BE SOFTWARE ARCHITECTURE WISE · KAPITEL 10

Cloud-native Architektur und Infrastrukturarchitektur

Wo und wie Software läuft, ist heute eine Architekturentscheidung. Dieses Kapitel behandelt Cloud-Service- und Bereitstellungsmodelle und das Modell der geteilten Verantwortung, Container und Kubernetes, die Twelve-Factor App, Infrastructure as Code und GitOps, CI/CD-Pipelines und Deployment-Strategien, Multi-Region-Design, Disaster Recovery und Edge Computing, Platform Engineering sowie FinOps und Anbieter-Lock-in. Es richtet sich an Entwicklungsteams und Architekten, die die Betriebsqualitäten aus Kapitel 9 kennen und die Plattformen, von denen ihre Systeme abhängen, auswählen, begründen und betreiben wollen, ohne sich einen Stack verkaufen zu lassen.

bewiseapps.com/de/software-architecture-wise/cloud-native/ · Version 2026-11-03. Nur Bildungsinformation, keine professionelle Beratung.

AUF EINEN BLICK

Das lernen Sie

- IaaS, CaaS, PaaS, FaaS und SaaS sowie die Bereitstellungsmodelle vergleichen und die Pflichten für eine Workload nach dem Modell der geteilten Verantwortung zuweisen.
- Erklären, wie Kubernetes einen deklarierten gewünschten Zustand über seine Control Plane, Nodes und Reconciliation Loops aufrechterhält, Requests, Limits und Probes konfigurieren und beurteilen, wann Kubernetes zu viel des Guten ist.
- Eine Anwendung anhand der zwölf Faktoren und der darüber hinausgehenden Prinzipien prüfen und die Korrekturen nach Risiko priorisieren.
- Infrastrukturcode mit Remote State, Locking, Modulen, Umgebungen, Policy-Prüfungen, Tests und GitOps-Abgleich entwerfen.
- Eine Deployment-Strategie wählen (Rolling, Blue-Green, Canary oder Shadow) und Deployment und Release mit Feature-Flags trennen.
- Eine Disaster-Recovery-Strategie anhand von RTO, RPO und Kosten auswählen und die Folgen erklären, wenn Daten über Regionen repliziert und Arbeit an den Edge verlagert wird.
- Erklären, wie eine Internal Developer Platform mit Golden Paths die kognitive Last senkt und wie man misst, ob sie erfolgreich ist.
- Stückkosten wie die Kosten pro Bestellung berechnen, die architektonischen Treiber der Cloud-Kosten erkennen und Lock-in als bepreiste Abwägung bewerten.

Bevor Sie beginnen

Kapitel 0, Abschnitt 0.7 (Server und die Cloud) und Abschnitt 0.8 (wie Software entsteht); Kapitel 9, besonders Skalierung, Zustandslosigkeit, SLOs und Observability. Kapitel 6 über verteilte Systeme hilft.

Dauer

Etwa 3½ Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

Cloud-native · Modell der geteilten Verantwortung · Availability Zone · Container-Image · Kubernetes · Reconciliation Loop · Twelve-Factor App · Infrastructure as Code · GitOps · Canary Release · RTO und RPO · Golden Path · FinOps · Lock-in

Warum Infrastruktur zur Architektur geworden ist

Über den größten Teil der Geschichte der Software kam die Infrastruktur nach der Architektur: Ein Betriebsteam erhielt eine fertige Anwendung und suchte Server, auf denen sie laufen konnte. Diese Trennung gibt es nicht mehr. In einem modernen System entscheidet die Wahl zwischen virtuellen Maschinen, Containern, Plattformen und Funktionen über einen Großteil der Kosten; die Zahl der Availability Zones und Regionen über die Verfügbarkeit; die Deployment-Pipeline darüber, wie schnell und sicher ein Team überhaupt etwas ändern kann; und die Infrastruktur selbst wird als Code geschrieben, in Pull Requests geprüft und

zusammen mit der Anwendung versioniert. Entscheidungen darüber, wo und wie Software läuft, gehören heute zu den bedeutsamsten, die ein Architekt trifft, genau im Sinne von Kapitel 1: Sie sind teuer rückgängig zu machen, betreffen alle Teams und legen die Qualitätsmerkmale fest, die Sie nach Kapitel 2 spezifizieren sollten.

Dieses Kapitel baut auf zwei früheren auf. Kapitel 0 hat Server, virtuelle Maschinen, Container, Cloud-Servicemodelle, Regionen und Availability Zones eingeführt (Server und die Cloud). Kapitel 9 hat erklärt, wie Systeme skalieren, kontrolliert degradieren und beobachtbar bleiben (ab Skalierungsstrategien). Hier treffen diese Ideen auf die Plattformen, auf denen die meisten Systeme heute laufen. Das durchgehende Beispiel ist die Essenslieferplattform, die im ganzen Kurs verwendet wird: Kunden- und Restaurant-Apps, ein Bestellservice, Speisekarten, Zahlungen, Kurierdisposition und ein heftiger Ansturm am Abend.

Wie die Abschnitte zusammenhängen

Die ersten vier Abschnitte behandeln die Grundlagen. Cloud-Service- und Bereitstellungsmodelle legt dar, wer bei IaaS, Containers as a Service, PaaS, FaaS und SaaS was verwaltet, erklärt das Modell der geteilten Verantwortung, die wirklichen Gründe für Private, Hybrid und Multi-Cloud, was die CNCF unter Cloud-native versteht und wann Mieten nicht billiger ist als Besitzen. Container und Orchestrierung erklärt Images und die OCI-Standards, Control Plane und Nodes von Kubernetes, seine Kernobjekte und den Reconciliation Loop in seinem Zentrum, mit echtem YAML, und sagt ehrlich, wann Kubernetes überdimensioniert ist. Die Twelve-Factor App geht alle zwölf Faktoren mit heutigen Einschränkungen durch, dazu die darüber hinausgehenden Prinzipien und die Prüfung eines bestehenden Service. Infrastructure as Code behandelt deklarative Werkzeuge (Terraform, OpenTofu, CloudFormation, Bicep, Pulumi), State, Drift, Module, Immutable Infrastructure, GitOps, Policy as Code und Tests.

Die zweiten vier Abschnitte behandeln Veränderung, Skalierung, Menschen und Geld. CI/CD-Pipelines und Deployment-Strategien verfolgt eine Änderung vom Commit bis in die Produktion, vergleicht Rolling-, Blue-Green-, Canary- und Shadow-Deployments und trennt Deployment und Release mit Feature-Flags. Multi-Region, Disaster Recovery und Edge Computing vergleicht Disaster-Recovery-Strategien nach RTO, RPO und Kosten, erklärt die Folgen für die Konsistenz, wenn Daten über Regionen repliziert werden, und behandelt zellbasierte Architektur und Edge Functions. Platform Engineering behandelt die Internal Developer Platform als Produkt mit Golden Paths und Guardrails und zeigt, wie man misst, ob sie hilft. FinOps, Kostenarchitektur und Lock-in macht Kosten zu einer Architekturqualität, gemessen pro Bestellung, und behandelt Anbieter-Lock-in als Abwägung, die zu bepreisen ist, statt als Sünde, die es zu vermeiden gilt.

Jeder Abschnitt lässt sich für sich lesen, aber die Reihenfolge ist bewusst gewählt: Über Deployment-Strategien kann man nicht nachdenken, ohne zu wissen, was Kubernetes während eines Rollouts tut, und über Platform Engineering nicht ohne Infrastrukturcode und Pipelines, aus denen die Plattform gebaut wird.

Eine Idee zieht sich durch jeden Abschnitt

Die wichtigste Idee dieses Kapitels ist **ein deklarierter gewünschter Zustand, den Automatisierung fortlaufend abgleicht**. Kubernetes hält die Zahl der laufenden Kopien gleich der Zahl, die Sie deklariert haben. Infrastrukturcode beschreibt Netzwerke und Datenbanken, und ein Werkzeug berechnet die Differenz zum Bestehenden und wendet sie an. GitOps hält den Cluster gleich dem Inhalt eines Git-Repositorys. Eine Deployment-Pipeline macht aus jeder Änderung dieselbe getestete, wiederholbare Abfolge von Schritten. Immutable Infrastructure ersetzt, statt zu reparieren. In jedem Fall ändert ein Mensch eine Beschreibung, die geprüft, versioniert und nachvollziehbar ist, und Maschinen erledigen den Rest. Das erlaubt einem kleinen Team, ein großes System sicher zu betreiben: Änderungen werden Routine, und Wiederherstellung wird zu einem erneuten Durchlauf statt zu einer Heldentat.

Die zweite Idee lautet: Jede Option auf der Speisekarte dieses Kapitels ist eine Abwägung, und die richtige Wahl hängt von der Arbeitslast und der Organisation ab. Kubernetes ist hervorragend für viele Services und viele Teams und Verschwendung für drei Services und zwei Entwickler. Multi-Region Active-Active erkauft Minuten an Wiederherstellungszeit zu einem Vielfachen der Kosten. Ein Plattformteam hilft nur, wenn die Entwicklungsteams wollen, was es baut. Die Cloud ist bei schwankender Last am billigsten und kann bei großer, gleichmäßiger Last von eigener Hardware unterboten werden. Lock-in ist ein Kostenfaktor, der gegen die Kosten der Portabilität abzuwägen ist. Das Kapitel gibt Ihnen deshalb Entscheidungshilfen, ausgearbeitete Beispiele mit Zahlen und die Argumente beider Seiten statt eines einzigen empfohlenen Stacks.

Wie Sie dieses Kapitel nutzen

Wer neu im Betrieb ist, sollte zuerst die Kästen für Einsteiger und den Abschnitt über Server und die Cloud in Kapitel 0 lesen und sich keine Sorgen machen, Werkzeugnamen auswendig zu lernen: Produkte ändern sich alle paar Jahre, während die Mechanismen (Abgleich, Unveränderlichkeit, State und Drift, Progressive Delivery, Wiederherstellungsziele, Stückkosten) bleiben. Erfahrene Fachleute beginnen vielleicht mit den ausgearbeiteten Beispielen und den Entscheidungshilfen und lesen dann die Abschnitte, deren Abwägungen sie nicht teilen. Anbieter wie AWS, Microsoft Azure, Google Cloud und Cloudflare werden durchweg als Beispiele genannt, nicht als Empfehlungen.

Kapitel 11 folgt mit Bewertung, Governance und Modernisierung: wie man eine Architektur einschließlich ihrer Infrastruktur beurteilt und wie man sie im Lauf der Zeit sicher verändert.

Cloud-Service- und Bereitstellungsmodelle

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.7 Server und die Cloud
- 2.5 Verfügbarkeit und Zuverlässigkeit

Begriffe von früher, die er verwendet: Virtuelle Maschine (VM), Container, Elastizität, Failover

DIE IDEE IN KÜRZE

Denken Sie daran, wie Menschen zu einer Wohnung kommen. Sie können Land kaufen und ein Haus bauen, eine leere Wohnung mieten und sie einrichten, eine möblierte Wohnung mieten oder im Hotel wohnen. Jeder Schritt überträgt mehr Arbeit an jemand anderen und lässt Ihnen weniger Wahl; das Hotel ist wunderbar für eine Woche und teuer für zehn Jahre. Beim Betrieb von Software ist es genauso: Sie können eigene Server besitzen, nackte Maschinen mieten, eine fertige Plattform mieten oder fertige Software mieten. Was Sie auch wählen, Ihre eigene Tür schließen Sie weiterhin selbst ab: Ihre Daten und wer darauf zugreifen kann, bleiben Ihre Aufgabe.

WAS SIE WISSEN MÜSSEN

- NIST SP 800-145 definiert drei Servicemodelle (IaaS, PaaS, SaaS) und vier Bereitstellungsmodelle (Public, Private, Community, Hybrid); Containers as a Service und Functions as a Service füllen die Lücken dazwischen.
- Jeder Schritt von IaaS in Richtung SaaS nimmt Betriebsarbeit und Kontrolle weg; die nützliche Frage ist, welche konkreten Aufgaben (Patchen, Skalieren, Backups, Failover, Upgrades) bei Ihnen bleiben.
- Im Modell der geteilten Verantwortung sichert der Anbieter die Cloud selbst und der Kunde das, was er in sie hineinstellt; Daten, Identitäten und Konfiguration liegen immer beim Kunden.
- Private, Hybrid und Multi-Cloud haben legitime Gründe (Regulierung, gleichmäßige Last, Migrationen, besondere Dienste, Exit-Pläne), aber jede Arbeitslast portabel über mehrere Clouds zu betreiben, ist teuer.
- Verteilen Sie Arbeitslasten auf mindestens zwei, besser drei Availability Zones; wissen Sie, ob jede Abhängigkeit zonal, regional oder global ist.

- Die CNCF definiert Cloud-native als Praktiken, um lose gekoppelte, sichere, resiliente, verwaltbare, nachhaltige und beobachtbare Systeme programmatisch und wiederholbar zu bauen; es geht um das Wie, nicht um das Wo.
- Die Cloud ist bei schwankender Last am billigsten; bei großer, gleichmäßiger Last kann eigene Hardware billiger sein, wie 37signals argumentierte, sobald Kapazitätsreserve, Personal und Rabatte für Nutzungszusagen ehrlich eingerechnet werden.

Von einer Definition zu vielen Modellen

Kapitel 0 hat Server und die Cloud eingeführt: die NIST-Definition (Special Publication 800-145, 2011) mit ihren fünf wesentlichen Merkmalen On-Demand-Selbstbedienung, breiter Netzwerkzugang, Ressourcenbündelung, schnelle Elastizität und gemessene Nutzung sowie die Idee, dass Servicemodelle mehr oder weniger vom Stack an einen Anbieter übergeben. Dieser Abschnitt vertieft diese Ideen so weit, wie ein Architekt es braucht, um zu entscheiden, wo ein System läuft, und diese Wahl zu verteidigen.

NIST definiert drei Servicemodelle (Infrastructure, Platform und Software as a Service) und vier Bereitstellungsmodelle: **Public** (für alle offen, im Besitz eines Anbieters), **Private** (für eine Organisation, egal wo sie steht und wer sie betreibt), **Community** (geteilt von Organisationen mit gemeinsamen Anliegen, etwa Behörden) und **Hybrid** (zwei oder mehr davon, so verbunden, dass Daten und Anwendungen zwischen ihnen wechseln können). Die Branche hat den Servicemodellen seitdem Functions und Containers as a Service hinzugefügt und dem Vokabular der Bereitstellung die Multi-Cloud.

Wo Software läuft, gehört zu dem, was Kapitel 1 die Zuordnungsstruktur einer Architektur genannt hat (Architektur definieren): die Abbildung von Software auf Hardware, Netzwerke und Organisationen. Bedeutsam im Sinne von Kapitel 1 ist sie, weil sie teuer rückgängig zu machen ist und weil sie einen Großteil der Kosten eines Systems festlegt, seine Verfügbarkeit, seine Sicherheitsgrenze und wie schnell seine Teams releasen können.

Servicemodelle: wer was verwaltet

Die Servicemodelle bilden ein Spektrum. Am einen Ende mieten Sie nackte Kapazität und erledigen alles darüber selbst; am anderen mieten Sie eine fertige Anwendung und konfigurieren sie nur. Jeder Schritt nach rechts nimmt Betriebsarbeit und Wahlmöglichkeiten weg. Die Tabelle ergänzt die zwei Modelle, die jünger sind als das NIST-Dokument: Containers as a Service und Functions as a Service.

Servicemodelle im Vergleich

Modell	Sie deployen und betreiben	Der Anbieter betreibt	Beispiele	Typischer Einsatz
IaaS	Virtuelle Maschinen: Betriebssystem, Patches, Laufzeitumgebung, Anwendung, Skalierungsregeln	Hardware, Netzwerk, Hypervisor, Rechenzentrum	Amazon EC2, Azure Virtual Machines, Google Compute Engine	Software, die ein bestimmtes Betriebssystem oder einen bestimmten Kernel braucht, unverändert umgezogene Altanwendungen, volle Kontrolle
CaaS (Containers as a Service)	Container-Images und ihre Konfiguration; bei Managed Kubernetes auch die Wahl und Upgrades der Worker Nodes	Container-Control-Plane, oft auch die Nodes; bei Serverless-Containern alle Server	Amazon EKS und ECS, Azure Kubernetes Service und Container Apps, Google Kubernetes Engine und Cloud Run	Viele als Container gebaute Services, Teams, die portable Pakete wollen
PaaS	Anwendungscode und seine Konfiguration	Betriebssystem, Laufzeitumgebung, Skalierung, Load Balancing, oft Logging	Heroku, Azure App Service, Google App Engine, AWS Elastic Beanstalk	Standard-Webanwendungen und APIs, die zu den Konventionen der Plattform passen
FaaS	Einzelne Funktionen und die Ereignisse, die sie auslösen	Alles andere, einschließlich Skalierung auf null; Abrechnung pro Aufruf und Laufzeit	AWS Lambda, Azure Functions, Google Cloud Run functions, Cloudflare Workers	Ereignisgesteuerte, stoßweise oder geringvolumige Arbeit in kurzen Schritten
SaaS	Konfiguration, Benutzer, Berechtigungen und Ihre Daten	Die gesamte Anwendung	Microsoft 365, Salesforce, Workday, Zendesk	Generische Fähigkeiten, die das Geschäft nicht unterscheiden

In der Praxis verschwimmen die Grenzen. Serverless-Container-Plattformen wie Google Cloud Run und Azure Container Apps führen ein Container-Image so aus, wie ein PaaS Code ausführt, und skalieren wie eine Funktion auf null. Managed Kubernetes reicht von einer gehosteten Control Plane, bei der Sie die Worker-Maschinen weiterhin selbst patchen und aktualisieren, bis zu einem vollständig verwalteten Modus, in dem der Anbieter auch die Nodes betreibt. Die

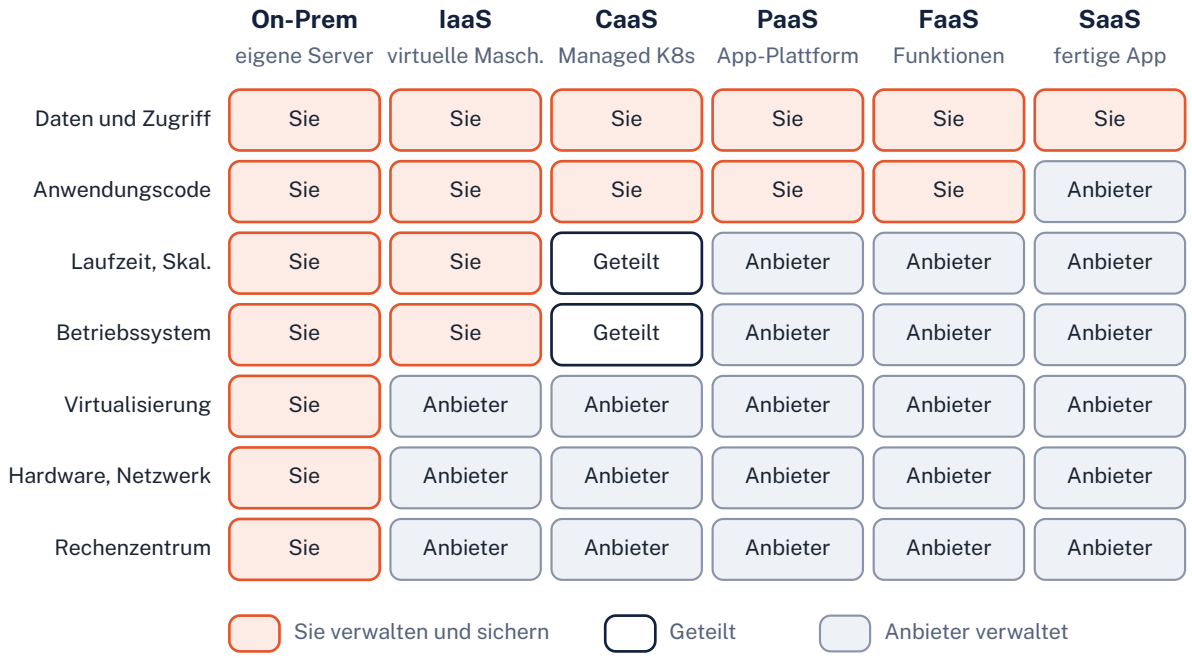
nützliche Frage ist nie, welches Etikett passt, sondern welche konkreten Aufgaben bei Ihnen bleiben: Patchen, Skalierungsentscheidungen, Kapazitätsplanung, Backups, Failover, Upgrades und Rufbereitschaft.

Eine zweite nützliche Unterscheidung ist die Einheit von Deployment und Abrechnung. IaaS rechnet Maschinen pro Sekunde oder Stunde ab, ob sie ausgelastet sind oder nicht; FaaS rechnet Aufrufe und Ausführungszeit ab und kostet im Leerlauf nichts; PaaS und Serverless-Container liegen dazwischen. Kapitel 6 erklärt, warum Abrechnung pro Anfrage unterhalb eines Schnittpunkts billiger ist als ständig laufende Kapazität und oberhalb teurer (Serverless-Architekturen). Die Form der Last einer Arbeitslast, nicht ihre Größe, entscheidet, auf welcher Seite des Schnittpunkts sie liegt.

Das Modell der geteilten Verantwortung

Jeder große Anbieter veröffentlicht ein Modell der geteilten Verantwortung. AWS fasst es so zusammen: Der Anbieter ist für die Sicherheit *der* Cloud verantwortlich, der Kunde für die Sicherheit *in* der Cloud. Der Anbieter sichert Gebäude, Hardware, Netzwerk, Hypervisor und die Dienste, die er betreibt; der Kunde sichert, was er darauf baut und konfiguriert. Wo die Grenze verläuft, hängt vom Servicemodell ab, wie das Diagramm zeigt.

Wer wofür verantwortlich ist, je Servicemodell



Nach rechts gehen die unteren Schichten an den Anbieter, aber Daten, Identitäten und Zugriffskonfiguration bleiben in jedem Modell beim Kunden.

Drei Dinge gehen nie über. Der Kunde besitzt immer seine Daten (was gespeichert wird, wo, wie lange, wer es lesen kann), seine Identitäten und Berechtigungen und die Konfiguration der Dienste, die er nutzt. Viele Cloud-Sicherheitsvorfälle gingen auf Konfigurationsfehler des Kunden zurück, nicht auf Fehler des Anbieters: für alle lesbarer Objektspeicher, Administratorschlüssel in öffentlichen Repositorys, zu weit gefasste Berechtigungen. 2019 etwa nutzte ein Angreifer bei Capital One eine falsch konfigurierte Web Application Firewall aus, um Cloud-Zugangsdaten zu erlangen und Daten zu mehr als 100 Millionen Menschen in den USA und Kanada zu kopieren; die Infrastruktur des Anbieters verhielt sich wie vorgesehen.

Für einen Architekten ist das Modell eine Checkliste von Arbeit, die jemandem zugewiesen werden muss. Verlegt ein Team die Bestelldatenbank der Essenslieferplattform von einer virtuellen Maschine in einen verwalteten Datenbankdienst, geht das Patchen der Datenbank-Engine an den Anbieter über, nicht aber die Wahl eines Wartungsfensters, das Testen von Hauptversions-Upgrades, die Vergabe von Zugriffen nach Least Privilege, das Aktivieren der Verschlüsselung und die Prüfung, ob sich Backups wiederherstellen lassen. Kapitel 8 behandelt die Sicherheitsprinzipien hinter diesen Pflichten ausführlich (Sicherheitsprinzipien für Architekten).

Bereitstellungsmodelle und die wirklichen Gründe dahinter

Organisationen wählen ein Bereitstellungsmodell selten aus nur einem Grund, und der genannte Grund ist nicht immer der wirkliche. Es hilft, die Motive, die einer Prüfung standhalten, von denen zu trennen, die es nicht tun.

Public, Private, Hybrid und Multi-Cloud

1. **Public Cloud**
   Gewählt wegen Elastizität, Tempo (eine neue Umgebung in Minuten statt eines Hardwarekaufs über Monate), eines großen Katalogs verwalteter Dienste und globaler Reichweite. Sie passt zu schwankender Last, zu neuen Produkten mit unbekannter Nachfrage und zu Teams, die ihre Kraft lieber ins Produkt als in Server stecken.

2. **Private Cloud**
   Gewählt wegen regulatorischer oder vertraglicher Kontrolle, vorhersehbarer gleichmäßiger Last, bestehender Investitionen in Rechenzentren oder sehr geringer Latenz zu Geräten vor Ort. Eine Private Cloud bedeutet Selbstbedienung und API-gesteuerte Bereitstellung auf dedizierter Infrastruktur (etwa mit OpenStack, VMware oder der On-Premises-Edition einer Public Cloud eines Anbieters), nicht bloß einen virtualisierten Serverraum, in dem Anfragen weiterhin über eine Ticket-Warteschlange laufen.

3 **Hybrid Cloud**

Meist das Ergebnis einer schrittweisen Migration, von Daten, die vor Ort bleiben müssen (ein Hauptbuch auf dem Mainframe, Fabrik- oder Krankenhaussysteme), oder von Edge-Standorten wie Filialen und Lagern. Das Ausweichen aus einem privaten Rechenzentrum in die Public Cloud zu Spitzenzeiten (Bursting) wird oft versprochen, ist aber selten praktikabel, weil die Daten, die die Arbeitslast braucht, zurückbleiben und ihr Kopieren Zeit und Geld kostet.

4 **Multi-Cloud**

Die Nutzung von zwei oder mehr Public-Cloud-Anbietern. Vertretbare Gründe sind etwa, von jedem Anbieter einen besonderen Dienst zu nutzen, Übernahmen, die auf einer anderen Cloud mitkommen, Verhandlungsmacht und Regulierung, die glaubwürdige Exit-Pläne verlangt. In der EU verlangt DORA (Verordnung über die digitale operationale Resilienz im Finanzsektor), die seit Januar 2025 gilt, von Finanzunternehmen Exit-Strategien für kritische IKT-Dienstleister. Jede Arbeitslast portabel über mehrere Clouds zu betreiben, ist ein anderes und weit teureres Ziel.

Der verbreitete Fehler ist, Multi-Cloud als abstrakte Absicherung einzuführen. Eine Arbeitslast auf zwei Anbietern identisch laufen zu lassen, heißt, nur die Dienste zu nutzen, die beide anbieten, zwei Sätze Infrastrukturcode zu bauen und zu testen, Fachwissen für beide vorzuhalten und oft Gebühren für den Datentransfer zwischen ihnen zu zahlen. Das ist die Falle des kleinsten gemeinsamen Nenners: Sie zahlen jeden Tag die Kosten der Portabilität für einen Ausstieg, den Sie vielleicht nie nutzen. Abschnitt 10.8 behandelt Lock-in als Abwägung, die zu bepreisen ist, nicht als Sünde, die um jeden Preis zu vermeiden ist (FinOps, Kostenarchitektur und Anbieter-Lock-in).

Regionen und Availability Zones als Entwurfsgrößen

Eine Region ist ein geografisches Gebiet mit einem eigenen, unabhängigen Satz von Rechenzentren; eine Availability Zone besteht aus einem oder mehreren Rechenzentren innerhalb einer Region mit eigener Stromversorgung, Kühlung und Vernetzung, so weit von den anderen entfernt, dass ein Brand, eine Überschwemmung oder ein lokaler Stromausfall nur eine trifft, und nah genug für Round Trips von höchstens wenigen Millisekunden. Kapitel 0 hat die Idee erklärt. Für den Entwurf zählen drei weitere Punkte.

Erstens hat jeder verwaltete Dienst einen Ausfallbereich: *Zonale* Ressourcen (eine einzelne virtuelle Maschine, ein Datenträger) fallen mit ihrer Zone aus; *regionale* Dienste (die meisten verwalteten Datenbanken im Multi-Zonen-Modus, Objektspeicher, Queues) repliziert der Anbieter über Zonen; einige wenige *globale* Dienste (DNS, Identität, manche Content-Delivery-

und Load-Balancing-Dienste) erstrecken sich über Regionen, was ihre seltenen Ausfälle weitreichend macht. Zu wissen, welchen Bereich jede Abhängigkeit hat, ist der erste Schritt jeder Verfügbarkeitsanalyse (siehe Verfügbarkeit und Zuverlässigkeit).

Zweitens ist die Verteilung über Zonen der günstige, wertvolle Standard. Laufen die zustandslosen Services der Essenslieferplattform in drei Zonen hinter einem Load Balancer, mit einer Datenbank, die einen synchronen Standby in einer zweiten Zone hält, übersteht die Plattform den Verlust eines Rechenzentrums mit automatischem Failover und geringen Latenzkosten. Drei Zonen statt zwei sind wichtig für alles, was Quoren nutzt, etwa etcd in Kubernetes oder eine konsensbasierte Datenbank, denn eine Mehrheit muss überleben. Verkehr zwischen Zonen ist allerdings nicht immer kostenlos: Die meisten großen Anbieter berechnen Datenübertragung zwischen Zonen, sodass gesprächige Services, die über Zonen verteilt sind, eine sichtbare Zeile auf der Rechnung hinzufügen.

Drittens sind Regionen nicht identisch. Neueren oder kleineren Regionen fehlen womöglich Dienste, manche haben weniger Zonen, die Preise unterscheiden sich je Region, und die Rechtsordnung folgt der Region. Abschnitt 10.6 behandelt mehrere Regionen, Disaster Recovery und Datenresidenz (Multi-Region, Disaster Recovery und Edge Computing).

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: was beim Ausfall einer Zone stillsteht

Die schwierigste Idee dieses Abschnitts ist der Ausfallbereich: Jeder Teil eines Systems lebt in einer Zone, über eine Region verteilt oder weltweit, und das entscheidet, was stillsteht, wenn etwas ausfällt. Verfolgen Sie einen Zonenausfall durch die Essenslieferplattform.

- Die Plattform betreibt ihre zustandslosen Services in drei Availability Zones hinter einem Load Balancer, mit einer verwalteten Bestelldatenbank, die einen synchronen Standby in einer zweiten Zone hält. Kunden finden sie über DNS.
- Jedes Teil hat einen Bereich. Eine einzelne virtuelle Maschine ist zonal: Sie lebt in einer Zone. Die Multi-Zonen-Datenbank ist regional: Der Anbieter hält sie über Zonen kopiert. DNS ist global: Es erstreckt sich über Regionen.
- Eine Überschwemmung legt während des abendlichen Ansturms eine Zone lahm. Die Service-Kopien in dieser Zone fallen mit ihr aus. Der Load Balancer schickt ihnen keine Anfragen mehr, und die Kopien in den anderen beiden Zonen arbeiten weiter und skalieren bei Bedarf hoch.
- Lag die primäre Kopie der Datenbank in der überschwemmten Zone, schaltet der Anbieter nach einer kurzen Pause auf den Standby um: ein automatisches Failover. Weil der Standby synchron ist, hat ihn jeder Schreibvorgang erreicht, bevor die Datenbank ihn bestätigte, wie Kapitel 7 erklärt hat; bestätigte Bestellungen bleiben also erhalten. Bestellungen, die während der Pause mittendrin waren, können fehlschlagen und müssen wiederholt werden.
- Globale Dienste fallen anders aus. Ihre Ausfälle sind selten, aber die Verteilung über Zonen oder sogar Regionen schützt Sie nicht davor, weil jede Zone und jede Region von demselben Dienst abhängt.
- **Die Lehre:** Kennen Sie den Bereich jeder Abhängigkeit, bevor Sie dem Entwurf vertrauen. Zonale Teile brauchen Kopien in mehr als einer Zone; regionale überstehen den Verlust einer Zone, aber nicht ihrer ganzen Region; globale fallen selten aus, dann aber überall zugleich.

Managed Services oder Eigenbetrieb

Ein Managed Service ist eine Komponente (eine Datenbank, Queue, ein Cache, eine Suchmaschine, eine Kubernetes-Control-Plane), die der Anbieter für Sie installiert, betreibt, patcht, sichert und skaliert. Die Alternative ist, dieselbe Open-Source-Software selbst auf virtuellen Maschinen oder in Containern zu betreiben. Die Abwägung lautet Arbeit und Risiko gegen Kontrolle, Kosten und Portabilität.

Managed Services gewinnen, wenn die Komponente nicht differenzierend ist: Jedes Essenslieferunternehmen braucht eine zuverlässige PostgreSQL-Datenbank, aber keines gewinnt Kunden dadurch, wie es Datenbank-Patches einspielt. In der Sprache des Domain-Driven Design sind das generische Subdomänen, die man kaufen statt bauen sollte. Eine verwaltete Datenbank bietet Multi-Zonen-Failover, Point-in-Time-Recovery und Patches für Nebenversionen, die sonst ein qualifiziertes Team in Rufbereitschaft bräuchten. Der Eigenbetrieb gewinnt, wenn Sie eine Version, Erweiterung oder einen Tuning-Parameter brauchen, den der Anbieter nicht anbietet, wenn seine Limits oder Kontingente nicht passen, wenn der Aufpreis für den verwalteten Dienst im großen Maßstab hoch wird oder wenn Portabilität über Umgebungen wichtiger ist als Bequemlichkeit.

Verwaltet heißt nicht wartungsfrei. Der Anbieter erzwingt Hauptversions-Upgrades nach seinem eigenen Zeitplan, wenn alte Versionen das Ende des Supports erreichen, ändert Standardeinstellungen, setzt Kontingente, die Sie im ungünstigsten Moment entdecken, und hat Ausfälle, die Sie nicht beheben, sondern nur aussitzen können. Bei Ihnen bleiben Schemadesign, Abfrageleistung, Kapazitätsentscheidungen, Kosten und der regelmäßige Test, der beweist, dass eine Wiederherstellung funktioniert. Behandeln Sie jeden Managed Service als Abhängigkeit mit eigenen Ausfallarten und eigenen Ausstiegskosten.

Cloud-Grundlagen: Landing Zones und die Kontenhierarchie

Bevor die erste Workload einzieht, braucht ein Cloud-Bestand Grundlagen, und die lassen sich früh viel leichter legen als nachträglich einbauen. Eine **Landing Zone** ist die vorbereitete Umgebung, in die Workloads deployt werden. AWS beschreibt sie als gut konzipierte, skalierbare und sichere Umgebung mit mehreren Konten. Microsofts Azure Landing Zone trennt eine *Plattform-Landing-Zone*, die zentrale Grundlage aus Governance, Sicherheit und gemeinsamen Ressourcen, von *Workload-Landing-Zones*, in denen Produktteams innerhalb dieser Guardrails deployen. So oder so sollte sie als Code aufgebaut und geändert werden (Abschnitt 10.4), und sie ist oft das Erste, was ein Plattformteam verantwortet (Abschnitt 10.7).

Die Entscheidung, die sich später am schwersten ändern lässt, ist die Hierarchie. Jeder Anbieter hat eine Grundeinheit der Isolation und einen Baum, der diese Einheiten gruppiert: AWS-Konten in Organisationseinheiten; Azure-Abonnements unter Verwaltungsgruppen; Google-Cloud-Projekte, optional in Ordnern, unter einer Organisation. Der Baum ist wichtig, weil Richtlinien an ihn gebunden sind. Google-Cloud-Richtlinien werden im Baum nach unten vererbt, und Azure-Abonnements erben die Richtlinien ihrer Verwaltungsgruppen. AWS Service Control Policies legen die maximal verfügbaren Berechtigungen in Mitgliedskonten fest; sie gewähren selbst nichts und wirken nicht auf das Verwaltungskonto, was ein Grund ist, Workloads davon fernzuhalten.

Ein Konto oder eine Subscription ist eine weit stärkere Grenze als ein Tag oder eine Namenskonvention. In AWS hat ein Konto standardmäßig keinen Zugriff auf ein anderes, Kontingente gelten pro Konto, und Kosten lassen sich ihm direkt zuordnen. Trennen Sie deshalb von Anfang an nach Workload und nach Umgebung, denn ein Live-System später zwischen Konten zu verschieben ist ein Migrationsprojekt. AWS berechnet die genutzten Ressourcen, nicht die Anzahl der Konten, auch wenn viele Konten durchaus Automatisierung erfordern.

Für die Essenslieferplattform hat ein sinnvoller Startbaum, angelehnt an die Beispielstruktur von AWS, vier Äste. *Security* enthält ein Log-Archiv-Konto, das Produktteams nicht verändern können, und ein Audit-Konto. *Infrastructure* enthält Netzwerk und gemeinsame Dienste. *Workloads* enthält getrennte Produktions- und Nichtproduktionskonten für Bestellungen, Disposition und Zahlungen, mit strengeren Richtlinien auf dem Zahlungsast. *Sandbox* ist für gefahrlose Experimente da. Guardrails an der Spitze des Baums könnten Regionen außerhalb der genehmigten verbieten und verhindern, dass jemand das Logging abschaltet.

Azures Liste der Designbereiche ist selbst auf anderen Clouds eine nützliche Checkliste: Abrechnung und Mandanten, Identität, Ressourcenorganisation, Netzwerk, Sicherheit, Management, Governance und Automatisierung. Sie müssen nicht alles auf einmal aufbauen; AWS weist darauf hin, dass Sie mit einem Teil seiner Beispieleinheiten beginnen und später weitere ergänzen können. Der übliche Fehler ist das Gegenteil von zu wenig Struktur: Guardrails, die so eng sind, und neue Konten, die so langsam zu bekommen sind, dass Teams sie umgehen. Bieten Sie neue Konten als Self-Service über die Plattform an, mit bereits eingerichteten Guardrails.

Netzwerkgrundlagen: VPCs, private Konnektivität und Egress

Eine **Virtual Private Cloud** (VPC; auf Azure ein virtuelles Netzwerk) ist Ihr privates Netzwerk bei einem Anbieter. Planen Sie zuerst die Adressen. Auf AWS lässt sich der Bereich, mit dem eine VPC angelegt wird, weder in der Größe ändern noch entfernen, sondern nur ergänzen, und zwei VPCs mit überlappenden Bereichen können kein Peering eingehen. Azure gibt für seine Netzwerke und für On-Premises-Standorte denselben Rat. Weisen Sie deshalb jedem Netzwerk aus einem zentralen Plan einen überschneidungsfreien Bereich zu, mit Platz zum Wachsen.

Subnetze unterteilen eine VPC, und auf AWS liegt jedes in einer einzigen Availability Zone, sodass Resilienz bedeutet, sie pro Zone zu wiederholen. Ob ein Subnetz öffentlich oder privat ist, entscheidet sein Routing: Ein öffentliches Subnetz hat eine Route zu einem Internet-Gateway, ein privates hat keine und braucht ein NAT-Gerät für Verbindungen nach außen, und ein isoliertes Subnetz hat keine Route über die VPC hinaus. Die Essenslieferplattform könnte

drei Schichten über drei Zonen betreiben: Load Balancer in öffentlichen Subnetzen, Services in privaten und Datenbanken in isolierten. Subnetze sind nur grobe Zonen; feinere Kontrolle kommt aus den Identitäts- und Segmentierungsmethoden von Abschnitt 8.5.

Als Nächstes die Konnektivität. Peering verbindet zwei Netzwerke, ist aber nicht transitiv: Hat A Peering mit B und mit C, kann B dennoch C nicht über A erreichen. Wenn sich die Netzwerke vervielfachen, wechseln Teams zu einem Hub wie AWS Transit Gateway oder Azures Hub-Spoke-Design, bei dem ein zentrales Netzwerk gemeinsame Gateways, Firewall und DNS enthält. Der Hub wird dann zu einer gemeinsamen Abhängigkeit, weshalb Microsoft einen pro Region empfiehlt. Um ein Rechenzentrum zu erreichen, führt ein Site-to-Site-VPN verschlüsselte Tunnel über das Internet, während AWS Direct Connect oder Azure ExpressRoute eine dedizierte Verbindung bietet; AWS weist darauf hin, dass Direct Connect standardmäßig nicht verschlüsselt ist. Um einen einzelnen Dienst statt eines ganzen Netzwerks zu erreichen, nutzen Sie AWS PrivateLink, Azure Private Link oder Google Cloud Private Service Connect, die ihn ohne Peering oder Internetzugang unter einer privaten Adresse bereitstellen.

Kontrollieren Sie schließlich, was hinausgeht. Viele Umgebungen filtern eingehenden Verkehr sorgfältig und lassen alles hinaus, dabei ist der ausgehende Zugriff der Weg, auf dem gestohlene Daten abfließen, auch durch eingeschleuste KI-Anweisungen (Abschnitt 12.6). Ein NAT-Gateway lässt private Services ausgehende Verbindungen aufbauen und hindert Außenstehende daran, welche aufzubauen, aber es übersetzt nur Adressen und wählt keine Ziele aus. Dafür braucht es eine Egress-Firewall, etwa AWS Network Firewall oder Azure Firewall mit Domainregeln, die so platziert ist, dass der gesamte ausgehende Verkehr sie durchläuft. Der Bestellservice der Plattform erreicht dann nur seine Zahlungs-, Karten- und Messaging-Anbieter, und alles andere wird abgewiesen und protokolliert.

Egress-Pfade haben auch eine Zuverlässigkeits- und eine Kostenseite. Ein Standard-NAT-Gateway von AWS liegt in einer Zone; teilen es sich Workloads aus anderen Zonen, verlieren sie den Internetzugang, wenn diese Zone ausfällt. AWS rät zu einem pro Zone und hat eine regionale Option ergänzt, prüfen Sie also die aktuelle Version. NAT-Gateways werden pro Stunde und pro Gigabyte berechnet, während AWS-Gateway-Endpoints für S3 weder Stunden- noch Datenverarbeitungsgebühren kosten, sodass sich mit Speicherverkehr auf diesem Weg leicht sparen lässt (Abschnitt 10.8).

Was Cloud-native bedeutet

Die Cloud Native Computing Foundation (CNCF), Teil der Linux Foundation und Heimat von Kubernetes, veröffentlicht die meistzitierte Definition von Cloud-native. Version 1.0 (2018) beschrieb Technologien, mit denen Organisationen skalierbare Anwendungen in dynamischen Umgebungen wie Public, Private und Hybrid Clouds bauen und betreiben können, und nannte Container, Service Meshes, Microservices, Immutable Infrastructure und deklarative APIs als

typische Beispiele. Version 1.1, 2024 verabschiedet, verlagert den Schwerpunkt von Technologien auf Praktiken: Arbeitslasten programmatisch und wiederholbar im großen Maßstab zu entwickeln, zu bauen und zu deployen, gekennzeichnet durch „lose gekoppelte Systeme, die auf sichere, resiliente, verwaltbare, nachhaltige und beobachtbare Weise zusammenwirken“.

Daraus folgen zwei Dinge. Cloud-native beschreibt, *wie* ein System gebaut und betrieben wird, nicht *wo*: Ein Kubernetes-Cluster in einem privaten Rechenzentrum, aus Code bereitgestellt und viele Male am Tag deployt, ist Cloud-native; eine von Hand konfigurierte virtuelle Maschine, unverändert in eine Public Cloud kopiert, ist es nicht. Und Cloud-native ist ein Mittel, kein Zweck. Seine Praktiken (Automatisierung, wegwerfbare Komponenten, deklarative Konfiguration, Elastizität, Observability) zahlen sich aus, wenn ein System häufige Änderungen, schwankende Last und hohe Verfügbarkeit braucht; für ein stabiles internes Werkzeug, das zwanzig Menschen nutzen, bringen sie Kosten und Komplexität.

Migrationen werden oft mit den sieben Rs beschrieben (siehe Abschnitt 11.6), die man kennen sollte, weil sie Erwartungen setzen: *Rehost* (Lift and Shift, unverändert), *Relocate* (ganze virtuelle Maschinen oder Plattformen unverändert in die Cloud verschieben), *Replatform* (kleine Änderungen wie der Umzug auf eine verwaltete Datenbank), *Refactor* oder Re-Architect (Neuentwurf für Cloud-nativen Betrieb), *Repurchase* (durch SaaS ersetzen), *Retain* (lassen, wo es ist) und *Retire* (abschalten). Rehosting geht schnell, bringt aber typischerweise wenige Vorteile der Cloud und kann, auf ständig laufenden, für die Spitze dimensionierten Maschinen, mehr kosten als das Rechenzentrum, das es verlassen hat.

Wann die Cloud nicht billiger ist

Das bekannteste Gegenbeispiel ist 37signals, das Unternehmen hinter Basecamp und HEY. Im Oktober 2022 kündigte sein Technikchef David Heinemeier Hansson den Abschied aus der Cloud an und verwies auf eine jährliche AWS-Rechnung von mehr als 3 Millionen USD für Arbeitslasten mit gleichmäßiger, vorhersehbarer Last. Das Unternehmen kaufte Dell-Server für rund 700.000 USD, stellte sie in gemietete Colocation-Flächen und zog seine Anwendungen 2023 von AWS-Rechenkapazität ab; nach eigenen Angaben senkte das seine Cloud-Rechnung um etwa 2 Millionen USD pro Jahr. 2025 kündigte es die zweite Stufe an: etwa 18 Petabyte an Dateien von Amazon S3 auf Storage-Arrays für rund 1,5 Millionen USD zu verlagern, um eine S3-Rechnung von etwa 1,5 Millionen USD pro Jahr zu ersetzen, mit dem erklärten Ziel, sein AWS-Konto zu schließen. Das Unternehmen hat Einsparungen von etwa 10 Millionen USD über fünf Jahre prognostiziert. Arbeitslasten auf diese Weise aus der Cloud zurückzuholen, nennt man Cloud-Repatriierung.

Beide Seiten verdienen Gehör. Für die Überlegungen von 37signals spricht: Die Last ist gleichmäßig, Elastizität brachte also wenig; das Unternehmen beschäftigte bereits erfahrene Betriebsfachleute und brauchte nach eigenen Angaben kein zusätzliches Personal; Colocation-Anbieter kümmern sich um Strom, Kühlung und physische Sicherheit; und Server lassen sich fünf Jahre oder länger nutzen, sodass ihre Kosten pro Rechenjahr weit unter der On-Demand-Miete liegen. Dagegen spricht: 37signals ist ein reifes Unternehmen mit vorhersehbarer Nachfrage, was die meisten neuen Produkte nicht sind; der Vergleich hängt stark davon ab, wie hoch die Cloud-Rechnung mit Rabatten für Nutzungszusagen und richtiger Dimensionierung gewesen wäre; eigene Hardware bedeutet Investitionsausgaben, Lieferzeiten von Wochen oder Monaten und im Voraus gekaufte Kapazität; und ein kleineres Team ohne tiefes Betriebswissen müsste solches einstellen, was die Einsparung zunichtemachen kann. Kritiker weisen außerdem darauf hin, dass die Zahlen zu den Einsparungen vom Unternehmen selbst stammen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Rechenkapazität für die Essenslieferplattform mieten oder besitzen

Alle Preise sind veranschaulichende Annahmen, die die Methode zeigen sollen, keine echten Angebote. On-Demand-Cloud-Kapazität kostet £0,040 pro vCPU-Stunde; eine dreijährige Nutzungszusage kostet £0,024 (40 % Rabatt); eigene Hardware in Colocation kostet £0,012 pro vCPU-Stunde einschließlich Strom und Fläche, braucht aber 30 % Kapazitätsreserve und Personalzeit. Die Plattform braucht den ganzen Tag 40 vCPUs, im vierstündigen Abendansturm bis zu 120 vCPUs (etwa 122 Stunden im Monat). Ein Monat hat 730 Stunden.

A: Cloud, für die Spitze dimensioniert, ständig in Betrieb	120 vCPU × 730 h × £0,040 = **£3504 pro Monat**
B: Cloud, zugesagte Grundlast plus automatisch skalierte Spitze	40 × 730 × £0,024 = £701; plus 80 × 122 × £0,040 = £390; zusammen **£1091 pro Monat**
C: eigene Hardware, für die Spitze dimensioniert	120 × 1,3 × 730 × £0,012 = £1367; plus ein Viertel einer Fachkraft zu £80.000 pro Jahr = £1667; zusammen **£3034 pro Monat**
Jetzt größer: gleichmäßig 2000 vCPUs, keine Stoßzeit	Cloud mit Zusage: 2000 × 730 × £0,024 = £35.040. Eigene Hardware: 2000 × 1,3 × 730 × £0,012 = £22.776 plus eine Fachkraft (£6667) = **£29.443**, etwa 16 % billiger

Bei einer kleinen, stoßweisen Arbeitslast dominiert die Elastizität: Option B kostet weniger als ein Drittel von Option A und etwa ein Drittel von Option C, weil sie Spitzenkapazität nur während des Ansturms bezahlt.

Bei einer großen, flachen Arbeitslast verteilen sich die Fixkosten des Besitzes auf mehr Kapazität, und Besitzen beginnt zu gewinnen – das ist die Lage von 37signals. Die Einsparung ist real, aber kleiner, als Schlagzeilenvergleiche nahelegen, sobald Kapazitätsreserve, Personal und Migrationsaufwand mitgezählt werden.

Die Methode zählt mehr als die Zahlen: Modellieren Sie die Form der Last, rechnen Sie Menschen und Kapazitätsreserve ein, vergleichen Sie mit dem besten Cloud-Preis (mit Zusagen), nicht mit dem Listenpreis, und wiederholen Sie die Rechnung, wenn sich das Geschäft ändert.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: ein Hosting-Modell für eine Arbeitslast wählen

Wenden Sie sie je Arbeitslast an, nicht einmal für die ganze Organisation: Ein einzelnes Unternehmen nutzt sinnvollerweise SaaS für die Gehaltsabrechnung, Funktionen für die Größenänderung von Bildern und Container für seine Kernservices.

Q1 **Ist die Fähigkeit generisch, mit einem Produkt am Markt, das gut genug passt (E-Mail, Gehaltsabrechnung, CRM, Zahlungen)?**

- **Ja** Kaufen Sie SaaS und integrieren Sie es über seine API; stecken Sie den Entwurfsaufwand in die Integration und den Exit-Plan.
- **Nein** Weiter: Sie müssen eigene Software betreiben.

Q2 **Schließen Recht, Vertrag oder physischer Standort die Public Cloud für diese Arbeitslast aus (strenge Souveränitätsregeln, Geräte vor Ort, Regelkreise im Millisekundenbereich)?**

- **Ja** Nutzen Sie eine Private Cloud, Colocation oder ein Edge-Deployment vor Ort, verbunden mit Public-Cloud-Diensten als Hybrid, wo erlaubt.
- **Nein** Weiter mit Optionen in der Public Cloud.

Q3 **Ist die Arbeit ereignisgesteuert, kurzlebig und stoßweise oder von geringem Volumen (Webhooks, Dateiverarbeitung, geplante Jobs)?**

- **Ja** Nutzen Sie Functions as a Service oder eine Serverless-Container-Plattform; prüfen Sie Grenzen bei Kaltstart, Laufzeit und Kosten bei hohem Volumen.
- **Nein** Weiter: Das ist ein lang laufender Service.

Q4 **Passt das Modell eines PaaS oder einer Serverless-Container-Plattform (zustandsloser HTTP-Service, Standard-Laufzeitumgebung, wenige Sonderwünsche)?**

- **Ja** Nutzen Sie es. Das nimmt einem kleinen Team die meiste Betriebsarbeit ab.
- **Nein** Weiter: Sie brauchen mehr Kontrolle.

Q5 **Betreiben Sie viele Services über mehrere Teams und haben (oder finanzieren) Sie ein Team, das Kubernetes betreiben kann?**

- **Ja** Nutzen Sie Managed Kubernetes (CaaS), idealerweise hinter einer internen Plattform (Abschnitt 10.7).
- **Nein** Nutzen Sie virtuelle Maschinen mit verwalteten Datenbanken und Autoscaling-Gruppen, aus Infrastrukturcode gebaut.

Q6 **Ist die Last groß, gleichmäßig und über Jahre vorhersehbar, mit Betriebswissen im eigenen Haus?**

Ja Stellen Sie eigene oder per Colocation betriebene Hardware den Kosten zugesagter Cloud-Kapazität gegenüber; Repatriierung kann sich lohnen.

Nein Bleiben Sie bei gemieteter Kapazität und nutzen Sie Rabatte für Nutzungszusagen für die gleichmäßige Grundlast.

ACHTUNG

Häufige Fehler bei der Wahl von Cloud-Modellen

- Alles unverändert auf ständig laufende, für die Spitze dimensionierte virtuelle Maschinen umziehen und dann schließen, die Cloud sei teuer.
- Multi-Cloud als allgemeine Absicherung einführen und die Kosten der Portabilität bei jeder Arbeitslast zahlen, statt einen Ausstieg für die wenigen zu planen, die einen brauchen.
- Annehmen, der Anbieter sichere, verschlüssele und beschränke den Zugriff auf Ihre Daten standardmäßig; in jedem Modell sind das Ihre Entscheidungen.
- Kubernetes wählen, bevor es ein Team gibt, das es betreibt, oder genug Services, die es rechtfertigen.
- Gebühren für Datentransfer ignorieren, besonders zwischen Zonen, Regionen und Anbietern, wenn das Deployment-Diagramm gezeichnet wird.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Ein Team verlegt seine Datenbank von einer selbst betriebenen VM in einen verwalteten Datenbankdienst. Nennen Sie zwei Verantwortlichkeiten, die an den Anbieter übergehen, und zwei, die beim Team bleiben.

Der Anbieter übernimmt das Patchen der Engine, die Hardware und meist automatische Backups und Zonen-Failover. Das Team behält Schema- und Abfragedesign, Zugriffsberechtigungen, Entscheidungen zur Verschlüsselung, Kapazität und Kosten, das Testen von Wiederherstellungen und die Planung von Hauptversions-Upgrades.

▼ Warum bevorzugt ein quorumbasiertes System wie etcd drei Availability Zones statt zwei?

Ein Quorum braucht eine Mehrheit. Bei zwei Zonen stoppt der Verlust der Zone mit der Mehrheit das System; bei drei kann jede einzelne Zone ausfallen, und eine Mehrheit überlebt.

▼ **Ist eine von Hand gebaute virtuelle Maschine, unverändert in eine Public Cloud kopiert, Cloud-native?**

Nein. Cloud-native beschreibt Praktiken wie Automatisierung, deklarative Konfiguration, wegwerfbare Komponenten und Observability, nicht einen Ort. Rehosting verlegt das System, ohne diese Praktiken zu übernehmen.

▼ **Warum kann dieselbe Organisation sinnvollerweise SaaS, Funktionen und Kubernetes gleichzeitig nutzen?**

Das Hosting-Modell sollte je Arbeitslast gewählt werden: Generische Fähigkeiten werden als SaaS gekauft, stoßweise ereignisgesteuerte Aufgaben passen zu Funktionen, und lang laufende Kernservices können Container auf Kubernetes rechtfertigen.

▼ **Probieren Sie es aus: Zur Veranschaulichung: On-Demand kostet £0,05 pro vCPU-Stunde, eine Nutzungszusage £0,03. Sie brauchen ständig 20 vCPUs und zusätzlich 60 für fünf Stunden am Tag, in einem Monat mit 30 Tagen. Vergleichen Sie reines On-Demand, die Zusage für die Grundlast und die Zusage für die gesamte Spitze.**

Grundlast: 20 x 720 = 14.400 vCPU-Stunden; zusätzliche Spitze: 60 x 150 = 9000. Alles On-Demand: 23.400 x £0,05 = £1170. Grundlast committen: 14.400 x £0,03 + 9000 x £0,05 = £882. 80 vCPUs den ganzen Monat committen: 80 x 720 x £0,03 = £1728. Committen Sie nur die sichere Grundlast, und lassen Sie On-Demand den Ansturm abdecken. Eigene Hardware wäre nur bei großer, gleichmäßiger Last sinnvoll.

QUELLEN

- NIST SP 800-145: The NIST Definition of Cloud Computing (2011): https://csrc.nist.gov/pubs/sp/800/145/final
- CNCF Technical Oversight Committee: Cloud Native Definition: https://github.com/cncf/toc/blob/main/DEFINITION.md
- AWS: Shared Responsibility Model: https://aws.amazon.com/compliance/shared-responsibility-model/
- Microsoft Learn: Shared responsibility in the cloud: https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility
- The Register (2025): 37signals is completing its on-prem move: https://www.theregister.com/2025/05/09/37signals_cloud_repatriation_storage_savings
- Designing an AWS Control Tower landing zone (AWS Prescriptive Guidance): https://docs.aws.amazon.com/prescriptive-guidance/latest/designing-control-tower-landing-zone/introduction.html

- Sample account structure and OUs (AWS Prescriptive Guidance): https://docs.aws.amazon.com/prescriptive-guidance/latest/designing-control-tower-landing-zone/account-structure.html
- What is an Azure landing zone? (Microsoft Cloud Adoption Framework): https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/
- Azure landing zone design areas (Microsoft): https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/design-areas
- Organizing your AWS environment using multiple accounts (AWS whitepaper): https://docs.aws.amazon.com/whitepapers/latest/organizing-your-aws-environment/organizing-your-aws-environment.html
- Service control policies (AWS Organizations): https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html
- Resource hierarchy (Google Cloud): https://docs.cloud.google.com/resource-manager/docs/cloud-platform-resource-hierarchy
- VPC CIDR blocks (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/vpc-cidr-blocks.html
- Subnets for your VPC (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/configure-subnets.html
- VPC peering basics (AWS): https://docs.aws.amazon.com/vpc/latest/peering/vpc-peering-basics.html
- What is AWS Transit Gateway?: https://docs.aws.amazon.com/vpc/latest/tgw/what-is-transit-gateway.html
- Hub-spoke network topology in Azure (Microsoft): https://learn.microsoft.com/en-us/azure/architecture/networking/architecture/hub-spoke
- AWS Direct Connect (Amazon VPC connectivity options whitepaper): https://docs.aws.amazon.com/whitepapers/latest/aws-vpc-connectivity-options/aws-direct-connect.html
- What is AWS PrivateLink?: https://docs.aws.amazon.com/vpc/latest/privatelink/what-is-privatelink.html
- Private Service Connect (Google Cloud): https://docs.cloud.google.com/vpc/docs/private-service-connect
- NAT gateways (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html
- NAT gateway basics (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/nat-gateway-basics.html
- Amazon VPC pricing: https://aws.amazon.com/vpc/pricing/
- What is AWS Network Firewall?: https://docs.aws.amazon.com/network-firewall/latest/developerguide/what-is-aws-network-firewall.html

Container und Orchestrierung mit Kubernetes

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.7 Server und die Cloud
- 9.1 Skalierung
- 9.2 Load Balancing

Begriffe von früher, die er verwendet: Container, Autoscaling, Availability Zone, PaaS (Platform as a Service)

DIE IDEE IN KÜRZE

Denken Sie an einen großen Hafen. Waren kommen in genormten Schiffscontainern an, sodass jeder Kran, jedes Schiff und jeder Lkw sie handhaben kann, ohne zu wissen, was darin ist. Der Hafenmeister bewegt die Kisten nicht selbst; er führt eine Tafel, auf der steht, was wo sein soll, und seine Leute gleichen den Kai ständig mit der Tafel ab und beheben jede Abweichung: eine fehlende Kiste, ein defekter Kran, ein verspätetes Schiff. Software-Container sind die genormten Kisten, und Kubernetes ist das Büro des Hafenmeisters: Sie schreiben auf, was laufen soll, und es prüft und korrigiert so lange, bis die Wirklichkeit übereinstimmt.

WAS SIE WISSEN MÜSSEN

- Ein Container-Image ist ein Stapel inhaltsadressierter Layer plus Konfiguration und Manifest; Tags können sich verschieben, Digests nicht, daher sollte die Produktion Images ausführen, die per Digest fixiert sind.
- Die OCI-Spezifikationen für Runtime, Image und Distribution erlauben, dass mit beliebigen Werkzeugen gebaute Images in jeder Registry gespeichert und von jeder konformen Runtime ausgeführt werden.
- Kubernetes hat eine Control Plane (API-Server, etcd, Scheduler, Controller Manager) und Worker Nodes (kubelet, Container-Runtime, kube-proxy); nur der API-Server spricht mit etcd, und die Nodes holen sich ihre Arbeit selbst.
- Controller führen pegelgesteuerte Reconciliation Loops aus, die den tatsächlichen mit dem gewünschten Zustand vergleichen und auf die Differenz reagieren; das macht das System robust gegen Ausfälle und verpasste Ereignisse.

- Pods sind wegwerfbar; Deployments rollen sie aus, Services geben ihnen stabile Namen, Gateway API leitet Verkehr hinein, StatefulSets geben Identität und Speicher, Jobs laufen bis zum Abschluss, und der HPA skaliert die Replikas.
- Requests steuern Scheduling und Autoscaling, Limits deckeln die Nutzung (CPU wird gedrosselt, bei Speicherüberschreitung wird beendet); Liveness Probes dürfen keine geteilten Abhängigkeiten prüfen.
- Kubernetes lohnt sich bei vielen Services, mehreren Teams und einem Plattformteam; für wenige zustandslose Services ist ein PaaS oder eine Serverless-Container-Plattform meist die bessere Wahl.

Images, Layer und Registrys

Kapitel 0 hat Container als isolierte Prozesse eingeführt, die sich den Kernel des Hosts teilen und als Images aus einem Dockerfile gebaut werden (Server und die Cloud). Ein Architekt braucht etwas mehr Detail, denn das Image-Design beeinflusst Build-Geschwindigkeit, Sicherheit und wie zuverlässig dieselbe Software jede Umgebung erreicht.

Ein Container-Image ist ein Stapel schreibgeschützter *Layer*, jeder eine komprimierte Menge von Dateiänderungen, plus ein kleines Konfigurationsdokument (der auszuführende Befehl, Umgebungsvariablen, der Benutzer) und ein *Manifest*, das sie auflistet. Jeder Layer und jedes Manifest wird durch einen *Digest* identifiziert, einen SHA-256-Hash seines Inhalts. Identische Layer werden nur einmal gespeichert und heruntergeladen, sodass hundert Images auf derselben Basis deren Layer teilen. Weil jede Anweisung in einem Dockerfile einen Layer erzeugt und Builder zwischengespeicherte Layer wiederverwenden, bis sich etwas ändert, ordnet man die Anweisungen von selten zu häufig geändert: Basis-Image, dann Abhängigkeiten, dann Anwendungscode. *Multi-Stage-Builds* kompilieren in einer Stufe und kopieren nur das Ergebnis in ein kleines finales Image; Compiler und Build-Werkzeuge bleiben zurück.

Images liegen in einer *Registry*: Docker Hub, GitHub Container Registry, Amazon ECR, Azure Container Registry, Google Artifact Registry oder eine selbst betriebene wie Harbor. Ein *Tag* wie `orders:2.4.1` ist ein verschiebbares Etikett; wer Push-Rechte hat, kann es morgen auf ein anderes Image zeigen lassen. Ein Digest wie `orders@sha256:3f1c...` kann seine Bedeutung nie ändern. Produktions-Deployments sollten Tags deshalb in Digests auflösen, damit genau das läuft, was getestet wurde, und nie `latest` verwenden.

Kleine Images sind schneller heruntergeladen und haben weniger Komponenten, die gepatcht werden müssen. Übliche Praxis ist eine minimale Basis (eine schlanke Distribution oder ein Distroless-Image, das nur die Laufzeitumgebung enthält), die Ausführung als Nicht-Root-Benutzer, regelmäßiges Neubauen, um Sicherheitskorrekturen aufzunehmen, und das Scannen

der Images auf bekannte Schwachstellen in der Pipeline. Das Signieren von Images und das Anhängen einer Software-Stückliste (SBOM) gehören zur Sicherheit der Software-Lieferkette, die Kapitel 8 behandelt (Sicherheit der Software-Lieferkette).

Die OCI-Standards

Docker machte Container ab 2013 populär, und 2015 gründeten Docker und andere unter dem Dach der Linux Foundation die Open Container Initiative (OCI), um eine Zersplitterung des Formats zu verhindern. Die OCI pflegt drei Spezifikationen. Die *Runtime-Spezifikation* legt fest, wie ein entpackter Container ausgeführt wird (Namespaces, Control Groups, Mounts, der zu startende Prozess); `runc` ist ihre Referenzimplementierung. Die *Image-Spezifikation* definiert Layer, Manifeste und Konfiguration. Die *Distribution-Spezifikation* definiert die HTTP-API, die Registrys für Push und Pull bereitstellen. Version 1.1 der Image- und der Distribution-Spezifikation (2024) brachte Standardwege, andere Artefakte in Registrys abzulegen und mit einem Image zu verknüpfen; so reisen Signaturen und Software-Stücklisten heute neben den Images, die sie beschreiben.

Das praktische Ergebnis ist Austauschbarkeit. Ein mit Docker, Podman, Buildah oder einem Cloud-Build-Dienst gebautes Image kann in jeder konformen Registry gespeichert und von jeder konformen Runtime ausgeführt werden: containerd und CRI-O auf einem Kubernetes-Node, eine Serverless-Container-Plattform oder der Laptop eines Entwicklers. Deshalb wurden Container, anders als die meisten Paketformate davor, zu einer gemeinsamen Sprache zwischen Entwicklung, Betrieb und jeder großen Cloud.

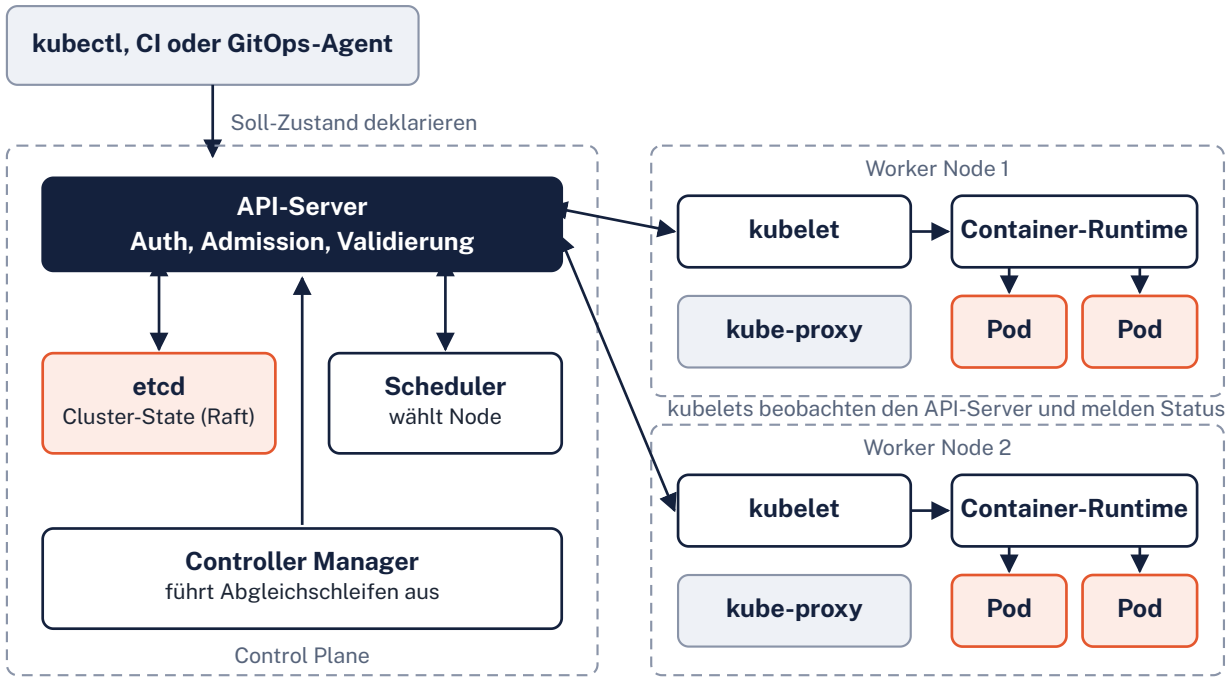
Warum Orchestrierung nötig ist

Einen Container zu starten, ist ein einziger Befehl. Die Essenslieferplattform zu betreiben, ist es nicht: vierzig Services mit je drei bis dreißig Kopien, verteilt über Dutzende Maschinen in drei Availability Zones, und stündlich geht irgendwo eine neue Version hinaus. Jemand muss entscheiden, auf welcher Maschine jede Kopie läuft, abgestürzte Kopien neu starten, ausgefallene Maschinen ersetzen, für den Abendansturm Kopien hinzufügen und danach wieder entfernen, neue Versionen ausrollen, ohne Anfragen zu verlieren, Services einander finden lassen, während Kopien kommen und gehen, ihnen Konfiguration und Secrets übergeben und den wenigen, die es brauchen, Speicher anhängen.

Container-Orchestrierung ist Software, die all das anhand einer Deklaration dessen erledigt, was laufen soll. Google betrieb dafür mehr als ein Jahrzehnt lang ein internes System namens Borg; 2014 kündigte es Kubernetes an, ein Open-Source-System, das auf dieser Erfahrung aufbaut, veröffentlichte im Juli 2015 Version 1.0 und übergab es der neu gegründeten Cloud Native Computing Foundation, wo es 2018 als erstes Projekt den Status „graduated“ erreichte.

Es gibt Alternativen (HashiCorp Nomad, Amazon ECS, Serverless-Container-Plattformen), aber Kubernetes ist heute die gemeinsame Grundlage, die jeder große Anbieter und die meisten On-Premises-Plattformen anbieten.

Kubernetes-Architektur



Jede Komponente spricht mit dem API-Server; nur der API-Server spricht mit etcd, und die Nodes holen sich ihre Arbeit, statt Befehle zugeschoben zu bekommen.

Wie Kubernetes funktioniert: Control Plane und Nodes

Ein Kubernetes-Cluster hat eine Control Plane, die Entscheidungen trifft, und Worker Nodes, die die Arbeitslasten ausführen. Der **API-Server** ist die Eingangstür: Jeder Client, vom `kubectl` eines Entwicklers bis zu den clustereigenen Komponenten, liest und schreibt Objekte über seine REST-API, und er authentifiziert den Aufrufer, prüft die Autorisierung und führt *Admission*-Kontrollen aus, die jede Anfrage validieren oder verändern können (etwa Container ablehnen, die als Root laufen). Er ist die einzige Komponente, die mit **etcd** spricht, einem stark konsistenten Key-Value-Store, der den Cluster-Zustand mit dem Konsensprotokoll Raft über typischerweise drei oder fünf Mitglieder repliziert, sodass beim Verlust von einem oder zwei eine Mehrheit überlebt (Konsistenzmodelle, CAP und PACELC).

Der **Scheduler** beobachtet neue Pods ohne zugewiesenen Node, filtert Nodes aus, die sie nicht ausführen können (zu wenig unreservierte CPU oder zu wenig Speicher, falsche Zone, fehlende Hardware, Taints, die der Pod nicht toleriert), bewertet die übrigen (Kopien über Zonen verteilen, effizient packen) und hält seine Wahl über den API-Server fest. Der **Controller Manager** führt Dutzende Controller aus, jeder zuständig für eine Art von Objekt: Der

ReplicaSet-Controller hält die richtige Zahl von Pods, der Node-Controller bemerkt Nodes, die sich nicht mehr melden, der Job-Controller führt Aufgaben bis zum Abschluss aus. In einer Cloud legt zusätzlich ein Cloud Controller Manager über die API des Anbieters Load Balancer an und hängt Datenträger an.

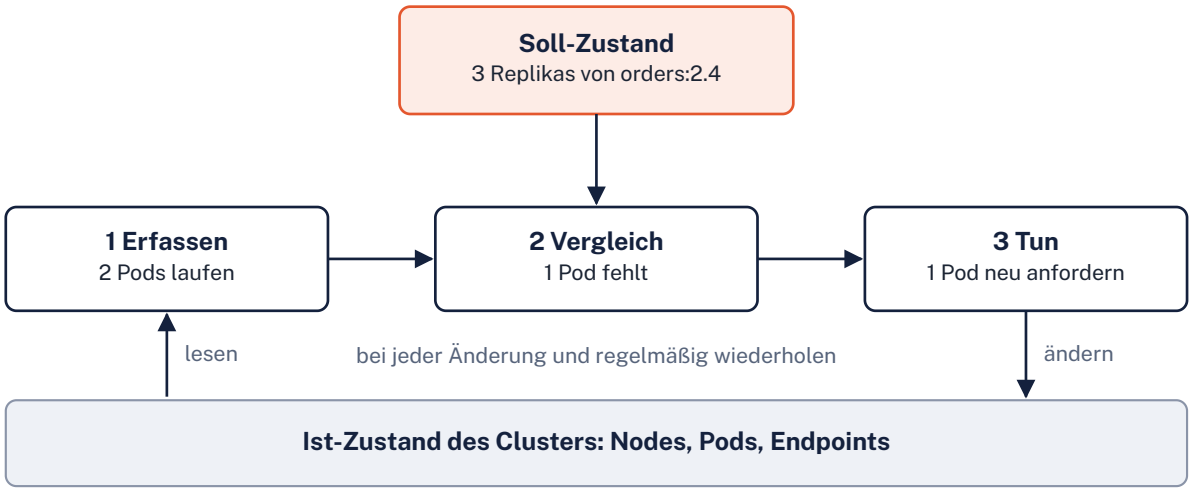
Auf jedem Worker Node läuft ein **kubelet**, ein Agent, der den API-Server auf Pods beobachtet, die seinem Node zugewiesen sind, die **Container-Runtime** (meist containerd oder CRI-O, über das Container Runtime Interface; der eingebaute Shim für Docker Engine wurde 2022 mit Version 1.24 entfernt) bittet, sie zu starten, ihre Health Probes ausführt und den Status zurückmeldet. **kube-proxy** programmiert die Paketweiterleitungsregeln des Nodes, sodass Verkehr an die virtuelle Adresse eines Service einen seiner Pods erreicht; manche Netzwerk-Plug-ins wie Cilium ersetzen es durch eine eBPF-Datenebene. Ein Netzwerk-Plug-in gibt jedem Pod eine eigene IP-Adresse, und ein Cluster-DNS-Dienst (meist CoreDNS) lässt Pods Services über ihren Namen finden.

Beachten Sie die Richtung der Pfeile. Die Control Plane meldet sich nicht auf einem Node an, um Workloads zu starten; die Nodes beobachten den Zustand, in dem sie sein sollen, und nähern sich ihm an. (Befehle auf Abruf sind die Ausnahme: Bei `kubectl exec`, `logs` und `port-forward` ruft der API-Server das Kubelet auf dem Node auf.) Dieses Pull-basierte Beobachten und Annähern macht Kubernetes robust: Ein Node, der zehn Minuten getrennt war, holt einfach auf, wenn er zurückkommt. Bei Managed Kubernetes (Amazon EKS, Azure Kubernetes Service, Google Kubernetes Engine und andere) betreibt und aktualisiert der Anbieter die Control Plane; der Kunde wählt weiterhin die Node-Typen, aktualisiert die Node-Pools und ist für alles verantwortlich, was in den Cluster deployt wird.

Deklarativer gewünschter Zustand und Reconciliation Loops

Die zentrale Idee von Kubernetes ist deklarative Konfiguration. Sie sagen dem Cluster nicht, er solle drei Container starten; Sie speichern ein Objekt, das besagt, dass drei Kopien eines bestimmten Images existieren sollen: den gewünschten Zustand. Controller führen dann Reconciliation Loops aus: den tatsächlichen Zustand beobachten, ihn mit dem gewünschten vergleichen und handeln, um die Differenz zu beseitigen, ohne Ende.

Der Reconciliation Loop



Controller sind pegelgesteuert: Sie reagieren auf die aktuelle Differenz, nicht auf eine gemerkte Abfolge von Ereignissen, daher richten verpasste Ereignisse und Neustarts keinen bleibenden Schaden an.

Verfolgen Sie, was geschieht, wenn während des Abendansturms ein Node in Zone B ausfällt. Der Node-Controller empfängt keine Heartbeats mehr und markiert den Node als nicht bereit; nach einem Timeout werden seine Pods zur Verdrängung (Eviction) vorgemerkt. Der ReplicaSet-Controller des Bestellservice beobachtet zwei laufende Pods, wo drei gewünscht sind, und legt ein neues Pod-Objekt an. Der Scheduler sieht einen nicht eingeplanten Pod und weist ihn einem gesunden Node zu, je nach Verteilungsregeln bevorzugt einem verbliebenen Node in Zone B oder einer anderen Zone. Das kubelet dieses Nodes sieht die Zuweisung, lädt das Image und startet den Container. Sobald seine Readiness Probe besteht, werden die Endpoints hinter dem Service orders aktualisiert, und Verkehr fließt zu ihm. Keine Komponente hat einer anderen gesagt, was zu tun ist; jede hat auf den Zustand reagiert, den sie beobachtet hat.

Weil Controller *pegelgesteuert* sind (sie reagieren auf die aktuelle Differenz) statt *flankengesteuert* (auf einzelne Ereignisse reagierend), kommt ein Controller, der abstürzt, eine Benachrichtigung verpasst oder doppelt läuft, trotzdem zum richtigen Ergebnis. Das Muster lässt sich erweitern: *Custom Resource Definitions* fügen neue Objekttypen hinzu, und das *Operator*-Muster, 2016 von CoreOS eingeführt, verpackt einen Controller, der Betriebswissen kodiert, etwa um einen PostgreSQL-Cluster aus einem einzigen deklarierten Objekt anzulegen, zu sichern und per Failover umzuschalten. Abschnitt 10.4 zeigt dieselbe Schleife, angewandt auf ganze Umgebungen durch GitOps.

Deklarative Steuerung hat ihren Preis: Sie müssen in Endzuständen denken und die genaue Abfolge loslassen. Wollen zwei Controller für dasselbe Feld unterschiedliche gewünschte Zustände, kämpfen sie gegeneinander. Ein klassisches Beispiel ist ein Deployment-Manifest in Git mit `replicas: 3`, während ein Autoscaler den Wert auf neun setzt; jede GitOps-Synchronisierung skaliert den Service mitten im Ansturm wieder herunter. Die Lösung ist, jedes Feld genau einem Eigentümer zu überlassen, hier durch das Entfernen von `replicas` aus dem Manifest.

Die zentralen Kubernetes-Objekte

Objekt	Was er ist	Beispiel Essenslieferung
Pod	Ein oder mehrere Container, die gemeinsam auf einem Node eingeplant werden und sich eine IP-Adresse und Volumes teilen. Pods sind wegwerfbar und werden ersetzt, nie repariert.	Eine Kopie des Bestellservice, vielleicht mit einem kleinen Sidecar für den Log-Versand.
ReplicaSet	Hält eine festgelegte Zahl identischer Pods am Laufen. Wird selten von Hand geschrieben.	Vom Deployment orders angelegt und verwaltet.
Deployment	Deklariert die Pod-Vorlage und die Zahl der Replikas einer zustandslosen Anwendung und rollt Änderungen aus, indem es Pods innerhalb von Surge- und Nichtverfügbarkeitsgrenzen von einem alten ReplicaSet in ein neues verschiebt.	Bestell-, Speisekarten- und Preisgestaltungsservice.
Service	Ein stabiler Name und eine virtuelle IP für eine wechselnde, per Label ausgewählte Menge von Pods, mit Lastverteilung über sie.	`orders.default.svc`, genutzt vom Checkout-Service.
Ingress und Gateway API	Leiten externen HTTP(S)-Verkehr nach Host und Pfad an Services. Gateway API (allgemein verfügbar seit Version 1.0 im Oktober 2023) ist der ausdrucksstärkere, rollenorientierte Nachfolger: Das Plattformteam besitzt das Gateway, die Anwendungsteams besitzen ihre HTTPRoutes.	`api.example.com/orders`, mit TLS an den Service orders geleitet.
ConfigMap	Nicht geheime Konfiguration, an Pods als Umgebungsvariablen oder Dateien übergeben.	Feature-Einstellungen, Timeouts, die URL des Speisekarten-Service.
Secret	Sensible Werte, übergeben wie ConfigMaps. In etcd nur Base64-kodiert gespeichert, sofern keine Verschlüsselung im Ruhezustand konfiguriert ist; viele Teams synchronisieren sie aus einem externen Secrets-Manager.	Der API-Schlüssel des Zahlungsanbieters, das Datenbankpasswort.
StatefulSet	Pods mit stabilen Namen, stabilen Netzwerkidentitäten und eigenen persistenten Volumes, die der Reihe nach gestartet und gestoppt werden.	Ein selbst betriebener Kafka- oder Redis-Cluster, falls kein Managed Service genutzt wird.
DaemonSet	Genau ein Pod auf jedem (oder jedem ausgewählten) Node.	Log-Collector oder Monitoring-Agent für die Nodes.

Objekt	Was er ist	Beispiel Essenslieferung
Job und CronJob	Pods, die bis zum Abschluss laufen, einmalig oder nach Zeitplan, mit Wiederholungen.	Datenbankmigration vor einem Release; nächtlicher Auszahlungsbericht für die Restaurants.
HorizontalPodAutoscaler	Passt die Zahl der Replikas eines Deployments an, um eine Metrik nahe einem Zielwert zu halten.	Den Bestellservice von 3 auf 12 Replikas skalieren, wenn die CPU-Nutzung am Abend steigt.

HINWEIS

Ingress-Controller und der Wechsel zu Gateway API

Das Ingress-Objekt deklariert nur Routing-Regeln; ein *Ingress-Controller* setzt sie um. Im November 2025 kündigte das Kubernetes-Projekt das Ende des weit verbreiteten Community-Controllers ingress-nginx an, mit einer Wartung nach bestem Bemühen bis März 2026 und danach keinen Sicherheitskorrekturen mehr, und empfahl Gateway API für neue Arbeiten. Die Ingress-API selbst funktioniert mit anderen Controllern weiter, aber Teams mit ingress-nginx mussten eine Migration planen – eine Erinnerung daran, dass selbst eine De-facto-Standardkomponente einer Plattform einen Lebenszyklus hat, den man verfolgen muss.

Ein Deployment und ein Service in YAML

Die folgenden Manifeste deklarieren den Bestellservice: Konfiguration aus einem Secret, Resource Requests und Probes, ein stabiler Service davor und ein Autoscaler, der zwischen drei und zwölf Replikas hält. Das Deployment hat bewusst kein Feld `replicas`, damit der Autoscaler der einzige Eigentümer der Anzahl ist. Angewendet werden sie mit `kubectl apply` oder, besser, durch eine Pipeline oder einen GitOps-Agenten aus einem Repository.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders
spec:
  # no replicas field: the autoscaler below owns the count
  selector:
    matchLabels: {app: orders}
  template:
    metadata:
      labels: {app: orders}
    spec:
      containers:
      - name: orders
        image: registry.example.com/orders:2.4.1
        ports: [{containerPort: 8080}]
        env:
        - name: DATABASE_URL
          valueFrom: {secretKeyRef: {name: orders-db, key: url}}
        resources:
          requests: {cpu: 500m, memory: 512Mi}
          limits: {memory: 512Mi}  # no CPU limit: may burst
        readinessProbe:
          httpGet: {path: /ready, port: 8080}
          periodSeconds: 5
        livenessProbe:
          httpGet: {path: /live, port: 8080}
          periodSeconds: 10
          failureThreshold: 3
```

Ein Deployment für den Bestellservice. In der Produktion wäre das Image per Digest statt per Tag fixiert.

```
apiVersion: v1
kind: Service
metadata:
  name: orders
spec:
  selector: {app: orders}
  ports: [{port: 80, targetPort: 8080}]
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: orders
spec:
  scaleTargetRef: {apiVersion: apps/v1, kind: Deployment, name: orders}
  minReplicas: 3
  maxReplicas: 12
  metrics:
  - type: Resource
    resource:
      name: cpu
      target: {type: Utilization, averageUtilization: 60}
```

Ein Service, der den Pods eine stabile Adresse gibt, und ein Autoscaler, der 60 % durchschnittliche CPU-Nutzung der angeforderten CPU anstrebt.

Resource Requests, Limits und Quality of Service

Requests und Limits sind die folgenreichsten Zeilen eines Manifests. Ein *Request* ist das, was der Scheduler reserviert: Ein Pod, der 500 Millicores (eine halbe CPU) und 512 MiB Speicher anfordert, wird nur auf einem Node mit so viel unreservierter Kapazität platziert, und dieser Anteil ist ihm bei Konkurrenz garantiert. Ein *Limit* ist eine vom Kernel durchgesetzte Obergrenze: Ein Container, der sein CPU-Limit überschreitet, wird gedrosselt (verlangsamt), und einer, der sein Speicherlimit überschreitet, wird beendet und neu gestartet, der bekannte Status `OOMKilled`. Kubernetes leitet aus diesen Einstellungen eine Quality-of-Service-Klasse ab: *Guaranteed* (Requests gleich Limits für jede Ressource), *Burstable* (Requests gesetzt, Limits höher oder nicht vorhanden) und *BestEffort* (nichts gesetzt, als Erstes verdrängt, wenn einem Node die Ressourcen ausgehen).

Bei CPU-Limits sind sich Fachleute uneinig. Das eine Lager setzt sie überall, um Vorhersehbarkeit und Fairness zwischen Teams zu erreichen; das andere setzt genaue CPU-Requests und keine CPU-Limits, weil Drosselung Latenzspitzen verursacht, selbst wenn der Node freie CPU hat, und Requests jedem Pod seinen Anteil ohnehin garantieren. Beim Speicher gibt es wenig Streit: Setzen Sie das Limit gleich dem Request, denn Speicher lässt sich einem Prozess nicht entziehen, ohne ihn zu beenden, und ein Pod, der mehr nutzt, als er angefordert

hat, ist der erste Kandidat für die Verdrängung. So oder so müssen Requests auf Messungen beruhen; überhöhte Requests verschwenden Geld (der Node ist reserviert, aber untätig), und zu niedrige verursachen Noisy-Neighbour-Probleme und irreführendes Autoscaling.

Der HorizontalPodAutoscaler berechnet *gewünschte Replikas = Aufrunden(aktuelle Replikas × aktuelle Metrik ÷ Zielwert)* und misst die CPU-Nutzung als Prozentsatz des Requests. Er skaliert schnell hoch, wartet aber vor dem Herunterskalieren standardmäßig ein Stabilisierungsfenster von fünf Minuten ab, um ein Hin- und Herpendeln zu vermeiden. Passen neue Pods nicht auf bestehende Nodes, bleiben sie im Status *Pending*, und ein Node-Autoscaler (der Cluster Autoscaler oder bei manchen Anbietern Karpenter) fügt Maschinen hinzu, was Minuten statt Sekunden dauert. Pod-Autoscaling und Node-Autoscaling sind getrennte Schleifen mit getrennten Verzögerungen, und beide müssen für den Ansturm eingeplant werden.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: den Bestellservice für den Abendansturm dimensionieren

Jeder Pod des Bestellservice fordert 500m CPU und 512 MiB an, mit einem Speicherlimit von 512 MiB und ohne CPU-Limit. Die Nodes haben 4 vCPUs und 16 GiB, von denen nach Systemreservierungen etwa 3,8 vCPUs und 14,5 GiB zuteilbar sind (veranschaulichende Zahlen). Der HPA zielt auf 60 % CPU, mit 3 bis 12 Replikas.

Pods pro Node	CPU: 3,8 ÷ 0,5 = 7 Pods. Speicher: 14,5 GiB ÷ 0,5 GiB = 29 Pods. Die CPU ist die bindende Grenze: **7 Bestell-Pods pro Node**, wenn dort nichts anderes läuft.
18:00, der Ansturm beginnt	4 Replikas bei durchschnittlich 90 % ihres CPU-Requests. Gewünscht = Aufrunden(4 × 90 ÷ 60) = **6 Replikas**.
18:20, weiter steigend	6 Replikas bei 85 %. Gewünscht = Aufrunden(6 × 85 ÷ 60) = Aufrunden(8,5) = **9 Replikas**, die zusammen 4,5 vCPUs anfordern.
Node-Kapazität	Wenn andere Services die Nodes bereits füllen, bleiben drei der neuen Pods im Status Pending; der Node-Autoscaler fügt einen Node hinzu, der vielleicht zwei bis vier Minuten braucht, um beizutreten und das Image zu laden.
Antwort im Entwurf	`minReplicas` vor dem vorhersehbaren Ansturm erhöhen (geplantes Skalieren), etwas freie Node-Kapazität vorhalten und das Image klein halten, damit neue Pods schnell starten.
Folge eines falschen Requests	Bräuchte jeder Pod in Wirklichkeit nur 250m, bliebe der Request aber bei 500m, würde dieselbe Ansturmlast (225m pro Pod) als 45 % statt 90 % erscheinen: Der HPA würde erst skalieren, wenn die Nutzung 300m pro Pod überschreitet, und der Cluster würde doppelt so viel CPU reservieren wie nötig.

Autoscaling reagiert auf Last, die bereits angekommen ist; bei einer vorhersehbaren täglichen Spitze ist geplantes Vorskalieren billiger als verlorene Bestellungen.

Requests steuern sowohl den Scheduler als auch den Autoscaler; sie zu messen ist daher eine Architekturaufgabe ersten Ranges, keine Tuning-Kleinigkeit.

Liveness, Readiness und Startup Probes

Probes sind die Art, wie Kubernetes etwas über die Gesundheit erfährt. Eine **Readiness** Probe beantwortet, ob dieser Pod jetzt Verkehr erhalten soll; solange sie fehlschlägt, wird der Pod aus den Endpoints seines Service entfernt, läuft aber weiter. Eine **Liveness** Probe beantwortet,

ob der Prozess unrettbar hängt; schlägt sie wiederholt fehl, startet das kubelet den Container neu. Eine **Startup** Probe hält die beiden anderen zurück, bis eine langsam startende Anwendung ihre Initialisierung abgeschlossen hat.

Der folgenschwerste Fehler ist eine Liveness Probe, die Abhängigkeiten prüft. Fragt der Liveness-Endpoint des Bestellservice die Datenbank ab, lässt ein dreißigsekündiges Datenbank-Failover jeden Pod des Bestellservice gleichzeitig an der Liveness scheitern, Kubernetes startet alle neu, sie verbinden sich alle im selben Moment erneut, und aus einer kurzen Störung wird ein Ausfall. Liveness sollte nur prüfen, ob der Prozess selbst antworten kann; Readiness darf berücksichtigen, ob der Pod nützliche Arbeit leisten kann, doch selbst dort vermeiden es viele Teams, wegen einer geteilten Abhängigkeit alle Pods als nicht bereit zu markieren, und lassen Anfragen lieber mit einem Circuit Breaker schnell fehlschlagen (siehe Resilienzmuster).

Probes arbeiten mit dem kontrollierten Herunterfahren zusammen. Wird ein Pod während eines Rollouts oder beim Herunterskalieren gelöscht, entfernt Kubernetes ihn aus den Endpoints und sendet dem Prozess `SIGTERM`; der Prozess sollte keine neue Arbeit mehr annehmen, laufende Anfragen abschließen und sich innerhalb der Schonfrist (standardmäßig 30 Sekunden) beenden, bevor er abgebrochen wird. Weil sich das Entfernen aus den Endpoints asynchron ausbreitet, verhindert eine kurze `preStop`-Verzögerung, dass Anfragen an einen Pod gehen, der bereits zu stoppen begonnen hat. Abschnitt 10.3 greift das als Twelve-Factor-Prinzip der Disposability wieder auf.

Wann Kubernetes überdimensioniert ist

Kubernetes ist eine Allzweckplattform zum Bauen von Plattformen, und für diese Allgemeinheit verlangt es seinen Preis. Jemand muss Netzwerk, Ingress, Identität, Secrets, Policies, Observability und Mandantenfähigkeit entwerfen; mit drei Minor Releases pro Jahr Schritt halten, die jeweils etwa vierzehn Monate unterstützt werden, dazu mit den Add-ons, die davon abhängen; und neben den Anwendungen auch für den Cluster Rufbereitschaft leisten. Verwaltete Control Planes nehmen einen Teil dieser Arbeit ab, nicht den größten.

Für ein kleines Team mit einer Handvoll zustandsloser Webservices bietet ein PaaS oder eine Serverless-Container-Plattform (Google Cloud Run, Azure Container Apps, Amazon ECS mit Fargate und Ähnliche) die meisten Vorteile (Container, Autoscaling, Rolling Deployments, verwaltetes TLS) bei einem Bruchteil der Betriebslast. Kubernetes beginnt sich zu lohnen, wenn es viele Services und mehrere Teams gibt, ein Plattformteam, das es betreibt, Arbeitslasten mit ungewöhnlichen Anforderungen (zustandsbehaftete Systeme, GPUs, Batch-Jobs, eigene Controller), die Anforderung, dieselbe Plattform vor Ort und in mehreren Clouds zu betreiben, oder ein Ökosystem von Werkzeugen, das die Organisation nutzen will. Viele

Organisationen haben es eingeführt, weil es in Mode war, und dann Jahre damit verbracht, darauf im Grunde ein schlechteres PaaS zu bauen. Abschnitt 10.7 bespricht, wie Plattformteams diesem Schicksal entgehen (Platform Engineering).

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Sie müssen sich nicht jedes Kubernetes-Objekt merken. Behalten Sie drei Ideen: Sie beschreiben in Dateien, was Sie wollen, und das System stellt es her; kleine, wegwerfbare Einheiten (Pods) werden ersetzt statt repariert; und stabile Namen (Services) verbergen, dass die Kopien dahinter ständig wechseln. Alles andere sind Details, die auf diesen drei aufbauen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum sollte ein Produktions-Deployment ein Image per Digest statt per Tag referenzieren?

Ein Tag ist ein verschiebbares Etikett, das auf anderen Inhalt umgelenkt werden kann; ein Digest ist ein Hash des Inhalts und identifiziert immer genau das Image, das getestet wurde.

▼ Ein Node fällt aus. Welche Komponenten handeln in welcher Reihenfolge, um die dritte Replika wiederherzustellen?

Der Node-Controller markiert den Node als nicht bereit, und seine Pods werden verdrängt; der ReplicaSet-Controller sieht zu wenige Pods und legt einen an; der Scheduler weist ihn einem gesunden Node zu; das kubelet dieses Nodes startet ihn über die Runtime; sobald Readiness besteht, wird er in die Endpoints des Service aufgenommen.

▼ Was geschieht mit einem Container, der sein Speicherlimit überschreitet, und mit einem, der sein CPU-Limit überschreitet?

Beim Überschreiten des Speicherlimits wird der Container beendet (OOMKilled) und neu gestartet; beim Überschreiten des CPU-Limits wird er nur gedrosselt.

▼ Warum ist es gefährlich, in einer Liveness Probe die Datenbank zu prüfen?

Ein kurzes Datenbankproblem lässt jeden Pod gleichzeitig an der Liveness scheitern, sodass Kubernetes alle auf einmal neu startet und aus einer kurzen Störung einer Abhängigkeit ein vollständiger Ausfall wird.

▼ **Probieren Sie es aus: Jeder Pod fordert 750m CPU und 1 GiB an. Die Knoten bieten 7,6 vCPU und 28 GiB zuweisbar. Die Stoßzeit braucht 24 Pods. Wie viele Pods passen auf einen Knoten, und wie viele Knoten brauchen Sie, wenn der Verlust eines Knotens überstanden werden muss?**

Die CPU erlaubt 7,6 / 0,75 = 10,1, also 10 Pods; der Speicher erlaubt 28, die CPU ist also die bindende Grenze. Vierundzwanzig Pods brauchen 2,4, also 3 Nodes. Um den Ausfall eines Nodes zu überstehen, braucht es einen mehr: 4 Nodes. Requests steuern das Scheduling, halten Sie sie also nahe der realen Nutzung. Verteilen Sie die Pods über Zonen. Prüfen Sie, ob das Maximum des Autoscalers 24 Replikate erlaubt und ob der Cluster vor dem Ansturm Nodes ergänzen kann.

QUELLEN

- Kubernetes documentation: Cluster components: https://kubernetes.io/docs/concepts/overview/components/
- Kubernetes documentation: Resource management for pods and containers: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
- Kubernetes documentation: Configure liveness, readiness and startup probes: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
- Open Container Initiative: https://opencontainers.org/
- Kubernetes blog (2025): Ingress NGINX retirement: https://kubernetes.io/blog/2025/11/11/ingress-nginx-retirement/

Die Twelve-Factor App und cloud-native Prinzipien

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.8 Wie Software entsteht
- 9.2 Load Balancing
- 10.2 Container und Kubernetes

Begriffe von früher, die er verwendet: Container-Image, Continuous Delivery, Idempotent, Horizontale Skalierung (Scale-out), Observability

DIE IDEE IN KÜRZE

Denken Sie an eine Kette von Lieferküchen. Eine gute Rezeptkarte funktioniert in jeder Filiale: Sie führt jede Zutat auf, sagt nichts darüber, in welcher Filiale sie liegt, und jeder Koch kann sie mitten in der Schicht übernehmen, weil nichts Wichtiges nur im Kopf eines einzelnen Kochs steckt. Lieferungen kommen in beschrifteten Kisten, und wenn sich die Speisekarte ändert, wird eine neue Karte gedruckt und nummeriert, statt auf der alten herumzukritzeln. Die Twelve-Factor App ist diese Art Rezeptkarte für Software: Regeln, mit denen Maschinen dieselbe Anwendung starten, kopieren, verschieben, ersetzen und skalieren können, ohne dass jemand ihre Eigenheiten kennen muss.

WAS SIE WISSEN MÜSSEN

- The Twelve-Factor App (Adam Wiggins, Heroku, 2011, seit 2024 Open Source) beschreibt den Vertrag zwischen einer Anwendung und einer modernen Plattform und gilt für Monolithen wie für Microservices.
- Config (III) und Backing Services (IV) halten alles, was sich zwischen Deployments unterscheidet, außerhalb des Artefakts, sodass ein Artefakt in jeder Umgebung läuft; Secrets gehören heute in einen Secrets Manager oder in eine Workload Identity statt in einfache Umgebungsvariablen.
- Build, Release, Run (V) macht Releases unveränderlich und nummeriert, sodass ein Rollback bedeutet, ein früheres Release auszuführen, und niemand einen laufenden Server bearbeitet.
- Zustandslose Prozesse (VI), Nebenläufigkeit nach Prozesstyp (VIII) und Disposability (IX) machen horizontale Skalierung, Rolling Updates und das Ersetzen ausgefallener Instanzen sicher.

- Dev/Prod-Parität (X), Logs als Streams (XI) und Admin-Prozesse als separate einmalige Prozesse (XII) verringern Überraschungen in der Produktion; Migrationen müssen mit altem und neuem Code kompatibel sein.
- Über die zwölf Faktoren hinaus werden cloud-native Anwendungen auf Disposability, Unveränderlichkeit, Observability und Sicherheit ausgelegt, und Hoffmans Erweiterung von 2016 ergänzt API First, Telemetrie sowie Authentifizierung und Autorisierung.
- Ein Audit anhand der Faktoren sollte Lücken nach Risiko ordnen: zuerst offengelegte Secrets, dann alles, was die Zielplattform blockiert, etwa lokaler Zustand, dann Zuverlässigkeit und Komfort.

Woher die zwölf Faktoren kommen

2011 veröffentlichte Adam Wiggins, Mitgründer von Heroku, *The Twelve-Factor App*, ein kurzes Webdokument, das bündelte, was das Heroku-Team aus der Beobachtung sehr vieler Anwendungen gelernt hatte, die auf seiner Plattform gebaut, deployt und skaliert wurden. Die Methodik der Twelve-Factor App beschreibt Anwendungen, die sich aus deklarativen, automatisierten Schritten selbst einrichten, einen sauberen Vertrag mit der Plattform darunter haben, sodass sie überall laufen können, Entwicklung und Produktion nah genug beieinander halten, um kontinuierliches Deployment zu ermöglichen, und ohne Neuentwurf horizontal skalieren können.

Das Dokument ist älter als Docker (2013) und Kubernetes (2014), beschreibt aber fast genau den Vertrag, den diese Plattformen später durchgesetzt haben: Eine Anwendung ist eine Menge zustandsloser Prozesse, gebaut zu einem unveränderlichen Artefakt, von außen konfiguriert, die Logs in einen Stream schreibt und mit allem anderen über das Netzwerk spricht. Deshalb ist es bis heute das gemeinsame Vokabular für den Entwurf cloud-nativer Anwendungen. Im November 2024 gab Heroku bekannt, dass die Methodik nun Open Source ist und von einer Community aus Beitragenden mehrerer Unternehmen gepflegt wird, mit dem Ziel, sie an die heutige Praxis anzupassen; die ursprünglichen zwölf Faktoren bleiben der Bezugspunkt.

Die Faktoren betreffen die Beziehung der Anwendung zu ihrer Laufzeitplattform, nicht ihre interne Architektur. Ein modularer Monolith kann vollständig Twelve-Factor-konform sein; eine Menge von Microservices kann jeden Faktor verletzen. Sie ergänzen die Anliegen Deploybarkeit und Betreibbarkeit aus Kapitel 2 und die Prinzipien zu Skalierung und Zustandslosigkeit aus Kapitel 9.

Die zwölf Faktoren, mit heutigen Einschränkungen

Faktor	Was sie bedeutet	Warum sie zählt	Heutige Einschränkung
I. Codebasis	Eine Codebasis in der Versionsverwaltung pro App, deployt in viele Umgebungen.	Jede Umgebung führt eine Version desselben Codes aus; keine Forks pro Kunde oder pro Umgebung.	Ein Monorepo mit mehreren Apps ist in Ordnung, wenn jede App eine klare Grenze hat; das Antipattern sind kopierte und abgewandelte Forks.
II. Abhängigkeiten	Jede Abhängigkeit explizit deklarieren und die App vom Hostsystem isolieren.	Builds sind auf jeder Maschine reproduzierbar; nichts hängt von einem Paket ab, das jemand von Hand installiert hat.	Lock-Dateien und Images machen das zum Standard; gepinnte Abhängigkeiten brauchen trotzdem regelmäßige Updates und Schwachstellenscans.
III. Konfiguration	Alles, was sich zwischen Deployments unterscheidet (URLs, Zugangsdaten, Feature-Einstellungen), aus dem Code heraushalten, in der Umgebung.	Ein Artefakt läuft in jeder Umgebung; Secrets gelangen nie ins Repository.	Umgebungsvariablen sickern in Logs, Absturzberichte und Kindprozesse durch; Secrets liest man besser aus einem Secrets Manager oder aus eingehängten Dateien.
IV. Backing Services	Datenbanken, Queues, Caches und externe APIs als angebundene Ressourcen behandeln, die über Konfiguration gefunden werden.	Eine lokale Datenbank lässt sich durch eine verwaltete ersetzen, eine ausgefallene durch eine wiederhergestellte Kopie, indem man eine URL ändert.	Ein Austausch ist selten nahtlos, wenn sich die Semantik unterscheidet; Unterschiede hinter Ports und Adaptern isolieren.
V. Build, Release, Run	Das Bauen eines Artefakts, seine Kombination mit Konfiguration zu einem Release und seine Ausführung trennen.	Releases sind unveränderlich und nummeriert; jedes Release kann erneut ausgeführt werden; niemand bearbeitet Code auf einem Server.	Heute: einmal in der CI bauen, das Image per Digest speichern, Releases in Git oder im Deployment-Tool festhalten.
VI. Prozesse	Die App als einen oder mehrere zustandslose Prozesse ausführen, die nichts teilen; Zustand in Backing Services halten.	Jede Kopie kann jede Anfrage bedienen, also lassen sich Kopien frei hinzufügen, entfernen und ersetzen.	Sticky Sessions verdecken das Problem, statt es zu lösen; die lokale Festplatte ist nur ein Arbeitsbereich für Temporäres.

Faktor	Was sie bedeutet	Warum sie zählt	Heutige Einschränkung
VII. Port-Bindung	Die App ist eigenständig und stellt ihren Dienst bereit, indem sie auf einem Port lauscht.	Die Plattform leitet Verkehr an sie, ohne sie in einen separaten Webserver einzubetten.	Heute universell; Queue-Consumer haben keinen Port, und Service Meshes können davorgeschaltet sein.
VIII. Nebenläufigkeit	Horizontal skalieren, indem mehr Prozesse laufen, mit verschiedenen Prozesstypen für verschiedene Arbeit (Web, Worker).	Jede Art von Arbeit skaliert unabhängig nach ihrer eigenen Last.	Autoscaler erledigen das automatisch, auch anhand der Queue-Länge; Threads innerhalb eines Prozesses spielen weiterhin eine Rolle.
IX. Disposability	Schnell starten, bei SIGTERM geordnet herunterfahren, einen plötzlichen Tod überstehen.	Schnelle Skalierung, sichere Rolling Deployments und Robustheit gegenüber Abstürzen.	Langsam startende Laufzeitumgebungen brauchen Startup Probes oder Techniken für schnelleren Start; Hintergrundjobs müssen idempotent und fortsetzbar sein.
X. Dev/Prod-Parität	Entwicklung, Staging und Produktion zeitlich, personell und bei den Tools so ähnlich wie möglich halten.	Weniger Überraschungen, die erst in der Produktion auftreten; kontinuierliches Deployment wird sicher.	Volle Parität ist zu teuer; vorrangig dieselben Arten und Versionen von Backing Services verwenden, lokal in Containern.
XI. Logs	Logs als ungepufferten Event-Stream auf die Standardausgabe schreiben; die Plattform leitet sie weiter.	Die App verwaltet keine Dateien; Logs aller Kopien werden zentral gesammelt.	Strukturierte Logs plus Metriken und Traces verwenden (zum Beispiel über OpenTelemetry); Log-Volumen und Kosten im Blick behalten.
XII. Admin-Prozesse	Einmalige Aufgaben (Migrationen, Skripte) als separate Prozesse mit demselben Code und derselben Konfiguration ausführen.	Admin-Aufgaben laufen gegen genau das Release in der Produktion, nicht gegen eine veraltete Kopie.	Als Pipeline-Schritte oder Kubernetes-Jobs ausführen; nie jedes Replikat beim Start ohne Lock Migrationen ausführen lassen.

Konfiguration, Secrets und Backing Services

Für den Konfigurationsfaktor gibt es einen einfachen Test: Könnte die Codebasis heute veröffentlicht werden, ohne eine einzige Zugangsinformation offenzulegen? Wenn nicht, ist Konfiguration in den Code gesickert. Die Trennung von Konfiguration und Code lässt ein

Artefakt unverändert durch jede Umgebung wandern; das ist die Grundlage des Build-once-Prinzips in Abschnitt 10.5 (CI/CD und Deployment). Zur Konfiguration gehören Datenbank- und Service-URLs, Zugangsdaten und umgebungsspezifische Einstellungen; nicht dazu gehört die interne Verdrahtung, die überall gleich ist, etwa welche Klasse welche Route behandelt: Sie gehört in den Code.

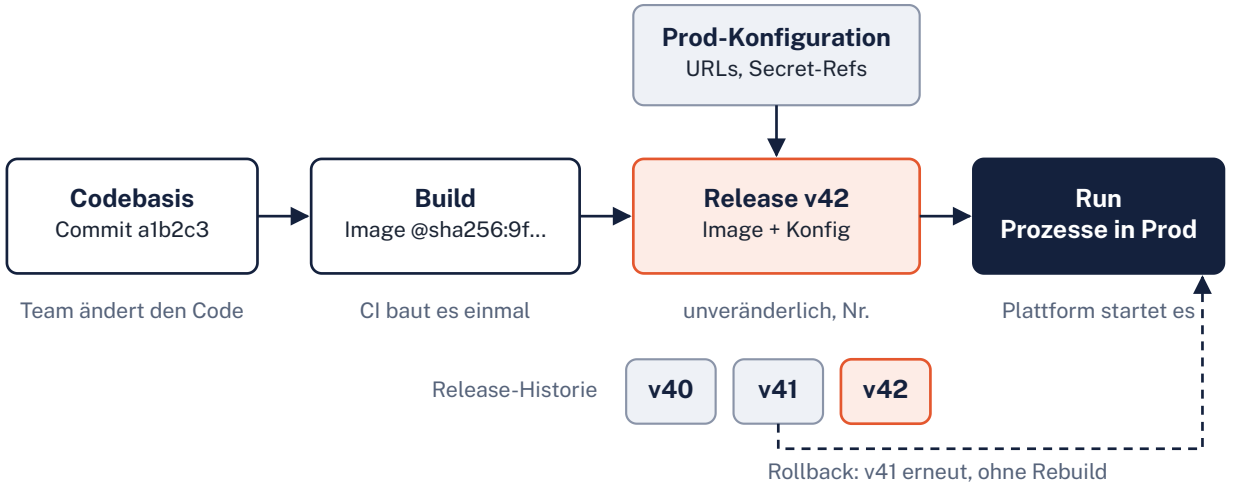
Die ursprüngliche Empfehlung, Umgebungsvariablen zu verwenden, ist ungleichmäßig gealtert. Sie sind universell und sprachneutral, und Kubernetes, PaaS-Plattformen und Funktionen unterstützen sie alle. Aber die Umgebung eines Prozesses ist für Kindprozesse sichtbar, landet oft in Absturzberichten und Debugging-Seiten und lässt sich ohne Neustart nicht rotieren. Die heutige Praxis für Secrets ist, sie beim Start aus einem Secrets Manager abzurufen (AWS Secrets Manager, Azure Key Vault, Google Secret Manager, HashiCorp Vault oder OpenBao) oder sie als Dateien einzuhängen, die die Plattform aktualisieren kann, und noch besser, langlebige Secrets durch Workload Identity zu ersetzen, bei der die Plattform jedem Service automatisch kurzlebige Zugangsdaten gibt. Die Absicht des Faktors, Konfiguration außerhalb des Artefakts zu halten, ist unverändert.

Einen Backing Service als angebundene Ressource zu behandeln heißt, dass der Code die Bestelldatenbank, den Zahlungsanbieter und den Message Broker auf dieselbe Weise erreicht: über eine URL und Zugangsdaten, die beim Deployment bereitgestellt werden. Das macht Umgebungen kombinierbar und die Wiederherstellung schneller. Es macht Backing Services nicht wie von Zauberhand austauschbar; wer einen Message Broker durch einen anderen ersetzt, ändert die Zustellsemantik. Der hexagonale Stil aus Kapitel 5 ergänzt den Faktor: Die Konfiguration sagt, wo die Ressource ist, und ein Adapter hinter einem Port verbirgt, wie sie sich verhält.

Build, Release, Run

Der Faktor Build, Release, Run trennt drei Stufen, die oft verschwimmen. Die *Build*-Stufe macht aus einem Commit ein ausführbares Artefakt, heute meist ein Container-Image, das über seinen Digest identifiziert wird. Die *Release*-Stufe kombiniert dieses Artefakt mit der Konfiguration einer Umgebung und gibt dem Ergebnis eine Identität, etwa Release v42 des Bestellservice in der Produktion. Die *Run*-Stufe startet Prozesse aus einem Release. Releases sind unveränderlich: Eine Änderung an Code oder Konfiguration erzeugt ein neues Release, nie eine Bearbeitung eines bestehenden.

Einmal bauen, pro Umgebung releasen, überall ausführen



Weil Releases unveränderlich und nummeriert sind, bedeutet ein Rollback, ein früheres Release erneut auszuführen, nicht etwas neu zu bauen oder zu bearbeiten.

Diese Trennung macht Deployments langweilig. Niemand bearbeitet Dateien auf einem Server, also driften Server nicht auseinander; dasselbe Image, das im Staging getestet wurde, läuft in der Produktion, also hat ein Testergebnis Aussagekraft; und ein Rollback ist ein schneller, geübter Vorgang statt eines Notfall-Rebuilds. Sie ist auch ein Audit-Trail: Für jeden Zeitpunkt lässt sich genau sagen, welcher Code und welche Konfiguration liefen. Kubernetes führt eine Revisionshistorie der ReplicaSets jedes Deployments, PaaS-Plattformen nummerieren Releases, und GitOps (Abschnitt 10.4) macht die Git-Historie des Release-Repositorys zum Nachweis.

Prozesse, Nebenläufigkeit und Disposability

Zustandslose Shared-Nothing-Prozesse sind die Voraussetzung für fast alles andere in diesem Kapitel: horizontale Skalierung, Rolling Deployments, Autoscaling, das Ersetzen ausgefallener Knoten und die Verteilung über Zonen. Alles, was ein Prozess nur in seinem Speicher oder auf seiner lokalen Festplatte hält (die Session eines Nutzers, eine hochgeladene Datei, ein halb erstellter Bericht), geht verloren, wenn dieser Prozess ersetzt wird, und ist für die anderen Kopien unsichtbar. Kapitel 9 behandelt den Umgang mit Sessions ausführlich (Load Balancing, Zustandslosigkeit und Sessions). Zustand verschwindet nicht; er wandert in Backing Services, die dafür gemacht sind, ihn zu halten.

Der Nebenläufigkeitsfaktor ergänzt, dass verschiedene Arten von Arbeit als verschiedene Prozesstypen laufen sollten, die getrennt skalieren. In der Essenslieferplattform skaliert der Webprozess, der die Kunden-App bedient, mit der Anfragerate, während ein Worker-Prozess,

der Bestellbestätigungen verschickt, mit der Queue-Länge skaliert und ein Consumer für Kurierstandorte mit dem Event-Durchsatz. Wer sie in einem Prozesstyp mischt, skaliert alle für den Bedarf eines einzigen.

Disposability verlangt, dass Prozesse in Sekunden starten und sauber stoppen. Bei `SIGTERM` nimmt ein Webprozess keine Verbindungen mehr an und beendet laufende Anfragen; ein Worker gibt seine aktuelle Nachricht an die Queue zurück oder schließt sie innerhalb der Karenzzeit ab. Weil Prozesse auch ohne Vorwarnung sterben können (ein Knotenausfall, ein Abbruch wegen Speichermangels), muss Arbeit gefahrlos wiederholbar sein: idempotente Nachrichtenverarbeitung wie in der ereignisgesteuerten Architektur und Jobs, die an Checkpoints fortsetzen. George Candea und Armando Fox trieben die Idee 2003 mit *Crash-only Software* auf die Spitze, die nur eine Art zu stoppen kennt (den Absturz) und eine Art zu starten (die Wiederherstellung), sodass der Wiederherstellungspfad ständig geübt statt erst während eines Incidents entdeckt wird.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: ein Worker, der stoppen soll

Die Idee, die hier die meiste Sorgfalt verlangt, ist Disposability: Ein Prozess muss damit rechnen, jederzeit gestoppt zu werden, höflich oder nicht, und dabei nichts zu verlieren. Begleiten Sie einen Worker-Prozess durch beide Arten des Stoppens.

- Die Essenslieferplattform betreibt einen Worker-Prozess, der Nachrichten aus einer Queue nimmt und Bestellbestätigungen verschickt. Bei einem routinemäßigen Deployment wird jede alte Kopie des Workers durch eine neue ersetzt, also muss jede alte Kopie stoppen.
- Der höfliche Weg ist ein Signal namens SIGTERM, das „bitte zum Ende kommen und beenden“ bedeutet, wie in Abschnitt 10.2 beschrieben. Die Plattform wartet eine Karenzzeit ab, bevor sie den Prozess zwangsweise beendet.
- Ein Worker mit Disposability reagiert sofort. Er nimmt keine neuen Nachrichten mehr an, schließt entweder die Bestätigung ab, die er gerade verschickt, oder gibt diese Nachricht an die Queue zurück, damit eine andere Kopie sie übernimmt, und beendet sich. Nichts geht verloren.
- Ein Prozess kann aber auch ohne Vorwarnung sterben: Seine Maschine fällt aus, oder er wird beendet, weil er zu viel Speicher verbraucht. Es bleibt keine Gelegenheit zum Aufräumen.
- Angenommen, der Worker stirbt, kurz nachdem er eine Bestätigung verschickt hat, aber bevor er der Queue gemeldet hat, dass er fertig ist. Die Nachricht wird erneut angeboten, und eine andere Kopie verarbeitet sie noch einmal. Deshalb muss die Verarbeitung idempotent sein, wie es in Kapitel 5 hieß: Sie zweimal auszuführen hat dieselbe Wirkung wie einmal. Das funktioniert meist, indem jede Nachricht einen eindeutigen Schlüssel erhält, der mit der Bestätigung weitergegeben wird, sodass eine Wiederholung erkannt und übersprungen wird.
- **Die Lehre:** Behandeln Sie Stoppen und Abstürze als alltägliche Ereignisse. Ein Prozess, der schnell startet, auf Anforderung sauber stoppt und nichts verliert, wenn er ohne Vorwarnung stirbt, lässt sich frei ersetzen, verschieben und skalieren.

Dev/Prod-Parität, Logs und Admin-Prozesse

Der Originaltext benennt drei Lücken zwischen Entwicklung und Produktion: Zeit (Code, der Wochen vor seinem Deployment geschrieben wird), Personal (Entwickler schreiben ihn, der Betrieb deployt ihn) und Tools (SQLite auf dem Laptop, PostgreSQL in der Produktion). Continuous Delivery schließt die erste, Teams, die betreiben, was sie bauen, schließen die zweite, und Container schließen einen großen Teil der dritten: Eine Entwicklerin kann dieselbe PostgreSQL-Version, denselben Message Broker und denselben Cache lokal ausführen, und

Tools wie Testcontainers starten echte Abhängigkeiten in Integrationstests. Perfekte Parität ist weder bezahlbar noch nötig; am wichtigsten ist, dieselben Arten und Versionen von Backing Services zu verwenden, denn dort liegen die überraschenden Unterschiede.

Logs als Event-Streams bedeutet, dass die Anwendung jedes Ereignis auf die Standardausgabe schreibt und Speicherung, Rotation und Weiterleitung der Plattform überlässt. Das ist weiterhin richtig, mit zwei heutigen Ergänzungen: Logs sollten strukturiert sein (zum Beispiel JSON mit einheitlichen Feldnamen und einer Anfrage- oder Trace-ID), und Logs sind eines von drei Signalen, neben Metriken und Traces, die ein System beobachtbar machen (Observability). Das Log-Volumen ist außerdem ein echter Kostentreiber; Abschnitt 10.8 kommt darauf zurück.

Admin-Prozesse wie Datenbankmigrationen, Datenreparaturen und einmalige Berichte sollten mit demselben Release (Code und Konfiguration) wie die Anwendung laufen, als separate Prozesse. In Kubernetes ist das typischerweise ein Job, den die Pipeline ausführt, bevor die neue Version ausgerollt wird. Migrationen brauchen besondere Sorgfalt: Sie müssen sowohl mit der alten als auch mit der neuen Codeversion kompatibel sein, die nebeneinander laufen; das ist die Disziplin Expand and Contract der Schemaevolution.

Über die zwölf Faktoren hinaus

Die zwölf Faktoren waren nie als vollständig gedacht, und spätere Autoren haben sie erweitert. Kevin Hoffmans *Beyond the Twelve-Factor App* (O'Reilly, 2016) erweiterte die Liste auf fünfzehn und ergänzte *API First* (den Vertrag vor der Implementierung entwerfen), *Telemetrie* (Metriken, Health und Tracing einbauen) sowie *Authentifizierung und Autorisierung* (Sicherheit als Entwurfsanliegen von Anfang an). Die heutige Praxis ergänzt vier Themen, die jede cloud-native Anwendung erfüllen sollte.

Disposability, ernst genommen. Darauf auslegen, jederzeit beendet zu werden: idempotente Operationen, kurze Anfrage-Timeouts, begrenzte Startzeit, geordnetes Leerlaufen (Drain) und kein einzelner Prozess, dessen Verlust ins Gewicht fällt. **Unveränderlichkeit.** Nie einen laufenden Container oder Server patchen; ein neues Image bauen und ihn ersetzen (Abschnitt 10.4 zu Immutable Infrastructure). So lässt sich jede laufende Instanz auf einen Commit zurückführen, und Drift entfällt. **Observability.** Strukturierte Logs, Metriken und Traces über einen Standard wie OpenTelemetry ausgeben, Health-Endpoints bereitstellen, die den eigenen Zustand des Prozesses widerspiegeln, und den Trace-Kontext über Aufrufe hinweg weitergeben, damit der Betrieb Fragen beantworten kann, die er nicht vorhergesehen hat. **Sicherheit.** Wo möglich als Nicht-Root-Benutzer mit schreibgeschütztem Dateisystem laufen; Zugangsdaten über Workload Identity statt über langlebige Schlüssel beziehen; für jeden Backing Service das Prinzip der geringsten Rechte anwenden; und aus minimalen, gescannten, signierten Images bauen (Sicherheit der Software-Lieferkette).

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Audit des Restaurant-Onboarding-Service

Der Restaurant-Onboarding-Service der Essenslieferplattform ist eine fünf Jahre alte Webanwendung auf zwei von Hand aufgesetzten virtuellen Maschinen hinter einem Load Balancer mit Sticky Sessions. Das Team möchte sie auf die Container-Plattform des Unternehmens umziehen. Ein Audit anhand der Faktoren findet neun Probleme; jedes erhält ein Risiko, eine Korrektur und eine Priorität.

1. Passwort der Produktionsdatenbank in einer Properties-Datei im Repository (III)	Risiko: Jeder mit Lesezugriff auf den Code, auch ehemalige externe Auftragnehmer, hat die Zugangsdaten. Korrektur: das Passwort sofort rotieren, in den Secrets Manager verschieben, beim Deployment injizieren. **Priorität 1**.
2. Hochgeladene Speisekarten und Logos auf der lokalen Festplatte der VM (VI, IV)	Risiko: Dateien verschwinden, wenn eine Anfrage auf der anderen VM landet oder eine VM ersetzt wird; blockiert jeden Umzug auf entsorgbare Container. Korrektur: Object Storage, mit einer einmaligen Kopie der vorhandenen Dateien. **Priorität 1**.
3. Sessions im Prozessspeicher, Sticky Load Balancing (VI)	Risiko: Jedes Deployment meldet das Restaurantpersonal mitten im Formular ab. Korrektur: signierte Session-Tokens oder ein gemeinsamer Session-Speicher. **Priorität 2**.
4. Deployment per SSH, `git pull` und Neustart auf jeder VM (V, II)	Risiko: Die beiden VMs führen nach einem fehlgeschlagenen Pull unterschiedlichen Code aus; kein Rollback. Korrektur: Die CI baut ein Image pro Commit; nummerierte Releases. **Priorität 2**.
5. Migrationen laufen beim Start automatisch auf beiden Instanzen (XII)	Risiko: Zwei Instanzen laufen bei derselben Migration in eine Race Condition. Korrektur: ein separater Migrationsschritt in der Pipeline, mit Änderungen nach Expand and Contract. **Priorität 2** (Priorität 1, wenn es schon eine Race Condition gab).
6. Bei Deployments mit `kill -9` beendet (IX)	Risiko: halb gespeicherte Onboarding-Formulare. Korrektur: SIGTERM behandeln, Anfragen auslaufen lassen, Readiness und eine Pre-Stop-Verzögerung ergänzen. **Priorität 2**.
7. Logs in rotierende Dateien auf jeder VM geschrieben (XI)	Risiko: Logs gehen mit der VM verloren; schwer über Instanzen hinweg zu durchsuchen. Korrektur: strukturiertes JSON auf die Standardausgabe, zentral gesammelt. **Priorität 3**, günstig zusammen mit Punkt 4 umzusetzen.
8. Nächtlicher Auszahlungsbericht als Cron-Eintrag nur auf VM 1 (XII, VIII)	Risiko: Der Job hört stillschweigend auf, wenn VM 1 neu aufgebaut wird, oder läuft doppelt, wenn er kopiert wird. Korrektur: ein geplanter Job auf der Plattform, idempotent gemacht. **Priorität 3**.
9. Entwickler nutzen SQLite, die Produktion PostgreSQL (X)	Risiko: SQL, das lokal funktioniert, schlägt in der Produktion fehl. Korrektur: PostgreSQL in einem Container für Entwicklung und Tests. **Priorität 3**.

Die Prioritäten folgen zuerst dem Risiko (offengelegte Zugangsdaten sind ein laufender Incident), dann dem, was die Zielplattform blockiert (lokaler Zustand), dann der Zuverlässigkeit, dann dem Komfort.

Das Team muss nicht alles vor dem Umzug beheben: Die Punkte 1 und 2 sind Voraussetzungen; die Punkte 3 bis 6 lassen sich während des Umzugs erledigen; die Punkte 7 bis 9 können folgen.

Ein solches Audit ist auch gutes Material für einen Architecture Decision Record: Es erklärt, warum jede Änderung vorgenommen wird, in welcher Reihenfolge und was bewusst zurückgestellt wurde.

ACHTUNG

Häufige Missverständnisse über die zwölf Faktoren

- Zustandslos heißt nicht, dass das System keinen Zustand hat; es heißt, dass Prozesse ihn nicht halten. Den Zustand in einen einzelnen, nicht replizierten Cache zu verschieben, verschiebt nur das Problem.
- Twelve-Factor ist keine Definition von Microservices. Es gilt ebenso für einen gut gebauten Monolithen, und eine Menge von Services kann es verletzen.
- Secrets in Umgebungsvariablen erfüllen den Wortlaut von Faktor III, aber nicht die heutige Sicherheitspraxis; besser sind ein Secrets Manager und Workload Identity.
- Dev/Prod-Parität bedeutet keine vollständige Kopie der Produktion für jede Entwicklerin und jeden Entwickler; sie bedeutet dieselben Arten und Versionen von Abhängigkeiten und kurze zeitliche und personelle Abstände.
- Port-Bindung und Logs auf der Standardausgabe sind leichte Erfolge; Teams setzen manchmal genau diese um und erklären sich für konform, während sie weiterhin Dateien auf der lokalen Festplatte ablegen.

Häufige Fragen

Muss jede Anwendung alle zwölf Faktoren erfüllen?
Nein. Sie sind eine Checkliste von Eigenschaften, die Deployment, Skalierung und Wiederherstellung auf einer modernen Plattform günstig machen. Behandeln Sie jede Lücke als Risiko mit Kosten, und beheben Sie zuerst die, die Ihre Plattform blockieren oder die Sicherheit gefährden, wie im Audit oben.

Gilt die Methodik für Serverless-Funktionen?
Weitgehend, und die Plattform erzwingt das meiste davon: Funktionen sind zustandslos, von außen konfiguriert, entsorgbar und werden von der Plattform geloggt. Aufmerksamkeit brauchen weiterhin Konfiguration und Secrets, Backing Services, Idempotenz und Dev/Prod-Parität, da die lokale Emulation der Event-Quellen eines Anbieters unvollkommen ist.

Ist Twelve-Factor noch relevant, seit es Kubernetes gibt?
Ja, denn Kubernetes setzt es voraus. Ein Pod, der Sessions im Speicher hält, Dateien lokal schreibt oder zwei Minuten zum Starten braucht, kämpft gegen die Rolling Updates, das Autoscaling und das Rescheduling der Plattform. Die Faktoren beschreiben, was eine Anwendung tun muss, damit die Automatisierung der Plattform hilft statt schadet.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Welcher einfache Test zeigt, ob Konfiguration in den Code gesickert ist?**

Fragen Sie, ob die Codebasis heute veröffentlicht werden könnte, ohne irgendwelche Zugangsdaten offenzulegen. Wenn nicht, steckt Konfiguration (zumindest die Secrets) im Code.

▼ **Warum macht Build, Release, Run ein Rollback schnell?**

Jedes Release ist eine unveränderliche Kombination aus einem gebauten Artefakt und Konfiguration mit eigener Identität, also bedeutet ein Rollback, das vorherige Release erneut auszuführen, ohne etwas neu zu bauen oder zu bearbeiten.

▼ **Ein Service speichert hochgeladene Dateien auf der lokalen Festplatte seines Servers. Welchen Faktor verletzt das, und was geht kaputt?**

Prozesse (VI) und wohl auch Backing Services (IV): Dateien fehlen, wenn Anfragen eine andere Kopie erreichen, und gehen verloren, wenn die Instanz ersetzt wird; das blockiert horizontale Skalierung und entsorgbare Container.

▼ **Warum sollten Migrationen nicht automatisch in jedem Replikat beim Start laufen?**

Mehrere Replikate können dieselbe Migration gleichzeitig ausführen und in eine Race Condition laufen, und der Start wird langsam und fragil. Migrationen sollten einmal laufen, als separater Admin-Schritt in der Pipeline, mit demselben Release.

▼ **Probieren Sie es aus: Bei der Prüfung eines Services finden Sie: ein Produktionspasswort im Repository; hochgeladene Speisekarten auf der lokalen Festplatte; Logs, die in eine Datei geschrieben werden; einmalige Admin-Skripte, die nach Anmeldung auf einem Server ausgeführt werden. Ordnen Sie diese Punkte nach Priorität für den Umzug auf eine Container-Plattform.**

Zuerst das Passwort: Es liegt jetzt offen, rotieren Sie es also, und verschieben Sie es in einen Secrets Manager oder zu Workload Identity. Zweitens lokale Dateien, weil sie den Umzug blockieren: Container sind Wegwerfware, Uploads gehen also als Backing Service in den Objektspeicher. Drittens einmalige Skripte: Führen Sie sie als separate Admin-Prozesse aus demselben Release aus, damit sie wiederholbar und auditiert sind. Viertens Logs: Schreiben Sie sie als Stream nach stdout. Das Risiko kommt zuerst, dann die Blocker, dann der Komfort.

QUELLEN

- The Twelve-Factor App: https://12factor.net/
- The Twelve-Factor App: III. Config: https://12factor.net/config
- Twelve-Factor App methodology is now open source (2024): https://www.12factor.net/blog/open-source-announcement
- Kubernetes documentation: Configure liveness, readiness and startup probes: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

Infrastructure as Code und Immutable Infrastructure

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.7 Server und die Cloud
- 0.8 Wie Software entsteht
- 10.2 Container und Kubernetes

Begriffe von früher, die er verwendet: Gewünschter Zustand (Desired State), Reconciliation Loop, Pull Request, Blast Radius

DIE IDEE IN KÜRZE

Stellen Sie sich vor, Sie richten ein Hotel mit zweihundert identischen Zimmern ein. Sie könnten jedem Dekorateur sagen, was zu tun ist, und hoffen, dass er es sich merkt, oder Sie könnten eine präzise Spezifikation für ein Zimmer schreiben, sie in einem Ordner aufbewahren und jede Änderung schriftlich genehmigen lassen. Mit der Spezifikation lässt sich jedes Zimmer daran prüfen, ein beschädigtes Zimmer exakt wiederherstellen und ein neuer Flügel in Tagen ausstatten. Infrastructure as Code ist diese Spezifikation für die Computer, Netzwerke und Datenbanken, auf denen ein System läuft: aufgeschrieben, geprüft, versioniert und von Maschinen angewendet statt aus dem Gedächtnis.

WAS SIE WISSEN MÜSSEN

- Infrastructure as Code definiert Infrastruktur in versionierten Dateien, die von Tools angewendet werden, und macht Umgebungen reproduzierbar, überprüfbar, auditierbar und schnell erstellbar.
- Deklarative Tools beschreiben den gewünschten Endzustand und berechnen die Schritte, wodurch wiederholte Läufe idempotent werden; imperative Skripte listen Schritte auf und lassen sich nur schwer sicher erneut ausführen.
- HashiCorp stellte Terraform im August 2023 auf die Business Source License um; der Fork OpenTofu folgte unter der Linux Foundation (stabil seit Januar 2024, seit 2025 ein CNCF-Sandbox-Projekt); CloudFormation, Bicep, Pulumi und Crossplane sind die wichtigsten Alternativen.
- Der State ordnet Code realen Ressourcen zu; er muss remote, gesperrt, versioniert und zugriffsbeschränkt sein und aufgeteilt werden, um den Blast Radius zu begrenzen, und Drift muss durch geplante Plan-Läufe erkannt und dann in Code übernommen oder

rückgängig gemacht werden.

- Versionierte Module mit kleinen Schnittstellen, über Umgebungen hinweg mit unterschiedlichen Eingaben wiederverwendet, verhindern, dass per Copy-and-paste erstellte Umgebungen auseinanderlaufen.
- Immutable Infrastructure ersetzt Server und Container aus neuen Images, statt sie an Ort und Stelle zu ändern; Konfigurationsmanagement bleibt nützlich für langlebige Landschaften und zum Bauen von Images.
- GitOps hält den gewünschten Zustand in Git und lässt Agenten im Cluster (Argo CD, Flux) ihn abrufen und fortlaufend abgleichen; Policy as Code und mehrstufige Tests machen Änderungen sicher.

Was Infrastructure as Code ist

Infrastructure as Code (IaC) bedeutet, Netzwerke, virtuelle Maschinen, Cluster, Datenbanken, Queues, DNS-Einträge, Berechtigungen und jedes andere Stück Infrastruktur in Textdateien unter Versionsverwaltung zu definieren und Tools die realen Ressourcen aus diesen Dateien erstellen und ändern zu lassen. Statt dass sich jemand durch eine Cloud-Konsole klickt, ist eine Änderung ein Commit: in einem Pull Request geprüft, in einer Pipeline getestet, automatisch angewendet und für immer festgehalten.

Die Idee ist älter als die Public Cloud. Mark Burgess' CFEngine (1993) verwaltete Unix-Maschinen anhand deklarativer Richtlinien; Puppet (2005), Chef (2009) und Ansible (2012) brachten das Konfigurationsmanagement in den Mainstream. Die Cloud machte es unverzichtbar, weil die Infrastruktur selbst über APIs verfügbar wurde: AWS CloudFormation (2011) und HashiCorps Terraform (2014) beschreiben ganze Umgebungen, nicht nur die Software auf Servern. Kief Morris' Buch *Infrastructure as Code*, inzwischen in der dritten Auflage (2025), ist die Standarddarstellung der Praxis und ihrer Prinzipien.

Die Vorteile sind die des Software-Engineerings, angewandt auf Infrastruktur. Umgebungen werden reproduzierbar: Staging lässt sich aus denselben Definitionen bauen wie die Produktion, und eine Region lässt sich nach einer Katastrophe aus Code statt aus dem Gedächtnis neu aufbauen. Änderungen werden überprüfbar und auditierbar, was Aufsichtsbehörden und Sicherheitsteams schätzen. Wissen steckt nicht mehr nur im Kopf eines einzelnen Engineers. Und das Tempo steigt, weil das Erstellen einer Umgebung für einen neuen Service der Essenslieferplattform zu einem Pull Request wird statt zu einem Ticket, das zwei Wochen wartet.

Deklarativ oder imperativ

Ein *imperativer* Ansatz listet die Schritte auf: ein Netzwerk erstellen, dann ein Subnetz, dann eine Datenbank, dann diesen Port öffnen. Wer das Skript zweimal ausführt, versucht alles zweimal zu erstellen. Ein *deklarativer* Ansatz beschreibt den Endzustand, den gewünschten Zustand aus Abschnitt 10.2: ein Netzwerk mit diesen Subnetzen, eine Datenbank dieser Größe, eine Firewall, die diesen Port erlaubt. Das Tool vergleicht die Beschreibung mit dem, was existiert, und ermittelt die Schritte, sodass ein zweites Anwenden nichts mehr ändert. Diese Eigenschaft, die *Idempotenz*, macht Infrastrukturcode sicher für wiederholte und automatisierte Ausführung.

Die meisten ausgereiften Tools sind im Kern deklarativ, unterscheiden sich aber in der Sprache, in der die Deklaration geschrieben wird, und darin, wer festhält, was existiert. Die Tabelle fasst die wichtigsten Familien zusammen.

Ansätze für Infrastrukturcode

Ansatz	Sie schreiben	Beispiele	Stärken	Schwächen
Imperative Skripte	Folgen von CLI- oder SDK-Aufrufen	Shell-Skripte, die Cloud-CLIs aufrufen, SDK-Programme	Vertraut, flexibel, schnell für einmalige Aufgaben	Ohne große Sorgfalt nicht idempotent; kein Nachweis, was existiert; schwer sicher zu ändern
Deklarativ, domänenspezifische Sprache	Ressourcen und ihre Beziehungen in einer Konfigurationssprache	Terraform und OpenTofu (HCL), AWS CloudFormation (YAML oder JSON), Azure Bicep, Kubernetes-Manifeste	Lesbare, überprüfbare Pläne; idempotent; große Provider-Ökosysteme	Schleifen, Bedingungen und Abstraktion können umständlich sein; eine Sprache zum Lernen
Deklaratives Modell, Universalsprache	Code in TypeScript, Python, Go, C# oder Java, der ein deklaratives Modell aufbaut	Pulumi, AWS Cloud Development Kit (CDK)	Echte Abstraktionen, Testwerkzeuge und Editoren; Entwicklern vertraut	Leicht, cleveren, schwer zu prüfenden Infrastrukturcode zu schreiben; der Plan ist im Code weniger sichtbar
Kontinuierlicher Abgleich	Ressourcen im Kubernetes-Stil, die Cloud-Infrastruktur beschreiben	Crossplane, Kubernetes-Controller der Cloud-Anbieter	Derselbe Reconciliation Loop und dieselben Tools wie für Anwendungen; Drift wird fortlaufend korrigiert	Braucht eine Kubernetes-Control-Plane; die Fehlersuche erstreckt sich über zwei Systeme
Konfigurationsmanagement	Gewünschte Konfiguration von Software auf bestehenden Servern	Ansible, Chef, Puppet	Verwaltet langlebige Maschinen, Netzwerkgeräte und On-Premises-Landschaften	Server sammeln Historie und Drift an; langsamer und weniger vorhersehbar als ihr Ersetzen

Die Tools und der Fork OpenTofu

Terraform wurde zum meistgenutzten Multi-Cloud-Tool, weil eine Sprache und ein Workflow Tausende von Providern abdecken: AWS, Azure, Google Cloud, Cloudflare, Kubernetes, GitHub, Monitoring-Dienste und mehr. Am 10. August 2023 stellte HashiCorp Terraform von der Open-

Source-Lizenz Mozilla Public License 2.0 auf die Business Source License 1.1 um, die interne und kommerzielle Nutzung erlaubt, aber verbietet, es in Produkten anzubieten, die mit HashiCorps eigenen konkurrieren. Version 1.5.7 war das letzte Release unter der alten Lizenz.

Die Antwort der Community war ein Fork. Innerhalb weniger Tage erschien ein Manifest, das eine Rückkehr zu Open Source forderte; im September 2023 nahm die Linux Foundation den Fork als OpenTofu auf; sein erstes stabiles Release, 1.6.0, folgte im Januar 2024; und 2025 wurde es als Sandbox-Projekt in die CNCF aufgenommen. IBM schloss die Übernahme von HashiCorp im Februar 2025 ab. Die beiden Tools bleiben in Sprache und Providern weitgehend kompatibel, entwickeln sich aber bei den Funktionen auseinander: OpenTofu hat zum Beispiel eine clientseitige Verschlüsselung des State ergänzt, während Terraforms kommerzielle Plattform eigene Fähigkeiten hinzugefügt hat. Für Architektinnen und Architekten erinnert die Episode daran, dass Lizenz und Governance eines Tools Teil seines Risikoprofils sind und dass offene Standards und eine glaubwürdige Alternative dieses Risiko beherrschbar halten.

Unter den Tools für eine einzelne Cloud werden AWS CloudFormation (2011) und Microsofts Bicep (2020 eingeführt, kompiliert zu Azure-Resource-Manager-Templates) vom Anbieter betrieben, der festhält, was existiert, sodass kein State-File zu verwalten ist, um den Preis der Bindung an eine Cloud. Pulumi (2018) und das AWS CDK lassen Entwickler Infrastruktur in Universalsprachen schreiben; Crossplane erweitert Kubernetes, um Cloud-Ressourcen per Abgleich zu verwalten. Die richtige Wahl hängt davon ab, welche Clouds Sie nutzen, von den Fähigkeiten Ihres Teams, der Reife der benötigten Provider, der Lizenzpolitik und dem, was der Rest der Organisation bereits einsetzt; Einheitlichkeit über Teams hinweg ist meist mehr wert als die Vorzüge eines einzelnen Tools im Detail.

```
terraform {
  required_version = "≥ 1.6"
  backend "gcs" {                 # remote state; this backend locks natively
    bucket = "fd-infra-state"
    prefix = "orders/prod"
  }
}

variable "environment" { type = string }

module "orders_db" {
  source      = "git::https://git.example.com/modules/postgres.git?ref=v3.2.0"
  name        = "orders-${var.environment}"
  size        = var.environment == "prod" ? "large" : "small"
  high_avail  = var.environment == "prod"   # standby in a second zone
  backup_days = 14
}

output "orders_db_host" {
  value = module.orders_db.host
}
```

OpenTofu oder Terraform (HCL): Remote State und ein versioniertes hauseigenes Modul, das Provider-Details hinter einer kleinen Schnittstelle verbirgt.

Das Ausführen von `tofu plan` (oder `terraform plan`) zeigt, was sich ändern würde, bevor irgendetwas passiert. Der Plan ist das wichtigste Artefakt im Workflow: Reviewer lesen ihn im Pull Request neben dem Code, und genau er wird angewendet.

```
  # module.orders_db.google_sql_database_instance.this will be updated in-place
  ~ resource "google_sql_database_instance" "this" {
      ~ settings {
          ~ tier = "db-custom-2-8192" → "db-custom-4-16384"
        }
    }

Plan: 0 to add, 1 to change, 0 to destroy.
```

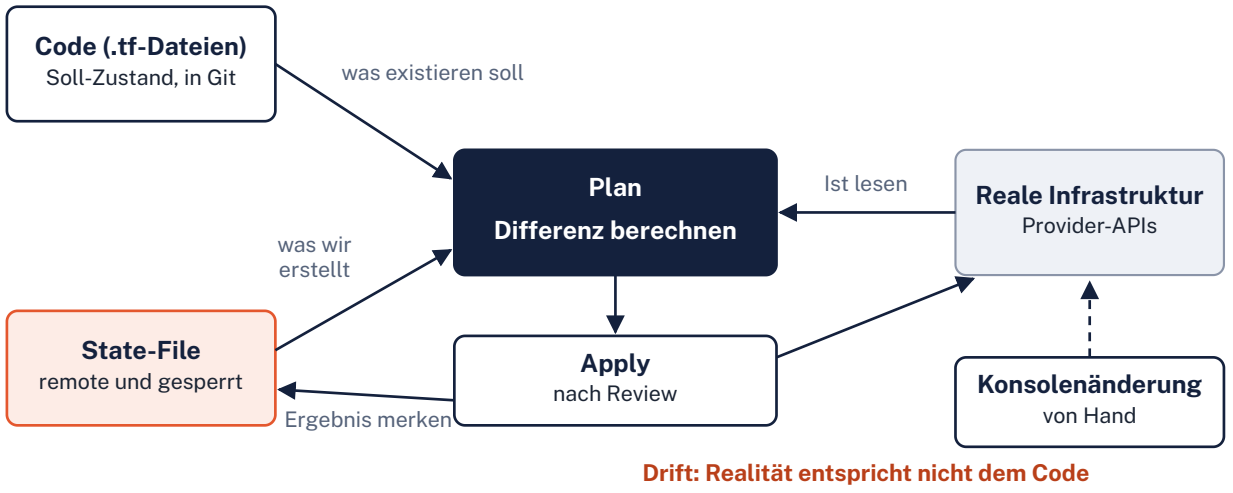
Ein gekürzter Plan für die Größenänderung der Bestelldatenbank in der Produktion. Die Tilde markiert ein In-place-Update; ein -/+ würde Zerstören und Neuerstellen bedeuten, was bei einer Datenbank ein Alarmsignal ist.

State, Locking und Drift

Deklarative Tools müssen wissen, welche realen Ressourcen zu welchen Definitionen gehören. Terraform und OpenTofu halten das im State fest: einer Datei, die jede Ressource im Code der Kennung des realen Objekts zuordnet, mit ihren zuletzt bekannten Attributen. Die Planung

vergleicht drei Dinge: den Code (was existieren soll), den State (was das Tool erstellt hat) und die reale Infrastruktur, gelesen über die API des Providers (was tatsächlich existiert).

Plan, Apply, State und Drift



Ein Plan vergleicht Code, State und Realität; eine manuelle Änderung in der Konsole lässt die Realität von beiden abweichen, und der nächste Plan wird versuchen, sie rückgängig zu machen.

Der State braucht Sorgfalt, weil er kritisch und sensibel ist. Er muss remote gespeichert werden (in Object Storage oder einem verwalteten Dienst), nicht auf einem Laptop, damit das ganze Team und die Pipeline dieselbe Version teilen. Er muss während Änderungen *gesperrt* sein, damit sich zwei Applies nicht überlagern und ihn beschädigen. Er muss gesichert und versioniert werden, weil ein verlorenes State-File das Tool ohne Kenntnis der Ressourcen lässt, die es erstellt hat. Und er ist ein Secret: Der State enthält oft Datenbankpasswörter und Schlüssel im Klartext, also muss der Zugriff darauf beschränkt und er im Ruhezustand verschlüsselt sein (OpenTofu kann ihn auch clientseitig verschlüsseln).

Drift ist jede Abweichung zwischen Code und Realität, meist durch hastige manuelle Änderungen, durch ein anderes Tool, das dieselbe Ressource anfasst, oder durch einen teilweisen Fehlschlag beim Apply. Drift ist gefährlich, weil der nächste Apply stillschweigend eine Änderung zurücknehmen kann, auf die sich jemand verlassen hat, oder in der Produktion an einer Abweichung scheitert, von der niemand wusste. Teams erkennen sie, indem sie regelmäßig (zum Beispiel nächtlich) einen Plan ausführen und bei jeder vorgeschlagenen Änderung alarmieren, und verhindern sie, indem sie Schreibzugriff in der Konsole auf ein protokolliertes Break-Glass-Verfahren beschränken.

Der State bestimmt auch den *Blast Radius*. Ein riesiger State für ein ganzes Unternehmen bedeutet, dass jeder Plan langsam ist, jeder Apply alles riskiert und ein einziger Lock jedes Team blockiert. Den State nach Komponente und Umgebung aufzuteilen (Netzwerk,

gemeinsamer Cluster, die Ressourcen jedes Service; Produktion getrennt von Staging) hält Pläne schnell und Ausfälle begrenzt, um den Preis, Ausgaben wie Netzwerkkennungen zwischen States weiterzugeben. Refactoring wird unterstützt: `moved`-Blöcke benennen Ressourcen um, ohne sie zu zerstören, und `import` übernimmt bestehende, von Hand erstellte Ressourcen in die Verwaltung.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine abgedriftete Firewall-Regel

Während eines Incidents am Freitagabend öffnet ein Engineer über die Cloud-Konsole die Firewall der Bestelldatenbank in der Produktion für die IP-Adresse eines externen Dienstleisters, damit dieser bei der Diagnose einer langsamen Abfrage helfen kann. Niemand aktualisiert den Code.

Erkennung	Der nächtliche Drift-Job führt `tofu plan` für jeden Produktions-State aus. Am Samstag meldet er *1 to change* für die Firewall der Bestelldatenbank und postet den Plan in den Kanal des Plattformteams.
Diagnose	Der Plan zeigt eine zusätzliche erlaubte Quelladresse. Das Audit-Log der Cloud zeigt, wer sie hinzugefügt hat, wann und zu welchem Incident.
Entscheidung	Der Zugriff war vorübergehend, und der Incident ist abgeschlossen, also ist der Code richtig und die Realität falsch. Das Anwenden des Plans entfernt die Regel. Wäre die Änderung dauerhaft nötig gewesen, hätte die Korrektur darin bestanden, sie stattdessen über einen geprüften Pull Request in den Code aufzunehmen.
Vorbeugung	Schreibzugriff in der Produktionskonsole wird durch eine Break-Glass-Rolle ersetzt, die eine Begründung verlangt und bei Nutzung alarmiert. Eine Policy-Regel (siehe unten) lehnt jede Firewall-Regel für Datenbanken ab, die für das gesamte Internet offen ist.
Was es zeigt	Drift ist nicht nur ein Ordnungsproblem: Ein undokumentiertes Loch in einer Firewall ist ein Sicherheitsrisiko, und ohne Erkennung hätte der nächste, unabhängige Apply es entfernt, ohne dass jemand wusste, warum der Dienstleister den Zugang verlor.

Die Faustregel: Die Realität darf in einem Notfall für Minuten vom Code abweichen, aber die Abweichung muss bis zum nächsten Arbeitstag entweder in Code übernommen oder rückgängig gemacht sein.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: Warum das Tool Buch führt

Der Punkt, an dem hier die meisten stolpern, ist, warum Tools wie OpenTofu und Terraform einen State führen. Der Code sagt, was existieren soll, aber nicht, auf welche reale Ressource im Cloud-Konto er sich bezieht. Begleiten Sie die Bestelldatenbank der Produktion durch ihr Leben.

- Das Team beschreibt orders-prod im Code mit dem hauseigenen Datenbankmodul. Noch existiert nichts, also ist der State leer, und der erste Plan schlägt vor, sie hinzuzufügen.
- Beim Apply bittet das Tool den Provider, die Datenbank zu erstellen. Der Provider gibt ihr eine Kennung, und das Tool hält im State fest, dass diese Ressource im Code die reale Datenbank mit dieser Kennung ist, samt ihren Einstellungen.
- Monate später ändert das Team die Größe im Code. Der Plan liest den Code, findet im State, welche reale Datenbank das ist, fragt den Provider nach ihren aktuellen Einstellungen und schlägt eine In-place-Änderung vor.
- Ohne den State hätte das Tool keinen Nachweis, dass diese Datenbank zu diesem Code gehört, und würde vorschlagen, sie erneut zu erstellen.
- Der State zeigt auch, was der Code nicht mehr erwähnt. Löscht das Team das Modul aus dem Code, schlägt der Plan vor, die Datenbank zu zerstören, die der State noch aufführt.
- Deshalb ist der State gemeinsam genutzt, gesperrt, gesichert und vertraulich: Zwei gleichzeitige Applies könnten ihn beschädigen, sein Verlust ließe das Tool ohne Kenntnis dessen, was es erstellt hat, und er enthält oft Passwörter.
- **Die Lehre:** Der Code sagt, was existieren soll, die reale Infrastruktur ist das, was existiert, und der State verbindet beides. Lesen Sie jeden Plan mit allen dreien im Kopf, und behandeln Sie einen Plan, der eine Datenbank zerstören will, als Warnsignal.

Module und Umgebungen

Ein Modul ist eine wiederverwendbare Einheit von Infrastrukturcode mit Eingaben und Ausgaben: eine PostgreSQL-Datenbank mit Backups und Monitoring, das Netzwerk und der Load Balancer eines Service, ein Standard-Namespace in Kubernetes mit seinen Quotas und Policies. Ein gutes Modul wendet das Information Hiding aus Kapitel 3 an: eine kleine Schnittstelle (Name, Größe, Hochverfügbarkeit ja oder nein) vor den vielen providerspezifischen Details, die ein Service-Team nicht kennen müssen sollte. Module werden wie Bibliotheken versioniert und veröffentlicht, und Umgebungen verweisen auf eine bestimmte Version, wie das `ref=v3.2.0` im Beispiel, sodass eine Moduländerung die Produktion erst erreicht, wenn jemand bewusst aktualisiert.

Umgebungen sollten aus denselben Modulen mit unterschiedlichen Eingaben gebaut werden, nicht aus kopiertem Code. Ein gängiger Aufbau ist ein Verzeichnis (oder Repository) pro Umgebung und Komponente, jeweils mit eigenem State, das gemeinsame, versionierte Module aufruft; eine Änderung wird befördert, indem man Modulversion oder Eingaben zuerst im Staging, dann in der Produktion aktualisiert. Per Copy-and-paste erstellte Umgebungen laufen innerhalb von Monaten auseinander, bis Staging nichts mehr über die Produktion aussagt. Workspaces, die einer Konfiguration mehrere States geben, eignen sich besser für kurzlebige Kopien wie Testumgebungen pro Branch als für langlebige Produktions- und Staging-Umgebungen.

Hüten Sie sich vor Überabstraktion. Ein Modul mit vierzig booleschen Eingaben, das jedem Team dienen will, ist so schwer zu ändern wie der Code, den es ersetzt hat. Besser sind wenige meinungsstarke Module, die die Standards der Organisation abbilden; genau das bietet ein Plattformteam in Abschnitt 10.7 als Golden Path an.

Immutable Infrastructure oder Konfigurationsmanagement

Tools für Konfigurationsmanagement halten langlebige Server in einem beschriebenen Zustand: Ansible verbindet sich per SSH und pusht Änderungen ohne Agent; Chef und Puppet betreiben Agenten, die ihre Konfiguration abrufen und regelmäßig darauf konvergieren. Das funktioniert, aber jeder Server sammelt Historie an: Pakete, die während eines Incidents installiert wurden, von Hand bearbeitete Dateien, halb angewendete Updates. Martin Fowler nannte das Ergebnis einen *Snowflake Server*, einzigartig und nicht reproduzierbar, und schlug als Heilmittel den *Phoenix Server* vor, der regelmäßig niedergebrannt und von Grund auf neu aufgebaut wird.

Immutable Infrastructure, ein Begriff, den Chad Fowler 2013 bekannt machte, führt das Heilmittel zu Ende: einen laufenden Server nie ändern. Jede Änderung, ob ein Anwendungsrelease oder ein Betriebssystem-Patch, erfolgt durch den Bau eines neuen Images (eines Container-Images oder eines Maschinen-Images mit einem Tool wie Packer), seinen Test und das Ersetzen der alten Instanzen, typischerweise per Rolling- oder Blue-Green-Deployment. Container auf Kubernetes sind standardmäßig unveränderlich. Die Vorteile sind keine Drift, identische Instanzen, mühelos wiederholbare Umgebungen und Rollback durch erneutes Deployment des vorherigen Images; die Kosten sind eine Image-Pipeline, der Bedarf an entsorgbaren, zustandslosen Prozessen (Abschnitt 10.3) und eine längere Durchlaufzeit für eine einzelne dringende Änderung.

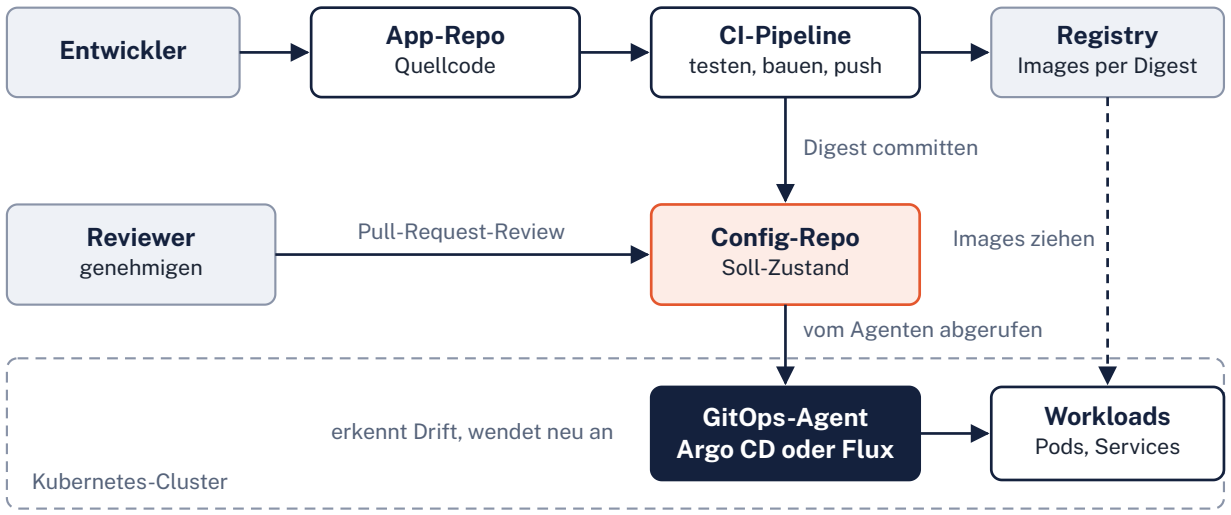
Konfigurationsmanagement ist nicht verschwunden. Es bleibt das richtige Werkzeug für Landschaften, die sich nicht als Ganzes ersetzen lassen: On-Premises-Server, Netzwerkausrüstung, Datenbanken auf dedizierter Hardware und die Basis-Images selbst, die

oft gebaut werden, indem Ansible innerhalb des Image-Builds läuft. Die architektonische Regel lautet, Veränderlichkeit so weit nach unten und so früh wie möglich zu verlagern: beim Bau der Images konfigurieren und dann das Laufende als schreibgeschützt behandeln.

GitOps

GitOps wendet den Reconciliation Loop aus Abschnitt 10.2 auf ganze Umgebungen an. Den Begriff prägte 2017 Alexis Richardson von Weaveworks, dem Unternehmen hinter dem Tool Flux. Das Projekt OpenGitOps, eine Arbeitsgruppe der CNCF, definiert vier Prinzipien: Der gewünschte Zustand ist *deklarativ*; er ist *versioniert und unveränderlich*, mit vollständiger Historie; er wird von Software-Agenten *automatisch abgerufen*; und er wird *fortlaufend abgeglichen*, wobei Agenten den tatsächlichen Zustand beobachten und den gewünschten anwenden. Die beiden führenden Agenten, Argo CD und Flux, sind graduierte CNCF-Projekte; Flux wurde nach der Schließung von Weaveworks 2024 als Community-Projekt fortgeführt.

Ein GitOps-Auslieferungsablauf



Die CI baut und veröffentlicht Images; Git hält den gewünschten Zustand; ein Agent im Cluster ruft ihn ab, wendet ihn an und korrigiert fortlaufend Drift.

Das Pull-Modell hat einen Sicherheitsvorteil gegenüber einer Pipeline, die Änderungen in den Cluster pusht: Das CI-System braucht keine Zugangsdaten für die Produktion, weil der Agent im Cluster selbst abruft, was er ausführen soll. Git wird zum Audit-Log und zur Schnittstelle für Änderungen, auch im Betrieb: Einen Service zu skalieren oder zurückzurollen ist ein geprüfter Commit oder ein Revert. Drift wird automatisch korrigiert, sodass ein manuelles `kubectl edit` nur bis zur nächsten Synchronisierung hält.

GitOps bringt eigene Entwurfsfragen mit. Secrets dürfen nicht im Klartext in Git liegen; Teams verschlüsseln sie (SOPS, Sealed Secrets) oder speichern Referenzen, die der External Secrets Operator aus einem Secrets Manager auflöst. Die Beförderung zwischen Umgebungen braucht eine Konvention, etwa ein Verzeichnis pro Umgebung mit Pull Requests, die einen neuen Image-Digest von Staging in die Produktion kopieren. Die automatische Drift-Korrektur muss in manchen Notfällen bewusst pausiert werden, und die Struktur des Repositorys muss die Zuständigkeiten abbilden, damit Teams ihre eigenen Services ändern können, ohne die anderer anzufassen. GitOps ist für Kubernetes am natürlichsten, aber dieselben Prinzipien gelten für Cloud-Infrastruktur über Crossplane oder über Pipelines, die Infrastrukturcode aus gemergten Commits planen und anwenden.

Policy as Code

Policy as Code drückt Regeln über Infrastruktur als maschinell prüfbaren Code aus statt als Wiki-Seite: Storage-Buckets dürfen nicht öffentlich sein, Datenbanken müssen verschlüsselt und gesichert sein, jede Ressource muss ein Tag für Verantwortliche und Kostenstelle tragen, nur zugelassene Regionen dürfen genutzt werden, Container dürfen nicht als Root laufen, Produktions-Images müssen aus der Registry des Unternehmens stammen. Die Prüfungen laufen an mehreren Stellen: auf dem Plan im Pull Request (Open Policy Agent mit Conftest, Checkov, HashiCorp Sentinel in seinen kommerziellen Produkten), bei der Zulassung in einen Kubernetes-Cluster (OPA Gatekeeper, Kyverno) und fortlaufend gegen laufende Ressourcen (Policy-Dienste der Cloud-Anbieter).

Policy as Code macht Governance von einem Review-Meeting zu einer schnellen, einheitlichen Leitplanke, die sich im Pull Request selbst erklärt. Sie funktioniert am besten, wenn Policies wenige, klar begründet, versioniert und wie jeder Code getestet sind, wenn sie im Warnmodus beginnen, bevor sie blockieren, und wenn es ein protokolliertes Ausnahmeverfahren gibt. Kapitel 11 behandelt, wie solche automatisierten Prüfungen in die Architektur-Governance passen (Architektur-Governance in agilen Organisationen).

Infrastrukturcode testen, von schnell bis langsam

1. **Statische Prüfungen bei jedem Commit**
   Formatierung, `validate`, Linting (zum Beispiel TFLint) und Sicherheitsscanner finden Syntaxfehler, veraltete Argumente und offensichtliche Fehlkonfigurationen in Sekunden.

2. **Policy-Prüfungen auf dem Plan**
   Den Plan als JSON ausgeben und gegen die Policies der Organisation auswerten; Reviewer sehen Verstöße im Pull Request.

3. **Modultests**
   `terraform test` und `tofu test` führen Assertions gegen ein Modul aus, mit gemockten Providern für schnelle Tests im Stil von Unit-Tests oder mit echten für Integrationstests.

4. **Integrationstests in kurzlebigen Umgebungen**
   Das Modul oder die Umgebung auf ein Wegwerfkonto anwenden, prüfen, ob es funktioniert (eine Datenbank nimmt Verbindungen an, ein Service antwortet), und es dann zerstören. Tools wie Terratest automatisieren das; es ist langsam und kostet Geld, also reservieren Sie es für gemeinsame Module.

5. **Plan-Review und gestufter Rollout**
   Zuerst auf Staging anwenden, dann auf die Produktion, aus denselben Versionen; ein Mensch liest jeden Produktionsplan, besonders Zerstörungen und Ersetzungen.

6. **Drift-Erkennung und Wiederherstellungsübungen**
   Geplante Plan-Läufe finden Drift; eine Umgebung regelmäßig aus Code neu aufzubauen beweist, dass der Code und nicht das Gedächtnis die Quelle der Wahrheit ist.

Checkliste: Infrastrukturcode bereit für die Produktion

Verwenden Sie sie, wenn Sie das Infrastruktur-Repository eines Teams oder ein neues Plattformmodul prüfen.

- [ ] **Alles ist Code**
  Keine Produktionsressource existiert nur, weil jemand geklickt hat; bestehende, von Hand erstellte Ressourcen wurden importiert.
- [ ] **Remote, gesperrter, verschlüsselter State**
  Gemeinsames Backend mit Locking, Versionierung und eingeschränktem Zugriff; State nach Komponente und Umgebung aufgeteilt, um den Blast Radius zu begrenzen.
- [ ] **Geprüfte Pläne**
  Jede Änderung ist ein Pull Request, der den Plan zeigt; Zerstörungen und Ersetzungen zustandsbehafteter Ressourcen erfordern eine ausdrückliche Genehmigung.
- [ ] **Versionierte Module**
  Umgebungen pinnen Modulversionen; Staging und Produktion nutzen dieselben Module mit unterschiedlichen Eingaben, keine Kopien.
- [ ] **Automatisch durchgesetzte Policies**
  Regeln zu Verschlüsselung, Netzwerkfreigabe, Tagging und Regionen werden bei jedem Plan und bei der Zulassung in den Cluster geprüft.
- [ ] **Secrets außerhalb von Code und Git**
  Zugangsdaten kommen aus einem Secrets Manager oder über Workload Identity; Zugriff auf den State wird wie Zugriff auf Secrets behandelt.
- [ ] **Tests in der Pipeline**
  Statische und Policy-Prüfungen bei jeder Änderung; Modultests für gemeinsame Module; Integrationstests für kritische.
- [ ] **Drift erkannt und behandelt**
  Geplante Plan-Läufe alarmieren bei Abweichungen; Schreibzugriff in der Konsole ist ein protokolliertes Break-Glass-Verfahren.
- [ ] **Wo möglich unveränderlich**
  Server und Container werden aus Images ersetzt statt an Ort und Stelle gepatcht; Konfiguration geschieht beim Bau der Images.
- [ ] **Wiederherstellung geübt**
  Eine Umgebung wurde innerhalb der vereinbarten Wiederherstellungszeit aus Code neu aufgebaut, einschließlich Datenwiederherstellung (siehe Abschnitt 10.6).

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum ist Idempotenz für Infrastrukturcode wichtig?**

Weil Automatisierung Code oft erneut ausführt. Ein deklaratives, idempotentes Tool ändert nichts, wenn die Realität bereits dem Code entspricht, also ist ein erneuter Lauf sicher; ein nicht idempotentes Skript versucht womöglich, alles noch einmal zu erstellen.

▼ **Nennen Sie drei Dinge, die ein Plan vergleicht, und was jedes darstellt.**

Den Code (was existieren soll), den State (was das Tool erstellt hat und nachverfolgt) und die reale Infrastruktur, gelesen über die API des Providers (was tatsächlich existiert).

▼ **Warum sollte der State nicht als eine Datei für eine ganze Organisation geführt werden?**

Ein einziger State bedeutet langsame Pläne, einen Lock, der jedes Team blockiert, und einen Blast Radius, der alles umfasst; die Aufteilung nach Komponente und Umgebung hält Änderungen schnell und Ausfälle begrenzt.

▼ **Welchen Sicherheitsvorteil hat das Pull-Modell von GitOps gegenüber einer Pipeline, die in den Cluster pusht?**

Das CI-System braucht keine Zugangsdaten für die Produktion, weil ein Agent im Cluster den gewünschten Zustand aus Git abruft und anwendet.

▼ **Wann ist Konfigurationsmanagement noch eine gute Wahl?**

Für langlebige Maschinen, die sich nicht einfach ersetzen lassen, etwa On-Premises-Server und Netzwerkgeräte, und zum Konfigurieren von Basis-Images während des Image-Builds.

▼ **Probieren Sie es aus: Ein Team bewahrt sein Infrastruktur-State-File auf dem Laptop eines Ingenieurs auf, mit allen Umgebungen in einem State, und hat Staging erstellt, indem es den Produktionscode kopiert und bearbeitet hat. Listen Sie die Risiken auf, und sagen Sie, wie Sie das neu strukturieren würden.**

Der State könnte verloren gehen, niemand sonst kann sicher Änderungen ausführen, und zwei Personen könnten kollidieren. Ein einziger State bedeutet auch, dass ein Fehler jede Umgebung beschädigen kann. Kopierter Code driftet auseinander. Verschieben Sie den State in einen entfernten, gesperrten, versionierten und zugriffskontrollierten Speicher, getrennt nach Umgebung und Komponente, um den Blast Radius zu begrenzen. Bauen Sie versionierte Module mit kleinen Schnittstellen, und verwenden Sie sie für jede Umgebung mit anderen Eingaben. Ergänzen Sie geplante Plan-Läufe, um Drift zu erkennen.

QUELLEN

- OpenTofu: https://opentofu.org/
- Terraform documentation: State: https://developer.hashicorp.com/terraform/language/state
- Terraform documentation: Tests: https://developer.hashicorp.com/terraform/language/tests
- OpenGitOps: GitOps principles: https://opengitops.dev/
- Martin Fowler: PhoenixServer: https://martinfowler.com/bliki/PhoenixServer.html

CI/CD-Pipelines und Deployment-Strategien

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.8 Wie Software entsteht
- 2.6 Änderbarkeit und Deploybarkeit
- 7.7 Schemaevolution

Begriffe von früher, die er verwendet: Container-Image, Expand and Contract, Service Level Objective (SLO), Verteilter Monolith

DIE IDEE IN KÜRZE

Stellen Sie sich eine Bäckereikette vor, die ein neues Brot einführt. Sie backt eine Testcharge, prüft sie in der eigenen Backstube und verkauft sie dann einen Vormittag lang in einer Filiale, bevor sie es an alle Filialen schickt. Beschweren sich Kunden, waren nur die Kunden einer Filiale betroffen, und das alte Brot liegt noch in den Regalen. Eine Software-Pipeline ist dieselbe Routine, nur automatisiert: Jede Änderung wird einmal gebaut, in Stufen geprüft, an wenigen Nutzern ausprobiert und erst weiter verteilt, wenn die Belege zeigen, dass sie gut ist. Ein separater Schalter entscheidet, wann Kunden ein neues Feature tatsächlich sehen.

WAS SIE WISSEN MÜSSEN

- Eine Deployment-Pipeline (Humble und Farley, 2010) schickt jeden Commit durch automatisierte Stufen, die schnellsten zuerst; ein Fehlschlag in einer beliebigen Stufe stoppt diese Änderung und benachrichtigt ihren Autor.
- Build once, deploy many: Verpacken Sie jede Änderung in ein einziges unveränderliches Artefakt, legen Sie es in einem Artefakt-Repository ab und reichen Sie genau dieses Artefakt durch jede Umgebung weiter; nur die Konfiguration ändert sich.
- Trunk-Based Development hält Branches kurzlebig und Änderungen klein; das macht Continuous Integration erst wirklich möglich und jede Änderung leichter zu prüfen, zu testen und rückgängig zu machen.
- Recreate-, Rolling-, Blue-Green-, Canary- und Shadow-Deployments tauschen zusätzliche Infrastruktur und Komplexität gegen geringere Betroffenheit der Nutzer und schnelleres Rollback; alle außer Recreate verlangen, dass alte und neue Version kompatibel sind.

- Feature-Flags trennen Deployment von Release; Hodgsons Kategorien sind Release-, Experiment-, Ops- und Permissioning-Toggles, und jedes Flag verursacht Kosten, bis es entfernt wird.
- Datenbankänderungen müssen abwärtskompatibel sein (Expand and Contract) und als eigene Pipeline-Schritte laufen, denn Daten lassen sich nicht wie Code zurückrollen.
- Progressive Delivery automatisiert Canary-Entscheidungen, indem sie den Canary statistisch mit einer frischen Baseline vergleicht; die Standardreaktion auf ein schlechtes Release lautet: erst zurückrollen, dann diagnostizieren.

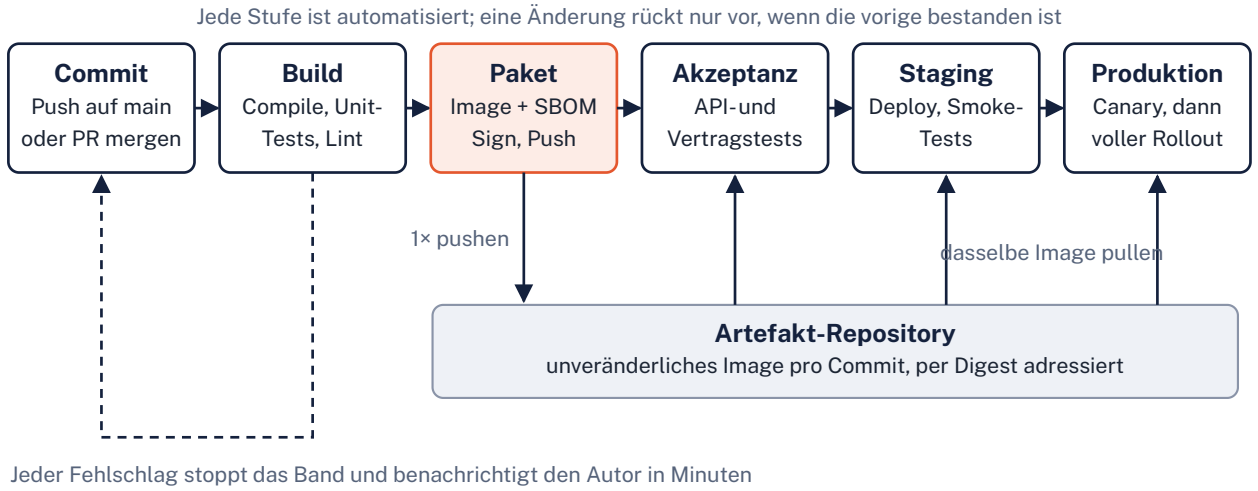
Vom Commit bis in die Produktion: wozu eine Pipeline da ist

Abschnitt 0.8 (Wie Software entsteht) hat Continuous Integration (jede Änderung wird automatisch zusammengeführt und geprüft), Continuous Delivery (der Hauptbranch ist jederzeit releasefähig) und Continuous Deployment (jede Änderung, die die Prüfungen besteht, geht ohne menschliche Entscheidung in Produktion) eingeführt. Dieser Abschnitt behandelt sie als Architektur: als Gestaltung des Wegs, den eine Änderung nimmt, und der Strategien, die sie in Produktion bringen, ohne Nutzern zu schaden.

Die zentrale Idee ist die Deployment-Pipeline, beschrieben von Jez Humble und David Farley in *Continuous Delivery* (2010). Jeder Commit löst eine automatisierte Folge von Stufen aus. Frühe Stufen sind schnell und günstig und fangen die meisten Probleme ab; spätere Stufen sind langsamer, realistischer und teurer. Eine Änderung gelangt nur dann in die nächste Stufe, wenn die vorige bestanden wurde; je weiter ein Build kommt, desto mehr Vertrauen hat er sich also verdient. Ein Fehlschlag an beliebiger Stelle stoppt diesen Build und geht direkt an die Person zurück, die die Änderung gemacht hat, solange sie ihr noch frisch im Gedächtnis ist.

Für Architekten ist die Pipeline aus drei Gründen wichtig. Erstens wird hier die Deploybarkeit, ein Qualitätsmerkmal aus Abschnitt 2.6, tatsächlich erreicht oder verspielt. Zweitens begrenzt ihre Geschwindigkeit, wie schnell die Organisation auf eine Marktveränderung oder eine Sicherheitslücke reagieren kann: Braucht ein einzeiliger Fix zwei Tage bis in die Produktion, ist die Architektur zwei Tage langsam, wie auch immer der Code aussieht. Drittens bestimmt die Gestalt des Systems die Gestalt der Pipeline: Unabhängig deploybare Services können unabhängige Pipelines haben, aber ein verteilter Monolith erzwingt koordinierte Releases, wie clever die Werkzeuge auch sein mögen.

Die Stufen einer typischen Pipeline



Ein Artefakt wird einmal gebaut und gepusht; jede spätere Stufe deployt genau dieses Artefakt, und jeder Fehlschlag stoppt das Band.

Was jede Stufe tut und wie lange sie dauern sollte

1. **Commit-Stufe (Ziel: unter 10 Minuten)**
   Kompilieren, Unit-Tests, Linter, statische Analyse und Schwachstellenscans der Abhängigkeiten ausführen. Auf diese Stufe warten die Entwickler, also muss sie schnell sein; langsame Tests gehören weiter nach hinten.

2. **Paket**
   Das deploybare Artefakt bauen (meist ein Container-Image), eine Software Bill of Materials (SBOM) erzeugen, es signieren und in das Artefakt-Repository pushen. Maßnahmen für die Software-Lieferkette behandelt Abschnitt 8.6.

3. **Akzeptanz und Integration**
   Das Artefakt in eine temporäre oder gemeinsame Testumgebung deployen und API-Tests, Consumer-Driven-Contract-Tests und einige End-to-End-Abläufe ausführen. Performance- und Sicherheitstests können hier parallel oder nach Zeitplan laufen.

4. **Staging**
   In eine Umgebung deployen, die wie die Produktion konfiguriert ist, Datenbankmigrationen anwenden, Smoke-Tests ausführen. Ihr Wert hängt davon ab, wie genau sie der Produktion in Konfiguration, Datenstruktur und Abhängigkeiten entspricht.

5 **Produktion**
Mit einer Strategie deployen, die die Betroffenheit begrenzt (Rolling, Blue-Green oder Canary), mit echten Metriken gegen Service Level Objectives prüfen und dann automatisch abschließen oder zurückrollen.

Build once, deploy many und das Artefakt-Repository

Eine Regel aus *Continuous Delivery*, die noch immer erstaunlich oft verletzt wird, lautet Build once, deploy many: Jede Änderung genau einmal kompilieren und verpacken und dann dasselbe Artefakt durch jede Umgebung weiterreichen. Baut ein Team für Staging und noch einmal für die Produktion, ist das Getestete nicht das Ausgelieferte. Ein anderer Compiler-Patch, eine über Nacht auf eine neuere Version aufgelöste Abhängigkeit oder ein geändertes Basis-Image können die beiden Builds unterscheiden, und die Tests beweisen nichts über den zweiten.

Was sich zwischen Umgebungen unterscheidet, ist die Konfiguration, die beim Deployment oder beim Start übergeben wird (Datenbankadressen, Zugangsdaten, Feature-Flag-Einstellungen) und nie in das Artefakt eingebacken ist. Das ist der Konfigurationsfaktor der Twelve-Factor App. Ein Container-Image mit dem Tag `orders:4f1c2ab` (dem Commit-Hash) wird in Test, Staging und Produktion deployt; nur seine Umgebungsvariablen und eingebundenen Secrets ändern sich.

Das Artefakt-Repository ist der Speicher zwischen Build und Deployment: eine Container-Registry (Amazon ECR, Azure Container Registry, Google Artifact Registry, GitHub Container Registry, Harbor) oder ein universelles Repository wie JFrog Artifactory oder Sonatype Nexus für Pakete und Binärdateien. Behandeln Sie es als kritische Infrastruktur. Deployen Sie über einen unveränderlichen Digest (`sha256:...`) statt über ein veränderliches Tag wie `latest` , das auf ein anderes Image umgesetzt werden kann; bewahren Sie Artefakte, die in Produktion sind oder waren, so lange auf, wie Sie womöglich auf sie zurückrollen oder sie untersuchen müssen; und halten Sie die Herkunft (Provenance) fest, also welcher Commit, welcher Pipeline-Lauf und welche Abhängigkeiten jedes Artefakt erzeugt haben.

Trunk-Based Development und kleine Batches

Wie Code verzweigt wird, entscheidet, wie oft er wirklich integriert wird. Beim Trunk-Based Development integrieren alle mindestens täglich kleine Änderungen in einen Hauptbranch (den Trunk), entweder durch direkte Commits oder über kurzlebige Branches und Pull Requests, die Stunden statt Wochen leben. Das DORA-Forschungsprogramm hat kurzlebige Branches und

häufiges Mergen wiederholt mit höherer Lieferleistung in Verbindung gebracht, und es ist eine Voraussetzung für Continuous Integration im wörtlichen Sinn: Code, der drei Wochen auf einem Branch liegt, wird nicht kontinuierlich integriert.

Langlebige Feature-Branches fühlen sich sicherer an, weil unfertige Arbeit getrennt bleibt, doch das Risiko wird nur aufgeschoben: Je länger zwei Branches auseinanderlaufen, desto größer und riskanter wird der spätere Merge und desto länger sieht die Pipeline Code, der nicht der Code ist, der ausgeliefert wird. GitFlow, ein Branching-Modell mit langlebigen Develop- und Release-Branches, das Vincent Driessen 2010 veröffentlichte, wurde breit übernommen; 2020 ergänzte Driessen einen Hinweis, dass Teams, die kontinuierlich ausliefern, etwa bei Webanwendungen, einen einfacheren Workflow erwägen sollten. Teams, die große Änderungen im Trunk vornehmen müssen, nutzen zwei Techniken: Feature-Flags, die unfertiges Verhalten verbergen, und Branch by Abstraction, das eine Schnittstelle vor die alte Implementierung setzt, die neue dahinter baut und umschaltet, sobald sie bereit ist.

An den Rändern sind sich Praktiker wirklich uneinig. Open-Source-Projekte mit vielen nicht vertrauenswürdigen Beitragenden setzen auf längerlebige Pull Requests und Reviews, und regulierte Teams brauchen für Audits manchmal explizite Release-Branches. Das Prinzip, das diese Uneinigkeit übersteht, ist die Batchgröße: Je kleiner jede Änderung, desto leichter lässt sie sich prüfen, testen, deployen, diagnostizieren und rückgängig machen.

```
name: delivery
on:
  push:
    branches: [main]
env:
  IMAGE: registry.example.com/food/orders
jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      digest: ${{ steps.push.outputs.digest }}
    steps:
      - uses: actions/checkout@v4
      - run: make lint test              # fast feedback first
      - id: push
        run: |
          docker build -t "$IMAGE:${{ github.sha }}" .
          docker push "$IMAGE:${{ github.sha }}"
          ref=$(docker inspect --format '{{index .RepoDigests 0}}' "$IMAGE:${{ github.sha
}}")
          echo "digest=${ref#*@}" >> "$GITHUB_OUTPUT"   # the image's sha256 digest
  staging:
    needs: build
    runs-on: ubuntu-latest
    environment: staging
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy.sh staging "$IMAGE@${{ needs.build.outputs.digest }}" && ./smoke-
test.sh staging
  production:
    needs: [build, staging]
    runs-on: ubuntu-latest
    environment: production            # can require an approval
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy.sh production "$IMAGE@${{ needs.build.outputs.digest }}" --canary 5
```

Ein minimaler Continuous-Delivery-Workflow in GitHub-Actions-Syntax: einmal bauen und testen, ein mit dem Commit getaggtes Image pushen, dann genau dieses Image per Digest nach Staging und anschließend als Canary in die Produktion deployen. GitLab CI, Azure Pipelines, Jenkins und andere drücken dieselben Stufen aus. Registry-Login und Deploy-Skripte sind weggelassen.

Deployment-Strategien

Eine Deployment-Strategie entscheidet, wie die neue Version die alte in der Produktion ersetzt, und damit, wie vielen Nutzern eine schlechte Version schaden kann, bevor sie gestoppt wird. Die Strategien unterscheiden sich in der zusätzlichen Infrastruktur, die sie brauchen, darin, wie

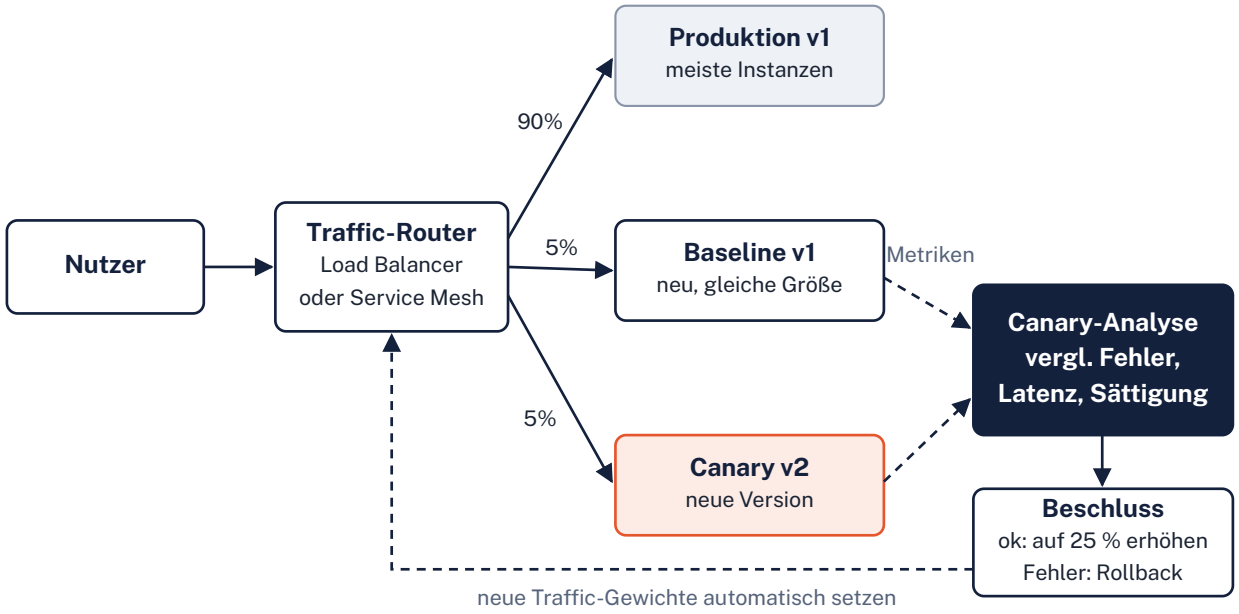
schnell sie zurückrollen, und ob zwei Versionen gleichzeitig laufen. Wann immer zwei Versionen gleichzeitig laufen, müssen sie mit den Daten und Nachrichten der jeweils anderen kompatibel sein; das ist der Preis jeder Strategie außer Recreate.

Deployment-Strategien im Vergleich

Strategie	So funktioniert es	Risiko für Nutzer	Rollback	Zusätzliche Kosten und Komplexität
Recreate	Alle alten Instanzen stoppen, dann die neuen starten.	Ausfallzeit während des Wechsels; jeder Nutzer trifft sofort auf eine schlechte Version.	Die alte Version erneut deployen, mit weiterer Ausfallzeit.	Keine. Akzeptabel für interne Tools, Batch-Jobs und Wartungsfenster.
Rolling	Instanzen jeweils zu wenigen ersetzen (Standard von Kubernetes für Deployments, wobei maxSurge und maxUnavailable beide standardmäßig 25 % betragen).	Die Betroffenheit wächst mit dem Fortschritt des Rollouts; Health Checks erkennen Abstürze, aber keine subtilen Fehler.	Rollback, indem die alte Version erneut ausgerollt wird; das dauert so lange wie der Rollout.	Gering. Alte und neue Version laufen nebeneinander, müssen also kompatibel sein.
Blue-Green	Eine vollständige neue Umgebung (Green) neben der Live-Umgebung (Blue) betreiben, sie testen und dann den gesamten Traffic am Router umschalten.	Alle Nutzer wechseln auf einmal, aber die neue Umgebung wurde vor dem Umschalten geprüft.	Schnell: den Router zurückschalten, solange Blue noch läuft.	Bis zu doppelte Kapazität während des Wechsels; gemeinsame Datenbanken müssen mit beiden Versionen funktionieren.
Canary	Einen kleinen Anteil des Traffics (etwa 1–5 %) an die neue Version senden, ihre Metriken vergleichen und dann schrittweise ausweiten.	Nur der Nutzeranteil des Canary trifft auf eine schlechte Version, und nur, bis die Analyse sie stoppt.	Schnell: das Gewicht des Canary auf null setzen.	Braucht gewichtetes Routing, gute Metriken und genug Traffic für einen aussagekräftigen Vergleich.
Shadow (Dark Launch)	Echte Anfragen an die neue Version kopieren; ihre Antworten werden verglichen oder verworfen, nie zurückgegeben.	Direkt keines, sofern Seiteneffekte (Zahlungen, E-Mails, Schreibzugriffe) unterdrückt oder simuliert werden.	Nichts zurückzurollen; Nutzer waren nie davon abhängig.	Doppelte Last auf dem gespiegelten Pfad und sorgfältiger Umgang mit Seiteneffekten; am besten für Lesepfade und Neuentwicklungen.

Das Blue-Green-Deployment beschrieb Martin Fowler 2010 und nannte Dan North und Jez Humble als Urheber; das Canary Release, benannt nach den Kanarienvögeln, die Bergleute zum Aufspüren von Gas mitnahmen, beschrieb Danilo Sato 2014 auf derselben Website. In Kubernetes sind Rolling Updates eingebaut; Blue-Green- und Canary-Releases kommen meist von einem Progressive-Delivery-Controller (Argo Rollouts, Flagger) oder von einem Service Mesh und dem gewichteten Routing der Gateway API. Auf verwalteten Plattformen heißen die entsprechenden Funktionen Traffic Splitting oder Deployment Slots.

Canary Release mit automatisierter Analyse



Der Vergleich des Canary mit einer frischen Baseline der alten Version statt mit der lange laufenden Flotte beseitigt Unterschiede, die nur durch Laufzeit und warme Caches entstehen.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: eine schlechte Version, vier Arten, sie zu deployen

Die schwierigste Idee dieses Abschnitts ist, dass die Deployment-Strategie entscheidet, wie viele Kunden eine schlechte Version erreicht, bevor sie gestoppt wird, und wie schnell sie sich rückgängig machen lässt. Verfolgen Sie eine fehlerhafte Version des Bestellservice durch vier Strategien.

- Die neue Version hat einen subtilen Fehler: Sie stürzt nicht ab, berechnet aber bei manchen Bestellungen die falsche Liefergebühr. Die Health Checks bestehen, weil der Prozess selbst gesund ist.
- Recreate: Alle alten Kopien stoppen, dann starten die neuen. Nach einem kurzen Ausfall trifft jeder Kunde auf einmal auf die fehlerhafte Version. Rückgängig machen heißt, die alte Version erneut zu deployen, mit einem weiteren Ausfall.
- Rolling: Kopien werden jeweils zu wenigen ersetzt. Da nichts abstürzt, lassen die Health Checks den Rollout weiterlaufen, bis jede Kopie fehlerhaft ist. Das Zurückrollen dauert so lange wie das Ausrollen.
- Blue-Green: Die neue Version läuft als vollständige zweite Umgebung und wird getestet, bevor der gesamte Traffic auf sie umgeschaltet wird. Übersehen die Tests den Fehler, wechselt jeder Kunde auf einmal, doch das Rückgängigmachen geht schnell: den Router auf die alte, noch laufende Umgebung zurückschalten.
- Canary: 5 % des Traffics gehen an die neue Version, und ihre Metriken werden mit einer frischen Kopie der alten Version verglichen. Bezieht der Vergleich ein Geschäftssignal wie die berechneten Gebühren ein, kann er den Unterschied erkennen, und der Anteil des Canary wird auf null gesetzt. Nur Kunden in diesen 5 % trafen auf den Fehler, und nur, bis die Analyse ihn stoppte.
- **Die Lehre:** Wählen Sie eine Strategie danach, wie viele Kunden Sie einem Risiko aussetzen können und wie schnell Sie einen Fehler rückgängig machen müssen, und sorgen Sie dafür, dass Ihre Prüfungen die Ausfälle sehen, die Sie fürchten, nicht nur Abstürze.

Deployment und Release mit Feature-Flags trennen

Deployment installiert eine neue Version der Software in einer Umgebung; **Release** macht ein Verhalten für Nutzer verfügbar. Behandelt man beides als ein Ereignis, wird jedes Deployment zu einem Geschäftsereignis, mit all der Nervosität, die das mit sich bringt. Feature-Flags (Feature Toggles) trennen die beiden: Der neue Code wird ausgeschaltet deployt, und das Verhalten wird später durch eine Konfigurationsänderung freigegeben, schrittweise und

umkehrbar. Pete Hodgsons Artikel *Feature Toggles (aka Feature Flags)* auf martinfowler.com (2017) unterscheidet vier Kategorien danach, wie lange sie leben und wie dynamisch sie entscheiden.

Hodgsons vier Kategorien von Feature Toggles

Kategorie	Zweck	Typische Lebensdauer	Wie dynamisch
Release-Toggles	Unfertige oder unveröffentlichte Features im Trunk ausliefern, ausgeschaltet.	Tage bis wenige Wochen; nach dem Release entfernen.	Meist statisch pro Release.
Experiment-Toggles	A/B- oder multivariate Tests, die jeden Nutzer einer Kohorte zuordnen.	Lange genug für statistisch belastbare Ergebnisse, dann entfernt.	Pro Anfrage, nach Nutzerkohorte.
Ops-Toggles	Betriebliche Steuerungen, etwa das Abschalten eines teuren Empfehlungsbereichs unter Last (Kill Switches).	Meist kurz; einige Kill Switches bleiben langfristig erhalten.	Zur Laufzeit geändert, oft während eines Incidents.
Permissioning-Toggles	Features für bestimmte Nutzer: Premium-Stufen, Betatester, interne Mitarbeitende.	Können jahrelang bestehen.	Pro Anfrage, nach Nutzer.

Flags verursachen laufende Kosten. Jedes Flag verdoppelt die Pfade durch den Code, den es schützt, erhöht die Zahl der Testkombinationen und hinterlässt toten Code, sobald seine Entscheidung feststeht. Hodgson empfiehlt, Flags wie einen Lagerbestand zu behandeln: Beim Anlegen eines Release-Toggles eine Aufgabe zum Entfernen anlegen, Flags ein Ablaufdatum geben, Tests ergänzen, die fehlschlagen, sobald ein Flag sein Datum überschritten hat, und begrenzen, wie viele gleichzeitig existieren dürfen. Der Verlust von Knight Capital im Jahr 2012, beschrieben in Abschnitt 1.3, ging auf ein umfunktioniertes Flag zurück, das alten Code auf einem Server reaktivierte, der bei einem manuellen Deployment übersehen worden war: eine Mahnung, Flag-Namen nie wiederzuverwenden und toten Code hinter entschiedenen Flags zu löschen. Im großen Maßstab werden Flags von einem eigenen Dienst bereitgestellt (Open-Source-Optionen wie Unleash und Flagsmith, kommerzielle wie LaunchDarkly oder die herstellerneutrale OpenFeature-API), damit sich eine Änderung in Sekunden verbreitet und protokolliert wird.

Datenbankänderungen in der Pipeline

Code lässt sich in Sekunden zurückrollen, Daten nicht. Ein Deployment, das eine Spalte umbenennt und dann zurückgerollt wird, lässt den alten Code vor einem Schema zurück, das er nicht versteht. Die Antwort, ausführlich behandelt unter Schemaevolution, besteht darin, jede Schemaänderung abwärtskompatibel zur gerade laufenden Codeversion zu machen, mit Expand and Contract: zuerst die neue Struktur hinzufügen, Code deployen, der beide schreibt und jede liest, Daten nachfüllen (Backfill), die Leser umstellen und die alte Struktur erst in einem späteren Release entfernen, wenn nichts sie mehr nutzt.

In der Pipeline heißt das: Migrationen sind versionierte Artefakte, die als eigener Schritt laufen, bevor der Code deployt wird, der sie braucht, und sie werden an einer Kopie der Daten in Produktionsgröße getestet, damit eine Sperre auf einer großen Tabelle im Staging entdeckt wird statt zur Mittagszeit. Destruktive Schritte (Spalten oder Tabellen löschen) werden verzögert und separat ausgeführt, weil sie die einzige Art von Änderung sind, die ein Rollback nicht rückgängig machen kann. Eine nützliche Pipeline-Prüfung lässt den Build fehlschlagen, wenn eine Migration nicht additiv ist, es sei denn, sie ist ausdrücklich als Contract-Schritt gekennzeichnet.

Progressive Delivery und automatisierte Canary-Analyse

Progressive Delivery, ein Begriff, den James Governor vom Analystenhaus RedMonk 2018 populär machte, verbindet Canaries, Feature-Flags und automatisierte Prüfung: Eine Änderung erreicht schrittweise größere Zielgruppen, und bei jedem Schritt entscheidet das System, nicht eine Person vor Dashboards, ob es weitergeht. Die Entscheidung nutzt die automatisierte Canary-Analyse: Die Metriken des Canary (Fehlerrate, Latenzperzentile, Sättigung, Geschäftssignale wie aufgegebene Bestellungen) werden statistisch mit einer Baseline verglichen. Netflix und Google haben ihre Analyse-Engine Kayenta 2018 als Open Source veröffentlicht; Argo Rollouts und Flagger bieten in Kubernetes ähnliche Analyseschritte.

Drei Details entscheiden, ob man der Analyse trauen kann. Vergleichen Sie Gleiches mit Gleichem: Eine frische Baseline der alten Version, die denselben Traffic-Anteil erhält, verhindert, dass der neuen Version Unterschiede angelastet werden, die von kalten Caches oder kürzlichen Neustarts herrühren. Sorgen Sie für genug Traffic: Bei 5 % eines Service mit zehn Anfragen pro Minute sieht eine Stunde Canary dreißig Anfragen, zu wenige, um eine Fehlerrate von 1 % zu erkennen; Services mit wenig Traffic brauchen daher längere Schritte, synthetische Last oder eine andere Strategie. Wählen Sie Metriken, die an die Service Level Objectives und an Geschäftsergebnisse gebunden sind, denn eine Version kann schnell und fehlerfrei sein und dabei still die falsche Liefergebühr berechnen.

Rollback oder Roll-forward

Wenn ein Release schiefgeht, gibt es zwei Auswege. Ein Rollback kehrt zum vorherigen Artefakt zurück; das ist schnell und braucht keinen neuen Code, sofern die vorherige Version noch mit den aktuellen Daten und den Versionen anderer Services funktioniert. Ein Roll-forward liefert eine Korrektur über die Pipeline aus; das ist langsamer, aber manchmal die einzige Option, etwa nach einer nicht umkehrbaren Datenmigration oder einer Änderung, von der andere Services bereits abhängen.

Eine vernünftige Grundregel lautet „erst zurückrollen, später diagnostizieren“: den Dienst mit der schnellsten sicheren Maßnahme wiederherstellen und dann ohne Druck untersuchen. Ein Roll-forward unter Druck umgeht tendenziell die Prüfungen der Pipeline, und eine überhastete Korrektur eines Produktionsincidents ist eine klassische Quelle des nächsten Incidents. Ein Rollback ist nur dann eine echte Option, wenn er eingeplant ist: frühere Artefakte aufbewahrt, Schemaänderungen für mindestens eine Version in beide Richtungen kompatibel, Nachrichten für alte und neue Consumer lesbar und der Rollback-Pfad selbst regelmäßig geübt. Ein Team, das noch nie zurückgerollt hat, sollte davon ausgehen, dass es das nicht kann.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine neue Liefergebührenregel ausliefern

Der Bestellservice der Essenslieferplattform ändert die Berechnung der Liefergebühren: Statt einer Pauschale zählen nun Entfernung und Nachfrage. Das Team nutzt Trunk-Based Development, ein Canary-Deployment und ein Release-Flag.

Änderung	Neuer Gebührenrechner hinter einem Release-Flag `dynamic-delivery-fee`, standardmäßig aus; eine reine Expand-Migration fügt eine nullable Spalte `fee_rule_version` hinzu.
Vom Commit zum Artefakt	Um 10:02 in main gemergt. Lint, Unit-Tests und Scans dauern 6 Minuten; das Image `orders:4f1c2ab` wird mit SBOM und Signatur um 10:09 gepusht.
Migration	Die additive Migration läuft um 10:15 als eigener Pipeline-Schritt. Die laufende Version ignoriert die neue Spalte, also bricht nichts.
Staging	Dasselbe Image wird um 10:16 deployt; Vertragstests mit dem Zahlungs- und dem Restaurantservice sowie Smoke-Tests sind um 10:31 bestanden.
Canary	Um 10:35 gehen 5 % des Produktionstraffics an die neue Version, noch mit ausgeschaltetem Flag. Eine Analyse über 20 Minuten vergleicht p99-Latenz, 5xx-Rate und Bestellerfolg mit einer frischen Baseline: bestanden.
Rollout	Die Gewichte steigen in 15-Minuten-Schritten auf 25 %, 50 % und 100 %; abgeschlossen um 11:40. Das Deployment ist fertig, und doch hat noch kein Kunde die neue Gebühr gesehen.
Release	Am nächsten Tag aktiviert die Produktmanagerin das Flag für 10 % der Kunden in einer Stadt, beobachtet Bestellkonversion und Beschwerden und weitet es dann im Laufe einer Woche auf alle Städte aus.
Aufräumen	Das Flag und der Pauschalgebühren-Code werden im folgenden Sprint gelöscht; eine spätere Contract-Migration macht die Spalte zur Pflicht.

Deployment-Risiko (funktioniert der Code?) und Release-Risiko (mögen die Kunden die neuen Gebühren?) wurden getrennt behandelt, jedes mit eigenem kleinem Blast Radius.

Bei jedem Schritt bis zur Contract-Migration war die Wiederherstellung eine Konfigurationsänderung oder ein Router-Gewicht, kein Notfall-Fix. Vom Commit bis zum vollständigen Produktions-Deployment vergingen etwa 1 Stunde 40 Minuten.

Die Pipeline messen

Die in Abschnitt 2.6 eingeführten DORA-Metriken für die Softwareauslieferung (Change Lead Time, Deployment-Häufigkeit, Change Fail Rate, Wiederherstellungszeit nach fehlgeschlagenem Deployment und Deployment-Rework-Rate) sind größtenteils Eigenschaften von Pipeline und Architektur zusammen. Kleine Batches und Trunk-Based Development verkürzen die Lead Time und erhöhen die Häufigkeit; automatisierte Tests und Canary-Analyse senken die Change Fail Rate; schnelles Rollback und Flags verkürzen die Wiederherstellungszeit. Messen Sie sie pro Service, beobachten Sie Trends statt Einzelwerte, und machen Sie sie nie zu Zielvorgaben für Einzelpersonen; das lädt nur zum Schönen der Zahlen ein.

ACHTUNG

Häufige Pipeline-Fehler

- Das Artefakt für jede Umgebung neu bauen, sodass in Produktion etwas läuft, das nie getestet wurde.
- Eine Commit-Stufe, die 40 Minuten dauert, sodass Entwickler Änderungen bündeln, um nicht zu warten, was den Zweck vereitelt.
- Instabile Tests, die so lange wiederholt werden, bis sie grün sind, und so allen beibringen, rote Builds zu ignorieren.
- Manuelle Freigabeschritte in jeder Stufe, die Tage an Wartezeit hinzufügen, ohne Informationen hinzuzufügen.
- Migrationen, die in den Anwendungsstart eingebaut sind, sodass jede Instanz um ihre Ausführung wetteifert und ein Rollback auf ein inkompatibles Schema trifft.
- Feature-Flags, die nie entfernt werden und Hunderte ungetesteter Kombinationen hinterlassen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum ist es ein Problem, das Artefakt für die Produktion separat neu zu bauen, selbst wenn der Quellcode identisch ist?

Abhängigkeiten, Basis-Images oder Build-Werkzeuge können sich zwischen den Builds geändert haben, sodass das Produktionsartefakt nicht das getestete ist; einmal bauen und dasselbe Artefakt weiterreichen.

▼ Was ist der Unterschied zwischen dem Deployment eines Features und seinem Release?

Das Deployment installiert den Code in einer Umgebung; das Release macht das Verhalten für Nutzer verfügbar. Mit Feature-Flags lässt sich der Code ausgeschaltet deployen und später schrittweise und umkehrbar freigeben.

▼ Warum liefert ein Canary bei einem Service mit wenig Traffic oft unzuverlässige Ergebnisse?

Ein kleiner Anteil von wenig Traffic ergibt zu wenige Anfragen, um Unterschiede in Fehlerrate oder Latenz mit statistischer Sicherheit zu erkennen; daher müssen die Schritte länger sein, der Traffic synthetisch oder eine andere Strategie gewählt werden.

▼ Wann kann ein Roll-forward die einzige Option sein?

Wenn das Release eine nicht umkehrbare Änderung enthielt, etwa eine destruktive Migration, oder andere Services bereits vom neuen Verhalten abhängen, sodass die vorherige Version nicht mehr korrekt laufen kann.

▼ Probieren Sie es aus: Ein Team baut das Image für jede Umgebung neu, deployt, indem es zur Mittagszeit alles anhält und neu startet, und die neue Version entfernt im selben Release eine Datenbankspalte. Finden Sie drei Probleme, und nennen Sie einen besseren Ansatz.

Neu zu bauen heißt, dass in der Produktion etwas läuft, das nie getestet wurde; bauen Sie also einmal, und reichen Sie ein unveränderliches Artefakt weiter. Recreate zur Mittagszeit verursacht zur Spitzenzeit Ausfallzeit, verwenden Sie also ein Rolling oder Canary Deployment mit automatischem Vergleich gegen eine Baseline. Die Spalte im Release zu löschen macht ein Rollback unmöglich, und alte und neue Version können nicht zusammen laufen; verwenden Sie also Expand and Contract, und entfernen Sie sie in einem späteren Release. Ergänzen Sie ein Feature-Flag, um Deployment und Release zu trennen.

QUELLEN

- Pete Hodgson: Feature Toggles (aka Feature Flags), martinfowler.com: https://martinfowler.com/articles/feature-toggles.html
- Martin Fowler: BlueGreenDeployment: https://martinfowler.com/bliki/BlueGreenDeployment.html
- Danilo Sato: CanaryRelease: https://martinfowler.com/bliki/CanaryRelease.html
- Trunk Based Development: https://trunkbaseddevelopment.com/
- DORA: software delivery performance metrics: https://dora.dev/guides/dora-metrics-four-keys/

Multi-Region, Disaster Recovery und Edge Computing

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.5 Verfügbarkeit und Zuverlässigkeit
- 6.6 CAP und Konsistenz
- 7.4 Replikation und Partitionierung

Begriffe von früher, die er verwendet: Failover, Replikationsverzögerung (Replication Lag), PACELC, Blast Radius

DIE IDEE IN KÜRZE

Eine Ladenkette, die ihren gesamten Bestand in einem Lager hält, hat ein Problem, wenn dieses Lager überflutet wird. Sie könnte nur eine Bestandsliste und einen Ersatzschlüssel für ein leeres Gebäude anderswo bereithalten (günstig, aber langsam in der Wiederherstellung), ein kleines zweites Lager bestückt und besetzt halten (teurer, aber schnell) oder ständig zwei volle Lager betreiben (am teuersten, fast ohne Unterbrechung). Sie kann außerdem kleine Kioske mit den beliebtesten Artikeln in Kundennähe aufstellen. Multi-Region-und Edge-Design treffen dieselben Entscheidungen für Software: wie viel man im Voraus bezahlt, gemessen daran, wie lange Kunden warten können.

WAS SIE WISSEN MÜSSEN

- Betreiben Sie mehrere Regionen nur wegen Latenz, Disaster Recovery oder Datenresidenz; mehrere Availability Zones in einer Region sind die normale Grundausstattung und viel günstiger.
- RTO (wie lange Sie ausfallen dürfen) und RPO (wie viele Daten Sie verlieren dürfen) kommen aus dem Geschäft und bestimmen die Disaster-Recovery-Strategie und ihre Kosten.
- Die AWS-Well-Architected-Strategien Backup and Restore, Pilot Light, Warm Standby und Multi-Site Active-Active erkaufen schrittweise niedrigere RTO und RPO mit höheren laufenden Kosten und mehr Komplexität.
- Regionsübergreifende Daten sind der schwierige Teil: Synchrone Replikation fügt Schreibzugriffen lange Round Trips hinzu, die Verzögerung asynchroner Replikation wird zum tatsächlichen RPO, und Replikation kopiert auch Löschungen und Datenkorruption, daher bleiben Backups unverzichtbar.

- Ein Failover wird über DNS oder globale Anycast-Load-Balancer gesteuert; statisch stabile Designs halten vorab bereitgestellte Kapazität vor und vermeiden die Abhängigkeit von der Control Plane der ausgefallenen Region.
- Zellbasierte Architektur und Shuffle Sharding begrenzen den Blast Radius schlechter Deployments, Überlastungen und schädlicher Eingaben, unter denen eine zweite Region ebenfalls leiden würde.
- CDNs und Edge Functions, etwa Cloudflare Workers in V8-Isolates, bedienen Lesezugriffe und leichtgewichtige Logik in Nutzernähe, während die führenden Systeme (Systems of Record) meist in einer Region bleiben.

Warum in mehr als einer Region betreiben

Abschnitt 0.7 (Server und die Cloud) hat Regionen (getrennte geografische Gebiete) und Availability Zones (unabhängige Rechenzentrumsgruppen innerhalb einer Region) eingeführt, und Abschnitt 10.1 hat gezeigt, wie Anbieter sie organisieren. Der Betrieb über mehrere Availability Zones in einer Region ist die normale Grundausstattung für jedes Produktivsystem: Er übersteht den Verlust eines Rechenzentrums mit geringen Mehrkosten, weil Zonen nah genug für synchrone Replikation beieinanderliegen. Der Betrieb über mehrere *Regionen* ist ein viel größerer Schritt, und es gibt nur drei gute Gründe dafür.

Latenz. Licht in Glasfaser legt rund 200.000 km pro Sekunde zurück, daher kann ein Round Trip zwischen London und Sydney (etwa 17.000 km Luftlinie) nicht weniger als etwa 170 ms dauern, und reale Routen brauchen deutlich länger. Eine Anfrage, die mehrere Round Trips zu einer entfernten Region braucht, fühlt sich langsam an, wie schnell die Server auch sein mögen. Nutzer aus einer nahen Region oder von der Edge zu bedienen, ist die einzige Abhilfe.

Disaster Recovery. Ganze Regionen fallen tatsächlich aus, meist durch eine gemeinsame Abhängigkeit statt durch eine physische Katastrophe. Am 19. und 20. Oktober 2025 hinterließ eine Race Condition in der Automatisierung, die DNS-Einträge für Amazon DynamoDB verwaltet, für dessen Endpoint in US-EAST-1 einen leeren DNS-Eintrag; die DynamoDB-Fehler dauerten etwa drei Stunden, doch Folgeausfälle beim Starten neuer virtueller Maschinen und bei den Health Checks der Load Balancer zogen sich bis in den folgenden Nachmittag. Jeder große Anbieter hatte vergleichbare regionale Incidents. Disaster Recovery plant für sie.

Datenresidenz. Anforderungen an die Datenresidenz aus Gesetzen, von Aufsichtsbehörden oder aus Kundenverträgen können verlangen, dass bestimmte Daten in bestimmten Rechtsräumen gespeichert oder verarbeitet werden. Die DSGVO selbst beschränkt Übermittlungen personenbezogener Daten aus dem Europäischen Wirtschaftsraum, statt eine

lokale Speicherung zu verlangen, doch Branchenregeln, nationale Gesetze und Unternehmensverträge gehen oft weiter (siehe Privacy by Design). Residenz erzwingt oft Multi-Region-Designs, in denen Daten bewusst *nicht* überallhin repliziert werden.

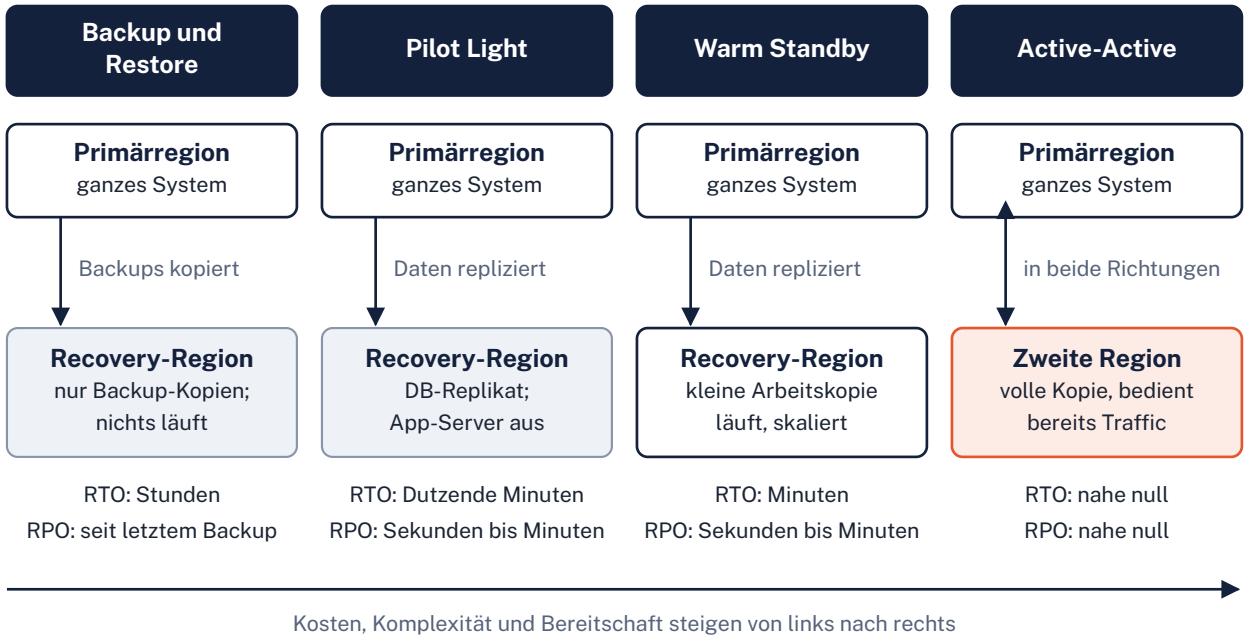
Multi-Region ist teuer: doppelte Infrastruktur, Gebühren für regionsübergreifende Datenübertragung, schwierigere Datenkonsistenz sowie komplexerer Betrieb und komplexere Tests. Viele Teams, die glauben, es zu brauchen, brauchen eigentlich ein gutes Multi-Zonen-Design, getestete Backups und ein CDN.

Wiederherstellungsziele und die vier Strategien

Abschnitt 2.5 (Verfügbarkeit und Zuverlässigkeit) hat die zwei Zahlen definiert, die jedes Disaster-Recovery-Design bestimmen. Das Recovery Time Objective (RTO) gibt an, wie lange der Dienst nicht verfügbar sein darf; das Recovery Point Objective (RPO) gibt an, wie viele aktuelle Daten verloren gehen dürfen, ausgedrückt als Zeitspanne. Beide kommen aus dem Geschäft, idealerweise aus den Kosten einer Stunde Ausfall und einer verlorenen Bestellung, nicht aus den Vorlieben der Technik. Niedrigere Werte kosten mehr, und die Kosten steigen nahe null steil an.

Das AWS-Well-Architected-Whitepaper *Disaster Recovery of Workloads on AWS: Recovery in the Cloud* beschreibt vier Strategien in der Reihenfolge steigender Kosten und Komplexität. Die Namen sind weit über AWS hinaus gebräuchlich und gelten ebenso für Azure, Google Cloud oder ein Paar privater Rechenzentren.

Disaster-Recovery-Strategien



Jeder Schritt nach rechts hält mehr vom System in der zweiten Region am Laufen und erkauft niedrigere RTO und RPO mit höheren laufenden Kosten.

Disaster-Recovery-Strategien im Vergleich

Strategie	Was in der zweiten Region läuft	Typisches RTO	Typisches RPO	Relative Kosten und Hauptrisiko
Backup and Restore	Nichts außer Backup-Kopien (Datenbank-Snapshots, Objektspeicher) und dem Infrastrukturcode für den Wiederaufbau.	Stunden, je nach Datenmenge und Automatisierung.	Das Backup-Intervall (Minuten bei kontinuierlichen Log-Backups, Stunden bei täglichen Snapshots).	Am niedrigsten. Wiederherstellungen sind langsam und oft ungetestet; ohne Infrastructure as Code ist das RTO eine Fiktion.
Pilot Light	Replizierte Datenbanken und Kerninfrastruktur; Anwendungsserver definiert, aber nicht laufend.	Dutzende Minuten: die Anwendungsschicht starten und skalieren, dann den Traffic umleiten.	Sekunden bis Minuten (Replikationsverzögerung).	Niedrig. Das Hochskalieren während einer regionalen Krise konkurriert mit allen anderen um Kapazität.
Warm Standby	Eine verkleinerte, aber voll funktionsfähige Kopie des gesamten Systems.	Minuten: Traffic umleiten, dann hochskalieren.	Sekunden bis Minuten.	Mittel. Der Standby lässt sich mit echtem Traffic testen, driftet aber ab, wenn er nicht mit jedem Release deployt wird.
Multi-Site Active-Active	Eine vollständige Kopie, die in jeder Region Live-Traffic bedient.	Nahe null: Der Traffic wird umverteilt, kein Failover nötig.	Nahe null für replizierte Daten; Datenkorruption braucht weiterhin Backups.	Am höchsten. Erfordert ein Datendesign für gleichzeitige Schreibzugriffe in mehreren Regionen.

Das Whitepaper selbst nennt für jede Strategie einen groben Wert, der RTO und RPO zusammen abdeckt; die Tabelle trennt sie, weil bei kontinuierlicher Replikation das RPO von Pilot Light und Warm Standby durch die Replikationsverzögerung bestimmt wird und nicht dadurch, wie viel von der Anwendung läuft. Zwei Warnungen gelten für jede Spalte. Replikation ist kein Backup: Eine versehentliche Löschung, eine fehlerhafte Migration oder Ransomware wird binnen Sekunden getreu repliziert, daher sind Point-in-Time-Backups, idealerweise unveränderlich und in einem separaten Konto, selbst in einem Active-Active-Design nötig. Und eine Strategie, die nicht geprobt wurde, ist eine Hoffnung, kein Plan. Teams, die mindestens

einige Male im Jahr eine regionale Failover-Übung durchführen, entdecken fehlende Berechtigungen, fest eincodierte Endpoints, abgelaufene Zertifikate und zu kleine Quotas in der Recovery-Region, solange es noch billig ist, sie zu finden.

Daten über Regionen replizieren

Zustandslose Anwendungsserver lassen sich leicht an zwei Orten betreiben; die Daten sind der schwierige Teil, und die Abwägungen sind die von Konsistenz, CAP und PACELC über kontinentale Entfernungen. Synchrone Replikation über Regionen bedeutet, dass jeder Schreibzugriff auf mindestens einen langen Round Trip wartet, vielleicht 70 bis 150 ms oder mehr; das ist für wenige kritische Schreibzugriffe akzeptabel und ruinös für gesprächige. Asynchrone Replikation hält Schreibzugriffe schnell, aber das Replikat hinkt hinterher, und was beim Ausfall der Primärregion noch nicht angekommen ist, geht verloren: Die Replikationsverzögerung *ist* das tatsächliche RPO.

Es gibt vier gängige Datendesigns. **Eine einzige Primärregion**: Alle Schreibzugriffe gehen an eine Region, und Replikate anderswo bedienen Lesezugriffe und die Wiederherstellung; einfach, aber entfernte Nutzer zahlen Schreiblatenz, und ein Failover bedeutet, ein Replikat zu befördern (mit den Split-Brain-Risiken, die unter Replikation und Partitionierung beschrieben sind). **Heimatregion pro Entität**: Jeder Kunde, jedes Restaurant oder jeder Mandant hat eine Heimatregion, in der seine Daten geschrieben werden; das hält Schreibzugriffe lokal und unterstützt die Residenz, um den Preis des Routings und seltener regionsübergreifender Operationen. **Mehrere Schreiber mit Konfliktauflösung**: Jede Region nimmt Schreibzugriffe an, und Konflikte werden zusammengeführt, etwa per Last-Writer-Wins (der Standard bei DynamoDB Global Tables, die seit 2025 auch starke Konsistenz über mehrere Regionen bieten) oder mit konfliktfreien replizierten Datentypen; sicher nur für Daten, bei denen ein Zusammenführen sinnvoll ist, etwa ein Warenkorb, und gefährlich für Geld oder Lagerbestand. **Global konsistente Datenbanken** wie Google Spanner oder CockroachDB zahlen bei Schreibzugriffen Konsens-Latenz, um starke Konsistenz über Regionen hinweg zu bieten; bei Azure Cosmos DB wählt jede Anwendung unter fünf Konsistenzstufen, von stark bis eventual.

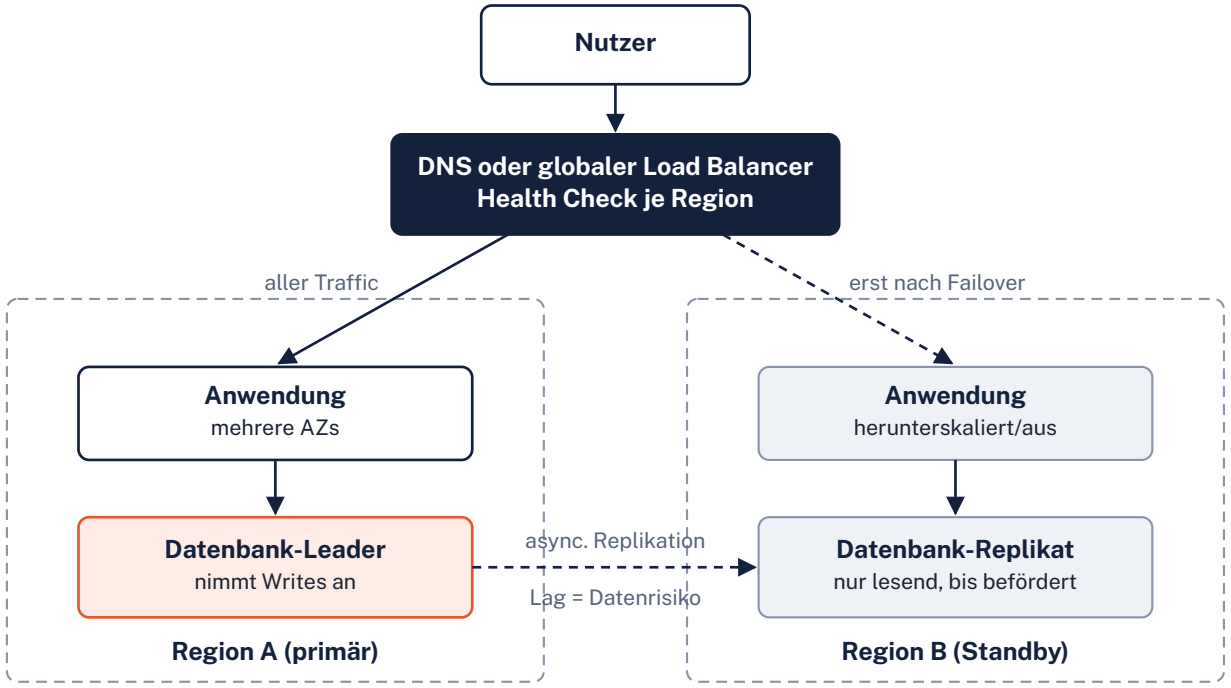
Die meisten realen Systeme mischen diese pro Datenbestand, genau wie Abschnitt 6.6 empfiehlt: Zahlungen und Bestellstatus in einer einzigen Primär- oder Heimatregion, Speisekarten und Restaurantdetails für Lesezugriffe überallhin repliziert und Analysedaten asynchron übertragen.

Failover, DNS und statische Stabilität

Traffic wird per DNS oder über einen globalen Load Balancer zwischen Regionen gesteuert. Beim DNS-basierten Failover prüft der DNS-Dienst jede Region per Health Check und liefert die Adresse einer gesunden zurück; seine Schwäche ist das Caching, weil Resolver und Clients

alte Antworten länger halten können als die TTL des Eintrags. Globale Load Balancer (etwa AWS Global Accelerator, Azure Front Door, der globale externe Load Balancer von Google Cloud, Cloudflare) nutzen Anycast-Adressen, die von vielen Standorten aus angekündigt werden, sodass Clients eine Adresse behalten und der Anbieter den Traffic binnen Sekunden verlagert.

Failover zwischen einer Primär- und einer Standby-Region



Was beim Ausfall von Region A noch nicht repliziert ist, geht verloren; daher bestimmt die Replikationsverzögerung den tatsächlichen Wiederherstellungspunkt.

Zu entscheiden, *wann* ein Failover ausgelöst wird, ist schwieriger als der Mechanismus. Ein automatischer Failover reagiert am schnellsten, kann aber durch einen Monitoring-Fehler oder eine kurze Netzwerkstörung ausgelöst werden, und der Failover einer Datenbank birgt das Risiko, dass zwei Primärknoten Schreibzugriffe annehmen. Viele Organisationen automatisieren daher die Erkennung und die Failover-Schritte, behalten aber eine menschliche Entscheidung für den Start eines regionalen Failovers bei, mit einem Runbook und einem Zeitlimit. Auch das Zurückschwenken nach dem Incident, einschließlich der Neusynchronisierung von Daten, die in der Recovery-Region geschrieben wurden, braucht einen Plan.

Der Incident vom Oktober 2025 veranschaulicht ein Prinzip, das die AWS Builders' Library statische Stabilität nennt: Ein System sollte während eines Ausfalls weiterarbeiten, ohne in diesem Moment Änderungen vornehmen zu müssen. Systeme, die bereits genug Kapazität am Laufen hatten, überstanden das Problem; Systeme, die neue Instanzen starten mussten oder

deren Wiederherstellung von Control-Plane-Aktionen in der betroffenen Region abhing, konnten sich erst erholen, als die Region es tat. Für Disaster Recovery spricht das für vorab bereitgestellte Kapazität (Warm Standby statt Pilot Light für kritische Pfade), für einen Failover, der nur von der Data Plane der gesunden Region abhängt, und für DNS, Identitäten und Secrets, die nicht ausschließlich in der Region liegen, aus der Sie fliehen.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: wiederherstellen, ohne um etwas zu bitten

Die Idee, die hier die meiste Sorgfalt verlangt, ist die statische Stabilität: Ein Wiederherstellungsplan ist nur so gut wie die Dinge, die er im Moment des Ausfalls braucht, und der sicherste Plan braucht nichts Neues. Vergleichen Sie zwei Arten, eine zweite Region für den Bestellkern der Essenslieferplattform vorzubereiten.

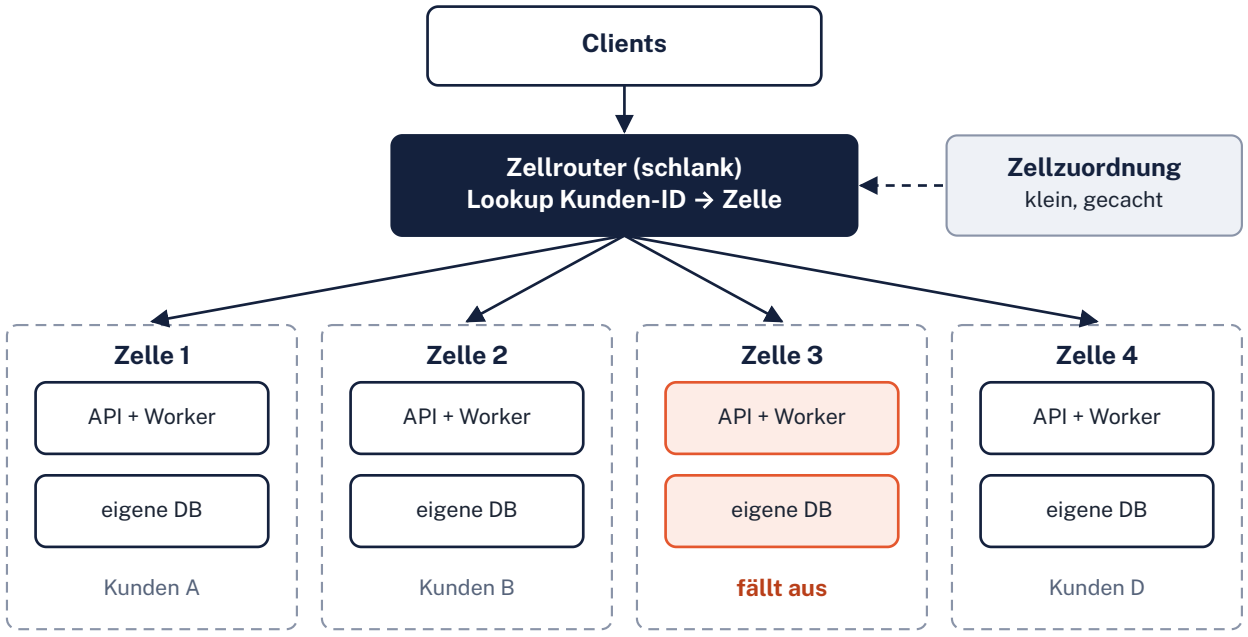
- Bei einem Pilot Light hält die zweite Region eine laufend aktualisierte Kopie der Daten und der Kerninfrastruktur, aber die Anwendungsserver sind aus. Bei einem Warm Standby läuft dort bereits eine kleine Arbeitskopie des Bestellkerns, die etwas echten Traffic erhält.
- Zur Abendessenszeit fällt die Primärregion aus. Bei Pilot Light bedeutet Wiederherstellung, jetzt Änderungen vorzunehmen: die Anwendungsserver starten, hochskalieren und dann den Traffic umleiten. Jeder Start ist eine Anfrage an die Control Plane des Anbieters, die Maschinerie, die Ressourcen anlegt und ändert.
- Das ist der schlechteste Moment, um zu fragen. Andere Kunden wollen zur selben Zeit Kapazität, und jeder Schritt, der auf etwas angewiesen ist, das nur in der ausgefallenen Region liegt, etwa DNS, Identitäten oder Secrets, muss warten, bis sich diese Region erholt.
- Bei Warm Standby läuft die Kopie bereits und hat sich mit echtem Traffic bewährt. Der Traffic wird sofort auf sie umgeleitet; sie bedient mit reduzierter Kapazität, während sie hochskaliert. Geht das Skalieren langsam, erhalten Kunden einen eingeschränkten Dienst statt gar keinen.
- Statische Stabilität kostet Geld: Eine ständig laufende Kopie kostet jeden Monat mehr als die abgeschalteten Server von Pilot Light, wie das Ausgearbeitete Beispiel weiter unten in diesem Abschnitt zeigt.
- **Die Lehre:** Bevorzugen Sie für kritische Pfade eine Wiederherstellung, die sich nur auf das stützt, was in der gesunden Region bereits läuft. Alles, was während der Katastrophe angelegt oder geändert werden muss, ist im Ernstfall womöglich nicht verfügbar.

Zellen: den Blast Radius begrenzen

Regionen schützen vor regionalen Katastrophen; die meisten Ausfälle werden jedoch durch die eigenen Änderungen des Systems, Überlastungen und schädliche Eingaben verursacht, unter denen eine zweite Region mit demselben Code und derselben Konfiguration ebenfalls leidet. Die zellbasierte Architektur begrenzt den Blast Radius solcher Ausfälle, indem sie viele vollständige, unabhängige Kopien des Systems betreibt, Zellen genannt, die jeweils eine feste Teilmenge der Kunden bedienen. Ein schlanker Router ordnet jede Anfrage ihrer Zelle zu, meist nach Kunden- oder Mandantenkennung. Zellen teilen zur Laufzeit nichts: Jede hat eigene Anwendungsinstanzen, Queues und Datenbanken.

Erleidet eine Zelle ein schlechtes Deployment, einen überlastenden Kunden oder Datenkorruption, sind nur die Kunden dieser Zelle betroffen: bei 20 Zellen etwa 5 % von ihnen. Deployments werden Zelle für Zelle in Wellen ausgerollt, sodass ein schlechtes Release in der ersten Zelle auffällt. Das Design ist im AWS-Well-Architected-Whitepaper *Reducing the Scope of Impact with Cell-Based Architecture* (2023) beschrieben und wird in großem Maßstab von AWS' eigenen Diensten und von Unternehmen wie Slack genutzt. Eine verwandte Technik, Shuffle Sharding, gibt jedem Kunden eine zufällige Kombination weniger Worker aus vielen (bei 8 Workern und 2 pro Kunde gibt es 28 Kombinationen), sodass ein Kunde, dessen Anfragen Worker zum Absturz bringen, nur seine eigene Kombination lahmlegt und die meisten anderen Kunden höchstens einen Worker mit ihm teilen.

Zellbasierte Architektur



Fehler, schlechtes Deployment oder lauter Kunde in Zelle 3 betrifft etwa ein Viertel der Kunden, nicht alle

Der Router muss die einfachste und zuverlässigste Komponente des Systems sein, denn er ist das Einzige, was alle Zellen teilen.

Zellen haben ihren Preis: mehr Instanzen und Datenbanken im Betrieb, zellübergreifende Operationen (Berichte über alle Kunden, das Verschieben eines Kunden zwischen Zellen), die eigene Mechanismen brauchen, und ein Router samt Zuordnungsdienst, die extrem einfach und zuverlässig sein müssen. Sie eignen sich für große mandantenfähige Systeme mit natürlichen Partitionsschlüsseln; für eine interne Single-Tenant-Anwendung sind sie Over-Engineering.

Die Edge: CDNs und Edge Functions

Zwischen Nutzern und Regionen liegt die Edge: die Hunderte von Standorten, die Content Delivery Networks wie Akamai, Cloudflare, Fastly, Amazon CloudFront und Azure Front Door betreiben. Ein CDN cacht statische und cachebare Inhalte (Bilder, Skripte, Speisekartenseiten) in Nutzernähe, nach den HTTP-Caching-Regeln, die unter Caching beschrieben sind, und fängt Traffic-Spitzen und viele Denial-of-Service-Angriffe ab, bevor sie den Origin erreichen.

Edge Functions gehen weiter, indem sie Code an diesen Standorten ausführen: Cloudflare Workers, Fastly Compute, CloudFront Functions und Lambda@Edge, Akamai EdgeWorkers sowie die Edge Functions von Plattformen wie Vercel und Netlify. Cloudflare Workers führt Code in leichtgewichtigen V8-Isolates aus, den Sandboxes, die JavaScript-Engines für Browser-Tabs nutzen, statt in einem Container pro Funktion; daher ist der Start sehr schnell, und viele Mandanten teilen sich sicher einen Prozess. Dieser Kurs selbst läuft auf Cloudflare Workers. Gute Einsatzzwecke sind Request-Routing und A/B-Zuordnung, Authentifizierungsprüfungen und Token-Validierung, Header- und Redirect-Regeln, Bildskalierung, einfache Personalisierung und API-Proxys, die geheime Schlüssel aus dem Browser heraushalten.

Die Grenzen sind architektonischer Art. Edge Functions haben enge Grenzen für CPU-Zeit und Speicher; Edge-Speicher ist meist eventually consistent (bei Cloudflares Workers KV kann es einige Zeit dauern, bis sich ein Schreibzugriff global verbreitet hat) oder bietet starke Konsistenz nur für ein einzelnes koordinierendes Objekt, wie es Durable Objects tun; und das führende System (System of Record) bleibt normalerweise in einer Region. Ein gängiges Muster lautet daher *an der Edge lesen, am Origin schreiben*: Speisekarten, Restaurantseiten und Sitzungsprüfungen von der Edge ausliefern und Bestellungen und Zahlungen an die Heimatregion schicken.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Wie viel Multi-Region brauchen Sie?

Beantworten Sie das für jede wichtige User Journey, nicht für das ganze System auf einmal.

Q1 **Kann das Geschäft verkraften, dass diese Journey mehrere Stunden nicht verfügbar ist, während eine Region aus Backups wieder aufgebaut wird?**

Ja Mehrere Availability Zones in einer Region plus getestetes, automatisiertes, regionsübergreifendes Backup and Restore nutzen. Neu bewerten, wenn sich das Geschäft ändert.

Nein Weiter: Ein Standby in einer anderen Region ist nötig.

Q2 **Ist eine Wiederherstellungszeit von Dutzenden Minuten akzeptabel, und können Sie darauf zählen, während eines regionalen Incidents Kapazität zu bekommen?**

Ja Pilot Light: Daten kontinuierlich replizieren, Infrastructure as Code bereithalten und das Hochskalieren und Umschalten des Traffics automatisieren.

Nein Weiter: Die Kapazität muss bereits laufen.

Q3 **Ist eine Wiederherstellungszeit von wenigen Minuten akzeptabel, anfangs mit reduzierter Kapazität?**

Ja Warm Standby: eine kleine Arbeitskopie betreiben, sie mit jedem Release deployen und ihr etwas echten Traffic schicken, um zu beweisen, dass sie funktioniert.

Nein Active-Active, mit einem Datendesign für Schreibzugriffe in mehreren Regionen (Heimatregionen oder eine global konsistente Datenbank); planen Sie Budget für die Entwicklungsarbeit ein, nicht nur für die Infrastruktur.

Q4 **Brauchen entfernte Nutzer schnelle Schreibzugriffe, oder müssen manche Daten in ihrem Rechtsraum bleiben?**

Ja Nach Geografie partitionieren: Kunden eine Heimatregion geben und ihre Daten dort halten, statt alles überallhin zu replizieren.

Nein Lesezugriffe entfernter Nutzer über ein CDN und Edge Functions bedienen und Schreibzugriffe in einer Primärregion halten.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine DR-Strategie für die Essenslieferplattform wählen

Die Plattform läuft in einer Region über drei Availability Zones. Der Bestellkern (Bestellungen, Zahlung, Disposition) kostet etwa 60.000 pro Monat an Infrastruktur. Er verarbeitet etwa 40.000 Bestellungen am Tag, in der Spitze 6000 pro Stunde zur Abendessenszeit, mit etwa 5 Plattformumsatz pro Bestellung. Die Geschäftsleitung verlangt einen Wiederherstellungsplan für einen regionalen Ausfall. Alle Zahlen sind beispielhaft, in Pfund.

Kosten eines Ausfalls	Der Umsatz liegt im Tagesdurchschnitt bei etwa 8000 pro Stunde und erreicht in der Spitze 30.000. Der Vertrauensverlust bei Kunden, Restaurants und Kurieren verdoppelt die tatsächlichen Kosten grob. Verträge mit großen Restaurantketten sichern zu, dass Bestellungen innerhalb einer Stunde wieder möglich sind.
Anforderung	RTO von höchstens 1 Stunde und RPO von unter 1 Minute für den Bestellkern; RTO von 24 Stunden für Analysen, Empfehlungen und Marketing.
Backup and Restore	Zusätzlich etwa 2000 pro Monat. Gemessener Wiederaufbau: 4 bis 6 Stunden (dominiert von der Wiederherstellung einer 2-TB-Datenbank). Verfehlt das RTO von 1 Stunde.
Pilot Light	Zusätzlich etwa 8000 pro Monat (kontinuierliches Datenbankreplikat, Netzwerk, bereitliegende Images). Geprobter Failover: 35 bis 50 Minuten, überwiegend für das Skalieren der Anwendungsschicht. Erfüllt das RTO nur, wenn während eines regionalen Ereignisses Kapazität zu bekommen ist.
Warm Standby	Zusätzlich etwa 18.000 pro Monat (Replikat plus Anwendung in 20 % der Größe). Geprobter Failover: etwa 10 Minuten, dann Hochskalieren im laufenden Betrieb. Erfüllt das RTO mit Reserve.
Active-Active	Zusätzlich etwa 45.000 pro Monat plus geschätzte sechs Personenmonate Entwicklung, um Bestell- und Zahlungsdaten für Schreibzugriffe in mehreren Regionen neu zu gestalten. RTO nahe null, weit über der Anforderung.
Entscheidung	Warm Standby für den Bestellkern; Backup and Restore für Analysen, Empfehlungen und Marketing; Speisekarten und Restaurantseiten aus dem CDN, damit das Stöbern während des Failovers weitergeht. Zusatzkosten insgesamt etwa 20.000 pro Monat, rund ein Drittel der Infrastruktur des Kerns.
Bedingungen	Der Standby wird mit jedem Release von derselben Pipeline deployt, erhält 2 % des echten Traffics, um seine Funktion zu beweisen, und ein vollständiger Failover wird jedes Quartal geprobt. Die Entscheidung wird in einem ADR festgehalten.

Der erwartete Verlust durch einen seltenen regionalen Ausfall allein würde Warm Standby nicht rechtfertigen; das vertragliche RTO und der Reputationsschaden eines langen Ausfalls zur Abendessenszeit tun es. DR wird wie eine Versicherung gekauft, und das Geschäft legt die Prämie fest, die es zu zahlen bereit ist.

Die Staffelung nach Journey senkte die Kosten erheblich: Nur der Teil des Systems, der sich schnell erholen muss, bezahlt für schnelle Wiederherstellung.

ACHTUNG

Fallstricke bei Multi-Region

- Eine Recovery-Region, die nie Traffic erhalten hat: Konfigurationsdrift, fehlende Secrets und kleine Quotas zeigen sich erst während der Katastrophe.
- Failover-Werkzeuge, DNS, Identitäten oder die Deployment-Pipeline, die nur in der ausgefallenen Region gehostet sind.
- Die Annahme, Replikation schütze vor Löschung oder Datenkorruption; sie kopiert beides.
- Synchrone regionsübergreifende Aufrufe auf dem kritischen Pfad des Nutzers, die jeder Anfrage lange Round Trips hinzufügen.
- Active-Active als Infrastruktureinstellung statt als Problem des Datendesigns zu behandeln.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum braucht ein Active-Active-Design dennoch Point-in-Time-Backups?

Replikation kopiert Löschungen, fehlerhafte Migrationen und Datenkorruption binnen Sekunden in jede Region; nur Backups ermöglichen es, die Daten so wiederherzustellen, wie sie vor dem Fehler waren.

▼ Was bestimmt bei einem Design mit asynchroner Replikation in eine Standby-Region, wie viele Daten bei einem regionalen Ausfall verloren gehen?

Die Replikationsverzögerung im Moment des Ausfalls: Schreibzugriffe, die das Replikat noch nicht erreicht hatten, gehen verloren, daher ist die Verzögerung das tatsächliche RPO.

▼ **Vor welchen Ausfällen schützt eine zellbasierte Architektur, vor denen eine zweite Region mit demselben Code nicht schützt?**

Vor Ausfällen, die das System selbst verursacht, etwa ein schlechtes Deployment, eine schädliche Eingabe oder ein überlastender Kunde, die jede Region treffen würden, aber nur eine Zelle treffen.

▼ **Warum ist Pilot Light bei einem großen regionalen Incident riskanter als Warm Standby?**

Pilot Light muss die Anwendungsschicht im Moment des Ausfalls starten und skalieren, wenn Kapazität und Control-Plane-Operationen knapp sein können; bei Warm Standby läuft bereits funktionierende Kapazität.

▼ **Probieren Sie es aus: Zur Veranschaulichung: Backup and Restore bedeutet einen regionalen Ausfall von 8 Stunden, bei £8000 Verlust pro Stunde. Ein Warm Standby kostet £6000 im Monat und senkt das auf 20 Minuten. Erwartet wird ein regionaler Ausfall alle fünf Jahre. Vergleichen Sie die jährlichen Kosten, und geben Sie eine Empfehlung.**

Backup: 8 x £8000 = £64.000 pro Ausfall oder £12.800 im Jahr. Standby: £72.000 im Jahr plus etwa £533 erwarteter Verlust. Allein vom Umsatz her ist es nicht gerechtfertigt. Prüfen Sie RTO und RPO mit dem Geschäft: Verträge, Betriebssicherheit oder verlorene Kunden könnten die Antwort ändern, und ein günstigeres Pilot Light könnte passen. Halten Sie in jedem Fall getestete Backups vor, denn Replikation kopiert Löschungen und Datenbeschädigungen.

QUELLEN

- AWS Whitepaper: Disaster Recovery of Workloads on AWS, disaster recovery options in the cloud: https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-options-in-the-cloud.html
- AWS Whitepaper: Reducing the Scope of Impact with Cell-Based Architecture: https://docs.aws.amazon.com/wellarchitected/latest/reducing-scope-of-impact-with-cell-based-architecture/reducing-scope-of-impact-with-cell-based-architecture.html
- AWS: Summary of the Amazon DynamoDB service disruption in the Northern Virginia (US-EAST-1) Region: https://aws.amazon.com/message/101925/
- Amazon Builders' Library: Workload isolation using shuffle-sharding: https://aws.amazon.com/builders-library/workload-isolation-using-shuffle-sharding/
- Cloudflare Workers documentation: How Workers works: https://developers.cloudflare.com/workers/reference/how-workers-works/

Platform Engineering und Internal Developer Platforms

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 3.7 Gesetz von Conway
- 10.4 Infrastructure as Code
- 10.5 CI/CD und Deployment

Begriffe von früher, die er verwendet: Kubernetes, Infrastructure as Code (IaC), Policy as Code, DevOps, Observability

DIE IDEE IN KÜRZE

Stellen Sie sich ein großes Büro vor, in dem jede Abteilung ihre eigenen Drucker kaufen, ihr eigenes Netzwerk verkabeln und ihre eigene Reinigungskraft einstellen muss. Die Arbeit wird erledigt, aber alle verschwenden Zeit mit Nebenaufgaben, und manche Abteilungen erledigen sie schlecht. Ein gutes Facility-Management bietet stattdessen fertige Dienste an: einen Besprechungsraum online buchen, einen Laptop aus einer kurzen Liste bestellen, einen Schreibtisch mit wenigen Klicks anfordern. Niemand ist gezwungen, sie zu nutzen, aber sie sind der einfachste Weg. Platform Engineering ist dieses Facility-Management für Softwareteams: Es macht die üblichen, notwendigen Nebenaufgaben beim Bauen und Betreiben von Software schnell, sicher und per Self-Service erledigbar.

WAS SIE WISSEN MÜSSEN

- Platform Engineering baut eine Internal Developer Platform auf: Self-Service-Werkzeuge, Dienste, Wissen und Unterstützung, mit denen Produktteams Software ohne Tickets bauen und betreiben können, was doppelten Aufwand und ungleichmäßige Qualität verringert.
- Ein Golden Path (Paved Road) ist ein unterstützter, durchgängiger und optionaler Weg mit klaren Vorgaben, eine häufige Aufgabe zu erledigen, etwa einen neuen Service zu erstellen und in Produktion zu deployen.
- Backstage, bei Spotify entstanden und 2020 als Open Source veröffentlicht, ist ein CNCF-Projekt mit Softwarekatalog, Vorlagen, TechDocs und Plug-ins; ein Portal ist die Eingangstür, nicht die Plattform selbst.
- Die Plattform ist ein Produkt mit internen Kunden: Sie braucht einen Product Owner, Nutzerforschung, Dokumentation, eigene SLOs und eine Akzeptanz, die verdient statt verordnet wird (X-as-a-Service und Thinnest Viable Platform aus Team Topologies).

- Self-Service wird durch Guardrails sicher: automatische Richtlinienprüfungen mit klaren Meldungen und einem Weg für Ausnahmen statt menschlicher Freigabe-Gates.
- Erfolg wird an den Ergebnissen anderer Teams gemessen: Akzeptanz, Zeit bis zum ersten Deployment, DORA-Metriken und Developer Experience, mit mehreren SPACE-Dimensionen statt einer einzigen Zahl.
- Typische Fehlschläge sind eine Plattform, die niemand will, ein umbenanntes ticketgetriebenes Betriebsteam, eine verordnete Nutzung und Abstraktionen, die zu viel verbergen, als dass man noch debuggen könnte.

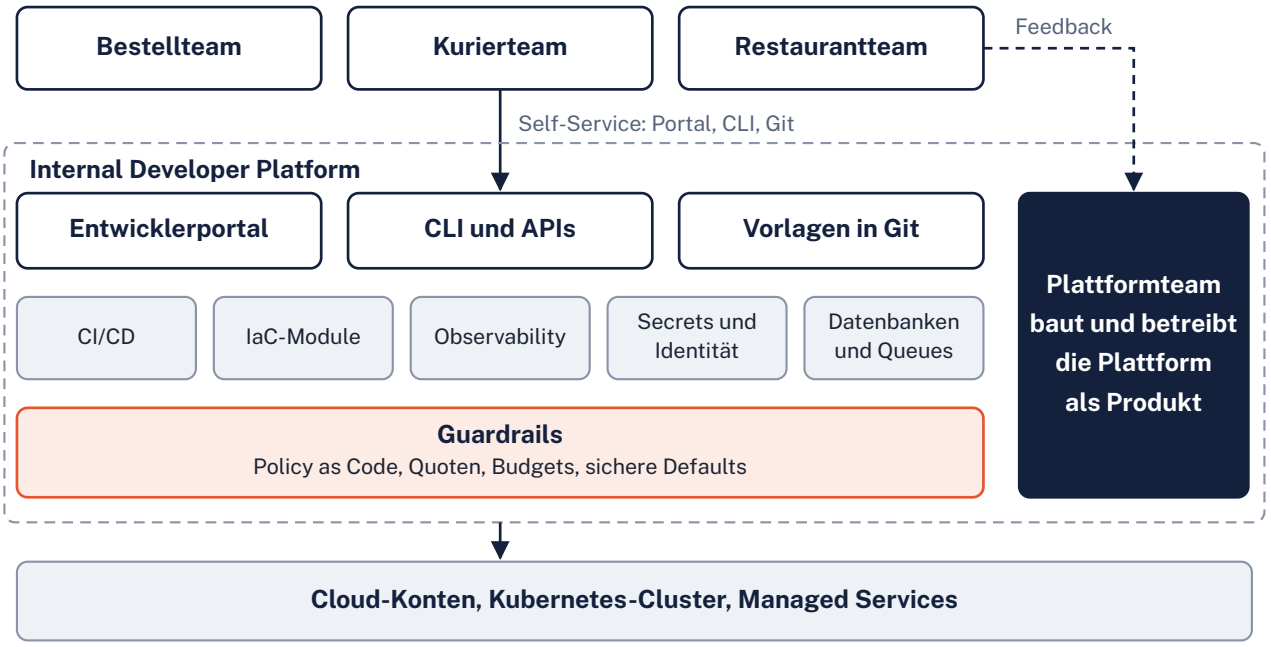
Das Problem, das Plattformen lösen

Im Jahr 2006 fasste Werner Vogels, Chief Technology Officer von Amazon, das Betriebsmodell seines Unternehmens als „you build it, you run it“ zusammen: Das Team, das einen Service schreibt, deployt, überwacht und repariert ihn auch. Die Idee wurde zentral für DevOps und für Microservices, und sie funktioniert: Teams, die ihre eigene Software betreiben, spüren die Folgen ihrer Designentscheidungen und beheben sie. Doch das in diesem Kapitel beschriebene cloud-native Werkzeugset ist groß. Ein Produktteam, das heute einen Service bauen, betreiben und absichern muss, muss womöglich Container-Images, Kubernetes-Objekte, Infrastructure as Code, Pipelines, Observability, Secrets-Management, Netzwerkrichtlinien, Scans der Software-Lieferkette und Kostenkontrollen verstehen, bevor es eine Zeile Geschäftslogik schreibt.

Wenn jedes Team all das allein löst, passieren drei Dinge. Aufwand fällt mehrfach an, denn zehn Teams schreiben zehn leicht unterschiedliche Pipelines. Die Qualität schwankt, denn manche Teams verschlüsseln ihre Datenbanken und konfigurieren Backups, andere vergessen es. Und die Zeit, die Produktteams mit Infrastruktur verbringen, fehlt für das Produkt. Platform Engineering ist die Antwort darauf: Ein eigenes Team baut eine Internal Developer Platform (IDP), die diese Fähigkeiten so bündelt, dass Produktteams sie per Self-Service nutzen können.

Evan Bottchers Artikel von 2018 auf martinfowler.com liefert eine viel zitierte Beschreibung: Eine digitale Plattform ist ein Fundament aus Self-Service-APIs, Werkzeugen, Diensten, Wissen und Unterstützung, gestaltet als überzeugendes internes Produkt. Jeder Teil dieser Beschreibung zählt. *Self-Service* heißt: keine Tickets für Routinebedarf. *Wissen und Unterstützung* heißt: Dokumentation und Menschen, nicht nur Software. *Überzeugend* heißt: Teams nutzen sie aus freien Stücken, weil sie besser ist, als die Arbeit selbst zu erledigen. *Internes Produkt* heißt: Sie wird wie ein Produkt mit Kunden geführt, nicht wie ein Projekt oder eine Kostenstelle.

Was eine Internal Developer Platform enthält



Produktteams nutzen die Plattform über Self-Service-Schnittstellen; das Plattformteam betreibt sie als Produkt und lernt aus ihrem Feedback.

Eine IDP ist kein einzelnes Produkt. Meist ist sie eine Integration vorhandener Werkzeuge (ein CI-System, ein GitOps-Controller, die Managed Services eines Cloud-Anbieters, ein Observability-Stack, ein Secret Store) hinter einheitlichen Schnittstellen und sinnvollen Standardwerten. Die eigentliche Arbeit des Plattformteams besteht darin, sie auszuwählen, zu integrieren, zu dokumentieren und zu unterstützen und die Erfahrung ihrer Nutzung zu gestalten, nicht darin, einen neuen Orchestrator zu schreiben.

Golden Paths und Paved Roads

Das Wertvollste, was eine junge Plattform bieten kann, ist ein Golden Path: ein unterstützter und dokumentierter Weg mit klaren Vorgaben, eine häufige Aufgabe durchgängig zu erledigen, etwa einen neuen Service zu erstellen und in Produktion zu bringen. Spotify machte den Begriff bekannt (sein Engineering-Blog beschrieb 2020 seine Golden Paths); Netflix spricht von der *Paved Road*. Die Metapher ist präzise. Eine befestigte Straße ist der einfachste Weg und wird instand gehalten, aber Sie dürfen trotzdem ins Gelände abbiegen, wenn Sie einen guten Grund haben; Sie bekommen dann nur nicht die Unterstützung der Straße.

Ein guter Golden Path hat fünf Eigenschaften. Er ist **durchgängig**: Er führt ein Team von einem leeren Repository bis zu einem überwachten Service in Produktion, nicht nur durch einen Schritt. Er hat **klare Vorgaben**: Er trifft die üblichen Entscheidungen (Sprachversionen, Log-Format, Deployment-Strategie, Datenbank-Standardwerte), damit Teams es nicht tun müssen.

Er ist **optional**: Teams mit ungewöhnlichen Anforderungen können ihn verlassen, was das Plattformteam zwingt, ehrlich zu prüfen, ob der Weg tatsächlich besser ist. Er ist **unterstützt**: Wenn er nicht funktioniert, repariert ihn das Plattformteam. Und er **entwickelt sich weiter**: Services, die aus der Vorlage des Vorjahres entstanden sind, erhalten Updates, statt auf ihrer Ausgangsversion stehen zu bleiben.

Was ein Golden Path für einen neuen Service typischerweise bietet

Element	Was das Produktteam bekommt	Was es ersetzt
Service-Vorlage	Ein Repository mit lauffähigem Grundgerüst, Tests, Dockerfile, Linting und README, in wenigen Minuten erzeugt.	Einen alten Service kopieren und löschen, was nicht passt.
Pipeline	Eine fertige CI/CD-Pipeline mit Scans, Signierung und schrittweisem Deployment, wie in Abschnitt 10.5.	Jedes Team schreibt und pflegt seine eigene Pipeline.
Laufzeitkonfiguration	Einstellungen für Kubernetes oder eine Managed Platform mit sinnvollen Resource Requests, Probes und Autoscaling.	Versuch und Irrtum in Produktion.
Infrastrukturmodule	Geprüfte Infrastructure-as-Code-Module für Datenbanken, Queues und Buckets, mit eingebauter Verschlüsselung, Backups und Tags.	Tickets an ein Betriebsteam oder handgeschriebene Module unterschiedlicher Qualität.
Observability	Strukturiertes Logging, Metriken, Traces und ein Einstiegs-Dashboard mit Alerts auf Service Level Objectives.	Monitoring, das erst nach dem ersten Incident ergänzt wird.
Katalogeintrag und Doku	Der Service ist mit Owner, Abhängigkeiten und Dokumentation registriert.	Tabellenkalkulationen und Kopfwissen darüber, wem was gehört.

Backstage und Entwicklerportale

Ein Entwicklerportal ist die Eingangstür vieler Plattformen. Das bekannteste ist Backstage, entwickelt bei Spotify, im März 2020 als Open Source veröffentlicht, im September 2020 von der Cloud Native Computing Foundation (CNCF) als Sandbox-Projekt aufgenommen und im März 2022 in den Incubation-Status befördert. Seine Kernfunktionen sind ein **Softwarekatalog**, in dem jeder Service, jede Website, Bibliothek, API und Ressource mit Owner und Beziehungen registriert ist; **Software-Templates**, die neue Projekte aus Golden-Path-Vorlagen erzeugen; **TechDocs**, das als Markdown neben dem Code geschriebene Dokumentation veröffentlicht; und ein großes Ökosystem von Plug-ins, die Pipelines, Kubernetes-Status, Kosten und Incidents neben jedem Katalogeintrag anzeigen.

Jede Komponente beschreibt sich selbst in einer kleinen Datei in ihrem eigenen Repository, sodass die Angaben zur Zuständigkeit beim Code liegen und wie Code reviewt werden:

```
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: restaurant-ratings
  description: Stores and serves customer ratings of restaurants
  tags: [python, http]
spec:
  type: service
  lifecycle: production
  owner: team-restaurants
  system: restaurant-discovery
  providesApis: [ratings-api]
  dependsOn: [resource:ratings-db]
```

Ein Backstage-Katalogeintrag (catalog-info.yaml) für einen Service der Essenslieferplattform: sein Owner, sein Lebenszyklus, das System, zu dem er gehört, die API, die er bereitstellt, und eine Ressource, von der er abhängt.

Zwei Vorbehalte. Backstage ist ein Framework, das eine Organisation selbst zusammenbaut und betreibt, wofür meist Engineers mit TypeScript- und React-Kenntnissen nötig sind, und die Qualität seiner Plug-ins schwankt; für Teams, die es nicht selbst betreiben wollen, gibt es gemanagte Backstage-Dienste und kommerzielle Portale. Wichtiger noch: **Ein Portal ist keine Plattform**. Ein schönes Portal vor manuellen, ticketgetriebenen Prozessen zeigt die Warteschlange bloß hübscher an. Beginnen Sie mit den Fähigkeiten und dem Golden Path; ergänzen Sie ein Portal, wenn dahinter genug steckt, dass es sich lohnt, es zu finden.

Die Plattform als Produkt

Der wichtigste Gedanke im Platform Engineering ist, dass die Plattform ein Produkt ist und die Entwicklungsteams, die sie nutzen, ihre Kunden sind. Das bedeutet: einen Produktmanager oder eine Plattformleitung, die die Roadmap verantwortet; Nutzerforschung (Interviews, einem Team beim Erstellen eines Service zusehen, Umfragen) vor dem Bauen; Dokumentation und Onboarding, die als Features behandelt werden; veröffentlichte Service Level Objectives für die Plattform selbst; Versionierungs- und Abkündigungsrichtlinien, damit Teams planen können; und vor allem eine Akzeptanz, die verdient statt verordnet wird. Wenn Teams die Plattform meiden, ist das Marktfeedback, keine Befehlsverweigerung.

Team Topologies (Matthew Skelton und Manuel Pais, 2019), behandelt in Abschnitt 11.5, liefert das Organisationsmodell. Ein Plattformteam ist dazu da, die Last der Stream-aligned Teams zu verringern, also der Teams, die an einem Strom fachlicher Änderungen wie Bestellungen oder Kurierdisposition ausgerichtet sind, und arbeitet mit ihnen vor allem im Modus *X-as-a-Service*

zusammen: ein klares Self-Service-Angebot mit wenig Abstimmungsbedarf. Der Collaboration-Modus wird bewusst und nur kurz eingesetzt, etwa beim Entwurf einer neuen Fähigkeit mit ein oder zwei Pionierteams. Skelton und Pais prägten auch den Begriff Thinnest Viable Platform: die kleinste Plattform, die wirklich hilft, was für eine kleine Organisation eine gut geschriebene Wiki-Seite mit freigegebenen Diensten und eine gemeinsame Pipeline-Vorlage sein kann.

Das bedeutet auch, dass die Plattform das Gesetz von Conway zu ihren Gunsten nutzen muss: Die Schnittstellen der Plattform werden zu den Grenzen zwischen dem Plattformteam und jedem Produktteam, deshalb sollten sie stabil, dokumentiert und schmal sein wie jede gute Abstraktion.

Self-Service mit Guardrails

Self-Service beseitigt die Warteschlange; Guardrails halten ihn sicher. Der Unterschied zu herkömmlichen Kontrollen liegt darin, wo die Prüfung stattfindet. Ein **Gate** ist eine Person, die eine Anfrage genehmigen muss, etwa ein Change Advisory Board oder ein Datenbankadministrator, der ein Ticket prüft. Ein **Guardrail** ist eine automatische Grenze, die alles durchlässt, was den Richtlinien entspricht, und alles andere im Moment der Anfrage blockiert oder markiert. Guardrails werden meist als Policy as Code geschrieben (eingeführt in Abschnitt 10.4) und in Pipelines, Admission Controllern und Richtlinien der Cloud-Organisation durchgesetzt.

Self-Service-Fähigkeiten und ihre Guardrails

Fähigkeit	Self-Service-Schnittstelle	Beispiele für Guardrails
Neuer Service	Vorlage im Portal oder in der CLI	Muss einen Owner im Katalog haben; muss das freigegebene Basis-Image verwenden.
Datenbank	Infrastrukturmodul oder Portalformular	Verschlüsselung ruhender Daten an; automatische Backups an; kein öffentlicher Netzwerkzugang; Größe durch Team-Quote begrenzt.
Deployment	Merge in main löst die Pipeline aus	Nur signierte Images aus der internen Registry; Resource Requests und Probes erforderlich; Canary-Analyse erforderlich für Tier-1-Services.
Cloud-Ausgaben	Team-Konten oder Namespaces, auf Anfrage erstellt	Pflicht-Kostentags; Budget-Alerts bei 80 % und 100 %; teure Instanztypen brauchen eine Begründung.
Zugriff	Rollenanfragen über das Portal	Produktionszugriff ist zeitlich begrenzt und wird protokolliert; keine langlebigen persönlichen Schlüssel.

Gute Guardrails haben drei Eigenschaften gemeinsam: Sie geben eine klare Meldung, die erklärt, was blockiert wurde und wie es sich beheben lässt; sie werden wie Code versioniert und getestet, mit einer Phase von Warnungen, bevor eine neue Regel zu blockieren beginnt; und sie haben einen dokumentierten Weg für begründete Ausnahmen, denn ein Guardrail ohne Ausnahmeweg wird beim ersten Irrtum wieder zum Gate.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: Gate oder Guardrail?

Der Punkt, an dem hier die meisten stolpern, ist der Unterschied zwischen einem Gate und einem Guardrail. Beide können dieselben Regeln durchsetzen, aber ein Gate stellt vor jede Anfrage eine Person, während ein Guardrail automatisch im Moment der Anfrage prüft. Verfolgen Sie eine Kontoanfrage auf beiden Wegen.

- Das Restaurantteam braucht ein Cloud-Konto für seinen neuen Restaurantbewertungsservice. Die Regeln besagen: Budget-Alerts müssen aktiv sein, jede Ressource muss Kostentags tragen, und teure Maschinentypen brauchen eine angegebene Begründung.
- Mit einem Gate eröffnet das Team ein Ticket. Wenn es an der Reihe ist, vielleicht Tage später, prüft ein Cloud-Operations-Engineer es anhand der Regeln und legt das Konto an. Die Regeln werden durchgesetzt, aber um den Preis des Wartens.
- Mit einem Guardrail beantragt das Team das Konto über das Portal der Plattform. Automatische Prüfungen, geschrieben als Policy as Code, gleichen den Antrag mit denselben Regeln ab. Er besteht, und das Konto wird ohne Ticket und ohne menschliche Freigabe angelegt.
- Später versucht das Team, einen Server ohne Kostentags zu starten. Die Prüfung blockiert das sofort, mit einer Meldung, die sagt, was blockiert wurde, warum und wie es sich beheben lässt. Das Team ergänzt die Tags und versucht es erneut.
- Hätte das Team einen Fall, den die Regeln nicht vorhergesehen haben, würde es den dokumentierten Ausnahmeweg nutzen. Ohne ihn bleibt das Team bei der ersten falschen Regel stecken, und der Guardrail ist wieder zum Gate geworden.
- **Die Lehre:** Ein Guardrail schwächt die Regeln nicht; er verlegt die Prüfung aus der Warteschlange einer Person in den Moment der Anfrage. Er funktioniert, wenn er sich selbst erklärt, wie Code getestet wird und einen Weg für begründete Ausnahmen offenlässt.

Kognitive Last reduzieren, nicht alles verbergen

Team Topologies übernimmt den Begriff der kognitiven Last aus John Swellers pädagogischer Forschung und unterscheidet drei Arten. Intrinsische Last steckt in der Aufgabe selbst, etwa zu verstehen, wie Liefergebühren funktionieren. Extrinsische Last kommt aus der Umgebung, etwa sich die zwölf Schritte zu merken, mit denen man eine Datenbank beantragt. Lernbezogene Last ist die Anstrengung des Lernens, die künftige Arbeit erleichtert, etwa die Domäne tiefer zu verstehen. Eine Plattform sollte die extrinsische Last beseitigen, damit Teams ihre begrenzte Aufmerksamkeit auf die intrinsische und lernbezogene Last in ihrer eigenen Domäne richten können.

Das heißt nicht, alles zu verbergen. Ein Team, das seinen Service betreibt, muss immer noch verstehen, wie er sich in Produktion verhält: seinen Ressourcenverbrauch, seine Fehlermodi, seine Latenz und seine Kosten. Abstraktionen, die zu viel verbergen, werden im ungünstigsten Moment undicht, nämlich während eines Incidents, wenn das Team feststellt, dass es die Logs nicht sehen kann oder nicht weiß, auf welcher Datenbankstufe es läuft. Gute Plattformen sind transparent: Sie bieten sinnvolle Standardwerte und einfache Schnittstellen und lassen Teams dennoch unter die Haube schauen und Einstellungen überschreiben, wenn sie es brauchen.

Messen, ob die Plattform funktioniert

Der Wert einer Plattform zeigt sich in den Ergebnissen anderer Teams, also müssen ihre Messgrößen dort ansetzen. Nützliche Messgrößen fallen in vier Gruppen, und ein ausgewogenes Set nimmt aus jeder mindestens eine.

Erfolgsmessgrößen einer Plattform

Gruppe	Beispielhafte Messgrößen	Achten Sie auf
Akzeptanz	Anteil der Services auf dem Golden Path; freiwillige Nutzung neuer Fähigkeiten; Teams, die den Pfad verlassen, und warum.	Eine verordnete Nutzung sagt nichts über den Wert aus.
Zustellung	DORA-Metriken für Teams auf und abseits der Plattform; Zeit vom Erstellen eines Service bis zu seinem ersten Produktions-Deployment; Einarbeitungszeit für neue Engineers.	Vergleiche zwischen Teams mit sehr unterschiedlicher Arbeit.
Developer Experience	Regelmäßige Umfragen zu Zufriedenheit, wahrgenommener Reibung und Zeitverlust durch Werkzeuge; Interviews.	Umfragemüdigkeit; Ergebnisse als Signale behandeln, nicht als Punktzahlen.
Zustand der Plattform	Die eigenen SLOs der Plattform (Verfügbarkeit der Pipelines, Bereitstellungszeit), Supportaufwand, Kosten pro Team.	Eine Plattform, die zuverlässig ist, aber nicht genutzt wird, ist trotzdem ein Fehlschlag.

Das SPACE-Framework, 2021 in ACM Queue von Nicole Forsgren, Margaret-Anne Storey und Kollegen von GitHub, Microsoft Research und der University of Victoria veröffentlicht, warnt, dass sich Entwicklerproduktivität nicht mit einer einzigen Zahl erfassen lässt. Es nennt fünf Dimensionen: Zufriedenheit und Wohlbefinden (Satisfaction and well-being), Leistung (Performance), Aktivität (Activity), Kommunikation und Zusammenarbeit (Communication and collaboration) sowie Effizienz und Flow (Efficiency and flow). Es empfiehlt, mehrere Dimensionen gleichzeitig zu messen, darunter mindestens eine Wahrnehmungsgröße wie eine Umfrage, und Einzelpersonen nie nach Aktivitätszählungen wie Commits oder Codezeilen zu beurteilen.

Die Belege zu Plattformen sind ermutigend, aber nicht bedingungslos. Der DORA-Bericht *Accelerate State of DevOps* 2024 ergab, dass Nutzer von Internal Developer Platforms eine um etwa 8 % höhere individuelle Produktivität und eine um 10 % höhere Teamleistung angaben, der Durchsatz der Softwareauslieferung und die Stabilität von Änderungen aber um 8 % bzw. 14 % niedriger lagen. Der Bericht bietet mögliche Erklärungen an, etwa zusätzliche Übergaben und zusätzliche Schritte durch die Plattformschicht selbst, und die Lehre für Architekten lautet, neben der Zufriedenheit auch die Auslieferungsergebnisse zu messen und mit einem Einbruch zu rechnen, solange eine Plattform reift.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: der erste Golden Path für einen neuen Service der Essenslieferplattform

Das Essenslieferunternehmen hat zwölf Produktteams. Eine Umfrage und eine Woche Hospitation zeigen, dass das Erstellen eines neuen Service etwa drei Wochen und vierzehn Tickets an vier verschiedene Teams erfordert. Ein Plattformteam aus vier Personen wird gebildet. Die ersten Produktteams, die einen neuen Service brauchen, sind das Restaurantteam, das einen Restaurantbewertungsservice möchte, und das Kurierteam.

Den Pfad wählen	Bestandsaufnahme der vorhandenen Services: 70 % sind zustandslose HTTP-Services in Python oder Go mit einer PostgreSQL-Datenbank und oft einem Queue-Abonnement. Diese Form wird zum Golden Path v1; alles andere bleibt vorerst abseits der Straße.
Mit den Nutzern entwerfen	Zwei Engineers aus dem Restaurant- und dem Kurierteam arbeiten drei Wochen im Plattformteam mit (Collaboration-Modus), um den Pfad mit ihnen zu bauen, nicht für sie.
Inhalt	Vorlagen-Repository; die Standard-Pipeline mit Canary-Deployment; Kubernetes-Einstellungen mit Requests, Probes und Autoscaling; ein Datenbankmodul mit Verschlüsselung, Backups und Kostentags; Logging, Tracing und ein SLO-Dashboard; ein Katalogeintrag; ein Einstiegsleitfaden.
Guardrails	Owner- und Kostentags verpflichtend; nur signierte Images; Datenbank ohne öffentlichen Zugriff. Alles wird automatisch geprüft; einen Monat lang Warnungen, bevor blockiert wird.
Erstes Ergebnis	Das Restaurantteam erstellt restaurant-ratings über das Portal und deployt eine Hello-World-Version in 5 Stunden in Produktion, gegenüber drei Wochen zuvor. Benötigte Tickets: keine.
Messen	Zeit bis zum ersten Produktions-Deployment, Umfrage unter den beiden Pilotteams und DORA-Metriken für Services auf dem Pfad, monatlich überprüft. Drei weitere Teams übernehmen den Pfad innerhalb eines Quartals freiwillig.
Nächste Schritte	Teams wünschen sich Unterstützung für Event-Consumer und Preview-Umgebungen je Pull Request; die Roadmap wird nach Nachfrage geordnet. Der Wunsch, eine Nischensprache zu unterstützen, wird vorerst abgelehnt und als abseits der Straße dokumentiert.

Die Plattform begann mit einem Pfad für den häufigsten Fall, gebaut mit ihren ersten Nutzern, und wurde an deren Ergebnissen gemessen statt an der Zahl der ausgelieferten Features.

Der Großteil der Arbeit bestand aus Integration, Standardwerten und Dokumentation vorhandener Werkzeuge, nicht aus neuer Software.

Risiken und Fehlermuster

Plattforminitiativen scheitern auf erkennbare Weise. Am häufigsten ist **eine Plattform, die niemand will**: Ein Team verbringt ein Jahr damit, eine ehrgeizige Allzweckplattform zu bauen, bringt sie heraus und stellt fest, dass die Produktteams bereits brauchbare Lösungen haben und keine Zeit für eine Migration. Schlank anzufangen, mit echten Nutzern und einem Golden Path, ist das Gegenmittel. Das zweite ist **das umbenannte Betriebsteam**: Ein Infrastrukturteam wird in Plattformteam umbenannt, arbeitet aber weiter aus einer Ticket-Warteschlange, sodass sich nichts ändert außer den Stellenbezeichnungen. Wenn Routineanfragen noch ein Ticket und eine Person brauchen, ist es noch keine Plattform.

Weitere Risiken sind eine verordnete Nutzung, die das Feedback beseitigt, das eine Plattform nützlich hält; Überabstraktion, bei der eine selbst gebaute Schicht die zugrunde liegenden Werkzeuge so vollständig verbirgt, dass Teams sie weder debuggen noch erweitern können und das Plattformteam zum Engpass für jede Änderung wird; die Vernachlässigung des Long Tail der Services, die aus alten Vorlagen entstanden sind; und eine Plattform, die zum Single Point of Failure wird, dessen Ausfall jedes Deployment stoppt. Umgekehrt braucht eine kleine Organisation vielleicht gar kein Plattformteam: Ein paar gemeinsame Vorlagen, die Freiwillige pflegen, können jahrelang die Thinnest Viable Platform sein.

Checkliste: eine Plattform starten

- [ ] **Belege für den Schmerz**
  Messen Sie, wie lange häufige Aufgaben heute dauern (neuer Service, neue Datenbank, erstes Deployment) und wie viele Tickets und Teams daran beteiligt sind.
- [ ] **Ein Product Owner**
  Jemand verantwortet die Roadmap, spricht regelmäßig mit Nutzern und kann Nein sagen.
- [ ] **Zuerst ein Golden Path**
  Wählen Sie die häufigste Service-Form und machen Sie sie durchgängig hervorragend, bevor Sie in die Breite gehen.
- [ ] **Self-Service statt Tickets**
  Jede Routineanfrage auf dem Pfad wird ohne menschliche Freigabe abgeschlossen; Ausnahmen haben einen dokumentierten Weg.
- [ ] **Guardrails als Code**
  Regeln für Sicherheit, Kosten und Zuverlässigkeit werden automatisch durchgesetzt, mit klaren Meldungen und einer Warnphase.
- [ ] **Transparente Abstraktionen**
  Teams sehen Logs, Metriken, Kosten und die zugrunde liegenden Ressourcen und können Standardwerte begründet überschreiben.
- [ ] **Ergebnismessgrößen**
  Akzeptanz, Zeit bis zum ersten Deployment, DORA-Metriken und Entwicklerumfragen werden regelmäßig überprüft und veröffentlicht.
- [ ] **Upgrade-Pfad**
  Services aus älteren Vorlagen erhalten Verbesserungen, und Abkündigungen kommen mit Vorlauf und Migrationshilfe.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Was ist der Unterschied zwischen einem Gate und einem Guardrail?**

Ein Gate braucht eine Person, die jede Anfrage genehmigt; ein Guardrail ist eine automatische Prüfung, die regelkonforme Anfragen sofort durchlässt und den Rest blockiert oder markiert.

▼ Warum sollte ein Golden Path optional sein?

Damit Teams mit ungewöhnlichen Anforderungen nicht blockiert werden und damit das Plattformteam daraus, wer den Pfad verlässt, lernt, ob er tatsächlich besser ist als die Alternativen.

▼ Ein Unternehmen installiert Backstage, aber für eine Datenbank braucht es immer noch ein Ticket und eine Woche Wartezeit. Was hat es gebaut?

Ein Portal vor einem manuellen Prozess, keine Plattform; Wert entsteht durch die Self-Service-Fähigkeiten hinter dem Portal.

▼ Warum rät das SPACE-Framework davon ab, Entwicklerproduktivität allein an Commits oder Codezeilen zu messen?

Aktivitätszählungen erfassen nur eine Dimension, lassen sich leicht manipulieren und sagen nichts über Zufriedenheit, Zusammenarbeit, Flow oder den Wert des Ergebnisses.

▼ Probieren Sie es aus: Die Unternehmensleitung benennt das Betriebsteam in „Plattform“ um, verpflichtet alle Teams, es zu nutzen, und misst den Erfolg an der Zahl der geschlossenen Tickets. Anfragen für einen neuen Service erfordern weiterhin vierzehn Tickets. Kritisieren Sie das, und sagen Sie, was Sie ändern würden.

Das ist ein umbenanntes, ticketgetriebenes Team, keine Plattform. Behandeln Sie sie als Produkt: Geben Sie ihr einen Product Owner, erforschen Sie, was Produktteams brauchen, und bauen Sie einen schlanken Golden Path für die häufigste Aufgabe, einen neuen zustandslosen Service, als Self-Service mit automatischen Guardrails. Machen Sie ihn optional, und verdienen Sie sich die Akzeptanz. Messen Sie die Ergebnisse anderer, etwa die Zeit bis zum ersten Deployment, die Akzeptanz, DORA-Metriken und die Developer Experience. Weniger Tickets sind das Ziel; geschlossene Tickets zu zählen belohnt das Falsche.

QUELLEN

- Evan Bottcher: What I Talk About When I Talk About Platforms, martinfowler.com: https://martinfowler.com/articles/talk-about-platforms.html
- CNCF TAG App Delivery: Platforms White Paper: https://tag-app-delivery.cncf.io/whitepapers/platforms/
- Backstage documentation: What is Backstage?: https://backstage.io/docs/overview/what-is-backstage
- Forsgren et al.: The SPACE of Developer Productivity, ACM Queue (2021): https://queue.acm.org/detail.cfm?id=3454124

FinOps, Kostenarchitektur und Anbieter-Lock-in

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.7 Sicherheit, Benutzbarkeit und mehr
- 5.6 Hexagonal und Clean
- 10.1 Cloud-Modelle

Begriffe von früher, die er verwendet: Managed Service, Multi-Cloud, Resource Requests und Limits, Ports and Adapters (hexagonale Architektur), Serverless

DIE IDEE IN KÜRZE

Ein Auto für einen Tag zu mieten ist billig; drei Jahre lang jeden Tag eines zu mieten ist weit teurer als ein Kauf. Den Motor auf dem Parkplatz laufen zu lassen kostet Kraftstoff für nichts. Und wer ein Auto mit einem speziellen Ladestecker least, braucht bei einem späteren Markenwechsel auch neue Ladegeräte. Cloud-Computing wirft alle drei Fragen auf einmal auf. FinOps ist die Haushaltsbuch-Gewohnheit für die Cloud: sehen, was jeder Teil kostet, Verschwendung abbauen, klug einkaufen. Lock-in ist die Frage nach dem Spezialstecker: Eine gewisse Abhängigkeit lohnt sich, solange Sie sie bewusst gewählt haben und wissen, wie Sie wieder herauskämen.

WAS SIE WISSEN MÜSSEN

- In der Cloud ist jede Architekturentscheidung auch eine Kaufentscheidung, daher sind Kosten ein Qualitätsmerkmal, das sich mit jedem Deployment ändert und sich am besten pro Einheit Geschäftswert ausdrücken lässt, etwa als Kosten pro Bestellung.
- FinOps, definiert von der FinOps Foundation (einem Projekt der Linux Foundation), verteilt die finanzielle Verantwortung auf Engineering, Finanzen und Fachbereiche, in drei iterativen Phasen: Inform, Optimise und Operate.
- Häufige Kostentreiber sind Datenabfluss (Egress) und zonenübergreifende Übertragung, ungenutzte und überdimensionierte Ressourcen, Logging-Volumen, Aufpreise für Managed Services und unkontrolliertes Speicherwachstum; die meisten werden von der Architektur geprägt.
- Rabatte für Nutzungszusagen eignen sich nur für die sichere Grundlast, Spot-Kapazität für unterbrechbare und wiederholbare Arbeit, und On-Demand-Autoscaling deckt die Spitzen ab; erst Rightsizing, dann Zusagen.

- Tagging ermöglicht die Zuordnung; Showback meldet Teams ihre Kosten, Chargeback belastet ihre Budgets und braucht eine genaue Zuordnung und vereinbarte Regeln für gemeinsame Kosten.
- Lock-in ist eine Abwägung: Hohpe (2019) unterscheidet viele Arten und wägt Wechselkosten mal Wahrscheinlichkeit gegen die Kosten der Vermeidung ab; akzeptieren Sie Lock-in dort, wo er einzigartigen Nutzen bringt.
- Portabilität hat echte Kosten; halten Sie die Geschäftslogik an wahrscheinlichen Nahtstellen hinter Ports, halten Sie Daten exportierbar und pflegen Sie eine getestete Exit-Strategie, die die EU-Finanzmarktregulierung inzwischen für kritische Dienste verlangt.

Kosten als Architekturqualität

Abschnitt 2.7 (Sicherheit, Benutzbarkeit und mehr) führte Kosten unter den Qualitätsmerkmalen auf und empfahl, sie pro Arbeitseinheit auszudrücken. Im Rechenzentrum waren die Kosten weitgehend fix: Hardware wurde alle paar Jahre gekauft, und die Entscheidungen eines Architekten bewegten die Monatsrechnung kaum. In der Cloud ist jede Architekturentscheidung auch eine Kaufentscheidung, laufend getroffen von Engineers statt einmal im Jahr vom Einkauf. Eine gesprächige Service-Grenze, ein Debug-Log-Level, eine vergessene Testumgebung oder ein synchroner regionsübergreifender Aufruf taucht jeweils auf der Rechnung des nächsten Monats auf. Kosten sind zu einem Qualitätsmerkmal geworden, das sich mit jedem Deployment ändert.

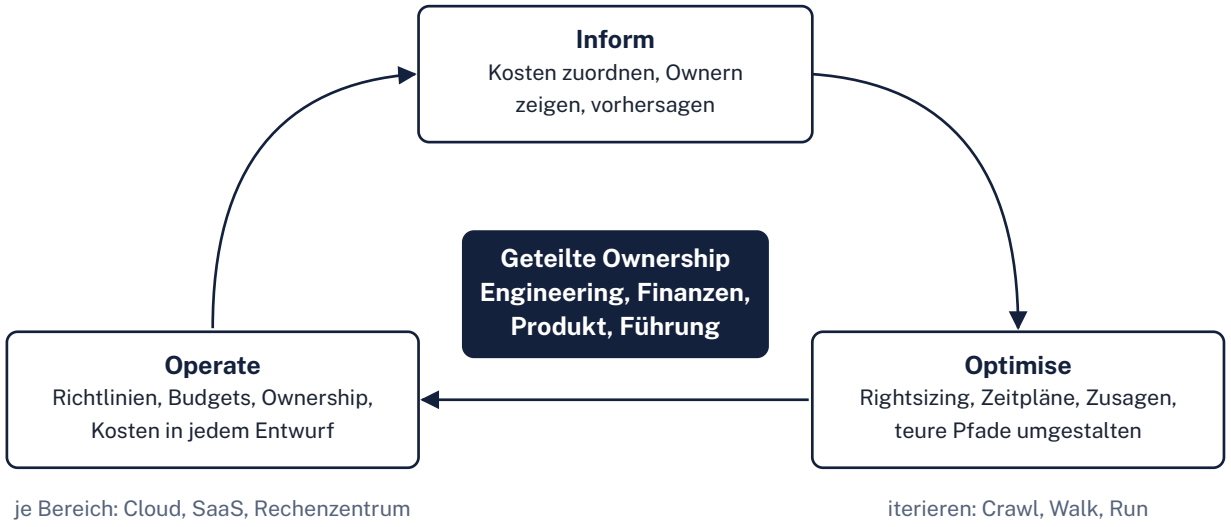
Zwei Disziplinen helfen. FinOps ist die betriebliche Praxis, die Cloud-Ausgaben sichtbar, verantwortet und gesteuert macht. Kostenbewusste Architektur ist die Entwurfspraxis, die Systeme von vornherein günstig im Betrieb pro Werteinheit macht. Dieser Abschnitt behandelt beide und wendet sich dann ihrem unbequemen Nachbarn zu: dem Lock-in, also den Kosten des Ausstiegs.

FinOps: Phasen, Prinzipien und wer was tut

FinOps wird von der FinOps Foundation, einem Projekt der Linux Foundation, definiert als operatives Framework und kulturelle Praxis, die den Geschäftswert von Technologie maximiert, durch zeitnahe, datengestützte Entscheidungen und eine finanzielle Verantwortung, die Engineering, Finanzen und Fachbereiche teilen. Der Name spielt bewusst auf DevOps an: So wie DevOps die Mauer zwischen dem Bauen und dem Betreiben von Software eingerissen hat, reißt FinOps die Mauer zwischen dem Ausgeben von Geld und der Entscheidung ein, wie Software gebaut wird.

Das FinOps Framework gliedert die Arbeit in drei iterative Phasen. **Inform** (Informieren): Kosten sichtbar machen, sie den Teams und Produkten zuordnen, die sie verursachen, prognostizieren und vergleichen. **Optimise** (Optimieren): Verschwendung abbauen und Preise verbessern, durch Rightsizing, Zeitpläne, Nutzungszusagen und Designänderungen. **Operate** (Betreiben): die Gewohnheiten aufbauen, die Kosten unter Kontrolle halten, etwa Budgets, Richtlinien, Zuständigkeiten und Kosten als Kriterium jeder Designentscheidung. Organisationen durchlaufen den Zyklus wiederholt und reifen in jeder Fähigkeit von Crawl über Walk zu Run. Zu den Prinzipien des Frameworks gehört, dass Teams zusammenarbeiten, dass der Geschäftswert Technologieentscheidungen bestimmt, dass alle Verantwortung für ihre Technologienutzung übernehmen, dass Kostendaten zugänglich, zeitnah und korrekt sind, dass FinOps zentral ermöglicht wird und dass das variable Kostenmodell der Cloud vorteilhaft genutzt werden sollte. Neuere Versionen erweitern das Framework über die Public Cloud hinaus auf andere Technologieausgaben wie SaaS-Lizenzen, Rechenzentren und KI-Dienste.

Der FinOps-Zyklus



FinOps ist ein wiederkehrender Zyklus, kein einmaliges Sparprojekt, und beruht auf gemeinsamer Verantwortung statt auf einem zentralen Team, das die Ausgaben überwacht.

Unit Economics: Kosten pro Bestellung

Eine Cloud-Rechnung, die von 80.000 auf 100.000 im Monat steigt, kann eine Katastrophe oder ein Triumph sein, je nachdem, ob das Geschäft um 5 % oder um 50 % gewachsen ist. Unit Economics drückt Kosten pro Einheit Geschäftswert aus: Kosten pro Bestellung bei der Essenslieferplattform, pro aktivem Kunden bei einem Streamingdienst, pro tausend API-Aufrufe bei einer Entwicklerplattform. Das trennt Effizienz von Wachstum, verbindet Engineering-Entscheidungen mit Geschäftsergebnissen und gibt Teams ein Ziel, das sie beeinflussen können.

Um das gut zu berechnen, braucht es drei Dinge. Erstens eine **Einheit, die für das Geschäft zählt**, gemeinsam mit Finanzen und Produkt gewählt, nicht von Engineers erfunden. Zweitens **Kostenzuordnung**: direkte Kosten (Rechenleistung und Datenbank des Bestellservice), zugeordnet über Tags oder Namespaces, und gemeinsame Kosten (die Kubernetes Control Plane, der Observability-Stack, die Werkzeuge des Plattformteams), aufgeteilt nach einer vereinbarten Regel wie der anteiligen Nutzung. Drittens eine Unterscheidung zwischen **Durchschnitts- und Grenzkosten**: Die durchschnittlichen Kosten pro Bestellung enthalten Fixkosten, die nicht mit den Bestellungen wachsen, daher sind die Kosten der nächsten tausend Bestellungen meist niedriger, als der Durchschnitt vermuten lässt, was bei der Preisgestaltung einer Aktion oder einer Wachstumsprognose wichtig ist.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: was die nächsten tausend Bestellungen kosten

Die schwierigste Idee dieses Abschnitts ist der Unterschied zwischen Durchschnitts- und Grenzkosten. Der Durchschnitt verteilt die gesamte Rechnung auf alle Bestellungen; die Grenzkosten sind das, was eine weitere Bestellung hinzufügt. Weil ein Teil der Rechnung fix ist, können beide weit auseinanderliegen. Verfolgen Sie eine Aktion mit beispielhaften Zahlen durch beide Rechnungen.

- Die monatliche Cloud-Rechnung der Essenslieferplattform beträgt £96.000 für 1,2 Millionen Bestellungen, also liegen ihre Kosten pro Bestellung, die Unit-Economics-Kennzahl, die dieser Abschnitt empfiehlt, bei £0,080. Das ist der Durchschnitt.
- Ein Teil der Rechnung bewegt sich nicht mit den Bestellungen: die im Voraus dimensionierte Bestelldatenbank, die ständig laufenden Server für die ruhigen Stunden, die Kubernetes Control Plane und die Werkzeuge des Plattformteams. Angenommen, £36.000 sind auf diese Weise fix und £60.000 wachsen mit den Bestellungen: vor allem autoskalierte Rechenleistung, Datenübertragung und Logs.
- Jede zusätzliche Bestellung kostet also etwa £60.000 geteilt durch 1,2 Millionen, also £0,050: die Grenzkosten, deutlich unter dem Durchschnitt.
- Das Unternehmen plant eine Aktion, die voraussichtlich 200.000 zusätzliche Bestellungen im Monat bringt. Zum Durchschnitt gerechnet, kostet ihre Abwicklung scheinbar 200.000 x £0,080 = £16.000. Zu Grenzkosten gerechnet, sind es etwa £10.000, solange die zusätzlichen Bestellungen in die bereits bezahlte fixe Kapazität passen.
- Grenzkosten sind nicht immer niedriger. Würde die Aktion die Bestelldatenbank über ihre aktuelle Größe hinaustreiben, käme mit dem Wechsel auf eine größere Stufe ein fixer Kostensprung hinzu, und die ersten Bestellungen über dieser Linie könnten weit mehr kosten als der Durchschnitt.
- **Die Lehre:** Verfolgen Sie die Effizienz mit den durchschnittlichen Kosten pro Bestellung, aber kalkulieren Sie zusätzliches Volumen, etwa eine Aktion oder eine Wachstumsprognose, mit den Grenzkosten. Fragen Sie zuerst, welche Kosten sich mit den Bestellungen bewegen und welche nicht.

Wohin das Geld fließt: häufige Kostentreiber

Die meisten überraschenden Cloud-Rechnungen gehen auf eine Handvoll Treiber zurück, und fast alle werden von der Architektur geprägt, nicht durch das Aushandeln besserer Preise.

Häufige Kostentreiber und Antworten der Architektur

Treiber	Warum er wächst	Antworten der Architektur
Datenabfluss (Egress) und Übertragung	Anbieter berechnen pro Gigabyte für Daten, die ins Internet abfließen, und oft auch für Datenverkehr zwischen Zonen und Regionen sowie über verwaltete NAT-Gateways. Gesprächige Services, die über Zonen verteilt sind, zahlen bei jedem Aufruf.	Öffentliche Inhalte über ein CDN ausliefern; gesprächige Services zonenlokal halten (topologiebewusstes Routing); private Endpoints zu Anbieterdiensten nutzen; komprimieren; synchronen regionsübergreifenden Verkehr vermeiden.
Ungenutzte und überdimensionierte Ressourcen	Testumgebungen, die 168 Stunden pro Woche für 50 Stunden Nutzung laufen; Container-Resource Requests weit über dem tatsächlichen Verbrauch; verwaiste Volumes, Snapshots und Load Balancer.	Rightsizing anhand des gemessenen Verbrauchs; Zeitpläne und Autoscaling auf null für Nicht-Produktionsumgebungen; zeitlich begrenzte Preview-Umgebungen; automatisches Aufräumen nicht zugeordneter Ressourcen.
Logging- und Observability-Volumen	Aufnahme und Aufbewahrung werden pro Gigabyte berechnet; Debug-Logs in Produktion, ausführliches Request-Logging und Metrik-Labels mit hoher Kardinalität wachsen mit dem Traffic.	Log-Level und Sampling für erfolgreiche Anfragen; gestaffelte Aufbewahrung (Hot für Tage, Archiv für Monate); Metriken statt Logs für Zählungen; Trace-Sampling.
Aufpreise für Managed Services	Ein Managed Service kostet pro Kapazitätseinheit mehr, als die Software selbst zu betreiben; Preise pro Anfrage wachsen linear mit dem Volumen.	Gesamtbetriebskosten einschließlich Personal und Risiko vergleichen; Dienste mit Abrechnung pro Anfrage bei hohem, gleichmäßigem Volumen überprüfen (siehe Serverless); sie dort nutzen, wo der Aufpreis echte Zuverlässigkeit oder Geschwindigkeit bringt.
Speicherwachstum	Daten, die nie gelöscht werden, Objektversionen, doppelte Kopien und alte Backups sammeln sich unbemerkt an.	Lifecycle-Richtlinien für günstigere Speicherklassen und das Löschen; mit dem Fachbereich und nach geltendem Recht vereinbarte Aufbewahrungsregeln; dedupizierte, inkrementelle Backups.

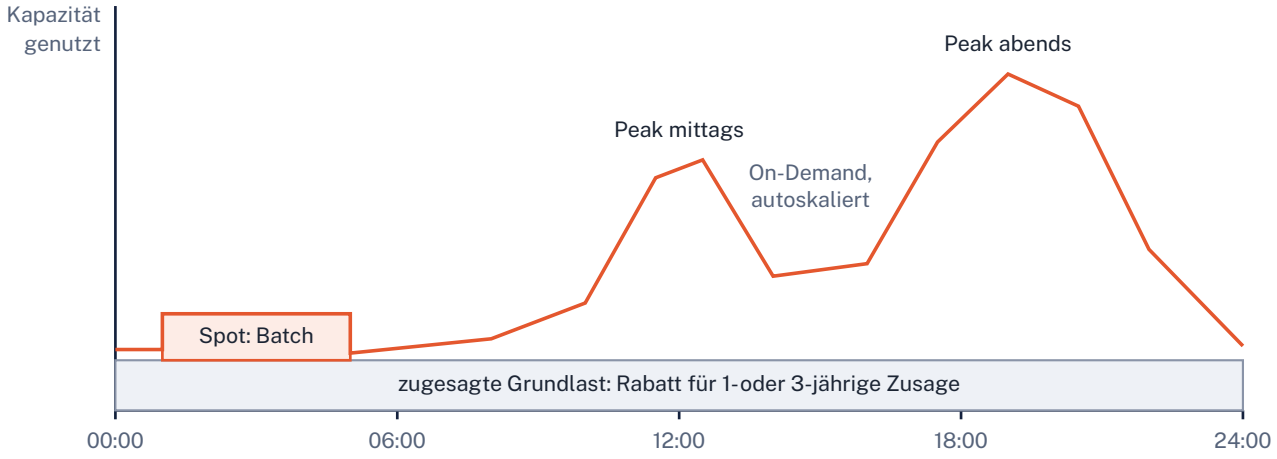
Weniger für dieselbe Kapazität zahlen

Ist die Verschwendung beseitigt, lässt sich der Preis für die verbleibende Kapazität senken. On-Demand-Kapazität ist am flexibelsten und am teuersten. Ein Rabatt für Nutzungszusagen tauscht Flexibilität gegen Preis: AWS Reserved Instances und Savings Plans, Azure Reservations und Savings Plans sowie Google Cloud Committed Use Discounts bieten erhebliche Nachlässe (AWS nennt bis zu 72 % für manche Zusagen), wenn man sich für ein

oder drei Jahre auf ein bestimmtes Nutzungs- oder Ausgabenniveau festlegt. Zusagen sollten nur die Grundlast abdecken, die für die Laufzeit sicher gebraucht wird; wer sich auf Spitzenkapazität festlegt, verwandelt einen Rabatt in Verschwendung.

Spot-Kapazität (AWS Spot Instances, Azure Spot Virtual Machines, Google Cloud Spot VMs) verkauft freie Kapazität mit hohen Rabatten, oft 60 % bis 90 %, doch der Anbieter kann sie kurzfristig zurückfordern: mit zwei Minuten Vorlauf bei AWS, etwa 30 Sekunden bei Azure und Google Cloud. Sie eignet sich für Arbeit, die zustandslos, unterbrechbar und wiederholbar ist, etwa Batch-Verarbeitung, CI-Runner, Rendering und manche zustandslosen Web-Schichten, die über viele Instanztypen verteilt sind. Sie eignet sich nicht für Datenbanken oder alles, was sich nicht mitten in einer Aufgabe stoppen lässt. Die Architektur entscheidet, wie viel einer Workload Spot nutzen kann: idempotente Jobs mit Checkpoints können es, lange, zustandsbehaftete nicht.

Preismodelle an die Form der Nachfrage anpassen



Sagen Sie die Untergrenze fest zu, skalieren Sie für die Spitzen on demand automatisch und verlagern Sie unterbrechbare Arbeit in ruhigen Zeiten auf Spot-Kapazität.

Tagging, Showback und Chargeback

Niemand optimiert Kosten, die er nicht sieht. Die Zuordnung beginnt mit einer Tagging-Richtlinie (oder Labeling-Richtlinie): Jede Ressource trägt einige wenige verpflichtende Tags, etwa verantwortliches Team, Service, Umgebung und Kostenstelle, durchgesetzt durch Guardrails im Infrastrukturcode und durch Richtlinien der Cloud-Organisation, denn von Hand vergebene Tags fehlen bald. Bei Kubernetes werden Kosten nach Namespace und Label zugeordnet, mit Werkzeugen wie dem Open-Source-Projekt OpenCost oder den Kostenzuordnungsfunktionen der Anbieter. Die FOCUS-Spezifikation der FinOps Foundation definiert ein gemeinsames Format für Abrechnungsdaten, sodass sich Kosten mehrerer Anbieter zusammenführen lassen.

Sind die Kosten zugeordnet, entscheidet eine Organisation, wie sie sie nutzt. Showback meldet jedem Team seine Kosten, ohne sein Budget zu berühren; das schafft Bewusstsein und ist der übliche Einstieg. Chargeback belastet das Budget jedes Teams mit den Kosten, was stärkere Anreize schafft, aber eine genaue Zuordnung, vereinbarte Regeln für gemeinsame Kosten und ein Verfahren für Streitfälle erfordert. In beiden Fällen gilt: Veröffentlichen Sie den Anteil der Ausgaben, der sich nicht zuordnen lässt; ein ungetaggter Anteil von mehr als ein paar Prozent bedeutet, dass die Zahlen noch nicht verlässlich genug sind, um danach zu handeln.

Architekturentscheidungen, die Kosten treiben

Die größten Einsparungen kommen aus dem Design, nicht aus Rabatten, und sie lassen sich am günstigsten erzielen, bevor ein System gebaut ist. Jede der folgenden Entscheidungen hat ein vorhersehbares Kostenprofil. **Service-Granularität**: Jeder Netzwerk-Hop kostet Rechenleistung, Serialisierung und eventuell Datenübertragung, daher kosten feingranulare Services in vielen Zonen pro Anfrage mehr als ein modularer Monolith. **Synchroner Fan-out**: Eine Anfrage, die zehn Services aufruft, hält für ihre Dauer elf Sätze von Ressourcen beschäftigt. **Anzahl der Umgebungen**: Vollständige Kopien der Produktion für jedes Team und jeden Branch vervielfachen die Rechnung; gemeinsame Umgebungen mit Isolation oder kurzlebige Previews kosten weit weniger. **Datenaufbewahrung und Replikation**: Jede zusätzliche Kopie und jede zusätzliche Region vervielfacht Speicher und Übertragung. **Caching und CDNs**: Ein Cache-Treffer ist meist weit billiger als eine Neuberechnung, und ein CDN-Treffer vermeidet Egress vom Ursprung. **Rechenmodell**: Serverless mit Abrechnung pro Anfrage ist unterhalb eines Schnittpunkts billiger als ständig laufende Kapazität und darüber teurer.

Halten Sie wesentliche Kostenfolgen in Entscheidungsdokumenten fest, neben den anderen Qualitätsabwägungen, und erwägen Sie, Kosten in die Fitness Functions aufzunehmen, die die Architektur absichern: zum Beispiel eine monatliche Prüfung, dass die Kosten pro Bestellung ohne Erklärung nicht um mehr als 10 % gestiegen sind.

Lock-in ist eine Abwägung, keine Sünde

Lock-in sind die Kosten und Schwierigkeiten, etwas zu verlassen, von dem man abhängt. Meist wird er als etwas diskutiert, das um jeden Preis zu vermeiden ist; Gregor Hohpes Artikel von 2019 auf martinfowler.com, *Don't get locked up into avoiding lock-in*, argumentiert, dass genau das selbst eine Falle ist. Er unterscheidet mehrere Arten von Lock-in: Anbieter (Verträge und kommerzielle Bedingungen), Produkt (die APIs und Funktionen eines bestimmten Produkts, auch wenn es Open Source ist), Version, Architektur (ein Stil wie Serverless), Plattform (Konten, Identitäts- und Sicherheitsmodelle), Fähigkeiten (was Ihre Leute können), rechtlich

(Compliance und Lizenzierung) und mental (Annahmen, die Sie davon abhalten, Alternativen zu erwägen). Open-Source-Software verringert den Anbieter-Lock-in, lässt die meisten anderen Arten aber bestehen.

Der entscheidende Schritt ist, Lock-in als Problem der Optionsbewertung zu behandeln. Die Wechselkosten sind das, was ein Umzug kosten würde; multipliziert mit der Wahrscheinlichkeit, jemals umzuziehen, ergeben sie die erwarteten Kosten des Lock-ins. Die Wechselkosten zu senken hat seinen eigenen Preis: zusätzliche Abstraktion, Entwürfe auf dem kleinsten gemeinsamen Nenner und Verzicht auf Funktionen. Ein Wechsel, der 50.000 kosten würde und mit 5 % Wahrscheinlichkeit eintritt, hat erwartete Kosten von 2500, daher ist es ein schlechtes Geschäft, weit mehr auszugeben, um ihn zu vermeiden. Hohpes Matrix aus einzigartigem Nutzen und Wechselkosten macht daraus eine schnelle Einordnung.

Einzigartiger Nutzen und Wechselkosten



Nach Gregor Hohpe (2019): Lock-in ist akzeptabel, wenn er einzigartigen Nutzen bringt; seien Sie vorsichtig bei teuren Abhängigkeiten, die wenig zurückgeben.

Die Kosten der Portabilität und die Exit-Strategie

Portabilität ist nicht kostenlos. Jede Komponente so zu entwerfen, dass sie auf mehreren Clouds läuft, bedeutet, nur Funktionen zu nutzen, die alle gemeinsam haben, Abstraktionsschichten zu bauen und zu testen, die undicht sind (siehe Information Hiding und Leaky Abstractions), Know-how für mehrere Plattformen vorzuhalten und auf Managed Services zu verzichten, die Aufwand gespart hätten. Viele Multi-Cloud-Strategien enden mit den Kosten aller ihrer Anbieter und den Vorteilen keines einzigen. Am günstigsten ist Portabilität dort, wo es bereits offene Standards gibt: OCI-Container-Images, Kubernetes-Manifeste, PostgreSQL-kompatible Datenbanken, S3-kompatible Object-Storage-APIs,

Standardprotokolle wie HTTP, AMQP und das Kafka-Protokoll sowie OpenTelemetry für Observability. Selbst die Werkzeuge, mit denen man portabel bleibt, etwa Terraform oder OpenTofu, sind für sich genommen Abhängigkeiten.

In der Praxis ist umstritten, wie weit man gehen sollte. Befürworter der Portabilität verweisen auf Preiserhöhungen, Ausfälle, Sanktionen, Regulierung und die Fälle, in denen sich der Auszug aus der Cloud gelohnt hat (siehe Cloud-Repatriierung in Abschnitt 10.1). Befürworter der Bindung an einen Anbieter verweisen auf die Geschwindigkeit und Zuverlässigkeit, die die volle Nutzung von Managed Services bringt. Eine solide Mittelposition ist, Lock-in bewusst dort zu akzeptieren, wo er klaren Nutzen bringt, die Geschäftslogik an den wenigen Nahtstellen, die sich am ehesten ändern, hinter Ports and Adapters unabhängig von Anbieter-APIs zu halten, Daten in exportierbaren Formaten zu speichern und eine Exit-Strategie schriftlich festzuhalten.

Regulierung verlangt diesen letzten Schritt zunehmend. Die EU-Verordnung über die digitale operationale Resilienz im Finanzsektor (DORA, Verordnung (EU) 2022/2554, anwendbar seit dem 17. Januar 2025; nicht zu verwechseln mit den DORA-Metriken für die Auslieferung) verpflichtet Finanzunternehmen zu Exit-Strategien für IKT-Dienstleistungen, die kritische oder wichtige Funktionen unterstützen. Die Regeln der EU-Datenverordnung (Data Act) zum Wechsel von Cloud-Anbietern gelten seit dem 12. September 2025: Kunden dürfen den Anbieter mit einer Frist von höchstens zwei Monaten wechseln, und Anbieter dürfen bis zum 12. Januar 2027 nur ihre direkten Wechselkosten berechnen, einschließlich Datenabfluss, danach gar nichts mehr. 2024 kündigten die größten Anbieter außerdem unter Bedingungen kostenlose Datenübertragung für Kunden an, die sie verlassen. Eine Exit-Strategie sollte für jeden kritischen Dienst den Ersatz benennen, Zeit und Kosten des Umzugs schätzen, zeigen, dass sich Daten in einem nutzbaren Format exportieren lassen, und zumindest teilweise geprobt werden, zum Beispiel durch das Wiederherstellen eines Backups in einer anderen Umgebung.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Kosten pro Bestellung und drei Architekturänderungen

Die Essenslieferplattform wickelt 1,2 Millionen Bestellungen im Monat ab. Ihre monatliche Cloud-Rechnung beträgt 96.000: Rechenleistung 42.000, Managed Databases 18.000, Observability 14.000, Datenübertragung 12.000, sonstige Managed Services 6000 und Speicher 4000. Alle Zahlen sind Beispielwerte in Pfund.

Ausgangslage	96.000 / 1.200.000 Bestellungen = 0,080 pro Bestellung. Showback zeigt, dass die Compute-Requests etwa doppelt so hoch sind wie der gemessene Verbrauch, dass die Hälfte des Log-Volumens Debug-Ausgaben sind und dass der Bestell- und der Preisgestaltungsservice pro Bestellung etwa 40 Aufrufe über Availability Zones hinweg austauschen.
Änderung 1: erst Right-sizing, dann Zusage	Requests, die anhand des gemessenen p95-Verbrauchs gesetzt werden, senken die Zahl der Nodes um 25 %: Rechenleistung von 42.000 auf 31.500. Die stabilen 60 % davon (18.900) werden durch eine einjährige Zusage mit etwa 30 % Rabatt abgedeckt, was 5670 spart. Die Rechenleistung kostet nun 25.830, eine Einsparung von 16.170.
Änderung 2: weniger loggen, kürzer aufbewahren	Debug-Logging in Produktion aus, erfolgreiche Anfragen zu 10 % gesampelt, Hot-Aufbewahrung von 90 auf 14 Tage verkürzt, mit einer Archivstufe. Die Aufnahme sinkt um etwa 60 %: Observability von 14.000 auf 6500, eine Einsparung von 7500.
Änderung 3: nicht mehr für Entfernung zahlen	Preislogik für den Hot Path in den Prozess des Bestellservice verlagert (womit die meisten der 40 Aufrufe entfallen), zonenbewusstes Routing für die übrigen Aufrufe und Restaurantbilder über ein CDN statt vom Ursprung ausgeliefert. Datenübertragung von 12.000 auf 6000; das CDN kostet 1000 zusätzlich. Nettoeinsparung 5000.
Ergebnis	96.000 minus 28.670 = 67.330 im Monat, also 0,056 pro Bestellung: 30 % weniger pro Bestellung, ohne Änderung an Features oder Zuverlässigkeitszielen.
Was sich nicht geändert hat	Datenbank- und Speicherkosten blieben in diesem Quartal unangetastet, weil die dort möglichen Einsparungen im Verhältnis zum Risiko klein waren; sie bleiben im Backlog.

Die größte strukturelle Änderung war architektonisch (eine gesprächige Grenze entfernen und ein CDN einführen), die Logging-Änderung war eine Designentscheidung zur Observability, und der Preisrabatt kam erst nach dem Rightsizing, sodass sich das Unternehmen nicht auf Verschwendung festlegte.

Wer die Kosten pro Bestellung statt der Gesamtrechnung verfolgt, sieht, ob künftiges Wachstum die Verbesserung hält: Wenn sich die Bestellungen verdoppeln und die Kosten pro Bestellung nahe 0,056 bleiben, hat die Effizienz gehalten.

Häufige Fragen

Sollte jedes Team mit seinen Cloud-Kosten belastet werden?
Beginnen Sie mit Showback, damit Teams ihre Kosten sehen und handeln können, und wechseln Sie erst zu Chargeback, wenn die Zuordnung genau ist und gemeinsame Kosten nach Regeln aufgeteilt werden, die alle akzeptieren; ungenaue Zahlen in Rechnung zu stellen erzeugt Streit, keine Einsparungen.

Ist Multi-Cloud der beste Schutz vor Lock-in?
Für sich allein selten. Dasselbe System auf mehreren Clouds zu betreiben ist teuer und beschränkt Sie auf gemeinsame Funktionen. Verschiedene Anbieter für verschiedene, unabhängige Workloads zu nutzen, plus eine getestete Exit-Strategie für die kritischen, bietet meist den Großteil des Schutzes zu einem Bruchteil der Kosten.

Wann ist Kostenoptimierung verfrüht?
Wenn das Produkt seinen Markt noch nicht gefunden hat und Engineering-Zeit die knappste Ressource ist. Selbst dann sind Tagging vom ersten Tag an und grobe Kosten pro Einheit günstig und machen spätere Optimierung weit einfacher.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum sind die Kosten pro Bestellung nützlicher als die gesamte Monatsrechnung?**

Sie trennen Effizienz von Wachstum: Eine steigende Rechnung bei sinkenden Kosten pro Bestellung ist gesund, eine gleichbleibende Rechnung bei sinkenden Bestellzahlen nicht.

▼ **Warum sollte Rightsizing vor dem Kauf von Rabatten für Nutzungszusagen kommen?**

Eine Zusage legt ein Ausgabenniveau für ein oder drei Jahre fest; wer sich festlegt, bevor die Überdimensionierung beseitigt ist, verwandelt Verschwendung in eine langfristige Verpflichtung.

▼ **Welche Art von Workload eignet sich für Spot-Kapazität, und warum?**

Zustandslose, unterbrechbare und wiederholbare Arbeit wie Batch-Jobs und CI-Runner, weil der Anbieter Spot-Kapazität mit Sekunden oder Minuten Vorlauf zurückfordern kann.

▼ **Wann ist Lock-in laut Hohpe akzeptabel?**

Wenn er einzigartigen Nutzen bringt, der die Wechselkosten wert ist, oder wenn ein Wechsel so unwahrscheinlich ist, dass die erwarteten Kosten niedriger sind als die Kosten, den Lock-in zu vermeiden.

▼ **Probieren Sie es aus: Die Rechnung beträgt £60.000 im Monat für 1.500.000 Bestellungen: Rechenleistung £30.000, Logging £9000. Rightsizing senkt die Rechenleistung um 30 %; der Verzicht auf Debug-Logs halbiert das Logging. Ermitteln Sie die Kosten pro Bestellung vorher und nachher. Was sollte vor jedem Rabatt für Nutzungszusagen kommen, und warum?**

Vorher: £60.000 / 1.500.000 = 4,0 Pence. Die Einsparungen betragen £9000 und £4500, die Rechnung sinkt also auf £46.500 oder 3,1 Pence. Führen Sie zuerst das Rightsizing durch, denn ein Commitment auf überdimensionierte Kapazität schreibt die Verschwendung für die gesamte Laufzeit fest. Committen Sie dann nur die gemessene Grundlast. Melden Sie den Teams die Kosten pro Bestellung per Showback, denn eine steigende Summe kann schlicht mehr Geschäft bedeuten, während steigende Stückkosten ein echtes Problem anzeigen.

QUELLEN

- FinOps Foundation: FinOps Framework: https://www.finops.org/framework/
- Gregor Hohpe: Don't get locked up into avoiding lock-in, martinfowler.com: https://martinfowler.com/articles/oss-lockin.html
- AWS Well-Architected Framework: Cost Optimization Pillar: https://docs.aws.amazon.com/wellarchitected/latest/cost-optimization-pillar/welcome.html
- Regulation (EU) 2023/2854 (Data Act), EUR-Lex: https://eur-lex.europa.eu/eli/reg/2023/2854/oj

Weiterführende Literatur

- *The NIST Definition of Cloud Computing (SP 800-145)*, Peter Mell and Timothy Grance (2011), National Institute of Standards and Technology. Die kurze, viel zitierte Definition der wesentlichen Merkmale, Servicemodelle und Bereitstellungsmodelle.
  https://csrc.nist.gov/pubs/sp/800/145/final
- *The Twelve-Factor App*, Adam Wiggins (2011), 12factor.net (Heroku; community-maintained since 2024). Zwölf knappe Praktiken für deploybare, skalierbare Anwendungen; noch immer das gemeinsame Vokabular des Cloud-nativen Anwendungsdesigns.
  https://12factor.net/
- *Kubernetes: Up and Running (3rd edition)*, Brendan Burns, Joe Beda, Kelsey Hightower and Lachlan Evenson (2022), O'Reilly Media. Eine klare Einführung in Kubernetes-Objekte und den Betrieb von Menschen, die an seiner Entstehung beteiligt waren.
- *Infrastructure as Code (3rd edition)*, Kief Morris (2025), O'Reilly Media. Das Standardwerk über Entwurf, Test und Auslieferung von Infrastrukturcode, unabhängig von einem bestimmten Werkzeug.
- *Continuous Delivery*, Jez Humble and David Farley (2010), Addison-Wesley. Der grundlegende Text über Deployment-Pipelines, „build once, deploy many“ und darüber, Releases zur Routine zu machen.
- *Accelerate: The Science of Lean Software and DevOps*, Nicole Forsgren, Jez Humble and Gene Kim (2018), IT Revolution. Die Forschung hinter den DORA-Metriken, die Auslieferungspraktiken und lose gekoppelte Architektur mit der Leistung von Organisationen verbindet.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (eds) (2016), O'Reilly Media (free online). Wie ein großer Betreiber Systeme mit Automatisierung, Error Budgets und Release Engineering betreibt.
  https://sre.google/books/
- *Don't get locked up into avoiding lock-in*, Gregor Hohpe (2019), martinfowler.com. Eine ausgewogene Analyse der Arten von Lock-in und der wirklichen Kosten, sie zu vermeiden.
  https://martinfowler.com/articles/oss-lockin.html

Glossar

Active-Active (Multi-Site)

Eine Anordnung, bei der zwei oder mehr Regionen gleichzeitig Live-Traffic bedienen, sodass der Verlust einer Region kein Failover erfordert, sondern nur eine Umverteilung des Traffics.

Artefakt-Repository
Ein Speicher für gebaute, versionierte Ergebnisse wie Container-Images und Pakete, aus dem jede Umgebung dieselbe Datei deployt.

Automatisierte Canary-Analyse
Statistischer Vergleich der Metriken eines Canary mit denen einer Baseline, die die alte Version ausführt; die Pipeline nutzt ihn, um ohne menschliches Urteil freizugeben oder zurückzurollen.

Backing Service
Jeder Dienst, den eine Anwendung über das Netzwerk nutzt, etwa eine Datenbank, Queue, ein Cache oder eine E-Mail-API, behandelt als angebundene Ressource, die sich per Konfiguration austauschen lässt.

Blue-Green-Deployment
Der Betrieb zweier vollständiger Produktionsumgebungen, wobei der gesamte Traffic auf einen Schlag von der alten (Blue) auf die neue (Green) umgeschaltet wird und die alte für ein sofortiges Zurückschalten bereitbleibt.

Build once, deploy many
Die Regel, dass eine Änderung genau einmal zu einem unveränderlichen Artefakt gebaut wird und dasselbe Artefakt durch alle Umgebungen befördert wird, wobei sich nur die Konfiguration unterscheidet.

Build, Release, Run
Die strikte Trennung zwischen dem Umwandeln von Code in ein Artefakt (Build), dessen Kombination mit der Konfiguration einer Umgebung (Release) und seiner Ausführung (Run).

Canary Release
Ein kleiner Anteil des echten Traffics wird zuerst an eine neue Version geleitet, ihr Verhalten mit der alten Version verglichen und der Anteil nur erhöht, wenn sie fehlerfrei läuft.

Chargeback
Die Belastung des Budgets jedes Teams oder Produkts mit den ihm zugeordneten Kosten, sodass die Ausgaben in seiner eigenen Rechnung erscheinen.

Cloud-native
Ein Ansatz, Software als lose gekoppelte, automatisierte, beobachtbare und resiliente Systeme zu bauen und zu betreiben, die programmatisch und wiederholbar in öffentlichen, privaten oder hybriden Clouds deployt werden. Er beschreibt, wie ein System gebaut ist, nicht, wo es läuft.

Cloud-Repatriierung
Die Rückverlagerung von Workloads aus einer Public Cloud auf eigene oder in Colocation betriebene Hardware, meist weil eine gleichmäßige Last den Eigenbetrieb günstiger macht.

Container-Orchestrierung
Das automatische Verteilen von Containern auf Maschinen und ihr Am-Laufen-Halten: Neustart nach Ausfällen, Skalierung, Ausrollen neuer Versionen und Anbindung an Netzwerke und Speicher.

Containers as a Service (CaaS)
Ein Cloud-Dienst, der Container-Images für Sie ausführt, entweder als verwaltetes Kubernetes oder als serverlose Container-Plattform, bei der Sie die Maschinen nie sehen.

Control Plane
Die Komponenten, die Entscheidungen für einen Cluster oder ein System treffen (in Kubernetes: API-Server, etcd, Scheduler und Controller), im Gegensatz zu den Knoten, auf denen die Workloads laufen.

Datenabfluss (Egress)
Daten, die das Netzwerk eines Cloud-Anbieters verlassen oder zwischen seinen Zonen oder Regionen wechseln, was meist pro Gigabyte berechnet wird.

Deployment (Kubernetes)
Ein Kubernetes-Objekt, das festlegt, wie viele identische Pods einer zustandslosen Anwendung laufen sollen, und neue Versionen schrittweise über ReplicaSets ausrollt.

Deployment-Pipeline
Die automatisierte Abfolge von Stufen (Build, Tests, Paketierung, Deployment, Verifikation), die jede Änderung auf dem Weg vom Commit bis in die Produktion durchläuft.

Disaster Recovery (DR)
Die Pläne, Kopien und Verfahren, die einen Dienst und seine Daten nach einem großen Ausfall wiederherstellen, etwa nach dem Verlust eines ganzen Rechenzentrums oder einer ganzen Region.

Disposability
Die Eigenschaft von Prozessen, schnell zu starten, sauber herunterzufahren und jederzeit ohne Schaden beendet werden zu können, sodass sie frei ersetzt und skaliert werden können.

Edge Function
Ein kleines Stück Code, das ein CDN- oder Edge-Anbieter an Standorten nahe bei den Benutzern ausführt, um Anfragen zu bearbeiten oder zu verändern, bevor (oder statt dass) sie den Ursprungsserver erreichen.

Entwicklerportal
Eine Weboberfläche wie Backstage, die einen Softwarekatalog, Vorlagen, Dokumentation und Links zu Plattformdiensten an einem Ort bündelt.

Exit-Strategie
Ein dokumentierter und idealerweise geprobter Plan für den Ausstieg bei einem Anbieter oder Produkt, der Datenexport, Ersatzdienste, Verträge, Zeit und Kosten abdeckt.

FinOps
Eine Praxis, bei der Engineering-, Finanz- und Fachteams gemeinsam Verantwortung für Cloud- und Technologieausgaben tragen und anhand zeitnaher Kostendaten Entscheidungen treffen, die den Geschäftswert maximieren.

Gateway API
Die Kubernetes-API zum Routing von externem und internem Traffic zu Services, seit 2023 allgemein verfügbar und der empfohlene Nachfolger von Ingress.

Gewünschter Zustand (Desired State)
Eine deklarative Beschreibung, wie ein System aussehen soll (zum Beispiel drei Kopien der Version 2.4), die eine Automatisierung mit dem tatsächlichen Zustand vergleicht und herzustellen versucht.

GitOps
Systeme betreiben, indem ihr deklarativer gewünschter Zustand in Git gehalten wird und Agenten ihn abrufen und das laufende System fortlaufend daran angleichen.

Golden Path (Paved Road)
Ein meinungsstarker, unterstützter und dokumentierter Weg für eine häufige Aufgabe, etwa das Anlegen eines neuen Service, der der einfachste, aber nicht der einzige erlaubte Weg ist.

Guardrail
Eine automatische Grenze oder Prüfung, die die Self-Service-Nutzung sicher hält (zum Beispiel eine Richtlinie, die öffentliche Storage-Buckets blockiert), ohne dass eine Person jede Anfrage genehmigen muss.

Horizontal Pod Autoscaler (HPA)
Ein Kubernetes-Controller, der die Zahl der Pod-Replikate ändert, um eine Metrik, etwa die durchschnittliche CPU-Nutzung im Verhältnis zu den Requests, nahe an einem Zielwert zu halten.

Hybrid Cloud
Eine Kombination aus Private und Public Cloud (oder On-Premises-Systemen), die so verbunden ist, dass Daten und Anwendungen übergreifend arbeiten können.

Immutable Infrastructure
Die Praxis, einen laufenden Server oder Container nie direkt zu verändern; jede Änderung erfolgt durch das Bauen eines neuen Images und das Ersetzen der alten Instanzen.

Infrastructure as Code (IaC)
Die Definition von Servern, Netzwerken, Datenbanken und anderer Infrastruktur in versionskontrollierten Dateien, die Werkzeuge automatisch anwenden, statt sie von Hand zu konfigurieren.

Internal Developer Platform (IDP)
Die Gesamtheit der Self-Service-Werkzeuge, -Dienste, -Vorlagen und -Dokumentation, die das Plattformteam einer Organisation bereitstellt, damit Produktteams Software ohne Tickets bauen und betreiben können.

Kognitive Last
Der geistige Aufwand, den eine Person oder ein Team aufbringen muss, um ein System zu verstehen und darin zu arbeiten; Team Topologies nutzt ihn, um den Zuschnitt der Teamverantwortlichkeiten zu bemessen.

Konfigurationsdrift
Abweichungen, die sich zwischen der im Code beabsichtigten Konfiguration und der realen Infrastruktur aufbauen, meist durch manuelle Änderungen oder unvollständige Updates.

Konfigurationsmanagement
Werkzeuge wie Ansible, Chef und Puppet, die bestehende, langlebige Server in eine beschriebene Konfiguration bringen, indem sie Pakete installieren und Dateien auf ihnen bearbeiten.

Kubernetes
Der am weitesten verbreitete Open-Source-Container-Orchestrator, entstanden bei Google, 2014 angekündigt und 2015 als Version 1.0 veröffentlicht; er sorgt dafür, dass der tatsächliche Zustand eines Clusters einem deklarierten gewünschten Zustand entspricht.

Lock-in
Die Kosten und Schwierigkeiten, von einem Anbieter, Produkt, einer Plattform oder Technologie wegzuwechseln, sobald ein System davon abhängt.

Modell der geteilten Verantwortung (Shared Responsibility)
Die Aufteilung der Sicherheits- und Betriebspflichten zwischen einem Cloud-Anbieter, der die Cloud selbst absichert, und dem Kunden, der seine Daten, Identitäten und Konfiguration in der Cloud absichert.

Multi-Cloud
Die Nutzung von Diensten zweier oder mehrerer Public-Cloud-Anbieter, etwa wegen besonderer Dienste, Übernahmen, Verhandlungsmacht oder regulatorisch geforderter Exit-Pläne.

Open Container Initiative (OCI)
Ein 2015 gegründetes Projekt der Linux Foundation, das offene Standards für Container-Image-Formate, Laufzeitumgebungen und Registries pflegt, sodass Werkzeuge verschiedener Anbieter zusammenarbeiten.

OpenTofu
Ein Open-Source-Fork von Terraform, entstanden 2023, nachdem HashiCorp Terraform unter die Business Source Licence gestellt hatte; unter dem Dach der Linux Foundation und ein CNCF-Projekt.

Pilot Light
Eine Disaster-Recovery-Strategie, die Daten repliziert und die Kerninfrastruktur in einer zweiten Region bereithält, die Anwendungsserver aber erst im Katastrophenfall startet und skaliert.

Platform Engineering
Die Disziplin, interne Plattformen zu bauen und zu betreiben, über die Produktteams Software per Self-Service bereitstellen, deployen und betreiben können.

Pod
Die kleinste deploybare Einheit in Kubernetes: ein oder mehrere Container, die gemeinsam auf einem Knoten eingeplant werden und sich eine Netzwerkadresse und Speicher-Volumes teilen.

Policy as Code
Das Formulieren von Regeln für Infrastruktur und Deployments (kein öffentlicher Speicher, Verschlüsselung vorgeschrieben, nur genehmigte Regionen) als Code, den Werkzeuge vor oder während Änderungen automatisch prüfen.

Private Cloud
Cloud-Infrastruktur, die einer einzigen Organisation vorbehalten ist und Self-Service-Bereitstellung über APIs bietet; sie kann On-Premises stehen oder von einem Drittanbieter gehostet werden.

Probe (Liveness, Readiness, Startup)
Eine regelmäßige Prüfung eines Containers durch Kubernetes: Liveness entscheidet, ob er neu gestartet wird, Readiness, ob er Datenverkehr erhält, Startup, ob er fertig gestartet ist.

Progressive Delivery
Das Ausliefern einer Änderung an schrittweise größere Nutzergruppen mit automatischen Prüfungen bei jedem Schritt, sodass eine fehlerhafte Änderung möglichst wenige Nutzer erreicht, bevor sie gestoppt wird.

Rabatt für Nutzungszusagen
Ein niedrigerer Preis im Gegenzug für die Zusage einer bestimmten Nutzung oder Ausgabenhöhe für ein oder drei Jahre, etwa Reserved Instances, Savings Plans oder Committed Use Discounts.

Reconciliation Loop
Der endlose Zyklus eines Controllers, der den tatsächlichen Zustand beobachtet, ihn mit dem gewünschten Zustand vergleicht und handelt, um den Unterschied zu beseitigen.

Resource Requests und Limits
Einstellungen für CPU und Arbeitsspeicher pro Container in Kubernetes: Requests reservieren Kapazität und steuern das Scheduling; Limits begrenzen die Nutzung, drosseln die CPU und beenden Container, die den Speicher überschreiten.

Rightsizing
Die Anpassung der Größe bereitgestellter Ressourcen (Maschinentypen, Container-Requests, Datenbankstufen) an das, was ein Workload tatsächlich nutzt.

Roll-forward
Die Behebung eines fehlerhaften Release durch das Deployment einer neuen Korrektur statt der Rückkehr zur vorherigen Version.

Rolling Deployment
Das Ersetzen der Instanzen eines Service in kleinen Gruppen, sodass alte und neue Version parallel laufen, bis alle Instanzen die neue Version ausführen.

Service (Kubernetes)
Ein Kubernetes-Objekt, das einer wechselnden Menge von Pods einen stabilen Namen und eine virtuelle Adresse gibt und den Datenverkehr auf sie verteilt.

Shadow Traffic (Dark Launch)
Das Kopieren echter Anfragen an eine neue Version, deren Antworten verglichen oder verworfen, aber nie an Nutzer zurückgegeben werden, um sie unter realer Last ohne Auswirkungen auf Nutzer zu testen.

Showback
Die Ausweisung der jedem Team oder Produkt zugeordneten Kosten, ohne dessen Budget tatsächlich zu belasten.

Shuffle Sharding
Die Zuordnung jedes Kunden zu einer kleinen, zufällig gewählten Kombination von Ressourcen, sodass zwei Kunden selten alle Ressourcen gemeinsam haben und ein problematischer Kunde nur wenige andere beeinträchtigt.

SPACE-Framework
Ein Framework von 2021 für die Produktivität von Entwicklern, das Zufriedenheit und Wohlbefinden, Leistung, Aktivität, Kommunikation und Zusammenarbeit sowie Effizienz und Flow abdeckt.

Spot-Kapazität
Freie Cloud-Kapazität, die mit hohem Rabatt verkauft wird und die der Anbieter kurzfristig zurückfordern kann; nur für Arbeit geeignet, die Unterbrechungen verträgt.

State (Infrastructure as Code)
Die Aufzeichnung, die ein Infrastrukturwerkzeug darüber führt, welche realen Ressourcen welchen Definitionen im Code entsprechen, und anhand derer es ermittelt, was zu erstellen, zu ändern oder zu löschen ist.

Statische Stabilität
Die Eigenschaft, dass ein System während eines Ausfalls weiterarbeitet, ohne im Moment des Ausfalls Änderungen vornehmen zu müssen, etwa neue Server zu starten oder die Konfiguration zu aktualisieren.

Thinnest Viable Platform

Die kleinste Menge gemeinsamer Dienste und Dokumentation, die Produktteams tatsächlich schneller macht; die Warnung von Team Topologies davor, mehr Plattform zu bauen als nötig.

Trunk-Based Development

Eine Arbeitsweise, bei der Entwickler kleine Änderungen mindestens täglich in einen gemeinsamen Hauptbranch integrieren, statt langlebige Feature-Branches zu pflegen.

Twelve-Factor App

Eine Methodik aus zwölf Praktiken für den Bau deploybarer, skalierbarer Webanwendungen, 2011 von Adam Wiggins bei Heroku verfasst und 2024 als Open Source freigegeben.

Unit Economics

Die Angabe von Kosten (und Erlösen) pro Einheit geschäftlichen Werts, etwa Kosten pro Bestellung oder pro aktivem Kunden, damit sich die Effizienz unabhängig vom Wachstum beurteilen lässt.

Warm Standby

Eine Disaster-Recovery-Strategie, die eine verkleinerte, aber voll funktionsfähige Kopie des Systems in einer zweiten Region laufen lässt, bereit, Datenverkehr zu übernehmen und hochzuskalieren.

Wechselkosten (Switching Cost)

Der gesamte Aufwand, das Geld und das Risiko, die nötig sind, um von einer Technologie oder einem Anbieter zu einem anderen zu wechseln.

Zellbasierte Architektur

Der Betrieb vieler vollständiger, unabhängiger Kopien (Zellen) eines Systems hinter einem schlanken Router, von denen jede eine feste Teilmenge der Kunden bedient, sodass ein Ausfall nur eine Zelle betrifft.