BE SOFTWARE ARCHITECTURE WISE · KAPITEL 6

Architekturstile II: Verteilte Systeme

Dieses Kapitel führt die Architektur über das Netzwerk hinaus. Es erklärt, wann sich die Verteilung eines Systems lohnt, welche Trugschlüsse sie schwieriger machen, als sie aussieht, und welche verteilten Hauptstile es gibt: servicebasierte Architektur, serviceorientierte Architektur, Microservices, ereignisgesteuerte Architektur, Serverless und Space-Based Architecture. Danach behandelt es die Mechanismen, auf die sie alle angewiesen sind: synchrone und asynchrone Kommunikation und API-Design, Konsistenzmodelle und das CAP-Theorem sowie verteilte Transaktionen mit Sagas und dem Outbox-Muster. Es richtet sich an Entwicklungs- und Architekturfachleute, die Systeme aus vielen Services entwerfen oder prüfen, und es ist präzise bei Ausfällen, Zustellgarantien und Konsistenz.

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

AUF EINEN BLICK

Das lernen Sie

- Die legitimen Gründe für die Verteilung eines Systems erklären und die Kosten, die jeder Grund mit sich bringt.
- Die acht Fallacies of Distributed Computing und den Teilausfall beschreiben und Antworten darauf entwerfen.
- Verfügbarkeit und Latenz einer Kette synchroner Aufrufe berechnen und Änderungen vorschlagen, die beide verbessern.
- Servicebasierte Architektur, klassische SOA und Microservices vergleichen und für eine gegebene Organisation zwischen ihnen wählen.
- Die Granularität von Services festlegen, indem man Disintegratoren gegen Integratoren abwägt, und den verteilten Monolithen erkennen.
- Ereignisse, Commands und Nachrichten, Broker- und Mediator-Topologie sowie Fowlers vier Event-Muster unterscheiden.
- Zustellgarantien erklären, warum „Exactly once“ sorgfältig definiert werden muss und wie idempotente Consumer funktionieren.
- Kommunikationsstile, Konsistenzstufen und Transaktionsmuster (Sagas, Outbox) für verteilte Abläufe wählen.

Bevor Sie beginnen

Kapitel 0 bis 5, besonders Verfügbarkeit in Reihe und parallel (Kapitel 2, Abschnitt 2.5), Kopplung und das Gesetz von Conway (Kapitel 3) sowie das Architekturquantum, der modulare Monolith und Bounded Contexts (Kapitel 5).

Dauer

Etwa 3½ Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

Verteiltes System · Teilausfall · Fallacies of Distributed Computing · Zeitliche Kopplung · Servicebasierte Architektur · Enterprise Service Bus · Microservices · Unabhängige Deploybarkeit · Datenbank pro Service · Ereignisgesteuerte Architektur · At-least-once-Zustellung · Idempotenter Consumer · CAP-Theorem · Saga

Warum dieses Kapitel wichtig ist

Kapitel 5 hat Stile beschrieben, bei denen das ganze System als eine Einheit deployt wird. Dieses Kapitel stellt dieselben Fragen über ein Netzwerk hinweg. Fast jedes System nennenswerter Größe ist heute in irgendeiner Form verteilt: Eine mobile App spricht mit Backend-Services, die Zahlungsanbieter, Kartendienste und Message Broker aufrufen, und die Services selbst laufen auf vielen Rechnern. Ein aufgeteiltes System kann unabhängige Skalierung, Fehlerisolation, unabhängige Releases und autonome Teams ermöglichen. Es bringt aber auch eine Klasse von Problemen mit sich, die innerhalb eines Prozesses nicht

auftreten können: verlorene oder doppelte Nachrichten, Services, die langsam statt tot sind, Daten, die zwischen Kopien voneinander abweichen, und Ausfälle von Rechnern, von denen Sie gar nicht wussten, dass Sie von ihnen abhängen.

Die wichtigste Idee, die Sie mitnehmen sollten: **Verteilung ist ein Tauschgeschäft, kein Upgrade**. Jeder Nutzen wird mit Latenz, Konsistenz, Observability und Betriebsaufwand bezahlt, und eine gute Architektin oder ein guter Architekt kann sagen, welcher Nutzen gekauft wird, zu welchem Preis und warum eine günstigere Lösung innerhalb eines Monolithen nicht genügt. Die Fallacies of Distributed Computing und die Rechnung für synchrone Aufrufketten im ersten Abschnitt sind die Werkzeuge, um dieses Tauschgeschäft zu bewerten, bevor man sich darauf festlegt.

Wie die Abschnitte zusammenhängen

Das Kapitel geht vom Warum zum Wie. Abschnitt 6.1 legt die Gründe für eine Verteilung dar, die acht Trugschlüsse, den Teilausfall und die Kosten, und zeigt an einem ausgearbeiteten Beispiel, wie fünf Services mit jeweils 99,9 % Verfügbarkeit einen Anfragepfad von nur etwa 99,5 % ergeben. Zum Schluss greift er das Architekturquantum wieder auf, das erklärt, warum synchrone Aufrufe separat deployte Services aneinanderbinden.

Die nächsten drei Abschnitte behandeln die verteilten Hauptstile, jeweils wie die Stile in Kapitel 5: Definition und Herkunft, ein Diagramm, Funktionsweise, ein Beispiel, eine Tabelle der Merkmale, wann man den Stil einsetzt und die üblichen Anti-Patterns. Abschnitt 6.2 behandelt die servicebasierte Architektur, einen pragmatischen Mittelweg aus wenigen grobgranularen Domain Services, die sich oft eine Datenbank teilen, und die klassische serviceorientierte Architektur mit Enterprise Service Bus und kanonischen Modellen, einschließlich der Frage, warum sie in Ungnade fiel. Abschnitt 6.3 behandelt Microservices: an Bounded Contexts ausgerichtete Services, unabhängige Deploybarkeit, Datenbank pro Service, unterstützende Muster wie API-Gateways und Service Meshes, die Wahl der Granularität sowie den verteilten Monolithen und andere Anti-Patterns. Abschnitt 6.4 behandelt die ereignisgesteuerte Architektur: Ereignisse und Commands, Broker- und Mediator-Topologie, Partitionen und Consumer Groups in Kafka, Zustellgarantien und idempotente Consumer.

Die zweite Hälfte wendet sich den Mechanismen zu, auf die jeder verteilte Stil angewiesen ist. Abschnitt 6.5 vergleicht synchrone und asynchrone Kommunikation und die Kopplung, die jede erzeugt, und behandelt REST, gRPC, GraphQL und Messaging, Praktiken des API-Designs, Versionierung und Contract Testing. Abschnitt 6.6 handelt davon, was Leser replizierter Daten erwarten können: Konsistenzmodelle von der Linearisierbarkeit bis zur Eventual Consistency, das CAP-Theorem und was es wirklich besagt, PACELC und Quoren. Abschnitt 6.7 erklärt, warum Two-Phase Commit zwischen Services vermieden wird und wie Sagas, kompensierende Transaktionen und die Transactional Outbox Geschäftsprozesse auch ohne ihn korrekt halten.

Abschnitt 6.8 schließt mit zwei Stilen für besondere Lastprofile: Serverless-Funktionen, die auf null skalieren und pro Aufruf abgerechnet werden, und Space-Based Architecture, die Daten im Arbeitsspeicher hält, um extreme Lastspitzen wie den Vorverkaufsstart von Tickets zu überstehen.

Durchgehende Beispiele und wie Sie das Kapitel lesen

Zwei Beispiele kehren immer wieder. Die Essenslieferplattform aus früheren Kapiteln (Kunden, Restaurants, Kuriere, Zahlungen) zeigt gewöhnliches verteiltes Design: Sie erscheint als fünf Domain Services, als Menge von Microservices, als Ereignisfluss `OrderPlaced` und als Bestell-Saga. Ein Ticketsystem, das Plätze für eine Stadiontournee verkauft, zeigt, was sich unter extremer, sprunghafter Last ändert, wenn Zehntausende Menschen in derselben Sekunde dieselben wenigen Plätze wollen.

Das Kapitel ist als „Mittelstufe bis Fortgeschritten“ eingestuft. Es setzt die Kapitel 0 bis 5 voraus, insbesondere Verfügbarkeit in Reihe und parallel (Kapitel 2), Kopplung und das Gesetz von Conway (Kapitel 3) sowie das Architekturquantum, den modularen Monolithen und Bounded Contexts (Kapitel 5). Einsteiger können zuerst die einleitende Zusammenfassung jedes Abschnitts und die Kästen „Wenn das neu für Sie ist“ lesen. Erfahrene Fachleute finden Präzision dort, wo es darauf ankommt: was eine At-least-once-Zustellung für jeden Consumer bedeutet, was „Exactly once“ bedeutet und was nicht, und warum Eventual Consistency bewusst eingeplant und nicht nur hingenommen werden muss.

Was Sie mitnehmen sollten

Drei Gewohnheiten ziehen sich durch jeden Abschnitt. Erstens: Zählen Sie synchrone Abhängigkeiten, nicht Services. Die Zahl der Dinge, die alle im selben Moment funktionieren müssen, damit die Anfrage eines Nutzers gelingt, bestimmt Verfügbarkeit und Latenz. Zweitens: Gehen Sie davon aus, dass Nachrichten verloren gehen, sich verzögern, doppelt ankommen und die Reihenfolge wechseln, und legen Sie Empfänger idempotent aus. Drittens: Legen Sie fest, wem jedes Datenstück gehört, denn die Datenhoheit entscheidet, wo Transaktionen möglich sind und wo Konsistenz über Services hinweg gesteuert werden muss. Kapitel 7 baut diese dritte Gewohnheit zu einer vollständigen Behandlung der Datenarchitektur aus.

Warum verteilen? Die Fallacies of Distributed Computing

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.4 Wie Computer kommunizieren
- 2.5 Verfügbarkeit und Zuverlässigkeit
- 5.1 Stile und Muster

Begriffe von früher, die er verwendet: Architekturquantum, Latenz, Verfügbarkeit, Gesetz von Conway, Modularer Monolith

DIE IDEE IN KÜRZE

Stellen Sie sich ein kleines Restaurant vor, in dem ein Koch in einer Küche alles macht. Das ist einfach: Ist der Koch krank, bleibt das Restaurant geschlossen, aber zwischen Menschen geht nichts verloren. Stellen Sie sich nun eine Kette mit einer zentralen Bäckerei, einer Saucenküche und vielen Filialen vor, verbunden durch Lieferwagen. Sie kann weit mehr Gäste bedienen, und eine Küche kann schließen, ohne dass die anderen schließen müssen. Aber Lieferwagen stehen im Stau, Bestellungen gehen verloren, und wenn eine Lieferung nicht ankommt, weiß niemand, ob die Bäckerei die Bestellung nie erhalten hat oder der Wagen liegen geblieben ist. Verteilte Software funktioniert wie die Kette.

WAS SIE WISSEN MÜSSEN

- Ein verteiltes System ist eine Menge von Programmen auf verschiedenen Rechnern, die durch das Senden von Nachrichten über ein Netzwerk zusammenarbeiten, um etwas bereitzustellen, das Nutzer als einen einzigen Dienst wahrnehmen.
- Berechtigte Gründe für eine Verteilung sind Skalierung, Fehlerisolation, unabhängige Deploybarkeit, Teamautonomie, Geografie und Datenresidenz, unterschiedliche Technologieanforderungen und Sicherheitsisolation; für jeden gibt es innerhalb eines Monolithen einen günstigeren Teilersatz, den man zuerst prüfen sollte.
- Der Teilausfall ist das grundlegende Problem: Nach einem Timeout kann der Aufrufer nicht erkennen, ob die Anfrage verloren ging, der Aufgerufene ausfiel oder nur die Antwort verloren ging; deshalb müssen wiederholte Operationen idempotent sein.
- Die acht Fallacies of Distributed Computing (Deutsch und Kollegen bei Sun, mit Goslings achtem Punkt) sind falsche Annahmen über Zuverlässigkeit, Latenz, Bandbreite, Sicherheit, Topologie, Administration, Transportkosten und Homogenität.

- Verteilung kostet Latenz, Observability, Konsistenz und Betriebsaufwand, und diese Kosten wachsen ungefähr mit der Zahl der deploybaren Einheiten.
- In einer synchronen Kette multiplizieren sich die Verfügbarkeiten und addieren sich die Latenzen: Fünf Services mit 99,9 % ergeben etwa 99,5 %, rund 3,6 Stunden Ausfallzeit im Monat.
- Synchrone Aufrufe binden Services zur Laufzeit aneinander (zeitliche Kopplung); daher zählt die Zahl synchroner Hops in wichtigen User Journeys mehr als die Zahl der Services.

Was ein System zu einem verteilten System macht

Ein verteiltes System ist eine Menge von Programmen auf verschiedenen Computern, die durch das Senden von Nachrichten über ein Netzwerk zusammenarbeiten, um etwas bereitzustellen, das Nutzer als einen einzigen Dienst wahrnehmen. Ihre Essensliefer-App ist eines: Das Smartphone, die Bestell- und Zahlungsservices des Unternehmens, das Tablet des Restaurants, das Smartphone des Kuriers und ein Kartenzahlungsabwickler wirken alle an einer einzigen Bestellung mit. Kapitel 5 hat Stile beschrieben, die als eine Einheit deployt werden (Monolithen); in diesem Kapitel geht es um Stile, deren Teile separat deployt werden und über das Netzwerk kommunizieren.

Leslie Lamport, der mehr als jeder andere dazu beigetragen hat, das Gebiet auf ein mathematisches Fundament zu stellen, fasste seine wesentliche Schwierigkeit in einer internen E-Mail vom Mai 1987 zusammen: Ein verteiltes System ist eines, in dem „der Ausfall eines Computers, von dessen Existenz Sie nicht einmal wussten“, Ihren eigenen Computer unbrauchbar machen kann. Der Witz funktioniert, weil er wahr ist. Sobald ein System von Rechnern abhängt, die es weder kontrolliert noch sieht, tritt eine ganze Klasse von Ausfällen auf, die innerhalb eines Prozesses schlicht nicht vorkommen können.

Deshalb ist Verteilung eine Entscheidung mit einem Preis, kein Zeichen von Reife. Der Rest dieses Abschnitts legt die Gründe dar, die den Preis rechtfertigen, die falschen Annahmen, die dazu führen, dass man ihn unterschätzt, und einen Weg, einen Teil davon zu berechnen, bevor Sie zahlen.

Warum überhaupt verteilen

Es gibt eine Handvoll berechtigter Gründe, ein System auf Prozesse und Rechner aufzuteilen. Jeder ist ein Qualitätsmerkmal aus Kapitel 2, das eine einzelne Deployment-Einheit ab einer gewissen Größe nur schwer liefern kann. Für jeden gibt es außerdem einen günstigeren Teilersatz innerhalb eines Monolithen, und wer gut architektiert, prüft zuerst diesen Ersatz.

Gründe für eine Verteilung und was ein gut gebauter Monolith bereits leisten kann

Grund	Was die Verteilung bietet	Was ein einzelnes Deployment bereits leisten kann
Skalierung	Teile mit sehr unterschiedlicher Last lassen sich getrennt skalieren: Der Feed der Kurierstandorte erhält Tausende Updates pro Sekunde, die Onboarding-Masken für Restaurants ein paar pro Stunde.	Viele identische Replikate hinter einem Load Balancer betreiben, Read Replicas und Caches hinzufügen. Das genügt für die meisten Systeme und ist weit günstiger.
Fehlerisolation	Ein Speicherleck in den Empfehlungen kann den Checkout nicht zum Absturz bringen, weil beide in verschiedenen Prozessen auf verschiedenen Rechnern laufen.	Getrennte Thread-Pools und Bulkheads innerhalb eines Prozesses bieten teilweise Isolation; ein Absturz reißt trotzdem alles mit.
Unabhängige Deploybarkeit	Ein Team kann seinen Teil zwanzigmal am Tag ausliefern, ohne ein Release mit irgendjemand anderem abzustimmen.	Ein modularer Monolith mit guter Automatisierung kann ebenfalls viele Male am Tag ausliefern, aber jedes Release liefert alles aus.
Teamautonomie	Teams verantworten einen Service von Ende zu Ende, bestimmen ihr Tempo und haben Bereitschaft dafür; das Gesetz von Conway arbeitet für Sie statt gegen Sie.	Moduleigentümerschaft funktioniert gut bis zu einigen wenigen Teams; darüber hinaus werden gemeinsamer Build und gemeinsames Release zur Warteschlange.
Geografie und Datenresidenz	Teile nahe bei den Nutzern betreiben, um die Latenz zu senken, oder personenbezogene Daten in einer Rechtsordnung halten, weil ein Gesetz oder ein Vertrag es verlangt.	Pro Region eine vollständige Kopie deployen (eine Zelle). Das ist selbst eine Form der Verteilung, aber eine einfache.
Unterschiedliche Technologieanforderungen	Ein Empfehlungsmodell braucht GPUs, die Suche eine Suchmaschine, eine Altsystem-Integration eine bestimmte Laufzeitumgebung.	Die spezialisierte Komponente aufrufen und alles andere beisammenhalten.
Sicherheitsisolation	Kartendaten, die von einem kleinen, separat gehärteten und auditierten Service verarbeitet werden, verkleinern den Umfang einer PCI-DSS-Prüfung.	Eigentlich nicht: Alles innerhalb des Prozesses kann im Speicher auf alles andere zugreifen.

Aus der Tabelle folgen zwei Beobachtungen. Erstens sind die stärksten Gründe oft organisatorischer Natur: Unabhängige Deploybarkeit und Teamautonomie zählen am meisten, wenn viele Teams an einem Produkt arbeiten; deshalb taucht das Gesetz von Conway (Kapitel

3) in diesem Kapitel immer wieder auf. Zweitens ist Verteilung selten eine Frage von alles oder nichts. Viele erfolgreiche Systeme sind ein modularer Monolith plus zwei oder drei separate Services, die jeweils aus einem bestimmten, benannten Grund abgespalten wurden.

Teilausfall: das grundlegende Problem

Innerhalb eines Prozesses kehrt ein Funktionsaufruf entweder zurück, oder der ganze Prozess stirbt mit ihm. Über ein Netzwerk gibt es ein drittes Ergebnis, und es ändert alles: den Teilausfall, bei dem einige Teile des Systems funktionieren, während andere ausgefallen, langsam oder nicht erreichbar sind. Schlimmer noch: Der Aufrufer kann oft nicht erkennen, welcher Fall vorliegt.

Nehmen Sie den Bestellservice, der den Zahlungsservice bittet, £24,50 zu autorisieren. Kommt vor dem Timeout keine Antwort, weiß der Bestellservice nicht, ob die Anfrage unterwegs verloren ging, ob der Zahlungsservice vor oder nach der Belastung der Karte abstürzte oder hängen blieb, oder ob die Karte belastet wurde und nur die Antwort verloren ging. Aus Sicht des Aufrufers sehen alle drei Fälle gleich aus.

Drei Ausfälle, die gleich aussehen



Für den Aufrufer sehen alle drei gleich aus: keine Antwort vor dem Timeout.

Nach einem Timeout kann der Aufrufer nicht wissen, ob die Arbeit erledigt wurde; nur idempotente Wiederholungen oder eine spätere Statusabfrage können das klären.

Die praktischen Folgen prägen jedes verteilte Design. Eine Wiederholung könnte den Kunden doppelt belasten; deshalb müssen Operationen, die wiederholt werden können, gefahrlos wiederholbar sein. Diese Eigenschaft heißt Idempotenz und wird meist erreicht, indem jede Anfrage einen eindeutigen Schlüssel erhält, den sich der Empfänger merkt. Eine langsame Abhängigkeit ist oft schlimmer als eine tote, weil Aufrufer weiter warten und Threads und Verbindungen blockieren; wird ein Service langsam, gehen seinen Aufrufern die Ressourcen

aus, und sie werden ihrerseits langsam: ein kaskadierender Ausfall. Und weil ein Netzwerk so zerfallen kann, dass zwei Gruppen von Rechnern einander nicht erreichen, während beide weiterlaufen, können Daten, die an beiden Orten gehalten werden, auseinanderlaufen: das Thema des CAP-Abschnitts später in diesem Kapitel.

Eine globale Uhr gibt es ebenfalls nicht. Die Uhr jedes Rechners driftet und wird separat korrigiert, sodass sich Zeitstempel zweier Services nicht mit völliger Sicherheit vergleichen lassen und „Was ist zuerst passiert?“ eine schwierige Frage sein kann. Lamports Artikel von 1978 über Zeit und Uhren in verteilten Systemen zeigte, wie man Ereignisse ordnet, ohne sich auf physische Uhren zu verlassen; für Architekturfachleute lautet die Lehre einfach, Korrektheit nicht auf den Vergleich von Uhrzeiten verschiedener Rechner zu bauen.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: wenn eine Antwort nie kommt

Die Idee, die hier die meiste Sorgfalt verlangt, ist der Teilausfall. Innerhalb eines Programms wird ein Aufruf eines anderen Programmteils entweder beendet, oder das ganze Programm hält an. Über ein Netzwerk gibt es eine dritte Möglichkeit: Sie fragen, und es kommt keine Antwort zurück.

- Der Bestellservice bittet den Zahlungsservice, £24,50 von der Karte eines Kunden abzubuchen, und wartet auf die Antwort.
- Er kann nicht ewig warten, also gibt er nach einer festgelegten Frist auf, dem sogenannten Timeout: zum Beispiel nach zwei Sekunden. Die zwei Sekunden vergehen ohne Antwort.
- Drei verschiedene Dinge könnten passiert sein, und aus Sicht des Bestellservice sehen sie genau gleich aus: Die Anfrage ging unterwegs verloren, und es wurde nichts abgebucht; der Zahlungsservice ist abgestürzt oder hängen geblieben, und niemand weiß, ob er die Karte belastet hat; oder die Karte wurde belastet, und nur die Antwort ging verloren.
- Versucht es der Bestellservice einfach noch einmal, könnte er den Kunden doppelt belasten. Gibt er auf, hat der Kunde vielleicht für eine Bestellung bezahlt, die nie aufgegeben wurde.
- Die sicheren Auswege sind, die Anfrage idempotent zu machen, sodass zweimaliges Senden dieselbe Wirkung hat wie einmaliges, oder den Zahlungsservice hinterher zu fragen, was tatsächlich passiert ist. Idempotenz funktioniert meist so, dass mit der Abbuchung eine eindeutige Anfragenummer gesendet wird: Der Zahlungsservice merkt sich die bereits bearbeiteten Nummern und bucht dieselbe nie zweimal ab.
- **Die Lehre:** Im Netzwerk ist Schweigen keine Antwort. Jeder entfernte Aufruf braucht eine Frist und einen Plan für den Fall, dass man es nicht weiß; deshalb kommt dieses Kapitel immer wieder auf Wiederholungen und Duplikate zurück.

Die acht Fallacies of Distributed Computing

Die Fallacies of Distributed Computing sind acht Annahmen, die Neulinge bei verteilten Systemen treffen; alle sind falsch, und alle verursachen früher oder später Ärger. Die Liste stammt von Sun Microsystems: L Peter Deutsch stellte 1994 sieben davon zusammen, aufbauend auf vier, die Kollegen wie Bill Joy bereits notiert hatten, und James Gosling, der Schöpfer von Java, fügte um 1997 die achte hinzu. Drei Jahrzehnte später beschreibt jeder Punkt noch immer irgendwo einen echten Ausfall in dieser Woche.

Die acht Trugschlüsse, jeweils mit einer Folge und einer Antwort im Design

Trugschluss	Was schiefgeht, wenn Sie daran glauben	Was Sie stattdessen tun sollten
1. Das Netzwerk ist zuverlässig	Ein Zahlungsaufruf hängt; der aufrufende Thread wartet ewig; unter Last warten bald alle Threads, und der Bestellservice antwortet niemandem mehr.	Für jeden entfernten Aufruf ein Timeout setzen, nur idempotente Operationen wiederholen, Queues dort einsetzen, wo Arbeit warten kann (Resilienzmuster).
2. Die Latenz ist null	Eine Warenkorbseite, die 40 prozessinterne Aufrufe machte, macht jetzt 40 Netzwerkaufrufe zu je 2 ms; 80 ms entstehen aus dem Nichts, und im Tail weit mehr.	Grobgranulare Schnittstellen entwerfen, Anfragen bündeln, unabhängige Services parallel aufrufen, stabile Daten cachen.
3. Die Bandbreite ist unendlich	Jedes Event `MenuUpdated` enthält alle Fotos der Gerichte; ein geschäftiger Abend sättigt die Leitung und verzögert die Standort-Updates, die sie mitnutzen.	Kennungen und die Felder senden, die Consumer brauchen, paginieren, komprimieren, Massendaten neben dem Code halten, der sie verarbeitet.
4. Das Netzwerk ist sicher	Services im Rechenzentrum akzeptieren jede Anfrage ohne Authentifizierung; ein kompromittierter Container kann die Erstattungs-API aufrufen.	Datenverkehr zwischen Services verschlüsseln und authentifizieren (zum Beispiel mit Mutual TLS) und jeden Aufruf autorisieren: Zero Trust.
5. Die Topologie ändert sich nicht	Eine in der Konfiguration fest eingetragene IP-Adresse funktioniert nicht mehr, wenn der Container auf einen anderen Rechner verlegt wird.	Service Discovery und DNS-Namen verwenden und damit rechnen, dass Instanzen jede Minute kommen und gehen.
6. Es gibt einen Administrator	Ein Cloud-Anbieter, das Firewall-Team eines Partners oder eine andere Abteilung ändert eine Regel, tauscht ein Zertifikat aus oder senkt ein Rate Limit, und Ihre Aufrufe schlagen fehl.	Wissen, wer jeden Hop kontrolliert, die Abhängigkeiten überwachen, die Ihnen nicht gehören, Verträge und Ankündigungsfristen für Änderungen vereinbaren.
7. Transportkosten sind null	Serialisierung verbraucht CPU, und Daten, die Availability Zones, Regionen oder das öffentliche Internet überqueren, werden pro Gigabyte abgerechnet; die Netzwerkrechnung wächst schneller als der Datenverkehr.	Die Kosten von Datenflüssen messen, gesprächigen Datenverkehr lokal halten, bei hohem Volumen kompakte Formate wählen.
8. Das Netzwerk ist homogen	Ein Service geht davon aus, dass jeder Client dieselbe Protokollversion, Zeichenkodierung und Uhr verwendet; ein Mobilfunknetz, ein altes Partnersystem oder ein Proxy widerlegt die Annahme.	Standardisierte, explizit versionierte Formate und Protokolle verwenden und die Interoperabilität mit den echten Clients testen.

WOHER DAS KOMMT

Die Liste wird immer länger

In einem Interview von 2021 befürwortete Deutsch einen vorgeschlagenen neunten Trugschluss, den er als Erweiterung des vierten beschrieb: dass die Gegenseite, mit der Sie kommunizieren, vertrauenswürdig ist. Mark Richards und Neal Ford schlugen in der zweiten Auflage von *Fundamentals of Software Architecture* (2025) drei weitere aus Microservice-Projekten vor: Versionierung ist einfach, kompensierende Updates funktionieren immer, und Observability ist optional. Alle drei kehren später in diesem Kapitel wieder: Versionierung in Kommunikation und APIs, Kompensation bei Sagas und Observability weiter unten.

Was Verteilung kostet

Jeder Grund für eine Verteilung wird mit derselben Währung bezahlt. **Latenz** ist der erste Kostenpunkt. Ein Funktionsaufruf innerhalb eines Prozesses dauert Nanosekunden; ein Netzwerk-Roundtrip zwischen Rechnern im selben Rechenzentrum dauert typischerweise einige hundert Mikrosekunden bis zu einer Millisekunde oder mehr, noch vor jeder Serialisierung oder Arbeit. Zwischen Kontinenten setzt die Physik die Untergrenze: Licht bewegt sich in Glasfaser mit rund 200.000 km pro Sekunde, sodass ein Roundtrip von London nach New York (etwa 5600 km pro Richtung) nicht weniger als etwa 56 ms dauern kann. Die Rechnung zur Tail-Latenz aus Kapitel 2 gilt für jeden Hop.

Observability ist der zweite. In einem Monolithen zeigt ein Stacktrace die ganze Geschichte einer fehlgeschlagenen Anfrage. In einem verteilten System ist die Geschichte über die Logs vieler Services auf vielen Rechnern verstreut. Observability (Beobachtbarkeit), die Fähigkeit, aus den Ausgaben eines Systems zu verstehen, was es tut, muss von vornherein eingeplant werden: Korrelations-IDs, die bei jedem Aufruf weitergegeben werden, zentralisierte Logs, Metriken pro Service und Distributed Tracing (Kapitel 9).

Konsistenz ist der dritte. Eine einzelne Datenbanktransaktion kann eine Bestellung und ihre Zahlung atomar aktualisieren. Liegen Bestellung und Zahlung in verschiedenen Services mit verschiedenen Datenbanken, gibt es keine gemeinsame Transaktion; das System muss eine Zeitspanne hinnehmen, in der beide nicht übereinstimmen, und dafür sorgen, dass sie konvergieren. Spätere Abschnitte behandeln die Werkzeuge: Konsistenzmodelle sowie Sagas und das Outbox-Muster.

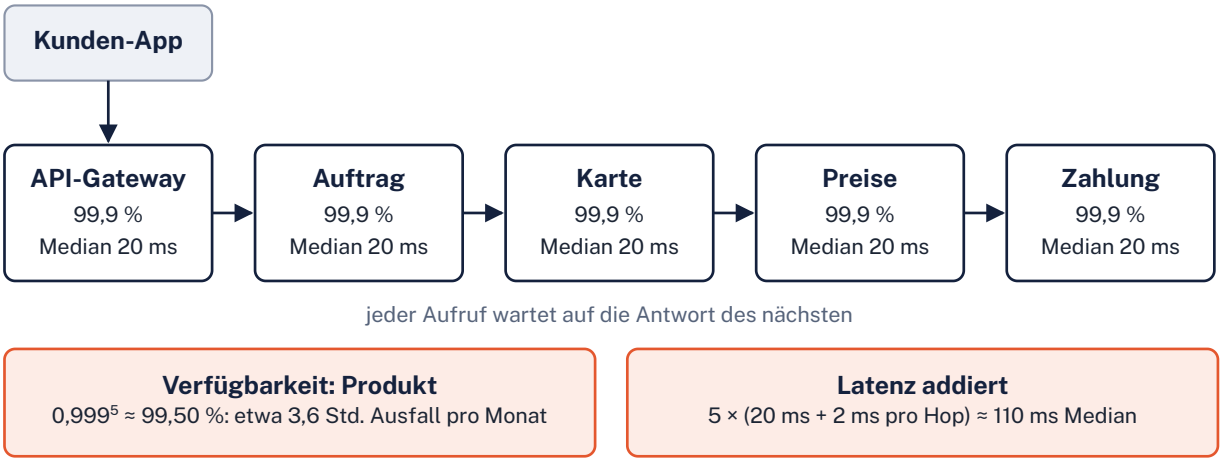
Betriebsaufwand ist der vierte und der am häufigsten unterschätzte. Jeder Service braucht eine Build-Pipeline, ein Deployment, Konfiguration, Secrets, Monitoring, Alarme, Kapazitätsplanung, Sicherheitspatches und jemanden in Bereitschaft. Auch das Testen ändert

sich: Eine Änderung ist nur dann sicher, wenn jeder Consumer ihrer Schnittstelle weiterhin funktioniert, und das erfordert Contract Tests und abwärtskompatible Änderungen. Diese Kosten sind ungefähr proportional zur Zahl der deploybaren Einheiten; deshalb kommt es auf die Zahl der Services an.

Nachgerechnet: synchrone Ketten

Der günstigste Moment, um festzustellen, dass ein verteiltes Design zu fragil oder zu langsam ist, ist auf dem Papier. Das Werkzeug ist die Reihenregel aus Kapitel 2: Braucht eine Anfrage jede Komponente einer Kette, ist die Verfügbarkeit der Kette das Produkt der Verfügbarkeiten der Komponenten, und ihre Latenz ist ungefähr die Summe ihrer Latenzen.

Eine synchrone Kette aus fünf Services



Fünf Services, die jeweils 99,9 % erreichen, ergeben einen Anfragepfad, der nur etwa 99,5 % erreicht, und die fünffache Latenz eines einzelnen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: das Verfügbarkeits- und Latenzbudget einer Aufrufkette

Eine Essenslieferplattform wickelt **Bestellen** über eine synchrone Kette ab: Das API-Gateway ruft den Bestellservice auf, der den Speisekartenservice aufruft, der die Preisgestaltung aufruft, die die Zahlung aufruft. Jeder Service erreicht für sich 99,9 % Verfügbarkeit (zeitbasiert, 30-Tage-Monat mit 43.200 Minuten), antwortet im Median nach 20 ms eigener Arbeit und hat ein p99 von 250 ms. Jeder Netzwerk-Hop fügt etwa 2 ms hinzu. Nehmen Sie großzügig an, dass Ausfälle unabhängig sind und dass jeder Ausfall die ganze Anfrage scheitern lässt.

Ein Service	Verfügbarkeit 0,999 = 99,9 %; zulässige Ausfallzeit 0,001 × 43.200 = 43,2 Minuten im Monat.
Zwei in Reihe	0,999 × 0,999 = 0,998 (99,8 %); etwa 86 Minuten im Monat.
Fünf in Reihe	$0{,}999^5 \approx 0{,}99501$ (99,50 %); nicht verfügbar etwa 0,005 × 43.200 ≈ 216 Minuten oder 3,6 Stunden im Monat. Fünf für sich genommen respektable Services haben ein System mit zweieinhalb Neunen ergeben.
Zehn in Reihe	$0{,}999^{10} \approx 0{,}99004$ (99,0 %); etwa 430 Minuten, über 7 Stunden, im Monat.
Median-Latenz, fünf in Reihe	Latenzen addieren sich: ungefähr 5 × (20 ms + 2 ms) ≈ 110 ms, gegenüber 20 ms für dieselbe Arbeit in einem Prozess. (Streng genommen addieren sich Mediane nicht exakt; die Summe ist eine gute erste Schätzung.)
Tail-Latenz, fünf in Reihe	Die Wahrscheinlichkeit, dass eine Anfrage bei mindestens einem Hop in dessen langsamstes 1 % fällt, beträgt $1 - 0{,}99^5 \approx 4{,}9$ %; also wird etwa 1 von 20 Anfragen durch mindestens einen Hop mit 250 ms verlangsamt. Bei zehn Hops sind es $1 - 0{,}99^{10} \approx 9{,}6$ %.
Was nötig ist, um 99,9 % von Ende zu Ende zu halten	Jeder der fünf Services bräuchte etwa 99,98 % ($0{,}9998^5 \approx 0{,}9990$): rund 8,6 Minuten Ausfallzeit pro Monat und Service, was in der Praxis automatische Wiederherstellung bedeutet.
Antworten im Design	Speisekarte und Preisgestaltung parallel aufrufen (die Latenz ist dann die des langsameren der beiden, nicht die Summe); Speisekartendaten im Bestellservice cachen; die Zahlungsbestätigung asynchron machen, sodass die Bestellung angenommen und kurz danach bezahlt wird; Hops entfernen, die Anfragen nur weiterreichen.

Kapitel 2 hat die Regel eingeführt, dass sich Verfügbarkeiten in Reihe multiplizieren; bei der Verteilung beginnt diese Regel zu schmerzen, weil jeder Netzwerkaufruf der Reihe eine Komponente hinzufügt.

Die Unabhängigkeitsannahme beschönigt das Ergebnis. Services, die sich eine Datenbank, einen Cluster, ein Konfigurationssystem oder einen Release Train teilen, fallen gemeinsam aus, und ein langsamer Aufgerufener kann die Threads des Aufrufers erschöpfen und den Ausfall nach oben ausbreiten.

Die Lösung besteht selten darin, jeden Service heldenhaft zuverlässig zu machen. Sie besteht darin, synchrone Ketten zu verkürzen und Teile, die nicht sofort antworten müssen, aus einer Queue arbeiten zu lassen.

Das Architekturquantum, noch einmal betrachtet

Kapitel 5 hat das Architekturquantum eingeführt: ein unabhängig deploybarer Teil mit hoher funktionaler Kohäsion, einschließlich seiner Daten, für den ein einziger Satz von Architekturmerkmalen gilt (Architekturstile, Muster und Taktiken). In *Software Architecture: The Hard Parts* (2021) schärfen Ford, Richards, Sadalage und Dehghani die Idee mit zwei Arten von Kopplung. Statische Kopplung ist das, was vorhanden sein muss, damit ein Teil deployt werden und laufen kann: sein Code, Bibliotheken, Datenbank und Infrastruktur. Dynamische Kopplung beschreibt, wie Teile zur Laufzeit voneinander abhängen: ob sie sich synchron oder asynchron aufrufen und wie sie sich koordinieren und konsistent bleiben.

Das ausgearbeitete Beispiel zeigt, warum die dynamische Seite zählt. Für die Dauer eines synchronen Aufrufs kann der Aufrufer seine Arbeit nur abschließen, wenn der Aufgerufene läuft und schnell ist; die beiden sind betrieblich aneinandergebunden, und ihre Verfügbarkeit und Latenz verbinden sich wie oben berechnet. Fünf separat deployte Services in einer synchronen Kette sind fünf Deployment-Einheiten, doch für die Verfügbarkeit von **Bestellen** verhalten sie sich wie eine fragile Einheit. Diese Abhängigkeit davon, dass die Gegenseite im selben Moment verfügbar ist, heißt zeitliche Kopplung. Asynchrones Messaging beseitigt einen Großteil davon: Legt der Bestellservice eine Nachricht in eine dauerhafte Queue, kann der Zahlungsservice eine Minute lang ausfallen und danach aufholen, und der Kunde erhält trotzdem eine Bestellnummer. Der Preis ist, dass der Kunde nicht sofort eine endgültige Antwort erhält, was ebenso eine Produktentscheidung wie eine technische ist.

Das ist die tiefste Lehre dieses Abschnitts. Service-Grenzen zu ziehen ist die Hälfte des Designs; zu entscheiden, welche Interaktionen synchron sein müssen, und diese Menge klein zu halten, ist die andere Hälfte.

ACHTUNG

Häufige Fehler bei der Entscheidung für eine Verteilung

- **Verteilen, um ein Problem der Codequalität zu lösen.** Ein verworrener Monolith, der in Services aufgeteilt wird, wird zu einem verworrenen verteilten System, zuzüglich Netzwerkausfällen. Bereinigen Sie zuerst die Grenzen; der modulare Monolith aus Kapitel 5 ist der übliche Weg.
- **Entfernte Aufrufe wie lokale behandeln.** Bibliotheken, die einen entfernten Aufruf wie einen Methodenaufruf aussehen lassen, verbergen Latenz und Ausfälle; Martin Fowlers „erstes Gesetz des verteilten Objektdesigns“ lautete schlicht, Ihre Objekte nicht zu verteilen.
- **Services statt synchroner Hops zählen.** Zehn Services, die überwiegend Events austauschen, können robuster sein als drei, die sich in einer Kette gegenseitig aufrufen.
- **Unabhängige Ausfälle annehmen.** Services, die sich eine Datenbank, einen Kubernetes-Cluster oder einen Konfigurationsspeicher teilen, fallen gemeinsam aus; die Reihenformel ist dann optimistisch und die Parallelformel Fiktion.
- **Die Kosten für Menschen ignorieren.** Zehn Services brauchen zehn Sätze von Pipelines, Dashboards und Bereitschaftsplänen. Verantwortet ein einziges Team von sechs Personen sie alle, fehlt der Hauptnutzen der Verteilung, die Teamautonomie, während alle Kosten bleiben.

Checkliste: bevor Sie verteilen

Stellen Sie sich diese Fragen, bevor Sie ein System oder einen Teil davon über das Netzwerk aufteilen.

- [ ] **Den Treiber benennen**
  Welcher Grund aus der Tabelle oben trifft zu, auf welchen Teil, mit welchen Belegen? „Microservices sind modern“ ist kein Treiber.
- [ ] **Zuerst die günstigere Lösung versuchen**
  Könnten Replikate, Caching, eine Read Replica, ein Background Worker oder eine klarere Modulgrenze innerhalb des Monolithen den Bedarf decken?
- [ ] **Den Aufrufgraphen zeichnen**
  Zählen Sie die synchronen Hops in jeder wichtigen User Journey und berechnen Sie das Verfügbarkeits-und Latenzbudget wie im ausgearbeiteten Beispiel.
- [ ] **Festlegen, wem welche Daten gehören**
  Jedes Datenstück braucht einen verantwortlichen Service; alles, was sich atomar zusammen mit ihm ändern muss, gehört wahrscheinlich in denselben Service.
- [ ] **Für Teilausfälle entwerfen**
  Timeouts, Wiederholungen mit Idempotenz, Fallbacks und was der Nutzer sieht, wenn eine Abhängigkeit ausgefallen ist, entschieden vor dem ersten Ausfall.
- [ ] **Die Voraussetzungen prüfen**
  Automatisierte Bereitstellung und automatisiertes Deployment, Monitoring, zentralisierte Logs und Distributed Tracing müssen vorhanden oder geplant und finanziert sein.
- [ ] **Teams an Grenzen ausrichten**
  Jeder Service sollte ein verantwortliches Team haben, das ihn ändern, deployen und betreuen kann, ohne andere zu fragen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Ein Service ruft einen Lagerbestandsservice ohne Timeout auf. Der Lagerbestandsservice wird sehr langsam, statt auszufallen. Warum kann das schlimmer sein, als wenn er abgestürzt wäre?**

Ein abgestürzter Service schlägt schnell fehl, und Aufrufer können reagieren; ein langsamer lässt Aufrufer warten, sodass sich ihre Threads und Verbindungen füllen, bis auch sie nicht mehr antworten und sich der Ausfall als kaskadierender Ausfall ausbreitet.

▼ **Drei Services haben jeweils 99,95 % Verfügbarkeit und werden bei jeder Anfrage in Reihe aufgerufen. Wie hoch ist ungefähr die Verfügbarkeit des Anfragepfads?**

$0{,}9995^3 \approx 0{,}9985$, etwa 99,85 %, unter der Annahme unabhängiger Ausfälle.

▼ **Warum ist der Trugschluss „Transportkosten sind null“ auch dann wichtig, wenn Latenz und Bandbreite in Ordnung sind?**

Das Bewegen von Daten kostet CPU für die Serialisierung und Geld für den Datenverkehr zwischen Zonen, Regionen oder Anbietern; ein gesprächiges oder datenlastiges Design kann daher teuer sein, selbst wenn es schnell genug ist.

▼ **Warum können sich fünf separat deployte Services in Bezug auf die Verfügbarkeit trotzdem wie eine Einheit verhalten?**

Ruft eine Anfrage sie synchron in einer Kette auf, müssen alle fünf im selben Moment laufen; ihre Verfügbarkeiten verbinden sich, sie sind also trotz getrennter Deployments zeitlich gekoppelt.

▼ **Probieren Sie es aus: „Bestellen“ ruft synchron vier Services auf, die jeweils zu 99,9 % verfügbar sind. Berechnen Sie die Verfügbarkeit und die monatliche Ausfallzeit (43.200 Minuten). Was ändert sich, wenn Benachrichtigung und Disposition asynchron werden und zwei synchron bleiben? Was müssen Sie hinzufügen, wenn auf einen Timeout eine Wiederholung folgt?**

Vier: 0,999^4 = 0,99601, etwa 99,6 %, also 173 Minuten im Monat. Zwei: 0,999^2 = 0,99800, etwa 86 Minuten, ungefähr die Hälfte, bei unabhängigen Ausfällen. Nach einem Timeout kann der Aufrufer nicht wissen, ob die Arbeit erledigt wurde, wiederholte Aufrufe müssen also idempotent sein, etwa mit einem Idempotenzschlüssel, sonst könnte ein Kunde doppelt belastet werden. Die asynchronen Schritte brauchen überwachte Queues, halten die Bestellung aber nicht mehr auf.

QUELLEN

- Leslie Lamport: the 1987 email defining a distributed system (lamport.org): https://lamport.org/pubs/distributed-system.txt
- The Eight Fallacies of Distributed Computing (James Gosling's page): https://nighthacks.com/jag/res/Fallacies.html
- Software Engineering Radio episode 470: L Peter Deutsch on the fallacies of distributed computing (2021): https://se-radio.net/2021/07/episode-470-l-peter-deutsch-on-the-fallacies-of-distributed-computing/
- Martin Fowler: Microservices and the First Law of Distributed Objects: https://martinfowler.com/articles/distributed-objects-microservices.html
- Google SRE book: Addressing Cascading Failures: https://sre.google/sre-book/addressing-cascading-failures/

Servicebasierte Architektur und serviceorientierte Architektur

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern
- 5.3 Modularer Monolith
- 6.1 Warum verteilen

Begriffe von früher, die er verwendet: ACID, Schema, Architekturquantum, Fachliche Partitionierung, Bounded Context

DIE IDEE IN KÜRZE

Denken Sie an ein Krankenhaus, das aus einem Gebäude herauswächst. Es könnte alles in Hunderte winziger Praxen zerschlagen, jede mit eigenem Empfang und eigenem Archiv, oder es könnte in fünf Abteilungen in getrennten Flügeln umziehen: Geburtshilfe, Chirurgie, Ambulanz, Apotheke, Notaufnahme. Jeder Flügel hat sein eigenes Personal und kann renoviert werden, ohne die anderen zu schließen, aber alle teilen sich weiterhin ein zentrales Patientenarchiv. Das ist servicebasierte Architektur: wenige große, getrennte Teile, die sich jeweils um einen ganzen Geschäftsbereich kümmern und sich oft noch eine Datenbank teilen.

WAS SIE WISSEN MÜSSEN

- Servicebasierte Architektur (so benannt von Richards und Ford) teilt ein System in wenige separat deployte, grobgranulare Domain Services auf, typischerweise 4 bis 12, die sich meist eine Datenbank teilen und oft hinter einer API-Schicht liegen.
- Weil die meisten Abläufe innerhalb eines Domain Service bleiben, können sie lokale ACID-Transaktionen nutzen, und Services rufen sich selten gegenseitig auf, was synchrone Ketten kurz hält.
- Eine gemeinsame Datenbank koppelt Schemaänderungen und bindet die Services zu einem Quantum zusammen; Schemas pro Service, kleine Entity-Bibliotheken pro Domäne und, wo gerechtfertigt, getrennte Datenbanken verringern diese Kopplung.
- Die klassische, orchestrierungsgetriebene SOA ordnete Services in technischen Schichten um einen Enterprise Service Bus an, mit einem kanonischen Datenmodell und WS-*-Standards, und zielte auf unternehmensweite Wiederverwendung.

- SOA enttäuschte, weil Wiederverwendung Kopplung erzeugte, der Bus zum Engpass in der Hand eines einzigen Teams wurde, kanonische Modelle niemandem passten und die Standards schwerfällig waren.
- Microservices behalten explizite Serviceverträge bei, legen die Logik aber in die Services (Smart Endpoints, Dumb Pipes), organisieren sich nach Geschäftsfähigkeiten und geben jedem Service eigene Daten.
- Servicebasierte Architektur ist ein pragmatischer Mittelweg für wenige Domänen und Teams und ein risikoarmer Schritt aus einem Monolithen heraus.

Zwischen Monolith und Microservices

Ein Großteil der Diskussion in der Branche stellt eine Wahl zwischen einem Monolithen und vielen Microservices dar. In der Praxis liegt ein großer Teil realer Systeme dazwischen: eine Handvoll separat deployter Services, die jeweils einen ganzen Geschäftsbereich abdecken und sich oft noch eine Datenbank teilen. Mark Richards und Neal Ford haben diesen Stil in *Fundamentals of Software Architecture* servicebasierte Architektur genannt und halten ihn für einen der pragmatischsten verteilten Stile überhaupt. Dieser Abschnitt beschreibt ihn, grenzt ihn von der älteren serviceorientierten Architektur (SOA) ab, mit der er oft verwechselt wird, und zeigt, worin sich Microservices von beiden unterscheiden.

Servicebasierte Architektur: die Topologie

Ein servicebasiertes System hat drei Teile. Eine oder mehrere Benutzeroberflächen rufen eine Reihe separat deployter Domain Services auf, die jeweils einen großen, zusammenhängenden Geschäftsbereich wie Bestellungen oder Zahlungen abdecken. Richards und Ford beobachten, dass solche Systeme meist zwischen 4 und 12 Domain Services haben, im Durchschnitt etwa 7. Die Services teilen sich meist eine Datenbank, und eine optionale API-Schicht (ein Reverse Proxy oder Gateway) kann zwischen den Benutzeroberflächen und den Services liegen und Routing, Authentifizierung und Rate Limiting übernehmen.

Jeder Domain Service ist grobgranular: Er enthält alles, was seine Domäne braucht, von der API über die Geschäftslogik bis zum Datenzugriff, und ist intern wie ein kleiner Monolith aufgebaut, oft in Schichten oder Modulen (Kapitel 5). Ein Service wird typischerweise als einige Instanzen hinter einem Load Balancer deployt.

Daraus folgen zwei Konsequenzen. Erstens laufen die meisten Geschäftsabläufe vollständig innerhalb eines Service und können daher eine gewöhnliche lokale Datenbanktransaktion mit vollen ACID-Garantien nutzen (Kapitel 0). Zweitens müssen sich Domain Services selten

gegenseitig aufrufen: Eine Anfrage von der Benutzeroberfläche geht an den einen Service, dem die Domäne gehört. Das hält synchrone Ketten kurz, was aus den in Abschnitt 6.1 berechneten Gründen wichtig ist.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: wie eine gemeinsame Datenbank Services aneinanderbindet

Der Punkt, an dem hier die meisten stolpern, ist, dass separat deployte Services trotzdem durch die Datenbank aneinandergebunden sein können, die sie sich teilen. Ein kleines Beispiel zeigt, wie.

- Angenommen, die Services der Essenslieferplattform lesen ungehindert die Tabellen der anderen. Bestellungen, Restaurant und Lieferung lesen alle die Tabelle restaurants in der einen gemeinsamen Datenbank.
- Das Restaurant-Team möchte die Adressspalte der Tabelle in getrennte Spalten für Straße, Ort und Postleitzahl aufteilen, um die Suche nach Postleitzahl zu erleichtern. Im eigenen Code ist das eine Änderung von einem Tag.
- Aber auch Bestellungen und Lieferung lesen die Adressspalte. In dem Moment, in dem sie verschwindet, gehen sie kaputt, obwohl niemand ihren Code angefasst hat.
- Also müssen alle drei Services gemeinsam geändert, getestet und ausgeliefert werden. Durch gemeinsame Tabellen verbundene Services können nicht unabhängig deployt werden; deshalb sagt der Abschnitt, dass sie ein einziges Architekturquantum bilden.
- Die wichtigste Abhilfe in diesem Abschnitt besteht darin, jedem Service einen eigenen Satz von Tabellen zu geben, ein eigenes Schema, das kein anderer Service anfassen darf. Bestellungen und Lieferung lesen Adressen dann über eine schreibgeschützte View, die Restaurant veröffentlicht, und Restaurant kann seine eigenen Tabellen frei umbauen, solange diese View dieselbe Form behält.
- **Die Lehre:** Eine gemeinsame Datenbank macht diesen Stil günstig und einfach, aber jede Tabelle, die mehrere Services lesen, ist ein Versprechen, das mehrere Teams gemeinsam halten müssen. Geben Sie jeder Tabelle einen verantwortlichen Service, und teilen Sie nur wenige veröffentlichte Views.

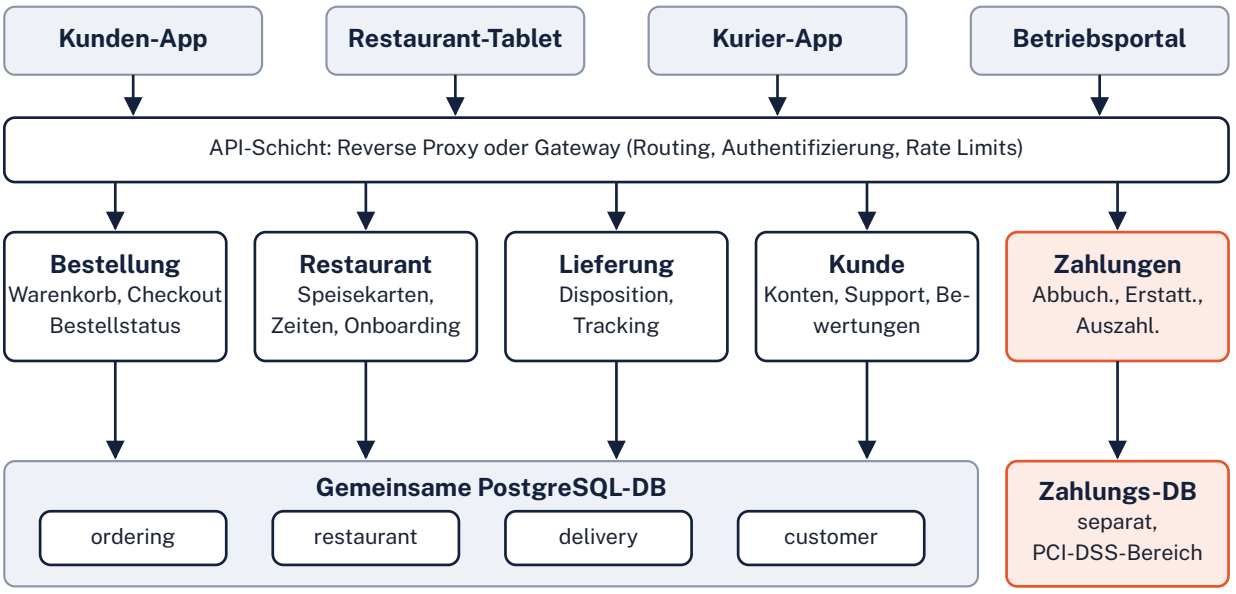
Die gemeinsame Datenbank und wie man sie zähmt

Eine gemeinsame Datenbank macht servicebasierte Architektur günstig und einfach: keine Datensynchronisation zwischen Services, keine Eventual Consistency, Reporting mit gewöhnlichem SQL. Sie ist zugleich ihre Hauptschwäche. Lesen und schreiben alle Services dieselben Tabellen, kann eine Änderung an den Spalten einer Tabelle mehrere Services kaputt

machen, die dann alle gemeinsam getestet und neu deployt werden müssen. Die Definition aus Kapitel 5 gilt: Services, die sich eine Datenbank teilen, sind zu einem Architekturquantum verbunden, sie teilen also ein Verfügbarkeitsniveau und einen Rhythmus für Schema-Releases.

In der Praxis gibt es drei Techniken, um die Kopplung zu verringern. Die erste ist logische Partitionierung: Jeder Service erhält sein eigenes Schema (oder seinen eigenen Satz von Tabellen) und darf die der anderen nicht anfassen, durchgesetzt sowohl durch Datenbankberechtigungen als auch durch Konvention. Die zweite besteht darin, die Entity-Klassen der gemeinsamen Datenbank in kleinen Bibliotheken pro Domäne statt in einer großen gemeinsamen Bibliothek zu halten, sodass eine Änderung an Restauranttabellen nur die Services betrifft, die sie nutzen. Die dritte besteht darin, einem Service eine eigene Datenbank zu geben, wenn er einen triftigen Grund hat, etwa den Sicherheitsumfang, ein ganz anderes Lastmuster oder eine andere Art von Datenspeicher. Teams, die diese Techniken konsequent anwenden, stellen fest, dass ein späterer Wechsel zu getrennten Datenbanken, falls er je nötig wird, viel leichter fällt.

Die Essenslieferplattform als fünf Domain Services



Jeder Service besitzt sein eigenes Schema; nur Zahlungen hat eine eigene DB

Fünf grobgranulare Services, die jeweils einem Team gehören, hinter einer API-Schicht; vier teilen sich eine Datenbank mit je einem Schema, und Zahlungen behält eine eigene.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Essenslieferplattform als fünf Domain Services

Ein Essenslieferunternehmen ist von einem Team auf fünf gewachsen, etwa 35 Entwicklerinnen und Entwickler. Sein Django-Monolith ist gut modularisiert, aber Releases sind zur Warteschlange geworden, und ein Fehler im Kurier-Tracking hat im letzten Quartal zweimal den Checkout lahmgelegt. Die Architektin schlägt eine servicebasierte Architektur mit fünf Domain Services vor, einem pro bestehendem Modul und Team.

Bestellung	Warenkorb, Checkout, Lebenszyklus und Status von Bestellungen. Besitzt das Schema `ordering`. Zu Essenszeiten stark ausgelastet; läuft mit 6 bis 20 Instanzen.
Restaurant	Speisekarten, Preise, Öffnungszeiten, Onboarding von Restaurants. Leselastig; Speisekarten werden in der API-Schicht gecacht. Läuft mit 3 Instanzen.
Zustellung	Kurierdisposition, Routenschätzungen, Live-Tracking. Erhält alle paar Sekunden Standort-Updates von jedem aktiven Kurier und wird daher getrennt von allem anderen skaliert.
Zahlungen	Kartenautorisierung über den Zahlungsanbieter, Erstattungen, Auszahlungen an Restaurants und Kuriere. Hat eine eigene Datenbank, damit der Umfang der PCI-DSS-Prüfung ein kleiner Service ist statt der ganzen Plattform.
Kunde	Konten, Adressen, Support-Tickets, Bewertungen. Geringe Last, hält aber die meisten personenbezogenen Daten, daher sind Zugriffs- und Aufbewahrungsregeln hier konzentriert.
Wie ein Ablauf läuft	**Bestellen** läuft innerhalb von Bestellung in einer lokalen Transaktion (Bestellung anlegen, den Zeitslot des Restaurants reservieren) und macht dann einen Aufruf an Zahlungen. Restaurant und Lieferung erfahren von der Bestellung über eine Nachricht, nicht über eine Kette von Aufrufen.
Was geteilt wird	Ein PostgreSQL-Server mit einem Schema pro Service. Jeder Service darf nur sein eigenes Schema lesen und schreiben; einige schreibgeschützte Views (zum Beispiel Restaurantnamen) werden für die anderen veröffentlicht.
Was es bringt	Fünf Deployment-Pipelines statt einer; ein Fehler im Tracking stoppt den Checkout nicht mehr; Zahlungen lassen sich allein auditieren; jedes Team liefert aus, wenn es so weit ist.
Was es kostet	Fünf deploybare Einheiten und Dashboards statt einer; Netzwerkaufrufe zwischen Bestellung und Zahlungen, die Timeouts und Idempotenz brauchen; Schemaänderungen müssen weiterhin über den gemeinsamen Datenbankserver koordiniert werden.

Der Schritt ist bewusst bescheiden. Er führt von einem Quantum zu ungefähr zweien (Zahlungen mit eigenen Daten und die vier Services, die sich die Datenbank teilen), und er gibt jedem Team sein eigenes Release, was der Hauptschmerz war.

Wächst der Standort-Stream von Lieferung später über die gemeinsame Datenbank hinaus, kann dieser eine Service einen eigenen Speicher erhalten und weiter aufgeteilt werden; der Rest kann bleiben, wie er ist.

Die Architektin hält die Grenzregeln als Architekturtests fest: Kein Service darf das Schema eines anderen abfragen, und Services dürfen sich nur über veröffentlichte APIs gegenseitig aufrufen.

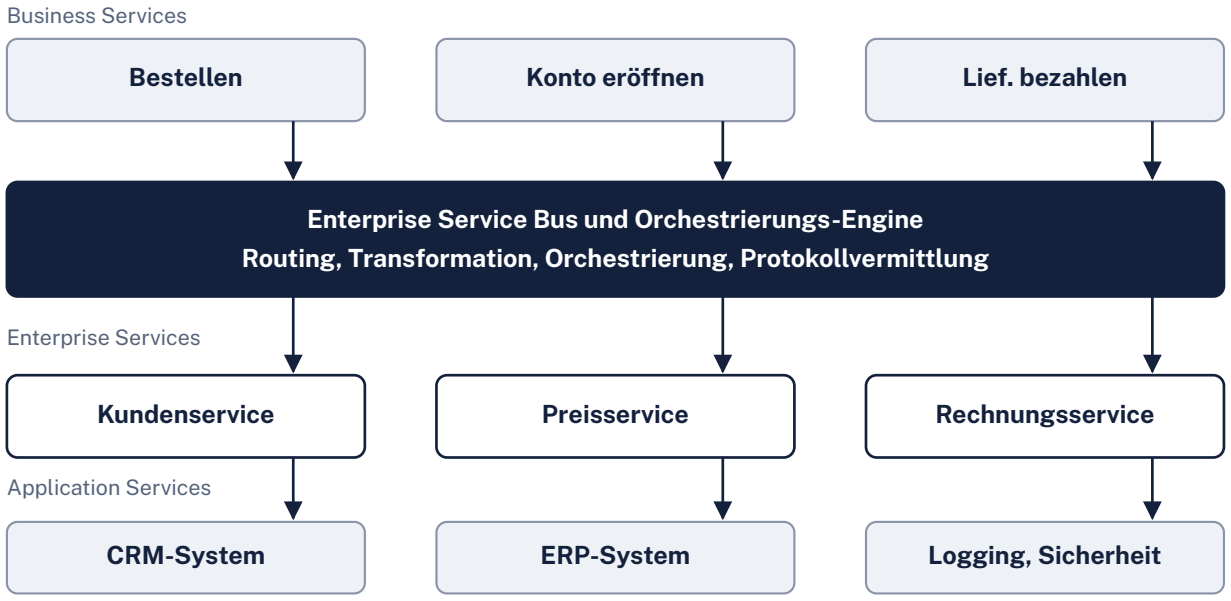
Klassische serviceorientierte Architektur

Serviceorientierte Architektur ist älter und breiter angelegt. Der Begriff kam 1996 durch einen Gartner-Bericht von Roy Schulte und Yefim Natis in die Branche, und ab den frühen 2000er-Jahren wurde sie zum vorherrschenden Modell der Unternehmensintegration. Ihre Kernidee war vernünftig: Geschäftsfähigkeiten als Services mit expliziten, technologieneutralen Verträgen bereitstellen, sodass Anwendungen in einer großen Organisation daraus zusammengesetzt werden können, statt neu gebaut oder Punkt zu Punkt integriert zu werden.

Was die meisten Unternehmen tatsächlich bauten, war jedoch das, was Richards und Ford orchestrierungsgetriebene SOA nennen. Services wurden in technischen Schichten angeordnet: Business Services, die Prozesse beschreiben (Bestellung aufgeben, Konto eröffnen), gemeinsame Enterprise Services, die wiederverwendbare Fähigkeiten bereitstellen (Kunde, Preisgestaltung, Rechnung), sowie Application und Infrastructure Services, die bestehende Systeme und technische Anliegen kapseln. Im Zentrum saß ein Enterprise Service Bus (ESB): Middleware, die Nachrichten weiterleitete, zwischen Formaten umwandelte, zwischen Protokollen vermittelte und über eine Orchestrierungs-Engine die Geschäftsprozesse selbst ausführte.

Zwei weitere Ideen vervollständigten das Bild. Ein kanonisches Datenmodell definierte eine unternehmensweite Darstellung jeder Geschäftsentität (ein Customer, ein Order), und jede Nachricht wurde im Bus in diese Darstellung und aus ihr übersetzt. Und eine Familie von Webservice-Standards, meist WS-* genannt, legte fest, wie Services einander beschreiben und aufrufen: SOAP für Nachrichten, WSDL für Schnittstellenbeschreibungen, UDDI für Registries, WS-Security, WS-ReliableMessaging und WS-AtomicTransaction für Qualitäten und WS-BPEL (Version 2.0, 2007 von OASIS verabschiedet) für ausführbare Prozessdefinitionen.

Orchestrierungsgetriebene SOA



Jede Nachricht wird im Bus in ein kanonisches Datenmodell und aus ihm übersetzt

In der klassischen SOA enthielt der Bus Routing-, Transformations- und Prozesslogik, und alle Services teilten ein kanonisches Modell des Geschäfts.

Warum die klassische SOA in Ungnade fiel

Ende der 2000er-Jahre hatten viele SOA-Programme enttäuscht. Anne Thomas Manes von der Burton Group fasste die Stimmung im Januar 2009 in einem viel gelesenen Beitrag mit dem Titel *SOA is Dead; Long Live Services* zusammen: Das Etikett war toxisch geworden, aber die Idee der Services überlebte. Die Gründe sind wissenswert, denn jeder ist eine Falle, in die moderne Architekturen erneut tappen können.

Wiederverwendung erzeugte Kopplung. Ein gemeinsamer Enterprise-Customer-Service, den vierzig Anwendungen nutzten, konnte sich nicht ändern, ohne dass vierzig Consumer zustimmten; die Wiederverwendung, das Hauptziel, machte jede Änderung zur Verhandlung. **Der Bus wurde zum Engpass.** Routing, Transformation und Prozesslogik in einen zentralen ESB zu legen, der meist einem einzigen Integrationsteam gehörte, bedeutete, dass fast jede geschäftliche Änderung dieses Team brauchte, und der Bus wurde zu einem Single Point of Failure und der Verzögerung. **Das kanonische Modell war zu groß.** Eine Definition von Customer, die Vertrieb, Abrechnung, Support und Logistik zufriedenstellt, ist entweder riesig oder für die meisten von ihnen falsch; die Bounded Contexts aus Kapitel 5 erklären, warum ein Begriff in verschiedenen Teilen eines Unternehmens berechtigterweise Verschiedenes bedeutet. **Die Standards waren schwerfällig.** WS-*-Spezifikationen waren komplex, die Werkzeuge teuer, und verteilte Transaktionen über Services hinweg erwiesen sich als fragil. **Technische Partitionierung** bedeutete, dass ein einfaches Feature mehrere Schichten von Services durchquerte, die verschiedenen Gruppen gehörten.

Die bleibenden Lehren sind die, auf denen Microservices aufgebaut wurden: explizite Serviceverträge, Services, die denen gehören, die sie bauen, Logik in den Services statt in der Middleware und Daten, die einem Service gehören, statt im ganzen Unternehmen geteilt zu werden.

Worin sich Microservices unterscheiden

Microservices, die im nächsten Abschnitt ausführlich behandelt werden, behalten die SOA-Idee von Services mit expliziten Verträgen bei und kehren die meisten ihrer Umsetzungsentscheidungen um. James Lewis und Martin Fowler fassten eine dieser Umkehrungen als *Smart Endpoints and Dumb Pipes* zusammen: Das Netzwerk transportiert nur Nachrichten, und die gesamte Logik liegt in den Services. Services sind um Geschäftsfähigkeiten statt um technische Schichten organisiert, Wiederverwendung ist der Unabhängigkeit bewusst nachgeordnet, jeder Service besitzt seine Daten, und Governance ist dezentralisiert.

Verglichen mit servicebasierter Architektur sind Microservices feingranularer, weit zahlreicher und bestehen auf getrennten Daten. Das bringt mehr Unabhängigkeit, gezieltere Skalierung und mehr Fehlerisolation und kostet viel mehr an Betrieb, Konsistenz und Koordination. Servicebasierte Architektur lässt sich als der Punkt im Spektrum sehen, an dem der Großteil des Nutzens für Deployment und Teams zu einem Bruchteil der Kosten eintritt.

Servicebasierte Architektur, klassische SOA und Microservices im Vergleich

Aspekt	Servicebasiert	Klassische (orchestrierungsgetriebene) SOA	Microservices
Hauptziel	Pragmatische Modularität und unabhängiges Deployment weniger Domänenbereiche	Unternehmensweite Wiederverwendung gemeinsamer Services	Unabhängige Änderung und unabhängiges Deployment durch viele autonome Teams
Partitionierung	Fachlich: ein Service pro großem Geschäftsbereich	Technisch: Business-, Enterprise-, Application- und Infrastructure-Schicht	Fachlich: ein Service pro Bounded Context oder kleiner
Typische Anzahl	Etwa 4 bis 12 pro System	Dutzende bis Hunderte im ganzen Unternehmen	Zehn bis Hunderte pro System
Granularität	Grob: eine ganze Domäne, zum Beispiel alles rund um Bestellungen	Gemischt; gemeinsame Enterprise Services sind oft feingranular	Fein bis mittel: eine kohäsive Fähigkeit
Daten	Oft eine gemeinsame Datenbank, idealerweise nach Schemas partitioniert	Gemeinsame Systems of Record hinter Enterprise Services	Jeder Service besitzt seine Daten; keine gemeinsamen Tabellen
Kommunikation	Meist von der Benutzeroberfläche zu einem Service; wenige Aufrufe zwischen Services	Über den Enterprise Service Bus, meist SOAP oder XML-Messaging	Leichtgewichtiges HTTP, gRPC oder Messaging; Smart Endpoints, Dumb Pipes
Wo die Logik liegt	In den Domain Services	Ein Großteil im Bus: Routing, Transformation, Orchestrierung	In den Services; das Netzwerk transportiert nur
Transaktionen	Lokale ACID-Transaktionen innerhalb eines Domain Service	Versuche mit verteilten Transaktionen (WS-AtomicTransaction, XA)	Keine verteilten Transaktionen; Sagas und Eventual Consistency
Governance	Leicht; ein Architekturteam genügt	Zentral und schwerfällig: kanonisches Modell, Service Registry, Standardgremium	Dezentralisiert; gemeinsame Plattform und Paved Roads
Betriebskosten	Moderat	Hoch, plus teure Middleware-Lizenzen	Hoch: viele deploybare Einheiten, Pipelines und Bereitschaftspläne

Merkmale der servicebasierten Architektur

Merkmal	Bewertung	Grund
Deploybarkeit	Mittel bis hoch	Jeder Domain Service wird für sich deployt, sodass eine Änderung an Speisekarten kein erneutes Deployment der Zahlungen auslöst; eine gemeinsame Schemaänderung braucht weiterhin Abstimmung.
Fehlertoleranz	Mittel bis hoch	Ein Absturz im Lieferservice lässt Bestellungen und Zahlungen weiterlaufen; eine gemeinsame Datenbank ist weiterhin ein gemeinsamer Ausfallpunkt.
Skalierbarkeit	Mittel	Services skalieren getrennt, aber jeder ist grob, und die gemeinsame Datenbank begrenzt, wie weit sich Schreibvorgänge skalieren lassen.
Elastizität	Niedrig bis mittel	Große Services starten langsam und skalieren in großen Schritten, sodass plötzliche Lastspitzen weniger elegant abgefangen werden als mit kleinen Services.
Datenkonsistenz	Hoch	Die meisten Geschäftsabläufe bleiben innerhalb eines Service, wo eine gewöhnliche Datenbanktransaktion die Daten konsistent hält.
Einfachheit	Mittel	Weit weniger bewegliche Teile als Microservices; dennoch ein verteiltes System mit Netzwerkaufrufen, Deployment-Pipelines und Monitoring pro Service.
Gesamtkosten	Mittel (moderat)	Eine Handvoll Services und eine Datenbank sind günstig im Betrieb verglichen mit Dutzenden Services und Datenbanken.
Testbarkeit	Mittel bis hoch	Jeder Service hat einen kleineren Umfang als der Monolith und kann für sich gegen sein Schema getestet werden.
Evolvierbarkeit	Mittel	Fachliche Partitionierung hält typische Änderungen innerhalb eines Service, aber die gemeinsame Datenbank koppelt die Schemaevolution.

Wann servicebasierte Architektur der pragmatische Mittelweg ist

Wählen Sie sie, wenn das System wenige klare Geschäftsdomänen hat, wenn zwei bis vielleicht acht Teams unabhängig ausliefern müssen, wenn die meisten Abläufe natürlicherweise innerhalb einer Domäne bleiben und wenn starke Konsistenz innerhalb einer Domäne wertvoll ist. Sie ist außerdem ein ausgezeichneter Migrationsschritt aus einem Monolithen heraus: Einen

gut modularisierten Monolithen in eine Handvoll Domain Services zu zerlegen ist weit weniger riskant als der Sprung zu fünfzig Microservices, und das Ergebnis ist oft gut genug, um dort aufzuhören.

Vermeiden Sie sie oder planen Sie, über sie hinauszugehen, wenn Teile einer Domäne sehr unterschiedliche Skalierbarkeit oder Verfügbarkeit brauchen (sie gehören in getrennte Quanten), wenn Dutzende Teams Autonomie brauchen, wenn die gemeinsame Datenbank zum Engpass für Schreibvorgänge geworden ist oder wenn sich die Domain Services ständig gegenseitig aufrufen, um gewöhnliche Anfragen zu erledigen, was ein Zeichen dafür ist, dass die Grenzen falsch sind.

ACHTUNG

Anti-Patterns

- **Die Datenbank, an der sich jeder bedient.** Services lesen direkt die Tabellen der anderen. Innerhalb von Monaten lässt sich das Schema nicht mehr ohne ein koordiniertes Release von allem ändern.
- **Gesprächige Domain Services.** Bestellung ruft Restaurant auf, das Kunde aufruft, das Bestellung aufruft. Brauchen gewöhnliche Abläufe mehrere Services, führen Sie diese zusammen oder verlagern Sie den Ablauf.
- **Die eine riesige Entity-Bibliothek.** Eine einzige gemeinsame Bibliothek mit den Klassen aller Tabellen, sodass jede Schemaänderung jeden Service zu einem neuen Build und Deployment zwingt.
- **SOA unter anderem Namen.** Eine zentrale Integrationsschicht, die nach und nach Routing-Regeln, Transformationen und Geschäftslogik ansammelt: ein neu erfundener ESB mit demselben Engpass.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum können die meisten Abläufe in einer servicebasierten Architektur eine gewöhnliche Datenbanktransaktion nutzen?

Weil jeder Domain Service einen ganzen Geschäftsbereich abdeckt und der Ablauf meist innerhalb dieses Service bleibt, sodass er nur Daten im eigenen Schema des Service in einer Datenbank berührt.

▼ **Was machte den Enterprise Service Bus in vielen SOA-Programmen zum Engpass?**

Routing, Transformation und Prozesslogik lagen im Bus, der typischerweise einem zentralen Team gehörte; daher brauchten die meisten geschäftlichen Änderungen dieses Team, und der Bus war ein gemeinsamer Ausfallpunkt.

▼ **Nennen Sie zwei Techniken, die die Kopplung durch eine gemeinsame Datenbank verringern.**

Jedem Service ein eigenes Schema geben und den Zugriff auf andere verbieten (durch Berechtigungen durchgesetzt) und Entity-Klassen in kleine Bibliotheken pro Domäne aufteilen; außerdem einem Service, wo gerechtfertigt, eine eigene Datenbank geben.

▼ **Was kehrt „Smart Endpoints and Dumb Pipes“ gegenüber der klassischen SOA um?**

Es verlagert die Logik aus der Middleware (dem ESB) in die Services, und das Netzwerk transportiert nur noch Nachrichten.

▼ **Probieren Sie es aus: Ein Ticketing-Unternehmen mit 12 Ingenieuren und einem aufgeräumten Monolithen leidet unter Release-Warteschlangen, und ein Fehler im Berichtswesen hat den Bezahlvorgang zweimal lahmgelegt. Würden Sie eine servicebasierte Architektur oder feingranulare Microservices empfehlen? Begründen Sie das, und nennen Sie ein Risiko Ihrer Wahl.**

Servicebasiert: etwa vier bis sechs grobe Domänen-Services wie Booking, Payments, Venues und Reporting. Die meisten Workflows bleiben innerhalb eines Service, lokale Transaktionen bleiben also erhalten, und synchrone Ketten bleiben kurz. Das isoliert Reporting-Fehler und entschärft Release-Warteschlangen ohne die Betriebskosten Dutzender Services, die 12 Fachleute nicht tragen könnten. Das Risiko ist die gemeinsame Datenbank, die Schemaänderungen koppelt und sie zu einem Quantum macht. Verwenden Sie Schemas pro Service, und trennen Sie die Daten von Reporting.

QUELLEN

- James Lewis and Martin Fowler: Microservices (2014), on smart endpoints and dumb pipes: https://martinfowler.com/articles/microservices.html
- OASIS: Reference Model for Service Oriented Architecture 1.0: https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
- W3C: Web Services Architecture (Working Group Note): https://www.w3.org/TR/ws-arch/
- OASIS: Web Services Business Process Execution Language Version 2.0: https://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

- Enterprise Integration Patterns: messaging patterns catalogue: https://www.enterpriseintegrationpatterns.com/patterns/messaging/

Microservices: Prinzipien, Nutzen und Kosten

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 5.7 Domain-Driven Design
- 6.1 Warum verteilen
- 6.2 Servicebasiert und SOA

Begriffe von früher, die er verwendet: Bounded Context, Architekturquantum, Eventual Consistency, Enterprise Service Bus (ESB), Gesetz von Conway

DIE IDEE IN KÜRZE

Stellen Sie sich statt eines großen Restaurants eine große Markthalle mit Essensständen vor. Jeder Stand hat sein eigenes kleines Team, seine eigene Küche, seinen eigenen Kühlschrank und seine eigene Speisekarte und kann seine Rezepte ändern oder für eine Renovierung schließen, ohne die anderen Stände zu fragen. Die Gäste bestellen an einem gemeinsamen Empfangstresen, der jede Bestellung an den richtigen Stand weiterleitet. Die Markthalle kann wachsen, indem sie Stände hinzufügt, und die ausgebrannte Küche eines Standes legt die anderen nicht lahm. Aber irgendjemand muss den gemeinsamen Tresen, die Lieferungen, die Reinigung und die Sicherheitskontrollen für jeden Stand betreiben, und ein Essen von drei Ständen erfordert sorgfältige Abstimmung. Microservices sind die Markthalle.

WAS SIE WISSEN MÜSSEN

- Microservices (Lewis und Fowler, 2014) bauen eine Anwendung aus kleinen Services, jeder in einem eigenen Prozess, um eine Geschäftsfähigkeit herum gebaut, unabhängig deploybar und über leichtgewichtige Mechanismen kommunizierend.
- Service-Grenzen funktionieren am besten, wenn sie an Bounded Contexts und an Teams ausgerichtet sind; Grenzen, die um technische Schichten oder Datenbanktabellen gezogen werden, ergeben gesprächige, eng gekoppelte Services.
- Unabhängige Deploybarkeit ist die bestimmende Eigenschaft; sie hängt von abwärtskompatiblen Verträgen, Contract Tests und Versionierung ab, nicht nur von getrennten Containern.
- Datenbank pro Service macht jeden Service zu seinem eigenen Quantum, ersetzt aber Joins und Transaktionen durch API-Aufrufe, Events, Sagas und Eventual Consistency.

- Zu den unterstützenden Mustern gehören API-Gateways, Backends for Frontends, Service Discovery und Service Meshes; jedes löst ein echtes Problem und bringt etwas mit, das betrieben werden muss.
- Über die Granularität entscheidet man, indem man Desintegratoren (Umfang, Volatilität, Skalierbarkeit, Fehlertoleranz, Sicherheit, Erweiterbarkeit) gegen Integratoren (Transaktionen, Workflow, gemeinsamer Code, Datenbeziehungen) abwägt.
- Typische Fehlschläge sind der verteilte Monolith, Entity-Services, gesprächige Services und gemeinsame Bibliotheken, die Deployments koppeln; der Stil braucht Automatisierung, Monitoring und Teams, die betreiben, was sie bauen.

Definition und Herkunft

Microservices sind ein Architekturstil, bei dem eine Anwendung als Verbund kleiner Services gebaut wird, von denen jeder in einem eigenen Prozess läuft, um eine Geschäftsfähigkeit herum gebaut ist und durch automatisierte Werkzeuge unabhängig deployt werden kann; sie kommunizieren über leichtgewichtige Mechanismen wie HTTP-APIs oder Messaging, mit einem Minimum an zentraler Steuerung. Diese Zusammenfassung folgt dem Artikel, der den Begriff für den größten Teil der Branche definiert hat: *Microservices* von James Lewis und Martin Fowler, im März 2014 auf martinfowler.com veröffentlicht. Der Artikel berichtet, dass der Name im Mai 2011 auf einem Workshop von Softwarearchitekten in der Nähe von Venedig diskutiert und von der Gruppe im Mai 2012 übernommen wurde; Unternehmen wie Amazon und Netflix arbeiteten bereits auf diese Weise.

Lewis und Fowler gaben keine formale Definition. Sie beschrieben neun gemeinsame Merkmale, die Fachleute bis heute als Checkliste nutzen: Komponentisierung über Services; Organisation um Geschäftsfähigkeiten; Produkte statt Projekte; Smart Endpoints and Dumb Pipes; dezentrale Governance; dezentrales Datenmanagement; Infrastrukturautomatisierung; Design for Failure; und evolutionäres Design. Die folgenden Abschnitte behandeln die architektonisch bedeutsamsten davon der Reihe nach.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Ein *Service* ist hier einfach ein Programm, das eigenständig läuft, meist auf mehreren Servern gleichzeitig, und anderen Programmen eine API anbietet (Kapitel 0). In einem Monolithen wird der Code für Speisekarten, Bestellungen und Zahlungen zu einem Programm kompiliert und gemeinsam deployt. Bei Microservices wird jeder dieser Teile zu einem eigenen Programm, das einem eigenen kleinen Team gehört, eine eigene Datenbank hat und immer dann ausgeliefert wird, wenn dieses Team so weit ist. Der Rest dieses Abschnitts handelt davon, wo die Grenzen zwischen diesen Programmen verlaufen sollten und was es kostet, so viele davon zu betreiben.

An Bounded Contexts ausgerichtete Services

Die schwierigste Frage bei Microservices ist, wo die Grenzen verlaufen. Die verlässlichste Antwort kommt aus dem strategischen Domain-Driven Design (Kapitel 5): Ein Bounded Context ist ein Teil des Geschäfts, innerhalb dessen ein Modell und seine Sprache konsistent sind. Bei der Essenslieferung bedeutet „Bestellung“ in den Bestellungen einen Warenkorb, der bezahlt wird, im Küchenkontext des Restaurants ein Ticket, das gekocht werden muss, und in der Disposition eine Abholung und eine Zustellung. Jeder Kontext ist ein natürlicher Kandidat für einen Service oder für eine kleine Zahl von Services, die einem Team gehören.

Services an Bounded Contexts auszurichten, verleiht ihnen hohe Kohäsion (was sich gemeinsam ändert, liegt zusammen) und geringe Kopplung (Kontexte kommunizieren über wenige explizite Verträge, die an den Rändern übersetzt werden). Es richtet sie außerdem an Teams aus, weil ein Bounded Context meist die Größe hat, die ein Team verstehen und verantworten kann. Services, die um technische Schichten oder um Datenbanktabellen gezogen werden, fehlen beide Eigenschaften, und die Anti-Patterns weiter unten in diesem Abschnitt sind meist die Folge.

Unabhängige Deploybarkeit: die bestimmende Eigenschaft

Wenn eine Eigenschaft Microservices definiert, dann die unabhängige Deploybarkeit: Ein Team kann seinen Service ändern und in Produktion ausliefern, ohne gleichzeitig einen anderen Service zu ändern, zu testen oder auszuliefern. Sam Newman behandelt sie in *Building Microservices* (zweite Auflage, 2021) als die Eigenschaft, die vor allen anderen zu schützen ist, weil fast jeder andere Nutzen von ihr abhängt. Schnellere Auslieferung, kleinerer Blast Radius, gezielte Skalierung und Teamautonomie verschwinden, wenn Releases abgestimmt werden müssen.

Unabhängige Deploybarkeit ist eine Eigenschaft der Verträge, nicht der Container. Zwei Services, die getrennt deployt werden, aber immer gemeinsam ausgeliefert werden müssen, weil eine Änderung am einen den anderen bricht, sind nicht unabhängig deploybar. Um sie zu schützen, braucht es abwärtskompatible Schnittstellenänderungen (Felder hinzufügen, sie aber nie ohne Deprecation-Phase entfernen oder umbenennen), Consumer-Driven Contract Tests und versionierte APIs, die alle unter Kommunikation und APIs behandelt werden.

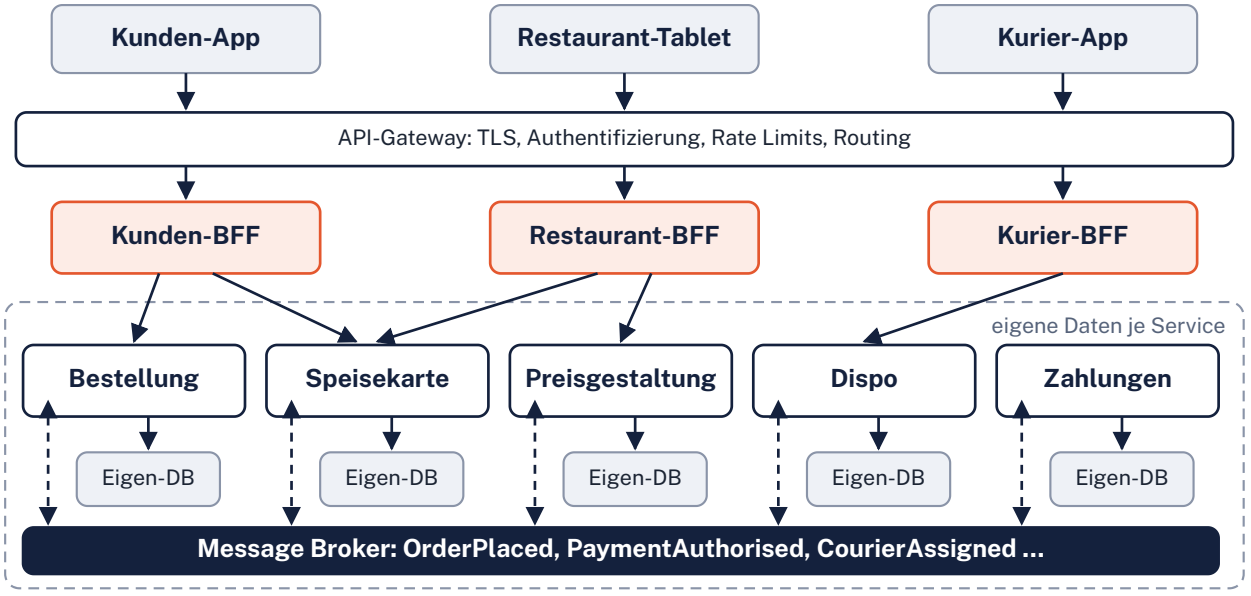
Datenbank pro Service sowie Smart Endpoints and Dumb Pipes

Das dezentrale Datenmanagement von Lewis und Fowler wird in der Praxis zur Regel Datenbank pro Service: Jeder Service besitzt seine Daten, und kein anderer Service liest oder schreibt sie direkt. Andere Services erhalten die Daten über die API des Eigentümers oder über Events, die der Eigentümer veröffentlicht. Diese Regel macht einen Service zu einem eigenen Quantum: Sein Schema kann sich ändern, und er kann skaliert, gesichert und wiederhergestellt werden, ohne jemanden zu fragen. Jeder Service kann außerdem den Speicher wählen, der zu ihm passt (eine relationale Datenbank für Bestellungen, einen Key-Value-Speicher für Sitzungen, eine Suchmaschine für Speisekarten); das nennt man polyglotte Persistenz.

Der Preis ist hoch und muss bewusst bezahlt werden. Abfragen, die früher ein SQL-Join über Tabellen waren, überqueren jetzt Service-Grenzen. Änderungen, die früher eine Transaktion waren, brauchen jetzt eine Saga und werden eventual consistent. Reporting braucht eine eigene Pipeline. Kapitel 7 behandelt die Datenhoheit ausführlich.

Das andere Merkmal mit den größten architektonischen Folgen ist *Smart Endpoints and Dumb Pipes*. Geschäftslogik gehört in die Services; das Netzwerk und der Message Broker transportieren nur Nachrichten. Das ist eine direkte Reaktion auf den Enterprise Service Bus der klassischen SOA, bei der sich Routing-Regeln, Transformationen und Prozesse in zentraler Middleware ansammelten.

Eine Microservice-Topologie für die Essenslieferung



Durchgezogen: synchrone Aufrufe. Gestrichelt: Events, über den Broker veröffentlicht und konsumiert.

Clients erreichen die Services über ein Gateway und je ein Backend for Frontend; jeder Service besitzt seine Daten, und Events fließen über einen Broker.

Unterstützende Muster

Rund um den Stil ist eine Reihe unterstützender Muster entstanden. Jedes löst ein echtes Problem, und jedes fügt eine Komponente hinzu, die betrieben werden muss; führen Sie sie also ein, wenn das Problem auftritt, statt alle auf einmal.

Gängige unterstützende Muster

Muster	Gelöstes Problem	Was er ist	Worauf zu achten ist
API-Gateway	Clients sollten weder die Adressen Dutzender Services kennen noch sich bei jedem einzeln authentifizieren müssen.	Ein einziger Einstiegspunkt, der TLS terminiert, authentifiziert, Rate Limits durchsetzt und Anfragen an Services weiterleitet; er kann mehrere Aufrufe zu einer Antwort zusammenfassen.	Geschäftslogik, die sich ins Gateway einschleicht und einen ESB nachbildet; ein Single Point of Failure, der hohe Verfügbarkeit braucht.
Backend for Frontend (BFF)	Eine mobile App, eine Web-App und eine Partner-API brauchen unterschiedlich geformte Daten; eine allgemeine API bedient keine davon gut.	Ein eigenes, schlankes Backend pro Client-Art, das dem Team dieses Clients gehört, Services aufruft und Antworten formt. Der Name stammt von Phil Calçado bei SoundCloud; Sam Newman machte das Muster 2015 bekannt.	Duplizierte Logik in mehreren BFFs; ein BFF, das zu einem Mini-Monolithen wird.
Service Discovery	Service-Instanzen starten, stoppen und wandern ständig; Adressen lassen sich nicht von Hand konfigurieren.	Ein Verzeichnis der aktuell gesunden Instanzen, das von Clients oder einem Load Balancer abgefragt wird. Kubernetes stellt es über DNS-Namen für Services bereit.	Veraltete Einträge im Verzeichnis; Clients, die Adressen zu lange cachen.
Service Mesh	Jeder Service braucht Mutual TLS, Wiederholungen, Timeouts, Traffic Shifting und Telemetrie; all das in jeder Sprache zu implementieren, ist Verschwendung.	Eine Infrastrukturschicht aus Proxys (traditionell ein Sidecar neben jeder Service-Instanz), die zentral gesteuert wird, etwa Istio oder Linkerd.	Beträchtliche betriebliche Komplexität; Wiederholungen, die sowohl im Mesh als auch im Code konfiguriert sind, vervielfachen die Last.

Fowlers Voraussetzungen und die organisatorischen Kosten

Im August 2014 nannte Martin Fowler drei Dinge, die eine Organisation haben sollte, bevor sie überhaupt Microservices baut. **Schnelles Provisioning**: die Fähigkeit, einen neuen Server oder eine neue Umgebung in Stunden bereitzustellen, was Automatisierung bedeutet. **Grundlegendes Monitoring**: sowohl technische Kennzahlen (Fehler, Latenz, Verfügbarkeit) als auch geschäftliche Kennzahlen (Bestellungen pro Minute) für jeden Service, damit Probleme schnell auffallen. **Schnelles Deployment von Anwendungen**: eine automatisierte Pipeline, die einen Service schnell in Test und Produktion bringt, mit einfachem Rollback. Ohne diese

Voraussetzungen vervielfachen viele kleine Services die manuelle Arbeit. Seine begleitende Idee, die *Microservice Premium* (2015), besagt, dass der Stil feste Kosten an Komplexität und Produktivität mit sich bringt, die sich nur für Systeme auszahlen, die komplex genug sind, um ihn zu brauchen.

Die organisatorischen Kosten sind ebenso real wie die technischen. Der Stil setzt langlebige Teams voraus, die ihre Services in Produktion verantworten, nach dem Prinzip, das Amazons CTO Werner Vogels 2006 als „You build it, you run it" beschrieb. Das bedeutet, dass jedes Team betriebliche Fähigkeiten und einen Rufbereitschaftsplan braucht. Er braucht eine Plattform: Deployment-Pipelines, Observability, Secrets-Management und Service-Vorlagen, meist von einem Plattformteam bereitgestellt (Kapitel 10). Er braucht Einigkeit über übergreifende Standards (Logging-Format, Korrelations-IDs, API-Konventionen), ohne dass jede Änderung zentral genehmigt wird. Und er verlangt, das Gesetz von Conway bewusst zu nutzen: Team- und Service-Grenzen sollten übereinstimmen, sonst müssen sich bei jeder Änderung mehrere Teams abstimmen.

Die Client-Seite: Rendering, Micro-Frontends, Mobile, Offline und das BFF

Kunden begegnen einem System über seine Clients, und Clients haben ihre eigene Architektur. Die erste Entscheidung ist, wo eine Seite gerendert wird. In einer **Single-Page-Anwendung** (SPA) baut der Browser jeden Bildschirm mit JavaScript auf; das passt zu langen, interaktiven Sitzungen wie dem Bestellbildschirm eines Restaurants, doch der Code wächst tendenziell mit der Anwendung und kann die Reaktion auf Berührungen verlangsamen. Beim **serverseitigen Rendering** (SSR) sendet der Server fertiges HTML, sodass Inhalte schnell erscheinen und weniger Code auf dem Gerät läuft, um den Preis von Serverarbeit für jede Anfrage. **Statisches Rendering** erzeugt HTML zur Build-Zeit für ein CDN, was zu Hilfeseiten passt, aber nicht zu personalisierten Seiten. Viele Frameworks rendern auf dem Server und führen dann eine *Hydration* durch, die im Browser Verhalten anfügt; Googles web.dev warnt, dass eine solche Seite bereit aussehen kann, bevor sie reagiert. Die Wahl kann von Seite zu Seite unterschiedlich ausfallen.

Micro-Frontends tragen das unabhängige Deployment in die Oberfläche: „Unabhängig auslieferbare Frontend-Anwendungen werden zu einem größeren Ganzen zusammengesetzt" (Cam Jackson, 2019). Die Teams für Speisekarte und Checkout könnten je ihren eigenen Teil der Website releasen, der zur Laufzeit von einem Tool wie Module Federation von webpack zusammengesetzt wird. Die Kosten sind duplizierte Bibliotheken in jedem Download oder gemeinsame Versionen, die die Kopplung zurückbringen, dazu mehr Pipelines und mehr Aufwand für ein einheitliches Erscheinungsbild. Der Stil antwortet auf das Problem vieler Teams an einem Frontend; ein kleines Team gewinnt wenig.

Mobilgeräte ändern die Release-Regeln. Ein Update erreicht die Nutzer schrittweise, über Apples siebentägige stufenweise Veröffentlichung oder den gestaffelten Rollout von Google Play, und wer einen Play-Rollout stoppt, belässt das Update bei denen, die es schon haben. Alte Versionen bleiben im Einsatz, also müssen APIs lange kompatibel bleiben (siehe Abschnitt 6.5).

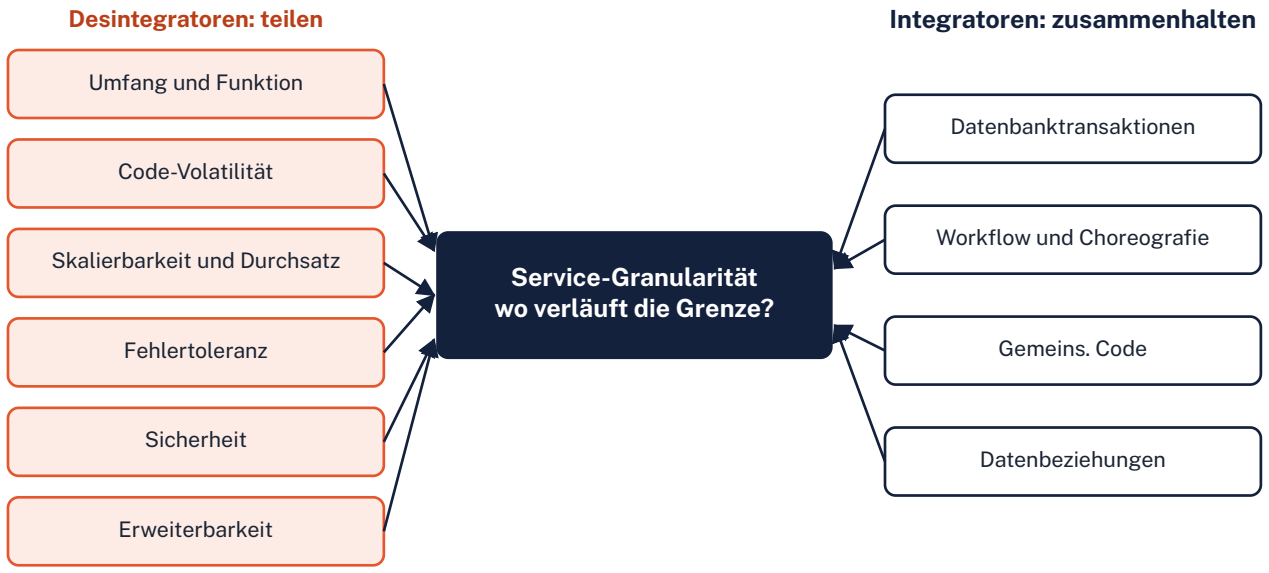
Ein **Offline-First**-Client behandelt einen lokalen Speicher als seine Source of Truth: Bildschirme lesen lokale Daten, und ein Synchronisationsprozess gleicht sie mit dem Server ab, der die letzte Instanz bleibt. Die Android-Dokumentation nennt drei Schreibstrategien: nur online, passend für Zahlungen; in eine Warteschlange gestellt; und verzögert (lazy), wobei zuerst lokal geschrieben wird und Konflikte aufgelöst werden müssen. Ein Kurier in einer Tiefgarage kann eine Bestellung als geliefert markieren und die App das später senden lassen, mit einem Idempotenzschlüssel. In Browsern können Service Worker zwischengespeicherte Seiten ausliefern, aber die Background Sync API funktioniert nicht in allen großen Browsern, prüfen Sie also die aktuelle Unterstützung.

Das **Backend for Frontend** verbindet all das. Sam Newmans Leitlinie lautet: ein Erlebnis, ein BFF, gepflegt vom Team, das diese Oberfläche baut, sodass das Kurier-BFF für schlechte Netze kleine Payloads liefern kann, während das Web-BFF Daten Seite für Seite aufbereitet. Es kann auch Tokens aus dem Browser fernhalten (siehe Abschnitt 8.3). Geschäftsregeln bleiben in den Services: Newman schlägt vor, Logik, die sich BFFs teilen, in einen Domain-Service zu verlagern. Seine Warnung vor der Alternative, einer allgemeinen API für jeden Client, lautet, dass sie zum Engpass wird.

Granularität: Wie groß sollte ein Service sein?

Es gibt keine richtige Größe in Codezeilen oder in Personen. Lewis und Fowler fanden Teams von etwa einem Dutzend Personen, die einen Service verantworteten, und Teams von etwa sechs Personen, die sechs Services verantworteten, und bedauerten, dass das Wort „Micro“ die Aufmerksamkeit auf die Größe lenkt. Ford, Richards, Sadalage und Dehghani bieten in *Software Architecture: The Hard Parts* (2021) ein besseres Werkzeug: Wägen Sie die Kräfte, die auf eine Aufteilung eines Service drängen und die sie **Granularitäts-Desintegratoren** nennen, gegen diejenigen ab, die darauf drängen, ihn zusammenzuhalten, die **Integratoren**.

Kräfte, die auf die Service-Granularität wirken



Kräfte nach Ford, Richards, Sadalage und Dehghani, Software Architecture: The Hard Parts (2021)

Sechs Desintegratoren sprechen für kleinere Services, vier Integratoren für größere. Granularität ist eine Abwägung, die im Einzelfall entschieden wird.

Granularitäts-Desintegratoren und -Integratoren mit Beispielen aus der Essenslieferung

Kraft	Richtung	Frage	Beispiel
Umfang und Funktion	Teilen	Erledigt der Service eine zusammenhängende Aufgabe oder mehrere, die nichts miteinander zu tun haben?	Ein Benachrichtigungsservice, der auch Treuepunkte berechnet, erledigt zwei Aufgaben.
Code-Volatilität	Teilen	Ändert sich ein Teil viel häufiger als der Rest?	Aktionsregeln ändern sich wöchentlich, der Rest der Preisgestaltung vierteljährlich.
Skalierbarkeit und Durchsatz	Teilen	Brauchen Teile sehr unterschiedliche Kapazität?	Standortmeldungen der Kuriere kommen tausendfach pro Sekunde an, Kurier-Onboardings ein paar Mal am Tag.
Fehlertoleranz	Teilen	Würde ein Ausfall in einem Teil unnötigerweise einen anderen lahmlegen?	Ein Absturz im PDF-Quittungsgenerator sollte das Aufgeben von Bestellungen nicht stoppen.
Sicherheit	Teilen	Brauchen Teile unterschiedliche Zugriffskontrollen oder Audits?	Gespeicherte Kartentokens gegenüber öffentlichen Speisekartendaten.
Erweiterbarkeit	Teilen	Werden häufig neue Varianten hinzukommen?	Alle paar Monate neue Zahlungsmethoden, jede eine eigene deploybare Einheit.
Datenbanktransaktionen	Zusammenhalten	Müssen diese Änderungen atomar gelingen oder scheitern?	Eine Bestellung anlegen und ihr Lieferzeitfenster reservieren.
Workflow und Choreografie	Zusammenhalten	Würde eine Aufteilung für gewöhnliche Anfragen lange Aufrufketten erzeugen?	Warenkorb, Checkout und Bestellstatus rufen sich bei jeder Anfrage gegenseitig auf.
Gemeins. Code	Zusammenhalten	Teilen die Teile wesentliche Geschäftslogik, die sich gemeinsam ändert?	Speisekarte und Preisgestaltung teilen komplexe Regeln für Gerichtsoptionen.
Datenbeziehungen	Zusammenhalten	Sind die Daten eng verbunden, mit Joins und Fremdschlüsseln?	Bestellungen und Bestellpositionen; sie zu trennen bringt nichts.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Sollte die Preisgestaltung aufgeteilt werden?

Die Essenslieferplattform hat einen Preisgestaltungsservice, der Warenkorbsummen berechnet: Gerichtspreise, Aufpreise für Optionen, Liefergebühren (die von Entfernung und Nachfrage abhängen) und Aktionen (Gutscheine, Rabatte auf die erste Bestellung, Restaurantangebote). Das Aktionsteam liefert mehrmals pro Woche Änderungen aus, und jedes Release der Preisgestaltung erfordert einen vollständigen Regressionslauf der Gebührenberechnung. Jemand schlägt vor, die Preisgestaltung in die Services Gebühren und Aktionen aufzuteilen. Die Architektin wägt die Kräfte ab.

Code-Volatilität (teilen)	Stark: Aktionen ändern sich wöchentlich, Gebührenregeln vierteljährlich. Eine Trennung ließe Aktionen ausliefern, ohne die Gebühren erneut zu testen.
Skalierbarkeit (teilen)	Schwach: Beide werden einmal pro Warenkorbansicht aufgerufen, mit derselben Rate.
Fehlertoleranz (teilen)	Mittel: Eine fehlerhafte Aktionsregel hat die Preisgestaltung zweimal zum Absturz gebracht. Wären die Aktionen getrennt, ließe sich ein Ausfall auf „kein Rabatt angezeigt“ herabstufen, während die Gebühren weiter funktionieren.
Sicherheit (teilen)	Schwach: gleiche Datensensibilität.
Datenbanktransaktionen (zusammenhalten)	Schwach: Kein Schreibvorgang muss beide umfassen; ein Gutschein wird beim Aufgeben der Bestellung eingelöst, im Bestellservice.
Workflow (zusammenhalten)	Mittel: Jede Warenkorbsumme braucht beide. Nach einer Aufteilung muss die Preisgestaltung (oder ein BFF) die Aktionen bei jeder Warenkorbansicht synchron aufrufen, was einen Hop und dessen Fehlermodi hinzufügt.
Gemeinsamer Code (zusammenhalten)	Mittel: Beide brauchen das Warenkorbmodell und die Rundungsregeln für Währungen, die sich selten ändern.
Datenbeziehungen (zusammenhalten)	Schwach: Aktionen verweisen auf Restaurants und Gerichte nur über Kennungen.
Entscheidung	Aufteilen, mit Bedacht: Aktionen herauslösen; die Preisgestaltung ruft sie mit einem Timeout von 150 ms auf und berechnet den Warenkorb bei einem Fehler ohne Rabatte und markiert ihn zur Neuberechnung im Bezahlvorgang. Nur eine kleine, versionierte Bibliothek für das Warenkorbmodell teilen.

Die Entscheidung beruht darauf, dass zwei starke Desintegratoren (Volatilität und Fehlerisolierung) mittlere Integratoren überwiegen. Hätten Aktionen dieselben Zeilen wie die Gebührenberechnung in einer Transaktion aktualisieren müssen, wäre die Antwort wahrscheinlich anders ausgefallen.

Wer die Kräfte aufschreibt, macht aus einer Geschmacksfrage ein Argument, das andere prüfen können, und die Aufzeichnung gehört in einen Architecture Decision Record.

Anti-Patterns

Die meisten gescheiterten Microservice-Programme scheitern auf erkennbare Weise. Am gravierendsten ist der verteilte Monolith: ein System mit allen Kosten der Verteilung und keiner der Unabhängigkeit, weil seine Services eine Datenbank teilen, einander in langen synchronen Ketten aufrufen oder von denselben deploybaren Bibliotheken abhängen.

Microservice-Anti-Patterns

Anti-Pattern	Symptom	Ursache und Abhilfe
Verteilter Monolith	Services müssen gemeinsam in einer festen Reihenfolge ausgeliefert werden; der Ausfall eines Service stoppt alle.	Gemeinsame Datenbank, synchrone Ketten und gemeinsame deploybare Bibliotheken. Abhilfe: Grenzen um Bounded Contexts neu ziehen, jedem Service seine Daten geben, Aufrufe auf Events umstellen oder alles wieder zu einem modularen Monolithen zusammenführen.
Entity-Services	Services, die nach Substantiven benannt sind (Customer, Order, Restaurant), mit Operationen zum Anlegen, Lesen, Aktualisieren und Löschen; jeder Geschäftsprozess ruft mehrere von ihnen auf.	Grenzen um Daten statt um Verhalten gezogen. Michael Nygard nennt das das Entity-Service-Anti-Pattern. Abhilfe: nach Geschäftsfähigkeiten und Prozessen organisieren.
Gesprächige Services	Eine Benutzeranfrage löst Dutzende Aufrufe zwischen Services aus; Latenz und Fehlerraten steigen mit jedem Feature.	Grenzen schneiden durch einen zusammenhängenden Ablauf, oder die Schnittstellen sind zu feingranular. Abhilfe: gröbere APIs, lokale Kopien von Referenzdaten, Zusammenlegen von Services, die sich immer gemeinsam ändern.
Gemeinsame Bibliotheken, die Deployments koppeln	Eine Änderung an einer gemeinsamen Bibliothek zwingt jeden Service zu neuem Build und Deployment; die Versionen driften auseinander.	Geschäftslogik oder Domänenmodelle in gemeinsamem Code. Abhilfe: nur stabile, technische Bibliotheken teilen; kleine Mengen Geschäftscode duplizieren; alles Geteilte versionieren und mehrere Versionen parallel zulassen.
Microservice-Neid	Ein kleines Team mit einem neuen Produkt startet mit dreißig Services und verbringt die meiste Zeit mit Infrastruktur.	Den Stil aus Mode- oder Lebenslaufgründen statt wegen der Treiber wählen. Abhilfe: mit einem modularen Monolithen beginnen und Services herauslösen, wenn ein Treiber auftaucht.
Nanoservices	Services, die so klein sind, dass der Großteil ihres Codes Infrastruktur-Klempnerei ist; ein Feature erfordert Änderungen in fünf von ihnen.	„Klein“ überstrapaziert. Abhilfe: die Granularitäts-Integratoren nutzen; Services nach Kohäsion bemessen, nicht nach Codezeilen.

Merkmale von Microservices

Merkmal	Bewertung	Grund
Deploybarkeit	Hoch	Jeder Service wird eigenständig ausgeliefert, viele Male am Tag, mit kleinem Blast Radius.
Skalierbarkeit und Elastizität	Hoch	Jeder Service skaliert unabhängig; kleine Services starten schnell und skalieren in feinen Schritten.
Fehlertoleranz	Hoch (wenn darauf ausgelegt)	Ausfälle lassen sich pro Service isolieren, aber nur mit Timeouts, Bulkheads, Fallbacks und asynchroner Kommunikation; sonst pflanzen sich Fehler kaskadenartig fort.
Evolvierbarkeit	Hoch	Kleine, gut abgegrenzte Services lassen sich einzeln neu schreiben oder ersetzen.
Testbarkeit eines Service	Hoch	Jeder Service ist klein; Contract Tests prüfen seine Interaktionen.
Test des Gesamtsystems	Niedrig	Das End-to-End-Verhalten entsteht aus vielen Services und ist in einer Testumgebung schwer nachzustellen.
Performance	Niedrig bis mittel	Netzwerk-Hops, Serialisierung und Sicherheitsprüfungen fügen jeder Interaktion Latenz hinzu.
Datenkonsistenz	Niedrig	Keine gemeinsamen Transaktionen; Geschäftsinvarianten über Services hinweg beruhen auf Eventual Consistency und Sagas.
Einfachheit	Niedrig	Viele bewegliche Teile, viele Fehlermodi, viel Infrastruktur.
Gesamtkosten	Niedrig (teuer)	Plattform, Werkzeuge, Observability und Personal, um Dutzende Services zu betreiben.

Wann Microservices passen und wann nicht

Microservices passen zu Organisationen mit vielen Teams, die an einem Produkt arbeiten und unabhängig ausliefern müssen, zu Systemen, deren Teile wirklich unterschiedliche Anforderungen an Skalierbarkeit, Verfügbarkeit oder Sicherheit haben, und zu Domänen, die gut genug verstanden sind, um stabile Grenzen zu ziehen. Sie erfordern Investitionen in eine Plattform und in betriebliche Fähigkeiten, bevor der erste Service abgespalten wird.

Sie passen nicht zu einem kleinen Team, zu einem neuen Produkt, dessen Domäne noch erkundet wird (zu früh gezogene Grenzen sind teuer über das Netzwerk hinweg zu verschieben), zu einem System, dessen Kernabläufe über den Großteil seiner Daten starke transaktionale Konsistenz brauchen, oder zu einer Organisation ohne automatisiertes Deployment und Monitoring. Fowlers Rat *MonolithFirst* (Kapitel 5) und die Kehrtwenden, die

manche Unternehmen dokumentiert haben, etwa die Zusammenführung von mehr als 140 Destination-Services zurück in einen einzigen bei Segment im Jahr 2018, zeigen, dass der Stil ein Werkzeug mit Kosten ist, kein Ziel.

HINWEIS

Fachleute sind sich uneinig, und das ist nützlich

Manche erfahrenen Architekten argumentieren, dass die meisten Organisationen nie über einen modularen Monolithen oder eine servicebasierte Architektur hinausgehen sollten; andere argumentieren, dass kleine Services mit einer guten Plattform für viele Teams die produktivste Arbeitsweise sind. Beide Lager sind sich über die Bedingungen einig: klare Domänengrenzen, unabhängige Deploybarkeit, Datenhoheit und Automatisierung. Wo diese Bedingungen fehlen, ist die Zahl der Services nicht das Problem, das es zu lösen gilt.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Zwei Services werden in getrennten Containern deployt, müssen aber immer gemeinsam ausgeliefert werden. Sind sie unabhängig deploybar?

Nein. Bei unabhängiger Deploybarkeit geht es darum, einen Service ohne den anderen ändern und ausliefern zu können; ein gekoppelter Vertrag oder eine gemeinsame Datenbank macht sie für Releases zu einer Einheit.

▼ Was ist der Unterschied zwischen einem API-Gateway und einem Backend for Frontend?

Ein Gateway ist ein allgemeiner Einstiegspunkt, der übergreifende Anliegen wie TLS, Authentifizierung, Rate Limiting und Routing übernimmt; ein BFF ist ein clientspezifisches Backend, das Daten für eine Art von Frontend aufbereitet und dem Team dieses Frontends gehört.

▼ Welcher Granularitäts-Integrator spricht am stärksten dagegen, zwei Teile aufzuteilen, die Daten atomar aktualisieren müssen?

Datenbanktransaktionen: Eine Aufteilung würde eine ACID-Transaktion durch einen verteilten Ablauf wie eine Saga ersetzen.

▼ **Warum sind Services namens Customer, Order und Restaurant, die nur Operationen zum Anlegen, Lesen, Aktualisieren und Löschen bieten, ein Warnsignal?**

Es sind Entity-Services: Grenzen um Daten statt um Verhalten, sodass jeder Geschäftsprozess mehrere Services aufrufen muss, was gesprächige, eng gekoppelte Interaktionen erzeugt.

▼ **Probieren Sie es aus: Ein Team schlägt CustomerService, OrderService und OrderLineService vor, die jeweils eine Tabelle kapseln. Um eine Bestellung aufzugeben, müssen eine Bestellung und fünf Positionen gemeinsam geschrieben werden. Ist OrderLineService nach Desintegratoren und Integratoren eine gute Aufteilung? Was würden Sie tun?**

Nein. Es ist ein Entity-Service um eine Tabelle, keine Geschäftsfähigkeit. Bestellungen und Positionen teilen eine Transaktion, einen Workflow und Datenbeziehungen, alles starke Integratoren, und kein Desintegrator wie unterschiedliche Skalierung, Volatilität oder Sicherheit greift. Getrennt wäre es gesprächig, mit einer Saga für das, was jetzt eine lokale Transaktion erledigt. Führen Sie die Positionen in einen Ordering-Service zusammen, der auf seinen Bounded Context ausgerichtet ist. Trennen Sie erst, wenn echte Kräfte auftreten.

QUELLEN

- James Lewis and Martin Fowler: Microservices (2014): https://martinfowler.com/articles/microservices.html
- Martin Fowler: Microservice Prerequisites (2014): https://martinfowler.com/bliki/MicroservicePrerequisites.html
- Sam Newman: Pattern: Backends For Frontends: https://samnewman.io/patterns/architectural/bff/
- Michael Nygard: The Entity Service Antipattern: https://michaelnygard.com/blog/2017/12/the-entity-service-antipattern/
- Thoughtworks: Software Architecture: The Hard Parts, chapter 7 on service granularity (sample PDF): https://www.thoughtworks.com/content/dam/thoughtworks/documents/books/bk_software_architecture_hard_parts_ch7_en.pdf
- Cam Jackson: Micro Frontends (martinfowler.com, 2019): https://martinfowler.com/articles/micro-frontends.html
- web.dev: Rendering on the Web: https://web.dev/articles/rendering-on-the-web
- webpack: Module Federation: https://webpack.js.org/concepts/module-federation/
- Apple: Release a version update in phases: https://developer.apple.com/help/app-store-connect/update-your-app/release-a-version-update-in-phases/

- Google Play Console Help: staged rollouts: https://support.google.com/googleplay/android-developer/answer/6346149
- Android Developers: Build an offline-first app: https://developer.android.com/topic/architecture/data-layer/offline-first
- MDN: Offline and background operation: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Offline_and_background_operation
- MDN: Background Synchronization API: https://developer.mozilla.org/en-US/docs/Web/API/Background_Synchronization_API

Ereignisgesteuerte Architektur: Broker, Mediatoren und Event-Streams

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern
- 0.6 APIs und Integration
- 6.1 Warum verteilen

Begriffe von früher, die er verwendet: Asynchrones Messaging, Idempotent, Transaktion, Bounded Context, Kopplung

DIE IDEE IN KÜRZE

Denken Sie an eine geschäftige Restaurantküche. Wenn ein Kellner einen Bestellzettel an die Leiste heftet, bekommt niemand gesagt, was zu tun ist: Der Grillkoch sieht, dass ein Steak zu braten ist, die Salatstation sieht einen Salat, und der Patissier sieht ein Dessert für später. Jede Station reagiert auf den Zettel in ihrem eigenen Tempo. Macht der Patissier gerade Pause, wartet der Zettel an der Leiste, bis er zurückkommt. Eine neue Station hinzuzufügen, ändert nichts daran, wie Kellner ihre Zettel schreiben. Ein ereignisgesteuertes System funktioniert genauso: Teile verkünden, was geschehen ist, und andere Teile reagieren.

WAS SIE WISSEN MÜSSEN

- Ein Ereignis ist eine eingetretene Tatsache (Vergangenheitsform, viele mögliche Empfänger); ein Command ist eine Anfrage an einen Handler, die abgelehnt werden kann; eine Nachricht ist der Umschlag, der beides transportiert.
- Die Broker-Topologie verteilt einen Ablauf auf unabhängige Reaktionen (Choreografie); die Mediator-Topologie legt die Abfolge in einen Koordinator (Orchestrierung). Choreografie koppelt am wenigsten; Orchestrierung macht Abläufe sichtbar und Ausfälle leichter beherrschbar.
- Fowler unterscheidet Event Notification, Event-Carried State Transfer, Event-Sourcing und CQRS; die ersten beiden sind Wege, wie Services kommunizieren, die letzten beiden sind Wege, wie ein Service Daten speichert.
- In Kafka ist ein Topic in Partitionen aufgeteilt; Ereignisse mit demselben Schlüssel landen in derselben Partition, die Reihenfolge ist nur innerhalb einer Partition garantiert, und jede Consumer Group verteilt die Partitionen auf ihre Mitglieder und verfolgt ihre eigenen

Offsets.

- Die Zustellung kann at-most-once, at-least-once oder effectively-once erfolgen; eine Exactly-once-Zustellung ist im Allgemeinen nicht erreichbar, daher sollten Sie von at-least-once ausgehen und jeden Consumer idempotent machen.
- Veröffentlichte Ereignisse sind langlebige Verträge: Geben Sie ihnen Kennungen, Versionen und Zeitstempel, entwickeln Sie sie kompatibel weiter und prüfen Sie sie mit einer Schema Registry.
- Wiederholen Sie vorübergehende Fehler mit Backoff, verschieben Sie Poison Messages in eine überwachte Dead Letter Queue, und lassen Sie nie zu, dass eine einzige fehlerhafte Nachricht eine Partition blockiert.

Definition: auf Geschehnisse reagieren

In einer ereignisgesteuerten Architektur (EDA) kommunizieren Komponenten, indem sie Ereignisse erzeugen und auf sie reagieren: Aufzeichnungen von Tatsachen, die im Geschäft eingetreten sind, etwa *eine Bestellung wurde aufgegeben* oder *ein Kurier wurde zugewiesen*. Die Komponente, die ein Ereignis veröffentlicht, ruft die Komponenten, die darauf reagieren, nicht auf und weiß meist nicht einmal, dass es sie gibt. Ein Message Broker transportiert Ereignisse von Produzenten zu Consumern, hält sie vor, während Consumer beschäftigt oder ausgefallen sind, und erlaubt es, neue Consumer hinzuzufügen, ohne den Produzenten zu ändern.

Ereignisgesteuerte Designs sind nicht neu: Grafische Benutzeroberflächen, Betriebssysteme und Handelssysteme arbeiten seit Jahrzehnten so, und Enterprise-Messaging-Produkte sind älter als das Web. Was sich in den 2010er-Jahren änderte, waren Größenordnung und Dauerhaftigkeit. Logbasierte Broker wie Apache Kafka, entstanden bei LinkedIn und 2011 als Open Source veröffentlicht, machten es praktikabel, jedes Geschäftsereignis tage- oder jahrelang aufzubewahren und erneut abzuspielen. EDA kann der Hauptstil eines Systems sein oder, häufiger, die Art, wie die Services eines Microservice- oder servicebasierten Systems miteinander sprechen.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Ein *Message Broker* ist ein Programm, das zwischen anderen Programmen sitzt und Nachrichten weitergibt, ein wenig wie eine Poststelle. Ein Sender übergibt dem Broker eine Nachricht und arbeitet weiter; der Broker bewahrt sie sicher auf, bis jeder Empfänger bereit ist, sie abzuholen. Weil der Sender nicht wartet, spielt es keine Rolle, ob ein Empfänger beschäftigt oder kurz offline ist. Kapitel 0 hat diese Idee als asynchrones Messaging über eine Queue eingeführt; dieser Abschnitt zeigt, wie ganze Architekturen darum herum gebaut werden.

Ereignisse, Commands und Nachrichten

Drei Wörter werden oft unscharf verwendet, und der Unterschied ist wichtig. Eine Nachricht ist der Umschlag: beliebige Daten, die von einem Programm über einen Kanal an ein anderes gesendet werden. Was die Nachricht transportiert, kann ein Ereignis sein (eine Tatsache, in der Vergangenheitsform benannt), ein Command (eine Anfrage an einen Handler, als Anweisung benannt) oder eine Abfrage und ihre Antwort. Gregor Hohpe und Bobby Woolf haben diese Nachrichtentypen und die Kanäle, die sie transportieren, in *Enterprise Integration Patterns* (2003) katalogisiert, und ihr Vokabular ist bis heute der Standard.

Ereignis, Command und Nachricht

	Ereignis (Event)	Command	Nachricht (der Umschlag)
Bedeutung	Eine Tatsache: etwas, das geschehen ist	Eine Anfrage: Bitte tu etwas	Beliebige Daten, die von einem Programm über einen Kanal an ein anderes gesendet werden
Benennung	Vergangenheitsform: `OrderPlaced`, `PaymentAuthorised`	Imperativ: `AuthorisePayment`, `AssignCourier`	Transportiert ein Ereignis, ein Command, eine Abfrage oder eine Antwort
Empfänger	Null, einer oder viele; der Publisher kennt sie nicht	Genau ein vorgesehener Handler	Hängt vom Kanal ab: eine Queue (einer) oder ein Topic (viele)
Kann es abgelehnt werden?	Nein; es ist bereits geschehen. Die Empfänger entscheiden, wie sie reagieren	Ja; der Handler kann es ablehnen oder daran scheitern	Nicht anwendbar
Kopplung	Der Publisher hängt nicht von den Consumern ab	Der Sender hängt von der Fähigkeit und dem Vertrag des Empfängers ab	Hängt davon ab, was sie transportiert

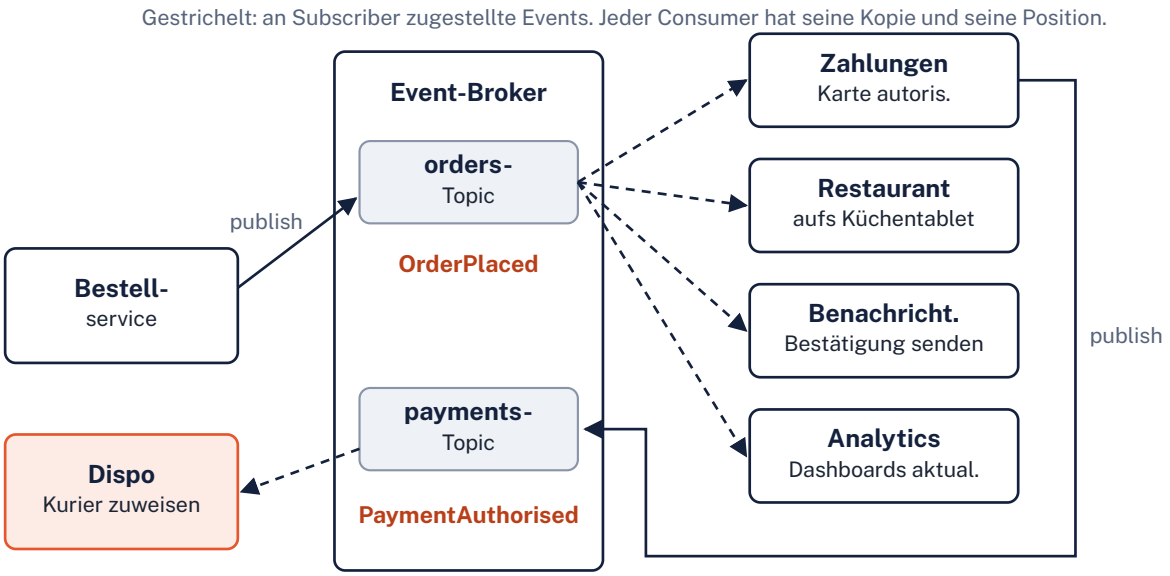
Nachrichten laufen über zwei Arten von Kanälen. Eine Queue stellt jede Nachricht einem ihrer Consumer zu, was zu Commands und zur Arbeitsverteilung passt. Ein Topic stellt jedem Subscriber eine Kopie jeder Nachricht zu; das ist Publish-Subscribe (Pub-Sub) und passt zu Ereignissen. Ein Design, das `OrderPlaced` an ein Topic sendet, lässt Zahlungen, Restaurant, Benachrichtigungen und Analytics reagieren, jeden in seinem eigenen Tempo; einen Betrugserkennungsservice im nächsten Jahr hinzuzufügen, bedeutet ein neues Abonnement und keine Änderung an den Bestellungen.

Broker-Topologie und Mediator-Topologie

Richards und Ford beschreiben zwei Topologien für ereignisgesteuerte Systeme. In der Broker-Topologie gibt es keinen zentralen Koordinator. Ein auslösendes Ereignis wird veröffentlicht; jeder interessierte Prozessor reagiert und kann weitere Ereignisse veröffentlichen, auf die wiederum andere reagieren. Der Ablauf existiert nur als Summe dieser Reaktionen. Das ist Choreografie: Wie Tänzer, von denen jeder seine eigenen Schritte kennt, lenkt niemand das Ganze.

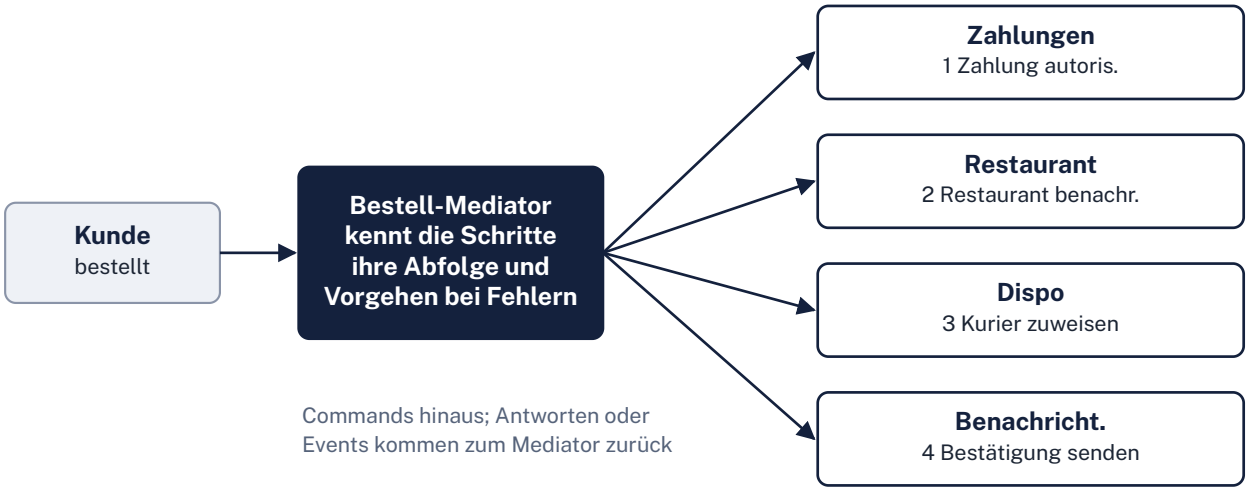
In der Mediator-Topologie empfängt ein Event-Mediator das auslösende Ereignis und steuert den Ablauf, indem er Commands in einer festgelegten Reihenfolge an Prozessoren sendet und auf ihre Antworten reagiert. Das ist Orchestrierung: Ein Dirigent kennt die Partitur. Mediatoren reichen von einem einfachen Service mit Zustandsautomat bis zu einer Workflow-Engine wie Temporal, AWS Step Functions oder einer BPMN-Engine.

Broker-Topologie: der OrderPlaced-Ablauf



In der Broker-Topologie verkündet der Bestellservice nur die Tatsache; jeder Consumer reagiert unabhängig, und das eigene Ereignis des Zahlungsservice treibt den nächsten Schritt an.

Mediator-Topologie: derselbe Ablauf, orchestriert



In der Mediator-Topologie verantwortet eine Komponente die Abfolge, sodass der Ablauf sichtbar ist und Ausfälle an einer Stelle behandelt werden.

Choreografie oder Orchestrierung?

Frage	Choreografie (Broker)	Orchestrierung (Mediator)
Kopplung	Am geringsten: Publisher kennen die Consumer nicht	Der Mediator hängt vom Vertrag jedes Beteiligten ab
Den gesamten Ablauf sehen	Schwer: Er ist über Services verteilt; zur Rekonstruktion braucht es Tracing	Leicht: Er ist an einer Stelle niedergeschrieben
Fehlerbehandlung	Jeder Beteiligte muss Fehler erkennen und darauf reagieren, oft durch Veröffentlichen kompensierender Ereignisse	Der Mediator entscheidet zentral über Wiederholungen, Timeouts und Kompensationen
Einen Schritt hinzufügen	Einen Subscriber hinzufügen; sonst ändert sich nichts	Den Mediator ändern
Risiken	Zyklische Ereignisketten; niemand bemerkt, wenn ein Schritt stillschweigend ausfällt	Der Mediator wird zum Engpass oder sammelt Geschäftslogik an
Gut geeignet für	Einfache, stabile Abläufe; viele unabhängige Reaktionen auf eine Tatsache	Lange, bedingte Abläufe mit Kompensation, etwa die Bestell-Saga

Die meisten realen Systeme mischen beides: Choreografie zwischen Bounded Contexts (die Bestellungen verkünden Tatsachen und kümmern sich nicht darum, wer zuhört) und Orchestrierung innerhalb eines Kontexts, wo ein mehrstufiger Prozess einen klaren Verantwortlichen braucht.

Vier Arten, Ereignisse zu nutzen

In einem Artikel von 2017 wies Martin Fowler darauf hin, dass „ereignisgesteuert“ für mindestens vier ganz verschiedene Muster verwendet wird und dass Gespräche schiefgehen, wenn Menschen verschiedene davon meinen. Sie auseinanderzuhalten, ist eine der nützlichsten Gewohnheiten, die ein Architekt haben kann.

Fowlers vier Event-Muster (2017)

Muster	Was das Ereignis transportiert und warum	Nutzen	Kosten
Event Notification	Ein kleines Signal, dass sich etwas geändert hat, oft nur eine Kennung: `OrderPlaced {order_id}`. Empfänger fragen bei Bedarf die Details nach.	Sehr geringe Kopplung; das Modell des Publishers bleibt privat.	Rückfragen erzeugen Last und eine Laufzeitabhängigkeit vom Publisher; der Ablauf über viele Ereignisse hinweg ist schwer zu überblicken.
Event-Carried State Transfer	Genug Daten, damit Empfänger eine eigene Kopie führen können: die vollständige Bestellung mit Positionen und Summen.	Empfänger arbeiten, ohne den Publisher aufzurufen, selbst wenn er ausgefallen ist; geringere Latenz.	Daten werden weit verteilt kopiert und sind eventual consistent; größere Ereignisse; Schemas werden zu Verträgen.
Event-Sourcing	Jede Änderung am Zustand einer Entität wird als Ereignis gespeichert; das Ereignisprotokoll ist die maßgebliche Quelle, und der aktuelle Zustand wird durch erneutes Abspielen abgeleitet.	Vollständiger Audit-Trail; frühere Zustände rekonstruieren; neue Sichten aus alter Historie.	Komplex; erneutes Abspielen ist schwierig, wenn Ergebnisse von externen Systemen abhingen; Ereignisschemas müssen sich für immer weiterentwickeln.
CQRS	Getrennte Modelle (oft getrennte Speicher) für das Schreiben und das Lesen, häufig über Ereignisse gespeist.	Jede Seite ist für ihre Aufgabe optimiert; passt zu leseintensiven Systemen mit komplexen Abfragen.	Zwei Modelle, die synchron gehalten werden müssen; oft dort eingesetzt, wo ein Modell genügen würde.

Event Notification und Event-Carried State Transfer betreffen die Kommunikation zwischen Services und gehören in dieses Kapitel. Event-Sourcing und CQRS betreffen die Art, wie ein Service seine Daten speichert; sie werden oft mit EDA kombiniert, sind aber unabhängige Entscheidungen und werden in Kapitel 7 behandelt.

Event-Streams und Logs: wie Kafka funktioniert

Herkömmliche Message Queues löschen eine Nachricht, sobald ein Consumer sie bestätigt. Ein logbasierter Broker wie Apache Kafka speichert Ereignisse stattdessen in einem nur anfügbaren Log und bewahrt sie für eine konfigurierte Aufbewahrungsdauer (oder für immer)

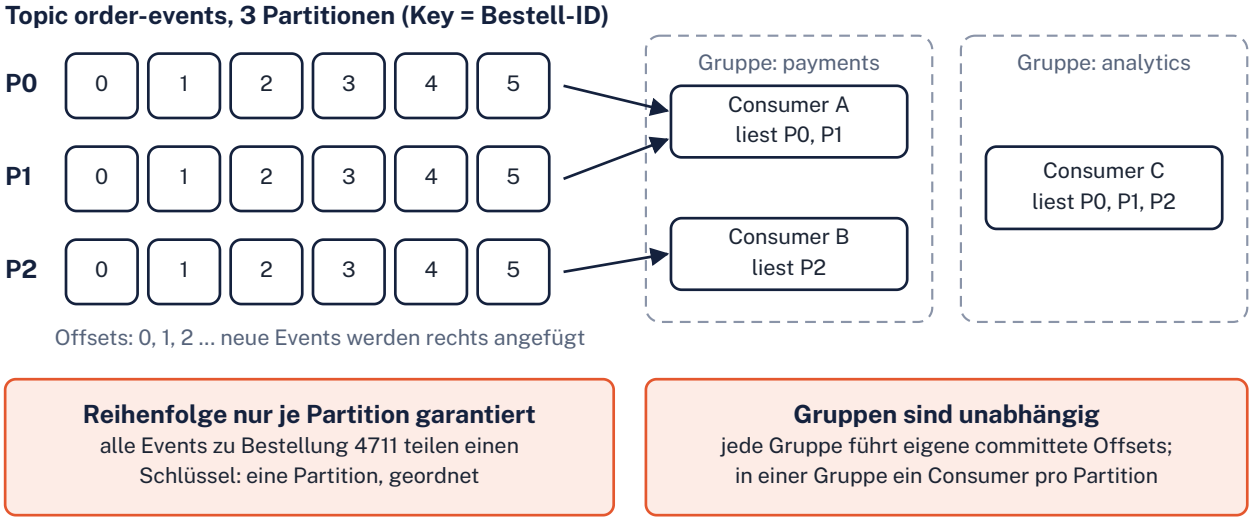
auf, unabhängig davon, ob sie gelesen wurden. Aus diesem Design folgen mehrere Dinge, und sie erklären sowohl Kafkas Stärken als auch seine Fallstricke.

Ein Topic ist in Partitionen aufgeteilt, von denen jede ein geordnetes Log ist, das über Broker hinweg gespeichert und repliziert wird. Ein Produzent wählt die Partition, normalerweise durch Hashing des Ereignisschlüssels, sodass alle Ereignisse mit demselben Schlüssel (etwa derselben Bestell-ID) in derselben Partition landen. Jedes Ereignis in einer Partition hat eine fortlaufende Position, seinen Offset. Consumer holen Ereignisse ab und speichern den Offset, den sie erreicht haben; sie können auch zurückspulen und erneut abspielen.

Consumer treten einer Consumer Group bei. Innerhalb einer Gruppe wird jede Partition zu jedem Zeitpunkt von genau einem Consumer gelesen, sodass sich die Gruppe die Arbeit teilt; verschiedene Gruppen lesen unabhängig voneinander, jede mit ihren eigenen Offsets, und so wird Pub-Sub erreicht. Die Zahl der Partitionen begrenzt daher die Parallelität einer Gruppe: Ein Topic mit 12 Partitionen kann höchstens 12 Consumer einer Gruppe beschäftigen. (Die *Share Groups* von Kafka, in früheren 4.x-Releases als Vorschau verfügbar und seit Kafka 4.2 von 2026 produktionsreif, sind eine Queue-ähnliche Alternative: Mehrere Consumer können sich eine Partition teilen und Nachrichten einzeln bestätigen, die Reihenfolge innerhalb der Partition bleibt dann aber nicht erhalten. Dieser Abschnitt beschreibt klassische Consumer Groups.)

Kafka garantiert die Reihenfolge nur innerhalb einer Partition. Ereignisse zu einer Bestellung kommen in der Reihenfolge an, in der sie geschrieben wurden, wenn sie einen Schlüssel teilen; Ereignisse zu verschiedenen Bestellungen in verschiedenen Partitionen haben zueinander keine definierte Reihenfolge. Designs, die eine globale Reihenfolge brauchen, müssen eine einzige Partition verwenden (und damit auf Parallelität verzichten) oder, besser, so umgestaltet werden, dass nur die Reihenfolge pro Entität zählt.

Partitionen, Offsets und Consumer Groups



Reihenfolge nur je Partition garantiert
alle Events zu Bestellung 4711 teilen einen Schlüssel: eine Partition, geordnet

Gruppen sind unabhängig
jede Gruppe führt eigene committete Offsets; in einer Gruppe ein Consumer pro Partition

Jede Partition ist ein geordnetes Log; eine Consumer Group verteilt die Partitionen auf ihre Mitglieder, und jede Gruppe verfolgt ihre eigenen Offsets.

Zustellgarantien und was „exactly once“ wirklich bedeutet

Weil Netzwerke und Prozesse ausfallen (Abschnitt 6.1), müssen ein Broker und seine Clients festlegen, was geschieht, wenn nach einem Absturz oder Timeout unklar ist, ob eine Nachricht verarbeitet wurde. Es gibt drei mögliche Garantien, und welche gilt, hängt davon ab, wann Bestätigungen und Offsets gespeichert werden.

Zustellgarantien

Garantie	Was sie verspricht	Wie sie entsteht	Typische Folge
At-most-once	Jede Nachricht wird einmal oder gar nicht verarbeitet, nie zweimal.	Der Consumer speichert seine Position (committet den Offset oder bestätigt) vor der Verarbeitung, oder der Produzent wiederholt nie.	Nachrichten können nach einem Absturz verloren gehen. Akzeptabel für Messwerte oder Standortpings von Kurieren, bei denen die nächste Aktualisierung die letzte ersetzt.
At-least-once	Jede Nachricht wird einmal oder mehrmals verarbeitet, geht aber nie verloren.	Der Produzent wiederholt, bis er eine Bestätigung erhält, und der Consumer committet seine Position erst nach der Verarbeitung.	Duplikate nach Abstürzen, Wiederholungen und Rebalances. Der Standard der meisten Broker und die übliche Wahl für Geschäftsereignisse.
Effectively-once	Die Wirkung jeder Nachricht wird genau einmal angewendet, auch wenn sie mehr als einmal zugestellt werden kann.	At-least-once-Zustellung plus ein idempotenter Consumer oder eine Transaktion, die das Ergebnis und die Position des Consumers gemeinsam speichert.	Die praktische Bedeutung von „exactly once“ in Geschäftssystemen; erfordert bewusstes Design in jedem Consumer.

„Exactly once“ muss sorgfältig definiert werden. Eine Exactly-once-*Zustellung* über ein unzuverlässiges Netzwerk ist im Allgemeinen nicht erreichbar: Ein Sender, der keine Bestätigung erhält, kann nicht wissen, ob er erneut senden soll. Was Systeme bieten können, ist eine Exactly-once-*Wirkung* innerhalb einer definierten Grenze. Kafka zum Beispiel bietet seit Version 0.11 (2017) einen idempotenten Produzenten und Transaktionen, sodass ein Lesen-Verarbeiten-Schreiben-Zyklus von Kafka-Topics zu Kafka-Topics seine Ausgabe und seine Consumer-Offsets atomar committet. Diese Garantie endet an der Grenze von Kafka. Belastet der Consumer eine Karte, sendet er eine E-Mail oder schreibt er in eine andere Datenbank, führt ein Absturz zwischen dieser Aktion und dem Offset-Commit dazu, dass die Aktion wiederholt wird, es sei denn, der Consumer selbst ist idempotent oder speichert seine Offsets in derselben Transaktion wie seine Ergebnisse.

Die praktische Regel für Architekten ist daher einfach: Gehen Sie von At-least-once-Zustellung aus und machen Sie jeden Consumer zu einem idempotenten Consumer. Geben Sie jedem Ereignis eine eindeutige Kennung; speichern Sie im Consumer die verarbeiteten Kennungen in derselben lokalen Transaktion wie die Zustandsänderung, und überspringen Sie jedes bereits gespeicherte Ereignis. Machen Sie die Operation nach Möglichkeit auch von Natur aus idempotent, etwa „Status auf PAID setzen, wenn er AWAITING_PAYMENT ist“ statt „eine Zahlung hinzufügen“.

```
# Idempotent consumer: processing the same event twice has the effect of once
def handle_payment_authorised(event, db):
    with db.transaction():                      # one local ACID transaction
        inserted = db.execute(
            "INSERT INTO processed_events (event_id) VALUES (%s) "
            "ON CONFLICT (event_id) DO NOTHING",
            [event["event_id"]],
        ).rowcount
        if inserted == 0:
            return                              # seen before: do nothing
        db.execute(
            "UPDATE orders SET status = 'PAID' WHERE order_id = %s "
            "AND status = 'AWAITING_PAYMENT'",
            [event["order_id"]],
        )
    # commit the broker offset only after the transaction has committed
```

Ein idempotenter Consumer: Der Eintrag der verarbeiteten Ereignisse und die Zustandsänderung werden gemeinsam committet, sodass ein erneut zugestelltes Ereignis nichts ändert.

Ereignisschemas und Versionierung

Sobald andere Teams ein Ereignis konsumieren, ist seine Struktur ein veröffentlichter Vertrag, genau wie eine API, und er hält länger: Bei einem logbasierten Broker können Consumer Ereignisse lesen, die vor Monaten geschrieben wurden. Ein gutes Ereignis hat eine eindeutige Kennung, einen Typ, eine Schemaversion, den Zeitpunkt seines Eintretens, die Kennungen der beteiligten Entitäten und die Geschäftsdaten, die Consumer brauchen. Geldbeträge werden in der kleinsten Währungseinheit mit Angabe der Währung gespeichert, nie als Gleitkommazahl (Kapitel 0).

```
{
  "event_id": "9f1c2e7a-4b0d-4e55-9a51-0c3d8f2b6a10",
  "type": "OrderPlaced",
  "schema_version": 2,
  "occurred_at": "2026-10-22T18:42:07Z",
  "order_id": "ord-4711",
  "customer_id": "cus-208",
  "restaurant_id": "res-31",
  "items": [{"dish_id": "d-5", "qty": 2, "price_minor": 1150}],
  "total_minor": 2450,
  "currency": "GBP"
}
```

Ein OrderPlaced-Ereignis mit Event-Carried State Transfer: Kennungen, Version, Zeit und die Daten, die Consumer brauchen.

Schemas sollten sich abwärtskompatibel weiterentwickeln: optionale Felder hinzufügen, Felder nie ohne neue Version entfernen, umbenennen oder in ihrer Bedeutung ändern und einen Feldnamen nie für etwas anderes wiederverwenden. Viele Teams setzen das mit einem Schemaformat wie Avro, Protocol Buffers oder JSON Schema und einer Schema Registry durch, die inkompatible Änderungen beim Veröffentlichen zurückweist. Die CloudEvents-Spezifikation, ein graduiertes Projekt der Cloud Native Computing Foundation, standardisiert die gemeinsamen Umschlagattribute wie id, source, type und time. Kompatibilitätsmodi und Migrationstechniken werden unter Schemaevolution behandelt.

Fehlerbehandlung: Wiederholungen, Poison Messages und Dead Letter Queues

Consumer scheitern aus zwei sehr verschiedenen Gründen, und diese brauchen eine unterschiedliche Behandlung. **Vorübergehende Fehler**, etwa eine kurzzeitig nicht verfügbare Datenbank oder ein Timeout beim Aufruf eines anderen Service, gelingen meist beim erneuten Versuch; wiederholen Sie einige Male mit exponentiellem Backoff und Jitter (zunehmenden, zufällig gestreuten Verzögerungen), damit eine angeschlagene Abhängigkeit nicht überlastet wird. **Dauerhafte Fehler**, etwa ein fehlerhaft aufgebautes Ereignis oder eines, das gegen eine Geschäftsregel verstößt, scheitern jedes Mal. Eine solche „Poison Message“ darf den Consumer nicht für immer blockieren, was auf einem partitionierten Log jedes spätere Ereignis in dieser Partition aufhalten würde.

Die Standardlösung ist eine Dead Letter Queue (DLQ): Nach einer festgelegten Zahl von Versuchen wird die Nachricht in einen separaten Kanal verschoben, der Fehler wird festgehalten, und der Consumer macht weiter. Eine DLQ ist nur nützlich, wenn jemand sie beobachtet: Alarmieren Sie bei ihrer Größe, geben Sie einem benannten Team die Verantwortung, stellen Sie Werkzeuge bereit, um Nachrichten zu untersuchen, zu korrigieren und erneut abzuspielen, und legen Sie vorab fest, welche Reihenfolgegarantien verloren gehen, wenn eine Nachricht verspätet erneut abgespielt wird. Manche Teams schalten vor der DLQ ein oder mehrere Retry-Topics mit längeren Verzögerungen dazwischen.

Merkmale der ereignisgesteuerten Architektur

Merkmal	Bewertung	Grund
Skalierbarkeit und Elastizität	Hoch	Consumer skalieren unabhängig und fangen Lastspitzen aus dem Puffer des Brokers ab; Partitionen ermöglichen Parallelität.
Reaktionsfähigkeit	Hoch	Der Publisher kehrt zurück, sobald das Ereignis dauerhaft gespeichert ist; langsame Arbeit geschieht später.
Fehlertoleranz	Hoch	Ein Consumer kann minutenlang ausfallen und vom Broker aus aufholen; Ausfälle pflanzen sich nicht zum Publisher zurück fort.
Erweiterbarkeit	Hoch	Neue Consumer abonnieren bestehende Ereignisse, ohne dass sich am Publisher etwas ändert.
Datenkonsistenz	Niedrig	Zustand verbreitet sich mit Verzögerung über Services; Leser sehen eine Zeit lang veraltete Daten.
Testbarkeit und Debugging	Niedrig	Das Verhalten entsteht aus asynchronen Reaktionen; eine Abfolge von Ereignissen über Services hinweg nachzustellen, ist ohne gutes Tracing schwierig.
Sichtbarkeit des Ablaufs	Niedrig (Choreografie) bis mittel (Mediator)	Bei Choreografie verantwortet niemand den End-to-End-Ablauf; ein Mediator macht ihn explizit.
Einfachheit	Niedrig	Broker, Schemas, Wiederholungen, Dead-Letter-Behandlung, Reihenfolge und Duplikate müssen alle entworfen werden.

Wann man sie einsetzt, und häufige Fehler

Ereignisgesteuerte Architektur passt, wenn viele Teile eines Geschäfts auf dieselben Tatsachen reagieren müssen, wenn Arbeit erledigt werden kann, nachdem die Anfrage des Benutzers beantwortet ist, wenn die Last schubweise kommt und geglättet werden muss und wenn mit der Zeit neue Consumer zu erwarten sind. Weniger gut passt sie, wenn ein Benutzer eine sofortige, endgültige Antwort braucht, die von mehreren Services abhängt (obwohl ein synchroner Aufruf für die Antwort plus Ereignisse für die Folgeschritte ein gängiger Hybrid ist), oder wenn starke Konsistenz über Services hinweg erforderlich ist.

ACHTUNG

Häufige Fehler

- **Als Ereignisse getarnte Commands.** `SendConfirmationEmail`, auf einem Topic veröffentlicht, ist ein Command mit versteckter Kopplung; veröffentlichen Sie `OrderPlaced` und lassen Sie den Benachrichtigungsservice entscheiden.
- **Von exactly once ausgehen.** Ein Consumer, der ohne Idempotenzprüfung eine Karte belastet oder eine E-Mail sendet, wird es irgendwann zweimal tun.
- **Von einer globalen Reihenfolge ausgehen.** Ereignisse in verschiedenen Partitionen oder von verschiedenen Produzenten können in beliebiger Reihenfolge eintreffen; gestalten Sie die Reihenfolge pro Entität mit Schlüsseln und Versionsnummern.
- **In die Datenbank schreiben und getrennt veröffentlichen.** Stürzt der Service zwischen beidem ab, geht das Ereignis verloren oder verkündet etwas, das nie geschehen ist. Das Muster Transactional Outbox löst das.
- **Event-Spaghetti.** Dutzende Services, die auf die Ereignisse der anderen reagieren, ohne eine Landkarte der Abläufe. Führen Sie einen Ereigniskatalog, verfolgen Sie Abläufe mit Korrelations-IDs, und setzen Sie einen Mediator ein, wo ein Ablauf einen Verantwortlichen braucht.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Ist `ReserveCourier` ein Ereignis oder ein Command, und woran würden Sie das erkennen?

Ein Command: Es ist eine Anweisung an einen Handler (die Disposition), der sie ablehnen kann. Ein Ereignis wäre in der Vergangenheitsform benannt, etwa `CourierReserved`.

▼ Ein Kafka-Topic hat 6 Partitionen, und eine Consumer Group hat 10 Consumer. Wie viele Consumer sind beschäftigt?

Höchstens 6. Innerhalb einer Gruppe wird jede Partition einem Consumer zugewiesen, sodass 4 Consumer untätig bleiben.

▼ Warum verhindert Kafkas Exactly-once-Unterstützung nicht, dass ein Consumer eine Bestätigungs-E-Mail zweimal sendet?

Seine Transaktionen umfassen das Lesen aus und Schreiben in Kafka-Topics sowie das Committen von Offsets; eine an ein externes System gesendete E-Mail liegt außerhalb dieser Grenze, sodass ein Absturz vor dem Offset-Commit zu erneuter Verarbeitung führt, es sei denn, der Consumer ist idempotent.

▼ **Was ist der Unterschied zwischen Event Notification und Event-Carried State Transfer?**

Notification sendet ein kleines Signal, oft nur eine Kennung, sodass Empfänger die Details nachfragen; State Transfer enthält die Daten, die Empfänger brauchen, sodass sie eine eigene Kopie führen und Aufrufe beim Publisher vermeiden können.

▼ **Probieren Sie es aus: Ein Consumer sieht bei einer Bestellung manchmal OrderCancelled vor OrderPaid. Das Topic hat 12 Partitionen, und die Events haben zufällige Schlüssel. Außerdem verarbeitet er gelegentlich ein Event zweimal. Diagnostizieren Sie beide Probleme, und nennen Sie Abhilfen.**

Reihenfolge gilt nur innerhalb einer Partition, und zufällige Schlüssel verstreuen die Ereignisse einer Bestellung über Partitionen. Verwenden Sie die Bestell-ID als Schlüssel, damit alle Ereignisse einer Bestellung in derselben Partition und in Reihenfolge landen. Duplikate sind bei At-least-once-Zustellung normal, machen Sie den Consumer also idempotent, etwa indem Sie verarbeitete Ereignis-IDs in derselben Transaktion wie seine Aktualisierung speichern. Fügen Sie eine Versionsnummer hinzu, und schicken Sie fehlerhafte Nachrichten in eine überwachte Dead Letter Queue, statt die Partition zu blockieren.

QUELLEN

- Martin Fowler: What do you mean by "Event-Driven"? (2017): https://martinfowler.com/articles/201701-event-driven.html
- Apache Kafka: Introduction (topics, partitions and ordering): https://kafka.apache.org/intro
- Confluent documentation: Kafka message delivery guarantees: https://docs.confluent.io/kafka/design/delivery-semantics.html
- Enterprise Integration Patterns: Idempotent Receiver: https://www.enterpriseintegrationpatterns.com/patterns/messaging/IdempotentReceiver.html
- CloudEvents specification: https://cloudevents.io/

Synchrone und asynchrone Kommunikation: REST, gRPC, GraphQL und Messaging

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.6 APIs und Integration
- 6.1 Warum verteilen
- 6.4 Ereignisgesteuert

Begriffe von früher, die er verwendet: HTTP, HTTP-Statuscode, Teilausfall, Breaking Change, Publish-Subscribe

DIE IDEE IN KÜRZE

Services kommunizieren auf zwei Arten. Ein Telefonanruf ist synchron: Sie bleiben in der Leitung, bis Sie eine Antwort bekommen, und wenn die andere Person nicht da ist, bekommen Sie nichts. Ein Brief ist asynchron: Sie werfen ihn ein und gehen Ihrem Tag nach, und die Antwort kommt später. Telefonanrufe sind einfach, binden aber beide an denselben Moment; Briefe sind flexibler, aber Sie wissen nicht sofort, ob Ihr Anliegen Erfolg hatte. Dazu brauchen beide Seiten eine gemeinsame, schriftliche Vereinbarung darüber, was gefragt werden kann und was zurückkommt, und eine Möglichkeit, diese Vereinbarung zu ändern, ohne diejenigen zu verärgern, die sich darauf verlassen.

WAS SIE WISSEN MÜSSEN

- Synchrone Aufrufe erzeugen zeitliche Kopplung: Aufrufer und Aufgerufener müssen gleichzeitig laufen, sodass sich Latenzen entlang einer Kette addieren und Verfügbarkeiten multiplizieren; asynchrones Messaging beseitigt diese Kopplung, nicht aber die Format- oder semantische Kopplung.
- REST ist Fieldings Satz von Einschränkungen (Client-Server, zustandslos, cachebar, einheitliche Schnittstelle, Schichtensystem, optional Code on Demand); die meisten realen APIs erreichen Stufe 2 von Richardsons Reifegradmodell, nicht Stufe 3 mit Hypermedia.
- RFC 9110 definiert GET, HEAD, OPTIONS, PUT und DELETE als idempotent und POST als weder sicher noch idempotent; Clients sollten nach einem Timeout, 502, 503 oder 504 nur idempotente Anfragen automatisch wiederholen, daher braucht POST einen Idempotenzschlüssel, bevor es sicher wiederholt werden kann.

- gRPC nutzt Protocol Buffers über HTTP/2 mit generiertem Code und Streaming und eignet sich für interne Aufrufe; GraphQL lässt unterschiedliche Clients Felder wählen, braucht aber Batching, Grenzen für Abfragekosten und Autorisierung auf Feldebene.
- Queues stellen jede Nachricht einem von mehreren konkurrierenden Consumern zu; Topics stellen jedem Abonnenten eine Kopie zu.
- Langlebige APIs modellieren fachliche Ressourcen, paginieren mit Cursorn, liefern Problem Details nach RFC 9457, ändern sich nur abwärtskompatibel und versionieren oder nutzen Expand and Contract, wenn sie brechen müssen.
- OpenAPI und AsyncAPI beschreiben HTTP- und Nachrichtenverträge; Consumer-Driven-Contract-Tests wie Pact zeigen Providern, welche Änderungen welche Consumer brechen würden.

Zwei Arten der Kommunikation und die Kopplung, die jede erzeugt

Sobald ein System in getrennt deployte Teile zerlegt ist (siehe Warum verteilen), müssen diese Teile über ein Netzwerk kommunizieren, und die Art, wie sie kommunizieren, prägt das System ebenso stark wie die Art, wie es aufgeteilt ist. Es gibt zwei grundlegende Modi. Bei synchroner Kommunikation sendet der Aufrufer eine Anfrage und wartet auf die Antwort, bevor er weitermacht, wie bei einem Telefonanruf. Bei asynchroner Kommunikation übergibt der Sender eine Nachricht an einen Vermittler, meist einen Message Broker, und macht weiter; der Empfänger holt die Nachricht ab, wenn er bereit ist, wie einen Brief aus dem Briefkasten. Abschnitt 0.6 hat den Unterschied eingeführt; hier wird er als Architekturentscheidung untersucht.

Synchrone Aufrufe erzeugen zeitliche Kopplung: Beide Seiten müssen im selben Moment laufen und erreichbar sein. Der Aufrufer ist daher bestenfalls so verfügbar wie das Produkt der Verfügbarkeiten seiner Abhängigkeiten, und seine Antwortzeit, einschließlich seiner Tail-Latenz, enthält deren Antwortzeiten: die Arithmetik aus Abschnitt 6.1. Ein langsamer Zahlungsservice macht den Bestellservice langsam, und wenn alle Threads des Bestellservice auf ihn warten, antwortet der Bestellservice niemandem mehr. Timeouts, Wiederholungen und Circuit Breaker begrenzen den Schaden; sie werden unter Resilienzmuster behandelt.

Asynchrones Messaging beseitigt die zeitliche Kopplung: Der Broker speichert die Nachricht, sodass der Empfänger langsam, beschäftigt oder kurz ausgefallen sein kann, ohne dass der Sender es bemerkt. Es beseitigt nicht jede Art von Kopplung. Beide Seiten hängen weiterhin vom Format und von der Bedeutung der Nachricht ab (Format- und semantische Kopplung) sowie vom Broker selbst, der zu kritischer Infrastruktur wird. Außerdem erfährt der Sender das Ergebnis nicht mehr sofort, daher muss der Entwurf mit Arbeit umgehen, die später scheitert, mit Nachrichten, die doppelt ankommen, und mit Nachrichten, die in anderer Reihenfolge ankommen. Asynchronität verlagert Komplexität, statt sie zu beseitigen.

Eine nützliche Faustregel: Nutzen Sie einen synchronen Aufruf, wenn der Aufrufer ohne die Antwort wirklich nicht weitermachen kann („Ist diese Karte autorisiert?“), und asynchrones Messaging, wenn der Aufrufer nur wissen muss, dass die Arbeit erledigt werden wird („Beleg senden“, „Restaurant informieren“). Viele gute Entwürfe halten den Pfad, den Nutzer sehen, kurz und synchron und schieben alles andere hinter eine Queue.

Zeitliche Kopplung mit und ohne Broker



In einer synchronen Kette ist der Aufrufer blockiert und hängt von jedem nachgelagerten Service ab; mit einem Broker dazwischen müssen Sender und Empfänger nicht mehr im selben Moment verfügbar sein.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: Telefonanruf oder Brief?

Die schwierigste Idee dieses Abschnitts ist, wovon es Sie befreit und wovon nicht, eine Nachricht abzuschicken, statt auf eine Antwort zu warten. Kapitel 0 hat die beiden Modi eingeführt; verfolgen Sie eine Essensbestellung durch beide.

- Ein Kunde tippt auf „Bestellen“. Der Bestellservice muss wissen, ob die Karte autorisiert ist, bevor er zusagen kann, also ruft er den Zahlungsservice auf und wartet: ein synchroner Aufruf, wie ein Telefonanruf.
- Ist der Zahlungsservice ausgefallen, kann die Bestellung nicht aufgegeben werden: Beide müssen im selben Moment laufen und erreichbar sein. Das ist zeitliche Kopplung.
- Als Nächstes braucht der Kunde einen Beleg per E-Mail. Darauf muss der Bestellservice nicht warten, also hinterlegt er eine ReceiptNeeded-Nachricht bei einem Message Broker, einer Art Briefkasten, und antwortet dem Kunden sofort: asynchron, wie ein Brief.
- Angenommen, der E-Mail-Service fällt zehn Minuten lang aus. Der Broker behält die Nachricht, und der Beleg kommt einfach später an. Hätte der Bestellservice den E-Mail-Service „angerufen“ und gewartet, wäre jede Bestellung in diesen zehn Minuten gescheitert oder hängen geblieben.
- Der Brief befreit die beiden nicht vollständig. Benennt der Bestellservice ein Feld in der Nachricht um, etwa von email in emailAddress, kann der E-Mail-Service sie nicht mehr lesen: Beide Seiten verlassen sich weiterhin auf denselben vereinbarten Aufbau und dieselbe Bedeutung des Gesendeten. Und der Bestellservice erfährt nicht mehr sofort, ob die E-Mail verschickt wurde.
- **Die Lehre:** Rufen Sie nur dann auf und warten Sie, wenn Sie ohne die Antwort nicht weitermachen können; schicken Sie eine Nachricht, wenn Sie nur wissen müssen, dass die Arbeit erledigt werden wird. In beiden Fällen ist das Format des Gesendeten ein Versprechen, das mit Sorgfalt geändert werden muss, wie das Ende dieses Abschnitts erklärt.

REST, wie Fielding es definiert hat

REST (Representational State Transfer) wurde von Roy Fielding in seiner Dissertation von 2000 als Architekturstil für große, vernetzte Hypermedia-Systeme definiert: das Web selbst. Fielding leitete ihn als Satz von Einschränkungen her, von denen jede ein Qualitätsmerkmal einbringt. **Client-Server** trennt die Benutzeroberfläche von der Datenspeicherung, sodass sich beide unabhängig weiterentwickeln. **Zustandslose** Interaktion bedeutet, dass jede Anfrage alles enthält, was zu ihrem Verständnis nötig ist, sodass jede Serverinstanz sie bearbeiten kann, was Skalierbarkeit und Wiederherstellung hilft. **Cachebare** Antworten geben an, ob sie

wiederverwendet werden dürfen, was Latenz und Last senkt. Eine **einheitliche Schnittstelle** (Ressourcen, die durch URIs identifiziert und über Repräsentationen manipuliert werden, selbstbeschreibende Nachrichten und Hypermedia als Motor des Anwendungszustands) lässt generische Vermittler wie Caches und Proxys den Datenverkehr verstehen. Ein **Schichtensystem** lässt Proxys, Gateways und Load Balancer unsichtbar im Pfad sitzen. **Code on Demand**, etwa an einen Browser gesendetes JavaScript, ist die einzige optionale Einschränkung.

Die meisten als RESTful bezeichneten APIs erfüllen nur einige davon. Leonard Richardsons Reifegradmodell, 2008 vorgestellt und 2010 von Martin Fowler bekannt gemacht, beschreibt die Stufen. Auf **Stufe 0** ist HTTP nur ein Tunnel: Jeder Aufruf ist ein POST an einen einzigen Endpoint, die Operation steht im Body, wie bei SOAP. Auf **Stufe 1** hat die API getrennte Ressourcen (`/orders/1042` , `/restaurants/7`). Auf **Stufe 2** nutzt sie HTTP-Methoden und Statuscodes so, wie HTTP sie definiert. Auf **Stufe 3** enthalten Antworten Hypermedia Controls: Links, die dem Client sagen, was er als Nächstes tun kann (eine aufgegebene Bestellung enthält einen Link zum Stornieren, bis eine Stornierung nicht mehr möglich ist).

Über Stufe 3 sind sich Fachleute ernsthaft uneinig. Fielding besteht darauf, dass eine API ohne Hypermedia nicht REST ist. Die meisten öffentlichen APIs bleiben auf Stufe 2, weil ihre Clients anhand der Dokumentation geschrieben werden, statt zur Laufzeit entdeckt zu werden, und der Nutzen des Link-Folgens den zusätzlichen Entwurfsaufwand selten aufwiegt. Eine pragmatische Lesart ist, dass Stufe 2, konsequent umgesetzt, den größten Teil des Werts bringt; Hypermedia lohnt sich, wenn der Server viele unabhängig geschriebene Clients durch einen sich ändernden Ablauf steuern muss.

HTTP-Semantik, die für die Architektur zählt

Die Semantik von HTTP ist in RFC 9110 (2022) definiert. Zwei Eigenschaften von Methoden zählen für verteilte Entwürfe mehr als alle anderen. Eine **sichere** Methode liest nur: GET, HEAD und OPTIONS ändern nichts, was der Client ändern wollte. Eine idempotente Methode darf den Zustand ändern, aber sie zu wiederholen hat dieselbe beabsichtigte Wirkung wie ein einmaliges Senden: PUT (diese Ressource durch diese Repräsentation ersetzen) und DELETE sind idempotent, ebenso alle sicheren Methoden. POST ist keines von beiden, und PATCH ist im Allgemeinen nicht idempotent. Die Eigenschaft zählt, weil Netzwerke zwischen der Verarbeitung einer Anfrage und dem Eintreffen der Antwort ausfallen. RFC 9110 sagt, dass idempotente Anfragen nach einem solchen Ausfall automatisch wiederholt werden können und dass Clients nicht idempotente Anfragen nicht automatisch wiederholen sollten, es sei denn, sie wissen, dass die Anfrage tatsächlich gefahrlos wiederholt werden kann.

Statuscodes sind Teil des Vertrags, weil Client-Code anhand von ihnen verzweigt; Abschnitt 0.6 hat die gängigen eingeführt. Erfolgscodes sollten sagen, was passiert ist: 201 Created (mit einem `Location`-Header), wenn eine Ressource angelegt wird, 202 Accepted, wenn die Arbeit nur in eine Queue gestellt wurde, 204 No Content für einen Erfolg ohne Body. Bei Konflikten bedeutet 409, dass die Anfrage mit dem aktuellen Zustand der Ressource kollidiert, 412, dass eine Vorbedingung nicht erfüllt war, und 422, dass eine wohlgeformte Anfrage semantisch ungültig ist. Die architektonische Regel betrifft Wiederholungen: Eine 4xx-Antwort wird erneut scheitern, wenn die Anfrage unverändert gesendet wird (429 Too Many Requests ist die Ausnahme: so lange warten wie angegeben, dann wiederholen), während ein Timeout, 502, 503 oder 504 später gelingen kann. Clients sollten nur diese wiederholen, mit Backoff und Jitter, und nur, wenn die Anfrage idempotent ist oder einen Idempotenzschlüssel trägt.

Zwei weitere HTTP-Features lösen verteilte Probleme günstig. **Bedingte Anfragen** ermöglichen optimistische Nebenläufigkeitskontrolle: Ein Client liest eine Bestellung mit ihrem Versionskennzeichen (`ETag`) und sendet beim Aktualisieren `If-Match` mit diesem Kennzeichen; hat jemand anderes die Bestellung inzwischen geändert, antwortet der Server mit 412, und nichts wird überschrieben. **Caching-Header** (`Cache-Control`, `ETag`) lassen Browser, CDNs und Proxys Antworten wiederverwenden, was oft die günstigste verfügbare Performance-Taktik ist.

HTTP-Methoden: sicher, idempotent oder keines von beiden (RFC 9110; PATCH aus RFC 5789)

Methode	Typische Verwendung	Sicher?	Idempotent?	Blind wiederholbar?
GET, HEAD	Eine Ressource lesen	Ja	Ja	Ja
PUT	An einer bekannten URI anlegen oder ersetzen	Nein	Ja	Ja
DELETE	Eine Ressource entfernen	Nein	Ja	Ja (ein zweiter Aufruf kann 404 liefern, aber die Wirkung ist dieselbe)
POST	In einer Collection anlegen oder eine Aktion ausführen	Nein	Nein	Nur mit einem Idempotenzschlüssel oder einer anderen Duplikatprüfung
PATCH	Teilaktualisierung	Nein	Im Allgemeinen nicht	Nur wenn der Patch idempotent geschrieben ist oder an eine Version gebunden ist

gRPC: zuerst der Vertrag, binär auf der Leitung

gRPC ist ein Framework für Remote Procedure Calls, das Google 2015 als Nachfolger von Stubby als Open Source veröffentlichte, dem RPC-System, das es über ein Jahrzehnt intern genutzt hatte; Version 1.0 folgte 2016. Der Vertrag wird in einer `.proto`-Datei mit Protocol Buffers geschrieben, einem kompakten binären Serialisierungsformat, und Codegeneratoren erzeugen typisierte Client-Stubs und Server-Skelette in vielen Sprachen. Aufrufe laufen über HTTP/2, das viele gleichzeitige Anfragen über eine Verbindung multiplext.

gRPC bietet vier Arten von Aufrufen: **unär** (eine Anfrage, eine Antwort), **Server-Streaming** (eine Anfrage, ein Strom von Antworten, etwa Live-Positionen der Fahrer), **Client-Streaming** (ein Strom von Anfragen, eine Antwort, etwa das Hochladen von Telemetrie) und **bidirektionales Streaming**. Clients setzen eine **Deadline**; ist sie abgelaufen, endet der Aufruf mit `DEADLINE_EXCEEDED`, und Deadlines lassen sich entlang einer Aufrufkette weitergeben, sodass die Arbeit überall stoppt, sobald niemand mehr auf die Antwort wartet.

Seine Stärken sind Effizienz, starke Typisierung und generierter Code, was es für interne Aufrufe zwischen Services in polyglotten Organisationen beliebt macht. Seine Kosten sind das Spiegelbild davon. Binäre Payloads sind in Logs und mit gewöhnlichen Werkzeugen schwer zu lesen. Browser können gRPC nicht direkt aufrufen, daher brauchen öffentliche und webseitige APIs meist einen Proxy oder eine REST- oder GraphQL-Fassade. HTTP-Caching greift nicht auf die übliche Weise. Kompatibilität hängt von Disziplin bei den Feldnummern ab: Die Nummer eines Felds, nicht sein Name, identifiziert es auf der Leitung, daher dürfen Nummern nie wiederverwendet werden, und entfernte Felder sollten als reserved markiert werden.

```
syntax = "proto3";

service Dispatch {
  rpc AssignCourier (AssignRequest) returns (Assignment);
  rpc TrackCourier (TrackRequest) returns (stream Position);
}

message AssignRequest {
  string order_id = 1;
  string restaurant_id = 2;
  reserved 3;            // field removed in v2: never reuse 3
}

message Position {
  double lat = 1;
  double lng = 2;
  int64 recorded_at_ms = 3;
}
```

Ein gRPC-Vertrag für den Dispositionsservice der Essenslieferplattform mit einem unären und einem Server-Streaming-Aufruf (Protocol-Buffers-Syntax).

GraphQL: Der Client bestimmt die Form

GraphQL ist eine Abfragesprache und Laufzeitumgebung für APIs, die Facebook ab 2012 intern für seine mobilen Apps entwickelte und 2015 mit einem Spezifikationsentwurf und einer Referenzimplementierung öffentlich freigab; 2018 ging sie an die herstellerneutrale GraphQL Foundation. Der Server veröffentlicht ein typisiertes **Schema**; Clients senden Abfragen an einen einzigen Endpoint, die genau die gewünschten Felder nennen, möglicherweise über verknüpfte Objekte hinweg, und erhalten eine JSON-Antwort derselben Form. Mutations ändern Daten, und Subscriptions schicken Aktualisierungen aktiv an den Client.

Ihr Hauptnutzen liegt bei unterschiedlichen Clients. Eine mobile App der Essenslieferplattform mit langsamer Verbindung kann in einem Roundtrip den Status einer Bestellung, den Namen des Restaurants und den Vornamen des Kuriers abfragen, ohne die zwanzig anderen Felder zu erhalten, die das Web-Dashboard braucht. Das vermeidet sowohl Over-Fetching als auch die Kette von Anfragen, die eine REST-API Ressource für Ressource erfordern kann, und kann ein eigenes Backend für jedes Frontend ersetzen. Das Schema dokumentiert sich außerdem selbst und lässt Werkzeuge Abfragen prüfen, bevor sie laufen.

Die Abwägungen sind real und werden größtenteils vom Serverteam getragen. Jedes Feld wird von einem **Resolver** befüllt, daher erzeugt eine naive Implementierung, die das Restaurant jeder Bestellung einzeln abruft, das N+1-Abfrageproblem: eine Abfrage für fünfzig Bestellungen, dann fünfzig weitere für ihre Restaurants. Die übliche Abhilfe sind Batching und Caching pro Anfrage, wie es die Bibliothek DataLoader bekannt gemacht hat. Weil Clients

beliebig tiefe Abfragen zusammensetzen können, brauchen Server Grenzen für Abfragetiefe und -kosten, Timeouts und oft eine Allow-List genehmigter (persistierter) Abfragen, um versehentliche oder böswillige Überlastung zu verhindern. HTTP-Caching ist schwächer, wenn alles ein POST an eine URL ist. Autorisierung muss pro Feld oder pro Objekt durchgesetzt werden statt pro URL. Und Fehler werden typischerweise in einem `errors`-Array innerhalb einer erfolgreichen HTTP-Antwort gemeldet, was ein Monitoring auf Basis von Statuscodes übersieht.

Messaging: Queues und Topics

Messaging verlagert die Kommunikation in einen Broker. Wie ereignisgesteuerte Architektur erklärt, stellt eine Queue jede Nachricht einem von mehreren konkurrierenden Consumern zu, der Standardweg, einen Worker-Pool zu skalieren, während ein Topic jedem Abonnenten eine Kopie zustellt, sodass die Services für Restaurant, Disposition und Benachrichtigungen jeweils auf `OrderPlaced` reagieren können, ohne dass der Bestellservice von ihnen weiß. Dieser Abschnitt behandelt auch Zustellgarantien, Reihenfolge, Duplikate und Dead Letter Queues; Queues und Backpressure behandelt, wie Queues Lastspitzen abfangen.

Messaging kann auch Anfrage und Antwort transportieren: Der Anfragende sendet ein Command mit einer **Korrelations-ID** und einer Antwortadresse und ordnet die spätere Antwort seiner Anfrage zu. So bleibt die zeitliche Entkopplung eines Brokers erhalten, und man bekommt trotzdem eine Antwort, um den Preis von mehr beweglichen Teilen.

REST, gRPC, GraphQL und Messaging im Vergleich

Aspekt	REST über HTTP	gRPC	GraphQL	Messaging (Queues und Topics)
Interaktion	Synchron, Anfrage-Antwort	Synchron; unär oder Streaming	Synchrone Queries und Mutations; Subscriptions pushen	Asynchron; Fire-and-Forget oder Anfrage-Antwort über Korrelations-IDs
Vertrag	OpenAPI-Dokument (optional, aber üblich)	.proto-Dateien (erforderlich)	Schema (erforderlich)	AsyncAPI-Dokument und Nachrichtenschemas (optional, aber ratsam)
Payload	Meist JSON-Text	Protocol Buffers, binär	JSON	Beliebig: JSON, Avro, Protocol Buffers
Zeitliche Kopplung	Ja	Ja	Ja	Nein: Der Broker puffert
Caching	Ausgezeichnet: Standard-HTTP-Caches und CDNs	Wenig eingebaut	Schwieriger: braucht persistierte Abfragen oder Client-Caches	Nicht anwendbar
Browser-Unterstützung	Nativ	Braucht einen Proxy (gRPC-Web)	Nativ	Nicht direkt; über ein Backend oder WebSockets
Am besten geeignet für	Öffentliche und Partner-APIs; ressourcenorientiertes CRUD	Interne Service-Aufrufe mit hohem Volumen und niedriger Latenz; Streaming	Viele Client-Typen mit unterschiedlichem Datenbedarf; Bündeln mehrerer Services	Arbeit, die später erledigt werden kann; Fan-out an viele Consumer; Glätten der Last
Hauptfallen	Gesprächige (chatty) APIs; uneinheitliche Konventionen	Undurchsichtig beim Debugging; Disziplin bei Feldnummern	N+1-Resolver; unbegrenzte Abfragekosten; Autorisierung auf Feldebene	Duplikate, Reihenfolge, unsichtbare Abläufe, Betrieb des Brokers

Die Messaging-Landschaft: Queue oder Log wählen

Die erste Entscheidung fällt zwischen Queue und Log. Eine **Queue** verteilt Arbeit, und eine Nachricht ist nach der Verarbeitung weg; das passt zu Aufgaben wie dem Senden einer Quittung, wo Wiederholungen pro Nachricht und Dead Lettering zählen. Ein **Log** bewahrt Nachrichten für spätere Leser auf; das passt zu Fakten, die mehrere Consumer brauchen oder die erneut abgespielt werden müssen, etwa Bestellereignisse, die ein Lesemodell neu aufbauen. Abschnitt 6.4 erklärt Kafka, das bekannteste Log. Die Grenze ist verschwommen: Kafka 4.2 erklärte seine queue-ähnlichen Share Groups für produktionsreif, und RabbitMQ ergänzte Append-only-Streams.

Vier weitere Namen tauchen oft auf. **RabbitMQ** ist ein Broker, den Sie selbst betreiben oder als gehosteten Dienst einkaufen; seine Queues eignen sich zur Arbeitsverteilung, und seine Streams ergänzen Replay und großes Fan-out. **Amazon SQS und SNS** sind verwaltete Dienste: Consumer pollen eine SQS-Queue, und ein SNS-Topic schiebt eine Kopie an jede abonnierte Queue, sodass ein Ereignis die Queues für Restaurant, Disposition und Benachrichtigungen erreicht, die jede in ihrem eigenen Tempo abgearbeitet werden. **Google Cloud Pub/Sub** ist verwaltetes Publish-Subscribe mit At-least-once-Zustellung. **NATS** ist leichtgewichtig: Core NATS arbeitet at-most-once, ein nicht verbundener Subscriber sieht die Nachricht also nie, während die JetStream-Schicht Speicherung, Replay und Bestätigungen ergänzt. Limits und Funktionen ändern sich, prüfen Sie also jeweils die aktuelle Version.

Competing Consumers und strikte Reihenfolge ziehen in entgegengesetzte Richtungen. Selbst wenn ein Broker Nachrichten in der richtigen Reihenfolge ausgibt, werden Worker in beliebiger Reihenfolge fertig: Ein langsamer Worker mit `OrderPaid` kann nach einem schnellen mit `OrderCancelled` fertig werden. Erneute Zustellung verschärft das; die RabbitMQ-Dokumentation warnt, dass bei mehreren aktiven Consumern „jede erneute Zustellung die Reihenfolge ändern kann“. Die übliche Antwort ist, die Gesamtreihenfolge aufzugeben und die Reihenfolge pro Schlüssel zu wahren, mit höchstens einem Worker pro Schlüssel zugleich. Kafka tut das mit Partitionen nach Schlüssel und SQS-FIFO-Queues mit Nachrichtengruppen: Die Reihenfolge gilt innerhalb einer Gruppe, und Gruppen laufen parallel. Standard-SQS-Queues bieten nur Best-Effort-Reihenfolge. Pub/Sub nutzt Ordering Keys, die im Abonnement aktiviert werden. RabbitMQs Single Active Consumer wahrt die Reihenfolge, indem nur ein Consumer aus der Queue empfängt, was dort die Parallelität aufgibt.

Reihenfolge pro Schlüssel hat ihren Preis. Eine langsame oder fehlschlagende Nachricht hält jede spätere Nachricht zu diesem Schlüssel auf, und ein sehr aktiver Schlüssel baut einen Rückstand auf; die Pub/Sub-Dokumentation sagt zudem, dass die erneute Zustellung einer Nachricht auch die späteren zu diesem Schlüssel erneut zustellt, selbst bereits bestätigte. Geordnete Zustellung beseitigt auch keine Duplikate, Consumer müssen also weiterhin

idempotent sein. Wo sich eine veraltete Aktualisierung erkennen lässt, besteht eine Alternative darin, in jedem Ereignis eine Versionsnummer mitzuführen und Consumer alles ignorieren zu lassen, was älter ist als das bereits Angewendete.

APIs entwerfen, die Bestand haben: Ressourcen, Paginierung und Filterung

Modellieren Sie Ressourcen nach der Fachdomäne, nicht nach der Datenbank. Legen Sie die Substantive offen, in denen Ihre Consumer denken (`/orders`, `/orders/1042/items`, `/restaurants/7/menu`), nicht Tabellen oder interne Services. Aktionen, die nicht zu Anlegen, Lesen, Aktualisieren und Löschen passen, lassen sich meist selbst als Ressourcen modellieren: `POST /orders/1042/cancellation` legt eine Stornierung an, die einen Grund tragen und mit 409 abgelehnt werden kann, wenn das Essen schon unterwegs ist. So bleibt die einheitliche Schnittstelle erhalten, und der Geschäftsprozess wird trotzdem ehrlich ausgedrückt.

Paginieren Sie jede Collection, denn Collections wachsen. **Offset-Paginierung** (`?offset=40&limit=20`) ist einfach und erlaubt den Sprung auf Seite 9, wird aber auf tiefen Seiten langsam, weil die Datenbank alle früheren Zeilen überspringen muss, und sie zeigt Duplikate oder lässt Einträge aus, wenn Zeilen eingefügt werden, während jemand blättert. Cursor-Paginierung liefert ein undurchsichtiges Token, das direkt hinter den letzten Eintrag zeigt (`?after=eyJpZCI6MTA0Mn0&limit=20`); sie bleibt bei Einfügungen schnell und stabil, um den Preis, dass kein wahlfreier Zugriff möglich ist. Für Feeds und große oder sich ändernde Daten sind Cursor der sicherere Standard. Begrenzen Sie die Seitengröße immer auf dem Server.

Filtern und Sortieren sollten vorhersehbare Query-Parameter nutzen (`?status=delivered&placed_after=2026-10-01&sort=-placed_at`) und durch Indizes gestützt sein; eine API, die Clients nach jedem beliebigen Feld filtern lässt, lädt zu Abfragen ein, die die Datenbank nicht schnell beantworten kann. Unterstützen Sie nur die Filter, die Consumer brauchen, und dokumentieren Sie sie im Vertrag.

Lang laufende Arbeit: 202, Polling und zuverlässige Webhooks

Manche Anfragen dauern zu lange, um offen gehalten zu werden, etwa der Import der gesamten Speisekarte einer Restaurantkette. Die API prüft die Anfrage und antwortet mit `202 Accepted` und einem `Location`-Header, der eine Statusressource nennt, und vielleicht `Retry-After`. RFC 9110 nennt 202 „absichtlich unverbindlich“: Die Arbeit kann später noch abgelehnt werden. Der Client pollt, erhält 200 mit dem Zustand und dann 303 zum Ergebnis. Um Polling zu vermeiden, kann der Server stattdessen einen **Webhook** aufrufen. Ein Sender sollte mit Backoff und Jitter wiederholen, über Wiederholungen hinweg eine Ereignis-ID beibehalten und jede Zustellung mit einem Zeitstempel signieren. Der Empfänger sollte die

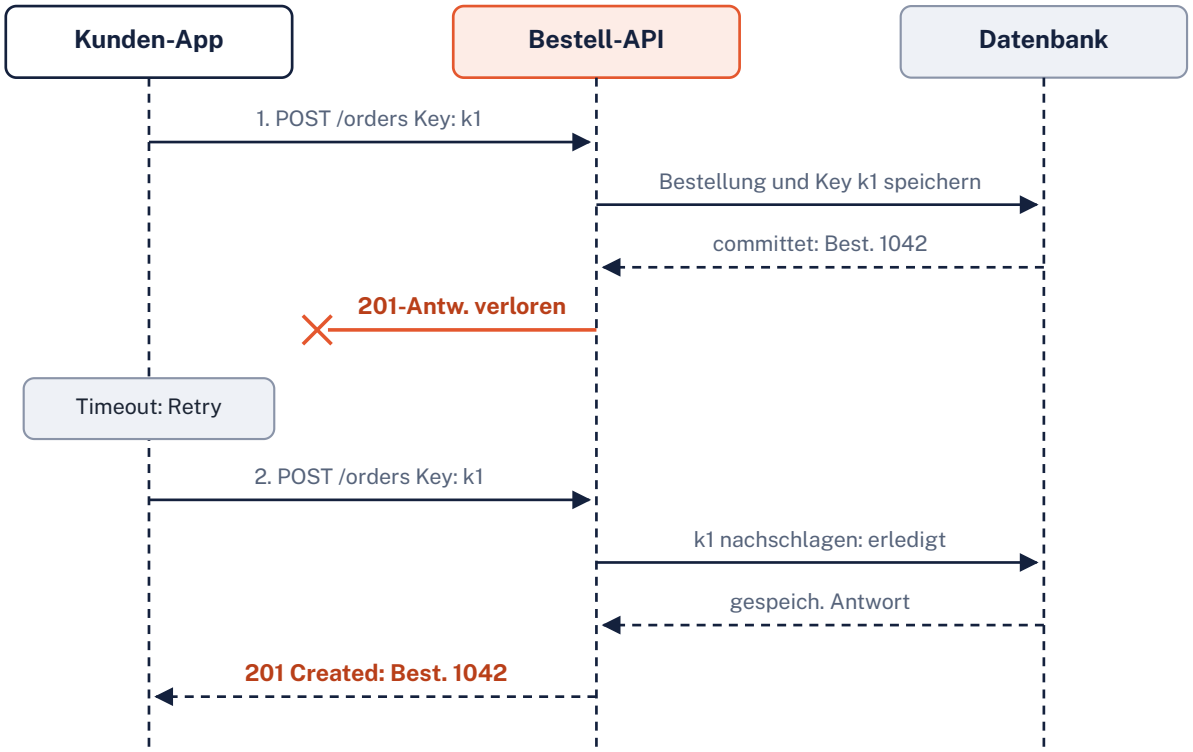
Signatur prüfen (Abschnitt 8.4), schnell mit 2xx antworten, die Arbeit in eine Queue stellen und Duplikate ignorieren. Stripe zum Beispiel garantiert keine Reihenfolge. Behalten Sie den Status-Endpoint als Weg, verpasste Zustellungen nachzuholen.

Idempotenzschlüssel: POST sicher wiederholbar machen

Ein Teilausfall (Abschnitt 6.1) lässt einen Client, der keine Antwort bekommen hat, im Unklaren darüber, ob seine Anfrage verarbeitet wurde. Ein GET zu wiederholen ist harmlos; `POST /orders` zu wiederholen kann eine zweite Bestellung anlegen und den Kunden doppelt belasten. Ein Idempotenzschlüssel macht aus der unsicheren Wiederholung eine sichere. Der Client erzeugt für jede logische Operation einen eindeutigen Wert (typischerweise eine zufällige UUID) und sendet ihn in einem Header wie `Idempotency-Key`. Der Server speichert den Schlüssel zusammen mit dem Ergebnis der ersten Anfrage; jede spätere Anfrage mit demselben Schlüssel liefert das gespeicherte Ergebnis, statt die Arbeit erneut auszuführen.

Zahlungsanbieter haben die Technik bekannt gemacht, und die IETF-Arbeitsgruppe HTTP APIs hat einen Standard-Header `Idempotency-Key` entworfen. Zum Zeitpunkt der Erstellung ist er nicht als RFC veröffentlicht (der jüngste Entwurf vom Oktober 2025 ist abgelaufen), aber seine Regeln spiegeln die gängige Praxis: Ein Schlüssel, der mit einem anderen Request-Body wiederverwendet wird, sollte mit 422 abgelehnt werden, eine Wiederholung, die eintrifft, während die erste Anfrage noch läuft, sollte 409 Conflict erhalten, damit nie zwei Kopien gleichzeitig laufen, und Schlüssel gelten pro Client und verfallen nach einer veröffentlichten Frist, oft 24 Stunden. Das ausgearbeitete Beispiel wendet jede Regel an.

Eine wiederholte Bestellung mit Idempotenzschlüssel



Der erste Versuch gelingt, aber seine Antwort geht verloren; die Wiederholung trägt denselben Schlüssel, daher liefert die API das gespeicherte Ergebnis, statt Bestellung 1043 anzulegen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: der Doppeltipp beim Bezahlvorgang

Ein Kunde mit schwacher Mobilfunkverbindung tippt auf „Bestellen“. Die App erzeugt den Schlüssel `7f3c…` und sendet `POST /v1/orders`. Der Bestellservice speichert den Schlüssel zusammen mit einem Hash des Request-Bodys und später der Antwort.

Versuch 1	Bestellung 1042 wird angelegt und die Zahlung autorisiert; Schlüssel, Body-Hash und Antwort 201 werden in derselben Datenbanktransaktion wie die Bestellung gespeichert. Die Antwort geht verloren, als das Smartphone von WLAN auf mobile Daten wechselt.
Versuch 2 (automatische Wiederholung, derselbe Schlüssel)	Der Service findet den Schlüssel mit dem Status *abgeschlossen*, prüft, ob der Body-Hash übereinstimmt, und liefert die gespeicherte 201 für Bestellung 1042. Keine zweite Bestellung, keine zweite Belastung.
Versuch 3 (der Kunde tippt erneut, während Versuch 2 noch unterwegs ist)	Die App verwendet für denselben Warenkorb denselben Schlüssel erneut. Würde die erste Anfrage noch verarbeitet, antwortete der Service mit 409, und die App würde warten und es erneut versuchen.
Ein Bug sendet denselben Schlüssel mit einem anderen Warenkorb	Die Body-Hashes unterscheiden sich: Der Service antwortet mit 422, statt stillschweigend die falsche Bestellung zu liefern.
Nach 24 Stunden	Der Schlüssel verfällt und wird gelöscht; die App wiederholt diesen alten Warenkorb nie.

Der Schlüssel muss einmal pro logischer Operation erzeugt werden (pro Absenden eines Warenkorbs), nicht pro HTTP-Versuch; ein neuer Schlüssel bei jeder Wiederholung verfehlt den Zweck.

Den Schlüssel in derselben Transaktion wie die Bestellung zu speichern, macht die Prüfung zuverlässig: Ein Absturz kann keine Bestellung ohne ihren Schlüssel und keinen Schlüssel ohne seine Bestellung hinterlassen.

Dieselbe Idee, auf Nachrichten statt auf HTTP-Anfragen angewandt, ist der idempotente Consumer der ereignisgesteuerten Architektur.

```
def create_order(request, db):
    key = request.headers.get("Idempotency-Key")
    if not key:
        return problem(400, "missing-idempotency-key")
    body_hash = sha256(request.body)
    with db.transaction():
        row = db.find_idempotency(request.client_id, key)
        if row and row.body_hash ≠ body_hash:
            return problem(422, "idempotency-key-reused")
        if row and row.status == "in_progress":
            return problem(409, "request-in-progress")
        if row:
            return row.saved_response          # replay, do nothing new
        db.insert_idempotency(request.client_id, key, body_hash, "in_progress")
        order = db.insert_order(request.json)
        response = created(order)
        db.complete_idempotency(request.client_id, key, response)
    return response
```

Serverseitige Behandlung eines Idempotenzschlüssels (vereinfacht: eine Datenbanktransaktion pro Anfrage; der Unique Constraint auf dem Schlüssel verhindert, dass zwei gleichzeitige Anfragen beide laufen).

Fehler, die Maschinen lesen können: RFC 9457

Clients brauchen Fehler, auf die sie programmatisch reagieren können, und Menschen brauchen Fehler, die sie verstehen. Statt dass jedes Team ein eigenes Fehlerformat erfindet, nutzen Sie Problem Details for HTTP APIs, RFC 9457 (Juli 2023), das RFC 7807 (2016) abgelöst hat. Ein Problem-Details-Objekt hat den Medientyp `application/problem+json` und fünf Standardfelder: `type` (eine URI, die die Art des Problems identifiziert und auf Dokumentation verweisen kann), `title` (eine kurze, menschenlesbare Zusammenfassung dieser Art), `status` (der HTTP-Statuscode), `detail` (eine Erklärung dieses Vorkommens) und `instance` (eine URI, die dieses Vorkommen identifiziert). Erweiterungsfelder können maschinenlesbare Einzelheiten tragen, etwa welche Felder die Validierung nicht bestanden haben.

Zwei Regeln halten Fehlerantworten sicher. Client-Code sollte anhand von `type` und `status` verzweigen, nie anhand des Wortlauts von `title` oder `detail`, der übersetzt oder verbessert werden kann. Und Fehler dürfen keine Interna preisgeben: Stacktraces, SQL-Fragmente und interne Hostnamen helfen Angreifern mehr als Aufrufern. Protokollieren Sie die Einzelheiten mit einer Korrelations-ID und liefern Sie stattdessen die ID zurück.

```
{
  "type": "https://api.fooddelivery.example/problems/outside-delivery-area",
  "title": "Address is outside the delivery area",
  "status": 422,
  "detail": "Restaurant 7 delivers within 4 km; the address is 6.2 km away.",
  "instance": "/v1/orders/attempts/9b1e",
  "max_distance_km": 4,
  "actual_distance_km": 6.2
}
```

Eine Problem-Details-Antwort (HTTP 422, Content-Type: application/problem+json) für eine Bestellung außerhalb des Liefergebiets.

Eine API weiterentwickeln, ohne ihre Consumer zu brechen

Eine API mit Consumern ist ein Versprechen, und Hyrums Gesetz (Abschnitt 0.6) warnt, dass jemand von jedem beobachtbaren Verhalten abhängt. Änderungen folgen daher einem Prinzip: Nehmen Sie abwärtskompatible Änderungen frei vor und vermeiden Sie brechende. Consumer helfen, indem sie **Tolerant Reader** sind (ein Begriff, den Martin Fowler verwendet): Sie ignorieren Felder, die sie nicht verstehen, und lesen nur, was sie brauchen, sodass Ergänzungen sie nie brechen. Die Tabelle unten fasst die üblichen Regeln zusammen; dieselbe Logik, angewandt auf gespeicherte Daten und Event-Schemas, findet sich unter Schemaevolution.

Ist eine brechende Änderung unvermeidbar, versionieren Sie. Die gängigen Strategien sind eine Version im URL-Pfad (`/v2/orders`: explizit, leicht zu routen und zu cachen, verleitet aber zu großen, seltenen Sprüngen), ein eigener Header oder ein Parameter im Medientyp (`Accept: application/vnd.fooddelivery.v2+json`: hält URLs stabil, ist aber weniger sichtbar und schwerer im Browser zu testen) und datierte Versionen, die pro Client-Konto festgelegt werden und die einige große API-Anbieter nutzen, um kleine brechende Änderungen fortlaufend zu veröffentlichen, während jeder Client aktualisiert, wann er will. Interne APIs zwischen Services derselben Organisation vermeiden Versionen oft ganz durch den Ansatz **Expand and Contract**: das neue Feld oder den neuen Endpoint hinzufügen, jeden Consumer migrieren, dann das alte entfernen. Unabhängig von der Strategie: Kündigen Sie Abkündigungen im Voraus an; die Response-Header `Deprecation` (RFC 9745, 2025) und `Sunset` (RFC 8594, 2019) lassen einen Server Clients in maschinenlesbarer Form mitteilen, dass ein Endpoint abgekündigt ist und wann er nicht mehr funktionieren wird.

Abwärtskompatible und brechende Änderungen an einer API

Änderung	Kompatibel?	Warum
Ein optionales Anfragefeld oder einen Query-Parameter hinzufügen	Ja	Bestehende Clients senden es nicht; der Server muss einen sinnvollen Standardwert anwenden.
Ein Feld zu einer Antwort hinzufügen	Meistens	Tolerant Reader ignorieren es; strikte Clients, die unbekannte Felder ablehnen, brechen, daher sollten Sie dokumentieren, dass Clients Ergänzungen tolerieren müssen.
Einen neuen Endpoint oder eine neue Ressource hinzufügen	Ja	Noch ruft ihn niemand auf.
Einer Aufzählung in einer Antwort einen Wert hinzufügen	Riskant	Clients mit erschöpfenden switch-Anweisungen können am unbekannten Wert scheitern; dokumentieren Sie, dass neue Werte auftreten können.
Ein optionales Anfragefeld zur Pflicht machen	Nein	Bestehende Clients, die es weglassen, scheitern ab sofort mit 400.
Ein Feld entfernen oder umbenennen oder seinen Typ oder seine Einheit ändern	Nein	Clients, die das alte Feld lesen, bekommen nichts oder das Falsche.
Die Bedeutung eines Felds oder Statuscodes ändern	Nein	Der gefährlichste Bruch: Nichts scheitert sichtbar, aber das Verhalten ist falsch.
Die Validierung verschärfen (kürzere Maximallänge, strengeres Format)	Nein	Anfragen, die früher gelangen, scheitern jetzt.

Verträge beschreiben: OpenAPI und AsyncAPI

Ein in einem maschinenlesbaren Format geschriebener Vertrag kann Dokumentation, Client-Generierung, Server-Stubs, Mock-Server, Linting und Tests antreiben. OpenAPI ist der Standard für HTTP-APIs. Es begann als Swagger-Spezifikation, die SmartBear 2015 der neu gegründeten OpenAPI Initiative unter dem Dach der Linux Foundation übergab und die 2016 umbenannt wurde; Version 3.1 (2021) glich ihre Schemas an JSON Schema an, und Version 3.2

folgte im September 2025. AsyncAPI wendet dieselbe Idee auf nachrichtengesteuerte APIs an: Es beschreibt Kanäle (Queues und Topics), die Operationen, die darauf senden oder empfangen, und die Nachrichtenschemas; Version 3.0 erschien im Dezember 2023.

Teams wählen zwischen **Design-First** (den Vertrag schreiben und prüfen, dann implementieren) und **Code-First** (den Vertrag aus annotiertem Code generieren). Design-First macht die API zu einer bewussten Entscheidung, die Consumer prüfen können, bevor etwas gebaut wird, und eignet sich für öffentliche und teamübergreifende APIs; Code-First hält Vertrag und Implementierung mit weniger Aufwand im Gleichschritt und eignet sich für interne APIs, die sich schnell ändern. In beiden Fällen gilt: Halten Sie den Vertrag in der Versionsverwaltung, linten Sie ihn in der Pipeline und führen Sie eine Prüfung auf Breaking Changes aus, die jede vorgeschlagene Version mit der veröffentlichten vergleicht.

Contract Testing und Consumer-Driven Contracts

End-to-End-Tests, die alle Services zusammen deployen, sind langsam, unzuverlässig und werden mit jedem weiteren Service langsamer, doch Unit-Tests können nicht erkennen, dass ein Provider ein Feld geändert hat, auf das sich ein Consumer verlässt. Contract Testing schließt die Lücke: Jede Seite wird isoliert gegen eine gemeinsame Beschreibung ihrer Interaktionen getestet. Bei Consumer-Driven Contracts, einem Muster, das Ian Robinson 2006 beschrieb, legt jeder Consumer die Teile der Provider-API fest, die er tatsächlich nutzt, und der Provider führt die Erwartungen aller seiner Consumer als Teil seines eigenen Builds aus. Der Provider weiß dann genau, welche Felder niemand liest (gefahrlos änderbar) und welche Änderungen einen namentlich bekannten Consumer brechen würden.

Pact, 2013 bei realestate.com.au entstanden, ist das bekannteste Werkzeug. Die Tests des Consumers laufen gegen einen Pact-Mock-Provider und zeichnen die gestellten Anfragen und die erwarteten Antworten in einer Vertragsdatei auf; die Pipeline des Providers spielt diese Anfragen gegen den echten Provider ab und prüft die Antworten. Ein Broker speichert Verträge und Verifikationsergebnisse, sodass eine Pipeline vor dem Deployment fragen kann, ob diese Version eines Service mit den Versionen seiner Consumer und Provider kompatibel ist, die bereits in Produktion laufen. Contract-Tests ersetzen nicht einige wenige End-to-End-Smoke-Tests, und sie prüfen die Struktur statt der fachlichen Bedeutung, aber sie erlauben es, Services mit Zuversicht unabhängig zu deployen, und genau darum geht es beim Aufteilen.

ACHTUNG

Häufige Fehler bei der Kommunikation zwischen Services

- Lange synchrone Ketten für Arbeit bauen, auf die der Nutzer nicht wartet, sodass ein langsamer Service die ganze Anfrage ausbremst.
- Nicht idempotente Aufrufe automatisch wiederholen und dann doppelte Bestellungen und Doppelbelastungen entdecken.
- 4xx-Fehler wiederholen oder ohne Backoff und Jitter wiederholen, was einen kurzen Ausfall in eine selbst verschuldete Überlast verwandelt.
- Datenbanktabellen als Ressourcen offenlegen, sodass jede Schemaänderung zu einer API-Änderung wird.
- Unbegrenzte Collections ohne Paginierung liefern, was im Test funktioniert und in Produktion scheitert.
- GraphQL oder gRPC aus Modegründen wählen und dann für N+1-Abfragen, Grenzen für Abfragekosten oder Browser-Proxys bezahlen, die das System nicht gebraucht hätte.
- Brechende Änderungen an einer internen API vornehmen, weil „uns beide Seiten gehören“, während ein Team anderswo sie unbemerkt aufrief.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum beseitigt eine Queue zwischen zwei Services nicht jede Kopplung zwischen ihnen?

Sie beseitigt die zeitliche Kopplung, aber beide Services hängen weiterhin vom Format und von der Bedeutung der Nachricht sowie vom Broker selbst ab, daher kann eine Änderung der Nachricht den Empfänger immer noch brechen.

▼ Warum ist DELETE idempotent, obwohl ein zweites DELETE 404 statt 204 liefern kann?

Idempotenz betrifft die beabsichtigte Wirkung auf dem Server, nicht die Antwort: Nach einem oder zwei DELETEs ist die Ressource in jedem Fall weg.

▼ Ein mobiler Client erzeugt bei jeder Wiederholung einen neuen Idempotenzschlüssel. Was geht schief?

Der Server sieht jede Wiederholung als neue Operation, daher kann eine verlorene Antwort mit anschließender Wiederholung trotzdem eine doppelte Bestellung erzeugen; der Schlüssel muss zur logischen Operation gehören, nicht zum HTTP-Versuch.

▼ **Warum kann Cursor-Paginierung eine stabile Liste zeigen, wenn neue Bestellungen eintreffen, während der Nutzer blättert, Offset-Paginierung aber möglicherweise nicht?**

Der Cursor zeigt direkt hinter einen bestimmten Eintrag, daher verschieben Einfügungen an anderer Stelle die nächste Seite nicht; ein Offset zählt Zeilen, sodass neue Zeilen Einträge über Seitengrenzen schieben und Duplikate oder Lücken verursachen.

▼ **Probieren Sie es aus: Wählen Sie jeweils einen Stil: Die Kunden-App gibt eine Bestellung auf; zwei stark ausgelastete interne Services tauschen Preisaufrufe aus; fünf Services müssen reagieren, wenn eine Bestellung aufgegeben wird. Wäre eine Wiederholung des ersten Falls durch die Mobil-App sicher? Begründen Sie Ihre Antworten.**

Eine Bestellung aufgeben: REST über HTTP mit einem stabilen öffentlichen Vertrag, aber POST ist nicht idempotent, eine Wiederholung ist also nur mit Idempotenzschlüssel sicher. Interne Preisaufrufe: gRPC passt zu häufigen internen Aufrufen, mit generiertem Code und kompakten Nachrichten. Reagierende Services: Veröffentlichen Sie in ein Topic, sodass jeder Subscriber seine eigene Kopie erhält und der Bestellservice sie nicht kennen muss. Eine Queue gäbe jede Nachricht nur einem Consumer, was hier falsch ist.

QUELLEN

- RFC 9110: HTTP Semantics (IETF, 2022): https://www.rfc-editor.org/rfc/rfc9110.html
- RFC 9457: Problem Details for HTTP APIs (IETF, 2023): https://www.rfc-editor.org/rfc/rfc9457.html
- Martin Fowler: Richardson Maturity Model: https://martinfowler.com/articles/richardsonMaturityModel.html
- gRPC documentation: Core concepts, architecture and lifecycle: https://grpc.io/docs/what-is-grpc/core-concepts/
- Ian Robinson: Consumer-Driven Contracts: A Service Evolution Pattern: https://martinfowler.com/articles/consumerDrivenContracts.html
- Apache Kafka 4.2.0 release announcement: https://kafka.apache.org/blog/2026/02/17/apache-kafka-4.2.0-release-announcement/
- RabbitMQ: Queues: https://www.rabbitmq.com/docs/queues
- RabbitMQ: Streams: https://www.rabbitmq.com/docs/streams
- RabbitMQ: Consumers: https://www.rabbitmq.com/docs/consumers
- Amazon SQS Developer Guide: queue types: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-queue-types.html
- Amazon SNS Developer Guide: fanout to Amazon SQS queues: https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html

- Google Cloud Pub/Sub: Order messages: https://docs.cloud.google.com/pubsub/docs/ordering
- NATS: Core NATS: https://docs.nats.io/nats-concepts/core-nats
- NATS: JetStream: https://docs.nats.io/nats-concepts/jetstream
- RFC 9110: HTTP Semantics, section 15.3.3: https://www.rfc-editor.org/rfc/rfc9110.txt
- Microsoft Learn: Asynchronous Request-Reply pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/asynchronous-request-reply
- Stripe: Receive events in your webhook endpoint: https://docs.stripe.com/webhooks
- Standard Webhooks specification: https://raw.githubusercontent.com/standard-webhooks/standard-webhooks/main/spec/standard-webhooks.md

Konsistenzmodelle, CAP und PACELC

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern
- 2.5 Verfügbarkeit und Zuverlässigkeit
- 6.1 Warum verteilen

Begriffe von früher, die er verwendet: ACID, Transaktion, Verfügbarkeit, Latenz, Invariante

DIE IDEE IN KÜRZE

Stellen Sie sich ein Geschäft mit drei Filialen vor, die jeweils eine Kopie der Bestandsliste führen und einander bei Änderungen anrufen. Fällt die Telefonleitung aus und fragt ein Kunde eine Filiale, ob das letzte Fahrrad noch vorrätig ist, hat diese Filiale zwei Möglichkeiten: „Ich kann es nicht sicher sagen, bitte warten Sie“ zu sagen, oder anhand ihrer eigenen Liste zu antworten und zu riskieren, ein Fahrrad zu verkaufen, das eine andere Filiale gerade verkauft hat. Selbst wenn die Telefone funktionieren, lässt das Anrufen der anderen Filialen vor jeder Antwort die Kunden warten. Verteilte Systeme stehen vor genau diesen Entscheidungen, und die richtige hängt davon ab, wie viel eine falsche Antwort kosten würde.

WAS SIE WISSEN MÜSSEN

- Ein Konsistenzmodell ist der Vertrag, der festlegt, welche veralteten oder umgeordneten Werte Clients beobachten dürfen; das C von CAP (Single-Copy, linearisierbar) unterscheidet sich vom C von ACID (gültiger Zustand) und von Isolationsebenen.
- Vom stärksten zum schwächsten: linearisierbar (eine Kopie, Echtzeit-Reihenfolge), sequenziell (eine vereinbarte Reihenfolge), kausal (Ursachen vor Wirkungen), Session Guarantees wie Read-your-Writes und Monotonic Reads und Eventual Consistency (Konvergenz, wenn keine Schreibvorgänge mehr kommen).
- Das CAP-Theorem (Brewers Vermutung von 2000, 2002 von Gilbert und Lynch bewiesen) besagt, dass ein repliziertes System während einer Netzwerkpartition pro Anfrage zwischen Konsistenz und Verfügbarkeit wählen muss; Partitionen sind nicht optional, daher ist „wähle zwei von dreien“ eine Fehldeutung.
- Brewers Artikel von 2012 empfiehlt, Partitionen zu erkennen, in einem expliziten Partitionsmodus zu arbeiten und sich danach mit Zusammenführungen und Kompensationen zu erholen, wobei für verschiedene Operationen unterschiedlich

entschieden wird.

- PACELC (Abadi, 2010 und 2012) ergänzt, dass Systeme andernfalls (else), ohne Partition, bei jeder Anfrage Latenz gegen Konsistenz abwägen.
- Mit N Replikaten, W Schreibbestätigungen und R Leseantworten sorgt die Wahl R + W > N dafür, dass sich Lese- und Schreibmengen überschneiden, aber Quoren allein sind nicht linearisierbar.
- Wählen Sie die Konsistenz je Anwendungsfall: stark für Geld, eindeutige Ressourcen und Invarianten; Session Guarantees für die eigenen Daten eines Nutzers; Eventual Consistency für Zähler, Feeds und Positionen.

Was „konsistent“ bedeutet, sobald Daten kopiert werden

Sobald ein System mehr als eine Kopie eines Datums hält, auf Replikaten in verschiedenen Zonen, in Caches oder in anderen Services, können die Kopien eine Zeit lang voneinander abweichen. Ein Konsistenzmodell ist der Vertrag, der festlegt, welche Abweichungen Clients beobachten dürfen. Es ist ein Versprechen darüber, was Lesevorgänge angesichts der erfolgten Schreibvorgänge liefern können, und damit ein Versprechen darüber, mit welchen Überraschungen der Anwendungscode umgehen muss.

Vorsicht beim mehrdeutigen Wort. Das C in ACID (Abschnitt 0.5) bedeutet, dass eine Transaktion die Datenbank von einem gültigen Zustand in einen anderen überführt und dabei Regeln einhält wie „ein Kontostand wird nie negativ“. Das C in CAP bedeutet etwas anderes und Engeres: Jeder Lesevorgang verhält sich so, als gäbe es eine einzige, aktuelle Kopie. Isolationsebenen von Datenbanken wie Serializable oder Read Committed sind eine dritte Familie, die gleichzeitige Transaktionen betrifft; sie werden unter Transaktionen und Isolation behandelt. In diesem Abschnitt geht es um die Garantien replizierter Daten.

Konsistenzmodelle vom stärksten zum schwächsten

Linearisierbarkeit, 1990 von Maurice Herlihy und Jeannette Wing definiert, ist das stärkste Modell für ein einzelnes Objekt. Jede Operation scheint zu einem Zeitpunkt zwischen ihrem Beginn und ihrem Ende augenblicklich wirksam zu werden, und alle Clients sehen dieselbe Reihenfolge, die die Echtzeit respektiert: Sobald ein Schreibvorgang abgeschlossen ist, sieht jeder danach beginnende Lesevorgang ihn oder etwas Neueres. Das System verhält sich, als gäbe es eine einzige Kopie. Beispiel aus der Essenslieferung: Zwei Kunden versuchen, den letzten Rabattcode einzulösen; mit einem linearisierbaren Speicher gelingt es genau einem, und der andere sieht, dass er weg ist, egal welches Replikat sie erreichen.

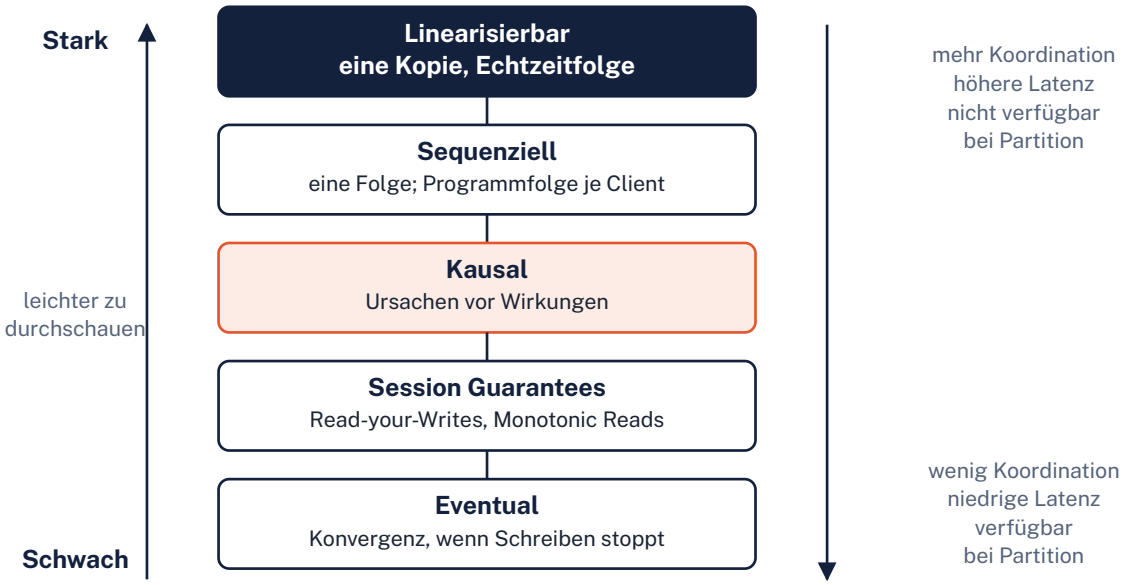
Sequenzielle Konsistenz, 1979 von Leslie Lamport für Multiprozessoren definiert, behält die eine globale Reihenfolge bei und hält die Operationen jedes Clients in der Reihenfolge, in der dieser Client sie ausgelöst hat, verzichtet aber auf die Echtzeitanforderung. Ein Client darf einen leicht veralteten Wert lesen, solange sich alle über die Reihenfolge aller Schreibvorgänge einig sind. In verteilten Datenbanken wird sie selten ausdrücklich angeboten, erklärt aber den Unterschied zwischen „eine vereinbarte Reihenfolge“ und „eine vereinbarte Reihenfolge, die zur Uhr passt“.

Kausale Konsistenz verlangt nur, dass kausal zusammenhängende Operationen von allen in derselben Reihenfolge gesehen werden: Hing das Schreiben von B davon ab, A gesehen zu haben, darf niemand B ohne A sehen. Gleichzeitige, voneinander unabhängige Schreibvorgänge dürfen von verschiedenen Clients in unterschiedlicher Reihenfolge gesehen werden. Beispiel: Ein Restaurant antwortet im Bestell-Chat auf die Frage eines Kunden „Ja, es ist glutenfrei“; niemand sollte je die Antwort ohne die Frage sehen. Kausale Konsistenz ist attraktiv, weil Jepsens viel genutzte Übersicht der Konsistenzmodelle sie als mit *Sticky*-Verfügbarkeit erreichbar einstuft: Sie kann während einer Partition weiterarbeiten, solange jeder Client beim selben Replikat bleibt.

Die Session Guarantees, 1994 von Douglas Terry und Kollegen beschrieben, sind schwächere Versprechen pro Client, die die verwirrendsten Anomalien beseitigen. Read-your-Writes: Nachdem ein Kunde die Lieferadresse geändert hat, zeigt sein nächster Lesevorgang die neue Adresse, auch wenn andere Nutzer sie erst später sehen. Monotonic Reads: Hat ein Client einmal den Bestellstatus „unterwegs“ gesehen, sieht er nie später „in Zubereitung“, weil ein späterer Lesevorgang ein nachhinkendes Replikat getroffen hat. (Zwei weitere Garantien, Monotonic Writes und Writes-Follow-Reads, schränken die Reihenfolge ein, in der die Schreibvorgänge eines Clients angewendet werden.)

Eventual Consistency, 2008 von Werner Vogels bekannt gemacht, ist die schwächste nützliche Garantie: Kommen keine neuen Schreibvorgänge, konvergieren alle Replikate irgendwann auf denselben Wert. Sie sagt nichts darüber, wie lange das dauert oder was in der Zwischenzeit gelesen werden kann, und eine einfache Implementierung, die widersprüchliche Schreibvorgänge löst, indem sie den mit dem jüngsten Zeitstempel behält, bekannt als Last Write Wins, verwirft den anderen stillschweigend. Stärkere Formen wie Strong Eventual Consistency mit konfliktfreien replizierten Datentypen (CRDTs) garantieren, dass Replikate, die dieselben Aktualisierungen erhalten haben, denselben Zustand halten, indem sie Datentypen nutzen, deren gleichzeitige Aktualisierungen sich immer zusammenführen lassen, etwa Zähler und Mengen.

Die Konsistenzleiter



Jede Stufe nach unten entfernt eine Garantie und etwas Koordination; die Session Guarantees sind Versprechen pro Client, die zwischen kausaler und einfacher Eventual Consistency liegen.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Denken Sie an eine Einkaufsliste der Familie auf drei Smartphones, die sich synchronisieren, sobald sie Empfang haben. Linearisierbar bedeutet, dass sich die Liste verhält, als gäbe es ein einziges Blatt Papier am Kühlschrank. Eventual bedeutet, dass die Smartphones übereinstimmen, sobald alle eine Weile Empfang hatten, aber bis dahin kauft vielleicht jemand Milch, die ein anderer schon gekauft hat. Alles dazwischen ist ein anderer Kompromiss zwischen „immer übereinstimmen“ und „immer funktionieren, selbst im Tunnel“.

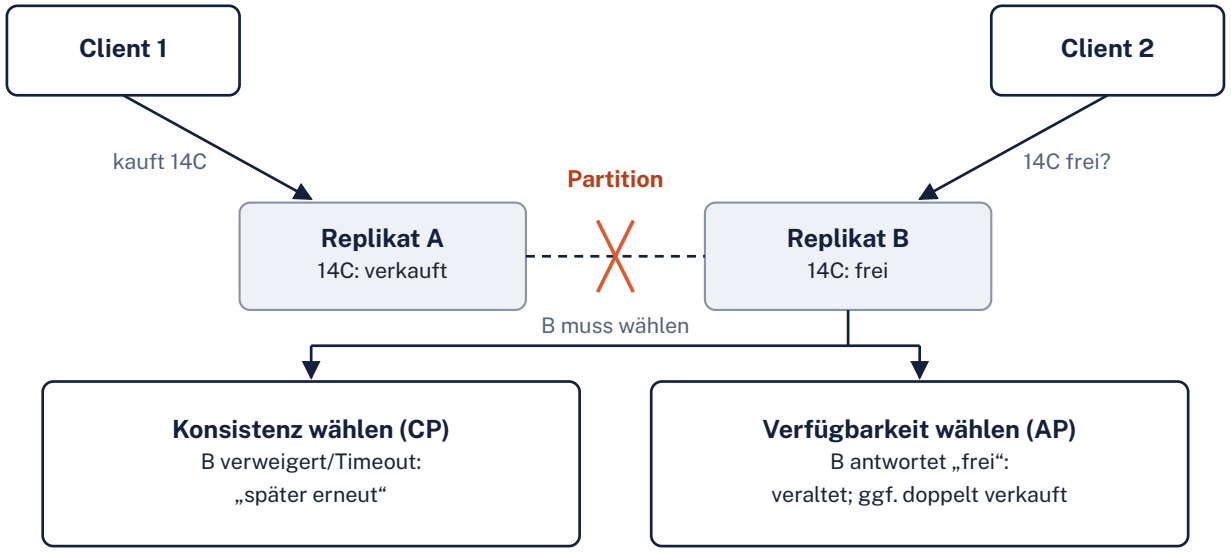
CAP: was es tatsächlich besagt

Eric Brewer stellte die CAP-Idee im Jahr 2000 in seiner Keynote auf dem ACM Symposium on Principles of Distributed Computing (PODC) als Vermutung vor. 2002 veröffentlichten Seth Gilbert und Nancy Lynch einen formalen Beweis, der sie für ihre präzisen Definitionen zu einem Theorem machte. **Konsistenz** bedeutete Linearisierbarkeit. **Verfügbarkeit** bedeutete, dass jede Anfrage, die ein nicht ausgefallener Knoten erhält, zu einer Antwort führen muss. **Partitionstoleranz** bedeutete, dass das System seine Versprechen auch dann hält, wenn das Netzwerk beliebig viele Nachrichten zwischen Knoten verliert.

Genau gelesen ist das CAP-Theorem eng. Es besagt, dass ein repliziertes System, wenn eine Netzwerkpartition auftritt, sodass manche Knoten andere nicht erreichen, für jede Anfrage zwischen Konsistenz und Verfügbarkeit wählen muss. Ein Knoten, der von den neuesten Schreibvorgängen abgeschnitten ist, kann entweder die Antwort verweigern (oder warten, bis die Partition behoben ist) und so die Konsistenz wahren, oder mit dem antworten, was er hat, und so die Verfügbarkeit wahren, mit dem Risiko, veraltete Daten zu liefern oder widersprüchliche Schreibvorgänge anzunehmen. Das Diagramm zeigt das Dilemma für ein Ticketsystem, das Platz 14C verkauft.

Mehrere verbreitete Lesarten sind falsch. **„Wähle zwei beliebige von dreien“** legt nahe, Partitionstoleranz sei optional, doch in jedem System, das über ein Netzwerk kommuniziert, treten Partitionen auf, ob Sie sie wählen oder nicht; die einzige echte Wahl ist, wie sich das System dann verhält. Ein sogenanntes „CA“-System ist eines, das nicht entschieden hat, was bei einer Partition geschehen soll; vertretbar ist das nur dort, wo Partitionen praktisch unmöglich sind, etwa auf einem einzelnen Knoten. **„Unsere Datenbank ist CP (oder AP)“** ist meist ungenau: CAP-Verfügbarkeit ist eine sehr starke Eigenschaft, die viele als hochverfügbar vermarktete Systeme nicht haben, und viele als CP bezeichnete Systeme sind auch nicht linearisierbar. Martin Kleppmann argumentierte 2015, Datenbanken sollten überhaupt nicht als CP oder AP etikettiert werden, weil die meisten keine der beiden Definitionen erfüllen und ihr Verhalten von der Konfiguration abhängt. **„CAP erklärt unsere Latenz“** ist ebenfalls falsch: CAP sagt nichts darüber, was geschieht, wenn keine Partition vorliegt, also die meiste Zeit.

Die CAP-Entscheidung, konkret gemacht



Replikat B kann nicht erfahren, dass Platz 14C gerade auf Replikat A verkauft wurde, daher muss es entweder die Antwort verweigern oder riskieren, den Platz doppelt zu verkaufen.

CAP zwölf Jahre später

2012 erklärte Brewer selbst in einem Artikel in IEEE Computer mit dem Titel „CAP Twelve Years Later: How the Rules Have Changed“, dass die Formel „zwei von dreien“ in die Irre geführt habe. Partitionen sind selten, daher kann ein System fast immer sowohl Konsistenz als auch Verfügbarkeit bieten; die Wahl gilt nur während einer Partition und nur für die betroffenen Operationen. Die Wahl muss auch nicht global sein: Verschiedene Operationen und verschiedene Daten können unterschiedlich entscheiden.

Brewer empfahl, in drei Schritten ausdrücklich für Partitionen zu entwerfen: die Partition **erkennen**, in einen expliziten **Partitionsmodus** wechseln, in dem manche Operationen eingeschränkt sind (zum Beispiel Bestellungen annehmen, aber Rückerstattungen aufschieben), und danach einen **Wiederherstellungsprozess** ausführen, der den Zustand zusammenführt und Fehler aus der Zeit der Partition kompensiert. Er wies auch darauf hin, dass das C von CAP Single-Copy-Konsistenz ist, eine strengere Teilmenge dessen, was das C von ACID schützt, und dass Invarianten wie „nie überverkaufen“ unter Umständen nachträglich repariert werden müssen. Diese Sichtweise führt direkt zu den fachlichen Kompensationen der Sagas: Eine Fluggesellschaft, die einen Flug überbucht, entschädigt den Fluggast, statt während einer Netzwerkstörung alle Buchungen abzulehnen.

Vector Clocks und Hybrid Logical Clocks

Wer nach einer Partition zusammenführt, muss wissen, welche Schreibvorgänge wirklich in Konflikt stehen, und Wanduhren können das nicht sagen (Abschnitt 6.1). Eine **Vector Clock** hängt an jede Version eine Liste aus Paaren von Knoten und Zähler. Ist jeder Zähler in einer Liste kleiner oder gleich denen in einer anderen, ist die erste Version ein Vorfahre und kann entfallen; andernfalls sind beide nebenläufig und müssen zusammengeführt werden. Amazons Dynamo-Paper von 2007 nutzte das für Warenkörbe: Keine Hinzufügung geht verloren, gelöschte Artikel können aber wieder auftauchen. Die Listen wachsen, deshalb kürzte Dynamo sie. Eine **Hybrid Logical Clock** (Kulkarni und andere, 2014) ist ein einziger kompakter Zeitstempel: eine Zeit nahe der physischen Uhr plus ein Zähler, sodass ein Ereignis, das zu einem anderen geführt hat, immer den kleineren Wert trägt. Sie ordnet Ereignisse, doch anders als bei einer Vector Clock beweist ein kleinerer Wert keinen kausalen Zusammenhang. CockroachDB nutzt Hybrid Logical Clocks.

PACELC: die Abwägung, die Sie jeden Tag bezahlen

Daniel Abadi wies 2010 in einem Blogbeitrag und 2012 in einem Artikel in IEEE Computer, „Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story“, darauf hin, dass CAP die Abwägung ignoriert, vor der Entwerfende tatsächlich die

meiste Zeit stehen. PACELC liest sich so: Gibt es eine **P**artition, wähle zwischen Verfügbarkeit (**A**vailability) und Konsistenz (**C**onsistency); andernfalls (**E**lse), im Normalbetrieb, wähle zwischen Latenz (**L**atency) und Konsistenz (**C**onsistency).

Die „Else"-Hälfte ist die wichtige Ergänzung. Um einen Lesevorgang konsistent zu machen, muss ein repliziertes System koordinieren: den Schreibvorgang an andere Replikate senden und auf Bestätigungen warten, jeden Lesevorgang an einen Leader leiten oder mehrere Replikate kontaktieren und vergleichen. Jedes davon kostet Netzwerk-Roundtrips, und über Regionen hinweg kann ein Roundtrip Dutzende oder sogar Hunderte Millisekunden kosten. Ein System, das ohne Koordination vom nächstgelegenen Replikat antwortet, ist schneller, aber möglicherweise veraltet. Diese Abwägung wird bei jeder Anfrage bezahlt, mit oder ohne Partition.

PACELC-Einstufungen werden oft zitiert, etwa PA/EL für Systeme, die auf Geschwindigkeit und Verfügbarkeit ausgelegt sind, und PC/EC für Systeme, die immer Konsistenz bevorzugen. Behandeln Sie sie als Beschreibungen einer Konfiguration, nicht eines Produkts: Dieselbe Datenbank kann in einem Deployment PA/EL und in einem anderen PC/EC sein, je nach Einstellungen, die pro Tabelle, pro Abfrage oder pro Anfrage gewählt werden.

Konsens und Koordination: Leader, Leases und Sperren

Failover, Shard-Karten und geplante Jobs brauchen alle, dass sich mehrere Maschinen auf eine Antwort einigen, etwa welcher Knoten nun führt, selbst wenn einige abgestürzt oder Nachrichten verspätet sind. Das ist das **Konsens**problem, und eine einmal getroffene Entscheidung ist endgültig. Leslie Lamports Paxos ist der klassische Algorithmus. Raft (Ongaro und Ousterhout, 2014) bietet dieselbe Fehlertoleranz und Performance, wurde aber so entworfen, dass es leichter zu verstehen ist. Ein Raft-Leader sendet Heartbeats; ein Follower, der vor seinem zufälligen Timeout keinen hört, startet eine Wahl in einem neuen nummerierten *Term*, und eine Mehrheit der Stimmen macht ihn zum Leader. Fünf Server arbeiten mit zwei ausgefallenen weiter. Bei mehr Ausfällen kommt der Cluster nicht mehr voran, liefert aber nie ein falsches Ergebnis. Cluster haben meist eine ungerade Größe, denn die etcd-Dokumentation merkt an, dass vier Mitglieder nicht mehr Ausfälle verkraften als drei, und sie rät zu höchstens sieben, da jeder Schreibvorgang mehr Maschinen erreichen muss.

Nur wenige Teams implementieren das selbst. Sie nutzen **etcd**, das auf Raft beruht und der Backing Store für alle Kubernetes-Clusterdaten ist, oder **Apache ZooKeeper**, dessen Wahlrezept den Kandidaten mit dem am niedrigsten nummerierten temporären Knoten zum Leader macht. Beide halten kleine Mengen an Koordinationsdaten, keine Geschäftsdatensätze.

Ein **Lease** ist ein Anspruch mit Zeitlimit: Der Inhaber muss es laufend erneuern, und stirbt er, läuft das Lease ab und ein anderer Knoten übernimmt. Kubernetes nutzt Leases für Knoten-Heartbeats und die Leader-Wahl. Die Gefahr ist ein Inhaber, der pausiert, vielleicht wegen einer Garbage Collection, dann aufwacht, ohne zu wissen, dass sein Lease abgelaufen ist, und trotzdem schreibt; vorher zu prüfen hilft nicht, weil die Pause zwischen Prüfung und Schreiben fallen kann. Ein **Fencing Token** löst das: eine Zahl, die mit jeder Vergabe steigt und mit jedem Schreibvorgang gesendet wird, sodass der Speicher jedes Token ablehnt, das niedriger ist als eines, das er schon gesehen hat.

Verteilte Sperren werfen dieselbe Frage auf. Martin Kleppmann unterscheidet Sperren für Effizienz, bei denen ein Fehler doppelte Arbeit bedeutet, etwa eine wiederholte E-Mail, von Sperren für Korrektheit, bei denen ein Fehler Daten beschädigt. Der Redlock-Algorithmus von Redis setzt eine Sperre auf einer Mehrheit unabhängiger Redis-Master. Kleppmann argumentierte 2016, dass er von Timing-Annahmen abhängt und keine Fencing Tokens ausgibt und daher nicht die Korrektheit schützen sollte. Salvatore Sanfilippo, der Schöpfer von Redis, erwiderte, er sei unter seinem angegebenen Modell sicher, und akzeptierte zugleich den Punkt zu monotonen Uhren. Die Redis-Dokumentation verlinkt beide; die Debatte ist ungelöst, lesen Sie also beide. Konsens-Speicher brauchen dieselbe Sorgfalt: Jepsens Test von etcd 3.4.3 ergab, dass zwei Clients dieselbe Sperre halten konnten, und riet zu Fencing. Die praktische Lehre lautet, dass sich die geschützte Ressource selbst verteidigen muss, durch eine Fencing-Prüfung oder einen Datenbank-Constraint. Betreibt die Disposition einen einzigen Zuweisungs-Scheduler, darf ein gelegentlicher doppelter Lauf keinen Schaden anrichten.

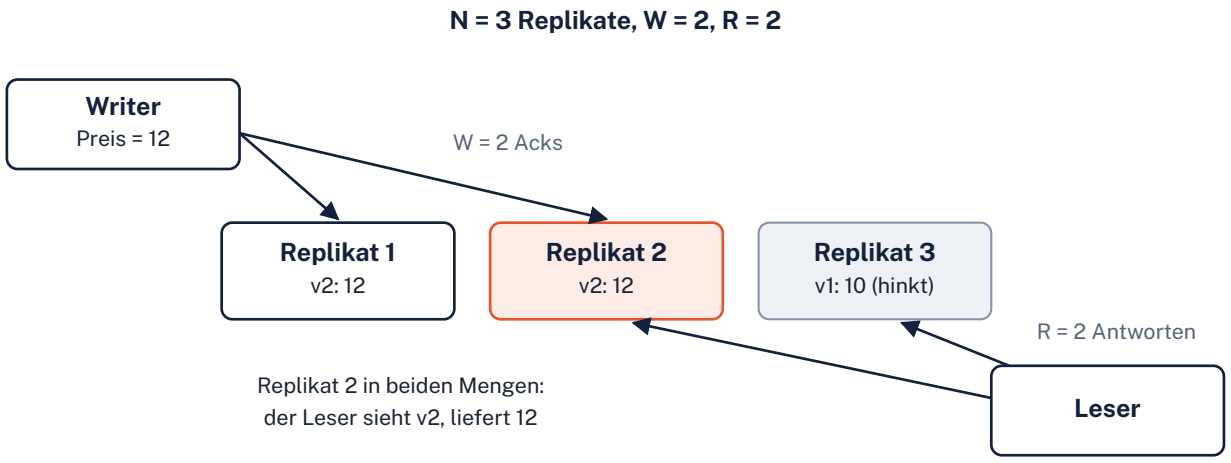
Quoren: Konsistenz mit Arithmetik einstellen

Leaderlose replizierte Speicher nach dem Vorbild von Amazons Dynamo-Paper (2007) lassen jede Operation wählen, wie viele Replikate sie einbezieht. Bei *N* Replikaten eines Eintrags gilt ein Schreibvorgang als bestätigt, wenn *W* Replikate ihn gespeichert haben, und ein Lesevorgang sammelt Antworten von *R* Replikaten und liefert die neueste Version darunter. Gilt R + W > N, überschneidet sich jede Lesemenge mit jeder Schreibmenge in mindestens einem Replikat, sodass ein Lesevorgang mindestens eine Kopie des neuesten bestätigten Schreibvorgangs einschließt.

Die Wahl von *R* und *W* wägt Konsistenz, Latenz und Verfügbarkeit gegeneinander ab. Niedrigere Werte bedeuten, auf weniger und schnellere Replikate zu warten und mehr Ausfälle zu tolerieren, um den Preis möglicher veralteter Lesevorgänge. Höhere Werte bedeuten das Gegenteil. Schreibvorgänge werden meist ohnehin an alle *N* Replikate gesendet; *W* entscheidet nur, auf wie viele Bestätigungen der Client wartet.

Die Überschneidung der Quoren ist für aktuelle Lesevorgänge notwendig, aber für sich genommen keine Linearisierbarkeit. Gleichzeitige Schreibvorgänge können nach Zeitstempeln von Uhren geordnet werden, die nicht übereinstimmen, sodass ein späterer Schreibvorgang unter Last-Write-Wins verlieren kann. Ein Schreibvorgang, der scheiterte, nachdem er einige Replikate erreicht hatte, kann auf diesen noch sichtbar sein. „Sloppy“ Quoren, die während Ausfällen Schreibvorgänge auf Ersatzknoten annehmen, heben die Überschneidung auf. Und zwei Lesevorgänge, die sich mit einem noch laufenden Schreibvorgang überschneiden, können erst neue, dann alte Werte sehen, sofern der Lesende nachhinkende Replikate nicht vor der Rückgabe repariert. Speicher, die echte Compare-and-Set-Semantik brauchen, ergänzen für genau diese Operationen ein Konsensprotokoll, etwa die auf Paxos basierenden Lightweight Transactions in Apache Cassandra.

Warum R + W > N funktioniert



Der Schreibvorgang wurde von den Replikaten 1 und 2 bestätigt; der Lesevorgang fragt die Replikate 2 und 3; die Überschneidung bei Replikat 2 bedeutet, dass die neueste Version in der Lesemenge liegt.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: N, W und R für einen Speicher mit Speisekartenpreisen wählen

Die Essenslieferplattform speichert Speisekartenpreise in einem leaderlosen Speicher mit N = 3 Replikaten, eines pro Availability Zone. Jedes Replikat antwortet innerhalb seiner Zone in etwa 2 ms und aus einer anderen Zone in etwa 5 ms. Der Quorum-Client wartet auf die W-t- (bzw. R-t-)schnellste Antwort.

W = 2, R = 2 (R + W = 4 > 3)	Überschneidung garantiert. Lese- und Schreibvorgänge warten jeweils auf das zweitschnellste Replikat (etwa 5 ms). Beide gelingen auch bei Ausfall einer Zone. Ein gängiger, ausgewogener Standard.
W = 3, R = 1 (4 > 3)	Überschneidung garantiert, und Lesevorgänge sind schnell (nächstes Replikat, etwa 2 ms), aber jeder Schreibvorgang scheitert, wenn auch nur ein Replikat ausgefallen ist: Die Schreibverfügbarkeit ist das Produkt dreier Verfügbarkeiten. Geeignet für Daten, die selten geschrieben und ständig gelesen werden, sofern gescheiterte Schreibvorgänge später wiederholt werden können.
W = 1, R = 3 (4 > 3)	Schreibvorgänge sind schnell und überstehen zwei Ausfälle; jeder Lesevorgang muss alle drei Replikate erreichen, daher scheitern Lesevorgänge, wenn irgendeine Zone nicht erreichbar ist. Selten eine gute Abwägung.
W = 1, R = 1 (2, nicht > 3)	Am schnellsten und am verfügbarsten, aber ein Lesevorgang kann den neuesten Schreibvorgang komplett verpassen. Akzeptabel für einen Beliebtheitszähler; nicht für einen Preis, der dem Kunden berechnet wird.
N = 5, W = 3, R = 3 (6 > 5)	Überschneidung garantiert bei Toleranz von zwei ausgefallenen Replikaten für Lese- und Schreibvorgänge, um den Preis von mehr Speicher und etwas langsameren Quoren.

Allgemeine Regel: Mit R + W > N tolerieren Schreibvorgänge N − W Ausfälle und Lesevorgänge N − R Ausfälle.

Die Überschneidung liefert aktuelle Lesevorgänge nur für bestätigte Schreibvorgänge und nur mit zuverlässiger Versionierung; lesen Sie die Einschränkungen oben, bevor Sie sich bei Invarianten wie Lagerbeständen auf Quoren verlassen.

Für Preise wählte die Plattform W = 2, R = 2 und berechnete Kunden den Preis, der beim Bezahlvorgang in der Bestellung gespeichert wurde, sodass ein veralteter Speisekarten-Lesevorgang den berechneten Betrag nie ändern konnte.

Wie sich bekannte Datenbanken konfigurieren lassen

Reale Produkte sitzen selten an einem einzigen Punkt der Leiter. Sie bieten Einstellungen, und die richtige Beschreibung lautet „wie dieses Deployment konfiguriert ist“. Es folgen vier gängige Beispiele; prüfen Sie die aktuelle Dokumentation der Version, die Sie betreiben, denn Standardwerte ändern sich zwischen Releases.

Konfigurierbare Konsistenz in vier weit verbreiteten Datenbanken (Verhalten laut Dokumentation zum Zeitpunkt der Erstellung)

Datenbank	Typisches Standardverhalten	Einstellungen, die die Konsistenz stärken	Worauf zu achten ist
PostgreSQL mit Streaming Replication	Ein Primary nimmt alle Schreibvorgänge an; die Replikation zu Standbys ist standardmäßig asynchron, daher können Lesevorgänge von einem Standby veraltet sein.	`synchronous_standby_names` mit `synchronous_commit` (zum Beispiel `remote_apply`) lässt einen Commit warten, bis die Standbys ihn haben; lesen Sie für Read-your-Writes vom Primary.	Synchrone Replikation erhöht die Commit-Latenz, und Schreibvorgänge stocken, wenn die erforderlichen Standbys nicht erreichbar sind.
MongoDB-Replica-Set	Schreibvorgänge verwenden in den meisten Deployments standardmäßig Majority-Bestätigung; Lesevorgänge vom Primary verwenden standardmäßig den Read Concern `local`, der Daten liefern kann, die später zurückgerollt werden.	Read Concern `majority` oder `linearizable`; kausal konsistente Sessions, wenn Lese- und Schreibvorgänge beide Majority verwenden.	Lesen von Secondaries (Read Preference) tauscht Aktualität gegen Latenz und Last.
Apache Cassandra	Das Konsistenzlevel wird pro Anfrage gewählt (ONE, QUORUM, LOCAL_QUORUM, ALL und andere); Konflikte werden über Last-Write-Wins-Zeitstempel aufgelöst.	QUORUM (oder LOCAL_QUORUM innerhalb eines Rechenzentrums) für Lese- und Schreibvorgänge, sodass R + W > Replikationsfaktor; Lightweight Transactions für Compare-and-Set.	Clock Skew beeinflusst Last-Write-Wins; LOCAL_QUORUM gibt keine rechenzentrumsübergreifende Garantie.
Amazon DynamoDB	Lesevorgänge sind standardmäßig eventual consistent; Global Tables replizieren standardmäßig asynchron zwischen Regionen.	`ConsistentRead=true` auf Tabellen und lokalen sekundären Indizes; Global Tables mit Multi-Region Strong Consistency (seit 2025 allgemein verfügbar).	Stark konsistente Lesevorgänge sind auf globalen sekundären Indizes nicht verfügbar, verbrauchen die doppelte Lesekapazität und können langsamer sein.

Konsistenz je Anwendungsfall wählen

Die nützlichste Gewohnheit ist, nicht mehr zu fragen „Ist unser System konsistent?“, sondern für jedes Datum und jede Operation: *Was geht schief, wenn ein Client einen veralteten oder umgeordneten Wert sieht, und was kostet das?* Geldbewegungen, eindeutige Ressourcen und Invarianten rechtfertigen meist starke Konsistenz und ihre Latenz. Zähler, Feeds, Empfehlungen und Analysen meist nicht. Viele für Nutzer sichtbare Anomalien verschwinden schon mit den Session Guarantees, die günstig sind: Leiten Sie die Lesevorgänge eines Nutzers für einige Sekunden nach dessen Schreibvorgang an den Primary, oder führen Sie ein Versions-Token in der Session mit und warten Sie, bis das Replikat damit gleichgezogen hat.

Konsistenzbedarf je Anwendungsfall

Anwendungsfall	Kosten eines veralteten oder umgeordneten Lesevorgangs	Sinnvolle Garantie	Typischer Mechanismus
Kontostand vor einer Abhebung	Überziehung über das Limit; Geld verloren	Linearisierbar (und serialisierbare Transaktionen)	Ein einzelner Leader für das Konto, synchrone Replikation oder Konsens
Letzter Platz oder letztes Ticket beim Vorverkaufsstart	Einen Platz doppelt verkaufen; Erstattungen und verärgerte Kunden	Linearisierbar pro Platz	Bedingter Schreibvorgang (Compare-and-Set) auf dem Platz-Datensatz
Eindeutiger Benutzername bei der Registrierung	Zwei Konten mit demselben Namen	Linearisierbar bei der Eindeutigkeitsprüfung	Unique Constraint in einer Datenbank oder eine konsensgestützte Registry
Eigene Lieferadresse des Kunden nach der Bearbeitung	Essen geht an die alte Adresse	Read-your-Writes	Nach einem Schreibvorgang vom Leader lesen oder ein Versions-Token in der Session
Dem Kunden angezeigter Bestellstatus	Status scheint zurückzuspringen; Verwirrung und Support-Anrufe	Monotonic Reads plus Read-your-Writes	Sticky Routing zu einem Replikat oder Versions-Tokens
Restaurant-Chat mit dem Kunden	Eine Antwort erscheint vor ihrer Frage	Kausal	Kausal konsistenter Speicher oder Reihenfolge pro Unterhaltung
Warenkorb	Ein entfernter Artikel taucht wieder auf; leicht ärgerlich	Eventual, mit Zusammenführung	Hinzufügungen immer annehmen; Versionen zusammenführen (das Beispiel aus dem Dynamo-Paper)
Like-Zähler oder Restaurantbewertung	Zahl weicht ein paar Sekunden lang um einige ab	Eventual	Asynchrone Replikation, Zähler oder CRDTs
Kurierstandort auf der Karte	Punkt ein paar Sekunden hinterher	Eventual, neuester Wert gewinnt	Schreibvorgänge mit niedriger Latenz an das nächste Replikat; alte Positionen verworfen

ACHTUNG

Häufige Fehler

- Für ein ganzes System zu sagen „wir haben AP gewählt“, statt pro Operation und pro Datum zu entscheiden.
- Annehmen, eine replizierte Datenbank sei linearisierbar, weil ihr Marketing „stark konsistent“ sagt, ohne nachzulesen, was die Garantie abdeckt (einzelne Schlüssel? eine Region? Indizes?).
- Einem System Read Replicas oder Caches hinzufügen und vergessen, dass damit jeder für Nutzer sichtbare Lesepfad eventual consistent geworden ist.
- Sich auf Last Write Wins mit nicht synchronisierten Uhren verlassen und stillschweigend Aktualisierungen verlieren.
- Quoren mit R + W > N nutzen und glauben, dass Lagerreservierungen damit ohne Compare-and-Set sicher sind.
- Eventual Consistency als technisches Detail behandeln, obwohl sie für Nutzer sichtbar ist und Produktentscheidungen darüber erfordert, was sie sehen, während die Daten konvergieren.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Was ist der Unterschied zwischen Linearisierbarkeit und sequenzieller Konsistenz?

Beide verlangen eine einzige Reihenfolge der Operationen, auf die sich alle Clients einigen und die die eigene Reihenfolge jedes Clients respektiert; Linearisierbarkeit verlangt zusätzlich, dass diese Reihenfolge die Echtzeit respektiert, sodass ein Lesevorgang, der nach Abschluss eines Schreibvorgangs beginnt, ihn sehen muss.

▼ Warum ist eine „CA“-Datenbank in einem verteilten System keine sinnvolle Wahl?

Weil Netzwerkpartitionen in jedem System auftreten, das über ein Netzwerk kommuniziert; ein System, das annimmt, dass sie nie vorkommen, hat einfach nicht entschieden, was es tun wird, wenn sie auftreten.

▼ Was ist bei N = 5 und W = 2 das kleinste R, das eine Überschneidung von Lese- und Schreibmengen garantiert?

R = 4, weil R + W größer als 5 sein muss.

▼ **Ein Nutzer bearbeitet sein Profil, und die neu geladene Seite zeigt die alte Version, obwohl es keine Rolle spielt, wenn andere Nutzer die Änderung erst spät sehen. Welche Garantie fehlt, und was ist eine günstige Abhilfe?**

Read-your-Writes; die Lesevorgänge dieses Nutzers kurz nach einem Schreibvorgang an den Leader leiten oder ein Versions-Token mitführen und auf ein Replikat warten, das gleichgezogen hat.

▼ **Probieren Sie es aus: Ein Speicher für Platzpreise verwendet N = 5, W = 2, R = 3, und ein Kollege sagt, Lesevorgänge überlappten immer mit dem neuesten Schreibvorgang. Prüfen Sie das, und schlagen Sie eine Korrektur vor. Würden Sie allein auf Quoren setzen, um zu verhindern, dass zwei Fans denselben Platz kaufen?**

R + W = 5, nicht größer als N, ein Lesevorgang kann also den letzten Schreibvorgang verpassen. Setzen Sie W = 3 und R = 3, das ergibt 6 > 5; beide funktionieren noch, wenn zwei Replikate ausfallen, bei etwas höherer Latenz. Wählen Sie nach Verwendung: Veraltete Preise mögen kurz tolerierbar sein. Quoren allein sind nicht linearisierbar, sie schützen also keinen eindeutigen Platz. Der Verkauf eines Platzes braucht starke Konsistenz, etwa eine Single-Leader-Transaktion oder eine Eindeutigkeitsbedingung.

QUELLEN

- Eric Brewer: CAP Twelve Years Later: How the Rules Have Changed (InfoQ reprint of the IEEE Computer article): https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/
- Daniel Abadi: Problems with CAP, and Yahoo's little known NoSQL system (2010, origin of PACELC): https://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html
- Jepsen: Consistency models: https://jepsen.io/consistency/models
- Martin Kleppmann: Please stop calling databases CP or AP (2015): https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html
- Apache Cassandra documentation: Dynamo (tunable consistency and quorums): https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html
- Raft consensus algorithm site: https://raft.github.io/
- Ongaro and Ousterhout: In Search of an Understandable Consensus Algorithm (2014): https://raft.github.io/raft.pdf
- etcd FAQ: https://etcd.io/docs/v3.5/faq/
- Kubernetes: Operating etcd clusters: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/
- Kubernetes: Leases: https://kubernetes.io/docs/concepts/architecture/leases/

- Apache ZooKeeper: Recipes and solutions: https://zookeeper.apache.org/doc/current/recipes.html
- Martin Kleppmann: How to do distributed locking (2016): https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html
- Salvatore Sanfilippo: Is Redlock safe?: https://antirez.com/news/101
- Redis: Distributed locks: https://redis.io/docs/latest/develop/clients/patterns/distributed-locks/
- Jepsen: etcd 3.4.3: https://jepsen.io/analyses/etcd-3.4.3
- DeCandia and others: Dynamo: Amazon's Highly Available Key-value Store (2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- Kulkarni and others: Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases: https://cse.buffalo.edu/tech-reports/2014-04.pdf
- CockroachDB: Transaction layer: https://docs.cockroachlabs.com/docs/stable/architecture/transaction-layer

Verteilte Transaktionen, Sagas und das Outbox-Muster

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern
- 6.3 Microservices
- 6.4 Ereignisgesteuert

Begriffe von früher, die er verwendet: ACID, Datenbank pro Service, Choreografie, Orchestrierung, At-least-once-Zustellung

DIE IDEE IN KÜRZE

Einen Urlaub zu buchen heißt, einen Flug, ein Hotel und einen Mietwagen bei drei verschiedenen Unternehmen zu buchen. Niemand kann alle drei Buchungen im exakt selben Augenblick zustande bringen. Stattdessen buchen Sie eines nach dem anderen, und wenn sich herausstellt, dass der Mietwagen nicht verfügbar ist, stornieren Sie Hotel und Flug, vielleicht gegen eine Stornogebühr. Das ist eine Saga: eine Reihe getrennter Schritte, jeder mit einem geplanten Weg, ihn rückgängig zu machen. Software, die mehrere Services umfasst, funktioniert genauso, und die Kunst liegt darin, die Reihenfolge der Schritte zu wählen und die Stornierungen zu planen, bevor etwas schiefgeht.

WAS SIE WISSEN MÜSSEN

- Two-Phase Commit sorgt über Prepare- und Commit-Phasen für Atomarität über alle Teilnehmer hinweg, blockiert aber, solange der Koordinator ausgefallen ist, und hält Sperren über das Netzwerk; XA (X/Open, 1991) standardisiert es.
- Über unabhängig verantwortete Services hinweg vervielfacht 2PC die Nichtverfügbarkeit, hält Sperren aus der Ferne, koppelt Deployments und wird von den meisten APIs und vielen Brokern nicht unterstützt, daher meidet man es im Allgemeinen.
- Eine Saga (Garcia-Molina und Salem, 1987) ist eine Folge lokaler Transaktionen, jede mit einer kompensierenden Transaktion; entweder werden alle Schritte abgeschlossen, oder die abgeschlossenen werden in umgekehrter Reihenfolge kompensiert.
- Kompensationen sind neue, vorwärtsgerichtete Transaktionen, keine Rollbacks: Zwischenzustände können bereits gesehen worden sein, und Kompensationen müssen idempotent und wiederholbar sein und auch in falscher Reihenfolge eintreffen dürfen.

- Sagas fehlt die Isolation; zu den von Chris Richardson beschriebenen Gegenmaßnahmen gehören semantische Sperren, kommutative Updates, pessimistische Sicht, erneutes Lesen des Werts, Versionsdateien und die Wahl nach Wert.
- Die Transactional Outbox schreibt die fachliche Änderung und ihre Nachricht in einer lokalen Transaktion, und ein Relay (Polling oder Change Data Capture wie Debezium) veröffentlicht sie mindestens einmal; idempotente Empfänger mit einer Inbox entfernen Duplikate.
- Bevorzugen Sie nach Möglichkeit eine lokale Transaktion, gestalten Sie Grenzen neu, wenn die Aufteilung zufällig ist, und setzen Sie Sagas ein, wenn sich Schritte kompensieren oder wiederholen lassen.

Das Problem: eine Geschäftsaktion, mehrere Datenbanken

Innerhalb einer Datenbank macht eine Transaktion mehrere Änderungen atomar: Die Bestellzeile, ihre Positionen und die Zahlungsreferenz werden alle gespeichert oder keine (Abschnitt 0.5). Sobald die Daten auf Services verteilt sind, die jeweils ihre eigene Datenbank besitzen, wie es Microservices und Datenbank pro Service empfehlen, berührt eine einzelne Geschäftsaktion wie das Aufgeben einer Essensbestellung mehrere Datenbanken verschiedener Services: die des Bestellservice, die des Zahlungsservice und die des Dispositionsservice. Keine einzelne lokale Transaktion kann sie alle abdecken.

Es gibt zwei Familien von Antworten. Die eine besteht darin, viele Ressourcenmanager über ein verteiltes atomares Commit-Protokoll wie einen einzigen handeln zu lassen; der Klassiker dafür ist Two-Phase Commit. Die andere akzeptiert, dass die Aktion eine Folge lokaler Transaktionen ist, und plant für die Momente dazwischen: die Saga. Eine dritte Antwort, oft die beste, besteht darin zu erkennen, dass Transaktionsgrenze und Servicegrenze nicht übereinstimmen, und eine von beiden zu verschieben. Dieser Abschnitt behandelt alle drei und endet mit einer Entscheidungshilfe.

Two-Phase Commit und XA

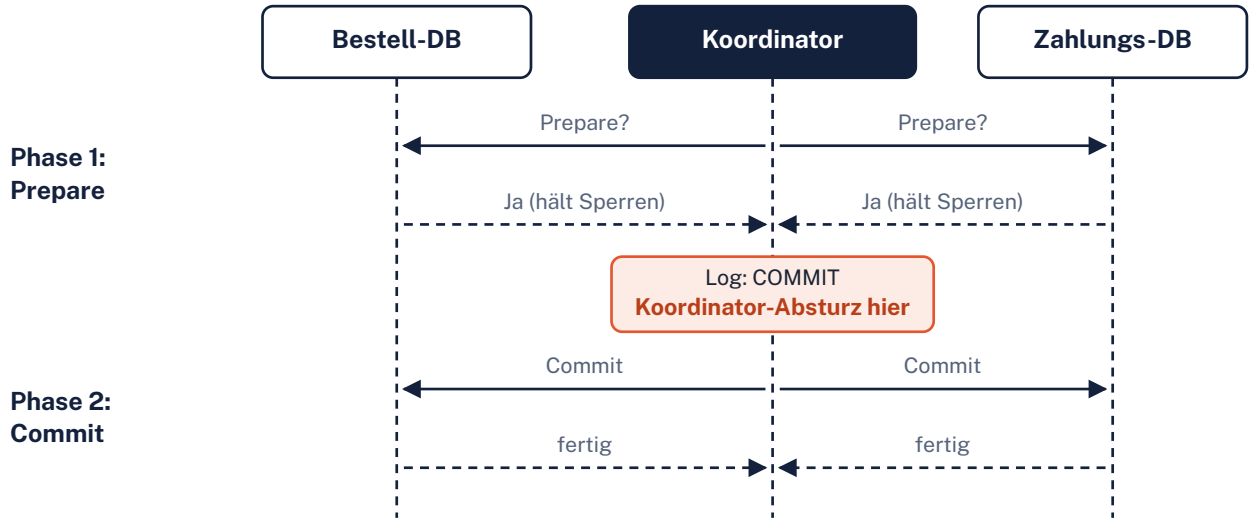
Two-Phase Commit (2PC) ist ein Protokoll aus den späten 1970er-Jahren, das Alles-oder-nichts-Atomarität über mehrere Teilnehmer hinweg herstellt, etwa zwei Datenbanken oder eine Datenbank und einen Message Broker. Ein **Koordinator** (der Transaktionsmanager) führt es aus. In **Phase 1, Prepare**, fragt der Koordinator jeden Teilnehmer, ob er committen kann. Ein Teilnehmer, der mit Ja antwortet, muss die Änderungen zuvor dauerhaft speichern und die betreffenden Sperren halten, denn mit seinem Ja gibt er das Recht auf, eigenständig abzubrechen. Lauten alle Stimmen Ja, schreibt der Koordinator seine Entscheidung, COMMIT,

in sein eigenes Log; lautet eine Stimme Nein oder antwortet ein Teilnehmer nicht, lautet die Entscheidung ABORT. In **Phase 2, Commit**, teilt er jedem Teilnehmer das Ergebnis mit, und diese wenden es an und geben ihre Sperren frei.

Die Schwäche: 2PC ist ein **blockierendes** Protokoll. Ein Teilnehmer, der mit Ja gestimmt hat, ist *im Ungewissen*: Er kennt das Ergebnis nicht und kann es nicht gefahrlos erraten, weil ein anderer Teilnehmer womöglich schon committet hat. Stürzt der Koordinator genau dann ab und bleibt ausgefallen, behalten die ungewissen Teilnehmer ihre Sperren, bis er sich erholt, und andere Transaktionen, die diese Zeilen berühren, warten ebenfalls. Der Betrieb kann eine *heuristische* Entscheidung erzwingen, um die Sperren freizugeben, riskiert damit aber, die Atomarität zu brechen.

Die XA-Spezifikation, 1991 von X/Open veröffentlicht, standardisierte die Schnittstelle zwischen einem Transaktionsmanager und Ressourcenmanagern, sodass beispielsweise ein Java-Anwendungsserver ein Datenbank-Update und das Senden einer Nachricht atomar committen kann. Viele relationale Datenbanken und einige Message Broker unterstützen sie.

Two-Phase Commit und wo es blockiert



Fällt der Koordinator nach den Stimmen aus, können Teilnehmer mit Ja nicht allein entscheiden und behalten ihre Sperren

Teilnehmer, die mit Ja gestimmt haben, halten ihre Sperren, bis sie die Entscheidung erfahren; fällt der Koordinator an der markierten Stelle aus, hängen sie fest.

Warum man 2PC zwischen Microservices meidet

2PC funktioniert und wird innerhalb der Grenzen eines einzelnen Systems weiterhin eingesetzt. Über unabhängig verantwortete Services hinweg passt es meist schlecht, aus Gründen, die sich gegenseitig verstärken.

Die Verfügbarkeit multipliziert sich nach unten. Jeder Teilnehmer und der Koordinator müssen laufen, damit eine Transaktion committen kann; die Verfügbarkeit der Transaktion ist also das Produkt ihrer Verfügbarkeiten, dieselbe Rechnung wie bei einer synchronen Aufrufkette (siehe Warum verteilen). **Sperren werden über das Netzwerk gehalten**, für mindestens zwei Roundtrips und viel länger, wenn irgendetwas langsam ist; das senkt den Durchsatz und erhöht die Konkurrenz um Ressourcen. **Die Kopplung kehrt zurück**: Alle Teilnehmer müssen dasselbe Protokoll sprechen, einen Transaktionsmanager teilen und sich auf Timeouts einigen, was die unabhängige Deploybarkeit untergräbt, die die Aufteilung der Services gerechtfertigt hat. **Viele Teilnehmer können gar nicht mitmachen**: Die meisten HTTP-APIs, Dienste Dritter wie Zahlungsanbieter, viele NoSQL-Datenbanken und mehrere verbreitete Broker unterstützen XA nicht. Und die API des Zahlungsanbieters wird niemals eine Sperre für Ihren Koordinator halten.

Innerhalb verteilter Datenbanken sieht das Bild anders aus. Systeme wie Google Spanner führen Two-Phase Commit intern zwischen Replikatgruppen aus, die ihrerseits per Konsens konsistent gehalten werden; ein abgestürzter Koordinator wird daher ersetzt, statt dass man auf ihn wartet. Das macht partitionsübergreifende Transaktionen innerhalb einer Datenbank praktikabel; Transaktionen über getrennt verantwortete Services mit ihren unterschiedlichen Technologien macht es nicht praktikabel.

Sagas: eine Folge lokaler Transaktionen

Hector Garcia-Molina und Kenneth Salem führten die Saga 1987 in einem SIGMOD-Paper über *langlebige Transaktionen* ein, also solche, die Datenbanksperren stundenlang halten würden. Ihre Idee war, eine solche Transaktion in eine Folge gewöhnlicher Transaktionen T_1, T_2 … T_n zu zerlegen, von denen jede für sich committet, und jeder eine kompensierende Transaktion C_i zur Seite zu stellen, die sie semantisch rückgängig macht. Das System garantiert dann, dass entweder die ganze Folge abgeschlossen wird oder auf die abgeschlossenen Schritte ihre Kompensationen in umgekehrter Reihenfolge folgen: T_1 … T_j, C_j … C_1.

Microservice-Architekturen übernahmen die Idee, weil sie genau zu ihren Randbedingungen passt: Jeder Schritt ist eine lokale ACID-Transaktion in der eigenen Datenbank eines Service, und die Schritte sind durch Nachrichten verbunden. Eine Saga liefert Atomarität im letztendlichen, fachlichen Sinn (alle Schritte finden statt, oder ihre Wirkungen werden rückgängig gemacht), aber keine Isolation: Zwischen den Schritten können andere Transaktionen den Zwischenzustand sehen.

Es hilft, die Schritte einzuordnen, wie Chris Richardson es in *Microservices Patterns* (2018) tut. **Kompensierbare Transaktionen** lassen sich durch eine Kompensation rückgängig machen. Die **Pivot-Transaktion** ist der Go/No-go-Punkt: Sobald sie committet, läuft die Saga bis zum Ende.

Wiederholbare Transaktionen folgen nach dem Pivot und müssen so entworfen sein, dass sie bei Wiederholung irgendwann gelingen. Schritte, die wahrscheinlich fehlschlagen, vor den Pivot zu legen und Schritte, die sich nicht rückgängig machen lassen, dahinter, ist eine zentrale Entwurfsentscheidung.

Choreografie oder Orchestrierung

Die Schritte einer Saga müssen irgendwie koordiniert werden, und die beiden Möglichkeiten entsprechen der Choreografie und Orchestrierung aus ereignisgesteuerte Architektur.

In einer **choreografiebasierten Saga** gibt es keinen Koordinator. Jeder Service lauscht auf Ereignisse und reagiert, indem er seine lokale Transaktion ausführt und ein neues Ereignis veröffentlicht: Der Bestellservice veröffentlicht `OrderCreated`, der Zahlungsservice reagiert und veröffentlicht `PaymentAuthorised`, der Dispositionsservice reagiert darauf und so weiter; Fehlerereignisse lösen Kompensationen in den Services aus, die auf sie lauschen. Für zwei oder drei Schritte ist das einfach und hält die Services lose gekoppelt. Mit wachsender Zahl der Schritte existiert der Ablauf nur noch implizit, verstreut über die Services, und ist daher schwer zu überblicken, zu testen und zu ändern; außerdem schleichen sich zyklische Ereignisabhängigkeiten zwischen Services ein.

In einer orchestrierungsbasierten Saga sendet ein einzelner Orchestrator, meist im Service, dem der Geschäftsprozess gehört, Commands an die Teilnehmer und reagiert auf ihre Antworten. Der Orchestrator ist ein persistenter Zustandsautomat: Er hält in seiner eigenen Datenbank fest, bei welchem Schritt er ist, und kann daher nach einem Absturz weitermachen. Der Ablauf ist explizit und an einer Stelle testbar, und die Teilnehmer müssen nichts voneinander wissen. Das Risiko besteht darin, dass sich im Orchestrator Geschäftslogik ansammelt, die in die Teilnehmer gehört. Workflow-Engines wie Temporal, Camunda und AWS Step Functions bieten dauerhafte Orchestrierung als Plattform, um den Preis einer weiteren Abhängigkeit. Bei mehr als etwa drei Schritten oder bei jedem Ablauf mit Verzweigungen und Timeouts bevorzugen die meisten Fachleute Orchestrierung.

Durable Execution: Temporal und Step Functions

Durable Execution bedeutet, dass ein einmal gestarteter Workflow trotz Abstürzen bis zum Ende läuft. In Temporal ist ein Workflow gewöhnlicher Code; was geschieht, wird in einer Ereignishistorie gespeichert, und nach einem Absturz spielt ein anderer Worker den Code gegen diese Historie erneut ab, um seinen Zustand wiederherzustellen und weiterzumachen. Workflow-Code muss daher deterministisch sein, und Arbeit nach außen wandert in *Activities*, die wiederholt werden können und deshalb Wiederholung vertragen sollten. AWS Step Functions definiert stattdessen einen Zustandsautomaten in der Amazon States Language; Standard-Workflows können bis zu einem Jahr laufen, Express-Workflows fünf Minuten. Das

Warten einer Bestell-Saga darauf, dass ein Restaurant annimmt, kann dann ein schlichter Timer sein. Die Kosten sind eine neue kritische Abhängigkeit und Sorgfalt, wenn der Code noch laufender Workflows geändert wird. Prüfen Sie die aktuellen Limits.

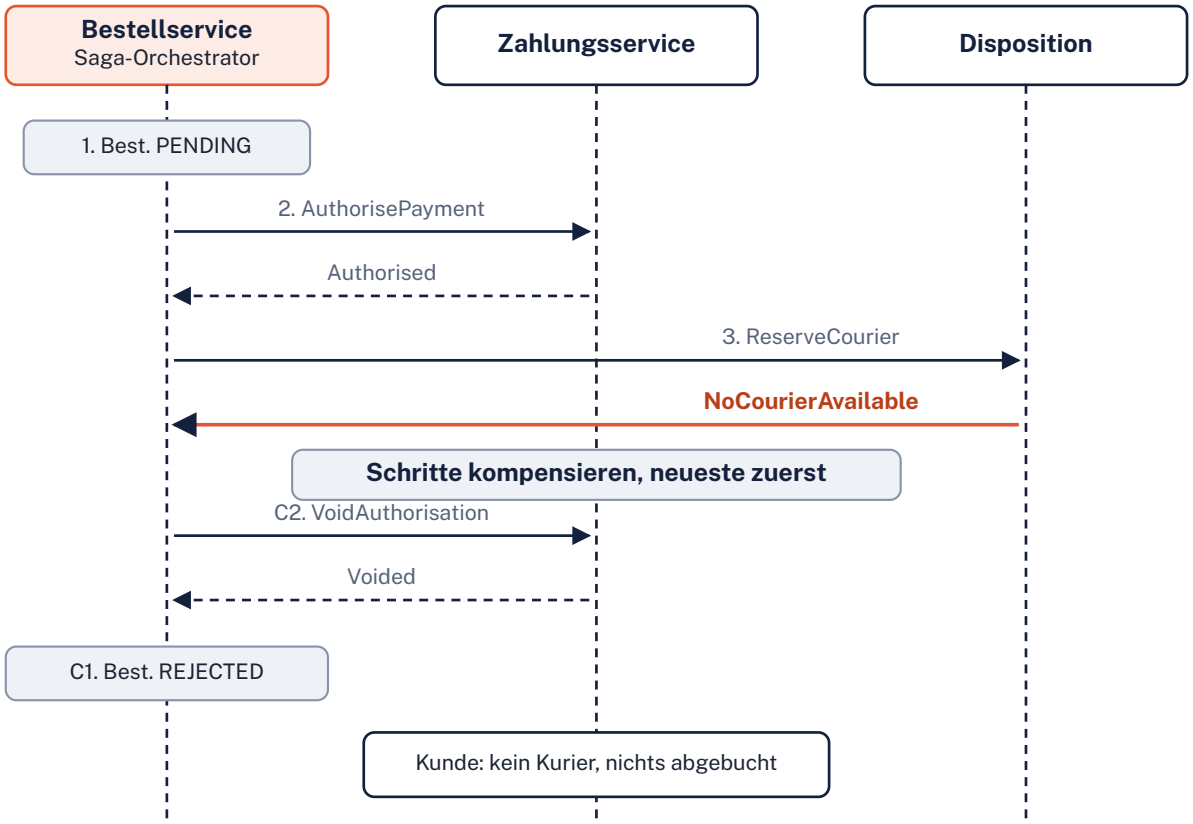
Ausgearbeitetes Beispiel: die Bestell-Saga

Die Bestell-Saga der Essenslieferplattform wird vom Bestellservice orchestriert. Die Reihenfolge der Schritte ist bewusst gewählt: Die Zahlungsautorisierung ist eine günstige Vormerkung, die sich leicht stornieren lässt, die Reservierung eines Kuriers ist der Schritt, der zu Stoßzeiten am ehesten scheitert, und die Bestätigung an das Restaurant ist der Pivot, denn sobald eine Küche mit dem Kochen begonnen hat, lässt sich die Bestellung nicht einfach rückgängig machen.

Die Bestell-Saga Schritt für Schritt

Schritt	Service und lokale Transaktion	Art	Kompensation, falls ein späterer Schritt scheitert
1. Bestellung anlegen	Der Bestellservice legt die Bestellung mit dem Zustand `PENDING` an (eine semantische Sperre).	Kompensierbar	C1: Zustand auf `REJECTED` setzen und den Grund festhalten. Die Zeile bleibt für die Prüfspur erhalten und wird nicht gelöscht.
2. Zahlung autorisieren	Der Zahlungsservice bittet den Kartenanbieter, den Gesamtbetrag vorzumerken, mit der Bestell-ID als Idempotenzschlüssel.	Kompensierbar	C2: die Autorisierung stornieren und die Vormerkung freigeben. Der Kunde sieht in seiner Banking-App eventuell kurz einen vorgemerkten Betrag.
3. Kurier reservieren	Der Dispositionsservice reserviert einen Kurier in der Nähe für das Abholfenster.	Kompensierbar	C3: den Kurier freigeben, damit er andere Aufträge übernehmen kann.
4. Bestellung bestätigen	Der Bestellservice setzt den Zustand auf `CONFIRMED` und sendet den Bon an das Restaurant.	Pivot	Keine: Ab hier muss die Saga vorwärtslaufen. Spätere Probleme werden durch neue Geschäftsprozesse behandelt (Erstattungen, Gutscheine).
5. Kunden benachrichtigen; Zahlung bei Abholung einziehen	Der Benachrichtigungsservice sendet „Bestellung bestätigt“; der Zahlungsservice zieht den vorgemerkten Betrag ein, wenn der Kurier das Essen abholt.	Wiederholbar	Keine: Diese Schritte werden idempotent wiederholt, bis sie gelingen oder an einen Menschen eskaliert werden.

Die Bestell-Saga, wenn kein Kurier verfügbar ist



Schritt 3 scheitert, also führt der Orchestrator die Kompensationen für die Schritte 2 und 1 aus, die neueste zuerst, und der Kunde wird informiert, bevor etwas gekocht oder abgebucht wurde.

Kompensation ist kein Rollback

Ein Datenbank-Rollback macht es so, als hätte die Transaktion nie stattgefunden: Niemand sonst hat ihre Änderungen je gesehen. Eine Kompensation ist eine neue, vorwärtsgerichtete Transaktion, die einer früheren, die *tatsächlich* stattgefunden hat und möglicherweise gesehen wurde, semantisch entgegenwirkt. In der Bestell-Saga zeigt die Bank des Kunden vielleicht schon eine Vormerkung an, und eine E-Mail hat womöglich „Zahlung autorisiert“ gemeldet. Die Kompensation kann die E-Mail nicht zurückholen; sie kann nur die Vormerkung stornieren und eine weitere Nachricht senden. Manche Aktionen lassen sich überhaupt nicht kompensieren, etwa bereits gekochtes Essen oder eine bereits gelesene Benachrichtigung; sie gehören hinter den Pivot oder brauchen eine fachliche Kompensation wie einen Gutschein oder eine Erstattung.

Kompensationen brauchen daher dieselbe Sorgfalt wie Vorwärtsschritte. Sie müssen **idempotent** sein, weil die auslösende Nachricht zweimal zugestellt werden kann. Sie müssen **wiederholbar sein, bis sie gelingen**, denn eine dauerhaft scheiternde Kompensation hinterlässt das System in einem Zustand, den niemand entworfen hat; wenn sie wirklich nicht gelingen können, müssen sie einen Alarm auslösen, den ein Mensch auflöst. Sie müssen eventuell **mit**

dem Vorwärtsschritt kommutativ sein: Trifft ein `VoidAuthorisation` vor dem `AuthorisePayment` ein, das es rückgängig macht (nach Timeouts und Wiederholungen möglich), sollte sich der Zahlungsservice die Stornierung merken und die verspätete Autorisierung ablehnen, statt zu scheitern. Und sie sollten für das Geschäft sichtbar sein: „abgelehnt, kein Kurier“ ist eine Information, die das Betriebsteam auf einem Dashboard sehen will.

Leben ohne Isolation: Anomalien und Gegenmaßnahmen

Weil jeder Schritt für sich committet, fehlt Sagas das I in ACID, und nebenläufige Sagas können sich gegenseitig stören. Richardson beschreibt drei daraus folgende Anomalien. Ein **Lost Update** (verlorenes Update) tritt auf, wenn eine Saga eine Änderung einer anderen überschreibt, ohne sie zu lesen, etwa wenn eine Stornierungs-Saga eine Bestellung als storniert markiert und die Bestell-Saga sie danach auf bestätigt setzt. Ein **Dirty Read** tritt auf, wenn eine Saga Daten liest, die eine andere Saga geändert hat, aber noch kompensieren könnte, etwa eine Kreditlimitprüfung, die eine Vormerkung mitzählt, die gleich storniert wird. Ein **Fuzzy oder Non-Repeatable Read** tritt auf, wenn zwei Schritte einer Saga dieselben Daten lesen und unterschiedliche Werte erhalten, weil eine andere Saga sie zwischendurch geändert hat.

Er katalogisiert außerdem Gegenmaßnahmen, Entwurfstechniken, die die Anomalien verringern, gestützt auf frühere Forschung zu Transaktionen über mehrere Datenbanken. Am weitesten verbreitet sind diese.

Gegenmaßnahmen gegen die fehlende Isolation in Sagas (nach Richardson)

Gegenmaßnahme	Idee	Beispiel Essenslieferung
Semantische Sperre	Ein kompensierbarer Schritt setzt ein Kennzeichen auf Anwendungsebene, meist einen Zustand `*_PENDING`, das anderen Transaktionen mitteilt, dass der Datensatz im Fluss ist; sie warten, scheitern oder behandeln ihn gesondert.	Solange eine Bestellung `PENDING` ist, wird ein Stornierungswunsch in eine Warteschlange gestellt oder mit „Bestellung wird bearbeitet“ abgelehnt, statt mittendrin angewendet zu werden.
Kommutative Updates	Updates so entwerfen, dass sie in jeder Reihenfolge dasselbe Ergebnis liefern, was Lost Updates beseitigt.	Das Verdienstkonto des Kuriers erfasst Gutschriften und Belastungen als getrennte Einträge, statt einen Saldo zu überschreiben.
Pessimistische Sicht	Schritte so umordnen, dass das geschäftliche Risiko eines Dirty Reads minimal wird, indem riskante Updates in wiederholbare Schritte nach dem Pivot gelegt werden.	Die „verfügbare Kapazität“ eines Restaurants nur in einem wiederholbaren Schritt erhöhen, nachdem die Stornierung feststeht, nie in einem Schritt, der kompensiert werden könnte.
Wert erneut lesen	Vor dem Update den Datensatz erneut lesen und prüfen, dass er sich seit dem Lesen nicht geändert hat (optimistisches Sperren); falls doch, abbrechen und die Saga eventuell neu starten.	Vor der Bestätigung prüft der Orchestrator, dass die Version der Bestellung noch die von ihm angelegte ist; hat eine Stornierung sie geändert, kompensiert er stattdessen.
Versionsdatei	Die auf einen Datensatz angewendeten Operationen festhalten, damit Operationen, die in falscher Reihenfolge eintreffen, umgeordnet oder erkannt werden können.	Der Zahlungsservice hält fest, dass eine Stornierung für eine Autorisierung eingetroffen ist, die er noch nicht gesehen hat, und lehnt die verspätete Autorisierung ab.
Nach Wert	Den Koordinationsmechanismus pro Anfrage nach ihrem geschäftlichen Risiko wählen.	Gewöhnliche Bestellungen nutzen die Saga; ein hochwertiger Catering-Auftrag eines Unternehmens könnte einen strengeren, langsameren Prozess durchlaufen.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: der Moment zwischen Speichern und Senden

Die Idee, die hier die meiste Sorgfalt verlangt, ist das Dual-Write-Problem und wie die Transactional Outbox es löst. Jeder Saga-Schritt speichert eine Änderung in seiner eigenen Datenbank und sendet eine Nachricht, damit der nächste Schritt beginnen kann. Das Problem liegt im Moment zwischen beidem.

- Der Bestellservice speichert Bestellung 1042 mit dem Zustand PENDING in seiner Datenbank. Als Nächstes soll er eine OrderCreated-Nachricht an den Broker senden, damit der Zahlungsservice beginnen kann.
- Angenommen, der Server stürzt nach dem Speichern und vor dem Senden ab. Die Bestellung existiert, aber die Nachricht ging nie hinaus; der Zahlungsservice erfährt also nie davon, und Bestellung 1042 wartet ewig.
- Andersherum ist es nicht besser: Sendet der Service zuerst und scheitert dann das Speichern, beginnt der Zahlungsservice mit der Arbeit an einer Bestellung, die es nicht gibt.
- In getrennten Schritten so an zwei Stellen zu schreiben, nennt man einen Dual Write: Es gibt immer einen Moment, in dem das eine geschehen ist und das andere nicht.
- Die Transactional Outbox beseitigt diesen Moment. Der Service speichert die Bestellung und die ausgehende Nachricht, als Zeile in einer Outbox-Tabelle, in einer einzigen Datenbank-Transaktion, sodass beides gespeichert wird oder keines. Ein separates Programm, das Relay, liest dann die Outbox und sendet jede Nachricht an den Broker.
- Stürzt das Relay ab, nachdem es eine Nachricht gesendet, aber bevor es das vermerkt hat, sendet es diese Nachricht erneut; jeder Empfänger muss daher mit Duplikaten zurechtkommen, wie das Ende dieses Abschnitts erklärt.
- **Die Lehre:** Ohne Two-Phase Commit, von dem dieser Abschnitt abrät, lassen sich eine Datenbank und ein Broker nicht wie eine Einheit aktualisieren. Speichern Sie die Nachricht stattdessen mit der Änderung, senden Sie sie danach, und nehmen Sie gelegentliche Duplikate in Kauf.

Zuverlässig veröffentlichen: die Transactional Outbox

Jeder Saga-Schritt tut zwei Dinge: Er aktualisiert seine eigene Datenbank und sendet eine Nachricht (ein Ereignis oder eine Antwort). Beides getrennt zu tun, erzeugt das Dual-Write-Problem. Committet der Service und stürzt dann vor dem Veröffentlichen ab, erfährt der Rest der Saga nie von dem Schritt. Veröffentlicht er zuerst und scheitert dann der Commit, handeln andere Services auf Grundlage von etwas, das nie geschehen ist. Datenbank und Broker in 2PC einzubinden, ist genau das, was wir vermeiden wollten.

Die Transactional Outbox löst das mit einer lokalen Transaktion. Der Service schreibt die fachliche Änderung und die ausgehende Nachricht, als Zeile in einer `outbox`-Tabelle, in derselben Datenbanktransaktion; entweder committen beide oder keines. Ein separates **Relay** liest dann die Outbox und veröffentlicht jede Nachricht beim Broker. Zwei Relay-Entwürfe sind verbreitet. Ein **Polling Publisher** fragt die Outbox-Tabelle in Abständen nach unveröffentlichten Zeilen ab und markiert sie als gesendet: einfach, erzeugt aber Last und Latenz. **Transaction Log Tailing** nutzt Change Data Capture (CDC): Ein Tool liest das datenbankeigene Replikationslog (das Write-Ahead-Log von PostgreSQL oder das Binary Log von MySQL) und macht aus neuen Outbox-Zeilen Nachrichten. Debezium, eine Open-Source-CDC-Plattform, die meist mit Kafka Connect betrieben wird, ist das bekannteste Beispiel und enthält einen Outbox Event Router, der jede Zeile anhand einer Spalte wie dem Aggregattyp an ein Topic leitet.

Das Relay kann nach dem Veröffentlichen und vor dem Vermerken abstürzen und veröffentlicht daher manchmal eine Nachricht zweimal. Die Outbox liefert also At-least-once-Zustellung, und Consumer müssen mit Duplikaten zurechtkommen. Die Reihenfolge der Veröffentlichung bleibt pro Aggregat erhalten, wenn das Relay die Zeilen in Commit-Reihenfolge veröffentlicht und die Aggregat-ID als Nachrichtenschlüssel verwendet.

```
BEGIN;

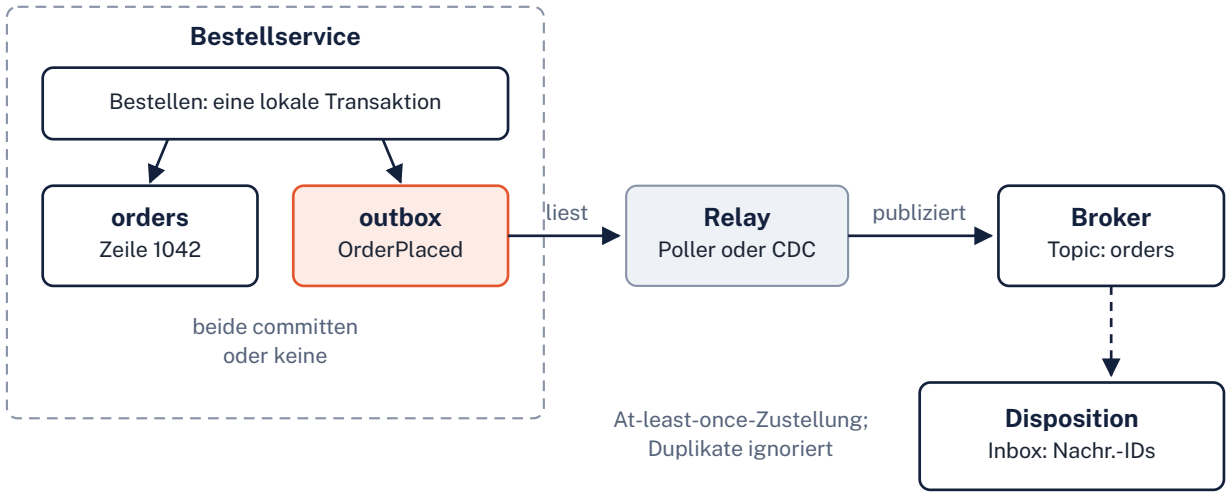
INSERT INTO orders (id, customer_id, state, total_pence)
VALUES ('1042', 'c-77', 'PENDING', 2350);

INSERT INTO outbox (id, aggregate_type, aggregate_id, type, payload)
VALUES ('e-9f1c', 'order', '1042', 'OrderCreated',
        '{"orderId":"1042","customerId":"c-77","totalPence":2350}');

COMMIT;
-- A relay (poller or CDC such as Debezium) later publishes e-9f1c to the broker.
```

Eine Bestellung mit Transactional Outbox aufgeben: Die Bestellung und ihr Ereignis werden gemeinsam in einer lokalen Transaktion committet.

Outbox auf dem Hinweg, Inbox auf dem Rückweg



Die Outbox beseitigt den Dual Write auf der Senderseite; die Inbox oder eine andere Duplikatprüfung macht die Empfängerseite bei At-least-once-Zustellung sicher.

Zuverlässig empfangen: idempotente Empfänger und die Inbox

Weil die Outbox, Wiederholungen des Brokers und Wiederholungen der Saga allesamt Duplikate erzeugen, muss jeder Teilnehmer ein idempotenter Empfänger sein, so der Name, den Gregor Hohpe und Bobby Woolf dem Muster in *Enterprise Integration Patterns* (2003) gaben. Manche Operationen sind von Natur aus idempotent („setze Bestellung 1042 auf CONFIRMED“); andere brauchen eine Duplikatprüfung. Die übliche Umsetzung ist eine **Inbox**: eine Tabelle verarbeiteter Nachrichten-IDs, geschrieben in derselben lokalen Transaktion wie die fachliche Änderung, sodass „die Nachricht verarbeiten“ und „sich merken, dass sie verarbeitet wurde“ gemeinsam committen. Eine erneut zugestellte Nachricht findet ihre ID bereits vor und wird bestätigt, ohne dass etwas geschieht. Der Code auf Consumer-Seite steht in ereignisgesteuerte Architektur.

Zusammen liefern Outbox, At-least-once-Zustellung und idempotente Empfänger das praktische Ergebnis, das meist gemeint ist, wenn von „Exactly-once“ die Rede ist: Jede fachliche Wirkung tritt einmal ein, auch wenn Nachrichten mehr als einmal unterwegs sind.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: eine Transaktion, eine Saga oder neue Grenzen?

Nutzen Sie diese Hilfe, wenn eine Geschäftsaktion Daten an mehr als einer Stelle ändern muss.

Q1 **Gehören alle Daten, die sich gemeinsam ändern müssen, einem Service und liegen sie in einer Datenbank?**

- **Ja** Verwenden Sie eine lokale ACID-Transaktion. Das ist die einfachste und stärkste Option; verteilen Sie sie nicht.
- **Nein** Weiter zur nächsten Frage.

Q2 **Ändern sich die Teile, die sich gemeinsam ändern müssen, fast immer gemeinsam, gehören sie einem Team und teilen sie einen Bounded Context, sodass die Aufteilung zufällig ist?**

- **Ja** Gestalten Sie die Grenze neu: Führen Sie die Services zusammen oder verschieben Sie die Daten, sodass die Aktion zu einer lokalen Transaktion wird. Sagas sind keine Reparatur für eine falsche Aufteilung.
- **Nein** Weiter zur nächsten Frage.

Q3 **Muss jeder Teilnehmer alle Änderungen im selben Augenblick sehen, ohne dass je ein Zwischenzustand sichtbar ist (zum Beispiel ein Hauptbuch, das immer ausgeglichen sein muss)?**

- **Ja** Halten Sie diese Datensätze in einer Datenbank, oder verwenden Sie eine verteilte Datenbank, die intern partitionsübergreifende Transaktionen bietet. Setzen Sie 2PC nur innerhalb der Grenzen eines Systems ein, mit Teilnehmern, die Sie kontrollieren.
- **Nein** Weiter zur nächsten Frage.

Q4 **Lässt sich jeder Schritt entweder kompensieren oder hinter einen Pivot legen und wiederholen, bis er gelingt?**

- **Ja** Verwenden Sie eine Saga: orchestriert, wenn sie mehr als etwa drei Schritte oder Verzweigungen hat, choreografiert, wenn sie kurz und stabil ist. Ergänzen Sie eine Outbox, idempotente Empfänger und Gegenmaßnahmen wie semantische Sperren.
- **Nein** Ordnen Sie die Schritte um oder ändern Sie den Geschäftsprozess (zum Beispiel reservieren, bevor belastet wird, oder eine manuelle Prüfung einfügen), bis es geht; andernfalls überdenken Sie die Grenzen.

ACHTUNG

Häufige Fehler

- Ein Ereignis nach dem Commit ohne Outbox veröffentlichen und Ereignisse verlieren, wann immer ein Service im falschen Moment neu startet.
- Kompensationen als „Zeile löschen“ schreiben, was die Prüfspur zerstört und jeden verwirrt, der sie schon gesehen hat.
- Vergessen, dass Kompensationen scheitern oder vor dem Schritt eintreffen können, den sie rückgängig machen.
- Einen unumkehrbaren Schritt, etwa das Senden der Bestellung an die Küche oder das Belasten der Karte, vor einen Schritt legen, der häufig scheitert.
- Choreografie für einen langen, verzweigten Ablauf verwenden, bis niemand mehr sagen kann, was passiert, wenn Schritt 5 scheitert.
- Sagas nutzen, um eine Serviceaufteilung zu übertünchen, die mitten durch eine Konsistenzgrenze schneidet; die günstigere Lösung ist oft, die Grenze zu verschieben.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum kann ein Teilnehmer an einem Two-Phase Commit nicht einfach selbst committen, wenn der Koordinator verschwindet, nachdem er mit Ja gestimmt hat?

Weil er die Entscheidung nicht kennt: Ein anderer Teilnehmer hat vielleicht mit Nein gestimmt, sodass der Koordinator womöglich den Abbruch beschlossen hat; Raten könnte die Atomarität brechen.

▼ Warum ist in der Bestell-Saga die Bestätigung an das Restaurant der Pivot und nicht der erste Schritt?

Sobald die Küche zu kochen beginnt, lässt sich der Schritt nicht rückgängig machen; er muss daher nach den Schritten kommen, die scheitern könnten (Zahlungsautorisierung, Kurierreservierung) und sich günstig kompensieren lassen.

▼ Welches Problem löst die Transactional Outbox, und welche Garantie gibt sie?

Sie beseitigt den Dual Write zwischen Datenbank und Broker, indem sie Änderung und Nachricht gemeinsam committet; das Relay stellt die Nachricht dann mindestens einmal zu, Duplikate sind also möglich.

▼ **Nennen Sie zwei Unterschiede zwischen einer Kompensation und einem Datenbank-Rollback.**

Sie ist eine neue Transaktion, die läuft, nachdem das Original committet wurde, und andere haben das Original womöglich schon gesehen oder darauf reagiert; außerdem ist sie unter Umständen nur ein semantisches Rückgängigmachen (eine Erstattung, eine Stornierung), statt den exakten vorherigen Zustand wiederherzustellen.

▼ **Probieren Sie es aus: Eine Ticket-Saga hat die Schritte: Zahlung autorisieren, Plätze reservieren, Ticket ausstellen, Fan per E-Mail benachrichtigen. In Schritt zwei stellt sich heraus, dass die Plätze vergeben sind. Was passiert? Welcher Schritt ist der Pivot, und warum ist eine Kompensation kein Rollback?**

Die Saga kompensiert in umgekehrter Reihenfolge: Sie storniert die Zahlungsautorisierung, und der Fan erfährt, dass die Plätze weg sind. Die Ausstellung des Tickets ist der Pivot, denn ein vorhandenes Ticket kann nicht einfach verschwinden, und spätere Schritte wie die E-Mail müssen wiederholt werden, bis sie gelingen. Eine Kompensation ist eine neue Vorwärtstransaktion, kein Rollback: Andere haben die Reservierung vielleicht gesehen, sie muss also idempotent und sicher wiederholbar sein.

QUELLEN

- Garcia-Molina and Salem: Sagas (Princeton technical report, 1987): https://www.cs.princeton.edu/research/techreps/598
- Chris Richardson: Pattern: Saga (microservices.io): https://microservices.io/patterns/data/saga.html
- Chris Richardson: Pattern: Transactional outbox (microservices.io): https://microservices.io/patterns/data/transactional-outbox.html
- Microsoft Azure Architecture Center: Saga design pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/saga
- Debezium documentation: Outbox event router: https://debezium.io/documentation/reference/transformations/outbox-event-router.html
- Temporal: Understanding Temporal: https://docs.temporal.io/evaluate/understanding-temporal
- Temporal: Workflows: https://docs.temporal.io/workflows
- AWS Step Functions: Standard and Express workflows: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html

Serverless und Space-Based Architecture

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.7 Server und die Cloud
- 6.1 Warum verteilen
- 6.4 Ereignisgesteuert

Begriffe von früher, die er verwendet: Managed Service, Idempotent, Teilausfall, Eventual Consistency, Read Replica

DIE IDEE IN KÜRZE

Denken Sie an zwei Arten, mit Kundschaft umzugehen. Ein Taxistand berechnet nur jede einzelne Fahrt: Wenn niemand eine Fahrt braucht, zahlen Sie nichts, aber in einer belebten Nacht kostet jede Fahrt mehr, als ein eigenes Auto kosten würde, und das erste Taxi braucht vielleicht einen Moment, bis es kommt. Das ist Serverless. Stellen Sie sich nun eine Stadionkasse vor einem riesigen Konzert vor: Statt dass jeder Kassierer ansteht, um ein zentrales Verzeichnis zu prüfen, führt jeder Schalter seine eigene Kopie der freien Plätze und meldet seine Verkäufe später an das Backoffice. Das ist Space-Based Architecture, gebaut für den Ansturm.

WAS SIE WISSEN MÜSSEN

- Serverless verbindet Function as a Service, bei dem der Anbieter ereignisgesteuerten Code ausführt und ihn ab null skaliert, mit Backend as a Service, bei dem verwaltete Dienste serverseitige Komponenten ersetzen; AWS Lambda (angekündigt 2014) machte FaaS zum Mainstream.
- Funktionen sind zustandslos und kurzlebig, werden pro Aufruf und pro GB-Sekunde abgerechnet, unterliegen Kaltstarts und harten Grenzen für Laufzeit, Arbeitsspeicher und Payload und werden oft mindestens einmal ausgelöst, daher müssen Handler idempotent sein.
- Jeder dauerhafte Zustand liegt in verwalteten Diensten, daher ist ein Serverless-System von seiner Bauart her ein verteiltes System und eng an seinen Anbieter gekoppelt, sofern die Geschäftslogik nicht hinter Ports gehalten wird.

- Preise pro Anfrage sind unterhalb eines Break-even-Volumens günstiger als dauerhaft laufende Kapazität und oberhalb davon teurer; der Break-even-Punkt hängt von Laufzeit, Arbeitsspeicher, Verkehrsprofil und Personalkosten ab.
- Space-Based Architecture bedient Anfragen aus Processing Units mit replizierten In-Memory Data Grids, verwaltet von virtualisierter Middleware, und aktualisiert die Datenbank asynchron über Data Pumps und Data Writer, während Data Reader das Grid vorwärmen.
- Ihr Name stammt vom Tuple Space aus Gelernters Linda (1985); ihre Hauptrisiken sind Datenkollisionen zwischen Replikaten, Speichergrenzen, Verzögerung der Datenbank, Kosten und Komplexität.
- Setzen Sie Space-Based Architecture nur bei extremer, sprunghafter Last ein, die Caching, Replikate und Warteräume nicht bewältigen, und routen oder partitionieren Sie umkämpfte Daten so, dass sie nicht kollidieren können.

Zwei Antworten auf die Kapazitätsfrage

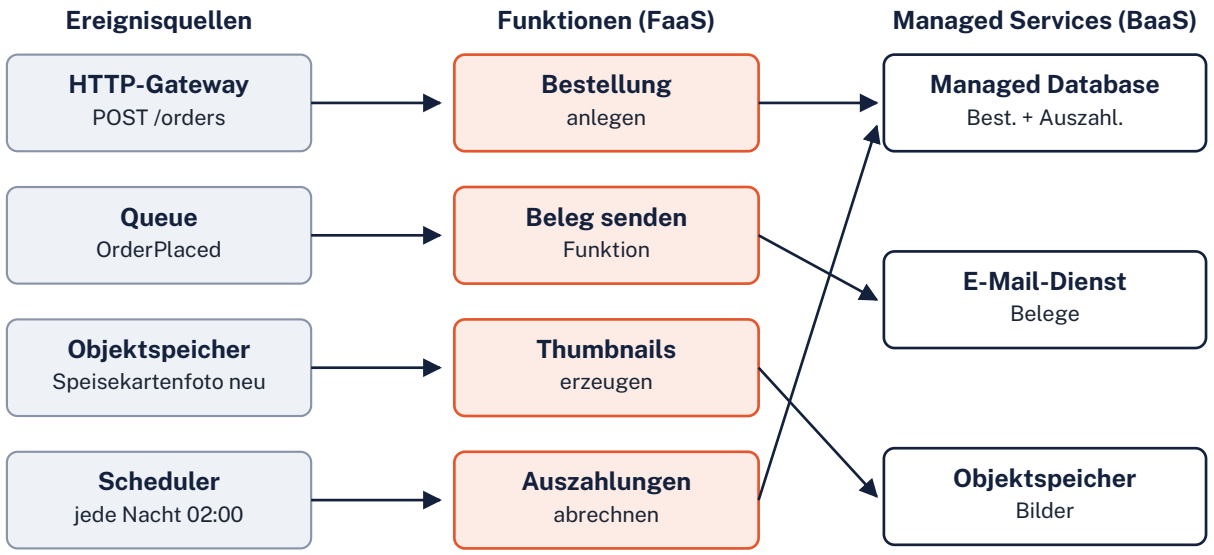
Jeder Stil in diesem Kapitel ging bisher davon aus, dass jemand Server betreibt und für die Last dimensioniert. Dieser Abschnitt behandelt zwei Stile, die die Kapazitätsfrage gegensätzlich beantworten. Serverless-Architektur überlässt die Server vollständig einem Cloud-Anbieter und zahlt pro Nutzung, was zu Arbeit passt, die klein, ereignisgesteuert und oft untätig ist. Space-Based Architecture hält alles, was eine stark belastete Anfrage braucht, im Arbeitsspeicher, verteilt auf viele identische Einheiten, sodass die Datenbank nicht mehr auf dem Hot Path liegt; das passt zu extremen und plötzlichen Spitzen wie einem Ticket-Vorverkaufsstart. Beide sind Spezialwerkzeuge; beide werden oft aus den falschen Gründen gewählt.

Serverless: Funktionen und verwaltete Backends

Serverless bedeutet nicht, dass es keine Server gibt; es bedeutet, dass das Team keine Server bereitstellt, patcht, dimensioniert oder für ungenutzte Server zahlt. Mike Roberts' viel zitierter Artikel von 2018 unterscheidet zwei Zutaten. Function as a Service (FaaS) führt kleine Stücke des eigenen Codes des Teams als Reaktion auf Ereignisse aus, in Containern, die der Anbieter erzeugt, skaliert und wieder abbaut und die womöglich nur für einen einzigen Aufruf existieren. AWS Lambda, im November 2014 angekündigt und seit April 2015 allgemein verfügbar, machte das Modell zum Mainstream; Azure Functions, Google Cloud Run functions und Cloudflare Workers sind weitere Beispiele. Backend as a Service (BaaS) ersetzt serverseitige Komponenten durch vollständig verwaltete Dienste Dritter, die der Client oder die Funktion direkt aufruft: Authentifizierung (Auth0, Amazon Cognito), Datenbanken (Firebase, DynamoDB), Objektspeicher, E-Mail und Queues.

Eine Serverless-Anwendung ist meist eine Kombination: Funktionen enthalten die Geschäftslogik, die für die Organisation spezifisch ist, und alles andere, einschließlich des gesamten dauerhaften Zustands, liegt in verwalteten Diensten. Der Anbieter kümmert sich um Kapazität, Verfügbarkeit über Zonen hinweg und das Betriebssystem. Das Team bleibt verantwortlich für seinen Code, seine Daten, seine Konfiguration und seine Sicherheitseinstellungen, wie das Modell der geteilten Verantwortung in Server und die Cloud erklärt.

Ereignisse lösen Funktionen aus; der Zustand liegt anderswo



Jede Funktionsinstanz ist zustandslos und kurzlebig; aller dauerhafte Zustand liegt in Managed Services

In einer Essenslieferplattform lösen vier Arten von Ereignissen vier Funktionen aus, die in Managed Services lesen und schreiben, statt selbst etwas zu behalten.

Wie Funktionen sich verhalten: die Eigenschaften, die den Entwurf prägen

Ereignisauslöser. Funktionen laufen als Reaktion auf Ereignisse: eine HTTP-Anfrage über ein Gateway, eine Nachricht in einer Queue oder einem Stream, eine Datei, die im Objektspeicher landet, eine Datenbankänderung oder einen Zeitplan. Auslöser, die die Funktion asynchron aufrufen, etwa Queues und Speicherbenachrichtigungen, stellen typischerweise *mindestens einmal* zu und wiederholen bei Fehlern; Handler müssen daher idempotent sein, genau wie in ereignisgesteuerte Architektur.

Auf null und in die Breite skalieren. Ohne Verkehr läuft keine Instanz, und für Rechenleistung wird nichts berechnet. Wenn Ereignisse eintreffen, startet die Plattform parallel weitere Instanzen, bis zu einem Konto-oder Funktionslimit für Nebenläufigkeit; bei AWS Lambda liegt das regionale Standardlimit zum Zeitpunkt der Erstellung bei 1000 gleichzeitigen

Ausführungen und lässt sich auf Antrag anheben. Schnelles Hochskalieren ist Feature und Gefahr zugleich: Tausend gleichzeitige Funktionen können tausend Datenbankverbindungen öffnen und eine relationale Datenbank überfordern, die für einige hundert ausgelegt ist; deshalb gibt es Verbindungsproxys und Nebenläufigkeitsgrenzen pro Funktion.

Abrechnung pro Aufruf. Die Kosten bestehen meist aus einer kleinen Gebühr pro Anfrage plus einer Gebühr für Arbeitsspeicher multipliziert mit der Ausführungszeit (Gigabyte-Sekunden). Code, der untätig auf eine langsame nachgelagerte API wartet, zahlt also fürs Warten.

Kaltstarts. Die erste Anfrage an eine neue Instanz muss warten, während die Plattform einen Container startet, die Laufzeitumgebung lädt und den Initialisierungscode ausführt: von einigen zehn Millisekunden bis zu mehreren Sekunden, je nach Sprache, Paketgröße und Netzwerkkonfiguration. Instanzen werden für spätere Anfragen wiederverwendet, solange sie warm sind; Kaltstarts treffen daher seltenen Verkehr und plötzliche Lastspitzen am härtesten. Gegenmittel sind kleinere Deployment-Pakete, leichtere Laufzeitumgebungen, Initialisierung außerhalb des Handlers und das Bezahlen dafür, Instanzen vorinitialisiert zu halten (AWS nennt das Provisioned Concurrency), was teilweise wieder Kosten für Dauerbetrieb einführt.

Grenzen. Funktionen laufen unter harten Grenzen für Ausführungszeit, Arbeitsspeicher, Payload-Größe und temporären Speicherplatz; für AWS Lambda zum Zeitpunkt der Erstellung 15 Minuten, 128 MB bis 10.240 MB Arbeitsspeicher (mit proportional zugeteilter CPU) und 6 MB für eine synchrone Anfrage oder Antwort. Lange Jobs müssen in Schritte zerlegt und große Payloads per Referenz auf den Objektspeicher übergeben werden.

Zustandslosigkeit. Nichts im Arbeitsspeicher oder auf der lokalen Festplatte einer Instanz überlebt garantiert bis zum nächsten Aufruf. Caching im Arbeitsspeicher zwischen Aufrufen ist eine Optimierung, die trifft oder auch nicht, nie ein Ort für Daten, auf die es ankommt.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: eine Funktion, die vergisst

Der Punkt, an dem hier die meisten stolpern, lautet: Eine Serverless-Funktion kann sich nicht darauf verlassen, irgendetwas von einem Lauf zum nächsten zu behalten; jede wichtige Information muss daher woanders liegen. Ein kleiner Fehler zeigt, warum.

- Das Team der Essenslieferplattform schreibt eine Funktion, die bei jeder aufgegebenen Bestellung läuft. Um eine Live-Zahl der heutigen Bestellungen anzuzeigen, hält ein Entwickler einen Zähler im Arbeitsspeicher der Funktion und erhöht ihn jedes Mal um eins.
- An einem ruhigen Morgen scheint das zu funktionieren: Eine Kopie der Funktion bearbeitet die Bestellungen nacheinander, und der Zähler geht 1, 2, 3.
- Zur Mittagszeit schnellen die Bestellungen hoch, und die Plattform startet 50 Kopien der Funktion auf einmal, jede mit eigenem Arbeitsspeicher. Jede Kopie führt ihren eigenen Zähler, sodass keine die echte Summe kennt.
- Wenn der Verkehr nachlässt, entfernt die Plattform stillschweigend die nicht mehr benötigten Kopien, und ihre Zählerstände verschwinden mit ihnen. Die erste Bestellung von morgen beginnt vielleicht wieder bei 1 oder bei dem, woran sich eine übrig gebliebene Kopie zufällig erinnert.
- Die Lösung: Den Zähler in einem Managed Service halten, den alle Kopien teilen, etwa einer Datenbank oder einem Cache, und dessen eigenen Befehl „eins addieren“ verwenden, damit sich zwei gleichzeitig addierende Kopien nicht gegenseitig überschreiben können.
- **Die Lehre:** Schreiben Sie jede Funktion zustandslos. Behandeln Sie alles in ihrem Arbeitsspeicher als Notizzettel, der jederzeit verschwinden kann, und legen Sie vor dem Schreiben der Funktion fest, wo jede Information liegt; darum geht es im nächsten Teil dieses Abschnitts.

Wohin der Zustand kommt

Weil Funktionen nichts behalten, verlagert sich die Architekturarbeit auf die Wahl, wo jede Art von Zustand liegt. Dauerhafte Geschäftsdaten kommen in eine verwaltete Datenbank, oft eine, die viele kurzlebige Verbindungen gut verkraftet (Key-Value- oder Dokumentspeicher oder eine relationale Datenbank hinter einem Verbindungsproxy). Große Objekte kommen in den Objektspeicher und werden zwischen Funktionen per Schlüssel übergeben. Kurzlebiger gemeinsamer Zustand, etwa Zähler für Rate Limiting, kommt in einen verwalteten Cache. Der Fortschritt eines mehrstufigen Prozesses kommt in einen Workflow-Dienst (AWS Step

Functions, Azure Durable Functions oder Ähnliches), der als dauerhafter Saga-Orchestrator fungiert, wie in Sagas beschrieben. Benutzersitzungen kommen in Tokens, die der Client hält, oder in einen Session Store.

Die Folge: Ein Serverless-System ist von seiner Bauart her ein verteiltes System, mit allen Anliegen dieses Kapitels: Netzwerklatenz zwischen jeder Funktion und ihren Daten, Teilausfälle, doppelte Ereignisse, Eventual Consistency und der Bedarf an verteiltem Tracing. Eine einzige Geschäftsaktion kann ein Gateway, drei Funktionen, eine Queue und zwei Datenbanken durchlaufen. Observability (siehe Observability) ist nicht optional.

Anbieterkopplung: der Preis dafür, keine Server zu betreiben

Serverless koppelt ein System enger an seinen Anbieter als das Mieten virtueller Maschinen oder der Betrieb von Containern. Ereignisformate, Auslöserkonfiguration, Identitäts- und Berechtigungsmodelle, Workflow-Definitionen und die verwalteten Dienste selbst sind allesamt anbieterspezifisch. Eine ausgereifte Serverless-Anwendung in eine andere Cloud zu bringen, ähnelt eher einer Neuentwicklung als einer Migration.

Fachleute sind uneins, wie sehr das zählt. Die eine Sicht lautet, Lock-in sei der Preis für eine große Entlastung im Betrieb, und die meisten Organisationen wechselten nie die Cloud, sodass die Optimierung auf Portabilität Aufwand verschwende. Die andere hält dagegen, dass Preisänderungen, regulatorische Anforderungen oder Übernahmen sehr wohl einen Wechsel erzwingen und dass tiefe Kopplung Verhandlungsmacht nimmt. Ein Mittelweg besteht darin, Kopplung an den Rändern zu akzeptieren und im Kern einzudämmen: Halten Sie die Geschäftslogik in einfachen Modulen hinter Ports, mit schlanken Funktions-Handlern als Adapter (der Ansatz der hexagonalen Architektur), pflegen Sie Infrastructure as Code, und halten Sie die Kopplung als explizite Entscheidung fest. Kosten und Lock-in werden in FinOps und Lock-in weiter erörtert.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Wann wird Dauerbetrieb günstiger?

Die Speisekarten-API der Essenslieferplattform könnte als Funktionen oder auf zwei dauerhaft laufenden virtuellen Maschinen laufen. **Alle Preise sind erfundene runde Zahlen, die der Rechnung dienen, keine aktuellen Listenpreise irgendeines Anbieters.** Jede Anfrage läuft im Schnitt 200 ms mit 0,5 GB Arbeitsspeicher und verbraucht also 0,1 GB-Sekunden.

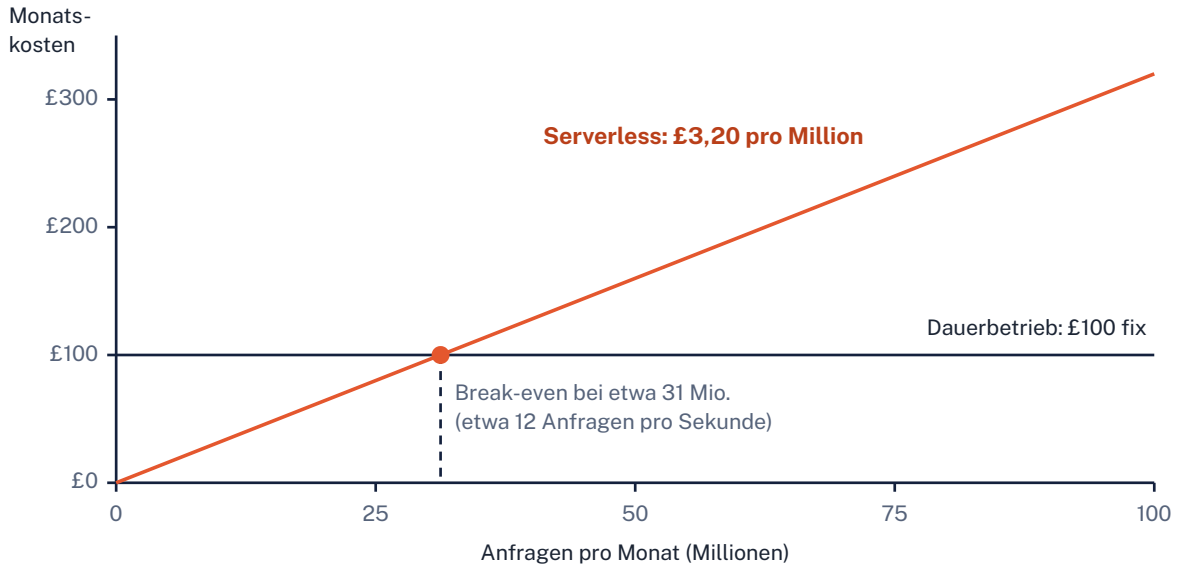
Beispielpreise Serverless	Rechenleistung £0,00002 pro GB-Sekunde; £0,20 pro Million Aufrufe; £1,00 pro Million Anfragen über das HTTP-Gateway.
Serverless-Kosten pro Million Anfragen	Rechenleistung: 1.000.000 × 0,1 GB-s × £0,00002 = £2,00. Aufrufe £0,20. Gateway £1,00. Insgesamt £3,20 pro Million.
Beispielkosten Dauerbetrieb	Zwei kleine VMs zu je £40 im Monat (zwei für Redundanz) plus ein Load Balancer zu £20 = £100 im Monat, dimensioniert für Spitzen von etwa 300 Anfragen pro Sekunde.
1 Million Anfragen im Monat (im Schnitt 0,4 pro Sekunde)	Serverless £3,20; Dauerbetrieb £100. Serverless ist etwa 30-mal günstiger.
10 Millionen im Monat (etwa 4 pro Sekunde)	Serverless £32; Dauerbetrieb £100.
Break-even	£100 ÷ £3,20 ≈ 31 Millionen Anfragen im Monat, im Schnitt etwa 12 Anfragen pro Sekunde (31,25 Millionen ÷ 2,59 Millionen Sekunden in 30 Tagen).
100 Millionen im Monat (etwa 39 pro Sekunde)	Serverless £320; Dauerbetrieb weiterhin £100, weil zwei VMs noch Reserven haben.
Wirkung einer Halbierung der Laufzeit auf 100 ms	Die Rechenkosten fallen auf £1,00, insgesamt £2,20 pro Million, und der Break-even verschiebt sich auf etwa 45 Millionen. Optimierung zahlt sich bei Serverless direkt aus.

Unterhalb des Break-even ist Serverless günstiger, weil Leerlauf nichts kostet; oberhalb zahlt ein gleichmäßig ausgelasteter Service mehr für dieselbe Arbeit. Stoßweiser Verkehr mit langen Leerlaufphasen spricht auch bei höheren Volumen für Serverless.

Der reine Vergleich lässt für viele Teams den größten Kostenpunkt weg: das Personal. Patchen, Skalieren und Rufbereitschaft für VMs können leicht mehr kosten als die Differenz, was den tatsächlichen Break-even nach oben verschiebt. Kosten für Datenübertragung, Logging und Observability fallen bei beiden an und sind oft höher als jede der beiden Rechenrechnungen.

Weil das Ergebnis von Laufzeit, Arbeitsspeicher, Verkehrsprofil und aktuellen Preisen abhängt, wiederholen Sie die Rechnung vor der Entscheidung mit echten Preisen und gemessenem Verkehr, und erneut, wenn der Verkehr wächst.

Der Break-even als Grafik



Mit diesen Beispielpreisen wachsen die Serverless-Kosten um £3,20 pro Million Anfragen und erreichen die fixen £100 des Dauerbetriebs bei etwa 31 Millionen Anfragen im Monat.

Wann Serverless passt und wann nicht

Serverless passt zu **sprunghaften, unvorhersehbaren oder geringvolumigen** Workloads, bei denen das Bezahlen ungenutzter Server Verschwendung ist; zu **ereignisgesteuertem Glue-Code**, etwa der Reaktion auf Uploads, Queue-Nachrichten und Datenbankänderungen; zu **geplanten Jobs** und Backoffice-Automatisierung; zu **Prototypen und neuen Produkten**, bei denen das Team seine Zeit in das Produkt statt in die Infrastruktur stecken sollte; und zu Teams ohne Plattform- oder Betriebskapazität.

Schlecht passt es zu **gleichmäßigem, hochvolumigem** Verkehr, bei dem die Preise pro Anfrage die dauerhaft laufende Kapazität übertreffen (wie im ausgearbeiteten Beispiel); zu **lang laufenden oder zustandsbehafteten** Prozessen, etwa Videotranscoding über das Zeitlimit hinaus oder WebSocket-Servern mit vielen Verbindungen; zu **strengen Anforderungen an niedrige Tail-Latenz**, die Kaltstarts verletzen würden; zu Workloads, die Spezialhardware oder feine Kontrolle über die Laufzeitumgebung brauchen; und zu feingranularen Entwürfen, in denen viele Funktionen einander synchron aufrufen, was bei jedem Schritt Netzwerk-Hops, Kosten und Fehlerquellen hinzufügt. Kapitel 1 beschrieb, wie Amazon Prime Video 2023 einen

Monitoring-Service von verteilten Serverless-Schritten auf einen einzigen Prozess umstellte und eine Senkung der Infrastrukturkosten um über 90 % meldete (Warum Architektur zählt): kein Urteil über Serverless, aber eine Erinnerung daran, dass Passung mehr zählt als Mode.

Space-Based Architecture: die Datenbank vom Hot Path nehmen

Die meisten Architekturen skalieren Web- und Anwendungsschicht problemlos, stoßen aber irgendwann an die Datenbank, die jede Anfrage berührt. Mehr Server davor verlagern die Warteschlange nur zur Datenbank. Space-Based Architecture, wie Mark Richards und Neal Ford sie in *Fundamentals of Software Architecture* beschreiben, nimmt die Datenbank ganz aus dem synchronen Pfad. Jede Anfrage wird vollständig im Arbeitsspeicher von einer von vielen identischen Processing Units bearbeitet, und die Datenbank wird asynchron im Hintergrund aktualisiert.

Der Name stammt vom Tuple Space, einem gemeinsamen assoziativen Speicher, den David Gelernters Koordinationssprache Linda 1985 einführte: Prozesse kommunizieren, indem sie Tupel in einen gemeinsamen Raum schreiben und passende Tupel lesen oder herausnehmen, ohne voneinander zu wissen. Suns JavaSpaces brachte die Idee Ende der 1990er-Jahre zu Java, und kommerzielle Plattformen bauten darauf verteilte, replizierte Spaces auf, aus denen sich die heutigen In-Memory Data Grids wie Hazelcast, Apache Ignite und Oracle Coherence entwickelten.

Die Teile eines Space-Based-Systems

1. **Processing Unit**
   Eine deploybare Einheit mit der Anwendungslogik und einem In-Memory Data Grid, das die von dieser Logik benötigten Daten hält, dazu eine Replikations-Engine. Die Einheiten sind identisch, also bringt jede zusätzliche Einheit mehr Kapazität. Im Ticketbeispiel hält jede Einheit den Saalplan und die aktuellen Reservierungen für die Veranstaltungen, die sie bedient.

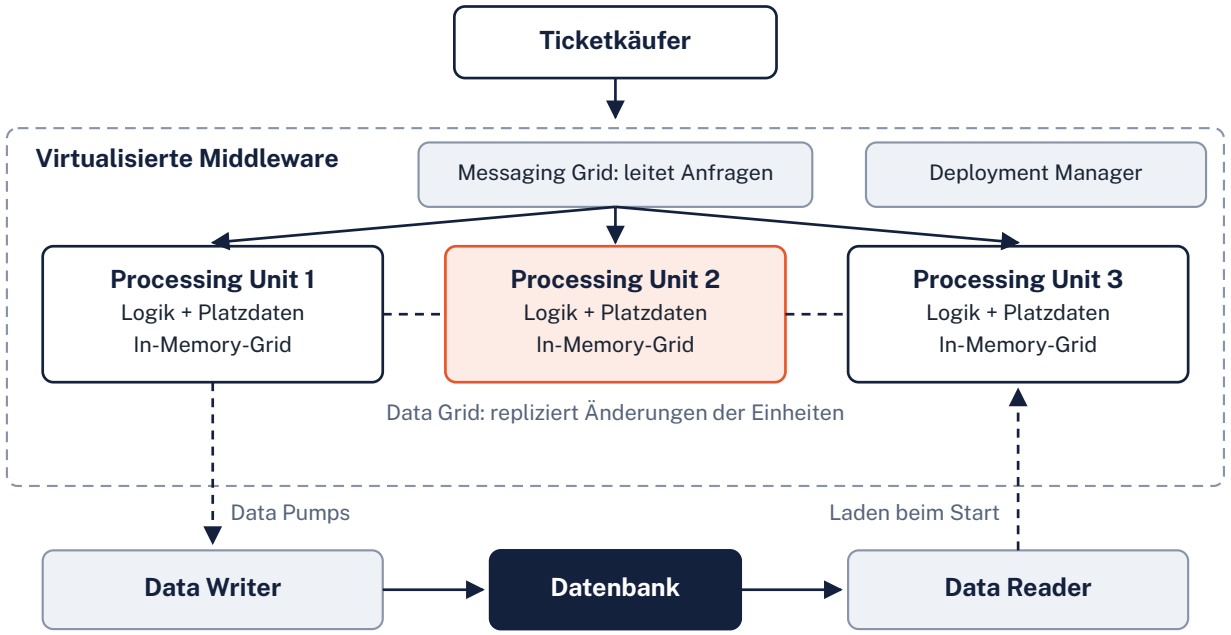
2 **Virtualisierte Middleware**
Die Infrastruktur, die die Einheiten verwaltet. Das
Messaging Grid
nimmt Anfragen entgegen und leitet jede an eine verfügbare Einheit. Das
Data Grid
repliziert Änderungen zwischen den In-Memory-Kopien der Einheiten. Das optionale
Processing Grid
koordiniert Anfragen, die mehrere Arten von Einheiten brauchen. Der
Deployment Manager
startet und stoppt Einheiten, wenn die Last steigt und fällt.

3 **Data Pumps**
Asynchrone Kanäle, meist persistente Message Queues, die jede in einer Einheit vorgenommene Änderung zur Datenbankseite transportieren. Ist die Datenbank langsam oder ausgefallen, warten die Änderungen in der Queue; die Käufer merken davon nichts.

4 **Data Writer**
Komponenten, die die Nachrichten der Pumps konsumieren und die Änderungen in die Datenbank schreiben. Sie lassen sich unabhängig von den Einheiten skalieren und pausieren.

5 **Data Reader**
Komponenten, die Daten aus der Datenbank in die Einheiten laden, wenn das Grid kalt startet, zum Beispiel nachdem jede Einheit neu gestartet oder neu deployt wurde. Bei der normalen Verarbeitung von Anfragen werden sie nicht verwendet.

Space-Based Architecture für einen Ticket-Vorverkaufsstart



Anfragen werden von identischen Processing Units aus dem Arbeitsspeicher bedient; die Datenbank wird asynchron über Data Pumps und Data Writer aktualisiert und nur gelesen, um das Grid vorzuwärmen.

Datenkollisionen und andere Kosten

Replizierte Kopien in vielen Einheiten bringen die Konsistenzprobleme aus den vorigen Abschnitten in den Arbeitsspeicher. Bei einem **replizierten** Cache hat jede Einheit eine vollständige Kopie, und Änderungen werden asynchron kopiert; das ergibt schnelle lokale Lesezugriffe, aber zwei Einheiten können dasselbe Element aktualisieren, bevor die Replikation nachgezogen hat. Richards und Ford nennen das eine Datenkollision und schätzen ihre Rate als Zahl der Einheiten mal Quadrat der Update-Rate, geteilt durch die Zahl der Elemente, mal Replikationslatenz. Bei 5 Einheiten, 20 Updates pro Sekunde, 50.000 Elementen und 100 ms Latenz sind das 5 × 400 ÷ 50.000 × 0,1 = 0,004 pro Sekunde, etwa 115 Kollisionen an einem Achtstundentag; eine Senkung der Latenz auf 1 ms bringt das auf etwa eine.

Für Daten, die niemals kollidieren dürfen, etwa den letzten Platz in einer Reihe, gibt es zwei Alternativen: einen **verteilten** Cache, in dem jedes Element in einer Partition liegt und jede Einheit es dort liest und schreibt (konsistent, aber jeder Zugriff ist ein Netzwerkaufruf, und die Partitionen des Grids müssen verfügbar bleiben), oder das Routing aller Anfragen für eine Veranstaltung oder einen Block an dieselbe Einheit, sodass nur eine einzige Einheit diese Plätze je ändert. Die meisten realen Entwürfe kombinieren beides: Sie partitionieren die heißen, umkämpften Daten nach Schlüssel und replizieren überwiegend gelesene Daten wie Saalpläne und Preise.

Die übrigen Kosten sind erheblich. Daten im Arbeitsspeicher sind durch dessen Größe begrenzt und müssen nach einem vollständigen Neustart wieder aufgebaut werden. Die Datenbank ist mit dem Grid nur letztendlich konsistent, sodass Reporting und andere Systeme Änderungen verspätet sehen, und der Verlust eingereihter Änderungen ist ein reales Risiko, das die Dauerhaftigkeit der Pumps abdecken muss. Tests unter realistischer Spitzenlast sind teuer und schwierig. Und der Stil ist komplex und kostspielig im Aufbau: Richards und Ford bewerten ihn niedrig bei Einfachheit und Testbarkeit und hoch bei den Kosten. Er lohnt sich nur, wenn sich die Last wirklich nicht mit Caching, Lesereplikaten und einer gut abgestimmten Datenbank bedienen lässt.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein Vorverkaufsstart im Stadion

Der Vorverkauf für eine Stadiontour beginnt um 10:00. Etwa 400.000 Menschen warten; in den ersten zehn Minuten liegt die Nachfrage beim Hundertfachen der normalen Rate und mehr, dann ebbt sie ab. Das Ticketsystem darf keinen Platz doppelt verkaufen und darf nicht zusammenbrechen.

Vor dem Vorverkaufsstart	Data Reader laden die Saalpläne jeder Vorstellung in das Grid. Der Deployment Manager startet genug Processing Units für die prognostizierte Spitze.
Routing	Das Messaging Grid leitet Anfragen nach Vorstellung und Block, sodass alle Reservierungen für Block 112 der Samstagsvorstellung von derselben Einheit bearbeitet werden. Platzreservierungen kollidieren nie.
Ein Käufer wählt zwei Plätze	Die Einheit markiert sie für diesen Käufer 8 Minuten lang als HELD, im Arbeitsspeicher, in Mikrosekunden, und antwortet. Die Saalpläne für andere Käufer werden über die Replikation aktualisiert.
Aus der Reservierung wird ein Verkauf	Der Zahlungsschritt läuft (ein externer Aufruf mit Idempotenzschlüssel); bei Erfolg markiert die Einheit die Plätze als SOLD, und eine Data Pump reiht den Verkauf ein.
Die Datenbank	Data Writer schreiben die Verkäufe in einem gleichmäßigen Tempo, das die Datenbank durchhält. Während der Spitze wächst die Queue; einige Minuten nach dem Nachlassen der Nachfrage ist sie abgearbeitet. Keine Käuferanfrage wartet auf die Datenbank.
Eine Reservierung läuft ab	Die Einheit gibt die Plätze im Arbeitsspeicher frei; die Pump reiht einen Freigabedatensatz ein.

Viele Ticketsysteme schalten zusätzlich einen virtuellen Warteraum vor den Verkauf, der Käufer in kontrolliertem Tempo einlässt. Das ist weit einfacher und reicht allein oft aus; Space-Based Architecture ist für den Fall gedacht, dass schon die eingelassene Last für einen datenbankzentrierten Entwurf zu hoch ist.

Das Routing nach Block ist die Entwurfsentscheidung, die das Kollisionsproblem für die wichtigsten Daten von „selten, aber möglich“ zu „unmöglich“ macht.

Eigenschaften im Vergleich (qualitativ, für typische Entwürfe)

Merkmal	Serverless (FaaS und BaaS)	Space-Based
Elastizität	Sehr hoch, automatisch, einschließlich Skalierung auf null	Sehr hoch bei Spitzen, aber Einheiten brauchen Zeit zum Starten und Laden der Daten
Performance	Gut im warmen Zustand; Kaltstarts und Netzwerk-Hops erhöhen die Latenz	Sehr hoch: Anfragen werden aus dem Arbeitsspeicher bedient
Kostenprofil	Zahlung pro Nutzung; hervorragend bei Leerlauf oder Spitzen, teuer bei gleichmäßiger Auslastung	Hoch: viele Einheiten, viel Arbeitsspeicher, komplexe Middleware, Dauerbetrieb
Einfachheit	Einfach pro Funktion; das Gesamtsystem kann unübersichtlich werden	Niedrig: Replikation, Kollisionen, Pumps und Laden beim Kaltstart
Konsistenz	Was die verwalteten Datenspeicher bieten	Die Replikation im Arbeitsspeicher ist letztendlich konsistent, sofern umkämpfte Daten nicht partitioniert sind; die Datenbank hinkt hinterher
Testbarkeit	Einheiten sind leicht zu testen; Integrationstests brauchen Cloud-Umgebungen	Schwierig: realistische Spitzenlasttests sind teuer
Betriebsaufwand	Niedrig für die Infrastruktur; hoch für Observability und Grenzen	Hoch
Übertragbarkeit	Niedrig: anbieterspezifische Auslöser und Dienste	Mittel: hängt vom Data-Grid-Produkt ab

Checkliste: bevor Sie sich für Serverless oder Space-Based entscheiden

Beantworten Sie diese Fragen, wo möglich, mit gemessenen Zahlen.

- [ ] **Verkehrsprofil**
  Ist die Last sprunghaft, über weite Strecken null oder gleichmäßig? Serverless belohnt Leerlauf; Space-Based belohnt extreme Spitzen; gleichmäßige Last spricht für gewöhnliche, dauerhaft laufende Services.

- [ ] **Kosten-Break-even**
  Haben Sie die Kosten pro Anfrage gegen die Kosten des Dauerbetriebs mit echten Preisen, Laufzeiten und Verkehrszahlen berechnet, einschließlich Gateway, Datenübertragung und Personal?

- [ ] **Latenzbudget**
  Verkraftet die langsamste akzeptable Antwort einen Kaltstart oder den Netzwerk-Hop zu dem Ort, an dem der Zustand liegt?

- [ ] **Grenzen**
  Überschreiten Jobs die Grenzen für Ausführungszeit, Arbeitsspeicher oder Payload? Lassen sie sich in Schritte zerlegen oder per Referenz übergeben?

- [ ] **Zustand und Verbindungen**
  Wo liegt jede Art von Zustand, und kommen die Datenspeicher mit plötzlichen Schüben gleichzeitiger Verbindungen zurecht?

- [ ] **Idempotenz**
  Kann jeder Handler gefahrlos zweimal für dasselbe Ereignis laufen?

- [ ] **Konkurrenz um Daten**
  Welche Daten sind bei Space-Based-Entwürfen umkämpft, und werden sie so partitioniert oder geroutet, dass sie nicht kollidieren können?

- [ ] **Dauerhaftigkeit der Verzögerung**
  Was geht verloren, wenn das Grid oder die Pumps ausfallen? Sind die Pumps persistent, und ist die Verzögerung der Datenbank für Reporting und andere Systeme akzeptabel?

- [ ] **Einfachere Alternativen**
  Würden Caching, Lesereplikate, eine Queue vor der Datenbank oder ein virtueller Warteraum den Bedarf mit weniger Komplexität decken?

- [ ] **Lock-in-Entscheidung**
  Ist die Kopplung an den Anbieter oder das Grid-Produkt als bewusste Entscheidung festgehalten, mit der Geschäftslogik hinter Ports?

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum muss eine von einer Queue ausgelöste Funktion idempotent sein?

Weil asynchrone Auslöser typischerweise mindestens einmal zustellen und bei Fehlern wiederholen, sodass dasselbe Ereignis die Funktion mehr als einmal aufrufen kann.

▼ Wie viel würden 50 Millionen Anfragen im Monat mit den Beispielpreisen aus dem ausgearbeiteten Beispiel bei Serverless ungefähr kosten, und welche Option ist günstiger?

50 × £3,20 = £160 im Monat; die Option im Dauerbetrieb zu £100 ist also bei den reinen Infrastrukturkosten günstiger, bevor Personal und weitere Gebühren mitgerechnet werden.

▼ Welche Aufgabe haben Data Reader bei Space-Based Architecture, und wann?

Sie laden Daten aus der Datenbank in die In-Memory-Grids der Processing Units, wenn das Grid kalt startet, etwa nach einem vollständigen Neustart oder Redeployment; für normale Anfragen werden sie nicht verwendet.

▼ Wie hilft es, alle Anfragen für einen Stadionblock an eine Processing Unit zu leiten?

Nur diese Einheit ändert je diese Plätze, sodass zwei Einheiten nicht denselben Platz aktualisieren können, bevor die Replikation nachgezogen hat; damit sind Datenkollisionen für diese Daten ausgeschlossen.

▼ Probieren Sie es aus: Beispielpreise: Rechenleistung £0,00002 pro GB-Sekunde, £0,20 pro Million Aufrufe, £1 pro Million Gateway-Anfragen. Jede Anfrage dauert 300 ms bei 1 GB. Zwei ständig laufende Server kosten £288 im Monat. Ermitteln Sie den Schnittpunkt. Was würden Sie bei 5 Millionen Anfragen wählen?

Jede Anfrage verbraucht 0,3 GB-Sekunden oder £6 pro Million, die Summe beträgt also £7,20 pro Million. Kreuzungspunkt: 288 / 7,20 = 40 Millionen Anfragen im Monat. Bei 5 Millionen kostet Serverless £36 gegenüber £288, wählen Sie also Funktionen. Auch Verkehrsprofil und Personalkosten zählen, und Kaltstarts können es ausschließen, wenn die Latenz knapp ist. Nähert sich das Volumen 40 Millionen, kehrt sich die Rechnung um, halten Sie die Logik also hinter Ports, um den Wechsel zu erleichtern.

QUELLEN

- Mike Roberts: Serverless Architectures (martinfowler.com, 2018): https://martinfowler.com/articles/serverless.html
- AWS: Introducing AWS Lambda (announcement, 13 November 2014): https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/
- AWS Lambda Developer Guide: Lambda quotas: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
- Adrian Colyer: Generative Communication in Linda (summary of Gelernter, 1985): https://blog.acolyer.org/2015/02/17/generative-communication-in-linda/

Weiterführende Literatur

- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. Klare Kapitel über servicebasierte, ereignisgesteuerte, Space-Based- und Microservice-Stile mit Bewertungen der Merkmale sowie die erweiterte Liste der Trugschlüsse verteilter Systeme.
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Analyse von Abwägungen für verteilte Systeme: Architekturquanten, Service-Granularität (Disintegratoren und Integratoren), Datenhoheit und verteilte Abläufe.
- *Building Microservices (2nd edition)*, Sam Newman (2021), O'Reilly Media. Der Standardleitfaden für die Praxis zu Microservices, aufgebaut um die unabhängige Deploybarkeit, mit ausgewogenem Rat dazu, wann man sie nicht einsetzen sollte.
- *Microservices*, James Lewis and Martin Fowler (2014), martinfowler.com. Der Artikel, der den Begriff für den Großteil der Branche definiert hat, mit seinen neun gemeinsamen Merkmalen.
  https://martinfowler.com/articles/microservices.html
- *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Gregor Hohpe and Bobby Woolf (2003), Addison-Wesley. Das Vokabular des Messaging (Kanäle, Event- und Command-Nachrichten, idempotente Empfänger, Dead-Letter-Kanäle), das jedes ereignisgesteuerte Design bis heute verwendet.
  https://www.enterpriseintegrationpatterns.com/
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. Die beste Erklärung von Replikation, Konsistenz, Logs, Stream-Verarbeitung und den Tücken verteilter Systeme.
  https://martin.kleppmann.com/2026/03/24/designing-data-intensive-applications-2e.html
- *Building Event-Driven Microservices (2nd edition)*, Adam Bellemare (2025), O'Reilly Media. Event-Streams als Rückgrat einer Service-Architektur: Event-Design, Schemas, Datenhoheit und Stream-Verarbeitung.

Glossar

Abwärtskompatible Änderung

Eine Änderung an einer Schnittstelle oder einem Schema, die bestehende Consumer ohne Anpassung überstehen, etwa das Hinzufügen eines optionalen Felds. Das Entfernen oder Umbenennen eines Felds ist ein Breaking Change.

API-Gateway
Ein einziger Einstiegspunkt vor den Services, der Querschnittsanliegen wie TLS, Authentifizierung, Rate Limiting und Routing übernimmt.

AsyncAPI
Eine maschinenlesbare Spezifikation zur Beschreibung nachrichtengesteuerter APIs: der Kanäle, der Operationen, die auf ihnen senden oder empfangen, und der Nachrichtenschemas.

Asynchrone Kommunikation
Eine Interaktion, bei der der Sender eine Nachricht an einen Vermittler wie einen Message Broker übergibt und weiterarbeitet, ohne zu warten; der Empfänger verarbeitet sie, wenn er bereit ist.

At-least-once-Zustellung
Eine Garantie, dass jede Nachricht verarbeitet wird, möglicherweise mehr als einmal; Duplikate muss der Empfänger behandeln.

Backend as a Service (BaaS)
Vollständig verwaltete Dienste von Drittanbietern, etwa Authentifizierung, Datenbanken, Speicher und Messaging, die serverseitige Komponenten ersetzen, die eine Organisation sonst selbst bauen und betreiben müsste.

Backend for Frontend (BFF)
Ein schlankes Backend, das für eine bestimmte Art von Client gebaut ist, etwa die mobile App, und Services aufruft und deren Daten für diesen Client aufbereitet.

Broker-Topologie
Eine ereignisgesteuerte Topologie ohne zentralen Koordinator: Prozessoren reagieren auf Ereignisse und veröffentlichen weitere Ereignisse, sodass sich der Ablauf aus ihren Reaktionen ergibt.

CAP-Theorem
Das Ergebnis, 2000 von Eric Brewer vermutet und 2002 von Gilbert und Lynch bewiesen, dass ein repliziertes System während einer Netzwerkpartition nicht zugleich linearisierbar und für jede Anfrage verfügbar sein kann.

Change Data Capture (CDC)
Das Auslesen des eigenen Änderungs- oder Replikationslogs einer Datenbank, um jedes festgeschriebene Insert, Update und Delete an andere Systeme zu streamen. Debezium ist eine weit verbreitete Open-Source-Plattform für CDC.

Choreografie
Die Koordination eines mehrstufigen Prozesses dadurch, dass die Beteiligten auf die Ereignisse der anderen reagieren, ohne dass eine einzelne Komponente die Steuerung übernimmt.

Command
Eine Nachricht, die einen bestimmten Handler auffordert, etwas zu tun, etwa AuthorisePayment. Anders als ein Ereignis kann sie abgelehnt werden.

Consumer Group
Eine Gruppe von Consumern, die sich die Arbeit des Lesens eines Topics teilen: Jede Partition wird von einem Mitglied gelesen, und jede Gruppe führt ihre eigenen Offsets.

Consumer-Driven Contract
Ein Vertrag, der sich daraus zusammensetzt, was jeder Consumer von der API eines Providers tatsächlich nutzt; der Provider führt die Erwartungen aller Consumer in seinem eigenen Build aus und weiß so, welche Änderungen wen beeinträchtigen würden.

Contract Testing
Das isolierte Testen jeder Seite einer Integration gegen eine gemeinsame Beschreibung ihrer Interaktionen, statt beide für End-to-End-Tests gemeinsam zu deployen.

CQRS
Command Query Responsibility Segregation: getrennte Modelle, oft auch getrennte Speicher, für das Ändern von Daten und für das Lesen.

Cursor-Paginierung
Das seitenweise Durchlaufen einer Sammlung, indem ein undurchsichtiges Token übergeben wird, das das zuletzt gesehene Element markiert, statt eines numerischen Offsets. Es bleibt schnell und stabil, wenn während des Blätterns Elemente hinzukommen.

Datenbank pro Service
Die Regel, dass jeder Service seine Daten besitzt und kein anderer Service sie direkt liest oder schreibt; andere nutzen die API des Eigentümers oder seine Ereignisse.

Datenkollision
In einem replizierten In-Memory-Cache aktualisieren zwei Einheiten denselben Eintrag, bevor die Replikation die erste Aktualisierung verbreitet hat, sodass eine Aktualisierung die andere überschreibt.

Domain Service
In einer servicebasierten Architektur ein separat deployter Service, der einen ganzen Geschäftsbereich wie Bestellungen oder Zahlungen abdeckt, von der API bis zum Datenzugriff.

Dual Write
Die Aktualisierung zweier Systeme, etwa einer Datenbank und eines Message Brokers, in getrennten Operationen; ein Absturz dazwischen lässt die beiden inkonsistent zurück.

Enterprise Service Bus (ESB)
Middleware im Zentrum klassischer SOA, die Nachrichten zwischen Services weiterleitet, transformiert und vermittelt und oft auch Geschäftsprozesse ausführt.

Ereignis (Event)
Eine unveränderliche Aufzeichnung über etwas, das geschehen ist, in der Vergangenheitsform benannt, etwa OrderPlaced. Beliebig viele Empfänger können darauf reagieren.

Ereignisgesteuerte Architektur

Ein Stil, bei dem Komponenten kommunizieren, indem sie Ereignisse über Geschehenes veröffentlichen und auf von anderen veröffentlichte Ereignisse reagieren, meist über einen Broker.

Event Notification

Ein Ereignis, das nur eine Änderung signalisiert und oft nur eine Kennung enthält, sodass Empfänger, die Details brauchen, beim Herausgeber nachfragen müssen.

Event-Carried State Transfer

Ereignisse, die genug Daten mitführen, damit Empfänger ihre eigene Kopie halten können und den Herausgeber nicht aufrufen müssen.

Event-Sourcing

Jede Änderung an einer Entität als Ereignis speichern, sodass das Ereignisprotokoll die maßgebliche Quelle ist und der aktuelle Zustand durch erneutes Abspielen abgeleitet wird.

Eventual Consistency

Eine Garantie, dass Kopien von Daten, die nicht mehr aktualisiert werden, nach einiger Zeit denselben Wert annehmen, während Leser zwischenzeitlich veraltete Werte sehen können.

Fallacies of Distributed Computing

Acht falsche Annahmen über Netzwerke, bei Sun Microsystems von L Peter Deutsch und Kollegen zusammengestellt, wobei James Gosling die achte ergänzte: zum Beispiel, dass das Netzwerk zuverlässig ist und die Latenz null beträgt.

GraphQL

Eine Abfragesprache und Laufzeitumgebung für APIs, 2015 von Facebook veröffentlicht, bei der der Client eine Abfrage sendet, die genau die gewünschten Felder aus einem typisierten Schema benennt, und eine Antwort derselben Form erhält.

gRPC

Ein Framework für Remote Procedure Calls, 2015 von Google als Open Source veröffentlicht, das Protocol-Buffers-Nachrichten über HTTP/2 sendet und typisierten Client- und Server-Code erzeugt. Es unterstützt Streaming in eine oder beide Richtungen.

Hypermedia Controls (HATEOAS)

Links in einer API-Antwort, die dem Client sagen, welche Aktionen er als Nächstes ausführen kann, sodass der Client Links folgt, statt URLs selbst zusammenzusetzen. Stufe 3 des Richardson Maturity Model.

Idempotenter Consumer

Ein Nachrichtenempfänger, der in derselben Transaktion wie ihre Auswirkungen festhält, welche Nachrichten er verarbeitet hat, sodass Duplikate nichts verändern.

Idempotenz
Die Eigenschaft einer Operation, dieselbe Wirkung zu haben, ob sie einmal oder mehrmals ausgeführt wird, was Wiederholungen sicher macht. Wird meist mit einer eindeutigen Anfrage- oder Ereigniskennung erreicht, die sich der Empfänger merkt.

Idempotenzschlüssel
Ein eindeutiger Wert, den ein Client mit einer nicht idempotenten Anfrage wie POST sendet, damit der Server eine Wiederholung derselben Operation erkennen und das ursprüngliche Ergebnis zurückgeben kann, statt die Arbeit doppelt zu erledigen.

In-Memory Data Grid
Software, die Daten im Arbeitsspeicher vieler Maschinen als einen logischen Speicher hält, mit Partitionierung und Replikation, für sehr schnellen Zugriff. Hazelcast, Apache Ignite und Oracle Coherence sind Beispiele.

Kaltstart (Cold Start)
Die zusätzliche Verzögerung, wenn eine Serverless-Plattform eine neue Funktionsinstanz erzeugen und initialisieren muss, bevor sie eine Anfrage bearbeitet, von einigen zehn Millisekunden bis zu mehreren Sekunden.

Kanonisches Datenmodell
Eine einzige unternehmensweite Darstellung jeder Geschäftsentität, in die und aus der alle Nachrichten übersetzt werden. Passt selten gut zu allen Teilen eines Unternehmens.

Kaskadierender Ausfall
Ein Ausfall, der sich ausbreitet: Eine langsame oder ausgefallene Komponente lässt ihre Aufrufer warten und deren eigene Ressourcen erschöpfen, sodass auch sie ausfallen, und so weiter die Kette hinauf.

Kausale Konsistenz
Ein Konsistenzmodell, bei dem kausal zusammenhängende Operationen (eine könnte die andere beeinflusst haben) von allen in derselben Reihenfolge gesehen werden; nicht zusammenhängende nebenläufige Operationen können in unterschiedlicher Reihenfolge erscheinen.

Kompensierende Transaktion
Eine neue Transaktion, die die Wirkung einer früheren, bereits festgeschriebenen Transaktion inhaltlich aufhebt, etwa die Stornierung einer Kartenautorisierung. Anders als bei einem Rollback kann das Original bereits gesehen worden sein.

Konsistenzmodell
Der Vertrag, den ein Datenspeicher darüber anbietet, welche Werte Lesezugriffe angesichts der erfolgten Schreibzugriffe zurückgeben dürfen, und damit, auf welche veralteten oder umgeordneten Ergebnisse Clients vorbereitet sein müssen.

Last Write Wins
Eine Regel zur Konfliktauflösung, die den Schreibvorgang mit dem jüngsten Zeitstempel behält und die anderen verwirft. Einfach, aber gleichzeitige Änderungen gehen stillschweigend verloren, und Uhrabweichungen können den falschen Gewinner bestimmen.

Linearisierbarkeit
Das stärkste Konsistenzmodell für einzelne Objekte: Jede Operation scheint zu einem einzigen Zeitpunkt zwischen ihrem Beginn und ihrem Ende stattzufinden, in einer Reihenfolge, die alle Clients teilen und die die reale Zeit respektiert, als gäbe es nur eine Kopie.

Mediator-Topologie
Eine ereignisgesteuerte Topologie, in der ein Mediator ein auslösendes Ereignis empfängt und den Workflow steuert, indem er in festgelegter Reihenfolge Commands an Prozessoren sendet.

Microservices
Ein Stil, bei dem eine Anwendung aus kleinen Services aufgebaut wird, die jeweils in einem eigenen Prozess laufen, um eine fachliche Fähigkeit herum geschnitten, unabhängig deploybar sind und über leichtgewichtige Mechanismen kommunizieren.

Monotonic Reads
Eine Garantie, dass die späteren Lesevorgänge eines Clients, sobald er einen Wert gesehen hat, nie einen älteren zurückgeben, sodass Daten nie in der Zeit zurückzuspringen scheinen.

N+1-Abfrageproblem
Ein Performance-Fehler, bei dem Code eine Abfrage stellt, um eine Liste von N Elementen zu holen, und dann pro Element eine weitere Abfrage, was N+1 Roundtrips ergibt, wo ein oder zwei genügen würden.

Nachricht (Message)
Beliebige Daten, die ein Programm über einen Kanal an ein anderes sendet; sie können ein Ereignis, einen Command, eine Abfrage oder eine Antwort enthalten.

Netzwerkpartition
Ein Netzwerkfehler, der ein System in Gruppen von Knoten aufspaltet, die nicht miteinander kommunizieren können, obwohl jede Gruppe weiterlaufen und für manche Clients erreichbar sein kann.

Offset
Die fortlaufende Position eines Ereignisses innerhalb einer Partition; Consumer speichern den erreichten Offset, damit sie nach einem Neustart dort weitermachen können.

OpenAPI
Ein standardisiertes, maschinenlesbares Format zur Beschreibung von HTTP-APIs (Pfade, Operationen, Parameter, Schemas), früher die Swagger-Spezifikation. Es dient als Grundlage für Dokumentation, Codegenerierung und Tests.

Orchestrierung
Die Koordination eines mehrstufigen Prozesses durch eine Komponente, die jedem Beteiligten sagt, was zu tun ist, und entscheidet, was als Nächstes geschieht.

PACELC
Daniel Abadis Erweiterung des CAP-Theorems: Bei einer Partition (P) ist zwischen Verfügbarkeit (A) und Konsistenz (C) abzuwägen; andernfalls (E, „Else“), im Normalbetrieb, zwischen Latenz (L) und Konsistenz (C).

Partition (eines Topics)
In Kafka und ähnlichen Logs ein geordneter Teil eines Topics, an den nur angehängt wird. Events mit demselben Schlüssel landen in derselben Partition; die Reihenfolge ist nur innerhalb einer Partition garantiert.

Problem Details (RFC 9457)
Ein Standard-JSON-Format für Fehler von HTTP-APIs mit den Membern type, title, status, detail und instance, ausgeliefert als application/problem+json. RFC 9457 (2023) hat RFC 7807 abgelöst.

Protocol Buffers
Ein sprachneutrales, kompaktes Binärformat für strukturierte Daten, definiert in .proto-Dateien, in denen jedes Feld eine dauerhafte Nummer hat. Wird von gRPC und vielen Messaging-Systemen verwendet.

Publish-Subscribe
Ein Messaging-Muster, bei dem ein Publisher eine Nachricht an ein Topic sendet und jeder Subscriber seine eigene Kopie erhält, ohne dass der Publisher weiß, wer sie sind.

Queue (Punkt-zu-Punkt-Kanal)
Ein Messaging-Kanal, bei dem jede Nachricht an genau einen der lesenden Consumer zugestellt wird, sodass sich mehrere Instanzen die Arbeit teilen können.

Quorum (R + W > N)
In einem Speicher mit N Replikaten wartet ein Schreibvorgang auf W Bestätigungen und ein Lesevorgang auf R Antworten; ist R + W > N, überschneidet sich jede Lesemenge mit jeder Schreibmenge in mindestens einem Replikat.

Read-your-Writes
Eine Garantie, dass ein Client, nachdem er einen Wert geschrieben hat, bei seinen eigenen späteren Lesevorgängen diesen Schreibvorgang oder einen neueren Stand sieht, auch wenn andere Clients das noch nicht tun.

Saga
Eine Geschäftstransaktion, umgesetzt als Folge lokaler Transaktionen, jede gepaart mit einer kompensierenden Transaktion, sodass entweder alle Schritte abgeschlossen oder bereits abgeschlossene Schritte semantisch rückgängig gemacht werden. 1987 von Garcia-Molina und Salem eingeführt.

Saga-Gegenmaßnahme
Eine Entwurfstechnik, die Anomalien verringert, die durch die fehlende Isolation von Sagas entstehen, etwa ein semantisches Lock (ein Zustand PENDING), kommutative Änderungen oder das erneute Lesen eines Werts vor seiner Änderung.

Saga-Orchestrator
Eine Komponente, die eine Saga koordiniert, indem sie Commands an die Beteiligten sendet, den Fortschritt in einem persistenten Zustandsautomaten verfolgt und bei Fehlern Kompensationen auslöst.

Sequenzielle Konsistenz
Ein Konsistenzmodell, bei dem alle Clients die Operationen in einer gemeinsamen Reihenfolge sehen, die die Reihenfolge jedes einzelnen Clients bewahrt, aber nicht der realen Zeit entsprechen muss.

Service Discovery
Ein Mechanismus, etwa eine Registry oder DNS, über den Clients die aktuellen Netzwerkadressen funktionsfähiger Service-Instanzen finden.

Service Mesh
Eine zentral konfigurierte Infrastrukturschicht aus Netzwerk-Proxys, die Verschlüsselung zwischen Services, Wiederholungen, Timeouts, Traffic-Steuerung und Telemetrie bereitstellt.

Servicebasierte Architektur
Ein verteilter Stil mit wenigen grobgranularen, separat deployten Domain-Services (typischerweise 4 bis 12), die sich oft eine Datenbank teilen; von Mark Richards und Neal Ford benannt.

Serviceorientierte Architektur (SOA)
Ein Ansatz, der ab Anfang der 2000er-Jahre in Unternehmen vorherrschte und fachliche Fähigkeiten als gemeinsam genutzte Services mit expliziten Verträgen bereitstellt, typischerweise über einen Enterprise Service Bus verbunden.

Session Guarantees
Konsistenzzusagen pro Client, 1994 von Terry und Kollegen beschrieben: Read-your-Writes, Monotonic Reads, Monotonic Writes und Writes-follow-Reads.

Space-Based Architecture
Ein Architekturstil, der Anfragen über viele identische Verarbeitungseinheiten bedient, die Daten in replizierten In-Memory-Grids halten und die Datenbank asynchron aktualisieren, sodass sie aus dem Anfragepfad herausgenommen ist.

Synchrone Kommunikation
Eine Interaktion, bei der der Aufrufer eine Anfrage sendet und auf die Antwort wartet, bevor er fortfährt, wie bei einem Telefonat. Beide Seiten müssen im selben Moment verfügbar sein.

Teilausfall
Eine Situation, in der einige Teile eines verteilten Systems funktionieren, während andere ausgefallen, langsam oder nicht erreichbar sind, oft ohne dass die funktionierenden Teile erkennen können, welche.

Transactional Outbox
Ein Muster, bei dem ein Service ausgehende Nachrichten in derselben lokalen Transaktion wie seine fachliche Änderung in eine Outbox-Tabelle schreibt; ein Relay veröffentlicht sie später und erreicht so At-least-once-Zustellung ohne verteilte Transaktion.

Tuple Space
Ein gemeinsamer assoziativer Speicher, in den Prozesse Tupel schreiben und aus dem sie passende Tupel lesen oder entnehmen, ohne voneinander zu wissen. Eingeführt mit David Gelernters Sprache Linda (1985).

Two-Phase Commit (2PC)
Ein Protokoll, das mehrere Beteiligte gemeinsam committen oder abbrechen lässt: Ein Koordinator fordert zunächst alle auf, sich vorzubereiten und abzustimmen, und teilt dann allen das Ergebnis mit. Beteiligte, die mit Ja gestimmt haben, blockieren, wenn der Koordinator ausfällt.

Unabhängige Deploybarkeit
Die Fähigkeit, einen Service zu ändern und in Produktion zu releasen, ohne gleichzeitig einen anderen Service ändern, testen oder releasen zu müssen.

Verteiltes System
Eine Menge von Programmen auf verschiedenen Computern, die zusammenarbeiten, indem sie Nachrichten über ein Netzwerk senden, sodass Benutzer einen einzigen Dienst sehen.

XA
Die X/Open-Standardschnittstelle (1991) zwischen einem Transaktionsmanager und Ressourcenmanagern wie Datenbanken und Message Brokern, mit der ein Two-Phase Commit über sie hinweg ausgeführt wird.