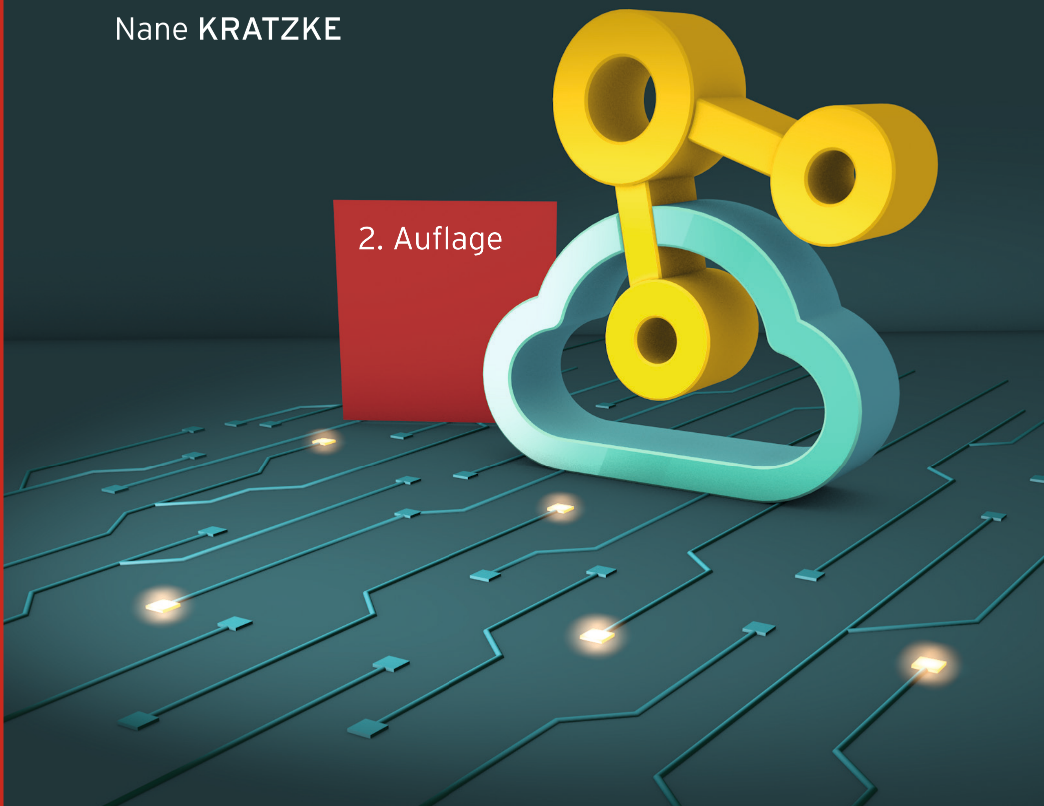
Nane KRATZKE

2. Auflage



CLOUD-NATIVE COMPUTING

Software Engineering von Diensten und Applikationen für die Cloud

HANSER

Kratzke

Cloud-native Computing

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

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.

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 Policys . . . 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 synonyme 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 Netzwerkisolation 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, Orchestrierungsplattformen, 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-nativen 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-nativer 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-nativen 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-nativen 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-nativer Systeme. Dabei werden zum einen die technische Sicherheit, aber auch regulatorische Aspekte betrachtet.

- **Kapitel 16** befasst sich eher 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 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-nativer 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 übersetzbar. 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 Softwareexperten 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

```python
# 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

```yaml
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

```yaml
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 Schulungsinhalte aus zahlreichen Quellen zusammentragen, 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 Typrepräsentanten 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/39L0BA3

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 *„allgegenwärtigen, 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.

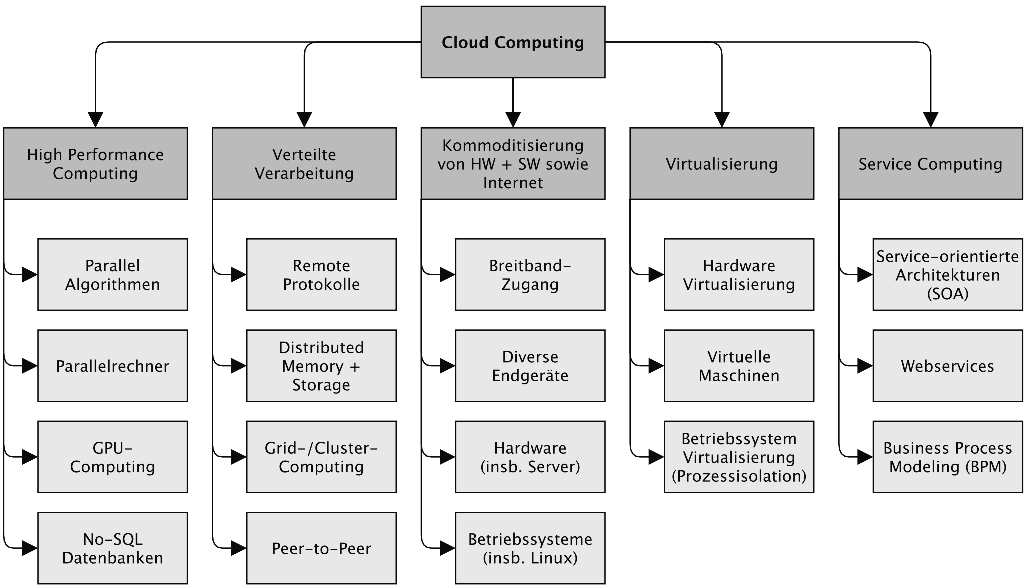


Bild 2.1 Einflussfaktoren auf das Cloud Computing

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 substanziell 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 substanziellen 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.

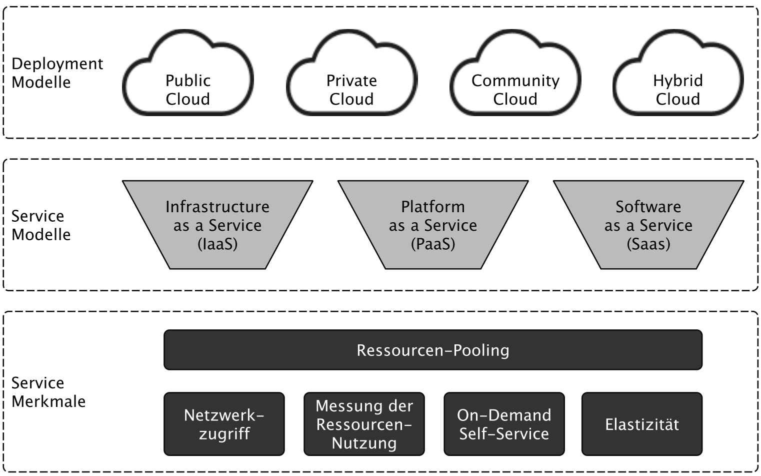


Bild 2.2 NIST-Modell des Cloud Computings

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 reallokiert. 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.

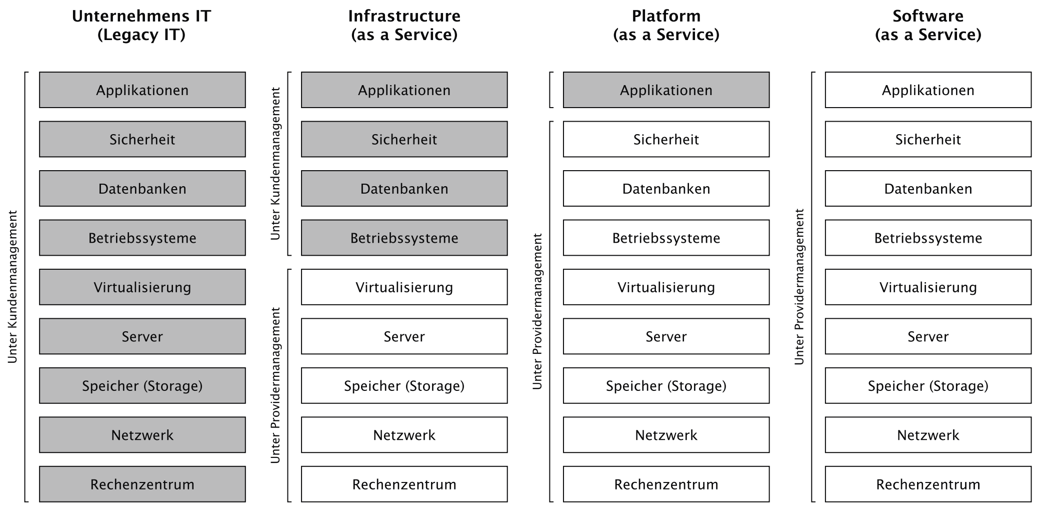


Bild 2.3 Auslagerung der Wertschöpfung bei IaaS, PaaS und SaaS

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 Bereitstellung 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 netzwerkbasierte 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-nativen 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 CO_2-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.

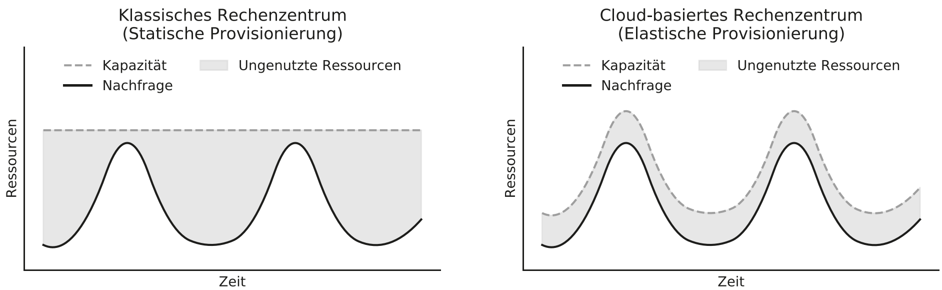


Bild 2.4 Statische und elastische Provisionierung von Ressourcen

2.2.1 Eignung von unterschiedlichen Arten von Workloads

Die Betrachtung von Workloads ist naturgegeben immer sehr anwendungsfallspezifisch, 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.

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

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 Spitze 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:

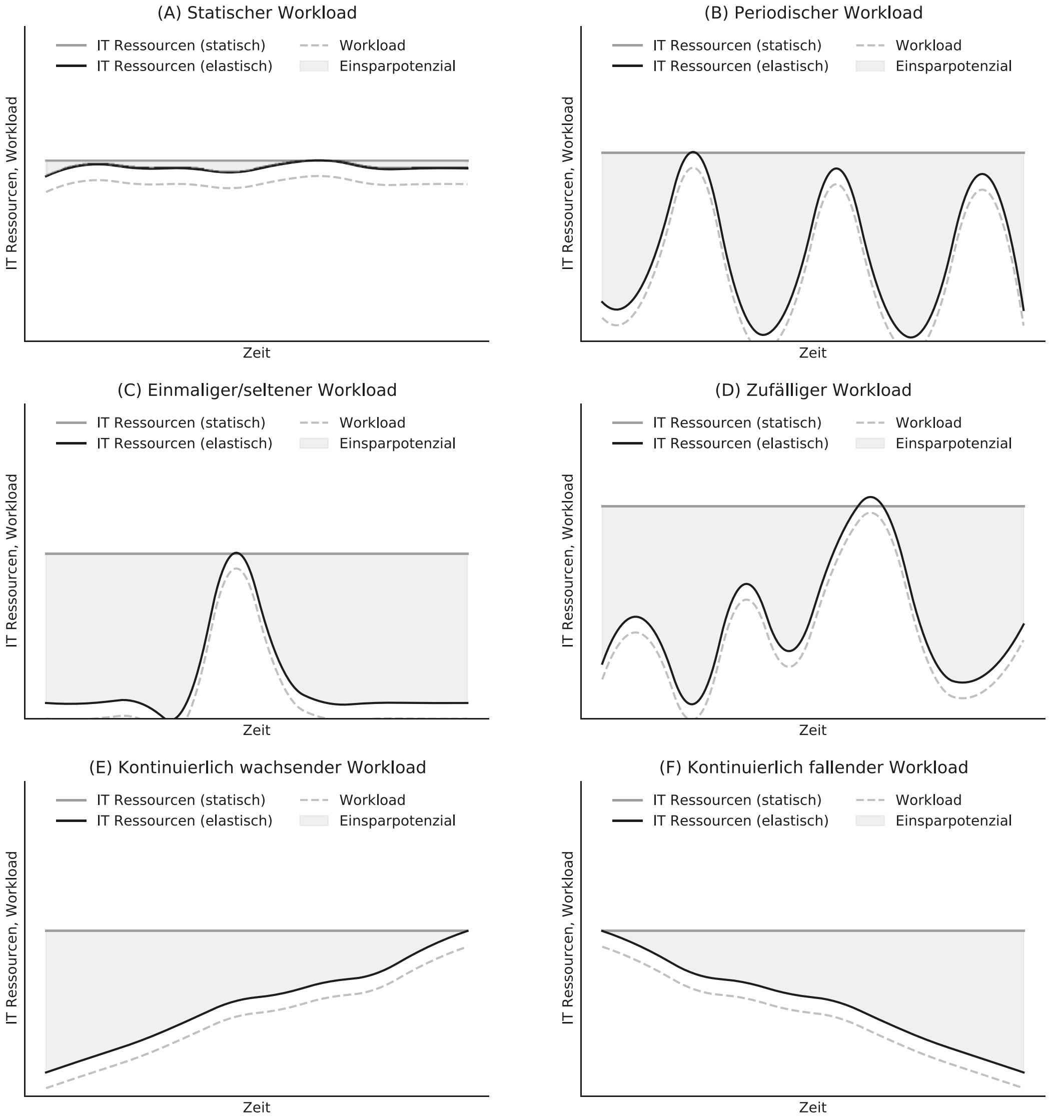


Bild 2.5 Zu berücksichtigende Workloads im Cloud Computing

- 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 halbieren 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 minutengenauen oder mittlerweile sogar einer sekundengenauen 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 feingranulareren Ressourcenallokation im Cloud-nativen Umfeld ihren Grund auch immer in der innewohnenden Cloud-Ökonomie haben – auch wenn dies häufig nicht (mehr) bewusst wahrgenommen wird.

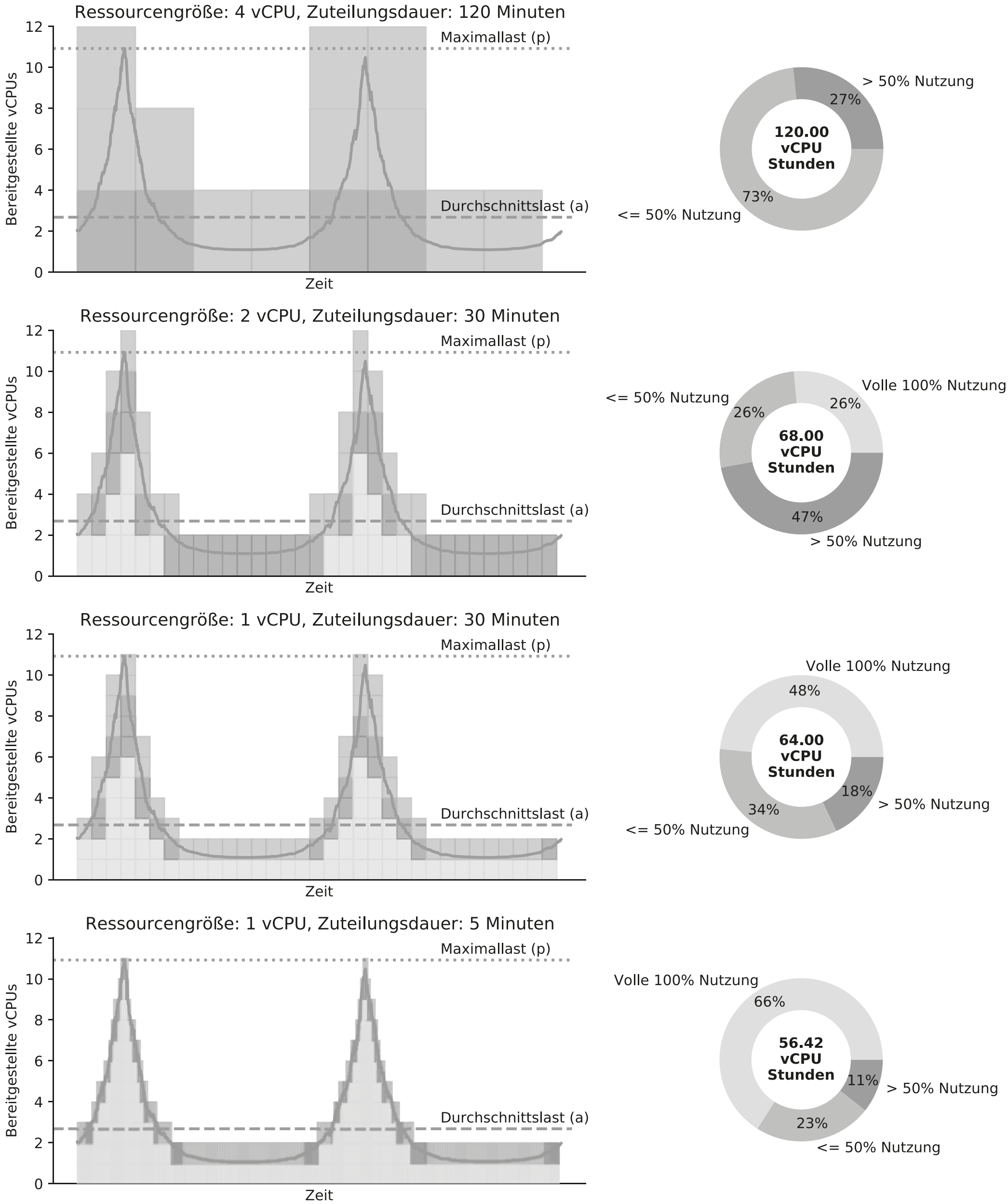


Bild 2.6 Effekt von Ressourcengröße und Zuteilungsdauer