer virtuellen Sandbox zu arbeiten, um das Host-Betriebssystem vor schädlichen oder zerstörerischen Operationen zu schützen.

Tabelle 16.1 zeigt einen Überblick von Container Runtime Environments mit erhöhter Isolation.

Tabelle 16.1 Container Runtime Environments mit erhöhter Isolation

| Technologie | Art | Beschreibung |
|------------------|---|---|
| Kata Container | Hypervisor-basiert Kernel-basiert | Kata Containers ist eine Open-Source-Technologie, die eine Kombination aus leichtgewichtigen Containern und der Sicherheit von virtuellen Maschinen (VMs) bietet. Kata Containers verwendet einen leichten Hypervisor, um jede Container-Instanz in einer eigenen VM zu isolieren. |
| Firecracker | Hypervisor-basiert | Firecracker ist eine Open-Source-Technologie, die von Amazon Web Services (AWS) entwickelt wurde. Es handelt sich um einen speziell entwickelten Hypervisor, der extrem leichte und schnelle Micro-VMs bereitstellt. |
| gVisor | Hypervisor-basiert Kernel-basiert Sandbox | gVisor ist eine Open-Source-Sandbox-Technologie, die von Google entwickelt wurde. Es verwendet eine Kombination aus Kernel-basierten Techniken und virtuellen Maschinen, um eine isolierte und sichere Umgebung für Container bereitzustellen. |
| Nabla Containers | Sandbox Microkernel | Nabla Containers ist eine Kernel-basierte Container-Lösung, die eine leichte und sichere Laufzeitumgebung auf Microkernel-Technologie nutzt. Nabla Containers reduziert mittels eines Microkernels die Angriffsfläche auf das Minimum, indem es nur die notwendigsten Funktionen zur Ausführung der Anwendung bereitstellt. |

16.2.5 Volume Hardening

Storage-basierte Angriffe in Container-Umgebungen beziehen sich auf Angriffe, bei denen ein Angreifer Zugriff auf den persistenten Speicher eines Containers erlangt, um sensible Daten zu stehlen, zu ändern oder zu löschen. Da Container häufig gemeinsam genutzte virtualisierte Storage-Ressourcen nutzen, können Schwachstellen in der Storage-Konfiguration zu solchen Angriffen führen.

Storage-basierte Angriffe in Container-Umgebungen nutzen oft eine der folgenden Schwachstellen aus:

- **Unzureichende Zugriffskontrollen:** Wenn die Zugriffskontrollen für den persistenten Speicher nicht korrekt konfiguriert sind, kann ein Angreifer möglicherweise auf Daten anderer Container oder Benutzer zugreifen, indem er diese mounten kann. Insbesondere NFS-basierte Storagesysteme sind hierfür sehr anfällig, da diese oft IP-basierten Zugriff gewähren (und alle Pods auf einem Node nutzen aus Sicht eines NFS-Servers die IP-Adresse des Nodes).
- **Schwache Verschlüsselung:** Wenn Daten im persistenten Speicher unzureichend verschlüsselt sind oder keine Verschlüsselung verwendet wird, kann ein Angreifer, der es schafft, ein Volume zu mounten, auf vertrauliche Informationen zugreifen.
- **Denial-of-Service-(DoS-)Angriffe:** Ein Angreifer kann absichtlich große Datenmengen in den persistenten Speicher schreiben, um den verfügbaren Speicherplatz zu erschöpfen und einen Denial-of-Service-Angriff gegen andere Container durchzuführen.

Diese Angriffsvektoren lassen sich vor allem durch die Nutzung von Storage-Klassen (siehe *Abschnitt 9.3.8*) und Storage-Systemen entschärfen, die Quotas durchsetzen und Inhalte verschlüsseln können. Es empfiehlt sich ferner, Pods nur das Mounten von PersistentVolumeClaims (PVCs), aber nicht das Mounten von Volumes zu gestatten. Andernfalls könnten Pods bspw. einen möglichst in der Verzeichnishierarchie hohen NFS-Mountpunkt raten und damit dann ganze Verzeichnisse unterhalb dieses Mountpunkts einsehen. Das könnten im schlimmsten Fall alle Verzeichnisse von Volumes anderer Nutzer eines Clusters sein, die über einen NFS Volume Provisioner bereitgestellt wurden.

Mittels der Einschränkung des Mountings von PVCs hat man hier mehr Kontrolle, und Pods können nur auf die Volumes zugreifen, die auch mittels eines PVC angefordert wurden. Namespaces, die als „Restricted“ gekennzeichnet sind (vgl. Abschnitt 16.2.2), gehen genauso vor und setzen so eine bessere Storage Isolation durch.

Um Storage-basierten Angriffen in Container-Umgebungen entgegenzuwirken, sollten daher folgende Maßnahmen ergriffen werden.

- **Zugriffskontrollen implementieren:** Es ist wichtig, Zugriffsrichtlinien und Quotas für den persistenten Speicher zu definieren und durchzusetzen. Stellen Sie sicher, dass nur berechtigte Container oder Benutzer auf die entsprechenden Speicherressourcen zugreifen können.
- **Datenverschlüsselung:** Verschlüsseln Sie sensible Daten im persistenten Speicher, um sicherzustellen, dass sie nicht von unbefugten Personen gelesen werden können, selbst wenn der Speicher kompromittiert wird.
- **Isolation:** Stellen Sie sicher, dass Container in isolierten Umgebungen laufen und nur die Berechtigungen haben, auf den Speicher zuzugreifen, den sie benötigen. Container sollten nicht in der Lage sein, auf den Speicher anderer Container zuzugreifen. Dies lässt sich bspw. wie gezeigt mit PVCs und als „Restricted“ gelabelten Namespaces erreichen. Es gibt dabei verschiedene Storage-Systeme und -Lösungen, die sowohl Zugriffskontrollen, Quotas und auch die Verschlüsselung von Volumes in Kubernetes unterstützen.
- **OpenEBS** ist eine Open-Source-Storage-Lösung für Kubernetes, die die Verschlüsselung von Volumes auf Storage-Ebene unterstützt. Entsprechende Einstellungen lassen sich in der OpenEBS-Konfiguration vornehmen, um die Verschlüsselung für bereitgestellte Volumes zu aktivieren.
- **Portworx** ist eine Kubernetes-native Storage-Plattform, die eine umfassende Palette an Storage-Funktionen bietet, darunter auch die Verschlüsselung von Volumes. Auch mit Portworx lässt sich die Verschlüsselung für persistente Volumes aktivieren und die erforderlichen Schlüssel verwalten.
- **Azure Disk Encryption:** Wenn Kubernetes auf der Azure-Cloud-Plattform ausgeführt wird, kann man die Azure Disk Encryption-Funktion verwenden, um die Verschlüsselung von persistenten Volumes zu aktivieren. Dadurch werden die Daten auf der Azure-Disk-Ebene verschlüsselt und können sicher in Kubernetes-Clustern gespeichert werden.
- **Amazon EBS Encryption:** Für Kubernetes-Cluster, die auf der Amazon Web Services-(AWS-)Plattform betrieben werden, lässt sich die Amazon Elastic Block Store-(EBS-)Verschlüsselung verwenden, um die Verschlüsselung von persistenten Volumes zu aktivieren. Dies ermöglicht die Verschlüsselung der Daten auf der EBS-Ebene.

- **Google Cloud Storage Encryption:** Auch auf der Google Cloud Platform (GCP) kann eine entsprechende Google Cloud Storage Encryption verwendet werden, um die Verschlüsselung von persistentem Speicher in Kubernetes zu aktivieren. Dies stellt sicher, dass die Daten auf der Storage-Ebene verschlüsselt werden.

Diese Liste stellt nur einige Beispiele dar und ist nicht vollständig.



Verschlüsselung von Volumes am Beispiel der Storage-Lösung OpenEBS

Das folgende Beispiel zeigt, wie einfach sich bspw. die Verschlüsselung von Volumes in Kubernetes mit der Storage-Lösung OpenEBS aktivieren lässt. Wir gehen von einer funktionstüchtigen OpenEBS-Installation in einem Cluster aus. Man befolge hierzu die Installationsanleitung von OpenEBS, um es in einem Kubernetes-Cluster einzurichten. Dazu gehört das Hinzufügen der OpenEBS-Repositories, das Installieren des OpenEBS-Operators und das Konfigurieren der OpenEBS-Konfigurationsdateien.

Um für einen Storage Pool in OpenEBS die Verschlüsselung zu aktivieren, muss nur folgende OpenEBS-spezifische Custom Ressource-Definition im Cluster angewendet werden:

Listing 16.30 Aktivierung der Verschlüsselung für einen OpenEBS Storage Pool

```
apiVersion: openebs.io/v1alpha1
kind: StoragePoolClaim
metadata:
  name: encrypted-disk-pool
spec:
  pools:
    - name: cstor-pool
      encrypted: true                                # Verschlüsselung aktivieren
```

Verschlüsselte Volumes können dann ganz normal und einfach mittels eines PVC und der Angabe der entsprechenden Storage-Klasse angefordert werden. Für die Anwendung ist die Verschlüsselung also vollkommen transparent. Sollten jedoch verschlüsselte Volumes von anderen Pods gemountet werden (oder anderweitig in fremde Hände fallen), können diese damit nichts anfangen.

Listing 16.31 Anforderung eines verschlüsselten OpenEBS Volumes mittels eines PVC

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-encrypted-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: encrypted-disk-pool
```

16.2.6 Workload Policing

Die bislang in diesem Kapitel gezeigten Maßnahmen sind hilfreich, um die Sicherheit containerisierter Workloads in Container-Plattformen zu erhöhen. Sie funktionieren aber natürlich nur, wenn deren Anwendung auch durchgesetzt werden kann. Kubernetes-Bordmittel unterstützen allerdings nicht die Durchsetzung aller genannten Maßnahmen von sich aus. Daher sind ergänzende Lösungen erforderlich.

Sowohl in Kubernetes als auch in anderen Container-Plattformen können und sollten daher zur Durchsetzung von Vorgaben sogenannte Policies verwendet werden, mittels derer Richtlinien für die Verwaltung, Sicherheit und Kontrolle von Ressourcen innerhalb eines Clusters festgelegt werden können. Policies definieren, wie bestimmte Aktionen oder Zustände behandelt werden sollen, und können dabei verschiedene Aspekte abdecken:

- **Zugriffskontrolle:** Policies werden verwendet, um den Zugriff auf Kubernetes-Ressourcen zu steuern. Sie definieren, welche Benutzer, Dienstkonten oder Gruppen Zugriff auf bestimmte Ressourcen haben dürfen und welche Berechtigungen ihnen gewährt werden. Kubernetes unterstützt dies mit dem (allerdings sehr komplexen rollenbasierten Zugriffsmodell) RBAC (vgl. *Abschnitt 9.3.9.1*).
- **Ressourcenverwaltung:** Policies können die Ressourcennutzung in einem Kubernetes-Cluster verwalten. Policies können beispielsweise festlegen, wie viele Ressourcen (wie CPU oder Speicher) ein bestimmter Pod verwenden darf oder wie viele Pods in einem Namespace erstellt werden dürfen. Kubernetes unterstützt dies mittels Quotas, LimitRanges, Requests und Limits (vgl. *Abschnitt 9.3.9.2*).
- **Netzwerksicherheit:** Policies ermöglichen die Definition von Netzwerkeinschränkungen innerhalb des Clusters. Diese können den ein- und ausgehenden Datenverkehr zwischen Pods oder Namespaces steuern, um Sicherheitsrichtlinien wie den Zugriff auf bestimmte Ports oder IP-Bereiche zu gewährleisten. Kubernetes unterstützt dies mit Network Policies (vgl. *Abschnitt 9.3.9.3* und *Abschnitt 16.2.2*).
- **Konfigurationsvalidierung:** Policies können dazu dienen, sicherzustellen, dass Kubernetes-Konfigurationsdateien den definierten Standards und Best Practices entsprechen. Derartige Policies prüfen die Konfiguration auf Fehler, Sicherheitslücken oder potenziell problematische Einstellungen (z. B. nicht gesetzte Ressourcenlimits, die im Rahmen von DoS-Attacken ausgenutzt werden könnten). Dieses Feld ist allerdings so weit, dass Kubernetes von Haus aus hier keine Unterstützung anbietet.
- **Compliance und Governance:** Policies helfen dabei, die Einhaltung von Richtlinien, Standards und Vorschriften sicherzustellen. Sie können sicherstellen, dass Sicherheitsrichtlinien, Datenschutzanforderungen oder andere Compliance-Richtlinien im Cluster durchgesetzt werden. Auch dieses Feld ist allerdings so umfangreich, dass Kubernetes von Haus aus hier keine Unterstützung anbietet.

Solche und weitere Policies können auf verschiedene Arten in Kubernetes durchgesetzt werden, z. B. mithilfe von Tools wie dem Open Policy Agent (OPA), Kubernetes Network Policies, Admission Controllern oder speziellen Sicherheitslösungen. Sie bieten eine granulare und flexible Möglichkeit, die Verwaltung und Sicherheit von Kubernetes-Clustern zu steuern und sicherzustellen, damit diese den erforderlichen Anforderungen entsprechen.

Die beiden folgenden Beispiele zeigen exemplarisch, wie diese Art von Policies eingesetzt werden können, um sehr spezifische Sicherheitsrichtlinien formulieren und automatisiert durchsetzen zu können.

Policy-Beispiel: Container-Images nur aus zugelassenen Registries zulassen

Listing 16.32 zeigt illustrativ eine Richtlinie für das Open Policy Agent-System, mit der formuliert und durchgesetzt werden kann, dass Container-Images von Pods nur aus einer definierten Registry stammen dürfen. Derartig feingranulare Regeln sind mit Kubernetes-Bordmitteln nicht ausdrückbar und wären nur durch aufwendiges und fehleranfälliges Skripting realisierbar.

Listing 16.32 OPA-Policy zur Durchsetzung der Image-Herkunft aus einer vertrauenswürdigen Registry

```
package kubernetes.admission

import future.keywords

deny contains msg if {
    input.request.kind.kind == "Pod"
    some container in input.request.object.spec.containers
    image := container.image
    not startswith(image, "registry.my.company.com/")
    msg := sprintf("image '%s' comes from untrusted registry", [image])
}
```

Policy-Beispiel: Mehrfache Vergabe von DNS-Namen in Ingress-Ressourcen vermeiden

Und folgendes Listing 16.33 zeigt, wie sich das Problem der DNS-Namenskollisionen bei Ingress-Ressourcen (vgl. Abschnitt 9.3.6) einfach mittels einer Richtlinie erkennen und damit unterbinden lässt. Eine Namenskollision bei Ingress-Ressourcen tritt auf, wenn mehrere Ingress-Routen den gleichen Hostnamen verwenden. Ingress ist ein Kubernetes-Objekt, das den Zugriff auf Dienste innerhalb des Clusters über externe HTTP- und HTTPS-Routen ermöglicht. Jeder Ingress kann dabei in einem Namespace angelegt werden und einen Hostnamen haben, der verwendet wird, um den eingehenden Datenverkehr einem bestimmten Dienst zuzuordnen. Wenn mehrere Ingress-Ressourcen (auch aus unterschiedlichen Namespaces) den gleichen Hostnamen haben, führt dies zu einer Namenskollision.

Eine Namenskollision kann unerwünschte Auswirkungen haben, da sie dazu führen kann, dass der eingehende Datenverkehr nicht richtig an den beabsichtigten Dienst weitergeleitet wird. Dies kann zu Fehlern, Konflikten oder Inkonsistenzen führen. Solche Namenskollisionen über mehrere Namespaces können unbeabsichtigt geschehen oder in Multi-Tenancy Clustern von böswilligen Akteuren auch absichtsvoll eingerichtet werden, um eingehenden Traffic (zumindest anteilig) an eigene Services umzuleiten, um dort beliebige Daten abgreifen zu können. Ob nun unbeabsichtigt oder böswillig, in beiden Fällen wird der Datenverkehr gestört. Daher sind solche Situationen zu vermeiden, können aber mit Kubernetes-Bordmitteln nicht wirkungsvoll erkannt oder gar aufgelöst werden.

Es lässt sich aber recht problemlos eine OPA-Richtlinie definieren, um Namenskollisionen bei Ingress-Routen zu erkennen. Auf diese Weise stellt man sicher, dass jeder Hostname nur einmal verwendet und dass der Datenverkehr korrekt an den entsprechenden Service weitergeleitet wird. Bei der Reservierung von DNS-Namen in einem Cluster gilt dadurch die First-Come-First-Served-Regel. Alle weiteren Ingress-Reservierungen mit demselben DNS-Namen aus dem gleichen oder anderen Namespaces würden abgelehnt werden.

Listing 16.33 OPA-Policy zur Durchsetzung unterschiedlicher Host-Namen in Ingress-Ressourcen

```
package kubernetes.admission

import future.keywords

# Alle Ingress mit demselben Host wie ein bestehender Ingress zurückweisen
deny contains msg if {
    input.request.kind.kind == "Ingress"

    # Iteriere über alle Hosts im Input Ingress
    some rule in input.request.object.spec.rules
    newhost := rule.host

    # Iteriere über alle vorhandenen Eingänge und alle Hosts für jeden Eingang
    some namespace, name
    some oldrule in data.kubernetes.ingresses[namespace][name].spec.rules
    oldhost := oldrule.host

    # Ablehnen, wenn der alte und der neue Host identisch sind
    newhost == oldhost
    msg := sprintf("ingress host conflicts with ingress %s/%s", [namespace, name])
}
```

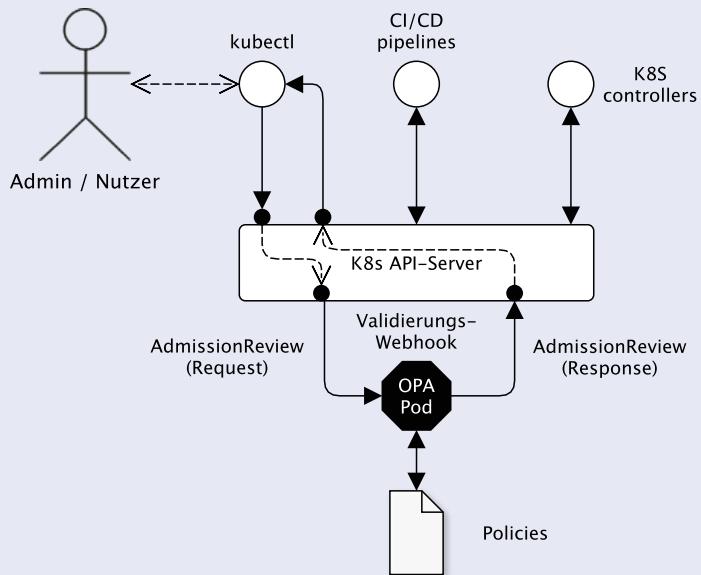


Open Policy Agent (OPA)

Der Open Policy Agent (OPA) ist eine Open-Source-Software, die als plattform-unabhängiger, allgemeiner „Richtlinienmotor“ entworfen wurde. Die Open Policy Agent-Initiative ist ein Projekt der Cloud Native Computing Foundation (CNCF) und hat eine wachsende Community von Entwicklern und Benutzern. OPA bietet eine deklarative Sprache und ein Framework zur Definition, Evaluierung und Durchsetzung von Richtlinien in verschiedenen Anwendungen und Infrastrukturen. OPA ist grundsätzlich unabhängig von spezifischen Domänen oder Plattformen und kann in einer Vielzahl von Umgebungen eingesetzt werden, darunter Cloud-Umgebungen und Container-Orchestrationsplattformen wie Kubernetes. OPA ist aber nicht auf Kubernetes beschränkt.

Der Kern des OPA ist die sogenannte „Rego“-Sprache, die eine deklarative und logische Syntax verwendet (Beispiele finden sich in Listing 16.32 und Listing 16.33). Mit Rego können Entwickler und Administratoren komplexe Richtlinien in einem kompakten Format formulieren. Diese Richtlinien definieren, was erlaubt oder verboten ist und welche Entscheidungen bei bestimmten Anfragen oder Aktionen getroffen werden sollen.

In Kubernetes kann ein Open Policy Agent bspw. in Form des OPA-Gatekeepers mittels Helm-Charts installiert und als Validierungswebhook registriert werden. Anfragen an den Kubernetes API-Server werden dann über den Validierungswebhook an einen OPA-Pod weitergeleitet, der die Einhaltung der definierten Rego-Regeln prüft. Werden Regelverstöße bei Ressourcen erkannt, werden diese nicht angelegt, sondern mit einer entsprechenden Fehlermeldung der Kubernetes API beantwortet. Die zu prüfenden Rego-Regeln können bspw. mittels ConfigMaps oder Secrets dem OPA-Agenten in Kubernetes bereitgestellt werden.



OPA kann dabei in verschiedenen Anwendungsfällen eingesetzt werden, einschließlich Zugriffskontrolle, Validierung von Konfigurationen, Netzwerksicherheit und Compliance.

Insgesamt sind die Installation und Einrichtung von OPA allerdings nicht trivial, sodass wir hier auf die begleitenden und vertiefenden Materialien auf der Website zu diesem Buch und die original OPA-(Gatekeeper)-Dokumentation verweisen. Der Leser findet dort entsprechende Links und Hinweise zur Installation und Konfiguration von OPA in Kubernetes, ebenso wie umfangreiche Erläuterungen, wie Rego-Richtlinien formuliert werden können.

16.2.7 Intrusion Detection

Der Kubernetes Hardening Guide der NSA (NSA 2022) empfiehlt grundsätzlich das Logging interner Kubernetes-Events. Der bereits in Abschnitt 16.1.4.3 gezeigte Ansatz des Log-Forwardings trägt auch zur Intrusion Detection auf Container-Plattform-Ebene bei. Um ein klares Bild und eine Zuordnung der im Cluster auftretenden Ereignisse zu erhalten, ist es jedoch

- wichtig zu wissen, woran Bedrohungen in Logs erkannt werden können, um ein diesbezügliches automatisches Alerting einzurichten,
- sowie zu wissen, was überhaupt geloggt werden sollte und wie Container-Plattformen (insbesondere Kubernetes) entsprechend konfiguriert werden müssen.

Kubernetes ist grundsätzlich in der Lage, Audit-Protokolle zu erfassen, um zugewiesene Cluster-Aktionen zu verfolgen und grundlegende Informationen zur CPU- und Speicher Nutzung zu überwachen; es bietet jedoch von Haus aus keine umfassenden Überwachungs- oder Alarmierungsdienste. Entsprechende Audit-Protokolle erfassen u. a. alle Aktivitäten im Cluster. Eine effektive Protokollierungslösung trägt somit auch zur Sicherheit der Container-Plattform-Ebene bei.

Die Protokollierung sollte dabei auf allen Ebenen der Umgebung erfolgen, einschließlich des Hosts, der Anwendung, des Containers, der Container-Engine, der Image-Registrierung, des API-Servers und der Cloud, sofern zutreffend. Sobald diese Protokolle erfasst sind, sollten sie in einem einzigen Dienst zusammengefasst werden, um allen Beteiligten einen vollständigen Überblick über die in der gesamten Umgebung durchgeföhrten Aktionen zu geben.

Innerhalb der Kubernetes-Umgebung sollten unter anderem die folgenden Ereignisse überwacht und protokolliert werden (NSA 2022):

- API-Request-Historie
- Leistungsmetriken
- Deployments
- Ressourcenverbrauch
- Betriebssystemaufrufe
- Protokolle
- Änderungen von Berechtigungen
- Netzwerk-Traffic
- Pod-Skalierungen
- Volume Mounts
- Anpassungen an Images und Containern
- Erstellung und Änderung von geplanten Jobs (Cronjobs)

Das Streaming von Logs an einen externen Logging Service Provider kann ferner dazu beitragen, die Verfügbarkeit für Sicherheitsexperten außerhalb des Clusters zu gewährleisten, sodass diese Anomalien in Echtzeit erkennen können. Durch Nutzung von Third-Party Logging Services erschafft man sich natürlich auch wieder eine größere Angriffsfläche. Bei dieser Methode sollten daher die Logs verschlüsselt übertragen werden, um sicherzustellen, dass Cyber-Akteure während der Übertragung nicht auf die Protokolle zugreifen und wertvolle Informationen über die Umgebung gewinnen oder Logs kompromittieren können.



Aktivierung der Audit-Protokollierung in Kubernetes

Achtung! Die Audit-Protokollierungsfunktionen von Kubernetes sind (abhängig von der Distribution) oft standardmäßig deaktiviert.

Um die Audit-Protokollierung zu aktivieren, müssen Sie die kube-apiserver-Konfiguration anpassen. Hierzu muss die Datei `kube-apiserver.yaml` editiert werden. Die Bearbeitung der kube-apiserver-Konfiguration erfordert im Allgemeinen Administratorrechte.

```
sudo nano /etc/kubernetes/manifests/kube-apiserver.yaml
```

Ergänzen Sie folgenden Text in der `kube-apiserver.yaml`

```
--audit-policy-file=/etc/kubernetes/policy/audit-policy.yaml
--audit-log-path=/var/log/audit.log
--audit-log-maxage=1825
```

Wie schon in *Abschnitt 9.3.1* erläutert, befindet sich der kube-apiserver auf der Kubernetes-Kontrollebene und fungiert als Front-End, das interne und externe Anfragen für einen Cluster bearbeitet. Jede Anfrage, ob von einem Benutzer, einer Anwendung oder der Steuerebene generiert, erzeugt in jeder Phase ihrer Ausführung ein Audit-Ereignis. Wenn ein Audit-Ereignis registriert wird, sucht der kube-apiserver nach einer Audit-Policy-Datei und einer anwendbaren Regel. Wenn eine solche Regel existiert, protokolliert der Server das Ereignis auf der Ebene, die durch die erste übereinstimmende Regel definiert ist.

Hierzu muss eine YAML-Datei mit einer AuditPolicy erstellt werden, um die Regeln festzulegen und die gewünschte Audit-Ebene zu spezifizieren, auf der jede Art von Audit-Ereignis protokolliert werden soll. Regeln haben dabei eine der folgenden vier Prüfstufen:

- None
- Metadata
- Request
- RequestResponse

Durch die Protokollierung aller Ereignisse auf RequestResponse-Ebene erhalten die Administratoren im Falle einer Sicherheitsverletzung die größtmögliche Menge an Informationen, die sie den Einsatzkräften zur Verfügung stellen können. Dies kann jedoch dazu führen, dass Secrets in den Protokollen erfasst werden. Die NSA und CISA (Cybersecurity and Infrastructure Security Agency, US-Behörde) empfehlen daher, die Protokollierungsebene von Anfragen, die Secrets beinhalten, auf die Metadatenebene zu reduzieren, um die Erfassung von Geheimnissen in den Protokollen zu vermeiden (vgl. Listing 16.34).

Listing 16.34 Beispiel einer AuditPolicy

```
# Diese Prüfrichtlinie protokolliert Ereignisse, die Secrets betreffen,
# auf der Metadatenebene und alle anderen Ereignisse auf der RequestResponse-Ebene
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
  resources:
  - group: "" #Core API group
    resources: ["secrets"]
- level: RequestResponse
```

Wie diese Audit-Logs genutzt werden können, zeigt Tabelle 16.2.

Tabelle 16.2 Empfehlungen der NSA zur Threat Detection (NSA 2022)

| Angriffsvektor | Log Detection |
|--|---|
| Angreifer können versuchen, einen Pod oder Container bereitzustellen, um ihre eigene Schadsoftware auszuführen oder um ihn als Ausgangspunkt für ihren Angriff zu nutzen. Angreifer können versuchen, ihre Bereitstellung als legitimes Image zu tarnen, indem sie die Namen und Namenskonventionen kopieren. Sie können auch versuchen, einen Container mit Root-Rechten zu starten, um ihre Rechte zu erweitern. | Filtern Sie auf atypische Pod- und Container-Installationen. Filtern Sie auf Image-IDs und Layer-Hashes für Vergleiche von verdächtigen Images. Filtern Sie Pods oder Anwendungscontainer, die mit Root-Rechten gestartet werden. |
| Angreifer können versuchen, ein bösartiges Image in eine Registry zu importieren; entweder um sich selbst Zugang zu ihrem Image für die Bereitstellung zu verschaffen oder um legitime Parteien dazu zu bringen, ihr bösartiges Image anstelle des legitimen Images zu installieren. | Dies lässt sich in den Protokollen der Container-Engine oder des Image-Repositorys erkennen. Man sollte dabei Abweichungen vom Standard-Bereitstellungsprozess von Images untersuchen. |
| Wenn es einem Angreifer gelingt, eine Anwendung so weit auszunutzen, dass er Befehlausführungsfähigkeiten auf dem Container erlangt, kann er je nach Konfiguration des Pods API-Anfragen aus dem Pod herausstellen, wodurch er möglicherweise seine Privilegien ausweiten, ein Lateral Movement begehen oder sogar aus dem Container auf den Host ausbrechen kann. | Man sollte daher auf ungewöhnliche API-Anfragen (aus den Kubernetes-Audit-Protokollen) oder ungewöhnliche Systemaufrufe (aus den Seccomp-Protokollen), die aus dem Inneren eines Pods stammen, filtern. |

Tabelle 16.2 (Fortsetzung) Empfehlungen der NSA zur Threat Detection (NSA 2022)

| Angriffsvektor | Log Detection |
|---|---|
| Angreifer, die sich zunächst Zugang zu einem Kubernetes-Cluster verschafft haben, werden wahrscheinlich versuchen, weiter in den Cluster einzudringen, was eine Interaktion mit dem Kube-Apiserver erfordert. | Während Angreifer versuchen herauszufinden, welche anfänglichen Berechtigungen vorliegen, kann es passieren, dass sie mehrere fehlgeschlagene Anfragen an den API-Server stellen. Wiederholte fehlgeschlagene API-Anfragen und Anfragemuster, die für ein bestimmtes Konto untypisch sind, sind ein Warnsignal. |
| Angreifer können versuchen, einen Cluster zu kompromittieren, um die Ressourcen des Opfers zu nutzen, um bspw. ihren eigenen Cryptominer auszuführen (d. h. ein Cryptojacking-Angriff). | Würde ein Angreifer erfolgreich einen Cryptojacking-Angriff starten, würde dies wahrscheinlich in den Protokollen als plötzlicher Anstieg des Ressourcenverbrauchs angezeigt. |
| Angreifer können versuchen, einem Container, den sie kompromittiert haben oder gerade erstellen, einen Volume Mount hinzuzufügen, um Zugriff auf den Host zu erhalten. | Aktionen zu Volume Mounts sollten genau auf Anomalien hin überwacht werden (insbesondere HostPath Volumes). |
| Angreifer, die in der Lage sind, CronJobs zu erstellen, können versuchen, automatisch und wiederholt Malware auf dem Cluster auszuführen. | Die Einrichtung und Änderung von CronJobs sollte daher genau überwacht werden. |

16.2.8 Sicherung der Supply Chain

Während bislang alle Maßnahmen dazu dienten, Angriffsvektoren abzuwehren, die Schwachstellen in Software oder fehlerhaften Konfigurationen ausnutzen, hat der vierte Angriffsvektor aus Bild 16.2, zum Ziel, die Software Supply Chain anzugreifen, um Schwachstellen in zu deployende Systeme einzubauen, die dann von den anderen Angriffsvektoren ausgenutzt werden können. Zum einen kann man sich dagegen durch die Infrastrukturmaßnahmen des Abschnitts 16.1 absichern. Doch ist dies nicht immer ausreichend, da sich Software auch über Import Dependencies von anderen Softwarepaketen abhängig macht, die außerhalb der Kontrolle der eigenen Infrastruktur liegen. Bekanntestes Beispiel der jüngeren Vergangenheit dürfte die Log4Shell-Schwachstelle gewesen sein (siehe Info-Kasten). Solche Arten von Angriffen mit indirektem Effekt nennt man Angriffe auf die Software Supply Chain. Dabei müssen Schwachstellen auf drei Ebenen entlang der Supply Chain erkannt und kontinuierlich überprüft werden, um zu vermeiden, dass entlang dieser Kette künstlich Schwachstellen durch Angreifer eingebracht werden können. Dies umfasst:

1. Vermeidung bzw. Detektion bekannter Schwachstellen in genutzten Abhängigkeiten (insb. Libraries und Container-Images, siehe Abschnitt 16.2.8.1)
2. Vermeidung bzw. Detektion von potenziellen Schwachstellen in eigener Software (siehe Abschnitt 16.2.8.2)

3. Betreiben eigener Container-Images auf einer kontinuierlich auf Schwachstellen überwachten Container-Plattform (siehe Abschnitt 16.2.8.3)



Log4Shell

„Log4Shell“ bezieht sich auf eine schwerwiegende Sicherheitslücke (CVE-2021-44228) in der Java-Bibliothek „Log4j“ (Version 2 bis 2.14.1), die Ende 2021 entdeckt wurde. Log4j ist eine weit verbreitete Logging-Bibliothek, die in vielen Java-Anwendungen verwendet wird.

Die Log4Shell-Lücke ermöglicht es Angreifern, beliebigen Java-Code auf einem betroffenen Server auszuführen, indem sie speziell gestaltete Eingaben an die Anwendung senden. Der Kern des Problems liegt in der Art und Weise, wie Log4j JNDI-Lookups (Java Naming and Directory Interface) verarbeitet. Durch die Ausnutzung dieser Schwachstelle können Angreifer Schadcode in Log-Meldungen einfügen und den Server dazu bringen, diesen Schadcode auszuführen.

Diese Sicherheitslücke verursachte großes Aufsehen, da Log4j in zahlreichen Anwendungen und Diensten integriert ist und Logging-Bibliotheken eigentlich sicherheitstechnisch als vollkommen „unbedenklich“ galten (es sei denn, man protokolliert Passwörter oder Zugangstokens etc.). Viele Unternehmen mussten schnell reagieren, um ihre Systeme zu patchen oder Workarounds zu implementieren, um die Ausnutzung dieser Schwachstelle zu verhindern.

16.2.8.1 Statische Software Composition Analysis (SCA)

Unter Software Composition Analysis (SCA) versteht man einen Prozess zur Identifizierung und Bewertung von Software-Komponenten in einer Anwendung. Dieser Prozess beinhaltet die Analyse von Open-Source-Komponenten, kommerziellen Bibliotheken und anderen Drittanbieter-Komponenten, die in einer Anwendung verwendet werden. Das Hauptziel von SCA besteht darin, potenzielle Sicherheitsrisiken, Compliance-Verletzungen und Lizenzkonflikte in den verwendeten Komponenten zu erkennen (die bspw. entlang des dritten Angriffsvektors aus Bild 16.2 platziert worden sein könnten). Insbesondere bei der Entwicklung von Cloud-nativen Anwendungen spielen dabei auch Container eine zunehmend wichtigere Rolle. Container ermöglichen es einerseits Entwicklern, Anwendungen in einer konsistenten und isolierten Umgebung auszuführen, unabhängig von der zugrunde liegenden Infrastruktur. Andererseits können Container jedoch auch Sicherheitsrisiken mit sich bringen, insbesondere wenn sie auf unsicheren oder nicht verifizierten Basis-Images basieren.

Durch die Integration von SCA in den Entwicklungsprozess von Cloud-nativen Anwendungen können potenzielle Sicherheitslücken in den verwendeten Komponenten identifiziert werden. SCA-Tools analysieren die Container-Images und deren zugrunde liegende Schichten, um bekannte Schwachstellen in den verwendeten Komponenten aufzudecken und damit zur Sicherheit von Cloud-nativen Anwendungen beitragen, indem es Folgendes ermöglicht:

- **Schwachstellen-Scanning:** SCA-Tools durchsuchen Datenbanken (CVE, siehe Info-Kasten) mit bekannten Sicherheitslücken und Schwachstellen, um potenzielle Risiken in den verwendeten Komponenten zu erkennen. Dadurch können Entwickler frühzeitig auf diese Lücken aufmerksam gemacht werden und Maßnahmen ergreifen, um sie zu beheben oder zu vermeiden.

- **Compliance-Überprüfung:** SCA kann auch bei der Einhaltung von Compliance-Richtlinien und -Lizenzen helfen. Es identifiziert Komponenten mit Lizenzkonflikten oder solchen, die gegen interne Richtlinien verstößen. Dadurch können Unternehmen rechtliche Probleme und Verstöße gegen Open-Source-Lizenzen vermeiden.
- **Aktualisierungsmanagement:** SCA-Tools ermöglichen die Überwachung von Komponenten auf verfügbare Updates und Patches. Durch regelmäßige Aktualisierungen können Sicherheitslücken behoben und die Anwendung vor bekannten Bedrohungen geschützt werden.
- **Risikobewertung:** SCA hilft bei der Bewertung des Risikos, das von bestimmten Komponenten ausgeht. Es bietet Informationen über die Schwere von Sicherheitslücken und die potenziellen Auswirkungen auf die Anwendung, um Prioritäten bei der Behebung von Schwachstellen zu setzen.

Durch die Integration von SCA in den Entwicklungsprozess von Cloud-nativen Anwendungen kann man so sicherstellen, dass Anwendungen auf einer sicheren Basis aufbauen und eine proaktive Identifizierung und systematische Behebung von Sicherheitslücken ermöglichen. Dies trägt dazu bei, die Risiken im Zusammenhang mit der Verwendung von Drittanbieter-Komponenten und Containern zu minimieren.



Tipp

Es bietet sich an, insbesondere das Schwachstellen-Scanning in Deployment-Pipelines (vgl. Kapitel 6) zu integrieren und automatisiert sowie periodisch auszuführen.



Common Vulnerabilities and Exposures (CVE)

CVE steht für Common Vulnerabilities and Exposures (Bekannte Schwachstellen und Expositionen). Es handelt sich um eine öffentlich zugängliche Liste von Informationen zu bekannten Sicherheitslücken und Schwachstellen in Software und Hardware. Das CVE-System wurde entwickelt, um eine standardisierte Methode zur Identifizierung, Verfolgung und Berichterstattung über Sicherheitslücken bereitzustellen.

Jede CVE-ID ist ein eindeutiger Bezeichner für eine bestimmte Schwachstelle. Sie besteht aus dem Präfix „CVE-“, gefolgt von einer eindeutigen Nummer, zum Beispiel „CVE-2021-1234“. Jede CVE-ID wird in einer zentralen Datenbank gespeichert und enthält Informationen über die betroffenen Produkte, die Schwere der Schwachstelle, eine Beschreibung des Problems und gegebenenfalls weitere Details.

Das CVE-System ermöglicht es Sicherheitsforschern, Herstellern und Anwendern, über bekannte Schwachstellen zu kommunizieren und Informationen auszutauschen. Wenn eine Sicherheitslücke entdeckt wird, kann sie dem CVE-System gemeldet und mit einer eindeutigen CVE-ID versehen werden. Dies ermöglicht eine standardisierte Referenzierung und erleichtert den Austausch von Informationen über die Schwachstelle.

CVE-IDs werden von der CVE-Mitgliedschaftsorganisation (CVE Numbering Authority) zugewiesen und verwaltet. Diese Organisation überwacht die Einreichung von Schwachstellen und vergibt die eindeutigen IDs. Sicherheitsforscher, Unternehmen und andere Beteiligte können Schwachstellen beim CVE-System melden, um sie der Liste hinzuzufügen.

Die Verwendung von CVE-IDs ist weit verbreitet und wird von Sicherheitsorganisationen, Herstellern, Sicherheitsforschern und in Sicherheitsdatenbanken verwendet. Die CVE-Datenbank ist öffentlich zugänglich und kann über verschiedene Quellen abgerufen werden.

- National Vulnerability Database (NVD): Die National Vulnerability Database, verwaltet vom National Institute of Standards and Technology (NIST) in den USA, ist eine umfassende Datenbank für Schwachstellen und Sicherheitslücken. Auf der Website des NIST (NVD CVE 2023) kann nach CVE-IDs und Schwachstellendetails gesucht werden.
- Mitre CVE-Liste: Mitre, die Organisation hinter dem CVE-System, stellt die offizielle CVE-Liste auf ihrer Website zur Verfügung. Auf die Mitre CVE-Liste kann über die Website des Mitre-CVE-Projekts (MITRE-CVE 2023) zugegriffen werden.
- Sicherheitstools: Diverse Sicherheits- und Schwachstellen Scanning-Tools wie bspw. Grype, berücksichtigen die CVE-Daten in ihren Repositorien und nutzen diese für Kritikalitätsbewertungen und Empfehlung von Gegenmaßnahmen.

Entsprechende Links finden sich auf der Website zum Buch.



Beispiel: Schwachstellen-Scanning mit Grype

Grype ist ein Schwachstellenscanner für Container-Images und Dateisysteme. Grype ist als Typvertreter zu verstehen. Andere containerbasierte Schwachstellenscanner funktionieren ähnlich. Weitere SCA-Tools finden sich auf der Website zum Buch.

Grype scannt den Inhalt eines Container-Images oder Dateisystems, um bekannte Sicherheitslücken (CVE) zu finden.

Unterstützte Linux-Distributionen:

Grype unterstützt insbesondere die folgenden Linux-Distributionen: Alpine, Amazon Linux, BusyBox, CentOS, Debian, Oracle Linux, Red Hat (RHEL) sowie Ubuntu und mehr.

Unterstützte Programmiersprachen und Paket-Manager:

Grype unterstützt die folgende Programmiersprachen- und Dependency Management-Systeme: Ruby (Gems), Java (JAR, WAR, EAR, JPI, HPI), JavaScript (NPM, Yarn), Python (Egg, Wheel, Poetry, requirements.txt/setup.py files), Dotnet (deps.json), Golang (go.mod), PHP (Composer) sowie Rust (Cargo)

Beispiel:

Alle bekannten Schwachstellen der Ubuntu 22.04-Distribution können bspw. wie folgt mittels Grype ermittelt werden.

Listing 16.35 Beispiel eines Schwachstellen-Scans mittels Grype

```
> grype ubuntu:22.04 --fail-on medium

✓ Vulnerability DB      [no update available]
✓ Loaded image
✓ Parsed image
✓ Cataloged packages    [101 packages]
✓ Scanning image...     [17 vulnerabilities]
  └─ 0 critical, 0 high, 7 medium, 7 low, 3 negligible
    └─ 2 fixed

NAME      INSTALLED      FIXED-IN      TYPE      VULNERABILITY      SEVERITY
bash      5.1-6ubuntu1      deb      CVE-2022-3715      Low
coreutils 8.32-4.1ubuntu1      deb      CVE-2016-2781      Low
gpgv      2.2.27-3ubuntu2.1      deb      CVE-2022-3219      Low
libc-bin   2.35-0ubuntu3.1      deb      CVE-2016-20013     Negl
libc6      2.35-0ubuntu3.1      deb      CVE-2016-20013     Negl
libcap2    1:2.44-1build3      1:2.44-1ubuntu deb      CVE-2023-2602      Low
libcap2    1:2.44-1build3      1:2.44-1ubuntu deb      CVE-2023-2603      Medium
libpcre3   2:8.39-13ubuntu0.22.04.1      deb      CVE-2017-11164     Negl
libsystemd0 249.11-0ubuntu3.9      deb      CVE-2023-31437     Medium
libsystemd0 249.11-0ubuntu3.9      deb      CVE-2023-31438     Medium
libsystemd0 249.11-0ubuntu3.9      deb      CVE-2023-31439     Medium
libudev1    249.11-0ubuntu3.9      deb      CVE-2023-31437     Medium
libudev1    249.11-0ubuntu3.9      deb      CVE-2023-31438     Medium
libudev1    249.11-0ubuntu3.9      deb      CVE-2023-31439     Medium
libzstd1    1.4.8+dfsg-3build1      deb      CVE-2022-4899      Low
login      1:4.8.1-2ubuntu2.1      deb      CVE-2023-29383     Low
passwd     1:4.8.1-2ubuntu2.1      deb      CVE-2023-29383     Low

1 error occurred:
  * discovered vulnerabilities at or above the severity threshold
```

Es bietet sich an, solche Schwachstellen-Scans in Deployment-Pipelines automatisiert auszuführen. Mittels des `--fail-on`-Schalters kann so sichergestellt werden, dass Schwachstellen nicht nur angezeigt werden, sondern die Pipeline auch bei einer erkannten Schwachstelle definierten Schweregrads abbricht.

16.2.8.2 Static Application Security Testing (SAST)

SAST steht für Static Application Security Testing und bezieht sich auf Testverfahren, die im Softwareentwicklungslebenszyklus eingesetzt werden, um potenzielle Sicherheitslücken und Schwachstellen in Anwendungen zu identifizieren. Im Kontext von SCA und der Sicherheit Cloud-nativer Anwendungen ist SAST eine weitere wichtige Methode zur Verbesserung der Sicherheit.

Während SCA sich auf die Analyse von Komponenten und deren Abhängigkeiten konzentriert, konzentriert sich SAST auf den Quellcode der Anwendung selbst. Es scannt den Quellcode statisch, ohne die Anwendung ausführen zu müssen, und identifiziert potenzielle Schwachstellen und Sicherheitslücken, die durch fehlerhafte Codierung oder unsichere Programmierpraktiken entstehen können. SAST-Tools durchsuchen dabei den Quellcode nach bestimmten Mustern, Anzeichen oder bekannten Schwachstellen, die auf Sicherheitsprobleme hinweisen.

Sie können beispielsweise nach unsicherer Datenvalidierung, unsicheren Datenbankabfragen, fehlerhafter Authentifizierung oder unsicherem Umgang mit sensiblen Daten suchen.

Die Integration von SAST in den Entwicklungsprozess ermöglicht es Entwicklern, potenzielle Sicherheitslücken frühzeitig zu erkennen und zu beheben, noch bevor die Anwendung bereitgestellt wird. Dadurch kann das Risiko von Sicherheitsverletzungen und Datenlecks minimiert werden.

SAST und SCA ergänzen sich gegenseitig in der Sicherheitsanalyse von Anwendungen. Während SCA sicherstellen kann, dass die Basis der eigenen Anwendung sicher ist, hilft SAST dabei sicherzustellen, dass die eigene Anwendung keine eigenen Schwachstellen einführt, obwohl auf einer sicheren und geprüften Basis aufgesetzt wird.



Tipp

SAST-Tools sind problemgegeben oft sehr sprachspezifisch. Daher ist es entscheidend, die Codebasis einer Cloud-nativen Anwendung mit geeigneten und sprachspezifischen SAST-Tools auf potenzielle Schwachstellen zu scannen. Es bietet sich dabei an, auch das SAST-Scanning (analog zum Schwachstellen-Scanning, SCA) in Deployment-Pipelines (vgl. Kapitel 6) zu integrieren und automatisiert bei jedem Build-Vorgang auszuführen. Es kann sinnvoll sein, mehrere SAST-Tools zu kombinieren.



Beispiel: SAST Scanning mit dem Python-Tool Bandit

Bandit ist ein Tool zur Detektion von potenziellen Sicherheitsproblemen in Python-Code. Bandit ist dabei bitte vom Leser wiederum als Typvertreter einer Klasse von SAST-Tools zu verstehen. Weitere SAST-Tools finden sich auf der Website zum Buch.

Im Rahmen eines Schwachstellen-Scans verarbeitet Bandit Python-Quellcode-Dateien, erstellt daraus einen Abstract Syntax Tree (AST) und führt geeignete Plug-ins gegen die AST-Knoten aus. Auf Basis dieser Analyse wird ein Bericht mit potenziellen Schwachstellen erstellt.

Bandit lässt sich dabei einfach mittels `pip install bandit` (oder anderen Python-Paketmanagern) installieren.

```
1. # example.py
2. import subprocess
3. user_input = input("Bitte geben Sie Ihren Namen ein: ")
4. cmd = "echo " + user_input
5. subprocess.call(cmd, shell=True) # Schwachstelle (potenzielle Shell Injection)
```

Oben gezeigtes Codebeispiel kann dabei wie folgt mittels Bandit auf Schwachstellen mit hohem Schweregrad gescannt werden:

```
> bandit example.py --severity-level high
```

Dies würde zu folgender (gekürzter) Ausgabe führen:

```
Test results:
>> Issue: [B602:subprocess_popen_with_shell_equals_true] subprocess call
with shell=True identified, security issue.
    Severity: High  Confidence: High
    CWE: CWE-78 (https://cwe.mitre.org/data/definitions/78.html)
    More Info: https://bandit.readthedocs.io/en/1.7.5/plugins/b602\_subprocess\_popen\_with\_shell\_equals\_true.html
    Location: example.py:5:0
4      cmd = "echo " + user_input
5      subprocess.call(cmd, shell=True)
```

In diesem veranschaulichenden Beispiel erkennt Bandit eine mögliche Sicherheitslücke in Zeile 4 und 5 und warnt vor der Verwendung der `subprocess.call()`-Funktion mit dem `shell=True`-Argument, da dies zu einer potenziellen Shell-Injection-Schwachstelle führen kann. Bandit bietet weitere Informationen zur Sicherheitslücke, einschließlich der Schwere sowie einem Link zur Dokumentation, der weitere Details zur spezifischen Sicherheitslücke bietet.

16.2.8.3 Kontinuierliches Schwachstellen-Scanning von Container-Plattformen

Während die SCA- und SAST-Maßnahmen den Vorteil haben, dass diese durch statische Analysen auf Basis einer bekannten Code-Basis bspw. im Rahmen von Deployment-Pipelines vorgenommen werden können, darf nicht außer Acht gelassen werden, dass sich damit wieder die Komponenten zur Laufzeit in der Laufzeit-Umgebung (zumeist Container-Plattformen wie Kubernetes) überwachen lassen, noch Fehlkonfigurationen außerhalb von Images identifizieren lassen.

In diesem Zusammenhang sind daher Tools hilfreich, die sich speziell auf Sicherheitsbewertung von Container-Plattform-Umgebungen spezialisieren. Diese helfen bei der Identifizierung von Sicherheitslücken, Compliance-Verstößen und Security Best Practices bspw. in Kubernetes-Infrastrukturen. Das Hauptziel ist es, Sicherheitsteams und Entwickler dabei zu unterstützen, potenzielle Sicherheitsprobleme in Container-Plattformen zu erkennen und zu beheben, bevor diese zu ernsthaften Sicherheitsvorfällen führen können. Hierzu führen diese Tools automatisierte Sicherheitsprüfungen durch und bieten eine umfassende Analyse der Sicherheitslage von Container-Plattformen. Typische Funktionen sind:

- **Cluster-Hardening-Checks:** Dabei werden die Sicherheitskonfiguration der Plattform, einschließlich der Überprüfung von Berechtigungen, Netzwerkeinstellungen, RBAC-Konfigurationen und anderen Sicherheitsaspekten geprüft und Konfigurationen identifiziert, die nicht den Best Practices entsprechen und potenzielle Sicherheitslücken verursachen könnten.
- **Compliance-Überprüfungen:** Die Überprüfung der Einhaltung von Sicherheitsstandards und Best Practices, wie z. B. CIS Kubernetes Benchmarks oder spezifischen Unternehmensrichtlinien. Dies umfasst Prüfungen, ob die Plattform die erforderlichen Sicherheitsmaßnahmen erfüllt, und identifiziert Abweichungen und potenzielle Compliance-Verstöße. In diesem Bereich gibt es Überschneidungen zu Policies (siehe Abschnitt 16.2.6).

- **Risikoanalyse:** Bewertung potenzieller Risiken und Schwachstellen der Container-Plattform hinsichtlich Schwere und potenzieller Auswirkungen. Dies ermöglicht es den Sicherheits-teams, Prioritäten für die Behebung von Sicherheitslücken zu setzen und ihre Ressourcen effektiv einzusetzen.
- **Container-Sicherheitschecks:** Durchführung Sicherheitsbewertungen für Container mittels Image Scans. Dabei werden Sicherheitskonfigurationen und Best Practices der Container-Images überprüft und potenzielle Schwachstellen und Risiken identifiziert. In diesem Bereich gibt es Überschneidungen zu SCA (siehe Abschnitt 16.2.8.1). Allerdings kann auf diese Art auch durch den Plattform-Betreiber potenziell schädliche Software auf der Plattform identifiziert (und gegebenenfalls deaktiviert) werden, selbst wenn die Entwicklungsteams keine kontinuierliche SCA oder SAST vornehmen (also bspw. seit dem letzten Build bzw. Deploy neue Schwachstellen entdeckt wurden). Ferner lässt sich mittels SCA oder SAST kein Fehler in der Konfiguration entdecken.



Kontinuierliches Schwachstellen-Scanning mittels Kubescape

Kubescape ist ein Open-Source-Tool, das speziell für die Sicherheitsbewertung von Kubernetes-Clustern entwickelt wurde. Der Leser möge Kubescape bitte als veranschaulichenden Typvertreter für Plattform Scanning-Tools verstehen. Andere Tools arbeiten ähnlich. Weitere Schwachstellen-Scanner finden sich auf der Webseite zum Buch.

Kubescape hilft bei der Identifizierung von Sicherheitslücken, Compliance-Verstößen und Best Practices in Kubernetes-Infrastrukturen. Kubescape kann als Teil des DevSecOps-Prozesses eingesetzt werden, um die Sicherheit von Kubernetes-Clustern und auf diesen ausgeführten Workloads kontinuierlich zu überwachen und sicherzustellen, dass diese den aktuellen Sicherheitsstandards entsprechen. Kubescape ermöglicht es Teams, Schwachstellen proaktiv zu erkennen und zu beheben, Compliance-Anforderungen zu erfüllen und die Sicherheit einer Kubernetes-Plattform und der auf ihr betriebenen Workloads zu stärken.

Das Scannen eines Clusters und der auf dem Cluster betriebenen Workloads ist mittels des Kommandozeilentools kubescape sehr einfach.

```
> kubescape scan --severity-threshold critical
```

Dies erzeugt eine Auswertung wie die folgende:

| Controls: 65 (Failed: 32, Passed: 20, Action Required: 13) Failed Resources by Severity: Critical - 0, High - 41, Medium - 314, Low - 102 | | | | | |
|--|--|------------------|---------------|----------------------|--|
| Severity | Control Name | Failed Resources | All Resources | % Compliance-Score | |
| Critical | Disable anonymous access to Kubelet service | 0 | 0 | Action Required * | |
| Critical | Enforce Kubelet client TLS authentication | 0 | 0 | Action Required * | |
| High | Forbidden Container Registries | 0 | 29 | Action Required ** | |
| High | Resource memory limit and request | 0 | 29 | Action Required *** | |
| High | Resource limits | 16 | 29 | 45% | |
| High | Applications credentials in configuration files | 3 | 189 | 97% | |
| High | List Kubernetes secrets | 17 | 183 | 83% | |
| High | HostNetwork access | 2 | 29 | 93% | |
| High | HostPath mount | 2 | 29 | 93% | |
| High | Resources CPU limit and request | 0 | 29 | Action Required ** | |
| High | Privileged container | 1 | 29 | 97% | |
| High | Workloads with Critical vulnerabilities exposed to external networks | 0 | 0 | Action Required *** | |
| High | Workloads with RCE vulnerabilities exposed to external networks | 0 | 0 | Action Required *** | |
| High | RBAC enabled | 0 | 0 | Action Required **** | |
| Medium | Exec into container | 4 | 183 | 96% | |
| Medium | Data Destruction | 10 | 183 | 98% | |
| Medium | Non-root containers | 28 | 29 | 3% | |
| Medium | Allow privilege escalation | 28 | 29 | 3% | |
| Medium | Ingress and Egress blocked | 28 | 29 | 3% | |
| Medium | Delete Kubernetes events | 4 | 183 | 96% | |
| Medium | Automatic mapping of service account | 58 | 94 | 38% | |
| Medium | Cluster-admin binding | 4 | 183 | 96% | |
| Medium | CoreDNS poisoning | 6 | 183 | 94% | |
| Medium | Malicious admission controller (mutating) | 1 | 1 | 0% | |
| Medium | Container hostPort | 2 | 29 | 93% | |
| Medium | Access container service account | 43 | 84 | 49% | |
| Medium | Cluster internal networking | 11 | 15 | 27% | |
| Medium | Linux hardening | 27 | 29 | 7% | |
| Medium | Configured liveness probe | 21 | 29 | 28% | |
| Medium | Portforwarding privileges | 5 | 183 | 95% | |
| Medium | No impersonation | 6 | 183 | 94% | |
| Medium | Secret/ETCD encryption enabled | 0 | 0 | Action Required **** | |
| Medium | Audit logs enabled | 0 | 0 | Action Required **** | |
| Medium | Containers mounting Docker socket | 1 | 29 | 97% | |
| Medium | Images from allowed registry | 0 | 29 | Action Required ** | |
| Medium | Workloads with excessive amount of vulnerabilities | 0 | 0 | Action Required *** | |
| Medium | CVE-2022-8492-cgroups-container-escape | 27 | 29 | 7% | |
| Low | Immutable container filesystem | 28 | 29 | 3% | |
| Low | Configured readiness probe | 22 | 29 | 24% | |
| Low | Kubernetes CronJob | 3 | 3 | 0% | |
| Low | Malicious admission controller (validating) | 1 | 1 | 0% | |
| Low | Network mapping | 11 | 15 | 27% | |
| Low | PSP enabled | 0 | 0 | Action Required **** | |
| Low | Label usage for resources | 20 | 29 | 31% | |
| Low | K8s common labels usage | 17 | 29 | 41% | |
| RESOURCE SUMMARY | | 117 | 398 | 56.19% | |
| FRAMEWORKS: AllControls (compliance: 55.60), NSA (compliance: 58.11), MITRE (compliance: 63.66) | | | | | |

Es bietet sich an, ein solches Scanning periodisch und kontinuierlich für eine Plattform durchzuführen und bei Verstößen ein entsprechendes Alerting einzurichten. Mittels des Parameters --severity-threshold kann bspw. in Pipelines festgelegt werden, ab wann eine Pipeline fehlschlägt.

■ 16.3 Zusammenfassung

Die Aufgabe, Cloud-native Anwendungen technisch sicher zu gestalten und vor Cyber-Angriffen zu schützen, umfasst verschiedene und aufeinander abzustimmende Aspekte, die im Rahmen der Sicherheit und Härtung von Infrastrukturen und containerisierter Workloads berücksichtigt werden müssen.

Ausgehend von einem Lebenszyklus-Modell von Cyber-Angriffen (Heartfield u. a. 2015, Aleroud u. a. 2017) sind die Phasen eines Angriffs für wirkungsvolle Gegenmaßnahmen zu berücksichtigen. Diese sollten darauf abzielen, lang andauernde Footholds zu verhindern und das Lateral Movement innerhalb von Infrastrukturen und Plattformen zu unterbinden. Dabei sind idealerweise potenzielle Angriffsvektoren von außen, gegen die Infrastruktur selbst, gegen verwendete Komponenten wie Image-Repositorien und gegen die Software Supply Chain konzeptionell zu berücksichtigen (Martin u. a. 2021) und zu entschärfen.

Im Hinblick auf die Härtung der Infrastrukturen werden bspw. verschiedene Maßnahmen durch die NSA empfohlen (NSA 2022). Systematisches tool-gestütztes System Hardening spielt dabei eine zentrale Rolle, um die Sicherheitskonfiguration der Systeme von Grund auf zu verbessern. Die kontinuierliche Aktualisierung virtueller Maschinen wird empfohlen, um bekannte Sicherheitslücken möglichst schnell zu schließen. Starke Authentifizierung ist ein weiterer wichtiger Aspekt, um den Zugriff auf Ressourcen zu beschränken. Die kontinuierliche Überwachung virtueller Maschinen mittels verhaltens- und signaturbasiertes Intrusion Detection sowie Log Forwarding ermöglicht die frühzeitige Erkennung und Reaktion auf Angriffe und vermeidet die Kompromittierung des „Cloud Forensic Trails“. Der Einsatz von Sicherheitsgruppen und Firewall-Lösungen trägt letztlich zur Netzwerksicherheit der Infrastruktur an sich bei.

Für containerisierte Workloads sind weitere spezifische Maßnahmen zur Härtung zu berücksichtigen (NSA 2022). Die Absicherung von Public Endpoints mittels Ingress-Ressourcen, einschließlich TLS-Terminierung und Authentifizierung mittels OAUTH2, trägt insbesondere zur Sicherheit der öffentlich zugänglichen Schnittstellen Cloud-nativer Anwendungen bei und soll Security Breaches bzw. Denial-of-Service-Angriffe erschweren oder unmöglich machen. Namespace- und Netzwerkisolierung helfen, die Abgrenzung und Sicherheit von Workloads zu gewährleisten, Workloads voneinander zu isolieren und das Lateral Movement zu erschweren. Die Härtung und Isolation von Pods kann durch die systematische Verwendung von Service-Accounts sowie dem Prinzip der geringst erforderlichen Rechte, die Limitierung von Ressourcen sowie die systematische Anwendung von Security Contexts und Security Policies erfolgen. Ferner kann die Container-Runtime-Umgebung durch erhöhte Isolationsmechanismen zusätzlich abgesichert werden. Ergänzend kann die Verschlüsselung von Volumes sensible Daten in containerisierten Umgebungen schützen und Datenausleitungen für Angreifer erschweren. Die Nutzung von Policies ermöglicht es, Richtlinien für die Verwaltung, Sicherheit und Kontrolle von Ressourcen innerhalb eines Clusters festzulegen und durchzusetzen. Audit-Protokollierung trägt zur Nachvollziehbarkeit von Aktivitäten bei. Die Sicherung der Software Supply Chain durch Software Composition Analysis, Static Application Security Testing und kontinuierliches Schwachstellen-Scanning von Laufzeitumgebungen wie Kubernetes ist ebenfalls von großer Bedeutung.

Die Sicherheit Cloud-nativer Anwendungen ist also im Sinne einer „Layered Defence“ nur durch komplementäre und vielschichtige Maßnahmen auf verschiedenen Ebenen der Infrastruktur, der Container-Plattform und containerisierten Workloads zu erzielen (vgl. Bild 16.3). Dabei erhebt dieses Kapitel keinen Anspruch auf Vollständigkeit, sondern soll nur einen Orientierung gebenden Überblick über die erforderlichen Maßnahmen auf unterschiedlichen Ebenen geben.

Ergänzende Materialien



<https://bit.ly/45KhXbH>

Auf der Website zum Buch finden sich zu diesem Kapitel ferner folgende Ergänzungsmaterialien:

- Übersichten und Links zu System Hardening sowie Intrusion Detection-Lösungen (Infrastruktur-Ebene)
- Übersichten und Links zu Container Runtime Environments mit erhöhter Isolation, verschlüsselbaren Storage-Lösungen, Policy-Systemen, Intrusion Detection-Lösungen (Container-Ebene), Software Composition Analysis-(SCA-)Tools, Static Application Testing-(SAST-)Tools sowie Container-Plattform-Schwachstellen-Scannern
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer

17

Regulatorische Anforderungen

„Ich bin für Regulierung, nicht für Strangulierung.“

Jean-Claude Juncker, ehemaliger Präsident der Europäischen Kommission (2014–2019)

Nach (Hennrich 2023) ist die Einhaltung datenschutzrechtlicher Regelungen im Cloud Computing nicht immer einfach, da im „Spannungsfeld des Datenschutzes“ zwei Welten aufeinanderprallen, die unterschiedlichen Mechanismen folgen.

- Auf der einen Seite die länderübergreifende Welt von Public-, Private-, Multi- und Hybrid-Cloud-Szenarien zahlreicher Provider, die unterschiedlichen Rechtsordnungen unterworfen sind und grundsätzlich eine globale Verteilung von IT-Ressourcen ermöglichen können.
- Auf der anderen Seite der auf EU-Recht und nationalen Rechtsordnungen geltende regulatorische Rahmen für Datenschutz, dessen Ziel und Zweck es ist, den Einzelnen vor einer missbräuchlichen Verwendung seiner personenbezogenen Daten im Sinne einer informationellen Selbstbestimmung zu schützen.

Die eine Seite ist dabei vom technisch Möglichen getrieben (der Autor rechnet sich dieser Seite zu), die andere Seite will die Gefahren des Datenmissbrauchs eindämmen. Dabei haben sich auf beiden Seiten einige Mythen gebildet, wie z. B. die Behauptung, dass die Nutzung von US-Cloud-Computing-Diensten zur Verarbeitung personenbezogener Daten grundsätzlich rechtlich nicht möglich sei; aber auch die Behauptung, die Datenschutzgrundverordnung (DSGVO) sei ein Hemmschuh für Innovation und Digitalisierung.

Die DSGVO hemmt erst einmal gar nichts, sondern regelt nur, wie mit personenbezogenen Daten umgegangen werden soll – nämlich sensibel und verantwortungsvoll. Das Gleiche erwarten wir wahrscheinlich auch von unserer Bank, wenn es um unser Geld geht. Also kein Grund zur Aufregung. Vor allem regelt die DSGVO nicht die Datenverarbeitung im Allgemeinen, sondern nur die Verarbeitung **PERSONENBEZOGENER** Daten. Wer also keine personenbezogenen Daten verarbeitet, ist von der DSGVO kaum betroffen. Oft besteht der Trick darin, auf die Erhebung personenbezogener Daten zu verzichten, um verschiedene regulatorische Hürden zu umgehen. Die DSGVO nennt dies Datensparsamkeit. Nur weil personenbezogene Daten oft einfach zu erfassen sind, bedeutet das nämlich nicht, dass man sie auch erfassen muss. Manchmal helfen einem diese Daten nämlich gar nicht weiter.

In diesem Kapitel sollen daher regulatorische Aspekte ein wenig beleuchtet und der eine oder andere Mythos hinterfragt und vor allem eingeordnet werden. Diese Darstellung hat im Wesentlichen Überblickscharakter und ist keinesfalls vollständig. Es wird grundsätzlich empfohlen, zu diesem Thema Fachliteratur wie z. B. (Hennrich 2023) zu konsultieren. Regulatorische Anforderungen sind nämlich ein Thema für sich und entfernen sich schnell

sehr weit von der technischen Sichtweise, die in diesem Buch bisher eingenommen wurde. Dennoch bietet es sich an, zumindest einen gewissen Überblick über die rechtlich nicht immer ganz einfache Lage zu gewinnen.

Wir gehen dazu in drei Schritten vor.

- Im Abschnitt 17.1 gehen wir insbesondere auf Compliance Reports von den großen internationalen und europäischen Cloud-Providern ein, um so zu ermitteln, welche regulatorischen Anforderungen überhaupt existieren.
- Ausgehend von diesem Blick in die Welt, widmen wir uns in Abschnitt 17.2 der DGSVO (international General Data Protection Regulation genannt, GDPR) und damit den regulatorischen Rahmenbedingungen der EU. Wir werden dabei insbesondere sehen, dass vor allem die Datenverarbeitung bzw. die Rechtslage hinsichtlich des Datenschutzes in Drittländern außerhalb der EU ein Problem sein kann.
- Der Abschnitt 17.3 geht daher abschließend darauf ein, welche Möglichkeiten und Optionen die DGSVO auch für diese Fälle grundsätzlich vorsieht.

■ 17.1 Cloud Compliance und Zertifizierungen

Auf Basis einer Analyse der folgenden fünf größten internationalen und der fünf größten europäischen Cloud-Anbieter (gemessen am Umsatz) wurden die von den Unternehmen angegebenen Zertifizierungen bzw. Konformitätserklärungen recherchiert. Auf diese sogenannten Compliances wird im Folgenden kurz zusammenfassend eingegangen, um einen Überblick über die regulatorischen Rahmenbedingungen im Cloud Computing zu geben. Dabei ergibt sich die Tabelle 17.1 anhand der Compliance-Angaben und Übersichten der Provider auf deren Webseiten.

Die Compliance-Auflistung zeigt, dass (bis auf Alibaba) alle Provider insbesondere die ISO 27001 + 27017/18-Reihe erfüllen und die amerikanischen Provider meist die umfangreichste Compliance-Abdeckung (auch für den deutschen bzw. europäischen Rechtsraum) bieten. Je kleiner die internationalen Hyperscaler (AWS, Azure, Google) werden, desto mehr Compliances berichten sie. Compliances scheinen also als ein Marktvorteil der kleineren Provider gesehen zu werden, um Marktanteile zu gewinnen. AWS berichtet weniger Compliance-Abdeckungen als Azure und Google. Und IBM scheint sich hier überwiegend auf den nordamerikanischen Markt zu konzentrieren.

Der größte Nicht-US-Hyperscaler Alibaba berichtet (zumindest auf seinen internationalen Seiten) allerdings so gut wie gar keine Compliances. Einzig und allein in Whitepapern werden Hinweise für Kunden gegeben, wie auch auf Alibaba-Infrastrukturen Compliance-konforme Anwendungen betrieben werden können. Insgesamt ist die Compliance-Situation bei Alibaba aber als wenig transparent einzuschätzen.

Die europäischen Provider (mit Ausnahme von SAP) scheinen sich auf die europäischen Anforderungen zu fokussieren und Compliance-Anforderungen des nordamerikanischen Marktes wie HIPAA, HITRUST, FIPS oder FedRAMP meist nicht auf dem Schirm zu haben.

Tabelle 17.1 Anzahl von Compliance-Zertifikaten großer internationaler und europäischer Cloud-Provider im Vergleich (eigene Angaben der Provider auf deren Compliance-Webseiten, Stand Mai 2023)

| Provider | Land | Zertifikate | Anmerkungen |
|-----------------------|-------|-------------|--|
| Amazon Web Services | USA | 75 | Internationaler Fokus, weltweit regionenbezogene Datenschutzstandards |
| Microsoft Azure | USA | 111 | Internationaler Fokus, weltweit länderbezogene Datenschutzstandards |
| Google Cloud | USA | 166 | Internationaler Fokus, weltweit länderbezogene Datenschutzstandards |
| IBM Cloud | USA | 39 | Fokus auf Nordamerika, EU, Australien und Singapur |
| Alibaba Cloud | China | 3 | Es werden 22 Whitepaper angegeben, um bis zu 22 Compliances zu erreichen. Chinesische Standards werden zumindest auf den internationalen Seiten nicht berichtet. |
| OVHcloud | FRA | 20 | Primär europäischer Fokus bei Datenschutzstandards |
| SAP | DEU | 296 | Es werden insgesamt 296 Treffer in der Compliance-Datenbank angegeben. Internationaler Fokus. |
| Telekom/ T-Systems | DEU | 20 | Primär europäischer Fokus bei Datenschutzstandards |
| Orange | FRA | ? | Unklar, Provider bietet keine eigene Compliance-Übersicht an. Tritt als Cloud Broker auf und reicht damit die Compliances anderer Provider weiter. |
| Vodafone | UK | ? | Unklar, Provider bietet keine eigene Compliance-Übersicht an. Tritt als Cloud Broker auf und reicht damit die Compliances anderer Provider weiter. |

Einige Ausnahme ist SAP. SAP ist sogar das Unternehmen, das die meisten Übereinstimmungen mit internationalen Standards und Normen meldet (insgesamt 296). Das dürfte auch die Obergrenze aller (zum Teil sehr kleinteilig von SAP berichteten) Cloud- und Datenschutzregulierungen weltweit darstellen. Die Provider Orange und Vodafone treten eher als Cloud Broker auf und verzichten weitestgehend auf die Compliance-Darstellung eigener Services und beziehen sich primär auf die Compliances der vermittelten Services anderer Provider.

Bei der Betrachtung der berichteten Compliances fällt eine Häufung einiger Normen auf, die somit eine besondere Bedeutung im Markt zu haben scheinen. Auf diese Normen und Anforderungen wird im Folgenden (Abschnitt 17.1.1 bis Abschnitt 17.1.11) zusammenfassend eingegangen, da sie von allgemeinem Interesse sind und deutlich häufiger genannt werden als andere. Europäische und deutsche Standards werden dabei überproportional berücksichtigt, da sie für ein deutschsprachiges Buch eine relativ höhere Bedeutung haben als z. B. südafrikanische Datenschutzstandards. Insgesamt liefern die folgenden Abschnitte ein interessantes (allerdings nicht vollständiges) Schlaglicht auf europäische und internationale Compliance-Erfordernisse im Cloud Computing.

17.1.1 ISO 9001

Die ISO 9001 ist eine internationale Norm für Qualitätsmanagementsysteme (QMS). Sie wurde von der Internationalen Organisation für Normung (ISO) entwickelt und legt Anforderungen an ein effektives Qualitätsmanagementsystem fest. Grundsätzlich ist die ISO 9001 weder auf IT-Systeme noch Cloud Computing fokussiert, sondern behandelt QMS ganz allgemein. Für sich genommen ist die ISO 9001 aus dem Blickwinkel des Cloud Computings weitestgehend aussagelos, wird aber von vielen Providern häufig genannt.

Die ISO 9001 ist eine prozessorientierte Norm, die Organisationen dabei unterstützt, die Kundenzufriedenheit zu verbessern, die Effizienz zu steigern und kontinuierliche Verbesserungen zu erreichen. Sie legt die Grundlagen für die Umsetzung einer systematischen Qualitätsmanagementstruktur fest und kann auf Organisationen unterschiedlicher Größe und Branchen angewendet werden. Die Norm umfasst verschiedene Anforderungen, darunter:

- Kontext der Organisation: Die Organisation muss den Kontext ihrer Tätigkeiten verstehen, um ihre Ziele und den Anwendungsbereich des Qualitätsmanagementsystems festzulegen.
- Führung: Die oberste Leitung der Organisation muss Verpflichtung und Engagement für das Qualitätsmanagementsystem zeigen und eine klare Qualitätsrichtlinie festlegen.
- Planung: Es müssen Ziele und Pläne zur Erreichung der Qualitätsziele festgelegt werden. Risiken und Chancen müssen bewertet und Maßnahmen zur Risikobehandlung festgelegt werden.
- Unterstützung: Ressourcen, Kompetenzen und Kommunikation müssen bereitgestellt werden, um das Qualitätsmanagementsystem zu unterstützen.
- Betrieb: Prozesse müssen identifiziert, geplant, implementiert und kontrolliert werden, um die Qualitätsziele zu erreichen. Dies umfasst auch die Überwachung von Produkten und Dienstleistungen.
- Bewertung der Leistung: Die Organisation muss die Leistung des Qualitätsmanagementsystems bewerten, einschließlich der Überwachung der Kundenzufriedenheit, der internen Audits und der Managementbewertung.
- Verbesserung: Die Organisation muss Maßnahmen zur kontinuierlichen Verbesserung des Qualitätsmanagementsystems ergreifen, einschließlich der Behandlung von Abweichungen, Korrekturmaßnahmen und der Umsetzung von Präventivmaßnahmen.

Die ISO 9001-Zertifizierung ist eine freiwillige Maßnahme, bei der ein unabhängiger Zertifizierungsprüfer die Konformität eines Unternehmens mit den Anforderungen der Norm überprüft. Eine erfolgreiche Zertifizierung nach ISO 9001 zeigt, dass das Unternehmen ein Qualitätsmanagementsystem implementiert hat, das den internationalen Standards entspricht und zur kontinuierlichen Verbesserung der Qualität beiträgt. Mehr aber auch nicht. Die ISO 9001 wird im Kontext des Cloud Computings erst dann aussagekräftig, wenn sie mit nachfolgenden Standards oder Zertifizierung gemeinsam genannt wird. Dann zeigt die ISO 9001, dass ein Unternehmen in allen Prozessen durchgängig um Qualität bemüht ist, also auch in dem Erbringen von Cloud Computing-Leistungen und -Diensten.

17.1.2 BSI-IT-Grundschutz und BSI-Standards

Der BSI-Grundschutz (Bundesamt für Sicherheit in der Informationstechnik-Grundschutz) ist ein Rahmenwerk des Bundesamts für Sicherheit in der Informationstechnik (BSI) in Deutschland. Es handelt sich um eine Sammlung von Sicherheitsstandards, Methoden und Best Practices, die Unternehmen und Organisationen bei der Einführung und Umsetzung von Informationssicherheitsmaßnahmen unterstützen.

Der BSI-Grundschutz basiert auf einer Risikoanalyse und bietet eine strukturierte Herangehensweise zur Auswahl, Implementierung und Überprüfung geeigneter Sicherheitsmaßnahmen. Er deckt verschiedene Aspekte der Informationssicherheit ab, wie z. B. physische Sicherheit, IT-Sicherheit, Notfallmanagement, Personal- und Organisationsmaßnahmen.

Im Zusammenhang mit Cloud Computing bezieht sich das BSI auf verschiedene BSI-Standards, die spezifische Aspekte der Cloud-Sicherheit abdecken:

- BSI-Standard 200-1: Der „Cloud Computing Compliance Controls Catalogue“-(C5)-Standard definiert Anforderungen an die Sicherheit von Cloud-Diensten und bietet einen Kriterienkatalog zur Bewertung der Sicherheit von Cloud-Providern (vgl. auch Abschnitt 17.1.3).
- BSI-Standard 100-4: Das Informationssicherheits-Managementsystem (ISMS) basiert auf der ISO/IEC 27001 (vgl. auch Abschnitt 17.1.4) für Behörden und Organisationen mit Sicherheitsaufgaben (ISMS-Grundschutzhandbuch). Dieser Standard gibt Anleitungen zur Implementierung eines Informationssicherheits-Managementsystems (ISMS) auf Basis von ISO/IEC 27001 für Behörden und Organisationen mit Sicherheitsaufgaben (z. B. öffentliche Institutionen).
- BSI-Standard 100-3 „Risikoanalyse auf Basis von IT-Grundschutz“: Dieser Standard beschreibt Methoden und Vorgehensweisen für die Durchführung einer Risikoanalyse auf Basis des BSI-Grundschutzes, um die spezifischen Risiken im Zusammenhang mit Cloud-Diensten zu identifizieren und zu bewerten.

Diese Standards bieten Unternehmen und Organisationen Orientierung und konkrete Anleitungen, um die Sicherheit von Cloud-Infrastrukturen, -Diensten und -Anwendungen zu bewerten, geeignete Maßnahmen umzusetzen und Risiken angemessen zu behandeln. Sie unterstützen Unternehmen bei der Auswahl und Zusammenarbeit mit Cloud-Providern und tragen dazu bei, die Sicherheit und den Schutz von Daten in der Cloud zu gewährleisten.

17.1.3 BSI-C5-Zertifizierung

Der BSI-C5-Kriterienkatalog ist ein von dem Bundesamt für Sicherheit in der Informationstechnik (BSI) in Deutschland entwickelter Katalog von Sicherheitsanforderungen für Cloud-Dienste. BSI-C5 steht für „Cloud Computing Compliance Controls Catalogue“ und dient als Leitfaden für Unternehmen, um die Sicherheit von Cloud-Diensten zu bewerten und geeignete Sicherheitsmaßnahmen zu implementieren.

Der Kriterienkatalog basiert auf nationalen und internationalen Sicherheitsstandards sowie bewährten Praktiken und berücksichtigt die spezifischen Anforderungen an die Sicherheit von Cloud-Diensten. Er bietet einen strukturierten Rahmen und eine systematische Herangehensweise, um die Sicherheit von Cloud-Infrastrukturen, Plattformen und Anwendungen

zu gewährleisten. Der BSI-C5-Kriterienkatalog umfasst hierzu verschiedene Sicherheitskategorien und -kontrollen, darunter:

- Organisatorische Sicherheit: Hier werden Aspekte wie Sicherheitsrichtlinien, Risikomanagement, Incident Management und Sicherheitsbewusstsein behandelt.
- Personal- und physische Sicherheit: Dies umfasst Maßnahmen zur Überprüfung der Mitarbeiter, physische Zugangskontrollen zu Cloud-Einrichtungen und Schutz vor Umweltgefahren.
- System- und Netzwerksicherheit: Hier werden Anforderungen an die Absicherung von Systemen, Netzwerken, Datenbanken, Firewalls und anderen technischen Komponenten definiert.
- Schutz von Informationen: Dies beinhaltet den Schutz von vertraulichen Informationen, Verschlüsselung, Zugriffskontrollen und Maßnahmen gegen Datenverlust.
- Rechenzentrums- und Infrastruktursicherheit: Hier werden Anforderungen an die physische Sicherheit von Rechenzentren, Energieversorgung, Klimatisierung, Backups und Wiederherstellung definiert.
- Compliance und Audit: Dies umfasst die Einhaltung von gesetzlichen und regulatorischen Anforderungen sowie die Durchführung von Sicherheitsaudits und -prüfungen.

Der BSI-C5-Kriterienkatalog richtet sich sowohl an Cloud-Dienstleister als auch an Cloud-Nutzer und bietet eine gemeinsame Basis für die Bewertung und Gewährleistung der Sicherheit in der Cloud. Unternehmen können den Katalog verwenden, um die Sicherheitsstandards von Cloud-Dienstleistern zu bewerten und geeignete Sicherheitsmaßnahmen bei der Nutzung von Cloud-Diensten umzusetzen.

17.1.4 ISO/IEC 27001 und 27017/27018

Die ISO/IEC 27001 ist eine internationale Norm für Informationssicherheits-Managementsysteme (ISMS). Sie legt Anforderungen und Best Practices für die Planung, Implementierung, Überwachung und kontinuierliche Verbesserung eines umfassenden Informationssicherheitsmanagementsystems fest. Die Norm zielt darauf ab, die Vertraulichkeit, Integrität und Verfügbarkeit von Informationen in einer Organisation zu schützen und Risiken im Zusammenhang mit Informationssicherheit zu identifizieren und zu behandeln.

Die ISO/IEC 27001 umfasst eine Reihe von Kontrollen und Maßnahmen, die in verschiedenen Bereichen der Informationssicherheit angewendet werden können, wie zum Beispiel:

- Organisation des Informationssicherheits-Managementsystems: Festlegung von Verantwortlichkeiten, Ernennung eines Informationssicherheitsmanagers, Einführung von Richtlinien und Verfahren.
- Risikomanagement: Identifizierung von Risiken, Bewertung ihrer Auswirkungen, Implementierung geeigneter Kontrollen und Überwachung des Risikostatus.
- Zugriffskontrolle: Verwaltung von Benutzerzugriffen, Authentifizierung, Berechtigungsvergabe und Überwachung von Zugriffsaktivitäten.
- Kryptografie: Schutz von vertraulichen Informationen durch den Einsatz von Verschlüsselung und anderen kryptografischen Verfahren.
- Sicherheit der Kommunikation: Sichere Übertragung und Verarbeitung von Informationen, Schutz vor unbefugtem Zugriff oder Manipulation.

ISO/IEC 27017 und ISO/IEC 27018 sind spezifische Ergänzungen zu ISO/IEC 27001. ISO/IEC 27017 legt zusätzliche Sicherheitskontrollen für die Bereitstellung und Nutzung von Cloud-Diensten fest, während sich ISO/IEC 27018 auf den Schutz personenbezogener Daten in der Cloud konzentriert.

Die Zertifizierung nach ISO/IEC 27001 wird von Unternehmen angestrebt, um ihre Bemühungen im Bereich der Informationssicherheit nachzuweisen. Eine erfolgreiche Zertifizierung nach ISO/IEC 27001 zeigt, dass das Unternehmen ein robustes Informationssicherheits-Managementsystem etabliert hat und sich kontinuierlich um die Sicherheit von Informationen und Daten kümmert. Es ist ein wichtiger Vertrauensindikator für Kunden, Geschäftspartner und andere Stakeholder und kann bei der Erfüllung gesetzlicher Anforderungen helfen.

17.1.5 CSA STAR

Das Cloud Security Alliance Security, Trust & Assurance Registry (CSA STAR) ist ein Rahmenwerk zur Bewertung der Sicherheit von Cloud-Anbietern und zur Offenlegung ihrer Sicherheitspraktiken. CSA STAR ist ein Programm der Cloud Security Alliance (CSA), einer internationalen Organisation, die sich der Förderung bewährter Sicherheitspraktiken im Cloud Computing widmet.

CSA STAR bietet einen Rahmen für Anbieter von Cloud-Diensten, um ihre Sicherheitsmaßnahmen transparent darzustellen und Kunden Informationen über den Sicherheitsstatus ihrer Cloud-Dienste zur Verfügung zu stellen. Ziel des Programms ist es, das Vertrauen und das Verständnis der Kunden hinsichtlich der Sicherheit von Cloud-Diensten zu fördern. Das CSA STAR-Programm besteht aus zwei Hauptkomponenten:

- **CSA STAR Certification:** Dieser Teil des Programms bietet eine unabhängige Bewertung und Zertifizierung von Cloud Service-Providern. Die Zertifizierung basiert auf den Anforderungen der CSA Cloud Control Matrix (CCM), einer Kontrollmatrix für Sicherheitskontrollen und -praktiken in Cloud-Umgebungen. Durch die Zertifizierung erhalten Cloud Service-Provider eine Bestätigung ihrer Sicherheitsmaßnahmen und können diese Informationen ihren Kunden zur Verfügung stellen.
- **CSA STAR Self-Assessment:** Dieser Teil ermöglicht es Cloud Service-Providern, eine Selbst-einschätzung ihrer Sicherheitspraktiken vorzunehmen und diese Informationen in einer öffentlich zugänglichen Datenbank zu veröffentlichen. Die Selbsteinschätzung basiert ebenfalls auf der CSA Cloud Control Matrix (CCM) und bietet Kunden einen Einblick in die Sicherheitsmaßnahmen des Anbieters.

Die CSA Cloud Control Matrix (CCM) umfasst viele spezifische Kontrollen und Praktiken für die Sicherheit von Cloud-Umgebungen und ist in die folgenden Hauptbereiche und Kontroll-kategorien unterteilt:

- **Compliance und Audit Assurance:** Kontrollen und Verfahren zur Einhaltung rechtlicher, regulatorischer und vertraglicher Anforderungen sowie zur Durchführung von Sicherheitsaudits.
- **Datenklassifizierung und Datenschutz:** Kontrollen zur Identifizierung, Klassifizierung und Schutz personenbezogener Daten sowie zum Umgang mit Datenschutzanforderungen.

- Sicherheitspersonal und -richtlinien: Kontrollen für die Organisation, Schulung und Zuständigkeiten des Sicherheitspersonals sowie die Entwicklung und Umsetzung von Sicherheitsrichtlinien.
- Asset Management: Kontrollen zur Identifizierung, Klassifizierung und Verwaltung von IT-Ressourcen und Informationen in der Cloud.
- Zugangskontrolle: Kontrollen zur sicheren Verwaltung von Benutzerzugriffen, Authentifizierung, Berechtigungsvergabe und Zugriffsüberwachung.
- Sicherheitseignismanagement, Notfallvorsorge und -reaktion: Kontrollen zur Erkennung, Überwachung, Meldung und Reaktion auf Sicherheitsvorfälle und Notfälle.
- Infrastruktur- und Virtualisierungssicherheit: Kontrollen zur Sicherung der zugrunde liegenden Infrastruktur und der Virtualisierungsumgebung in der Cloud.
- Daten- und Anwendungssicherheit: Kontrollen für den sicheren Umgang mit Daten und Anwendungen in der Cloud, einschließlich Verschlüsselung, Sicherheitskonfigurationen und Sicherheitslückenmanagement.
- Identitäts- und Zugriffsmanagement: Kontrollen für die Verwaltung von Identitäten, Benutzerzugriff und Rechteverwaltung in Cloud-Umgebungen.
- Compliance mit internationalen Standards: Kontrollen für die Einhaltung internationaler Sicherheitsstandards und -bestimmungen, wie beispielsweise ISO 27001 und GDPR.

Die CSA Cloud Control Matrix (CCM) enthält darüber hinaus detaillierte Kontrollen, Best Practices und Empfehlungen für die Sicherheit von Cloud-Umgebungen. Durch die Teilnahme am CSA STAR-Programm können Cloud-Anbieter die Transparenz ihrer Sicherheitsmaßnahmen erhöhen. Kunden können die Informationen des CSA STAR-Programms nutzen, um fundierte Entscheidungen bei der Auswahl von Cloud-Diensten zu treffen und deren Sicherheitslage besser zu verstehen. Eine CSA STAR-Zertifizierung bietet somit eine standardisierte Informationsquelle für Kunden, um die Sicherheit von Cloud-Diensten besser beurteilen zu können.

17.1.6 CISPE Code of Conduct

Diese Zertifizierung wird von CISPE (Cloud Infrastructure Services Providers in Europe) vergeben und stellt sicher, dass Cloud-Dienstleister bestimmte Datenschutzstandards einhalten. CISPE ist eine Organisation, die sich aus führenden europäischen Anbietern von Cloud-Infrastrukturen zusammensetzt.

Der CISPE Code of Conduct (Verhaltenskodex) wurde entwickelt, um den Schutz personenbezogener Daten in der Cloud zu gewährleisten und die Einhaltung der europäischen Datenschutzgesetze, insbesondere der Datenschutz-Grundverordnung (DSGVO), sicherzustellen:

- Datenlokalisierung: Der Verhaltenskodex verlangt, dass personenbezogene Daten innerhalb der Europäischen Union (EU) oder des Europäischen Wirtschaftsraums (EWR) gespeichert und verarbeitet werden.
- Schutz personenbezogener Daten: Der Verhaltenskodex fordert strenge Sicherheitsmaßnahmen zum Schutz personenbezogener Daten vor unbefugtem Zugriff, Verlust, Diebstahl oder Missbrauch.

- Datenzugriff und Übertragbarkeit: Der Verhaltenskodex gewährleistet, dass Kunden die volle Kontrolle über ihre Daten haben, einschließlich des Rechts auf Zugriff, Berichtigung, Löschung und Übertragbarkeit.
- Transparenz: Der Verhaltenskodex erfordert, dass Cloud-Dienstleister klare und verständliche Informationen über ihre Datenverarbeitungspraktiken, Sicherheitsmaßnahmen und Dienstleistungsniveaus bereitstellen.
- Unabhängige Auditierung: Cloud-Dienstleister, die den Verhaltenskodex einhalten, müssen sich regelmäßigen unabhängigen Audits unterziehen, um die Einhaltung der Datenschutzstandards nachzuweisen.

Dies bietet Unternehmen eine Orientierungshilfe bei der Auswahl von Cloud-Infrastrukturdielen, die den europäischen Datenschutzstandards entsprechen. Durch die Einhaltung des Codes können Cloud-Dienstleister das Vertrauen ihrer Kunden stärken und sicherstellen, dass personenbezogene Daten sicher und rechtskonform in der Cloud verarbeitet werden. Der CISPE Code of Conduct ist allerdings ein **IaaS-fokussiertes Regelwerk** und konzentriert sich vor allem auf Cloud-Infrastrukturdielen. Andere Arten von Cloud-Services, wie beispielsweise Software-as-a-Service (SaaS) oder Platform-as-a-Service (PaaS), können separate Compliance-Anforderungen haben.

17.1.7 EU Cloud Code of Conduct

Der EU Cloud Code of Conduct ist ein Verhaltenskodex speziell für Cloud-Anbieter. Er wurde von einer Gruppe von Cloud-Anbietern und -Nutzern in Zusammenarbeit mit der Europäischen Kommission entwickelt, um Vertrauen, Transparenz und den Schutz personenbezogener Daten in der Cloud zu fördern. Der Verhaltenskodex basiert auf den Grundsätzen der Datenschutz-Grundverordnung (DSGVO) und legt bestimmte Standards und Verhaltensregeln fest, die von Cloud-Anbietern eingehalten werden müssen. Der Kodex bietet Nutzern von Cloud-Diensten die Möglichkeit, Cloud-Anbieter auszuwählen, die sich zur Einhaltung bestimmter Datenschutz- und Sicherheitsstandards verpflichten. Die Hauptziele des EU Cloud Code of Conduct sind:

- Schutz personenbezogener Daten: Der Kodex soll den Schutz personenbezogener Daten in der Cloud gewährleisten und sicherstellen, dass Cloud-Anbieter angemessene Maßnahmen ergreifen, um die Vertraulichkeit, Integrität und Verfügbarkeit der Daten sicherzustellen.
- Transparenz und Information: Der Kodex fordert Transparenz und informiert die Nutzer über die Datenverarbeitung und -sicherheit in der Cloud. Cloud-Anbieter müssen klare und verständliche Informationen über ihre Datenschutzpraktiken und -verfahren bereitstellen.
- Vertrauen und Zuverlässigkeit: Der Kodex soll das Vertrauen der Nutzer in Cloud-Dienste stärken, indem er einen Rahmen für zuverlässige und vertrauenswürdige Cloud-Anbieter schafft. Durch die Einhaltung des Kodex können Cloud-Anbieter zeigen, dass sie sich hohen Datenschutz- und Sicherheitsstandards verpflichtet fühlen.

Der EU Cloud Code of Conduct ist ein freiwilliges Instrument für Cloud-Anbieter. Cloud-Anbieter können sich freiwillig zur Einhaltung des Kodex verpflichten und ihre Einhaltung durch unabhängige Zertifizierungsstellen überprüfen lassen. Durch die Einhaltung des Kodex können Cloud-Anbieter das EU Cloud Code of Conduct-Logo verwenden, um ihr Engagement zu demonstrieren.

Der EU Cloud Code of Conduct ist Teil der Bemühungen der Europäischen Kommission, den Datenschutz und die Datensicherheit in der Cloud zu fördern und einen vertrauenswürdigen und sicheren Rahmen für Cloud-Dienste in der EU zu schaffen.

17.1.8 SOC 1-3 (Service Organization Control)

SOC (Service Organization Control) ist ein Prüfungsstandard, der vom American Institute of Certified Public Accountants (AICPA) entwickelt wurde. Es handelt sich um eine Prüfungsnorm für Serviceorganisationen, die Dienstleistungen anbieten, bei denen Informationen eine wesentliche Rolle spielen, wie zum Beispiel Cloud Service-Provider, Rechenzentren, Software-as-a-Service-(SaaS)-Anbieter und andere Unternehmen, die Datenverarbeitungsdienste erbringen.

Mittels SOC 2 werden insbesondere die Sicherheit, Verfügbarkeit, Integrität, Vertraulichkeit und Datenschutzpraktiken einer Serviceorganisation bewertet. Die Bewertung basiert auf den Trust Services Criteria (TSC), einem Satz von Prinzipien und Kriterien, die von der AICPA entwickelt wurden, um die Sicherheit und den Schutz von Informationen zu gewährleisten. Es gibt fünf Hauptkriterien, die gemäß Trust Services Criteria bewertet werden:

- Sicherheit (Security): Sicherheit umfasst den Schutz von Systemen, Anwendungen und Informationen vor unberechtigtem Zugriff, Missbrauch, Offenlegung, Zerstörung oder Unterbrechung. Zu den Sicherheitskriterien gehören beispielsweise die Implementierung von Zugangskontrollen, die Überwachung von Systemen und Netzwerken, die Erkennung und Reaktion auf Sicherheitsvorfälle sowie die physische Sicherheit.
- Verfügbarkeit (Availability): Verfügbarkeit bezieht sich auf die Bereitstellung und Aufrechterhaltung von Systemen und Diensten gemäß den vereinbarten Service Level Agreements. Verfügbarkeitskriterien umfassen die Implementierung von Maßnahmen zur Ausfallsicherheit, Redundanz, Kapazitätsplanung und Notfallwiederherstellung.
- Vertraulichkeit (Confidentiality): Vertraulichkeit bezieht sich auf den Schutz von Informationen vor unberechtigtem Zugriff oder Offenlegung. Sie umfasst die Implementierung von Verschlüsselungsmechanismen, Zugriffskontrollen, sicheren Kommunikationskanälen und Maßnahmen zur Datensicherheit.
- Integrität (Verarbeitungsintegrität): Integrität bezieht sich auf die Genauigkeit, Vollständigkeit und Korrektheit von Informationen sowie die korrekte Verarbeitung von Transaktionen. Integritätskriterien umfassen die Implementierung von Validierungs- und Verifizierungsmechanismen, Kontrollen zur Fehlererkennung und -korrektur sowie zur Aufrechterhaltung der Datenintegrität.
- Datenschutz (Privacy): Datenschutz bezieht sich auf den Schutz personenbezogener Daten gemäß den geltenden Datenschutzgesetzen und -vorschriften. Dies beinhaltet die Einhaltung von Datenschutzrichtlinien, den korrekten Umgang mit personenbezogenen Daten, die Einwilligung der Betroffenen und die Bereitstellung von Datenschutzrechten.

Ein SOC 2-Audit wird von einem unabhängigen Dritten, einem Wirtschaftsprüfer oder einer Wirtschaftsprüfungsgesellschaft, durchgeführt und in einem Bericht dokumentiert. Der Bericht gibt Kunden und Interessenten einen Einblick in die Sicherheits- und Compliance-Praktiken einer Serviceorganisation. Es ist wichtig zu beachten, dass ein SOC 2-Bericht kein

Zertifikat ist, sondern eine Prüfungsdokumentation, die die Konformität der Dienstleistungsorganisation mit den Trust Services-Kriterien nachweist. Dennoch sind SOC 2-Berichte ein wichtiges Instrument zur Bewertung der Sicherheits- und Datenschutzpraktiken von Serviceorganisationen und helfen Kunden bei der Auswahl vertrauenswürdiger Dienstleister, insbesondere im Bereich Datenschutz und Datensicherheit.

17.1.9 FedRAMP

Das Federal Risk and Authorization Management Program (FedRAMP) ist ein Programm der US-Regierung zur Standardisierung des Sicherheits- und Compliance-Prozesses für Cloud-Anbieter. Das Programm wurde entwickelt, um sicherzustellen, dass US-Bundesbehörden sichere Cloud-Dienste nutzen können, die den Sicherheitsstandards der Regierung entsprechen. FedRAMP basiert auf einem standardisierten Ansatz für die Risikobewertung und das Risikomanagement von Cloud-Diensten. Es definiert Anforderungen an die Sicherheit von Cloud-Systemen und ermöglicht es Anbietern, ihre Dienste von einer unabhängigen dritten Partei überprüfen zu lassen:

- Sicherheitsstandards und Kontrollen: FedRAMP basiert auf dem NIST (National Institute of Standards and Technology) Framework und definiert spezifische Sicherheitsstandards und Kontrollen, die von Cloud-Dienstleistern erfüllt werden müssen. Dazu gehören Kontrollen zur Zugriffskontrolle, Verschlüsselung, Datensicherheit, Kontinuitätsplanung und mehr.
- Autorisierungsverfahren: Cloud-Dienstleister, die ihre Dienste für US-Bundesbehörden anbieten möchten, müssen einen umfangreichen Sicherheitsbewertungsprozess durchlaufen. Dieser Prozess umfasst die Erstellung und Einreichung von Sicherheitsdokumentationen, die Durchführung von Sicherheitsbewertungen und Audits durch autorisierte Experten.
- Wiederverwendbare Sicherheitspakete: FedRAMP ermöglicht die Wiederverwendung von bereits genehmigten Sicherheitsdokumentationen und -prüfungen. Dies erleichtert anderen Cloud-Dienstleistern den Zugang zum Markt und beschleunigt den Zulassungsprozess.
- Kontinuierliche Überwachung: FedRAMP legt Wert auf die kontinuierliche Überwachung und Bewertung der Sicherheitslage von Cloud-Dienstleistern. Dies umfasst regelmäßige Audits, Schwachstellenmanagement und die Einhaltung der vereinbarten Sicherheitsstandards während der gesamten Nutzungsdauer der Cloud-Dienste.

FedRAMP bietet somit Vorteile sowohl für Cloud-Dienstleister als auch für US-Bundesbehörden. Cloud-Dienstleister können ihren Kunden, insbesondere Bundesbehörden, ihre Sicherheits- und Compliance-Fähigkeiten nachweisen. Bundesbehörden profitieren von einer standardisierten Bewertung und einer größeren Auswahl an sicheren und genehmigten Cloud-Diensten, die sie nutzen können.

17.1.10 HIPAA

Der Health Insurance Portability and Accountability Act (HIPAA) ist ein Bundesgesetz der USA. Ziel des HIPAA ist es, die Privatsphäre und Sicherheit von Gesundheitsdaten zu schützen und die Übertragbarkeit von Krankenversicherungen zu gewährleisten. In seiner

Ausrichtung und Zielsetzung entspricht es somit der DSGVO (bzw. GDPR) – allerdings mit dem Fokus auf Gesundheitsdaten. HIPAA besteht aus verschiedenen Regelungen, von denen zwei besonders relevant sind:

- Privacy Rule (Datenschutzregelung): Die Privacy Rule legt Standards fest, um die Vertraulichkeit und den Schutz von personenbezogenen Gesundheitsdaten (Protected Health Information, PHI) zu gewährleisten. Sie regelt den Zugang, die Nutzung und die Offenlegung von PHI durch Krankenversicherer, Gesundheitsdienstleister, Auftragsverarbeiter und andere Organisationen im Gesundheitswesen. Die Privacy Rule gibt Patienten bestimmte Rechte in Bezug auf ihre Gesundheitsdaten und verlangt von Organisationen die Implementierung von Datenschutzpraktiken und -verfahren.
- Security Rule (Sicherheitsregelung): Die Security Rule setzt Standards für die Sicherheit von elektronischen Gesundheitsdaten (Electronic Protected Health Information, ePHI). Sie verlangt von Organisationen im Gesundheitswesen, angemessene technische, administrative und physische Sicherheitsmaßnahmen zu implementieren, um die Vertraulichkeit, Integrität und Verfügbarkeit von ePHI zu schützen. Die Verordnung umfasst Aspekte wie Zugangskontrolle, Verschlüsselung, Risikobewertung, Notfallmaßnahmen und Schulung des Personals.

HIPAA gilt für alle „Covered Entities“ wie Krankenhäuser, Ärzte, Krankenversicherer und andere Anbieter von Gesundheitsleistungen sowie für „Business Associates“, die im Auftrag von Covered Entities Dienstleistungen erbringen und dabei Zugriff auf PHI oder ePHI haben. Die Einhaltung des HIPAA ist für Organisationen im Gesundheitswesen in den USA verpflichtend. Verstöße gegen HIPAA können erhebliche rechtliche und finanzielle Folgen haben, einschließlich Geldstrafen. HIPAA gilt jedoch nur für Organisationen und Personen, die in den USA tätig sind und der Rechtsprechung der USA unterliegen.

17.1.11 PCI DSS

PCI DSS steht für Payment Card Industry Data Security Standard. Es handelt sich um einen Sicherheitsstandard, der vom Payment Card Industry Security Standards Council (PCI SSC) entwickelt wurde. Der PCI DSS wurde eingeführt, um die Sicherheit von Zahlungskarteninformationen zu gewährleisten und den Schutz von Kreditkarteninhabern und sensiblen Zahlungsdaten zu verbessern.

Der PCI DSS legt Anforderungen für Organisationen fest, die Kreditkartenzahlungen akzeptieren, verarbeiten, speichern oder übertragen. Er gilt für verschiedene Akteure in der Zahlungsbranche, darunter Händler, Zahlungsdienstleister, Zahlungsgateway-Anbieter, Acquirer und andere, die mit Kreditkartendaten in Berührung kommen. Die Anforderungen des PCI DSS umfassen unter anderem:

- Aufrechterhaltung einer sicheren Netzwerkumgebung: Dies beinhaltet die Installation und regelmäßige Aktualisierung von Firewalls, die Verwendung sicherer Netzwerkprotokolle, die Einschränkung des Zugriffs auf Systeme und Netzwerke sowie die Vermeidung von Standardpasswörtern.
- Schutz der gespeicherten Karteninhaberdaten: Dies beinhaltet die Verschlüsselung von Kreditkartendaten, die Beschränkung des Zugriffs auf die Daten, die Implementierung von Zugriffskontrollen und die regelmäßige Überwachung der Systeme.

- Durchführung regelmäßiger Schwachstellenprüfungen: Organisationen müssen regelmäßig Schwachstellenprüfungen durchführen, um potenzielle Sicherheitslücken zu identifizieren und zu beheben.
- Implementierung strenger Zugriffskontrollen: Dies umfasst die Zuweisung eindeutiger Identifikatoren für Benutzer, die Verwendung von Zwei-Faktor-Authentifizierung, die Begrenzung des Zugriffs auf kritische Systeme und Daten sowie die Überwachung der Zugriffsaktivitäten.
- Überwachung und regelmäßige Überprüfung der Sicherheitsmaßnahmen: Organisationen müssen Sicherheitsergebnisse und -aktivitäten protokollieren, regelmäßig Überwachungsmaßnahmen durchführen, verdächtige Aktivitäten identifizieren und Untersuchungen durchführen.
- Durchführung eines Informationssicherheitsprogramms: Organisationen müssen ein umfassendes Informationssicherheitsprogramm etablieren, das Richtlinien, Verfahren, Schulungen und Sensibilisierungsmaßnahmen für Mitarbeiter umfasst.

Die Einhaltung des PCI DSS ist für Organisationen, die Kreditkartendaten verarbeiten, obligatorisch. Die Nichteinhaltung kann zu rechtlichen Konsequenzen, Verlust des Kreditkartenverarbeitungsprivilegs und finanziellen Sanktionen führen. Regelmäßige Validierungen und Compliance-Audits sind erforderlich, um sicherzustellen, dass die Sicherheitsanforderungen des PCI DSS erfüllt werden.

17.1.12 Zusammenfassung

Im Bereich der **nationalen Empfehlungen** sind sicher die Vorgaben des BSI die prominentesten. Das BSI-Grundschutz-Rahmenwerk des Bundesamts für Sicherheit in der Informationstechnik (BSI) in Deutschland bietet eine Sammlung von Sicherheitsstandards und Best Practices zur Informationssicherheit. Es behandelt verschiedene Bereiche wie physische Sicherheit, IT-Sicherheit und Notfallmanagement. Der BSI-C5-Kriterienkatalog ist ein Leitfaden des BSI zur Bewertung der Sicherheit von Cloud-Diensten. Er basiert auf nationalen und internationalen Standards und behandelt verschiedene Sicherheitsaspekte, darunter organisatorische Sicherheit, Personal- und physische Sicherheit, System- und Netzwerksicherheit, Schutz von Informationen, Rechenzentrums- und Infrastruktursicherheit sowie Compliance und Audit. Der Katalog dient sowohl Cloud-Anbietern als auch Nutzern als Basis zur Sicherheitsbewertung in der Cloud.

Daneben gibt es **internationale Normen**. Die ISO/IEC 27001 ist eine internationale Norm für Informationssicherheits-Managementsysteme, die Best Practices für den Schutz von Informationen in Organisationen bietet. Sie umfasst Kontrollen und Maßnahmen in Bereichen wie Organisation, Risikomanagement, Zugriffskontrolle, Kryptografie und Kommunikationssicherheit. Die Normen ISO/IEC 27017 und ISO/IEC 27018 ergänzen sie mit spezifischen Sicherheitskontrollen für Cloud-Dienste und den Schutz von personenbezogenen Daten in der Cloud. Unternehmen streben eine Zertifizierung nach ISO/IEC 27001 an, um ihre Bemühungen im Bereich der Informationssicherheit zu belegen, was für Kunden und Partner ein Vertrauensindikator ist und bei der Einhaltung von gesetzlichen Anforderungen unterstützt.

Ferner sind oft **Rahmenwerke und Verhaltenskodexe** zu berücksichtigen. Hier sind insbesondere zu nennen:

- Das Cloud Security Alliance Security, Trust & Assurance Registry (CSA STAR) ist ein internationales Rahmenwerk zur Bewertung und Offenlegung der Sicherheitspraktiken von Cloud-Anbietern. Das Ziel ist es, das Vertrauen in Cloud-Dienste zu erhöhen. Es besteht aus zwei Hauptteilen: CSA STAR Certification und CSA STAR Self-Assessment, die auf der CSA Cloud Control Matrix (CCM) basieren. Diese Matrix beinhaltet verschiedene Kontrollen und Praktiken für Cloud-Sicherheit.
- Der europäische CISPE Code of Conduct wird von „Cloud Infrastructure Services Providers in Europe (CISPE)“ vergeben und stellt sicher, dass Cloud-Dienstleister europäische Datenschutzstandards einhalten. Der Code konzentriert sich insbesondere auf den Schutz personenbezogener Daten und die Einhaltung der DSGVO. Der CISPE Code of Conduct fokussiert sich hauptsächlich auf Cloud-Infrastrukturdienste.
- Der EU Cloud Code of Conduct ist ein Verhaltenskodex für Cloud-Anbieter, der in Zusammenarbeit mit der Europäischen Kommission entwickelt wurde. Er basiert auf den Grundsätzen der DSGVO und zielt darauf ab, Transparenz, Vertrauen und den Schutz personenbezogener Daten in der Cloud zu fördern. Cloud-Anbieter können sich freiwillig zu diesem Kodex verpflichten und das EU Cloud Code of Conduct-Logo verwenden, um ihre Verpflichtung zu zeigen. Es ist Teil der Bemühungen der Europäischen Kommission, den Datenschutz in der Cloud zu erhöhen.

Und letztlich sind **branchenspezifische Vorgaben** zu nennen. Hier werden insbesondere die folgenden Richtlinien oft genannt, wenn es um die Einhaltung US-spezifischer Vorgaben, Gesundheitsdaten oder bspw. kreditkartenbasierte Transaktionen geht.

- FedRAMP ist bspw. ein US-Regierungsprogramm zur Standardisierung von Sicherheit und Compliance für Cloud-Anbieter. Es definiert Sicherheitsanforderungen und ermöglicht Überprüfungen durch unabhängige Dritte. Es ist erforderlich, um Leistungen für US-Behörden erbringen zu dürfen.
- HIPAA ist ein US-Bundesgesetz, das die Privatsphäre und Sicherheit von Gesundheitsdaten schützt. Es hat Regeln für Datenschutz und Sicherheit und gilt für Organisationen im Gesundheitswesen in den USA. Verstöße können zu Strafen führen.
- Und letztlich ist PCI DSS ein oft genutzter Sicherheitsstandard für Kreditkartenzahlungen. Es legt Anforderungen für Organisationen fest, die Kreditkartendaten verarbeiten. Die Einhaltung ist verpflichtend, und Nichteinhaltung kann zu Strafen führen.

Die Komplexität und Vielfalt der Sicherheitsstandards und Datenschutzzvorgaben unterstreichen die Notwendigkeit für Organisationen, sowohl nationale als auch internationale Richtlinien sorgfältig zu prüfen und sicherzustellen, dass sie in ihrer Cloud-Infrastruktur und Cloud-Strategie adäquat berücksichtigt werden.

■ 17.2 Aus der DSGVO sich ergebende Anforderungen

Die Datenschutz-Grundverordnung (DSGVO) und die General Data Protection Regulation (GDPR) sind dasselbe Gesetz. Der Begriff GDPR wird häufig für die Datenschutz-Grundverordnung in englischsprachigen Ländern verwendet, während der Begriff DSGVO in Deutschland und anderen deutschsprachigen Ländern gebräuchlicher ist. Die Datenschutz-Grundverordnung ist ein EU-weites Gesetz, das 2018 in Kraft getreten ist. Sie harmonisiert das Datenschutzrecht in der gesamten Europäischen Union und soll den Schutz personenbezogener Daten verbessern und die Rechte der Betroffenen stärken.

Die DSGVO/GDPR enthält eine Reihe von Bestimmungen und Vorschriften zur Verarbeitung personenbezogener Daten, darunter die Definition personenbezogener Daten, die Rechte der betroffenen Personen, die Verantwortlichkeiten von Datenverantwortlichen und Auftragsverarbeitern, die Grundsätze der Datenverarbeitung, die Einwilligung der betroffenen Personen, die Datensicherheit, die Meldung von Datenschutzverletzungen, die Durchführung von Datenschutz-Folgenabschätzungen und vieles mehr.

Die DSGVO gilt für alle Unternehmen und Organisationen, die personenbezogene Daten von EU-Bürgern verarbeiten, unabhängig davon, ob sie ihren Sitz innerhalb oder außerhalb der EU haben. Sie soll den Schutz personenbezogener Daten stärken, den freien Datenverkehr innerhalb der EU erleichtern und ein einheitliches Datenschutzniveau in der gesamten EU gewährleisten.

Die Datenschutz-Grundverordnung (DSGVO) enthält daher allgemeine Grundsätze und Anforderungen zum Schutz personenbezogener Daten, die somit auch für Cloud-native Anwendungen gelten (siehe Abschnitt 17.2.2).

Da die spezifischen Anforderungen der DSGVO je nach den konkreten Umständen und der Art der Cloud-native Anwendung variieren können, ist es ratsam, eine detaillierte Prüfung und Bewertung durchzuführen, um sicherzustellen, dass die Anwendung den Anforderungen der DSGVO entspricht und die Datenschutzprinzipien insbesondere personenbezogener Daten eingehalten werden.

17.2.1 Personenbezogene Daten

Doch was sind überhaupt personenbezogene Daten? Personenbezogene Daten im Sinne der DSGVO sind alle Informationen, die sich auf eine bestimmte oder bestimmbare natürliche Person beziehen. Eine identifizierbare Person ist eine Person, die mithilfe dieser Informationen direkt oder indirekt identifiziert werden kann. Personenbezogene Daten können verschiedene Formen annehmen und umfassen, sind aber nicht beschränkt auf die folgende Liste (Hennrich 2023):

- Identifikationsdaten: Name, Geburtsdatum, Geschlecht, Nationalität, Personalausweisnummer, Sozialversicherungsnummer, Passnummer usw.
- Kontaktdata: Adresse, Telefonnummer, E-Mail-Adresse usw.
- Standortdata: GPS-Koordinaten, IP-Adresse, RFID-Tags usw.

- Online-Identifikatoren: Benutzernamen, Profilbilder, Social-Media-Handles usw.
- Biometrische Daten: Fingerabdrücke, Gesichtserkennungsdaten, Iris-Scans usw.
- Gesundheitsdaten: Informationen zum Gesundheitszustand, medizinische Diagnosen, Behandlungsdaten usw.
- Finanzdaten: Bankkontonummern, Kreditkartendaten, Transaktionsverlauf usw.
- Rassische oder ethnische Daten: Ethnische Herkunft, Religion, Sprache, politische Meinungen usw.
- Genetische und biologische Daten: Genetische Informationen, DNA-Proben, medizinische Tests usw.

Personenbezogene Daten umfassen daher nicht nur direkt angegebene Informationen, sondern auch Informationen, die durch Kombination oder Verknüpfung verschiedener Datenquellen indirekt eine Identifizierung ermöglichen können. Die DSGVO legt strenge Schutz- und Kontrollmaßnahmen für diese Arten personenbezogener Daten fest und gewährt den betroffenen Personen bestimmte Rechte, wie das Recht auf Auskunft, Berichtigung, Löschung, Einschränkung der Verarbeitung und Übertragbarkeit ihrer Daten.

17.2.2 Grundsätze der Verarbeitung personenbezogener Daten

Dabei ist es unerheblich, wo diese Daten verarbeitet werden, ob in der „Cloud“ oder anderweitig. Die Verarbeitung personenbezogener Daten hat noch folgenden Grundsätzen zu erfolgen (Hennrich 2023):

- **Rechtmäßigkeit**, Verarbeitung nach Treu und Glauben: Die Verarbeitung personenbezogener Daten muss auf einer rechtlichen Grundlage beruhen, und sie muss fair und transparent erfolgen. Die betroffene Person muss über den Zweck und die Modalitäten der Datenverarbeitung informiert werden.
- **Transparenz**: Verantwortliche für die Datenverarbeitung müssen sicherstellen, dass die betroffenen Personen über die Verarbeitung ihrer personenbezogenen Daten vollständig und klar informiert sind. Dies umfasst die Offenlegung von Informationen über Identität und Kontaktdata des Verantwortlichen, den Zweck der Verarbeitung, die Rechtsgrundlage, die Empfänger der Daten, die Dauer der Datenspeicherung und die Rechte der betroffenen Personen (wie das Recht auf Auskunft, Berichtigung, Löschung usw.).
- **Zweckbindung**: Personenbezogene Daten dürfen nur für festgelegte, eindeutige und legitime Zwecke erhoben werden. Sie dürfen nicht in einer Weise weiterverarbeitet werden, die mit diesen Zwecken unvereinbar ist.
- **Datenminimierung**: Es dürfen nur die für die Verarbeitungszwecke notwendigen personenbezogenen Daten erhoben werden. Die Daten müssen auf das erforderliche Maß beschränkt sein und aktuell gehalten werden.
- **Richtigkeit**: Personenbezogene Daten müssen korrekt und gegebenenfalls auf dem neuesten Stand sein. Angemessene Maßnahmen sollten ergriffen werden, um sicherzustellen, dass unrichtige oder unvollständige Daten berichtet oder gelöscht werden.
- **Speicherdauer**: Personenbezogene Daten dürfen nur für die Dauer gespeichert werden, die für die Zwecke, für die sie verarbeitet werden, erforderlich ist. Nach Ablauf dieser Frist müssen die Daten gelöscht oder anonymisiert werden.

- **Integrität und Vertraulichkeit:** Personenbezogene Daten müssen angemessen geschützt werden, um unbefugten Zugriff, unbefugte Verarbeitung oder Offenlegung zu verhindern. Geeignete technische und organisatorische Maßnahmen müssen ergriffen werden, um die Sicherheit der Daten zu gewährleisten.
- **Rechenschaftspflicht:** Der Verantwortliche für die Datenverarbeitung muss die Einhaltung der Grundsätze sicherstellen und nachweisen können. Es müssen geeignete Maßnahmen und Protokolle implementiert werden, um die Erfüllung der DSGVO-Anforderungen zu dokumentieren.

Die Verarbeitung personenbezogener Daten ist immer nur dann zulässig, wenn mindestens eine der folgenden genannten Rechtsgrundlagen erfüllt ist. Darüber hinaus müssen bei der Verarbeitung personenbezogener Daten die genannten Grundsätze der Datenverarbeitung eingehalten werden.

- **Einwilligung:** Die Verarbeitung personenbezogener Daten ist zulässig, wenn die betroffene Person ihre ausdrückliche Einwilligung für einen oder mehrere bestimmte Zwecke der Verarbeitung gegeben hat. Die Einwilligung muss freiwillig, informiert und eindeutig sein. Die betroffene Person hat das Recht, ihre Einwilligung jederzeit zu widerrufen.
- **Vertragserfüllung:** Die Verarbeitung personenbezogener Daten kann erforderlich sein, um einen Vertrag mit der betroffenen Person zu erfüllen oder um vorvertragliche Maßnahmen auf Anfrage der betroffenen Person durchzuführen.
- **Rechtliche Verpflichtung:** Wenn die Verarbeitung personenbezogener Daten zur Erfüllung einer rechtlichen Verpflichtung erforderlich ist, der der Verantwortliche unterliegt, ist dies zulässig. Dies kann beispielsweise für steuerliche oder regulatorische Zwecke gelten.
- **Schutz lebenswichtiger Interessen:** Die Verarbeitung personenbezogener Daten kann zulässig sein, wenn dies erforderlich ist, um das Leben oder die körperliche Unversehrtheit einer Person zu schützen, insbesondere in Notfällen.
- **Wahrnehmung einer Aufgabe im öffentlichen Interesse oder in Ausübung öffentlicher Gewalt:** Die Verarbeitung personenbezogener Daten kann erforderlich sein, um eine Aufgabe wahrzunehmen, die im öffentlichen Interesse liegt oder bei der Ausübung öffentlicher Gewalt erfolgt, die dem Verantwortlichen übertragen wurde.
- **Berechtigtes Interesse:** Die Verarbeitung personenbezogener Daten kann aufgrund eines berechtigten Interesses des Verantwortlichen oder eines Dritten zulässig sein, sofern das Interesse nicht durch die Interessen oder Grundrechte und Grundfreiheiten der betroffenen Person außer Kraft gesetzt wird. Vor der Verarbeitung personenbezogener Daten auf dieser Grundlage muss eine Interessenabwägung durchgeführt werden.

Es besteht die Verpflichtung, ein Verzeichnis aller Verarbeitungstätigkeiten personenbezogener Daten zu führen. Dieses sogenannte Verarbeitungsverzeichnis ist ein zentrales Instrument zur Dokumentation und Transparenz der Datenverarbeitung innerhalb einer Organisation und sollte eine umfassende Übersicht über die Datenverarbeitungsprozesse bieten, indem es folgende Informationen bereitstellt (Hennrich 2023):

- Angaben zum Verantwortlichen: Name und Kontaktdaten des Verantwortlichen für die Datenverarbeitung sowie gegebenenfalls des Vertreters des Verantwortlichen.
- Verarbeitungszwecke: Eine Beschreibung der Zwecke, für die die personenbezogenen Daten verarbeitet werden.

- Kategorien betroffener Personen: Angabe der Kategorien von Personen, deren Daten verarbeitet werden, beispielsweise Kunden, Mitarbeiter oder Lieferanten.
- Kategorien personenbezogener Daten: Auflistung der verschiedenen Arten von personenbezogenen Daten, die verarbeitet werden, wie beispielsweise Name, Adresse, E-Mail-Adresse usw.
- Kategorien von Empfängern: Nennung der Empfänger oder Kategorien von Empfängern, an die personenbezogene Daten weitergegeben werden können, wie beispielsweise Untertragnehmer, externe Dienstleister oder Behörden.
- Übermittlungen in Drittstaaten: Sofern personenbezogene Daten in Länder außerhalb der Europäischen Union übermittelt werden, müssen diese Übermittlungen angegeben werden.
- Speicherfristen: Angabe der geplanten Dauer, für die die personenbezogenen Daten gespeichert werden, oder, falls dies nicht möglich ist, der Kriterien zur Festlegung dieser Dauer.
- Technische und organisatorische Maßnahmen: Beschreibung der eingesetzten Sicherheitsmaßnahmen zum Schutz der personenbezogenen Daten.

Das Verarbeitungsverzeichnis dient der Transparenz und der Nachvollziehbarkeit der Datenverarbeitungstätigkeiten eines Unternehmens. Es unterstützt den Verantwortlichen dabei, seine Datenschutzpflichten zu erfüllen und gegenüber den Aufsichtsbehörden nachzuweisen, dass die Verarbeitung personenbezogener Daten rechtmäßig und im Einklang mit den Vorschriften der DSGVO erfolgt.

17.2.3 Auftragsverarbeitung

Eine Auftragsverarbeitung im Sinne der DSGVO liegt vor, wenn ein Verantwortlicher personenbezogene Daten an einen Auftragsverarbeiter weitergibt, der die Daten im Auftrag des Verantwortlichen verarbeitet. Dabei handelt es sich um eine spezifische Beziehung, in der der Verantwortliche die Kontrolle über die personenbezogenen Daten behält, während der Auftragsverarbeiter als Dienstleister handelt und die Daten gemäß den Anweisungen des Verantwortlichen verarbeitet. Kriterien, die erfüllt sein müssen, um eine Auftragsverarbeitung anzuerkennen, sind (Hennrich 2023):

- Zweck der Datenverarbeitung: Der Auftragsverarbeiter verarbeitet die personenbezogenen Daten nur im Auftrag des Verantwortlichen und gemäß dessen dokumentierten Anweisungen.
- Auftragsverarbeitungsvertrag: Zwischen dem Verantwortlichen und dem Auftragsverarbeiter muss ein schriftlicher Vertrag geschlossen werden, der bestimmte Aspekte der Verarbeitung regelt. Dieser Vertrag muss die Verpflichtungen des Auftragsverarbeiters, den Schutz der personenbezogenen Daten zu gewährleisten, und die Rechte und Pflichten des Verantwortlichen umfassen.
- Kontrolle über die Daten: Der Verantwortliche behält die Kontrolle über die personenbezogenen Daten und entscheidet über Zwecke und Mittel der Verarbeitung. Der Auftragsverarbeiter handelt nur nach den Anweisungen des Verantwortlichen.
- Sicherheitsmaßnahmen: Der Auftragsverarbeiter muss angemessene technische und organisatorische Maßnahmen ergreifen, um die Sicherheit der personenbezogenen Daten zu gewährleisten.

- Subunternehmer: Der Auftragsverarbeiter darf nur mit Zustimmung des Verantwortlichen Subunternehmer einsetzen. In diesem Fall muss der Auftragsverarbeiter die gleichen Verpflichtungen zum Datenschutz von den Subunternehmern einfordern.

Die Auftragsverarbeitung stellt eine rechtliche Beziehung zwischen Verantwortlichem und Auftragsverarbeiter dar, bei der der Verantwortliche weiterhin für die Einhaltung der Datenschutzbestimmungen verantwortlich ist. Der Auftragsverarbeiter agiert als Dienstleister und muss die Anweisungen des Verantwortlichen befolgen und den Datenschutz gewährleisten (Hennrich 2023).

Im Bereich des Cloud Computings gibt es zahlreiche Beispiele für Auftragsverarbeitung, bei denen Cloud-Dienstleister im Auftrag von Unternehmen personenbezogene Daten verarbeiten. Hier sind einige Beispiele:

- Storage/Backend as a Service (BaaS): Unternehmen können Cloud-Storage-Services nutzen, um personenbezogene Daten in der Cloud zu speichern. In diesem Fall agiert der Cloud-Anbieter als Auftragsverarbeiter und verarbeitet die Daten im Auftrag des Unternehmens.
- Software-as-a-Service (SaaS): Bei der Nutzung von SaaS-Diensten, wie beispielsweise Customer Relationship Management (CRM) oder Enterprise Resource Planning (ERP), werden personenbezogene Daten in der Cloud verarbeitet. Der SaaS-Anbieter handelt dann als Auftragsverarbeiter und verarbeitet die Daten im Auftrag des Unternehmens.
- E-Mail-Dienste: Viele Unternehmen nutzen cloud-basierte E-Mail-Dienste, bei denen die E-Mails und damit verbundene personenbezogene Daten in der Cloud gespeichert und verarbeitet werden. Der E-Mail-Anbieter agiert auch in diesen Fällen als Auftragsverarbeiter.
- Datenanalyse: Unternehmen können Cloud-Dienste zur Durchführung von Datenanalysen nutzen. Werden dabei personenbezogene Daten analysiert, verarbeitet der Cloud-Anbieter dabei Daten im Auftrag des Unternehmens, um beispielsweise Geschäftsanalysen oder Marketingkampagnen durchzuführen.
- Backups und Wiederherstellung: Storage Services werden häufig für das Erstellen von Backups und die Wiederherstellung von Daten verwendet. Diese umfassen oft auch personenbezogene Daten. Der Cloud-Anbieter übernimmt in diesem Fall die Verarbeitung der Daten im Auftrag des Unternehmens, um sicherzustellen, dass die Daten gesichert und bei Bedarf wiederhergestellt werden können.

Es ist also für eine rechtssichere Nutzung wichtig, dass Unternehmen, die Cloud-Dienste nutzen und dabei personenbezogene Daten verarbeiten lassen, sicherstellen, dass sie eine rechtsgültige Vereinbarung zur Auftragsverarbeitung (Auftragsverarbeitungsvertrag) mit dem Cloud-Dienstleister abschließen. Dadurch wird sichergestellt, dass der Cloud-Anbieter die Datenschutzanforderungen gemäß der DSGVO erfüllt und angemessene Sicherheitsmaßnahmen implementiert.

Werden dabei außereuropäische Cloud-Dienstleister genutzt, die nicht notwendig den Regeln der DGSVO folgen oder gegebenenfalls widersprechenden nationalen Regelungen unterworfen sind, sind die Ausführungen in Abschnitt 17.2.5 und Abschnitt 17.3 zu beachten.

17.2.4 Datenschutz-Folgenabschätzungen

Datenschutz-Folgenabschätzungen (engl.: Data Protection Impact Assessment, DPIA) sind ein Instrument zur Abschätzung möglicher Auswirkungen geplanter Datenverarbeitungsvorgänge auf den Datenschutz. Eine Folgenabschätzung dient dazu, Risiken für die Rechte und Freiheiten der Betroffenen im Zusammenhang mit der Verarbeitung personenbezogener Daten zu ermitteln und geeignete Maßnahmen zur Risikominimierung zu ergreifen. Eine Datenschutz-Folgenabschätzung ist nach DGSVO immer in folgenden Fällen erforderlich:

- Systematische und umfangreiche Bewertung von Personen: Wenn eine Art der Verarbeitung, einschließlich Profiling, eine umfangreiche und systematische Bewertung betroffener Personen ermöglicht, insbesondere wenn Entscheidungen auf automatisierter Verarbeitung basieren, die rechtliche Auswirkungen auf die betroffenen Personen haben.
- Umfangreiche Verarbeitung besonderer Kategorien personenbezogener Daten: Wenn besondere Kategorien personenbezogener Daten gemäß Artikel 9 Absatz 1 der DSGVO in großem Umfang verarbeitet werden, wie beispielsweise Gesundheitsdaten oder Daten über die ethnische Herkunft.
- Systematische Überwachung öffentlich zugänglicher Bereiche: Wenn eine systematische und umfangreiche Überwachung öffentlich zugänglicher Bereiche durchgeführt wird, beispielsweise durch Videoüberwachung.
- Großflächige Verarbeitung personenbezogener Daten: Wenn eine Verarbeitung personenbezogener Daten in großem Umfang erfolgt und die Verarbeitung einen erheblichen Einfluss auf betroffene Personen hat.

Ein Beispiel für eine großflächige Verarbeitung personenbezogener Daten könnte ein intelligentes Verkehrssystem in einer Stadt sein, das eine Vielzahl von personenbezogenen Daten erfasst und verarbeitet. Hier sind einige Aspekte, die in diesem Zusammenhang betrachtet werden müssten:

- Verkehrsüberwachung: Das System erfasst Daten von verschiedenen Sensoren wie Kameras, Verkehrsleitsystemen, Induktionsschleifen usw. Diese Sensoren sammeln Informationen über Fahrzeuge und deren Bewegungen, einschließlich Kennzeichen, Geschwindigkeit, Standort und Fahrtrichtung.
- Fahrzeugerkennung: Das System kann automatisch Fahrzeuge identifizieren und kategorisieren, beispielsweise anhand von Kennzeichenerkennungstechnologien. Dadurch können personenbezogene Daten der Fahrzeughalter erfasst und verarbeitet werden.
- Verkehrsanalysen: Das System analysiert die gesammelten Daten, um Verkehrsmuster, Engpässe und Verkehrsflüsse zu identifizieren. Dies kann auch indirekt personenbezogene Daten umfassen, wie beispielsweise Informationen über die Geschwindigkeit oder die Verweildauer von Fahrzeugen an bestimmten Orten.
- Öffentliche Verkehrsmittel: Das System kann auch Daten von öffentlichen Verkehrsmitteln wie Bussen oder Bahnen erfassen und verarbeiten. Hierbei können personenbezogene Daten von Fahrgästen oder Fahrern, wie etwa Ticketinformationen oder Zeitstempel der Nutzung, erfasst werden.

In einem solchen Szenario wäre eine Datenschutz-Folgenabschätzung erforderlich, um die Auswirkungen auf die Privatsphäre und die Rechte der betroffenen Personen zu bewerten.

Die Risiken und Maßnahmen zur Gewährleistung des Datenschutzes würden ermittelt und angemessene Schutzvorkehrungen implementiert, um die Rechte und Freiheiten der betroffenen Personen zu wahren.

Eine Datenschutz-Folgenabschätzung umfasst somit immer die Bewertung der **Notwendigkeit und Verhältnismäßigkeit** der Datenverarbeitung, die Bewertung der Risiken für die Rechte und Freiheiten der betroffenen Personen, die Ermittlung von Maßnahmen zur Risikominderung und die Konsultation der Datenschutzaufsichtsbehörde, wenn das Risiko hoch ist und keine geeigneten Maßnahmen ergriffen werden können. Die Durchführung einer Datenschutz-Folgenabschätzung ermöglicht es Unternehmen, Risiken frühzeitig zu erkennen, die Einhaltung der Datenschutzbestimmungen sicherzustellen und geeignete Maßnahmen zum Schutz der Rechte der betroffenen Personen zu ergreifen.

17.2.5 Internationale Datentransfers in Drittländer

Die DGSVO enthält spezifische Bestimmungen und Anforderungen für die Übermittlung personenbezogener Daten in Drittländer außerhalb der Europäischen Union (EU) und des Europäischen Wirtschaftsraums (EWR). Dabei legt die DSGVO besonderen Wert auf den Schutz personenbezogener Daten, unabhängig davon, ob sie innerhalb der EU oder in Drittländern verarbeitet werden. Sie zielt darauf ab, einen angemessenen Datenschutz und die Rechte der betroffenen Personen zu gewährleisten, auch bei grenzüberschreitenden Datenübermittlungen. Unternehmen sollten die Bestimmungen und Anforderungen der DSGVO in Bezug auf Datenübermittlungen in Drittländer daher geeignet berücksichtigen und umsetzen, um den Datenschutz zu gewährleisten. Diese Bestimmungen und Anforderungen umfassen:

- Angemessenheitsbeschluss: Die DSGVO ermöglicht die Übermittlung personenbezogener Daten in Drittländer, für die die Europäische Kommission einen sogenannten Angemessenheitsbeschluss erlassen hat. Ein solcher Beschluss besagt, dass das Datenschutzniveau in dem betreffenden Drittland als angemessen gilt und ein ausreichender Schutz für personenbezogene Daten gewährleistet ist. In diesem Fall können Daten ohne weitere Maßnahmen übertragen werden.
- Übermittlung auf Grundlage geeigneter Garantien: Wenn ein Drittland keinen Angemessenheitsbeschluss hat, kann die Datenübermittlung auf Grundlage geeigneter Garantien erfolgen. Hierzu gehören beispielsweise die Verwendung von Standardvertragsklauseln (SCCs) oder Binding Corporate Rules (BCR), die zwischen dem Datenexporteur und dem Datenimporteur vereinbart werden.
- Ausnahmen und spezifische Bedingungen: Die DSGVO enthält auch bestimmte Ausnahmen oder spezifische Bedingungen für die Übermittlung personenbezogener Daten in Drittländer. Dazu gehört beispielsweise die Einwilligung der betroffenen Person, die Erfüllung eines Vertrags mit der betroffenen Person, die Wahrnehmung öffentlicher Interessen oder die Durchführung rechtlicher Ansprüche.
- Zusätzliche Schutzmaßnahmen: Die DSGVO erlaubt es den Aufsichtsbehörden, zusätzliche Schutzmaßnahmen zu erlassen, um den Transfer personenbezogener Daten in Drittländer zu ermöglichen. Solche Schutzmaßnahmen können beispielsweise Standarddatenschutzklauseln, Verhaltensregeln oder Zertifizierungsmechanismen umfassen.

- Einzelfallprüfung: In einigen Fällen kann es erforderlich sein, eine Einzelfallprüfung durchzuführen, um sicherzustellen, dass angemessene Schutzmaßnahmen getroffen werden, wenn Daten in Drittländer übertragen werden. Dabei müssen die spezifischen Umstände der Datenübermittlung und die Datenschutzrisiken berücksichtigt werden.

Die Europäische Union (EU) hat Angemessenheitsbeschlüsse für bestimmte Drittländer erlassen (allerdings nicht den USA). Diese Liste umfasst Länder wie (Stand, August 2023):

- Andorra
- Argentinien
- Großbritannien (sowie Guernsey, Isle of Man, Jersey)
- Israel
- Japan
- Kanada (nur für kommerzielle Organisationen, nicht für öffentliche Stellen)
- Neuseeland
- Schweiz
- Südkorea
- Uruguay

Ferner hat die Europäische Kommission Standardvertragsklauseln (Standard Contractual Clauses, SCCs) entwickelt, die in verschiedenen Dokumenten veröffentlicht und kontinuierlich aktualisiert werden. Diese SCCs berücksichtigen die Anforderungen von SCHREMS II und den Schutz personenbezogener Daten und umfassen:

- SCCs für den Datentransfer zwischen Datenverantwortlichen und Datenverarbeitern: Dieses Dokument enthält Standardvertragsklauseln für den Datentransfer zwischen einem Datenverantwortlichen in der EU und einem Datenverarbeiter außerhalb der EU.
- SCCs für den Datentransfer zwischen zwei Datenverantwortlichen: Dieses Dokument enthält Standardvertragsklauseln für den Datentransfer zwischen zwei unabhängigen Datenverantwortlichen in und außerhalb der EU.
- SCCs für den Datentransfer von Datenverarbeitern an Subunternehmer: Dieses Dokument enthält Standardvertragsklauseln für den Datentransfer von einem Datenverarbeiter in der EU an einen Subunternehmer außerhalb der EU.

Die SCCs können spezifische Klauseln und Bedingungen enthalten, die an die jeweilige Situation und den Datentransfer angepasst werden müssen. Die genauen Inhalte der SCCs können in den offiziellen Veröffentlichungen der Europäischen Kommission nachgelesen werden.

Es wird empfohlen, gegebenenfalls Rechtsberatung in Anspruch zu nehmen, um sicherzustellen, dass die verwendeten SCCs den spezifischen Anforderungen und Bedürfnissen des Unternehmens entsprechen und den Datenschutzstandards gerecht werden.



Entsprechende Links zu den Standard Contract Clauses (SCC) finden sich auf der Website zum Buch.

■ 17.3 Europäischer Datenschutz und Drittländer

Es wurde bereits erwähnt, dass es zwar Angemessenheitsbeschlüsse für bestimmte Drittländer gibt, diese Liste der Drittländer allerdings ausgerechnet nicht die USA beinhaltet. In den USA sitzen aber nun mal die maßgeblichen Cloud-Provider wie AWS, Azure, IBM oder Google. Wenn es daher um den internationalen Datenschutz im Kontext von Cloud Computing geht, wird häufig mit den Akronymen SCHREMS I + II, CLOUD Act und DSGVO/GDPR auf eine komplexe und schwer zu durchschauende Rechtslage verwiesen, die es nahezu unmöglich zu machen scheint, insbesondere personenbezogene Daten bei US-amerikanischen Cloud-Anbietern zu verarbeiten. Aber stimmt das? Und was ist genau damit gemeint?

SCHREMS II bezieht sich auf ein Urteil des Europäischen Gerichtshofs (EuGH) aus dem Jahr 2020 zum Datenschutz und zum internationalen Datentransfer. Der Name SCHREMS II leitet sich vom Kläger in dieser Rechtssache, dem österreichischen Datenschutzaktivisten Max Schrems, ab. Die Entscheidung des EuGH betrifft den Datenschutz bei der Übermittlung personenbezogener Daten aus der Europäischen Union (EU) in Drittländer, insbesondere in die USA. Sie betrifft den rechtlichen Rahmen, der den Austausch personenbezogener Daten zwischen der EU und den USA regelt.

Die vorangegangene Entscheidung des EuGH aus dem Jahr 2015, bekannt als SCHREMS I, erklärte bereits das Safe-Harbor-Abkommen zwischen der EU und den USA für ungültig. Das Safe-Harbor-Abkommen ermöglichte die Übermittlung personenbezogener Daten aus der EU an Unternehmen in den USA, die sich zur Einhaltung bestimmter Datenschutzstandards verpflichteten. Die Ungültigkeitserklärung des Safe-Harbor-Abkommens führte zur Einführung des EU-US Privacy Shield, eines neuen Datenschutzrahmens für Datentransfers zwischen der EU und den USA.

Mit der Entscheidung SCHREMS II erklärte der EuGH dann allerdings auch das Abkommen EU-US Privacy Shield für ungültig. Der EuGH führte aus, dass das Datenschutzniveau in den USA den Anforderungen der EU nicht ausreichend entspreche und die Übermittlung personenbezogener Daten in die USA ein Risiko für die Datenschutzrechte der Betroffenen darstelle. Insbesondere bemängelte der EuGH die fehlende Beschränkung des Zugriffs der US-Behörden auf personenbezogene Daten von EU-Bürgern.

Die Entscheidung SCHREMS II hat somit erhebliche Auswirkungen auf Unternehmen, die personenbezogene Daten aus der EU in Drittländer, insbesondere in die USA, übermitteln. Es werden strengere Anforderungen an den Datenschutz und den internationalen Datentransfer gestellt. Unternehmen müssen seitdem geeignete Mechanismen für die Datenübermittlung implementieren, wie z. B. Standardvertragsklauseln (SCC) oder Binding Corporate Rules (BCR), um sicherzustellen, dass der Datenschutz bei der Übermittlung personenbezogener Daten gewährleistet ist, um die Vorgaben des EuGH umzusetzen und um angemessene Maßnahmen zu ergreifen, um die Datenschutzrechte der betroffenen Personen zu wahren.

17.3.1 Probleme am Beispiel des CLOUD Act

Da die meisten der Hyperscaler US-amerikanische Unternehmen sind, ist die US-amerikanische Gesetzgebung offenkundig ein nennenswerter Faktor in diesem Kontext. Ähnliche Probleme wie zwischen DSGVO und US CLOUD Act sind auch bei Alibaba und der chinesischen Rechtslage zu erwarten. Da China zudem kein Rechtsstaat im westlichen Verständnis ist, ist hier die Lage sogar noch unübersichtlicher, zumal Alibaba selber kaum Angaben zu Compliances macht (vgl. Tabelle 17.1). Der US CLOUD Act kann daher als Beispiel für international nicht vollständig aufeinander abgestimmte Regularien gelten.

Der CLOUD Act (Clarifying Lawful Overseas Use of Data Act) ist ein US-amerikanisches Gesetz, das 2018 verabschiedet wurde – unter anderem als Reaktion auf SCHREMS I des EuGH. Ziel des CLOUD Act ist es, den Zugang von US-Strafverfolgungsbehörden zu elektronischen Daten zu regeln, die von US-Unternehmen gespeichert werden, auch wenn sich diese Daten außerhalb der Vereinigten Staaten befinden. Der CLOUD Act ermöglicht somit US-Strafverfolgungsbehörden den Zugriff auf elektronische Daten, unabhängig davon, ob sich diese Daten in den USA oder im Ausland befinden. Das Gesetz erlaubt auch den Abschluss bilateraler Abkommen zwischen den USA und anderen Ländern, um den Zugriff auf Daten zu erleichtern. Dies hat Auswirkungen auf Cloud-Anbieter und Unternehmen, die elektronische Daten speichern und verarbeiten, insbesondere wenn die Anbieter der US-amerikanischen Rechtsprechung unterliegen. Das Gesetz kann somit zur Herausgabe von Daten führen, auch wenn diese außerhalb der USA gespeichert sind und möglicherweise den Datenschutzgesetzen anderer Länder unterliegen. Der CLOUD Act hat kontroverse Diskussionen ausgelöst, da er Bedenken hinsichtlich des Datenschutzes, der nationalen Souveränität und der möglichen Verletzung von Datenschutzgesetzen anderer Länder aufwirft. Kritiker befürchten, dass das Gesetz ermögliche den Zugriff auf Daten ohne ausreichende rechtliche Schutzmechanismen und gefährde die Privatsphäre des Einzelnen. Da der CLOUD Act ein US-amerikanisches Gesetz ist, regelt es primär die Beziehung zwischen US-Strafverfolgungsbehörden und US-Unternehmen. Die konkreten Auswirkungen des Gesetzes auf Unternehmen und die Datenhoheit können aber je nach Rechtsordnung und internationalen Vereinbarungen durchaus unterschiedlich sein.

Folgende Aspekte sollten dabei berücksichtigt werden:

- Rechtsgrundlage für den Zugriff: Gemäß dem CLOUD Act können US-Strafverfolgungsbehörden von Cloud-Dienstleistern Zugriff auf elektronische Daten verlangen, selbst wenn sich diese außerhalb der USA befinden. Dies kann durch eine gerichtliche Anordnung oder ein Abkommen zwischen den USA und einem anderen Land erfolgen.
- Auswirkungen auf Datenschutzgesetze anderer Länder: Da der CLOUD Act den Zugriff auf Daten unabhängig von deren Speicherort ermöglicht, können Cloud-Dienstleister gezwungen sein, Daten herauszugeben, die möglicherweise den Datenschutzgesetzen anderer Länder unterliegen. Dies kann zu Konflikten zwischen den Datenschutzvorschriften verschiedener Gerichtsbarkeiten führen.
- Bilaterale Abkommen: Der CLOUD Act ermöglicht den Abschluss bilateraler Abkommen zwischen den USA und anderen Ländern. Diese Abkommen sollen den Zugriff auf Daten erleichtern und den rechtlichen Rahmen für den Austausch von Informationen zwischen den Strafverfolgungsbehörden schaffen.
- Transparenz und Benachrichtigung: Der CLOUD Act enthält Bestimmungen, die den Cloud-Dienstleistern erlauben, über Anfragen nach Datenzugriff von Strafverfolgungsbehörden zu

informieren. Dies soll eine gewisse Transparenz gewährleisten, jedoch können bestimmte Umstände Ausnahmen von der Benachrichtigungspflicht vorsehen.

Die konkreten Auswirkungen des CLOUD Act hängen von verschiedenen Faktoren ab, wie der Rechtsprechung, den Vereinbarungen zwischen den USA und anderen Ländern sowie den spezifischen rechtlichen Bestimmungen und Datenschutzgesetzen in den betroffenen Gerichtsbarkeiten. Unternehmen, die Cloud-Dienste nutzen oder Daten speichern, sollten die Auswirkungen des CLOUD Act in Bezug auf ihre spezifische Situation und die geltenden Datenschutzgesetze sorgfältig prüfen und gegebenenfalls rechtlichen Rat einholen.

17.3.2 Lösungen trotz SCHREMS I + II

Obwohl es zum aktuellen Zeitpunkt kein spezifisches bilaterales Abkommen zwischen der Europäischen Union (EU) und den Vereinigten Staaten im Kontext des CLOUD Act gibt, ermöglicht der CLOUD Act jedoch grundsätzlich den Abschluss bilateraler Abkommen zwischen den USA und anderen Ländern, um den Zugriff auf elektronische Daten zu erleichtern.

Die Europäische Union und die Vereinigten Staaten haben verschiedene Mechanismen und Vereinbarungen zur Regelung des Datenflusses und des Datenschutzes. Das bekannteste Abkommen war das EU-US Privacy Shield, das als Mechanismus für den sicheren und datenschutzkonformen Datentransfer zwischen der EU und den USA diente. Das Privacy Shield wurde jedoch mit SCHREMS II im Jahr 2020 vom EuGH für ungültig erklärt.

Derzeit beruht der Datentransfer zwischen der EU und den USA hauptsächlich auf den Standardvertragsklauseln (Standard Contractual Clauses, SCCs), die von der Europäischen Kommission erstellt wurden. SCCs sind Standardverträge, die zwischen Datenexporteuren in der EU und Datenimporteuren außerhalb der EU abgeschlossen werden können, um den Datenschutz zu gewährleisten. Die Verwendung von SCCs erfordert jedoch eine sorgfältige Prüfung der rechtlichen und datenschutzrechtlichen Bedingungen.

Europäische Unternehmen stehen somit vor der Herausforderung, sowohl den Anforderungen von SCHREMS II als auch dem CLOUD Act gerecht zu werden. Hier sind einige Möglichkeiten, wie europäische Unternehmen diese Anforderungen erfüllen können:

- Verwendung von Standardvertragsklauseln (SCCs): Europäische Unternehmen können SCCs verwenden, um den internationalen Datentransfer zu regeln. SCCs sind von der Europäischen Kommission entwickelte Standardverträge, die bestimmte Datenschutzbestimmungen enthalten und den rechtlichen Schutz personenbezogener Daten gewährleisten sollen. SCCs können verwendet werden, um den Datentransfer sowohl mit den USA als auch mit anderen Ländern abzusichern.
- Binding Corporate Rules (BCRs): BCRs sind interne Datenschutzrichtlinien, die von multinationalen Unternehmen entwickelt werden können, um den internationalen Datentransfer innerhalb der Unternehmensgruppe zu regeln. BCRs erfordern die Genehmigung der zuständigen Datenschutzbehörden und bieten eine rechtliche Grundlage für den sicheren Datentransfer.
- Nutzung lokaler Cloud-Anbieter: Europäische Unternehmen können erwägen, lokale Cloud-Anbieter zu nutzen, die ihre Infrastruktur und Dienste innerhalb der EU bereitstellen. Dies kann dazu beitragen, den Datentransfer außerhalb der EU zu minimieren und somit den Anforderungen von SCHREMS II und dem CLOUD Act gerecht zu werden.

- Datensparsamkeit und Anonymisierung: Unternehmen können den Umfang der übertragenen personenbezogenen Daten begrenzen, indem sie das Prinzip der Datensparsamkeit anwenden. Darüber hinaus können sie Techniken wie Anonymisierung oder Pseudonymisierung verwenden, um personenbezogene Daten zu schützen und das Risiko von Datenschutzverletzungen zu verringern.

Angesichts der Komplexität und Dynamik von Datenschutzgesetzen und -vorschriften ist es ratsam, rechtliche Beratung in Anspruch zu nehmen und regelmäßige Compliance-Überprüfungen durchzuführen. Rechtsberater mit Fachwissen im Datenschutz und Kenntnis der Art der verarbeiteten Daten, der Geschäftstätigkeit und der internen Datenschutzrichtlinien können dabei unterstützen, die Anforderungen von SCHREMS II, dem CLOUD Act und anderen einschlägigen Datenschutzbestimmungen zu verstehen und geeignete Maßnahmen zur Einhaltung umzusetzen.

Es bleibt abzuwarten, ob in Zukunft bilaterale Abkommen zwischen der EU und den USA speziell im Zusammenhang mit dem CLOUD Act abgeschlossen werden. Der Fokus der EU liegt darauf, Datenschutzstandards und den Schutz der Privatsphäre zu gewährleisten, während die USA ihre Strafverfolgungsbefugnisse stärken möchten. Aktuell wird dieses Spannungsfeld weitestgehend auf dem Rücken der Unternehmen ausgetragen, die allerdings „Krücken“ wie SCCs an die Hand bekommen haben, diese Probleme rechtssicher untereinander regeln zu können. Dennoch führt dies immer wieder zu Herausforderungen und Diskussionen, die in weiteren Verhandlungen und Abkommen zwischen der EU und den USA hoffentlich irgendwann ausgeräumt werden können.

■ 17.4 Zusammenfassung

Die regulatorischen Anforderungen im Kontext der Sicherheit von Cloud-native Anwendungen sind sowohl durch europäische als auch internationale Standards, insbesondere US-Standards, gegeben. Dabei hat sich in einigen Köpfen der Mythos festgesetzt, dass Datenverarbeitung nach deutschem bzw. europäischem Recht mit internationalen Cloud Providern (den sogenannten US-Hyperscalern) kaum möglich ist. Das ist in dieser allgemeinen Aussage jedoch falsch und eben ein Mythos!

Grundsätzlich sind verschiedene rechtliche Regularien, Compliance- und Zertifizierungsvorgaben, die in der Cloud-Industrie relevant sind, zu berücksichtigen, um die Sicherheit und den Datenschutz (nicht nur) Cloud-native Anwendungen und Dienste zu gewährleisten. Das Gebiet ist vielfältig und komplex. Daher wird die Berücksichtigung von Spezialliteratur empfohlen (Hennrich 2023), und dieses Buch erhebt keinerlei Anspruch auf Vollständigkeit. Dieses Kapitel konnte nur einen groben Überblick und Eindruck gegeben. Für den deutschen und europäischen Markt sind jedoch für den Leser sicher erst einmal die folgenden Standards von übergeordneter Bedeutung:

- Standards des Bundesamts für Sicherheit in der Informationstechnik (BSI-Grundschutz, C5): Das Bundesamt für Sicherheit in der Informationstechnik (BSI) hat verschiedene Standards entwickelt, darunter den Grundschutz (BSI-GS 2023) und den C5-Standard (BSI C5 2020), die Sicherheitsanforderungen für Cloud-Dienstleister festlegen.

- ISO 27001: Die ISO 2700x Serie ist eine internationale Norm für Informationssicherheitsmanagementsysteme, die in der Cloud-Industrie weit verbreitet ist. Hier sind insbesondere die folgenden drei Standards zu nennen (ISO 27001, ISO 27017, ISO 27018).

Besonderes Augenmerk ist ferner auf die regulatorischen Anforderungen, die sich insbesondere aus der Datenschutz-Grundverordnung (DSGVO bzw. GDPR) ergeben, zu legen. Einen sehr detaillierten und zu empfehlenden Überblick liefert hier (Hennrich 2023). Die DSGVO legt grundsätzlich nur die Grundsätze für die Verarbeitung **personenbezogener** Daten fest. Personenbezogene Daten gemäß DSGVO sind Informationen, die sich auf eine identifizierte oder identifizierbare natürliche Person beziehen oder geeignet sind diese (gegebenenfalls auch nur mittelbar) zu identifizieren. Dies können bspw. Name, Adresse, Geburtsdatum, IP-Adressen, Log-in-Zeiten, Aufenthaltsorte und ähnliche Daten sein. Diese Daten werden durch die DSGVO geschützt und dürfen nur unter bestimmten Bedingungen (wie bspw. Zustimmung durch den Nutzer) verarbeitet und übermittelt werden. In diesem Zusammenhang sind auch Anforderungen an die Auftragsverarbeitung durch Drittparteien, wie bspw. Cloud Service-Provider, zu berücksichtigen, die in der DGSVO geregelt sind. Zudem sind bei besonders sensiblen personenbezogenen Daten – wie bspw. Gesundheitsdaten – eventuell Datenschutz-Folgenabschätzungen vorzunehmen. Bei internationalen Datentransfers in Drittländer, die nicht dem EU-Rechtsraum (GDPR) unterliegen (bspw. alle US-Provider, die dem US CLOUD Act unterliegen), können europäische Unternehmen jedoch sogenannte vorbereitete Standardvertragsklauseln (SCCs) der Europäischen Kommission übernehmen, mittels derer sich auch der internationale Datentransfer rechtssicher im Rahmen der Formulierungen der SCCs regeln lässt.

Ergänzende Materialien

Auf der Website zum Buch finden sich zu diesem Kapitel ferner folgende Ergänzungsmaterialien:

- Übersichten und Links zu Cloud-Standards und Compliance-Vorgaben
- Links zu den Standard Contract Clauses (SCC) der Europäischen Kommission
- Handouts (PDF) und bearbeitbare PowerPoint Slides (CC0-Lizenz) zum Download für Dozenten und Trainer



<https://bit.ly/3Pgr8cP>

18

Schlussbemerkungen

Sollte der Leser sich streng sequenziell durch dieses Buch gearbeitet haben, so endet seine Cloud-native Reise nun hier. Leser, die eher „nachschlagend“ und zielgerichtet in Kapitel gesprungen sind, haben vielleicht noch das eine oder andere Kapitel vor sich. Beiden Gruppen ist vermutlich jedoch bereits aufgefallen, dass Cloud-native Anwendungen und Dienste eine komplexe Kategorie von Softwaresystemen sind, die durch vielfältige Technologien und Methodiken geprägt sind. Dort den Überblick zu behalten, fällt nicht immer ganz leicht, zumal die „Moden“ sich in diesem Umfeld auch durchaus zügig entwickeln.

In **Teil I** wurden die Grundlagen behandelt. Es wurde die Basisterminologie wie beispielsweise IaaS, PaaS oder SaaS sowie Public, Private, Community oder Hybrid Clouds eingeführt. Es wurden aber auch die Grundlagen der Cloud-Ökonomie in Form einer Art Crash-Kurs erläutert. Insbesondere das Pay-as-you-go-Prinzip macht es für Cloud-Nutzer wirtschaftlich erstrebenswert, Systeme zu betreiben, die aus möglichst kleinen Komponenten bestehen, die nur für möglichst kurze Zeiträume ausgeführt werden und damit zu bezahlende Ressourcen binden. Insbesondere periodische oder zufällige Lasten profitieren davon. Genau diese Lasten treten im echten Leben durchaus häufig auf und machen daher Cloud Computing oft zu einer sinnvollen Option. Diese Erkenntnis erklärt viele technische Lösungen, die aktuell das Cloud-native Umfeld dominieren. So ermöglichen z. B. Container (kleine, in sich geschlossene Komponenten ohne externe Abhängigkeiten) und Functions (Scale-to-Zero), genau diese wirtschaftlichen Gesetzmäßigkeiten auch technisch effektiv nutzen zu können.

Anhand Bild 3.1 kann der Leser noch einmal reflektieren, anhand welcher Route das Cloud-native Land dabei durchfahren wurde.

Teil II fokussierte vor allem die **DevOps-Prinzipien des Flow** und griff diese in Form eines **Everything as Code**-Ansatzes auf und folgte mehr einer Bottom-up-Philosophie, um die Einzelbausteine Cloud-nativer Anwendungen anwendungsbezogen einzuführen. Die DevOps-Erfordernisse an Cloud-native Anwendungen und Dienste machen es erforderlich zu automatisieren (Prinzipien des Flow, vgl. Abschnitt 3.1). Diese Art der Automatisierung erfolgt in Cloud-nativen Anwendungen üblicherweise mit Deployment-Pipelines, die in **Kapitel 6** behandelt wurden. **Kapitel 7** hat gezeigt, wie auch die Bereitstellung von Infrastruktur für den Betrieb von Cloud-nativen Anwendungen und Diensten mittels Infrastructure as Code-Ansätzen automatisiert und damit beschleunigt werden kann. Während die dafür genutzten virtuellen Maschinen noch recht „massiven“ Charakter haben, wurde in **Kapitel 8** gezeigt, wie „leichtgewichtigere“ Komponenten in Form von **Containern** genutzt werden können, um sowohl das Dependency-Management in Cloud-nativen Anwendungen zu vereinfachen, als auch die genannten ökonomischen Gesetzmäßigkeiten im Betrieb möglichst wirtschaftlich nutzen zu können. Die Minimierung von Komponenten in Cloud-nativen Anwendungen führt allerdings auch dazu, dass man mehr davon zu betreiben hat. Dies kann im Betrieb sehr

aufwendig werden. Mittels Plattformen wie **Kubernetes** (vgl. **Kapitel 9**) lässt sich aber auch hier der Betrieb komplexer Multi-Container-Anwendungen automatisieren. Das **Kapitel 10** befasste sich dann mit noch kleineren Komponenten – **Funktionen**. Diese können bis auf null skaliert werden (**Scale-to-Zero**) und reizen damit die wirtschaftlichen Prinzipien des Pay-as-you-go-Prinzips bis zum Maximum aus.

Teil III griff hingegen eher architekturelle Fragestellungen auf und folgt damit eher einem Top-down-Ansatz, der zum Ziel hat zu erläutern, wie man Cloud-native Anwendung systematisch plant und baut. Dabei ging das **Kapitel 12** insbesondere auf den **Microservice**-Architekturansatz ein, zeigte aber Konsequenzen, die sich aus dem funktionsbasierten Technologieansatz für **Serverless-Architekturen** ergeben. Das **Kapitel 13** griff vor allem die Erfordernisse der **DevOps-Prinzipien des Feedbacks** auf und hat gezeigt, wie man beobachtbare Systeme mittels einer systematischen Erfassung von **Telemetriedaten** gestalten kann. Dies kann durch manuelle **Whitebox-Instrumentierung** von Code zum Zwecke des Logging, des Monitoring oder des Tracing erfolgen. Es ist aber auch möglich, dies mittels **Service-Meshes** in einem **Blackbox-Ansatz** zu automatisieren. Letztlich zeigt das **Kapitel 14**, wie man mittels der Methode des **Domain-driven Designs** methodisch von einem Problemraum zu einer Lösung für Cloud-native Anwendungsarchitekturen kommen kann, die die genannten Rand- und Rahmenbedingungen berücksichtigt.

Teil IV ging auf die Sicherheit Cloud-nativer Systeme auf zwei Ebenen ein. Kapitel 16 hat sich dabei auf die technische Sicherheit entlang der Abwehr typischer Angriffsvektoren fokussiert und insbesondere die Härtung von Systemen auf Infrastrukturebene und auf Workload-Ebene (insbesondere containerisierte Workloads) fokussiert.

Kapitel 17 befasste sich hingegen mit regulatorischen Fragestellungen, die sowohl durch europäische als auch internationale (insbesondere US-amerikanische) Regulatorien unterschiedlicher Rechtsräume geprägt sind. Besonderes Augenmerk wurde dabei vor allem auf die regulatorischen Anforderungen, die sich aus der Datenschutz-Grundverordnung (DSGVO bzw. GDPR) ergeben, gelegt. In diesem Zusammenhang sind auch Anforderungen an die Auftragsverarbeitung durch Drittparteien wie bspw. Cloud Service Provider zu berücksichtigen, die in der DGSVO geregelt sind. Zudem sind bei besonders sensiblen personenbezogenen Daten – wie bspw. Gesundheitsdaten – ggf. Datenschutz-Folgeabschätzungen vorzunehmen. Bei internationalen Datentransfers in Drittländer, die nicht dem EU-Rechtsraum (GDPR) unterliegen (bspw. alle US-Provider, die dem US CLOUD Act unterliegen) können europäische Unternehmen jedoch sogenannte und vorbereitete Standardvertragsklauseln (SCC) der Europäischen Kommission übernehmen mittels derer sich auch der internationale Datentransfer rechtssicher im Rahmen der Formulierungen der SCC regeln lässt.

Leser, die dieses Buch eher zielgerichtet auf der Suche nach Antworten für ganz spezifische Fragestellungen durchgearbeitet haben, mögen dabei daher die einen oder anderen Zusammenhänge überlesen haben. Das ist überhaupt nicht schlimm. Der Autor liest selber meist genauso. Aber möglicherweise entsteht dabei der Wunsch, das eine oder andere Kapitel noch einmal in einem anderen Zusammenhang oder mit einem anderen Fokus zu lesen.

Um dabei etwas zu unterstützen, dient die folgende Tabelle 18.1, in der alle in diesem Buch behandelten Pattern und Best Practices aufgeführt sind. Dadurch finden sich relevante Stellen gegebenenfalls etwas zielgerichtet als rein über das Inhaltsverzeichnis. Auch Querbezüge zwischen Abschnitten werden dadurch etwas expliziter.

Tabelle 18.1 Cloud-native Patterns und Best Practices

| Konzept | Pattern/Best Practices | Abschnitt(e) | Anmerkungen |
|------------------------|--|--------------------------------|---|
| Workload | Statische Last | 2.3.1 | eher ungeeignet |
| Workload | Steigende/sinkende Last | 2.3.1 | ggf. geeignet |
| Workload | Zufällige Last | 2.3.1 | sehr geeignet |
| Workload | Einmalige/seltene Last | 2.3.1 | max. geeignet |
| Pay-as-you-go | Ressourceneffizienz | 2.3.2 | Reduktion der Komponenten-Größe und/oder Zuteilungsdauer |
| DevOps | Deployment-Pipelines | 3.1, 3.1.3, 6.1, 12.5.2 | Prinzipien des Flow (Automatisierung) |
| DevOps | Orchestrierungsplattformen | 3.3.2, 9, 9.3, 12.5.2 | Prinzipien des Flow (Automatisierung) |
| DevOps | Telemetriedaten-konsolidierung | 3.2, 3.2.1, 12.5.7, 13.1 | Prinzipien des Feedbacks (Observability) |
| DevOps | Release-Risiken minimieren | 3.3.1, 13.3.2 | Blue/Green Releases, Canary Releases, Rolling Updates, Feature-Schalter |
| Cloud-native | Scale-out! Don't scale-up | 4.2, 8.5.5, 12.4, 12.4.1 | Horizontale Skalierung |
| Cloud-native | Elastizität durch Auto-Skalierung | 4.2, 9.3.5, 10.3, 12.4, 12.4.1 | Horizontale Skalierung |
| Cloud-native | In sich geschlossene Komponenten (Container) | 4.2, 8, 8.3, 8.4 | Stateless |
| Cloud-native | Isolation zustandsbehafteter Komponenten | 4.2, 12.4.3 | Stateful |
| Cloud-native | Betrieb auf elastischen Plattformen | 3.3.2, 4.2, 9, 12.5.2 | z. B. Kubernetes |
| Pipeline | Deployment-Pipelines as Code | 3.1.3, 6.1, 12.5.2 | Automatisierung |
| Pipeline | Phasen Pipelines | 6.1.1 | Build – Test – Deploy |
| Pipeline | Gerichtete Pipelines | 6.1.2 | Beschleunigung von Pipelines |
| Pipeline | Hierarchische Pipelines | 6.1.3 | Komplexe Systeme, Monorepositorys |
| Pipeline | Steuerung von Pipelines | 6.1.4 | Build-Trigger, Umgebungsvariable, Environments |
| Continuous Integration | Git-Flow Branching | 6.2.1 | Tendenz zu vielen Branches |
| Continuous Integration | GitHub-Flow Branching | 6.2.2 | Nur für sehr einfache Projekte |

Tabelle 18.1 (Fortsetzung) Cloud-native Patterns und Best Practices

| Konzept | Pattern/Best Practices | Abschnitt(e) | Anmerkungen |
|------------------------|--------------------------------|-------------------|---|
| Continuous Integration | Trunk-basiertes Branching | 6.2.3 | Pragmatisches Branching |
| Infrastructure as Code | Hardware Virtualisierung | 7.1.1 | Para-/Voll-Virtualisierung |
| Infrastructure as Code | Immutable Infrastructure | 7.2.1 | Infrastrukturparadigma |
| Infrastructure as Code | Provisionierung | 7.2.2 | Deklarativ und imperativ, Push und Pull von Konfigurationen |
| Infrastructure as Code | Single-VM-Provisionierung | 7.2.3 | Entwicklungs- und Testumgebungen, z. B. Vagrant |
| Infrastructure as Code | Multi-VM-Provisionierung | 7.2.4 | Produktions- und Testinfrastrukturen, z. B. Terraform |
| Container | Prozessisolierung | 8.3, 8.3.5 | Kernel Namespaces, Process Capabilities, Control Groups, Union Filesystem |
| Container | Container as Code | 8.4, 8.5.1, 8.5.4 | Dockerfile, Codebase |
| Container | Umgebungsvariablen | 8.5.2 | Konfiguration von 12-Faktor-Apps |
| Container | Port-Binding | 8.5.3 | Komposition von 12-Faktor-Apps |
| Container | Skalierung über Prozesse | 8.5.5, 12.4.1 | Skalierung von 12-Faktor-Apps |
| Container | Stdout-Logging | 8.5.6 | Beobachtbarkeit von 12-Faktor-Apps |
| Container | Image-Shrinking | begleitende Labs | Command Chaining, Small Base Images |
| Scheduling | Bin Packing | 9.1.2.1 | Einfacher Algorithmus, z. B. Docker |
| Scheduling | Spreading | 9.1.2.1 | Einfacher Algorithmus, z. B. Kubernetes |
| Scheduling | Multidimensionales Scheduling | 9.1.2.2 | z. B. Dominant Resource Fairness (Mesos) |
| Scheduling | Kapazitätsbasiertes Scheduling | 9.1.2.3 | Angabe min. und max. Ressourcen (z. B. Yarn) |
| Scheduling | Monolithisches Scheduling | 9.1.3.1 | z. B. Kubernetes |
| Scheduling | 2-Level Scheduling | 9.1.3.2 | z. B. Mesos |
| Scheduling | Shared State Scheduling | 9.1.3.3 | z. B. Omega |
| Orchestration | Blueprints | 9.2.1, 9.3.2 | z. B. Manifest-Dateien in Kubernetes |
| Orchestration | Regelkreisbasierte Überwachung | 9.2.2 | z. B. Controller in Kubernetes (Desired vs. Current State) |

Tabelle 18.1 (Fortsetzung) Cloud-native Patterns und Best Practices

| Konzept | Pattern/Best Practices | Abschnitt(e) | Anmerkungen |
|---------------------|--|---------------------------------------|---|
| Orchestration | Deployment Workload | 9.1.1, 9.3.3.1, 9.3.4 | kontinuierlich verfügbare Services |
| Orchestration | Job Workload | 9.1.1, 9.3.3.2, 9.3.4 | einmalig/periodisch auszuführende Tasks |
| Orchestration | Daemon Workload | 9.1.1, 9.3.3.3, 9.3.4 | Auf allen Knoten einer Plattform auszuführende Hintergrund-Tasks |
| Orchestration | Stateful Set Workload | 9.1.1, 9.3.3.4, 9.3.4 | Eindeutig über Re-Schedulings hinaus identifizierbare Komponenten |
| Orchestration | Horizontale Skalierung (Ressourcenverbrauch) | 4.2, 8.5.5, 9.3.5, 12.4, 12.4.1 | Horizontal Pod Autoscaler |
| Orchestra-tion/FaaS | Horizontale Skalierung (ereignisbasiert) | 4.2, 8.5.5, 9.3.5, 10.3, 12.4, 12.4.1 | KEDA, FaaS |
| Orchestration | Service Exposing (plattformintern) | 9.3.6 | z. B. Kubernetes Service |
| Orchestration | Service Exposing (plattformextern) | 9.3.6 | z. B. Kubernetes Ingress |
| Orchestration | Health-Checking | 9.3.7, 12.5.6 | z. B. Kubernetes Probes |
| Orchestration | Volume Claiming | 9.3.8 | Anforderung von persistentem Storage |
| Orchestration | Workload-Isolation | 9.3.9 | Multi-Tenancy mittels Namespaces, Quotas, Network Policies |
| FaaS | Scale-to-Zero | 10 | Time-Sharing von schwach genutzten Ressourcen |
| FaaS | Funktionsbasierte Workloads | 10.1, 10.1.1 | Nanoservices |
| FaaS | Funktions-Trigger | 10.1.1 | Event-gesteuertes Computing Modell |
| FaaS | Plattformagnostische Funktionsdefinition | 10.2 | Lösung für (noch) fehlende Standardisierung |
| Microservice | You-build-it-you-run-it | 12, 12.1 | Produkt- statt Projektphilosophie |
| Microservice | Unabhängige Aktualisierbarkeit von Komponenten | 12.1, 12.5.5 | Techn. Voraussetzung für Microservices |
| Microservice | Lose Kopplung von Komponenten | 12.1 | Techn. Voraussetzung für Microservices |
| Microservice | Datenbankbasierte Integration | 4.2, 12.2.1, 12.4.3 | Shared State, oft enge Kopplung |
| Microservice | (g)RPC-basierte Integration | 12.2.2 | Request-Response, mittlere Kopplung |

Tabelle 18.1 (Fortsetzung) Cloud-native Patterns und Best Practices

| Konzept | Pattern/Best Practices | Abschnitt(e) | Anmerkungen |
|------------------------------|------------------------------------|---------------------------------|--|
| Microservice | REST-basierte Integration | 12.2.3 | Request-Response, geringe Kopplung |
| Microservice | Queueing | 12.2.4 | Load Balancing, minimale Kopplung |
| Microservice | Publish/Subscribe | 12.2.4 | Fan-out, minimale Kopplung |
| Microservice | Reactive Systems | 12.2.4 | Siehe auch Reaktives Manifest |
| Microservice | API-Versioning | 12.2.5, 14.2.4.2 | Mittel der letzten Wahl |
| Microservice | Circuit-Breaker | 12.3, 12.3.1, 12.5.6 | Architectural Safety Pattern |
| Microservice | Bulkhead | 12.3, 12.3.2, 12.5.6 | Architectural Safety Pattern |
| Microservice | Idempotente API-Operationen | 12.3, 12.3.3, 12.5.6 | Architectural Safety Pattern |
| Microservice | Load Balancing | 9.3.6, 12.4.1 | Performance Pattern |
| Microservice | Horizontale Skalierung | 12.4, 12.4.1, 12.5.4 | Performance Pattern |
| Microservice | Clientseitiges Caching | 12.4.4 | Performance Pattern: clientseitig, serverseitig, proxy-basiert |
| Microservice | Scaling for Reads | 4.2, 12.4.3, 12.4.3.1 | Stateful Scalability Pattern |
| Microservice | Scaling for Writes | 4.2, 12.4.3, 12.4.3.2 | Stateful Scalability Pattern |
| Microservice | CQRS | 4.2, 12.4.3, 12.4.3.3, 14.3.2.3 | Stateful Scalability Pattern |
| Microservices/ Serverless | API-Gateway | 12.6.2 | API-Endpunkt-Konsolidierung |
| Serverless | Choreography by Client-Devices | 12.5.4, 12.6.1 | Vermeidung des Double-Spending-Problems |
| Serverless | Backend as a Service | 12.6.1 | Dezentralisierung |
| Observability | Telemetriedaten-konsolidierung | 12.5.7, 13.1 | Single Source of Observability-Data |
| Observability | Telemetriedaten-visualisierung | 12.5.7, 13.1 | Birds-Eye View |
| Observability | Logging-Instrumentierung | 12.5.7, 13.2.1 | Whitebox Instrumentierung |
| Observability | Metrikinstrumentierung | 12.5.7, 13.2.2 | Whitebox-Instrumentierung |
| Observability | Tracing-Instrumentierung | 12.5.7, 13.2.3 | Whitebox-Instrumentierung |
| Service Mesh | Sidecar-Proxy | 12.5.7, 13.3 | Blackbox-Instrumentierung |
| Service Mesh | Quantitatives Traffic Splitting | 13.3.2 | für Canary Releases |
| Service Mesh | Inhaltsbasiertes Traffic Splitting | 13.3.2 | für Blue/Green oder A/B Releases |

Tabelle 18.1 (Fortsetzung) Cloud-native Patterns und Best Practices

| Konzept | Pattern/Best Practices | Abschnitt(e) | Anmerkungen |
|-----------------------------|---------------------------------------|--|---|
| Service Mesh | Resilienzkonfiguration | 12.3.1, 12.3.2, 12.5.6, 13.3.3 | Timeout-, Retry-, Circuit-Breaker-Settings |
| Service Mesh | Visualisierung von Verkehrstopologien | 12.5.7, 13.3.5 | Birds-Eye View |
| Problemraum | Core Domain | 14.2, 14.2.1.1 | DDD-Strategic Design |
| Problemraum | Supporting Domain | 14.2, 14.2.1.2 | DDD-Strategic Design |
| Problemraum | Generic Domain | 14.2, 14.2.1.3 | DDD-Strategic Design |
| Problemraum | Ubiquitous Language | 14.1, 14.2, 14.2.2 | DDD-Strategic Design |
| Problemraum | Bounded Context | 14.2, 14.2.3 | DDD-Strategic Design |
| Context Map | Partnership | 14.2.4, 14.2.4.1, 14.2.4.4 | DDD-Strategic Design |
| Context Map | Shared Kernel | 14.2.4, 14.2.4.1, 14.2.4.4 | DDD-Strategic Design |
| Context Map | Conformist | 14.2.4.2, 14.2.4.4 | DDD-Strategic Design |
| Context Map | Open Host System (OHS) | 12.2.5, 14.2.4.2, 14.2.4.4 | DDD-Strategic Design |
| Context Map | Anti-Corruption-Layer (ACL) | 14.2.4.2, 14.2.4.4 | DDD-Strategic Design |
| Context Map | Separate Ways | 14.2.4.3, 14.2.4.4 | DDD-Strategic Design |
| Lösungsraum | Extract-Transform-Load | 14.3, 14.3.1.1 | DDD-Tactical Design für Supporting Domains |
| Lösungsraum | Active Record | 14.3, 14.3.1.2 | DDD-Tactical Logik-Design für Supporting Domains |
| Lösungsraum | Domain Model | 12.5.1, 14.3, 14.3.1.3 | DDD-Tactical Logik-Design für Core Domains |
| Lösungsraum | Event Sourcing | 14.3, 14.3.1.4 | DDD-Tactical Logik-Design für Core Domains |
| Lösungsraum | Layered Architecture | 14.3, 14.3.2.1 | DDD-Tactical Architektur-Design für Supporting Domains |
| Lösungsraum | Ports+Adaptor Architecture | 14.3, 14.3.2.2 | DDD-Tactical Architektur-Design für Core Domains |
| Lösungsraum | CQRS Architecture | 4.2, 12.4.3, 12.4.3.3, 14.3, 14.3.2.3 | DDD-Tactical Architektur-Design für Core Domains |
| Härtung von Infrastrukturen | Toolgestütztes System Hardening | 2.1.1 | Angriffsvektor 1 |
| Härtung von Infrastrukturen | Kontinuierliche System-aktualisierung | 2.1.2 | Unattended Upgrades Angriffsvektor 1 |

Tabelle 18.1 (Fortsetzung) Cloud-native Patterns und Best Practices

| Konzept | Pattern/Best Practices | Abschnitt(e) | Anmerkungen |
|--|---|-----------------------|--|
| Härtung von Infrastrukturen | Sichere Authentifizierung (mittels SSH) | 2.1.3 | Angriffsvektor 2 |
| Härtung von Infrastrukturen | Sicherheitsgruppen, Firewalls | 2.1.5 | Angriffsvektor 1, Reduktion der Angriffsoberfläche |
| Überwachung von virtuellen Maschinen und Workloads | Verhaltensbasierte Intrusion Detection | 2.1.4, 2.1.4.1, 2.2.7 | Erkennung verdächtiger Aktivitäten, Audit-Protokollierung |
| Überwachung von virtuellen Maschinen und Workloads | Signaturbasierte Intrusion Detection | 2.1.4.2, 2.2.7 | Erkennung von Änderungen, Audit-Protokollierung |
| Überwachung von virtuellen Maschinen und Workloads | Log Forwarding | 2.1.4.3, 2.2.7 | Sicherung des Cloud-Forensic Trails |
| Härtung containerisierter Workloads | Ingress | 2.2.1 | Angriffsvektor 1, 2 SSL-Terminierung, Authentifizierung, Rate Limiting |
| Härtung containerisierter Workloads | Namespace-basierte Netzwerkisolation | 2.2.2 | Angriffsvektor 1, 2 Network Policies |
| Härtung containerisierter Workloads | Workload Policing | 2.2.6 | Angriffsvektor 1, 2 Open Policy Agent |
| Härtung containerisierter Workloads | Pod Hardening | 2.2.3 | Angriffsvektor 1, 2 Service-Accounts, Ressourcen Limitierung, Security Context, Pod Security Standards, |
| Härtung containerisierter Workloads | Volume Hardening | 2.2.5 | Angriffsvektor 1, 2 Verschlüsselung von Volumes |
| Härtung containerisierter Workloads | Erhöhung der Runtime Isolation | 2.2.4 | Angriffsvektor 1, 2 Hypervisor-, Kernel-, sandbox-basierte Container Runtimes |

Tabelle 18.1 (Fortsetzung) Cloud-native Patterns und Best Practices

| Konzept | Pattern/Best Practices | Abschnitt(e) | Anmerkungen |
|-------------------------------------|---|--|--|
| Sicherung der Software Supply Chain | Statische Software Composition Analysis (SCA) | 2.2.8.1 | Angriffsvektor 3, 4 In CI/CD Pipeline realisierbar |
| Sicherung der Software Supply Chain | Static Application Security Testing (SAST) | 2.2.8.2 | Angriffsvektor 3, 4 In CI/CD Pipeline realisierbar |
| Sicherung der Software Supply Chain | Kontinuierliches Schwachstellen-Scanning | 2.2.8.3 | Angriffsvektor 3, 4 Image Scanning, Manifest Scanning, Config Scanning |
| Cloud Compliance | ISO 9001 | 3.1.1 | Allgemeine Qualitätssicherung |
| Cloud Compliance | BSI IT-Grundschutz | 3.1.2 | Sicherer Betrieb von Rechenzentren |
| Cloud Compliance | BSI C5-Zertifizierung | 3.1.3 | Sicherer Betrieb von Cloud Anwendungen und Infrastrukturen sowie Plattformen |
| Cloud Compliance | Internationale Standards | 3.1.4, 3.1.5, 3.1.6, 3.1.7, 3.1.8, 3.1.9, 3.1.10, 3.1.11 | ISO 27001ff, CSA STAR, CISPE, SOC, FedRAMP, HIPAA, PCI DSS ... |
| DSGVO/GDPR Compliance | Personenbezogene Daten | 3.2.1 | Die DSGVO gilt nur für die Verarbeitung personenbezogener Daten. |
| DSGVO/GDPR Compliance | Grundsätze der Verarbeitung | 3.2.2 | Rechtmäßigkeit, Transparenz, Zweckbindung, Datenminimierung, Richtigkeit, Speicherdauer |
| DSGVO/GDPR Compliance | Auftragsverarbeitung | 3.2.3 | Zweck, Auftragsverarbeitungsvertrag, Kontrolle, Sicherheitsmaßnahmen |
| DSGVO/GDPR Compliance | Datenschutz-Folgeabschätzungen | 3.2.4 | Gesundheitsdaten, ethnische Herkunft, Überwachung öffentlich zugänglicher Bereiche, Bewertung von Personen |
| DSGVO/GDPR Compliance | Standard Contractual Clauses (SCC) der EU | 3.2.5, 3.3 | Internationale Datentransfers in nicht EU-Länder oder Länder mit Angemessenheitsbeschlüssen |

Literaturverzeichnis

- Adersberger, Josef, M.-Leander Reimer, Moritz Kammerer und Sonja Wegner. 2018. „Vorlesung Cloud Computing“. <https://github.com/qaware/cloudcomputing>.
- Aleroud, Ahmed und Lina Zhou: Phishing environments, techniques, and countermeasures: A survey, Computers & Security, Volume 68, 2017, <https://doi.org/10.1016/j.cose.2017.04.006>.
- Aroundel, John und Justin Domingus. 2019. *Cloud Native DevOps mit Kubernetes*. dpunkt.verlag.
- Bainomugisha, Engineer, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx und Wolfgang de Meuter. 2013. „A Survey on Reactive Programming“. *ACM Comput. Surv.* 45 (4). <https://doi.org/10.1145/2501654.2501666>.
- Balalaie, Armin, Abbas Heydarnoori und Pooyan Jamshidi. 2016. „Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture“. *IEEE Software* 33: 42–52.
- Baldini, Ioana, Paul C. Castro, Kerry Shih-Ping Chang, P. Cheng, Stephen J. Fink, Vatche Ishakian, N. Mitchell u. a. 2017. „Serverless Computing: Current Trends and Open Problems“. In *Research Advances in Cloud Computing*.
- Bastos, Joel und Pedro Araújo. 2019. *Hands-On Infrastructure Monitoring with Prometheus: Implement and scale queries, dashboards, and alerting across machines and containers*. Packt Publishing.
- Bondi, André B. 2000. „Characteristics of Scalability and Their Impact on Performance“. In *Proceedings of the 2nd International Workshop on Software and Performance*, 195–203. WOSP '00. New York, NY, USA: ACM. <https://doi.org/10.1145/350391.350432>.
- Brikman, Yevgeniy. 2019. *Terraform: Up and Running*. O'Reilly Media. <https://learning.oreilly.com/library/view/terraform-up/9781492046899/>.
- BSI: Cloud Computing Compliance Criteria Catalogue – C5:2020, https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/CloudComputing/Anforderungskatalog/2020/C5_2020.pdf, 2020, letzter Zugriff am 04.07.2023
- BSI: IT-Grundschutz Kompendium, Bundesamt für Sicherheit in der Informationstechnik, https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium/IT_Grundschutz_Kompendium_Edition2023.pdf, 2023, letzter Zugriff am 04.07.2023
- Burns, Brendan, Brian Grant, David Oppenheimer, Eric Brewer und John Wilkes. 2016. „Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade“. *Queue* 14 (1): 70–93. <https://doi.org/10.1145/2898442.2898444>.
- Calcote, Lee und Zack Butcher. 2019. *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe*. O'Reilly.
- Carlson, Lucas. 2013. *Programming for PaaS: A Practical Guide to Coding for Platform-As-A-Service*. O'Reilly Media.
- Chow, Michael, David Meisner, Jason Flinn, Daniel Peek und Thomas F. Wenisch. 2014. „The Mystery Machine: End-to-End Performance Analysis of Large-Scale Internet Services“. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*, 217–31. OSDI'14. USA: USENIX Association.

- CNCF. 2015. „Cloud Native Computing Foundation“. <https://www.cncf.io/about/faq/>.
- CNCF. 2019. „Service Mesh Interface (SMI)“. <https://smi-spec.io>.
- Cockburn, Alistair. 2005. „The Pattern: Ports and Adapters (Object Structural)“. <https://alistair.cockburn.us/hexagonal-architecture/>.
- Collins, Austin, Ganesh Radakrishnan und Bill Fine. 2015. „serverless framework“. <https://serverless.com>.
- Conway, Melvin E. 1968. „How do committees invent“. *Datamation* 14 (4): 28–31.
- CVE Program, MITRE Corporation, <https://www.cve.org>, letzter Zugriff am 03.07.2023
- Davis, Cornelia. 2019. *Cloud Native Patterns*. Manning.
- Erl, Thomas, Robert Cope und Armin Naserpour. 2015. *Cloud Computing Design Patterns*. Springer.
- Evans. 2003. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley.
- Fehling, Christoph, Frank Leymann, Ralph Retter, Walter Schupeck und Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer. <https://doi.org/10.1007/978-3-7091-1568-8>.
- Fielding, Roy Thomas. 2000. „REST: Architectural Styles and the Design of Network-based Software Architectures“. Doctoral dissertation, University of California, Irvine. <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- Fonseca, Rodrigo, George Porter, Randy H. Katz, Scott Shenker und Ion Stoica. 2007. „X-Trace: A Pervasive Network Tracing Framework“. In *Proc. of the 4th USENIX Conf. on Networked Systems Design & Implementation*, 20. NSDI'07. USA: USENIX Association.
- Forsgren-Velasquez, Nicole, Gene Kim, Nigel Kersten und Jez Humble. 2014. „DevOps Research and Assessment (DORA) research program“. <https://www.devops-research.com/research.html>.
- Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*. USA: Addison-Wesley Longman Publishing Co., Inc.
- Fowler, Martin. 2014. „Microservices – A Definition of this new Architectural Term“. <http://martinfowler.com/articles/microservices.html>.
- Fowler, Martin. 2020. „Patterns for Managing Source Code Branches“. <https://martinfowler.com/articles/branching-patterns.html>.
- Fox, Armando und Eric A Brewer. 1999. „Harvest, yield, and scalable tolerant systems“. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems*, 174–78. IEEE.
- Ghodsi, Ali, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker und Ion Stoica. 2011. „Dominant Resource Fairness: Fair Allocation of Multiple Resource Types“. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*, 323–36. NSDI'11. USA: USENIX Association.
- Gilbert, Seth und Nancy Lynch. 2002. „Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services“. *Acm Sigact News* 33 (2): 51–59.
- Hammant, Paul. 2020. *Trunk-Based Development And Branch By Abstraction*. Leanpub. <https://leanpub.com/trunk-based-development>.
- Hashimoto, Mitchell. 2013. *Vagrant: Up and Running*. O'Reilly Media. <https://learning.oreilly.com/library/view/vagrant-up-and/9781449336103/>.
- Heartfield, Ryan und George Loukas: A Taxonomy of Attacks and a Survey of Defence Mechanisms for Semantic Social Engineering Attacks. ACM Computing Surveys, 2015, <https://doi.org/10.1145/2835375>
- Hennrich, Thorsten: Cloud Computing nach der Datenschutz-Grundverordnung, O'Reilly, 2023
- Herbst, Nikolas Roman, Samuel Kounev und Ralf Reussner. 2013. „Elasticity in Cloud Computing: What It Is, and What It Is Not“. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, 23–27. San Jose, CA: USENIX.

- Hindman, Benjamin, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker und Ion Stoica. 2011. „Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center“. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*, 295–308. NSDI’11. USA: USENIX Association.
- Inzinger, C., S. Nastic, S. Sehic, M. Vögler, F. Li und S. Dustdar. 2014. „MADCAT: A Methodology for Architecture and Deployment of Cloud Application Topologies“. In *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*, 13–22. <https://doi.org/10.1109/SOSE.2014.9>.
- ISO 27001: Information Security Management Systems Requirements, <https://www.iso.org/standard/27001>, 2022, letzter Zugriff am 04.07.2023
- ISO 27017: Security techniques – Code of practice for controls to protect personally identifiable information processed in public cloud computing services
- ISO 27018: Security techniques – Code of practice for information security controls for cloud computing services
- Jain, Shashank Mohan. 2020. *Linux Containers and Virtualization: A Kernel Perspective*. Apress. <https://doi.org/10.1007/978-1-4842-6283-2>.
- Jonas, Eric, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar u. a. 2019. „Cloud Programming Simplified: A Berkeley View on Serverless Computing“. <https://arxiv.org/abs/1902.03383>.
- Julian, Mike. 2017. *Practical Monitoring: Effective Strategies for the Real World*. O'Reilly.
- Kane, Sean P. und Karl Matthias. 2018. *Docker: Up & Running: Shipping Reliable Containers in Production*. 2. Aufl. O'Reilly Media, Inc.
- Katzer, Jason. 2020. *Learning Serverless: Reliability, Availability, Monitoring, Testing and Security*. O'Reilly Media, Inc.
- Khatri, Anjali und Vikram Khatri. 2020. *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. O'Reilly.
- Khononov, Vladik. 2019. *What Is Domain-Driven Design?* O'Reilly.
- Kim, Gene, Humble Jez, Patrick Debois und John Willis. 2017. *Das DevOps Handbuch*. O'Reilly.
- Kolb, Stefan. 2018. „On the Portability of Applications in Platform as a Service“. University of Bamberg. <https://doi.org/10.20378/irbo-54102>.
- Kolb, Stefan. 2019. „PaaSfinder“. <https://paasfinder.org>.
- Kratzke, Nane. 2018. „A Brief History of Cloud Application Architectures“. *Applied Sciences* 8 (8). <https://doi.org/10.3390/app8081368>.
- Kratzke, Nane. 2020. „Volunteer Down: How COVID-19 Created the Largest Idling Supercomputer on Earth“. *Future Internet* 12 (6). <https://doi.org/10.3390/fi12060098>.
- Kratzke, Nane und Peter-Christian Quint. 2017. „Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study“. *Journal of Systems and Software* 126 (April): 1–16. <https://doi.org/10.1016/j.jss.2017.01.001>.
- Kuhn, Roland, Brian Hanafee und Jamie Allen. 2017. *Reactive Design Patterns*. 1. Aufl. USA: Manning Publications Co.
- Love, Robert. 2007. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, Inc.
- Love, Robert. 2010. *Linux Kernel Development*. 3. Aufl. Addison-Wesley Professional.
- Luksa, Marko. 2018. *Kubernetes in action*. Hanser Verlag.
- Martin, Andrew und Michael Hausenblas: Hacking Kubernetes – Threat-Driven Analysis and Defense. O'Reilly, 2021

- McGrath, M. P. 2012. *Understanding PaaS: Unleash the Power of Cloud Computing*. O'Reilly Media.
- Mell, Peter M. und Timothy Grance. 2011. „SP 800-145. The NIST Definition of Cloud Computing“. Gaithersburg, MD, USA: National Institute of Standards & Technology.
- Mendonca, N. C., P. Jamshidi, D. Garlan und C. Pahl. 2021. „Developing Self-Adaptive Microservice Systems: Challenges and Directions“. *IEEE Software* 38 (2): 70–79.
<https://doi.org/10.1109/MS.2019.2955937>.
- Microsoft. 2014. „KEDA: Kubernetes Event-driven Autoscaling“. The Linux Foundation. <https://keda.sh>.
- Millett, Scott und Nick Tune. 2015. *Patterns, Principles, and Practices of Domain-Driven Design*. Wiley.
- Morris, K. 2021. *Infrastructure as Code: Dynamic Systems for the Cloud Age*. O'Reilly Media.
<https://books.google.de/books?id=Wz2KzQEACAAJ>.
- Namiot, Dmitry und Manfred Sneps-Sneppe. 2014. „On micro-services architecture“. *Int. Journal of Open Information Technologies* 2 (9).
- National Security Agency: Kubernetes Hardening Guide (Cybersecurity Technical Report), 2022,
https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF
- National Vulnerability Database, NIST, <https://nvd.nist.gov>, letzter Zugriff am 03.07.2023
- Newman, Sam. 2015. *Building Microservices*. O'Reilly.
- OCI. 2015. „Open Container Runtime Specification“. <https://github.com/opencontainers/runtime-spec>.
- OCI. 2016a. „OCI Image Specification“. <https://github.com/opencontainers/image-spec>.
- OCI. 2016b. „OCI Runtime Specification“. <https://github.com/opencontainers/runtime-spec>.
- Öggl, Bernd und Michael Kofler. 2020. *Git: Projektverwaltung für Entwickler und DevOps-Teams. Inkl. Praxistipps und Git-Kommandoreferenz*. Rheinwerk Computing.
- Ongaro, Diego und John Ousterhout. 2014. „In Search of an Understandable Consensus Algorithm“. In *Proc. of the 2014 USENIX Conf. on USENIX Annual Technical Conference*, 305–20. USENIX ATC'14. USA: USENIX Association.
- OpenAPI. 2018. „The OpenAPI Specification“. <https://github.com/OAI/OpenAPI-Specification>.
- Pahl, C., A. Brogi, J. Soldani und P. Jamshidi. 2019. „Cloud Container Technologies: A State-of-the-Art Review“. *IEEE Transactions on Cloud Computing* 7 (3): 677–92.
<https://doi.org/10.1109/TCC.2017.2702586>.
- Parker, Austin, Daniel Spoonhower, Jonathan Mace, Ben Sigelman und Rebecca Isaacs. 2020. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly.
- Picard, Romain. 2018. *Reactive Programming with Python*. 1. Aufl. Birmingham, UK: Packt Publishing.
- Preston-Werner, Tom. 2013. „Semantic Versioning 2.0.0“. <http://semver.org>.
- Redmond, Eric und Jim R. Wilson. 2012. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf.
- Reiss, Charles, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz und Michael A. Kozuch. 2012. „Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis“. In *Proc. of the 3rd ACM Symp. on Cloud Computing*. SoCC '12. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2391229.2391236>.
- Richardson, Leonard, Mike Amundsen und Sam Ruby. 2013. *RESTful Web APIs*. O'Reilly Media, Inc.
- Scholl, Boris, Trent Swanson und Peter Jausovec. 2019. *Cloud Native*. O'Reilly.
- Schwarzkopf, Malte, Andy Konwinski, Michael Abd-El-Malek und John Wilkes. 2013. „Omega: Flexible, Scalable Schedulers for Large Compute Clusters“. In *Proc. of the 8th ACM European Conf. on Computer Systems*, 351–64. EuroSys '13. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2465351.2465386>.
- Sciaibarrà, Michele. 2019. *Learning Apache OpenWhisk*. O'Reilly Media, Inc.

- Shukla, Pranv und Sharat Kumar. 2019. *Learning Elastic Stack 7.0: Distributed search, analytics, and visualization using Elasticsearch, Logstash, Beats, and Kibana*. Packt Publishing.
- Sigelman, Benjamin H., Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan und Chandan Shambhag. 2010. „Dapper, a Large-Scale Distributed Systems Tracing Infrastructure“. Google, Inc. <https://research.google.com/archive/paper2010-1.pdf>.
- Stine, Matt. 2015. *Migrating to Cloud-Native Application Architectures*. O'Reilly.
- Vavilapalli, Vinod Kumar, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves u. a. 2013. „Apache Hadoop YARN: Yet Another Resource Negotiator“. In *Proc. of the 4th Annual Symp. on Cloud Computing*. SOCC '13. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2523616.2523633>.
- Verma, Abhishek, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune und John Wilkes. 2015. „Large-Scale Cluster Management at Google with Borg“. In *Proc. of the 10th European Conf. on Computer Systems*. EuroSys '15. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2741948.2741964>.
- Vernon, Vaughn. 2017. *Domain-Driven Design kompakt*. dpunkt.verlag.
- Villamizar, Mario, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas u. a. 2017. „Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures“. *Service Oriented Computing and Applications* 11 (2): 233–47.
- Weinman, Joe. 2011. „Mathematical Proof of the Inevitability of Cloud Computing“. https://cloud-native-computing.de/materials/Joe_Weinman_Inevitability_Of_Cloud.pdf.
- White, Tom. 2015. *Hadoop: The Definitive Guide*. 4. Aufl. O'Reilly Media, Inc.
- Wiggins, Adam. 2017. „The Twelve-Factor App“. <http://12factor.net/>.
- Yussupov, Vladimir, Uwe Breitenbücher, Frank Leymann und Michael Wurster. 2019. „A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools“. In *Proc. of the 12th IEEE/ACM Int. Conf. on Utility and Cloud Computing (UCC 2019)*, 229–40. ACM. <https://doi.org/10.1145/3344341.3368803>.
- Zambrano, Brian. 2018. *Serverless Design Patterns and Best Practices*. Packt Publishing.

Stichwortverzeichnis

Symbol

1 vCPU-Schwellen 22, 84, 142

3-Tier-Architektur 252

12-Faktoren

- Abhängigkeiten 91
 - Administrative Prozesse (update, backup, restore) 96
 - Build, Release, Run 93
 - Codebase 91, 93
 - Environment 95
 - Horizontale Skalierung 94
 - Konfigurationen 91
 - Logging 95
 - Port Binding 92
 - Skalierung über Prozesse 94
 - Umgebung 95
 - Unterstützende Services 92
- 12-Faktoren-Methodik 113, 143, 204

A

Ablaufverfolgung 199

Abstract Syntax Tree (AST) 315

A/B-Tests 35

A/B-Testszenarien 220, 221

Abwärtskompatibilität 175

ACID 183

Active Record 247

Active Record-Pattern 247, 252

Admission Controller 303

Affinität 123

Affinity 122

Aggregat 248, 249, 250

Aggregate Root 249

Aggregatgrenze 249

Aggregatwurzel 249

Agilität 18

Aktion 147

Aktionspakete 149

Alert-Manager 200, 205

ALLOW-Regel 225

Analysemodell 236

Anforderungen 236

Angriffssoberfläche 279

Angriffsvektor 261, 264, 265, 280, 309, 310

Anti-Corruption-Layer 243

Anwendungsschicht 253

Anwendungsvirtualisierung 66, 67

API 175

API-Gateway 177, 193, 194

API-Versioning 167, 175

APM 201

AppArmor 295

Append-Only-Log 250

Architektur 39

- DevOps-geeignet 33
- Serverless 143

Architekturelle Sicherheit 178

Architekturmuster 251

Asynchrone Architektur 173

auditd 274

Auditierbarkeit 251

Auditierung 329

Auditing 223, 273, 280, 308

Audit-Protokolle 251

Auftragsverarbeitung 338

Authentication Policy 224

Authentifizierung 223, 268, 271, 279, 281, 283, 289

- Peer 225
- Request 225

Authorisation 223

Authorisation Policy 224, 225

Automatisierte Instrumentierung 215

Automatisierung 18, 187

Autorisierung 268, 279

Autoskalierung 125

- ereignisbasiert 152
- horizontal, Pod 125

- AWS Lambda 151
- Azure Lambda 151
- B**
 - Backend as a Service (BaaS) 191
 - BASE 183
 - Batch-Job 101
 - Batch-System 207
 - Beobachtbare Architekturen 199
 - Beobachtbarkeit 40, 189
 - Berechtigtes Interesse 337
 - Best Practices 146, 351
 - Betriebssystem-Virtualisierung 66
 - Betriebszustand 107
 - Big Five 1
 - Binding Corporate Rules (BCR) 341, 343, 345
 - Binpack 102
 - Blackbox-Monitoring 205
 - Blackbox-Tracing 208
 - Blackbox-Überwachung 202
 - Block-Storage 65
 - Blue/Green-Deployment 34
 - Blue/Green-Release 189
 - Blueprint 107, 112
 - Borg 105
 - Bounded Context 187, 231, 239, 240, 252, 254
 - Branching-Strategien 56
 - Breaking-Change 166, 176, 188
 - BSI 325
 - BSI-C5 325
 - BSI-Grundschutz 325
 - BSI-Standard 100-3 325
 - BSI-Standard 100-4 325
 - BSI-Standard 200-1 325
 - Build Phase 49
 - Bulkhead 179
- C**
 - CaaS 82
 - Caching 94, 170, 186
 - clientseitig 186
 - Proxy-Caching 187
 - serverseitig 186
 - Canary 219, 221
 - Canary-Release 34, 189
 - CAP-Theorem 183
 - Chaos Engineering 32
 - Checkpoint 255
 - Chef 70
 - Choreography-over-Orchestration 188
 - CI/CD 49
 - CIDR 288
 - Circuit-Breaker 178, 189, 222
 - CISA 308
 - CISPE (Cloud Infrastructure Services Providers in Europe) 328
 - CISPE Code of Conduct 329
 - C-Level-Funktion 3
 - Clientseitiges Tracing 214
 - Client-Server 168, 170
 - CLOUD Act 343, 344, 345, 346
 - Cloud Compliance 322, 323
 - Cloud Computing 11
 - NIST-Definition 11
 - Cloud-native 2, 16, 24, 37, 351
 - Definition 40
 - Cloud-native Computing Foundation (CNCF) 39, 109
 - Cloud-Ökonomie 19, 142
 - Cluster 100, 107, 122
 - Cluster-Awareness 100
 - Cluster-Scheduler 110
 - CNCF 109
 - CNI 110
 - CO₂-Footprint 18
 - Code Repository 49
 - Command 254
 - Command Execution-Modell 254
 - Command Query Responsibility Segregation (CQRS) 185, 253
 - Common Vulnerabilities and Exposures (CVE) 312
 - Community Cloud 13, 17
 - Compliance 16
 - Config Map 113
 - Constraint 122
 - Container 24, 40, 41, 66, 79, 83, 109, 110, 188
 - Laufzeitumgebung 84
 - Runtime 84
 - Container as a Service 82
 - Container Breakout 266
 - Container-Image 88
 - Container Network Interface (CNI) 110
 - Container Runtime Environment 88, 112, 280, 299
 - Container Runtime Isolation 299
 - Container Storage Interface (CSI) 110
 - Content-Delivery-Netzwerks (CDN) 187
 - Context Mapping 231, 241
 - Continuous Deployment 49

- Continuous Integration 49
 Controller 108
 Control Plane 217
 Conway's Law 165, 229, 241
 Copy-on-Write 86
 Core Subdomain 233, 248, 250, 251, 253, 254
 CQRS 185, 253
 Creative Commons-Lizenz (CC0) 7
 Cron-Job 116
 CRUD 172, 247, 253
 CSA Cloud Control Matrix (CCM) 327
 CSA STAR 327
 CSI 110
 Current State 108, 125
 Customer-Supplier 241, 243
 Cyber Attack Lifecycle 264, 273, 280
- D**
- Daemon-Set 114, 117
 DAG Pipeline 52
 Dapper 208
 Data Breach 280
 Data Plane 217
 Datenbankbasierte Integration 167
 Datenbasierte Integration 167
 Datenkopplung 167, 188
 Datenlokalisierung 328
 Datenminimierung 336
 Datenschutz 16, 321
 Datenschutz-Folgenabschätzung 340
 Datentransfer in Drittländer 341
 Defense in Depth 223
 Dekomposition 161, 229
 Denial of Service 266, 285, 291, 292, 300
 DENY-Regel 225
 Dependency Injection 253
 Deployment 113, 114, 126
 Deployment-Pipeline 17, 31, 34, 49, 89, 266, 269, 298, 312, 315
 - Job 50
 - Phase 49
 - Trigger 50
 Deployment Unit 40, 41, 66, 79, 165
 Deploy Phase 49
 Desired State 108, 125
 Development 56
 Development-Branch 58
 DevOps 18, 27, 90, 165, 194, 200
 - Flaschenhälse 31
 - Kultur 31
 - Prinzipien des Feedbacks 32, 40
 - Prinzipien des Flow 29, 41
 - Work in Progress 30
 - Zyklus 29, 34
 Docker 83
 Dockerfile 88
 Domain-driven 187
 Domain-driven Design 229
 Domain-Event 232, 249, 250
 Domain Model-Pattern 248
 Domänenmodell 229, 232, 248, 250
 Domänenwissen 236
 Dominant Resource Fairness 103
 Double-Spending-Problem 144, 191, 193
 Downstream-Service 162
 DSGVO 16, 261, 321, 328, 329, 332, 335, 338, 343, 344
 Dumb-Middleware-with-Smart-Endpoints 188
- E**
- Ebenen-Architektur 252
 Effektives Design 230
 Einwilligung 337
 ElasticSearch 201
 Elastisches System 174
 Elastizität 40, 143, 173
 Emulation 66
 Endbenutzer-Choreografie 193
 End-to-End-Tracing 212
 Enterprise Architecture Management (EAM) 239
 Entkopplung 65
 Environment 54, 56
 Ereignisbasiert 188
 Ereignisbasierte Integration 167, 173
 Ereignisbasierte Systeme 173
 Ereignisgesteuert 173
 Ereignisquelle 145
 ETL-Pattern 246
 EU Cloud Code of Conduct 329
 EU-US Privacy Shield 343, 345
 Event-driven 145
 Event-Emitting-Service 173
 Event-Sourcing 254
 Event-Sourcing-Pattern 250
 Event Store 250
 Eventual Consistency 184, 249
 Everything as Code, Deployment-Pipeline 50
 Evolutionäres Design 164, 229
 Execution-Monitor 104

Executor 105
 Exploit 264
 Exporter 204
 Extract-Transform-Load (ETL) 246
 Extraktion von Span-Kontexten 213

F

FaaS 126
 - Best Practices 146
 FaaS-Framework 149
 FaaS-Plattform 142, 190
 FaaS-Programmiermodell 145, 147, 190
 Fachlichkeit 229, 230, 248
 Fail early 222
 Fairness 102
 Fallacies of Distributed Computing 255
 Feature-Branch 58
 Feature Release 176
 Feature-Schalter 34
 Federal Risk and Authorization Management Program (FedRAMP) 331
 Feed-basierte Trigger 148
 Fehlertoleranz 102
 File-Storage 65
 Firewall 268, 277, 282
 Flooding 292
 Fluentd 201
 Foothold 265
 Function 200
 Function as a Service (FaaS) 22, 141
 Funktion 145, 147

G

GAE 80
 GAIA-X 1
 GDPR 321, 335, 343
 Gegenseitige Authentifizierung 224
 Gegenseitige TLS-Authentifizierung (mTLS) 224
 Generic Subdomain 234, 244
 Generische Subdomäne 234
 Gerichtete Pipeline 52
 Geschäftskonzept 187
 Geschäftslogik 246, 247, 250, 252, 253
 Gesetz von Conway 165
 Git-Flow 57
 GitHub-Flow 58
 GitLab CI/CD 50
 Google App Engine 80
 Google Cloud Functions 151
 Grafana 201

gRPC (gRPC Remote Procedure Call) 129, 168, 210
 Grundsätze der Verarbeitung 336

H

Hadoop 104
 HashiCorp Configuration Language 75
 HATEOAS 171
 HCL 75
 Health Checking 129
 Health Insurance Portability and Accountability Act (HIPAA) 331
 Heroku 81
 Hexagonale Architektur 253
 Hierarchische Pipeline 53
 High-Level Container Runtime 87
 High-Level-Design 245
 Horizontale Pod-Autoskalierung 125
 Horizontale Skalierung 180
 Horizontal Pod Autoscaler 125
 Horizontal Pod Autoscaling (HPA) 152
 HPA 125
 HTTP-/REST-basierte Integration 167
 HTTPS 282
 Hybrid Cloud 14, 17
 Hypermedia as the Engine of Application State (HATEOAS) 171
 Hyperthread 122
 Hypervisor 65

I

IaC 69, 270, 271, 279
 IDEAL-Modell 39
 Idempotente Operation 180
 Idempotenz 180
 Image Registry 266
 Immutable 248
 Immutable Infrastructure 68
 Implementierungsdetail 166, 188
 Infrastructure as a Service (IaaS) 14, 15
 Infrastructure as Code 63
 Infrastruktur
 - als Code 69
 - elastisch 15
 Infrastrukturkomponente 253
 Infrastrukturschicht 253
 Ingress 113, 128, 280, 284, 285, 304
 In-Process-Komponenten 161
 Instrumentierung 202
 Instrumentierungsbibliothek 209

Instrumenting Library 200
 Integrations-Branch 58
 Integrität 337
 Interprozesskommunikation 167
 Intrusion Detection 273, 274, 280, 282, 307
 ISO 9001 324
 ISO/IEC 27001 325, 326
 ISO/IEC 27017 327
 ISO/IEC 27018 327
 Isolation 65, 67, 280
 Isolationsmechanismus 88
 Istio 217, 221, 224, 226
 IT-Sicherheit 261

J

Jaeger 201
 Job 114, 116, 200, 206, 207

K

Kanban 29
 KEDA 152
 - ScaledJob 153
 - ScaledObject 153
 Kerndomäne 233
 Kiali 226
 Kibana 201
 Knotenaffinität 123
 Kohäsion 229
 Kommunikationsmuster 245
 Konfigurations-API-Server 224
 Konfigurationsmanagement 70
 Konformist-Pattern 243
 Kontrollgruppe 220
 Kritische Infrastruktur 16
 Kritischer Pfad 209
 Kubeless 151
 Kubernetes 34, 105, 109, 200, 218
 - Affinität 123
 - API-Server 111, 112
 - Architektur 111
 - Cloud-Manager 111
 - Cluster Role 133
 - Controller-Manager 111
 - Daemon-Set 117
 - Deployment 115
 - Horizontal Pod Autoscaler (HPA) 125
 - Ingress 128, 168, 195
 - Job 116
 - Kubelet 112
 - Kube-Proxy 112

- Limit 121, 134
- Master Node 111
- Namespace 133
- Network-Plug-in 111
- Network Policy 135, 225
- Persistent Volume Claim (PVC) 132
- Persistent Volume (PV) 132
- Quota 135
- RBAC 133
- Request 121
- Resource Quota 134
- Role 133
- Role Binding 133
- Scheduler 111
- Secret 133
- Selektor 122
- Service 128, 181
- Service-Account 133
- Stateful-Set 119
- Storage-Plug-in 111
- Worker Node 112
- Workload 114

Kubernetes Hardening Guide 266, 279, 307
 Kubernetes-Ressourcen 112

L

Lastausgleich 181
 Lateral Movement 264, 279, 289
 Laufzeitumgebung 67
 Layered Architecture 252
 Ledger 250
 Let's Encrypt 283
 LimitRange 292
 Limits 121
 Liveness Probe 129
 Load Balancer 127, 282
 Load Balancing 181, 184
 Local Procedure Call (LPC) 168
 Log-Aggregation 202, 276, 307
 Logge auf stdout 204
 Logging 40, 189, 199, 202, 266
 Logikebene 252
 Log-Level 203

- Debug 203
- Error 203
- Fatal 203
- Info 203
- Trace 203
- Warning 203

 Lokalität 102

- Lose Kopplung 164, 229
 Low-Level Container Runtime 87
- M**
- Machtgefälle 241
 - Machtverhältnis 245
 - Manifest 110, 112
 - Man-in-the-Middle-Angriff 223, 224
 - Marathon 114
 - Materialien 7, 43, 78, 97, 139, 257
 - Materialien (Slides, Handouts)
 - 12-Faktoren-Methodik 97
 - Architektur-Pattern für Core Subdomains (DDD) 257
 - Architektur-Pattern für Supporting Subdomains (DDD) 257
 - Beobachtbarkeit 228
 - Betriebssystemvirtualisierung 97
 - Cloud Computing Historie 43
 - Cloud-native Systeme 43
 - Cloud-Ökonomie 43
 - Container-Orchestrierung 139
 - Context Mapping (DDD) 257
 - Deployment-Pipelines 61
 - Deployment Units (Container) 97
 - DevOps 43, 61
 - DevOps-geeignete Architekturen 61
 - Docker 97
 - Domain-driven Design 257
 - Effektives Software-Design 257
 - FaaS-Plattformen 156
 - FaaS-Programmiermodell 156
 - Function as a Service (FaaS) 156, 197
 - Immutable Architectures 78
 - Infrastructure as a Service 78
 - Infrastructure as Code 78
 - Kubernetes 139
 - Kubernetes Blueprints (Manifests) 139
 - Logging 228
 - Materialien (Slides, Handouts) 97
 - Metriken und Monitoring 228
 - Microservices 197
 - Pattern für Geschäftslogiken (DDD) 257
 - Platform as a Service (PaaS) 139
 - Prinzipien des Feedbacks 61
 - Prinzipien des Flow 61
 - Resilienz 228
 - Serverless Computing 156, 197
 - Service-Meshs 228
 - Scheduling 139
 - Sicherheit 228
 - Strategisches Design (DDD) 257
 - Subdomains (DDD) 257
 - Taktisches Design (DDD) 257
 - Telemetriedaten 228
 - Terraform 78
 - Tracing 228
 - Traffic-Management 228
 - Ubiquitous Language (DDD) 257
 - Vagrant 78
 - Visualisierung von Verkehrstopologien 228
 - Was ist Cloud Computing? 43
 - Mehrdeutiger Begriff 238
 - Memory-Ballooning 65
 - Mentales Modell 239
 - Mesos 34, 103, 106, 110, 114
 - Messaging 181
 - Metriken 40, 189, 199, 204
 - Messung (Gauge) 206
 - Verteilung (Histogramm) 206
 - Zähler (Counter) 206
 - Metrikinstrumentierung 207
 - Microservice 18, 31, 39, 162, 231
 - Microservice-Architektur 144, 199, 229
 - Microservice-basierte Anwendung 163
 - Millicore 122
 - Monitoring 199, 204, 266
 - Monolithische Anwendung 163
 - Monopolisierung 285, 291
 - Monorepository 54
 - mTLS 224, 225
 - Multi-Cloud 72
 - Multiplizität 65
 - Multi-Tenancy 133, 135, 225
 - Mutable 248
 - Mutual Authentication 224
- N**
- Nachrichtenorientiertes System 174
 - Namespace 280, 287
 - Network Policy 287, 289, 303
 - Netzwerkssegmentation 279, 286
 - Netzwerkpartition 189
 - Nomad 34, 110
 - NoSQL 185
 - NoSQL-Datenbanken 183
 - NSA 307, 308
- O**
- OAuth 283, 284

- Objektmodell 254
 - lesend 253
 - schreibend 253
- Objektrelationales Mapping (ORM) 247
- Observability 40, 189, 202, 265, 276
- Observable 174
- OCI 84, 109
- Omega 106
- One-Service-per-Container 189
- Online-System 207
- OPA-Policy 304
- OpenAPI 244
- Open-Container-Initiative 84
- Open-Host-Service 244
- Open Policy Agent 303, 305
- OpenTracing-API 209, 212
- OpenWhisk 147, 151
- Orchestrierung 99, 107
- Orchestrierungsplattform 34, 188
- Orchestrierungsregelkreis 109, 126
- Ortsunabhängigkeit 174
- Out-of-Process-Komponenten 161
- Output Stream 96
- Overlay Network 110
- Over-Provisioning 24

- P**
- PaaS 79, 82
- Para-Virtualisierung 65
- Partnerschaftliche Kooperation 241
- Partnerschaftsmodell 241
- Partnership 241
- Patching 268, 270, 280
- Pattern 351
- Pay-as-you-go 2, 18, 19
- Payment Card Industry Data Security Standard (PCI DSS) 332
- Peak-to-Average 20
- Peer-to-Peer Computing 191
- Persistent Volume 132
- Persistent Volume Claim 113, 132, 301
- Persistenzebene 252
- Personenbezogene Daten 16, 335
- Phasen- 51
- Phishing 266
- Platform as a Service (PaaS) 14, 15, 79, 82
- Plattform
 - Container 82
 - elastisch 15, 41
 - Function as a Service (FaaS) 143
 - PaaS 80
- Pod 110, 152
- Pod-Affinität 124
- Pod Hardening 289, 297
- Pod Security Policy 296
- Pod Security Standard 297
- Policy 280, 303
- Policy Enforcement Point (PEP) 224
- Polyglotte Persistenz 254
- Polyglott Programming 67
- Ports & Adapter-Pattern 253
- Präsentationsebene 252
- Prinzip des geringsten Privilegs 291
- Private Cloud 13, 17
- Privilege Escalation 289, 297
- Probe 129
- Production 56
- Produktivsystem 33
- Projektion
 - asynchron 255
 - synchron 254
- Projektions-Engine 255
- Prometheus 201
- Protocol Buffers 168
- Protokollierung 40, 202
- Provisionierung 68
 - deklarativ 70
 - imperativ 70
 - Pull-basiert 70
- Proxy 216, 224, 282
- Prozessisolierung
 - Control Group (cgroup) 86
 - Namensräume für Dateisysteme 86
 - Namespace 84
 - Priorisierung 86
 - Process Capabilities 85
 - Quota 86
- PSP 296
- PSS 297
- Public Cloud 13, 16
- Publish/Subscribe 182, 250
- Puppet 70
- Push-Gateway 206, 207
- PVC 113
- Python 6

- Q**
- Qualitätsmanagementsystem (QMS) 324
- Query 254
- Querying-System 205

Queueing 181, 250
Quota 292, 301

R

RAFT 119
Rate Limiting 281, 284
RBAC 133, 134, 135, 290
ReactiveX-Programmiermodell 174
Readiness Probe 130
Reaktive Erweiterung (Rx) 174
Reaktives System 173
Rechenschaftspflicht 337
Rechtmäßigkeit 336
Rechtsordnung 321
Regel 145, 148
Regelkreis 108, 125
Regelkreis-basierte Orchestrierung 109
Region 63
Rego 305
Regulatorische Anforderungen 321
Release 33
Releaserisiken 34
Remote Procedure Call (RPC) 167, 177
Replay-Angriff 224
Replaying Time Machine 251
Replicas 118
Replica-Set 114
Replication Controller 113
Representational State Transfer (REST) 170
Requests 121
Resilient Software Design 32
Resilienz 32, 173, 221
Resilienz-Pattern 221
Responsivität 173
Ressourceneffizienz 25
Ressourcengröße 22
Ressourcenkontingent 135
REST 39, 129, 170, 180, 188, 192, 196, 210
REST-API 177
Restart Policy 117
Reverse-Proxy 187, 194
Richtigkeit 336
Richtlinie 280, 303
Role-based Access Model (RBAC) 133
Rolling-Updates 34
Rootkit 274
RPC

- Bidirectional-Streaming 169
- Client-Streaming 169
- Server-Streaming 169

- Unary 169
Runtime 67

S

Safe Harbor Abkommen 343
Sandbox 81
Scale-to-Zero 126, 141, 154, 162
Scaling for Reads 184
Scaling for Writes 184
Scaling out 180
Scaling up 180
Scheduler 100, 122

- 2-Level 105
- monolithisch 105
- Shared-State 106

Scheduling 99

- Algorithmus 102
- Architekturen 104
- Constraints 121
- einfache Algorithmen 102
- kapazitätsbasierte Algorithmen 103
- multidimensionale Algorithmen 103

SCHREMS I 343, 344
SCHREMS II 343, 345, 346
Schutz personenbezogener Daten 328, 329
Schwachstellen-Scanning 280, 311, 313, 316, 317
Seccomp 295
Secret 113, 291
Secure Shell 271
Security by Default 223
Security Context 293, 294
Security Group 277
Selektor 122, 225
Self-Healing 39, 109
Self-Service 188, 200
Self-Service-Cluster 71
SELinux 294
Semantic Versioning 176
Separate Way 241, 244
Separation of Duties 291
Serverless-Architektur 144, 190, 195
Serverless Computing 142, 191
Serverless-Effekt 192
Serverseitiges Tracing 213
Service 101, 113, 126, 188
Service-Account 289, 290
Service-API 129
Service Computing 12
Service-Discovery 126

- Service-Interaktion 207
 Servicekohäsion 188
 Service-Merkmale 13
 Service-Mesh 178, 189, 216
 Service-Mesh Interface (SMI) 218
 Service-Modell 12
 - IaaS 15
 - PaaS 15
 - SaaS 16
 Service-of-Services 40, 161
 Service Ownership 18, 164, 165
 Sharding 184
 Shared-Database-Pattern 167
 Shared-Kernel 242
 Shared Nothing 94
 Sicherheitsgruppe 268, 277
 Sicherheitsrichtlinie 298
 Sidecar 216, 224
 Single-Responsibility-Prinzip 164, 229
 Single Source of Truth 185, 250, 254
 Skalierbarkeit 18, 40, 143
 Skalierung 125
 - horizontal 180
 - vertikal 180
 Skalierungserfordernis 196
 Social Engineering 263, 266
 SOC (Service Organization Control) 330
 Software as a Service (SaaS) 14, 16
 Software Composition Analysis (SCA) 311
 Software Supply Chain 280, 310
 Software-Virtualisierung 66
 Span 208, 209
 Span-Kontext 208, 214
 Speicherdauer 336
 Spoofing-Angriff 224
 Spread 102
 SSH 271
 SSL/TLS-Terminierung 281
 Stabilitätsmuster 178
 Staging 56
 Standardisierung des Betriebs 18
 Standardvertragsklauseln (SCC) 341, 342, 343, 345
 Start-up Probe 131
 Stateful-Service 183, 197
 Stateful-Set 114, 118
 Stateless 87, 170
 Static Application Security Testing (SAST) 314
 Storage Class 113, 132
 Strategisches Design 231, 232
 Strict Consistency 249
 Stub 168
 Subdomain 232
 Subdomäne 232, 240
 Supporting Subdomain 234, 246, 247, 251, 252
 Swarm 34, 102, 105, 110, 114
 Synonymer Begriff 238
 Systementwurf 236
 System Hardening 267, 268
- T**
- Taktisches Design 246
 Telemetriedaten 32, 35, 40, 199
 - Konsolidierung 200
 Telemetriedaten Konsolidierung 268
 Terraform 74
 - Ausführungsplan 74
 - Data Source 75
 - Provider 75
 - Provisioner 76
 - Ressource 76
 - Ressourcengraph 74
 - Ressourcen-Scheduler 75
 Testing 56
 Test Phase 49
 Threat Detection 309, 310
 Threat Model 280
 Timeout 189, 221
 Time-to-Market 80, 195
 TLS-Endpunkt-Termination 223
 Token 290
 Topologieschlüssel 124
 Trace 207, 208
 Tracing 40, 189, 199, 207
 Tracing Backend 212
 Tracing-Instrumentierung 212
 Traffic Definition 218
 Traffic-Management 217, 218
 Traffic Policy 217
 Traffic Spec 218
 Traffic-Split 219
 Traffic Telemetry 217
 Transaktion 210, 247
 Transparenz 336
 Trigger 145, 148
 Trigger - Regel - Aktion 148
 Trunk 59
 Trunk-basierte Entwicklung 59
 Typ-1-Virtualisierung 65
 Typ-2-Virtualisierung 66, 73

U

- Ubiquitous Language 231, 236, 237, 239, 244
- Übungen (Labs)
 - Autoskalierung 139
 - Beobachtbarkeit 228
 - Container-Image Builds 97
 - Container-Image Builds durch Deployment-Pipelines 97
 - Container-Image Shrinking 97
 - Containerisierung 97
 - Deployment-Pipeline 61, 139
 - Docker 97
 - FaaS-Programmiermodell 197
 - GitLab CI/CD 61
 - Google Cloud Functions 156
 - Google Compute Engine 78
 - gRPC 197
 - IaC-basierte Provisionierung 78
 - Kubeless 156
 - Kubernetes 139
 - Logging 228
 - Log-Konsolidierung 228
 - Observability 228
 - OpenWhisk 156
 - Orchestrierung 139
 - Publish/Subscribe 197
 - Queuing 197
 - Representational State Transfer (REST) 197
 - Self-Healing 139
 - Service-Meshs und Traffic-Management 228
 - Service-Meshs und Verkehrstopologien 228
 - Software-defined Infrastructure 78
 - Swarm 139
 - Terraform 78
 - Tracing 228
 - Vagrant 78
 - Workload (interaktives Jupyter Notebook) 43
- Umgebungsvariable 54
- Unabhängige Aktualisierbarkeit 163, 229
- Unabhängige Austauschbarkeit 229
- Unattended Upgrade 270
- Uniform Resource Identifier (URI) 170
- Union Filesystem 86
 - Copy-on-Write 86
 - Layer 86
 - Namensraum 86
- Unterstützende Subdomäne 234
- Upstream-Service 162
- US CLOUD Act 1

V

- Vagrant 72
 - Box 72
 - Provider 73
 - Provisioner 73
 - Vagrantfile 72
- Value Object 248
- vCPU 65, 122
- Vendor Lock-in 14, 82
- Verarbeitung nach Treu und Glauben 336
- Verfügbarkeit 18
- Verfügbarkeitszone 63
- Verhaltensanalyse 251
- Verkehrsfluss 136
- Verkehrstopologie 226
- Verschlüsselung 223, 301, 302
- Versionierungsschema 176
- Versionsverwaltungssysteme 29
- Vertikale Skalierung 180
- Vertraulichkeit 337
- Verzeichnis von Verarbeitungstätigkeiten 337
- Virtualisierung 24, 65
 - Betriebssystem 82
 - Hardware 65
- Virtual Private Network (VPN) 224
- Virtual Service 221
- Virtuelle Netzwerkschnittstelle 65
- VLAN 65
- Voll-Virtualisierung 66
- Volume Hardening 300
- Volume Provisioner 132
- Volunteer Computing 191
- Vorwärtskompatibilität 175

W

- Wegwerf-Komponente 95
- Wegwerf-Umgebung 71
- Wertschöpfungskette 30
- Whitebox-Instrumentierung 202
- Whitebox-Monitoring 205
- Whitebox-Tracing 208
- Whitebox-Überwachung 202
- Widerstandsfähigkeit 174
- Wiederholung (Retry) 222
- Workload 19, 100
 - einmalig/selten 20
 - Heterogenität 101
 - Isolation 133
 - kontinuierlich sinkend 20
 - kontinuierlich steigend 20

- periodisch 20
- statisch 20
- zufällig 20
Workload-Allokation 100, 107
Workload-Ausführung 101
Workload Policing 303
Workload-Queue 104
Workload-Scheduler 104
wsk 147

X

X.509 224
X-Trace 208

Y

YAML, Notation 6
YARN 104, 105
You build it, you run it 33, 229

Z

Zeitbasierte Trigger 148
Zeitreihe 204
Zeitreihen-Datenbank 200, 205
Zero-Day Exploit 296
Zero-Trust Networking 223
Zertifikat-Handling 223, 282
Zertifizierung 322
Zertifizierungsstelle (CA) 224
Zone 63
Zugriffskontrolle 218
Zustandsanalyse 251
Zustandslosigkeit 143, 170
Zuteilungsdauer 22
Zweckbindung 336, 338
Zwei-Wege-Authentifizierung 224
Zwölf-Faktoren-Methodik 90
Zwölf-Faktoren-Modell 39

Mit Struktur geht's besser



Gernot Starke

Effektive Softwarearchitekturen

Ein praktischer Leitfaden

9., überarbeitete Auflage

461 Seiten. Inklusive E-Book

€ 49,99. ISBN 978-3-446-46376-9

Auch einzeln als E-Book erhältlich

Dieser Praxisleitfaden zeigt Ihnen, wie Sie Softwarearchitekturen effektiv und systematisch entwickeln können.

- Architekturmuster und -stile
- Technische Konzepte
- Microservices
- Blockchain
- Architekturanalyse und -bewertung
- Dokumentation von Architekturen
- Modernisierung bestehender Systeme
- Beispiele realer Architekturen
- iSAQB Curriculum

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

CLOUD-NATIVE COMPUTING //

- Grundlagen des Cloud Computings (Service-Modelle und Cloud-Ökonomie)
- Das Everything-as-Code-Paradigma (DevOps, Deployment Pipelines, IaC)
- Den Systembetrieb mit Container-Orchestrierung automatisieren
- Microservice- und Serverless-Architekturen verstehen und Cloud-native-Architekturen mit Domain Driven Design entwerfen
- **Neu in der 2. Auflage:** Sicherheit Cloud-nativer Systeme, Härtung von Infrastrukturen und Containern entlang von üblichen Angriffsvektoren, Compliance und regulatorische Anforderungen

Märkte verändern sich immer schneller, Kundenwünsche stehen im Mittelpunkt – viele Unternehmen sehen sich Herausforderungen gegenüber, die nur digital beherrschbar sind. Um diese Anforderungen zu bewältigen, bietet sich der Einsatz von Cloud-native-Technologien an.

Das Buch beleuchtet den Cloud-native-Wandel aus unterschiedlichen Perspektiven: von der Unternehmenskultur, der Cloud-Ökonomie und der Einbeziehung der Kunden (Co-Creation) über das Projektmanagement (Agilität) und die Softwarearchitektur bis hin zu Qualitätssicherung (Continuous Delivery) und Betrieb (DevOps). Anhand von realen Praxisbeispielen wird gezeigt, was bei der Umsetzung in unterschiedlichen Branchen gut und was schlecht gelaufen ist und welche Best Practices sich daraus ableiten lassen. Dabei wird auch die Migration von Legacy-Code berücksichtigt.

IT-Architekten vermittelt dieses Buch zudem das grundlegende Wissen, um Cloud-native-Technologien und die DevOps-Kultur in ihrem Projekt oder im gesamten Unternehmen einzuführen.

WEBSITE ZUM BUCH // mit Labs, Slides und Zusatzmaterialien für Leser, Trainer und Dozenten (CC0-Lizenz): <https://cloud-native-computing.de>

HANSER

www.hanser-fachbuch.de/computer

Nane KRATZKE ist Professor für Informatik an der Technischen Hochschule Lübeck und befasst sich seit mehr als 10 Jahren in Forschung, Beratung und Lehre mit Cloud-native Technologien.



AUS DEM INHALT //

- Einleitung
- Teil I: Grundlagen
 - Cloud Computing
 - DevOps
 - Cloud-native
- Teil II: Everything as Code
 - Deployment Pipelines
 - Infrastructure as Code
 - Standardisierung von Deployment Units
 - Container-Plattformen
 - Function as a Service
- Teil III: Architekturen
 - Microservices
 - Serverless-Architekturen
 - Beobachtbare Architekturen
 - Domain Driven Design
 - Cloud-native Patterns
- Teil IV: Sicherheit
Cloud-nativer Systeme
 - Härtung Cloud-nativer Anwendungen (DevSecOps)
 - Regulatorische Anforderungen (DSGVO u. a.)

ISBN 978-3-446-47914-2



9 783446 479142