BE SOFTWARE ARCHITECTURE WISE · KAPITEL 9

Performance, Skalierbarkeit, Resilienz und Betreibbarkeit

Dieses Kapitel macht aus den Performance- und Verfügbarkeitsszenarien aus Kapitel 2 funktionierende Systeme. Es behandelt Scale-up und Scale-out und den Scale Cube, Load Balancing und zustandslose Services, Queues, Backpressure und Rate Limiting, die Resilienzmuster, die die Ausbreitung von Ausfällen verhindern (Timeouts, Wiederholungen, Circuit Breaker, Bulkheads), SLIs, SLOs und Error Budgets, Observability mit Logs, Metriken und Traces, Kapazitätsplanung und Performancetests sowie Chaos Engineering und das Lernen aus Incidents. Es richtet sich an Entwicklungsteams und Architekten, die die verteilten Stile aus Kapitel 6 kennen und ein System unter realer Last schnell und lauffähig halten müssen.

bewiseapps.com/de/software-architecture-wise/resilience-and-operability/ · Version 2026-10-26. Nur Bildungsinformation, keine professionelle Beratung.

AUF EINEN BLICK

Das lernen Sie

- Zwischen vertikaler und horizontaler Skalierung und zwischen den X-, Y- und Z-Achsen des Scale Cube wählen und eine Flotte mit Littles Gesetz und Kapazitätsreserve für eine bekannte Spitzenlast dimensionieren.
- Autoscaling-Signale und -Einstellungen wählen, die Verzögerung und Flapping vermeiden, und erklären, warum die CPU oft die falsche Metrik ist.
- Load Balancing auf Layer 4 und Layer 7 sowie die wichtigsten Algorithmen vergleichen, Health Checks und Connection Draining entwerfen und den Session-Zustand auslagern.
- Queues, begrenzte Puffer, Backpressure, Load Shedding und Rate Limiting (einschließlich Token Bucket) einsetzen, um Überlast unter Kontrolle zu halten.
- Timeouts, Deadlines, Wiederholungen mit Backoff und Jitter, Retry-Budgets, Circuit Breaker, Bulkheads und Fallbacks kombinieren, um Ausfälle einzudämmen.
- SLIs und SLOs aus Sicht der Nutzer definieren, Error Budgets und Burn Rates berechnen und darauf basierende Alerts entwerfen.
- Ein System mit OpenTelemetry durch Logs, Metriken und Traces instrumentieren und Probleme mit der RED- und der USE-Methode diagnostizieren.
- Kapazitäts- und Performancetests planen, Chaos-Experimente sicher durchführen und schuldfreie Post-Incident-Reviews leiten.

Bevor Sie beginnen

Kapitel 2, besonders die Abschnitte über Performance und Skalierbarkeit sowie über Verfügbarkeit und Zuverlässigkeit; Kapitel 6, besonders warum man verteilt, Kommunikation und APIs; und die Abschnitte aus Kapitel 7 über Replikation, Partitionierung und Caching.

Dauer

Etwa 3½ Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

Scale Cube · Autoscaling · Zustandsloser Service · Layer-7-Load-Balancer · Power of Two Random Choices · Backpressure · Load Shedding · Token Bucket · Circuit Breaker · Bulkhead · Error Budget · Burn Rate · Distributed Tracing · Chaos Engineering

Von Qualitäten auf dem Papier zu Qualitäten in Produktion

Kapitel 2 hat Performance und Verfügbarkeit präzise gemacht: Latenzperzentile, Durchsatz, Littles Gesetz, Auslastung und die Neunen, formuliert als testbare Qualitätsszenarien. In diesem Kapitel geht es darum, diese Szenarien in einem laufenden System zu erreichen und sie auch am geschäftigsten Tag des Jahres einzuhalten. Ein Entwurf, der seine Ziele im Diagramm erfüllt, kann trotzdem an einem Freitag um 12:30 Uhr zusammenbrechen, wenn jedes Büro der Stadt Mittagessen bestellt, oder um 10:00 Uhr an dem Morgen, an dem der Vorverkauf für Konzertkarten beginnt. Der Unterschied liegt selten in cleverem Code. Er besteht aus einer

Reihe von Architekturentscheidungen: wie Kapazität hinzukommt, wie Last verteilt wird, wie Arbeit wartet, wie Ausfälle eingedämmt werden und wie die Menschen, die das System betreiben, erkennen, was es gerade tut.

Zwei durchgehende Beispiele begleiten das Kapitel. Die **Essenslieferplattform** hat eine ausgeprägte tägliche Spitze zur Mittags- und Abendzeit, Dutzende Services und Abhängigkeiten von Drittanbietern wie Zahlungsdienstleistern und Karten-APIs. Der **Ticket-Vorverkauf** ist der Extremfall: Die Nachfrage steigt innerhalb von Sekunden von fast nichts auf ein Vielfaches der Systemkapazität, für ein festes Kontingent an Plätzen, während das ganze Land zusieht. Zusammen decken sie jede Technik des Kapitels ab.

Last bewältigen: skalieren, verteilen und warten

In den ersten vier Abschnitten geht es um Last. Skalierungsstrategien vergleicht Scale-up mit Scale-out, trennt mit dem Scale Cube von Abbott und Fisher das Klonen, die Aufteilung nach Funktion und die Aufteilung nach Daten und erklärt, warum Autoscaling anhand der CPU oft die falsche Wahl ist und wie Sie eine Flotte mit Littles Gesetz und einer Kapazitätsreserve für eine bekannte Spitze dimensionieren. Load Balancing, Zustandslosigkeit und Sessions behandelt, was vor den Instanzen sitzt: Load Balancer auf Layer 4 und Layer 7, die Algorithmen, die sie verwenden (einschließlich der Power of Two Random Choices), Health Checks, die nicht eine ganze Flotte auf einmal herausnehmen, Connection Draining und warum Sticky Sessions einem ausgelagerten Session-Zustand weichen sollten.

In Asynchronität, Queues und Backpressure geht es darum, was passiert, wenn Arbeit schneller eintrifft, als sie erledigt werden kann. Queues verwandeln Lastspitzen in Rückstände, was nur nützt, wenn der Rückstand begrenzt ist und in akzeptabler Zeit abgebaut wird. Backpressure, Load Shedding und Rate Limiting entscheiden, welche Arbeit angenommen wird, welche wartet und welche höflich mit HTTP 429 abgelehnt wird. Timeouts, Wiederholungen, Circuit Breaker, Bulkheads und Graceful Degradation befasst sich dann mit Ausfällen zwischen Services: Timeouts und Deadlines, Wiederholungen mit Backoff und Jitter und die Budgets, die Retry-Stürme verhindern, Circuit Breaker, Bulkheads, Fallbacks, Idempotenz und Hedged Requests. Das ausgearbeitete Beispiel zeigt, wie ein einziger fehlender Timeout zulässt, dass eine einzige langsame Abhängigkeit alles lahmlegt, was sie aufruft.

Das System betreiben: Ziele, Observability, Tests und Lernen

Die zweite Hälfte wendet sich von den Mechanismen dem Betrieb zu. SLIs, SLOs, SLAs und Error Budgets zeigt, wie Sie Zuverlässigkeit aus Sicht der Nutzer formulieren, wie ein Error Budget ein Verfügbarkeitsziel in eine Menge erlaubter Ausfälle verwandelt, die Teams für Änderungen ausgeben können, und wie Burn-Rate-Alerts Menschen nur dann alarmieren,

wenn dieses Budget wirklich gefährdet ist. Observability behandelt die Logs, Metriken und Traces, mit denen Fachleute einem Produktionssystem neue Fragen stellen können, die RED- und die USE-Methode und die Four Golden Signals, Distributed Tracing mit OpenTelemetry und die Kosten von Daten mit hoher Kardinalität.

Kapazitätsplanung und Performancetests schließt den Kreis zwischen Prognosen und Belegen: Last-, Stress-, Soak-, Spike- und Breakpoint-Tests, offene und geschlossene Lastmodelle, Coordinated Omission und ein Kapazitätsplan für einen Ticket-Vorverkauf. Chaos Engineering, Incidents und schuldfreies Lernen befasst sich mit den Ausfällen, die trotzdem passieren: gezielte Experimente in Produktion, um Schwachstellen zu finden, bevor die Kundschaft es tut, Incidents mit klaren Rollen und schuldfreie Reviews, deren Maßnahmen das System ändern statt der Menschen.

Wie die Abschnitte zusammenhängen

Die Abschnitte bilden eine Kette. Skalierung und Load Balancing schaffen Kapazität; Queues und Backpressure entscheiden, was passiert, wenn die Kapazität trotzdem ausgeht; Resilienzmuster verhindern, dass ein lokales Problem zu einem globalen wird. SLOs legen fest, wie gut gut genug ist, Observability zeigt, ob das System diesen Maßstab erfüllt und warum nicht, Tests und Kapazitätsplanung sagen voraus, wann es ihn nicht mehr erfüllen wird, und Chaos Engineering und Incident-Reviews finden die Lücken in all dem. Die meisten Mechanismen lassen sich beim Entwurf günstig einbauen und nur teuer nachrüsten; deshalb sind sie Architektur und kein Betriebsdetail: Ob ein Service zustandslos ist, ob ein Aufruf idempotent ist, ob eine Queue begrenzt ist und ob eine Anfrage eine Deadline mitführt, wird lange vor dem ersten Incident entschieden.

Das Kapitel setzt die verteilten Stile aus Kapitel 6 und die Datenarchitektur aus Kapitel 7 voraus. Für Idempotenzschlüssel, Zustellgarantien, Replikation und Caching verweist es auf diese Kapitel zurück, statt sie zu wiederholen, und auf Kapitel 10 voraus für die Cloud-Plattformen, Kubernetes-Funktionen und Multi-Region-Designs, auf denen diese Systeme meist laufen.

Der eine Gedanke, den Sie mitnehmen sollten

Jede Ressource ist endlich, also hat jedes System eine Belastungsgrenze; die Aufgabe der Architektur ist es, im Voraus zu entscheiden, was passiert, wenn sie erreicht ist. Ein gut entworfenes System wird unter übermäßiger Last kontrolliert langsamer, wirft die unwichtigste Arbeit zuerst ab, lehnt klar ab statt stillschweigend in einen Timeout zu laufen, isoliert den ausfallenden Teil und erholt sich von selbst, wenn der Druck nachlässt. Ein schlecht entworfenes System reiht ohne Grenze in Warteschlangen ein, wiederholt Anfragen, bis es zusammenbricht, und reißt seine Nachbarn mit. Der Unterschied besteht aus kleinen,

bewussten Entscheidungen (hier ein Timeout, dort eine begrenzte Queue, ein Retry-Budget, eine Readiness-Prüfung) und aus der Gewohnheit, das System so zu messen, zu testen und aus ihm zu lernen, wie es tatsächlich läuft.

ABSCHNITT 9.1 VON 9.8 · MITTELSTUFE · 28 MIN. LESEZEIT

Skalierungsstrategien: vertikal, horizontal und der Scale Cube

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.7 Server und die Cloud
- 2.4 Performance und Skalierbarkeit
- 6.1 Warum verteilen

Begriffe von früher, die er verwendet: Load Balancer, Littles Gesetz, Amdahlsches Gesetz, Auslastung, Read Replica

DIE IDEE IN KÜRZE

Denken Sie an ein gut besuchtes Café. Sie können es mehr Gäste bedienen lassen, indem Sie eine größere, schnellere Kaffeemaschine kaufen (Scale-up) oder mehr Theken mit mehr Personal öffnen (Scale-out). Scale-out reicht weiter, funktioniert aber nur, wenn jede Theke jeden Gast bedienen kann, sodass keine Bestellung nur im Kopf eines Baristas gespeichert sein darf. Sie können die Arbeit auch aufteilen, mit einer Theke für Getränke und einer für Essen, oder mit einem Café pro Stadtviertel. Und weil zusätzliches Personal Zeit braucht, um anzukommen, plant eine vernünftige Leitung es vor dem Mittagsansturm ein, statt es zu rufen, wenn die Schlange schon bis vor die Tür reicht.

WAS SIE WISSEN MÜSSEN

- Vertikale Skalierung fügt einem Rechner Ressourcen hinzu, ohne die Software zu ändern, stößt aber an eine Größen- und Preisobergrenze und bleibt eine einzige Fehlerdomäne; horizontale Skalierung fügt Instanzen hinzu, reicht weiter und toleriert Ausfälle, braucht aber zustandslose, teilbare Arbeit.
- Der Scale Cube (Abbott und Fisher) unterscheidet drei Arten des Scale-out: X klont identische Kopien, Y teilt nach Funktion in Services auf, und Z teilt nach Daten auf, sodass jede Kopie einen Ausschnitt der Kunden oder Datensätze bedient.
- Zustandslosigkeit ist die Voraussetzung für die Skalierung auf der X-Achse: Sessions, Dateien, maßgebliche Caches, geplante Jobs, Sperren und Zähler müssen in gemeinsame Speicher wandern oder mit der Anfrage mitreisen.
- Autoscaling kann reaktiv, zeitgesteuert oder prädiktiv sein; reaktive Skalierung hinkt der Last um Minuten hinterher, daher brauchen sprunghafte Änderungen Kapazitätsreserve, Queues oder im Voraus eingeplante Kapazität.

- Skalieren Sie anhand eines Signals, das an die Ressource gekoppelt ist, die zuerst ausgeht, meist laufende Anfragen pro Instanz oder Queue-Rückstand pro Worker; die CPU führt bei I/O-gebundenen Services in die Irre, die Latenz dann, wenn eine gemeinsame Abhängigkeit langsam ist.
- Verhindern Sie Flapping, indem Sie schnell hoch- und langsam herunterskalieren, mit Stabilisierungsfenstern, Toleranzbändern sowie Mindest- und Höchstzahlen, und dimensionieren Sie Flotten mit Littles Gesetz für die Spitze bei Verlust einer Zone.
- Die Skalierung der Anwendungsschicht verlagert den Druck auf die Datenschicht; prüfen Sie Verbindungen und Hot Rows, und greifen Sie zu Abfrageoptimierung, Caching, Replikaten, Poolern und Scale-up, bevor Sie partitionieren.

Was Skalieren praktisch bedeutet

Abschnitt 2.4 hat Skalierbarkeit als die Fähigkeit definiert, durch zusätzliche Ressourcen mehr Last zu bewältigen, und mit dem Amdahlschen Gesetz und dem Universal Scalability Law gezeigt, warum mehr Ressourcen nicht immer mehr Kapazität bringen. In diesem Abschnitt geht es um die praktischen Entscheidungen: *welche* Ressourcen hinzukommen, *wo*, *wann* und *wie automatisch*. Der Ausgangspunkt ist immer dieselbe Frage: Was ist der Lastparameter, und welche Ressource geht zuerst aus? Die Bestell-API einer Essenslieferplattform ist zur Mittagszeit durch die Anfragen pro Sekunde begrenzt, ihr Dispositionsservice durch die Zahl aktiver Kuriere, deren Positionen alle paar Sekunden abgeglichen werden müssen, ihre Datenbank durch Schreibtransaktionen pro Sekunde und durch Verbindungen. Jede braucht eine andere Antwort.

Ein System zu skalieren heißt eigentlich, seinen aktuellen Engpass zu skalieren und dann den nächsten zu finden. Die Webschicht eines Systems zu verdoppeln, dessen Datenbank zu 95 % ausgelastet ist, bringt nichts außer einer höheren Rechnung und einer Datenbank, die jetzt noch stärker belastet wird. Deshalb beginnt jede Skalierungsentscheidung mit Messung: Auslastung und Sättigung jeder Ressource (die USE-Methode von Brendan Gregg, behandelt in Observability) und ein Lasttest, der zeigt, wo die Latenz zu steigen beginnt (Kapazitätsplanung und Performancetests).

Vertikale Skalierung: ein größerer Rechner

Vertikale Skalierung (Scale-up) bedeutet, dem vorhandenen Rechner mehr CPU-Kerne, mehr Arbeitsspeicher, schnellere Festplatten oder ein schnelleres Netzwerk zu geben. Ihr großer Vorzug: An der Software ändert sich nichts. Eine einzelne relationale Datenbank auf einem größeren Server behält ihre Transaktionen, ihre Joins und ihre einfache Konsistenz; eine Anwendung behält ihre In-Memory-Caches und ihre lokalen Dateien. In der Cloud ist eine Größenänderung ein API-Aufruf mit anschließendem Neustart. Die größten virtuellen

Maschinen in der Cloud bieten heute Hunderte virtueller CPUs und Terabytes an Arbeitsspeicher, weit mehr, als die meisten Geschäftssysteme je brauchen werden, und einige sehr stark besuchte Websites haben öffentlich beschrieben, wie sie enormen Traffic mit wenigen leistungsstarken Datenbankservern bedienen.

Die Grenzen sind allerdings real. Es gibt einen größten Rechner, und der Preis pro Kapazitätseinheit steigt im oberen Bereich steil an. Ein Rechner ist eine Fehlerdomäne: Fällt er aus oder wird er für eine Größenänderung oder ein Patch neu gestartet, steht alles darauf still; Scale-up allein tut also nichts für die Verfügbarkeit. Software kann einen größeren Rechner oft nicht voll nutzen: Ein Single-Thread-Prozess, eine globale Sperre oder eine Hot Row in der Datenbank begrenzt den Geschwindigkeitsgewinn genau so, wie das Amdahlsche Gesetz es vorhersagt. Und Kapazität kommt in großen, langsamen Stufen, sodass Sie rund um die Uhr für die Spitze bezahlen. Trotzdem ist vertikale Skalierung weit häufiger der richtige erste Schritt, als die Mode vermuten lässt, besonders bei Datenbanken, wo Scale-out viel Entwurfsaufwand kostet.

Horizontale Skalierung: mehr Rechner

Horizontale Skalierung (Scale-out) bedeutet, mehr Instanzen zu betreiben und die Arbeit auf sie zu verteilen, meist mit einem Load Balancer davor. Sie reicht viel weiter, in kleinen und günstigen Schritten, und dieselbe Redundanz, die Kapazität schafft, toleriert auch Ausfälle: Bei acht Instanzen kostet der Verlust einer Instanz ein Achtel der Kapazität statt der gesamten. Cloud-Plattformen machen das Hinzufügen einer Instanz zu einer Sache von Sekunden oder Minuten; darauf beruht das Autoscaling.

Der Preis ist, dass die Software dafür entworfen sein muss. Jede Instanz muss jede Anfrage bearbeiten können; Instanzen dürfen also nichts behalten, wovon eine spätere Anfrage abhängt (der nächste Abschnitt). Die Arbeit muss teilbar sein: Ein Job, der eine globale Sperre braucht, läuft auf zehn Rechnern nicht schneller. Die Daten müssen von jeder Instanz aus erreichbar sein, was den Druck auf die gemeinsame Datenschicht verlagert. Und das System wird verteilt, mit den Teilausfällen, Timeouts und Wiederholungen, die Abschnitt 6.1 beschreibt. Die meisten realen Systeme kombinieren beides: Zustandslose Anwendungsschichten skalieren horizontal, während Datenbanken so lange vertikal skalieren, wie es vernünftig möglich ist, und dann mit Replikaten und Partitionen horizontal.

Vertikale und horizontale Skalierung im Vergleich

Anliegen	Vertikal (Scale-up)	Horizontal (Scale-out)
Änderungen an der Software	Keine	Instanzen müssen zustandslos und die Arbeit teilbar sein
Obergrenze	Der größte verfügbare Rechner	Meist die gemeinsame Datenschicht oder Koordinationskosten
Kostenkurve	Steigt bei den größten Größen steil an	Ungefähr linear, in kleinen Schritten
Ausfall (Failure)	Eine Fehlerdomäne; eine Größenänderung erfordert einen Neustart	Der Verlust einer Instanz kostet einen Bruchteil der Kapazität
Änderungsgeschwindigkeit	Minuten plus ein Neustart	Sekunden bis Minuten, auch automatisch
Konsistenz und Transaktionen	Unverändert und einfach	Unverändert bei zustandslosen Schichten; schwierig bei Daten
Typische Verwendung	Datenbanken, Altsysteme und zustandsbehaftete Software, erste Antwort auf Wachstum	Web- und API-Schichten, Worker, alles Zustandslose

Der Scale Cube: drei verschiedene Arten des Scale-out

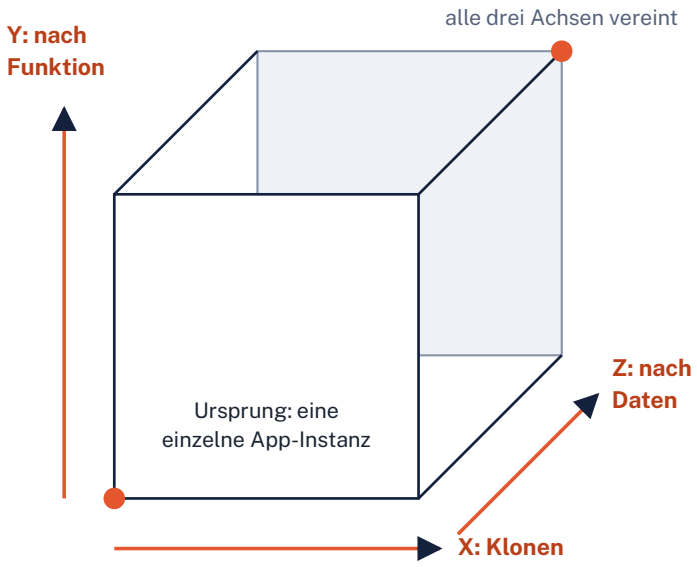
Martin Abbott und Michael Fisher, die die Technologie bei eBay und PayPal leiteten, beschreiben Scale-out mit dem Scale Cube in *The Art of Scalability* (erste Auflage 2009, zweite Auflage 2015). Er unterscheidet drei unabhängige Arten, ein System aufzuteilen, jede als eine Achse eines Würfels gezeichnet, dessen Ursprung eine einzelne Instanz ist, die alles erledigt.

Die **X-Achse** ist horizontale Vervielfältigung, also Klonen: Betreiben Sie N identische Kopien der ganzen Anwendung oder des Service und verteilen Sie die Anfragen auf sie. Sie ist die einfachste und günstigste Achse und eignet sich gut für zustandslose Anfrageverarbeitung und Lesetraffic. Für die Datenmenge oder das Schreibvolumen bringt sie nichts, weil jeder Klon weiterhin mit denselben Daten arbeitet, und ebenso wenig für die Größe der Codebasis oder die Zahl der Teams, die daran arbeiten.

Die **Y-Achse** teilt das System nach Funktion auf: verschiedene Services für Speisekarten, Bestellungen, Disposition und Zahlungen, die jeweils nur den Code und die Daten für ihre Fähigkeit ausführen. Jeder Service kann dann getrennt skalieren, ausfallen und deployt werden, und jeder ist kleiner und leichter zu verstehen. Das ist die Achse hinter Microservices (Abschnitt 6.3), mit all ihren Kosten. Die **Z-Achse** teilt nach Daten auf: Jede Kopie führt

denselben Code aus, ist aber nur für einen Ausschnitt der Kunden, Mandanten, Regionen oder Datensätze zuständig, gewählt über einen Schlüssel wie Stadt oder Kundenkennung. Datenbank-Sharding ist die Z-Achse, angewandt auf die Speicherung (Abschnitt 7.4); eine *zellbasierte* Architektur wendet sie auf ganze Stacks an, sodass ein Fehler oder ein fehlerhaftes Deployment in einer Zelle nur die Kunden dieser Zelle trifft.

Der Scale Cube



X-Achse: Klonen
N identische Kopien hinter einem Load Balancer, jede macht alles: 14 Bestell-API-Kopien

Y-Achse: nach Funktion
Services pro Fähigkeit: Speisekarte, Bestellung, Disposition, Zahlung

Z-Achse: nach Daten
Gleicher Code; jede Kopie hat einen Teil der Kunden/Daten: eine Zelle/Shard je Stadt

Der Scale Cube von Abbott und Fisher: Jede Achse ist eine andere Art, ein System aufzuteilen, und reale Systeme kombinieren meist alle drei.

Die drei Achsen in der Essenslieferplattform

Achse	Was aufgeteilt wird	Beispiel	Löst	Kosten und Grenzen
X: Klonen	Anfragen, über identische Kopien	14 Instanzen der Bestell-API hinter einem Load Balancer	Anfragedurchsatz, Ausfälle von Instanzen	Erfordert Zustandslosigkeit; alle Klone teilen eine Datenbank
Y: Funktion	Code und Daten, nach Fähigkeit	Getrennte Services für Speisekarte, Bestellungen, Disposition und Zahlungen	Unterschiedliche Anforderungen an Skalierung und Ausfallverhalten; Teamgröße	Netzwerkaufrufe, verteilte Daten, Betriebsaufwand
Z: Daten	Kunden oder Datensätze, nach Schlüssel	Eine Zelle pro Städtegruppe oder Bestellungen nach Stadt geshardet	Datenmenge, Schreibvolumen, Blast Radius	Routing-Schicht, partitionsübergreifende Abfragen, Rebalancing, Hot Keys

Die Achsen lassen sich kombinieren. Der Bestellservice der Plattform (eine Aufteilung auf der Y-Achse) läuft als viele Klone (X) in jeder regionalen Zelle (Z). Der praktische Rat von Abbott und Fisher lautet, die günstigste Achse zu wählen, die das tatsächliche Problem löst: X für das Anfragevolumen, Y, wenn Teile des Systems wirklich unterschiedliche Anforderungen an Skalierung, Ausfallverhalten oder Teams haben, und Z, wenn ein Datenbestand oder ein Blast Radius zu groß geworden ist. Wer Z wählt, weil es beeindruckend klingt, holt sich Routing, Rebalancing und partitionsübergreifende Abfragen lange bevor sie nötig sind.

Zustandslosigkeit: die Voraussetzung für das Klonen

Skalierung auf der X-Achse funktioniert nur, wenn jeder Klon jede Anfrage bedienen kann. Ein zustandsloser Service behält zwischen Anfragen nichts in seinem eigenen Arbeitsspeicher oder auf seiner eigenen Festplatte; alles, was eine spätere Anfrage braucht, liegt in einem gemeinsamen Speicher oder reist mit der Anfrage mit. Der Zustand, der sich typischerweise in einer Instanz versteckt, ist leicht aufzuzählen, sobald man hinsieht: im Arbeitsspeicher gehaltene Nutzer-Sessions, hochgeladene Dateien auf der lokalen Festplatte, In-Memory-Caches, die als maßgebliche Quelle behandelt werden, geplante Jobs, die auf genau einem Rechner laufen müssen, prozessinterne Sperren, die eine gemeinsame Ressource schützen, und Zähler für Rate Limits.

Für jeden gibt es ein Standardmittel: Sessions wandern in einen gemeinsamen Speicher oder in ein signiertes Token (Abschnitt 9.2); Dateien wandern in einen Object Storage; Caches werden zu einer Performanceoptimierung, die jederzeit verloren gehen darf, oder wandern in einen gemeinsamen Cache (Abschnitt 7.6); geplante Jobs nutzen einen Scheduler mit Leader Election oder eine Queue; Sperren wandern in die Datenbank oder einen Koordinationsdienst; Rate-Limit-Zähler wandern in einen gemeinsamen Speicher. Die Regel der Twelve-Factor App, dass Prozesse zustandslos sind und nichts teilen (Abschnitt 10.3), ist genau dieser Gedanke. Zustandslosigkeit lässt sich günstig von Anfang an einplanen und nur teuer nachrüsten, weil Zustand an vielen kleinen Stellen in den Code sickert.

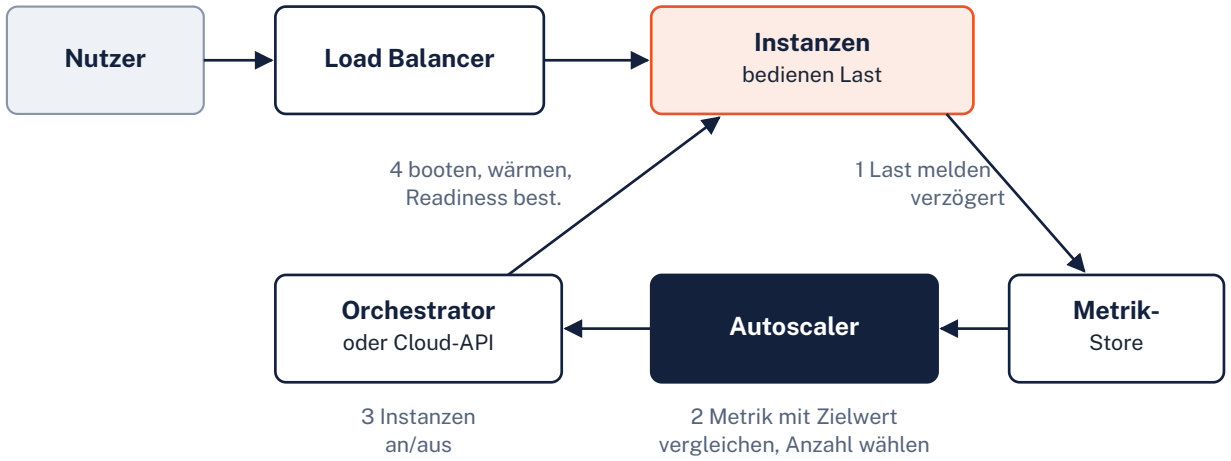
Autoscaling: reaktiv, zeitgesteuert und prädiktiv

Autoscaling schließt einen Regelkreis: Last messen, mit einem Zielwert vergleichen, die Zahl der Instanzen ändern. **Reaktives** Autoscaling tut das fortlaufend anhand einer Metrik. Die übliche Form ist Target Tracking: die durchschnittliche CPU bei 60 % halten oder die laufenden Anfragen pro Instanz bei 30. Der Kubernetes Horizontal Pod Autoscaler berechnet `desired = ceil(current replicas × current metric ÷ target)`: Aus sechs Pods mit durchschnittlich 45 laufenden Anfragen bei einem Zielwert von 30 werden ceil(6 × 45 ÷ 30) = 9. Standardmäßig wertet er alle 15 Sekunden neu aus und ignoriert Abweichungen von weniger als 10 % vom Zielwert.

Zeitgesteuertes Autoscaling ändert die Kapazität zu bekannten Zeiten: das Minimum an jedem Werktag um 11:15 Uhr vor der Mittagsspitze auf 18 Instanzen anheben und um 14:00 Uhr wieder senken. Das ist grob und bei vorhersehbaren Mustern sehr wirksam, und es ist der einzige Weg, Kapazität *vor* einer sprunghaften Änderung bereitzuhaben, etwa vor einem Ticket-Vorverkauf um Punkt 10:00 Uhr. **Prädiktives** Autoscaling, das die großen Cloud-Anbieter anbieten, prognostiziert die Last aus der Historie (Tages- und Wochenzyklen) und skaliert ihr voraus, mit reaktiver Skalierung als Sicherheitsnetz für Überraschungen.

Die entscheidende Eigenschaft jedes reaktiven Verfahrens ist die Verzögerung. Metriken werden über Dutzende Sekunden gesammelt und gemittelt; der Autoscaler handelt bei seiner nächsten Auswertung; eine neue virtuelle Maschine braucht eine Minute oder länger zum Booten, ein Container-Image muss geladen werden, ein JVM- oder Python-Service muss starten, seine Caches und Connection Pools aufwärmen und seine Readiness-Prüfung bestehen, bevor der Load Balancer ihm Traffic schickt. In einem Cluster müssen womöglich erst die Knoten selbst von einem zweiten, langsameren Autoscaler hinzugefügt werden. Vom Moment, in dem die Last steigt, bis zu dem Moment, in dem neue Kapazität nützt, vergehen üblicherweise zwei bis zehn Minuten. Reaktives Autoscaling bewältigt daher allmähliche Änderungen; plötzliche Änderungen müssen durch Kapazitätsreserve, durch Queues (Abschnitt 9.3) oder durch im Voraus eingeplante Kapazität aufgefangen werden.

Der Autoscaling-Regelkreis und woher die Verzögerung kommt



Lastanstieg bis nutzbare Kapazität: oft 2 bis 10 Minuten

Jeder Schritt des Regelkreises fügt Verzögerung hinzu, daher folgt reaktives Autoscaling der Last mit Minuten Abstand; Kapazitätsreserve und eingeplante Kapazität überbrücken die Lücke.

Anhand welcher Metrik skalieren und warum die CPU oft falsch ist

Die CPU-Auslastung ist auf den meisten Plattformen das Standardsignal, weil sie immer verfügbar ist. Das richtige Signal ist sie nur, wenn der Service CPU-gebunden ist: Bildskalierung, Verschlüsselung, aufwendige Serialisierung. Die meisten Geschäftsservices sind es nicht. Eine Bestell-API verbringt den größten Teil jeder Anfrage mit Warten auf die Datenbank, den Zahlungsdienstleister und andere Services; ihre CPU liegt womöglich bei 25 %, während jeder Worker-Thread mit Warten beschäftigt ist und neue Anfragen sich stauen. Skalierung anhand der CPU tut dann nichts, bis es viel zu spät ist. Auch der umgekehrte Fehler kommt vor: Ein Garbage-Collection-Sturm oder ein JIT-Warm-up treibt die CPU hoch, ohne dass die Last steigt, und der Autoscaler fügt Instanzen hinzu, die nicht gebraucht werden.

Besser ist ein Signal, das direkt an die Ressource gekoppelt ist, die ausgeht. Bei Request-Response-Services ist das meist die **Nebenläufigkeit**: die Zahl der laufenden Anfragen pro Instanz, verglichen mit der Zahl ihrer Worker-Threads oder Verbindungen. Littles Gesetz stellt den Zusammenhang exakt her (laufende Anfragen = Ankunftsrate × Antwortzeit), und das Signal reagiert sowohl auf mehr Traffic als auch auf langsamere Antworten. Bei Queue-Workern ist es der **Rückstand pro Worker** oder das Alter der ältesten Nachricht, was ausdrückt, wie lange der Abbau der Queue dauern wird. Die Latenz ist als Skalierungssignal verlockend, aber gefährlich: Sie steigt aus Gründen, die mehr Instanzen nicht beheben können, etwa bei einer langsamen Datenbank, und wer einen Service horizontal skaliert, dessen Abhängigkeit schon kämpft, schickt dieser Abhängigkeit noch mehr Last. Nutzen Sie die Latenz für Alerts und skalieren Sie anhand eines Maßes für Arbeit.

Skalierungssignale und wann sie in die Irre führen

Signal	Passt für	Führt in die Irre, wenn
CPU-Auslastung	CPU-gebundene Arbeit: Bildverarbeitung, Verschlüsselung, aufwendige Serialisierung	Arbeit wartet auf I/O, Threads oder Verbindungen; Garbage Collection oder Warm-up treibt die CPU hoch
Anfragen pro Sekunde und Instanz	Services, deren Anfragen ungefähr gleich viel kosten	Der Anfragemix ändert sich, zum Beispiel mehr Suchen und weniger Speisekarten-Abrufe
Laufende Anfragen pro Instanz	Die meisten Request-Response-Services; entspricht Thread- und Verbindungslimits	Eine nachgelagerte Abhängigkeit wird langsamer; Scale-out belastet sie dann zusätzlich, also das Maximum begrenzen
Queue-Rückstand pro Worker oder Alter der ältesten Nachricht	Asynchrone Worker	Worker sind durch ein nachgelagertes Rate Limit begrenzt statt durch ihre Anzahl
Latenz	Alerts und SLOs (Abschnitt 9.5)	Die Ursache ist eine gemeinsame Abhängigkeit, die mehr Instanzen nicht beheben können
Geschäftsmetrik (Bestellungen pro Minute, aktive Sessions)	Last mit einem stabilen, gemessenen Verhältnis zur Kapazität	Das Verhältnis verschiebt sich, wenn sich das Produkt ändert; mit Lasttests neu kalibrieren

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: beschäftigte Threads, untätiger Prozessor

Der schwierigste Gedanke dieses Abschnitts ist, dass ein Server voll sein kann, während sein Prozessor fast untätig wirkt; die CPU-Auslastung kann daher das falsche Signal zum Skalieren sein.

- Eine Instanz der Bestellverlauf-API hat 48 Worker-Threads, kann also höchstens 48 Anfragen gleichzeitig bearbeiten. Jede Anfrage verbringt die meiste Zeit mit Warten: auf Datenbankabfragen und auf Antworten des Speisekarten-Service.
- Ein wartender Thread verbraucht fast keine Prozessorzeit, ist aber trotzdem belegt: Er kann keine weitere Anfrage beginnen, bis die Antwort zurückkommt.
- Jede Anfrage belegt einen Thread etwa 0,14 Sekunden lang. Nach Littles Gesetz halten 250 Anfragen pro Sekunde etwa 35 Threads beschäftigt (250 × 0,14). Bei 450 pro Sekunde wären 63 nötig, mehr als die vorhandenen 48; Anfragen stauen sich also, und die Antwortzeiten steigen.
- Trotzdem liegt der Prozessor womöglich bei etwa 28 %. Ein Autoscaler, der die CPU bei 65 % halten soll, sieht kein Problem und fügt keine Instanzen hinzu, während die Kundschaft wartet.
- Das Zählen der laufenden Anfragen pro Instanz zeigt das Problem direkt: Der Wert klettert in Richtung 48. Wer anhand dieses Werts skaliert, mit einem Zielwert von etwa 36 (drei Viertel der Threads), fügt Instanzen hinzu, solange noch einige Threads frei sind.
- Für Arbeit, die wirklich rechnet, etwa das Skalieren von Bildern, ist die CPU weiterhin richtig. Und der Wert der laufenden Anfragen steigt auch, wenn die Datenbank selbst kämpft; dann schicken mehr Instanzen ihr nur mehr Last, weshalb der Abschnitt auch ein Maximum festlegt.
- **Die Lehre:** Skalieren Sie anhand der Ressource, die tatsächlich ausgeht. Bei den meisten Geschäftsservices sind das Threads oder Verbindungen; messen Sie also die laufende Arbeit, nicht wie hart der Prozessor arbeitet.

Cooldowns, Stabilisierung und Flapping

Ein Autoscaler, der auf jedes Schwanken reagiert, gerät ins Flapping: Er fügt Instanzen hinzu, wenn ein Lastschub kommt, entfernt sie eine Minute später, wenn der Schub vorbei ist, und fügt sie dann wieder hinzu, zahlt jedes Mal die Startkosten und stört Caches und Connection Pools. Er kann sich auch selbst im Kreis jagen: Neue Instanzen sind kalt und langsam, daher steigen Latenz oder CPU pro Anfrage nach dem Scale-out kurzzeitig an, was wie mehr Last aussieht.

Die üblichen Gegenmaßnahmen sind asymmetrisch. Skalieren Sie schnell hoch, weil zu wenig Kapazität den Nutzern schadet; skalieren Sie langsam herunter, weil zu viel Kapazität nur Geld kostet. Ein Cooldown oder Stabilisierungsfenster lässt den Autoscaler nach einer Aktion warten oder vor dem Herunterskalieren die höchste Empfehlung eines zurückliegenden Zeitfensters nehmen: Der Kubernetes-Autoscaler nutzt standardmäßig ein Fenster von fünf Minuten (300 Sekunden) für das Herunterskalieren und keines für das Hochskalieren. Ein Toleranzband ignoriert kleine Abweichungen. Mindest- und Höchstzahlen für Instanzen begrenzen den Regelkreis: Das Minimum hält genug Kapazität vor, um den Verlust einer Zone und die ersten Minuten eines Ansturms zu überstehen, und das Maximum schützt die Datenbank, nachgelagerte Services und das Budget vor einem Autoscaler, der ein Problem fälschlich als Last deutet. Auch das Herunterskalieren muss kontrolliert ablaufen: Eine zum Entfernen ausgewählte Instanz darf keine neuen Anfragen mehr erhalten und muss die laufenden abschließen (Connection Draining, Abschnitt 9.2).

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Bestell-API für die Mittagsspitze dimensionieren

Die Bestell-API der Essenslieferplattform muss das Mittagsgeschäft an Werktagen bewältigen. In den letzten acht Wochen erreichte die geschäftigste Minute gegen 12:30 Uhr 2050 Anfragen pro Sekunde (rps); das Marketing erwartet in diesem Quartal 15 % Wachstum. Ein Lasttest einer Instanz (4 vCPU, 32 Worker-Threads) ergab eine mittlere Antwortzeit von 120 ms und einen Latenzanstieg ab etwa 260 rps. Der Service läuft in drei Availability Zones und muss den Verlust einer Zone zur Spitzenzeit überstehen.

Prognostizierte Spitze λ	2050 × 1,15 ≈ 2360, aufgerundet auf 2400 rps
Littles Gesetz für die Flotte	L = λ × W = 2400 × 0,12 s = 288 laufende Anfragen zur Spitzenzeit
Kapazität einer Instanz	32 Threads ÷ 0,12 s ≈ 267 rps, was zum Knick passt, den der Lasttest fand (etwa 260 rps)
Zielauslastung	70 % von 32 Threads = 22,4 laufende Anfragen pro Instanz (etwa 187 rps), bleiben 30 % Reserve für Lastschübe
Instanzen für die Spitze	288 ÷ 22,4 = 12,9, also 13 Instanzen (2400 ÷ 13 ≈ 185 rps je Instanz)
Verlust einer Zone überstehen	13 verteilt auf 5/4/4: Der Verlust von 5 lässt 8 Instanzen mit je 288 ÷ 8 = 36 laufenden Anfragen, mehr als 32 Threads: Überlast. Wer beim Verlust einer Zone bis zu 85 % (27,2 laufende Anfragen) zulässt, braucht 288 ÷ 27,2 = 10,6, also 11 in zwei Zonen: 6 pro Zone, insgesamt 18
Hochlauf und Scale-out-Verzögerung	Die Nachfrage steigt von 600 rps um 11:30 Uhr auf 2400 um 12:15 Uhr: 40 rps pro Minute. Reaktives Scale-out dauert etwa 5 Minuten (1 Minute zum Erkennen, 4 zum Booten und Aufwärmen), in denen die Nachfrage um 200 rps steigt; die freie Kapazität bei 70 % beträgt etwa 80 rps pro Instanz, reaktive Skalierung allein könnte mit diesem allmählichen Hochlauf also Schritt halten; was sie nicht bieten kann, ist Schutz gegen den Verlust einer Zone
Entscheidung	Zeitgesteuerte Skalierung hebt das Minimum um 11:15 Uhr auf 18 (6 pro Zone) und senkt es um 14:00 Uhr auf 6 (2 pro Zone); reaktive Skalierung anhand der laufenden Anfragen pro Instanz (Zielwert 22) bleibt als Sicherheitsnetz, mit einem Maximum von 30
Datenbankprüfung	18 Instanzen × ein Pool von 10 Verbindungen = 180; beim Maximum von 30 Instanzen 300. Die Datenbank erlaubt 200, daher fügt das Team einen Connection Pooler hinzu, bevor es das Maximum anhebt

Littles Gesetz macht aus einer prognostizierten Rate und einer gemessenen Antwortzeit eine Nebenläufigkeitszahl, die sich direkt mit Thread- und Verbindungslimits vergleichen lässt.

Die Anforderung, den Verlust einer Zone zu überstehen, nicht die durchschnittliche Last, bestimmt die Flottengröße zur Spitzenzeit: 18 Instanzen statt 13. Reaktives Autoscaling kann diesen Schutz nicht bieten, weil das Ersetzen einer verlorenen Zone Minuten dauert.

Das Scale-out der Anwendungsschicht hat die Last auf die Verbindungen der gemeinsamen Datenbank vervielfacht, den nächsten Engpass. Jede Dimensionierung sollte mit einer Prüfung der darunterliegenden Schicht enden.

Die Datenschicht skalieren

Zustandslose Anwendungsschichten lassen sich leicht skalieren; genau deshalb macht man sie zustandslos. Der Druck landet dann auf den Datenbanken, Caches und Brokern, die sie gemeinsam nutzen; deshalb ist die Datenschicht meist die eigentliche Grenze eines Systems. Die Werkzeuge wurden in Kapitel 7 behandelt und sind hier in der Reihenfolge zusammengefasst, in der Fachleute meist zu ihnen greifen. Erstens: die Arbeit verringern, mit Indizes, besseren Abfragen, weniger Roundtrips pro Anfrage. Zweitens: Lesezugriffe mit Caching (Abschnitt 7.6) und Lesereplikaten auffangen und Replikationsverzögerung hinnehmen, wo der Anwendungsfall es erlaubt. Drittens: Verbindungen schützen. Jede Anwendungsinstanz hält einen Pool, die Gesamtzahl wächst also mit der Flotte, und ein Connection Pooler wie PgBouncer lässt Tausende Client-Verbindungen sich einige Hundert Datenbankverbindungen teilen (der Standardwert von `max_connections` in PostgreSQL ist nur 100). Viertens: den Primary vertikal skalieren, was oft günstiger ist als jeder Umbau. Erst dann Schreibzugriffe entlang der Z-Achse nach einem Schlüssel wie Stadt oder Kunde partitionieren (Abschnitt 7.4) oder auf eine Distributed-SQL-Datenbank wechseln.

Zwei Warnungen gelten. Erstens kann Autoscaling der Anwendungsschicht der Datenschicht schaden: Jede neue Instanz öffnet Verbindungen und wärmt ihren Cache aus der Datenbank auf, sodass ein großes Scale-out-Ereignis die Datenbank selbst überlasten kann. Zweitens sind manche Grenzen logisch, nicht physisch: Eine einzige Hot Row (der Bestandszähler für die letzten 500 Tickets eines Konzerts oder eine globale Sequenznummer) serialisiert Schreibzugriffe, egal wie groß der Server ist. Solche Grenzen beseitigt man durch Design, etwa indem man den Bestand auf mehrere Zähler aufteilt oder Plätze in Blöcken reserviert, nicht durch Hardware.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: wie Sie einer Komponente Kapazität hinzufügen

Gehen Sie die Fragen der Reihe nach für die Komponente durch, die das System heute begrenzt.

Q1 **Wurde der Engpass gemessen (Auslastung, Sättigung und Warteschlangen bei CPU, Arbeitsspeicher, Verbindungen, Sperren und nachgelagerten Aufrufen) statt geschätzt?**

- **Ja** Machen Sie mit der Komponente weiter, die zuerst gesättigt ist.
- **Nein** Zuerst einen Lasttest durchführen und messen. Wer die falsche Schicht skaliert, erhöht die Kosten ohne Kapazitätsgewinn und kann den echten Engpass noch stärker belasten.

Q2 **Lässt sich die Arbeit selbst verbilligen (ein Index, ein Cache, weniger Roundtrips, kleinere Payloads, weniger Logging)?**

- **Ja** Tun Sie das zuerst. Es senkt die Kosten auf jeder Größenordnung und macht Skalierung oft ganz überflüssig.
- **Nein** Weiter: Es muss Kapazität hinzukommen.

Q3 **Ist die Komponente zustandslos, oder lässt sich ihr Zustand (Sessions, Dateien, In-Memory-Sperren, Zeitpläne) in gemeinsame Speicher verlagern?**

- **Ja** Skalieren Sie auf der X-Achse hinter einem Load Balancer, mit Autoscaling anhand eines Lastsignals und genug Kapazitätsreserve für die Scale-out-Verzögerung.
- **Nein** Skalieren Sie vorerst vertikal und planen Sie die Arbeit, den Zustand auszulagern; eine zustandsbehaftete Einzelinstanz ist zudem ein Single Point of Failure.

Q4 **Liegt die Grenze in der Datenschicht: Schreibrate, Speichergröße oder Verbindungszahl einer Datenbank?**

- **Ja** Nutzen Sie Caching und Lesereplikate für Lesezugriffe, einen Connection Pooler für Verbindungen und Partitionierung auf der Z-Achse (Shards oder Zellen) für Schreibzugriffe und Speicherung.
- **Nein** Weiter.

Q5 **Haben Teile der Komponente sehr unterschiedliche Anforderungen an Last, Skalierung oder Ausfallverhalten?**

Ja Erwägen Sie eine Aufteilung auf der Y-Achse, sodass jeder Teil getrennt skaliert und ausfällt, und nehmen Sie die in Kapitel 6 beschriebenen Kosten der Verteilung in Kauf.

Nein Bleiben Sie beim Klonen auf der X-Achse: Es ist die am günstigsten zu betreibende Achse.

ACHTUNG

Häufige Fehler

- Die Schicht skalieren, die sich leicht skalieren lässt, statt der gesättigten, typischerweise Webserver vor einer Datenbank hinzufügen, die bereits der Engpass ist.
- I/O-gebundene Services anhand der CPU automatisch skalieren, sodass Kapazität erst ankommt, wenn die Latenz bereits eingebrochen ist.
- Sich bei sprunghaften Änderungen wie Vorverkäufen, Übertragungen oder Marketingaktionen, die schneller eintreffen, als Instanzen starten können, auf reaktives Autoscaling verlassen.
- Keine Höchstzahl an Instanzen festlegen, sodass eine langsame Abhängigkeit als Last fehlgedeutet wird und der Autoscaler die Verbindungen zu ihr vervielfacht.
- Für den Durchschnittstag dimensionieren statt für die Spitze bei Verlust einer Zone oder Instanz.
- Annehmen, dass Instanzen zustandslos sind, ohne nach In-Memory-Sessions, lokalen Dateien, geplanten Jobs auf Einzelinstanzen und prozessinternen Sperren zu suchen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Die CPU einer Bestell-API liegt bei 30 %, aber die p99-Latenz hat sich verdreifacht, und alle Worker-Threads sind belegt. Warum hat der CPU-basierte Autoscaler nicht reagiert, und welches Signal hätte reagiert?

Der Service wartet auf I/O, daher sind die Threads erschöpft, während die CPU untätig ist. Die laufenden Anfragen pro Instanz, verglichen mit der Thread-Zahl, wären mit der Last gestiegen.

▼ Welche Achse des Scale Cube hilft bei einer Datenbank, die der Schreibkapazität eines Servers entwachsen ist?

Die Z-Achse: die Daten nach einem Schlüssel partitionieren, sodass jeder Shard oder jede Zelle einen Teil der Schreibzugriffe übernimmt.

▼ Warum skalieren Autoscaler meist schnell hoch, aber langsam herunter?

Zu wenig Kapazität schadet den Nutzern sofort, zu viel kostet nur Geld, und zu eifriges Herunterskalieren verursacht Flapping, wenn die Last zurückkehrt.

▼ Warum kann reaktives Autoscaling einen Ticket-Vorverkauf, der um Punkt 10:00 Uhr beginnt, nicht absichern?

Die Nachfrage springt innerhalb von Sekunden, aber das Erkennen des Anstiegs, das Starten der Instanzen und ihr Aufwärmen dauern Minuten; die Kapazität muss also vor Verkaufsbeginn eingeplant werden.

QUELLEN

- Kubernetes documentation: Horizontal Pod Autoscaling: https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/
- Chris Richardson: The Scale Cube (microservices.io): https://microservices.io/articles/scalecube.html
- The Twelve-Factor App: VI. Processes: https://12factor.net/processes
- Brendan Gregg: The USE method: https://brendangregg.com/usemethod.html

ABSCHNITT 9.2 VON 9.8 · MITTELSTUFE · 27 MIN. LESEZEIT

Load Balancing, Zustandslosigkeit und Sessions

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.4 Wie Computer kommunizieren
- 7.4 Replikation und Partitionierung
- 9.1 Skalierung

Begriffe von früher, die er verwendet: Konsistentes Hashing, DNS, TCP, gRPC

DIE IDEE IN KÜRZE

Stellen Sie sich einen Supermarkt vor, in dem eine Mitarbeiterin die Kundschaft zu den Kassen leitet. Die Leute der Reihe nach zu jeder Kasse zu schicken, ist fair, aber naiv: Ein Kunde mit dem Wocheneinkauf für eine sechsköpfige Familie hält eine Kasse auf, während die nächste leer steht. Jeden Kunden zur Kasse mit der kürzesten Schlange zu schicken, funktioniert viel besser. Die Mitarbeiterin muss außerdem wissen, welche Kassen geöffnet sind, darf niemanden mehr zu einer Kasse schicken, die gleich schließt, und muss deren Schlange noch abarbeiten lassen; und am besten funktioniert es, wenn jede Kasse jeden Kunden bedienen kann, sodass niemand zur selben Kassiererin zurückkommen muss, die seine Kundenkarte hat.

WAS SIE WISSEN MÜSSEN

- Ein Load Balancer verteilt Verbindungen oder Anfragen auf gesunde Instanzen, verbirgt Änderungen an den Instanzen hinter einer Adresse und terminiert oft TLS; er kann am Edge oder in Clients und Sidecars sitzen.
- Layer-4-Balancer leiten Verbindungen nach Adresse und Port weiter, schnell und protokollunabhängig; Layer-7-Balancer lesen jede HTTP-Anfrage und können routen, wiederholen und Rate Limits durchsetzen, was Protokolle mit Balancing pro Anfrage wie gRPC über HTTP/2 erfordern.
- Round Robin ignoriert die Kosten einer Anfrage; Least Connections und die Power of Two Random Choices (Mitzenmacher) passen sich der Last an, und zwei Zufallsauswahlen vermeiden den Herdentrieb, den viele Balancer erzeugen, wenn alle der am wenigsten belasteten Instanz nachjagen.
- Liveness-Fehler starten eine Instanz neu, Readiness-Fehler nehmen sie nur aus der Rotation; halten Sie gemeinsame Abhängigkeiten aus Liveness-Prüfungen heraus, damit ein kurzer Datenbankaussetzer nicht die ganze Flotte neu startet.

- Connection Draining stoppt neue Anfragen an eine ausscheidende Instanz und lässt laufende Arbeit abschließen, sodass Deployments und Herunterskalieren keine Fehler verursachen.
- Sticky Sessions verursachen ungleichmäßige Last und verlorene Sessions bei Ausfällen; verlagern Sie den Session-Zustand in signierte Tokens, einen gemeinsamen Speicher oder die Datenbank, damit jede Instanz jede Anfrage bedienen kann.
- Globales Load Balancing lenkt Nutzer per DNS-Antworten zu Regionen, begrenzt durch Caching und TTLs, oder per Anycast, das eine Adresse von vielen Standorten aus ankündigt und das Routing den nächstgelegenen wählen lässt.

Was ein Load Balancer tut

Ein Load Balancer sitzt zwischen Clients und einem Pool austauschbarer Instanzen und entscheidet für jede Verbindung oder Anfrage, welche Instanz sie erhält. Die Verteilung ist nur ein Teil der Aufgabe. Ein Load Balancer prüft auch, welche Instanzen gesund sind, und schickt den anderen keinen Traffic mehr; er gibt Clients eine stabile Adresse, während Instanzen mit Deployments und Autoscaling kommen und gehen; er terminiert oft TLS, sodass Zertifikate an einer Stelle verwaltet werden; und auf Layer 7 kann er jede Anfrage routen, wiederholen, mit Rate Limits belegen und protokollieren. Er ist daher die Komponente, die horizontale Skalierung für Clients unsichtbar macht.

Load Balancing gibt es auf mehreren Ebenen. Am Edge steht meist ein verwalteter Cloud-Load-Balancer oder ein Paar von Proxys wie HAProxy, NGINX oder Envoy. Zwischen Services wird das Balancing zunehmend clientseitig erledigt: Eine gRPC-Client-Bibliothek oder ein Sidecar-Proxy eines Service Mesh (Abschnitt 6.3) hält die Liste gesunder Instanzen aus der Service Discovery und wählt selbst eine aus, was einen Netzwerk-Hop und eine zentrale Komponente einspart. Unabhängig von der Form kehren dieselben Entscheidungen wieder: was der Balancer sehen kann, welchen Algorithmus er verwendet, wie er entscheidet, ob eine Instanz gesund ist, und wie Instanzen ausscheiden, ohne Arbeit zu verlieren.

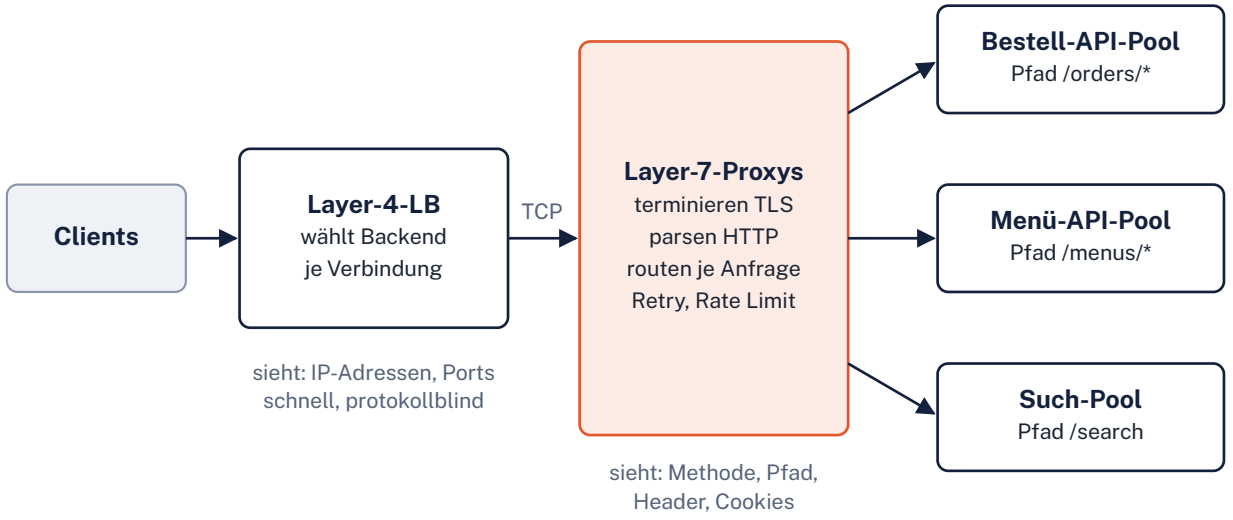
Load Balancing auf Layer 4 und Layer 7

Die Layer-Nummern stammen aus dem OSI-Netzwerkmodell. Ein Layer-4-Load-Balancer arbeitet auf der Transportschicht: Er sieht IP-Adressen und TCP- oder UDP-Ports, wählt beim Öffnen einer Verbindung ein Backend und leitet die Pakete dieser Verbindung dann weiter, ohne hineinzusehen. Weil er so wenig tut, ist er extrem schnell und funktioniert mit jedem Protokoll, auch mit Datenbanken, Message Brokern und verschlüsseltem Traffic, den er nicht lesen kann. Googles Maglev und die Network Load Balancer der großen Clouds sind Layer-4-Designs, die Millionen Pakete pro Sekunde verarbeiten.

Ein Layer-7-Load-Balancer ist ein Reverse Proxy, der das Anwendungsprotokoll versteht, fast immer HTTP. Er terminiert die Verbindung des Clients (und meist TLS), liest jede Anfrage und öffnet eigene Verbindungen zu den Backends oder nutzt sie wieder. So kann er nach Pfad, Host, Header oder Cookie routen (`/orders/*` zum Bestell-Pool, `/menus/*` zum Speisekarten-Pool), jede Anfrage statt jeder Verbindung verteilen, eine fehlgeschlagene Anfrage auf einer anderen Instanz wiederholen, Rate Limits durchsetzen, Tracing-Header hinzufügen und jede Anfrage mit Status und Latenz protokollieren. Der Preis sind mehr Verarbeitung pro Anfrage, eine Komponente, die die TLS-Schlüssel halten muss, und eine weitere Stelle, an der Timeouts und Wiederholungen konsistent mit dem übrigen System konfiguriert werden müssen (Abschnitt 9.4).

Ein Unterschied erwischt viele Teams kalt. HTTP/2 und gRPC multiplexen viele Anfragen über eine langlebige Verbindung. Ein Layer-4-Balancer wählt ein Backend einmal pro Verbindung, daher schickt ein Client, der eine Verbindung öffnet, alle seine Anfragen an eine Instanz, egal wie viele Instanzen es gibt: Neue Instanzen, die das Autoscaling hinzufügt, erhalten nichts, bis die Clients sich neu verbinden. Das Balancing von gRPC-Traffic braucht daher einen Layer-7-Proxy oder clientseitiges Balancing pro Anfrage.

Eine typische zweistufige Anordnung



Eine Layer-4-Stufe verteilt Verbindungen günstig; eine Layer-7-Stufe liest jede Anfrage und routet sie nach Pfad zum richtigen Pool.

Load-Balancing-Algorithmen

Der Algorithmus entscheidet, welche gesunde Instanz die nächste Verbindung oder Anfrage erhält. Die einfachen ignorieren, wie ausgelastet Instanzen sind; die besseren nutzen ein Maß für die Last; hashbasierte schicken zusammengehörige Anfragen bewusst an dieselbe Stelle.

Die Tabelle fasst die üblichen Optionen zusammen.

Gängige Load-Balancing-Algorithmen

Algorithmus	Wie er wählt	Stärken	Schwächen
Round Robin	Jede Instanz der Reihe nach	Einfach, vorhersehbar, kein Zustand über die Last	Blind für Anfragekosten und langsame Instanzen
Weighted Round Robin	Der Reihe nach, im Verhältnis zu konfigurierten Gewichten	Mischt Instanzgrößen; fährt neue Versionen schrittweise hoch (Canaries)	Gewichte sind statische Schätzungen; weiterhin blind für die tatsächliche Last
Zufällig	Gleichverteilt zufällig	Braucht keinen gemeinsamen Zustand, daher können viele Balancer es unabhängig nutzen	Zufällig ungleichmäßig, besonders bei wenigen Anfragen
Least Connections (Least Outstanding Requests)	Instanz mit den wenigsten laufenden Anfragen	Passt sich ungleichen Anfragekosten und langsamen Instanzen an	Braucht genaue Zählungen; jeder von vielen Balancern sieht nur seine eigenen; neue Instanzen können überflutet werden
Latenzgewichtet (zum Beispiel EWMA der Antwortzeit)	Instanz mit der besten jüngsten Latenz, oft kombiniert mit der Last	Lenkt schnell von beeinträchtigten Instanzen weg	Kann oszillieren; eine schnell scheiternde Instanz wirkt attraktiv, wenn Fehler nicht bestraft werden
Hash (Quell-IP, Header, konsistentes Hashing)	Derselbe Schlüssel landet immer bei derselben Instanz	Affinität für Caches und Sessions; konsistentes Hashing verschiebt wenige Schlüssel, wenn sich der Pool ändert	Hot Keys überlasten eine Instanz; ungleichmäßige Schlüsselverteilung
Power of Two Random Choices	Zwei Instanzen zufällig wählen; an die weniger belastete senden	Fast so gut wie Least Connections, kein Herdentrieb, wenig Koordination	Etwas schlechter als eine perfekte globale Wahl mit perfekter Information

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: warum Least Connections Round Robin schlägt, wenn Anfragekosten sich unterscheiden

Zwei Instanzen, A und B, bearbeiten jeweils eine Anfrage zur Zeit (echte Instanzen haben mehr Threads, aber der Effekt ist im größeren Maßstab derselbe). Acht Anfragen treffen im Abstand von 10 ms ein. Die erste ist ein Umsatzbericht eines Restaurants, der 100 ms kostet; die anderen sieben sind Speisekarten-Abrufe zu je 10 ms.

Round Robin: Zuteilung	A erhält die Anfragen 1, 3, 5, 7; B erhält 2, 4, 6, 8, unabhängig von den Kosten
Round Robin auf A	R1 läuft von 0 bis 100 ms. R3 (eingetroffen bei 20) wartet und ist bei 110 fertig: 90 ms. R5 (40) ist bei 120 fertig: 80 ms. R7 (60) ist bei 130 fertig: 70 ms
Round Robin auf B	R2, R4, R6 und R8 sind jeweils 10 ms nach dem Eintreffen fertig; B ist die halbe Zeit untätig
Ergebnis bei Round Robin	Speisekarten-Abrufe: 90, 80, 70, 10, 10, 10, 10 ms, Mittelwert 290 ÷ 7 ≈ 41 ms, schlechtester Wert 90 ms. Alle acht Anfragen: Mittelwert (100 + 290) ÷ 8 ≈ 49 ms
Least Connections: Zuteilung	R1 geht an A (beide untätig). Bei 10 ms hat A 1 laufende Anfrage und B 0, also geht R2 an B; B ist vor jedem nächsten Eintreffen wieder frei, also gehen R3 bis R8 alle an B
Ergebnis bei Least Connections	Jeder Speisekarten-Abruf dauert 10 ms; der Bericht dauert 100 ms. Mittelwert (100 + 70) ÷ 8 ≈ 21 ms, schlechtester Speisekarten-Abruf 10 ms
Warum	Round Robin zählt Anfragen, nicht Arbeit. Least Connections zählt laufende Arbeit, die nach Littles Gesetz (laufend = Rate × Zeit) auf der jeweils langsameren oder stärker belasteten Instanz wächst, sodass neue Anfragen automatisch von ihr wegfließen

Derselbe Effekt tritt immer dann auf, wenn Anfragekosten stark schwanken (Suchen gegenüber Nachschlagen, Exporte gegenüber Lesezugriffen) oder eine Instanz langsamer ist als die anderen, wegen eines Noisy Neighbour, einer Garbage-Collection-Pause oder eines kalten Caches.

Least Connections braucht eine aussagekräftige Zählung. Bei HTTP/2 oder gRPC teilen sich viele Anfragen eine Verbindung, daher zählen Balancer stattdessen offene Anfragen; bei langlebigen WebSocket-Verbindungen misst die Zählung Sessions, nicht Last.

Eine neu gestartete Instanz hat null Verbindungen und kann überflutet werden, bevor ihre Caches warm sind; viele Balancer bieten eine Slow-Start-Phase, die ihren Anteil über etwa eine Minute hochfährt.

Die Power of Two Random Choices

Wenn Least Connections so gut funktioniert, warum nicht immer die global am wenigsten belastete Instanz wählen? Weil es in einem großen System keine einzige, aktuelle Sicht auf die Last gibt. Dutzende Load Balancer oder Tausende clientseitige Balancer sehen jeweils nur ihre eigenen Anfragen oder einen Lastbericht, der einige Sekunden alt ist. Wenn alle die Instanz wählen, die am wenigsten belastet aussah, wählen alle *dieselbe*, die prompt überlastet wird, und dann ziehen alle zur nächsten weiter: eine Herde, die der Last durch den Pool hinterherjagt.

Michael Mitzenmachers Doktorarbeit (1996), veröffentlicht als *The Power of Two Choices in Randomized Load Balancing* (IEEE Transactions on Parallel and Distributed Systems, 2001) und aufbauend auf früheren Arbeiten von Azar, Broder, Karlin und Upfal, analysierte eine verblüffend einfache Alternative: zwei Instanzen zufällig wählen und die Anfrage an die weniger belastete der beiden senden. Im klassischen Bälle-in-Körbe-Modell ergibt die zufällige Verteilung von n Anfragen auf n Server einen am stärksten belasteten Server mit etwa log n ÷ log log n Anfragen; mit zwei Zufallsauswahlen sinkt das auf etwa log log n ÷ log 2, eine exponentielle Verbesserung, während eine dritte Auswahl nur noch wenig bringt. Der Zufall verhindert den Herdentrieb, und der Vergleich vermeidet die schlechtesten Instanzen.

Die Power of Two Random Choices ist heute in weit verbreiteten Proxys der Standard oder eine Standardoption: Der Least-Request-Balancer von Envoy vergleicht standardmäßig zwei zufällig gewählte Hosts, und NGINX bietet `random two least_conn`. Sie ist eine gute Voreinstellung für clientseitiges Balancing in jedem System mit vielen Balancern.

Hashing und Affinität

Manchmal geht es gerade darum, zusammengehörige Anfragen an *dieselbe* Instanz zu schicken: an einen Cache-Knoten, der die Speisekarte eines Restaurants bereits hält, oder an die Instanz, die eine WebSocket-Session hält. Das Hashen eines Schlüssels (Restaurantkennung, Nutzerkennung, Quell-IP) auf eine Instanz erreicht das. Einfaches `hash mod N` ordnet die meisten Schlüssel neu zu, sobald sich N ändert, und leert damit jeden Cache im ungünstigsten Moment, während eines Scale-out. Konsistentes Hashing (eingeführt in Abschnitt 7.4) verschiebt nur etwa 1 ÷ N der Schlüssel, wenn eine Instanz hinzukommt oder wegfällt. Seine Schwäche sind Hot Keys: Das beliebteste Restaurant zur Mittagszeit landet auf einer Instanz. Varianten wie Consistent Hashing with Bounded Loads begrenzen den Anteil jeder Instanz und geben den Überschuss an die nächste Instanz auf dem Ring weiter.

Health Checks: Liveness und Readiness

Ein Load Balancer muss wissen, welche Instanzen Traffic annehmen können. Ein aktiver Health Check prüft jede Instanz regelmäßig, zum Beispiel mit `GET /healthz` alle fünf Sekunden, markiert sie nach einigen aufeinanderfolgenden Fehlschlägen als ausgefallen und nach einigen Erfolgen wieder als verfügbar. Passive Prüfungen, oft Outlier Detection genannt, beobachten den echten Traffic und werfen eine Instanz hinaus, die etwa fünf aufeinanderfolgende Serverfehler liefert. Beide ergänzen sich: Aktive Prüfungen finden Instanzen, die keinen Traffic erhalten, passive Prüfungen reagieren auf Fehler, die eine triviale Prüfung nicht auslöst.

Kubernetes hat eine wichtige Unterscheidung explizit gemacht. Eine Liveness Probe fragt, ob der Prozess unrettbar hängt (Deadlock, Speichermangel in einer Schleife); schlägt sie fehl, wird die Instanz neu gestartet. Eine Readiness Probe fragt, ob die Instanz gerade jetzt Traffic erhalten sollte; schlägt sie fehl, wird die Instanz aus dem Load Balancer genommen, läuft aber weiter, etwa während sie beim Start ihre Caches aufwärmt oder während sie Last abwirft. Eine dritte, die Startup Probe, hält die anderen beiden zurück, während eine langsame Anwendung startet.

Der klassische Fehler besteht darin, in diesen Probes gemeinsame Abhängigkeiten zu prüfen. Ruft die Liveness Probe jeder Instanz die Datenbank auf, lässt ein kurzer Datenbankausfall jede Probe scheitern, der Orchestrator startet alle Instanzen gleichzeitig neu, und der Neustart-Sturm (kalte Caches, Tausende neuer Verbindungen) verlängert den Ausfall noch lange, nachdem die Datenbank wieder da ist. Readiness-Prüfungen, die eine gemeinsame Abhängigkeit einschließen, können ebenso die ganze Flotte als nicht bereit markieren und Teilfehler in einen Totalausfall verwandeln. Halten Sie Liveness lokal und minimal, lassen Sie Readiness die eigene Fähigkeit der Instanz zum Bedienen widerspiegeln, und behandeln Sie Ausfälle von Abhängigkeiten mit den Mustern aus Abschnitt 9.4. Load Balancer schützen sich gegen den Extremfall, indem sie offen ausfallen (Fail Open): Ein AWS Application Load Balancer routet zu allen Zielen, wenn alle ihre Health Checks nicht bestehen, und Envoy ignoriert den Gesundheitsstatus, wenn der Anteil gesunder Hosts unter eine Panikschwelle fällt (standardmäßig 50 %), mit der Begründung, dass Traffic an zweifelhafte Instanzen besser ist als Traffic ins Nichts.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: eine Prüfung, die alles neu startet

Der schwierigste Gedanke dieses Abschnitts ist, dass die beiden Health Checks Unterschiedliches bewirken und dass eine gemeinsame Abhängigkeit in der falschen Prüfung einen kurzen Aussetzer in einen Ausfall verwandeln kann. Eine fehlgeschlagene Liveness Probe führt zum Neustart einer Instanz; eine fehlgeschlagene Readiness Probe nimmt sie nur aus der Rotation des Load Balancers.

- Angenommen, die Bestell-API läuft als 30 Instanzen, jede mit einer Liveness Probe, die nur dann „gesund“ meldet, wenn sie den gemeinsamen Message Broker erreicht.
- Der Broker ist etwa eine Minute lang nicht erreichbar, während ein Knoten ersetzt wird. Jede Probe schlägt fehl, obwohl die Instanzen selbst in Ordnung sind.
- Nach einigen Fehlschlägen in Folge schließt der Orchestrator, dass jede Instanz unrettbar hängt, und startet alle 30 gemeinsam neu.
- Der Broker kommt zurück, aber jetzt verbinden sich 30 frische Instanzen gleichzeitig mit ihm und füllen leere Caches aus der Bestelldatenbank. Während sie sich aufwärmen, kann niemand bestellen, und der Neustart-Sturm trifft beide, sodass aus einem einminütigen Aussetzer ein langer Ausfall wird.
- Hätte die Liveness Probe nur die Instanz selbst geprüft, wäre nichts neu gestartet worden: Die Instanzen hätten ihre warmen Caches behalten und weiter bedient; nur Arbeit, die den Broker braucht, hätte gewartet.
- Auch Readiness braucht Sorgfalt. Ruft die Readiness-Prüfung jeder Instanz ebenfalls diesen Broker auf, wird die ganze Flotte gemeinsam als nicht bereit markiert, und der Load Balancer kann nichts mehr irgendwohin schicken. Deshalb fallen manche Load Balancer offen aus: Wenn jede Instanz ungesund aussieht, schicken sie trotzdem Traffic an alle.
- **Die Lehre:** Halten Sie Liveness lokal und minimal, lassen Sie Readiness widerspiegeln, ob diese Instanz gerade bedienen kann, und behandeln Sie eine ausfallende gemeinsame Abhängigkeit mit den Mustern aus Abschnitt 9.4, nicht durch einen Neustart der Flotte.

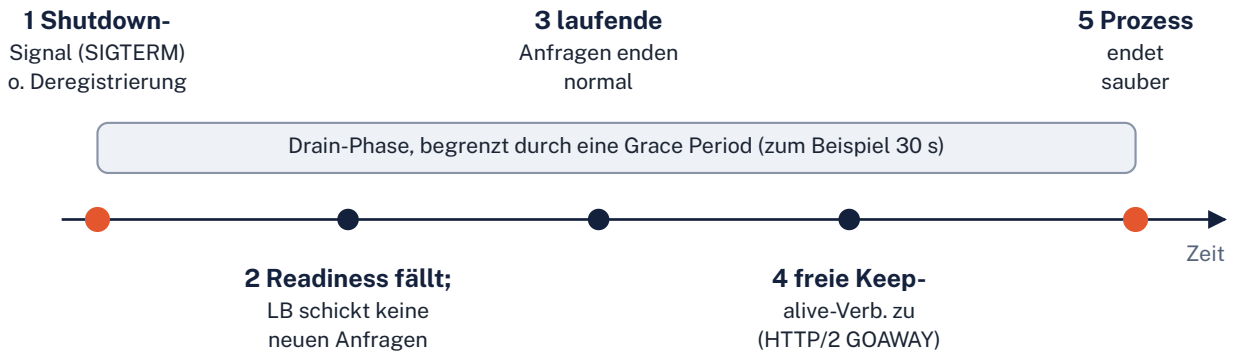
Connection Draining und kontrolliertes Herunterfahren

Instanzen verlassen den Pool ständig: Jedes Deployment ersetzt sie, das Autoscaling entfernt sie, und die Plattform verschiebt sie bei Wartungsarbeiten. Wird eine Instanz einfach beendet, scheitert jede Anfrage, die sie gerade bearbeitet, und Clients sehen Fehler bei einem eigentlich

routinemäßigen Release. Connection Draining (bei AWS Deregistration Delay genannt) nimmt die Instanz aus der Rotation des Balancers, sodass sie keine *neuen* Anfragen erhält, wartet, bis die bereits laufenden Anfragen abgeschlossen sind, und lässt sie erst dann stoppen.

Die Instanz muss mitwirken. Wenn sie ihr Shutdown-Signal erhält, sollte sie ihre Readiness-Prüfung nicht mehr bestehen, laufende Anfragen weiter bedienen, Clients auf Keep-alive-Verbindungen mitteilen, sich anderswohin zu wenden (ein `Connection: close`-Header bei HTTP/1.1, ein GOAWAY-Frame bei HTTP/2), Hintergrundarbeit abschließen oder zurückgeben und sich vor dem Ende der Grace Period beenden. In Kubernetes geschehen das Entfernen eines Pods aus dem Load Balancer und das Senden von SIGTERM gleichzeitig, daher kann kurz nach dem Signal noch eine Anfrage eintreffen; eine kurze Pre-Stop-Verzögerung von einigen Sekunden deckt diese Race Condition ab. Die Standard-Grace-Period beträgt 30 Sekunden, danach wird der Prozess beendet. Lang laufende Anfragen (große Uploads, Streaming-Antworten, WebSockets) brauchen entweder eine längere Grace Period oder ein Protokoll, über das sich Clients ohne Arbeitsverlust anderswo neu verbinden.

Kontrolliertes Herunterfahren in fünf Schritten



Draining macht das Entfernen einer Instanz von einer Fehlerquelle zu einem Nicht-Ereignis: Es kommt keine neue Arbeit an, vorhandene Arbeit wird abgeschlossen, dann beendet sich der Prozess.

Sticky Sessions und ihre Probleme

Ältere Webanwendungen hielten die Session jedes Nutzers (wer angemeldet ist, was im Warenkorb liegt) im Arbeitsspeicher des Servers, der sie erstellt hatte. Hinter einem Load Balancer funktioniert das nur, wenn alle Anfragen eines Nutzers zu diesem Server zurückgehen; deshalb bieten Balancer Sticky Sessions (Session-Affinität), meist über ein Cookie, das die Instanz benennt. Stickiness ist ein schneller Weg, eine zustandsbehaftete Anwendung hinter einen Balancer zu stellen, und sie hat eine berechtigte Verwendung als Performance-Hinweis, etwa um einen Nutzer auf einer Instanz zu halten, deren lokaler Cache seine Daten bereits enthält.

Als Mechanismus für Korrektheit macht sie Ärger. Die Last wird ungleichmäßig, weil einige intensive Nutzer oder ein großes Büro hinter einer IP-Adresse an einer Instanz festhängen, egal wie belastet sie ist. Fällt eine Instanz aus, verlieren ihre Nutzer ihre Sessions: Warenkörbe verschwinden, und Menschen werden abgemeldet. Herunterskalieren und jedes Deployment bewirken dasselbe, sofern Sessions nicht per Draining beendet werden, was so lange dauern kann wie die längste Session. Neue Instanzen, die das Autoscaling hinzufügt, erhalten nur neue Nutzer und entlasten die überlasteten Instanzen daher nur langsam. Und das System ist nicht mehr zustandslos im Sinne von Abschnitt 9.1, sodass jedes dieser Probleme beim Failover zwischen Zonen oder Regionen wieder auftaucht.

Den Session-Zustand auslagern

Die Abhilfe besteht darin, den Session-Zustand aus der Instanz herauszulösen, damit jede Instanz jede Anfrage bedienen kann. Es gibt drei Hauptoptionen, oft kombiniert. Ein **clientseitiges Token** trägt den Zustand mit jeder Anfrage: ein signiertes (und bei Bedarf verschlüsseltes) Cookie oder ein Token wie ein JWT mit der Identität des Nutzers und ein paar Daten. Server brauchen überhaupt keinen Speicher, aber Tokens müssen klein bleiben, lassen sich vor ihrem Ablauf nicht leicht widerrufen und müssen gegen Manipulation und Diebstahl geschützt werden (Abschnitt 8.3). Ein **gemeinsamer Session-Speicher**, typischerweise Redis oder Memcached, hält Sessions in einem schnellen externen Speicher, adressiert über eine Session-Kennung im Cookie; er bewältigt größeren Zustand und sofortiges Abmelden, auf Kosten eines Netzwerkaufrufs pro Anfrage und einer Abhängigkeit, die selbst hochverfügbar sein muss. Eine **Datenbank** eignet sich für Zustand, der alles überstehen muss, etwa einen Warenkorb, den der Nutzer morgen wiederzufinden erwartet; das sind eigentlich Geschäftsdaten und keine Session-Daten.

Eine praktische Aufteilung für die Essenslieferplattform: die Identität in einem kurzlebigen signierten Token, der Warenkorb in der Datenbank des Bestellservice (Nutzer wechseln vom Smartphone zum Laptop) und flüchtiger UI-Zustand im Client. Keine Instanz behält etwas, das eine spätere Anfrage braucht, daher kann der Load Balancer Least-Request-Balancing ohne Affinität verwenden, und Instanzen lassen sich nach Belieben hinzufügen, entfernen und ersetzen.

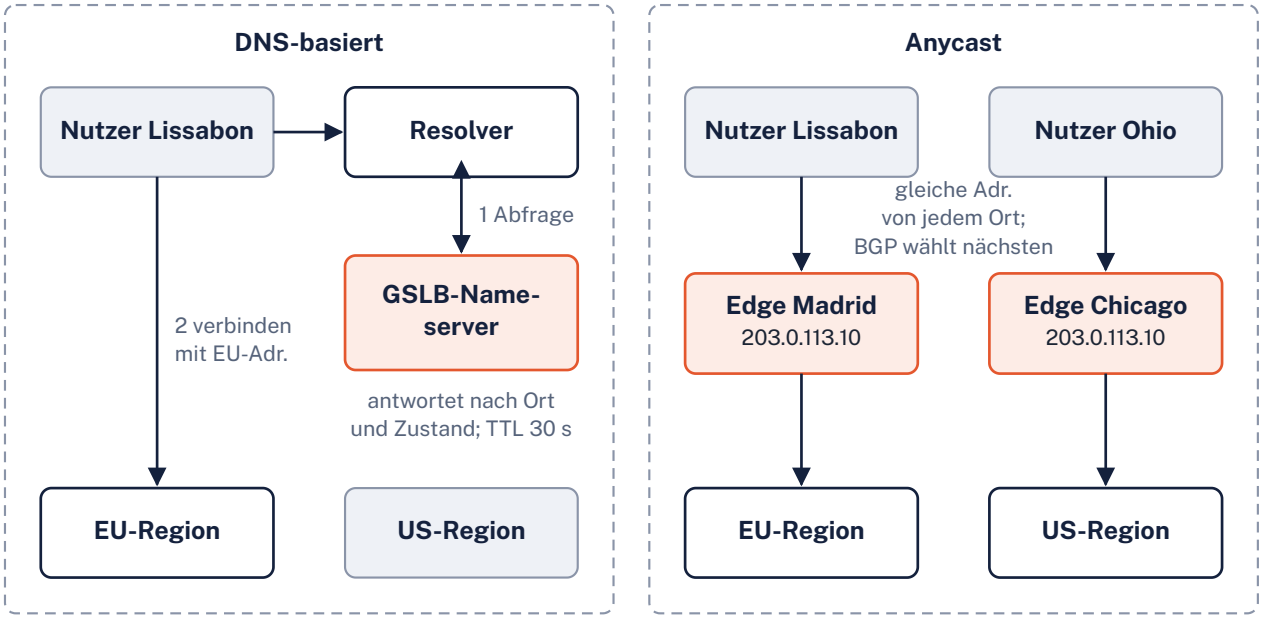
Globales Load Balancing: DNS und Anycast

Sobald ein System in mehreren Regionen läuft, muss etwas jeden Nutzer in eine geeignete schicken: die nächstgelegene gesunde Region oder die Region, die seine Daten hält. Globales Load Balancing erfolgt meist auf eine von zwei Arten, oft kombiniert.

DNS-basiertes Global Server Load Balancing nutzt einen autoritativen Nameserver, der denselben Namen mit unterschiedlichen Adressen beantwortet, abhängig davon, woher die Anfrage kommt, vom Gesundheitszustand jeder Region und von konfigurierten Gewichten: Ein Nutzer in Lissabon erhält die Adresse der EU-Region, ein Nutzer in Ohio die der US-Region. Es braucht kein besonderes Netzwerk, ist aber grob und reagiert langsam. Antworten werden von Resolvern mindestens für die Time to Live des Eintrags (oft 30 bis 300 Sekunden) und manchmal länger zwischengespeichert, daher kommen nach dem Ausfall einer Region einige Nutzer noch minutenlang dort an. Der Nameserver sieht den Standort des Resolvers statt den des Nutzers, wobei die Erweiterung EDNS Client Subnet (RFC 7871) es Resolvern erlaubt, einen Teil der Client-Adresse weiterzugeben.

Bei Anycast wird dieselbe IP-Adresse über das Routing-Protokoll des Internets (BGP) von vielen Standorten aus angekündigt, und die Pakete jedes Nutzers werden an den Standort zugestellt, der routingtechnisch am nächsten liegt. Failover bedeutet, die Ankündigung eines ausgefallenen Standorts zurückzuziehen, was schnell wirkt, ohne auf DNS-Caches zu warten. Anycast wird seit Langem für DNS-Root- und Resolver-Dienste genutzt; CDNs und die globalen Load Balancer der großen Cloud-Anbieter nutzen es, um Nutzer zu einem nahen Edge-Standort zu bringen, an dem Layer-7-Proxys die Verbindung terminieren und Anfragen über das Netzwerk des Anbieters an eine gesunde Region weiterleiten. Multi-Region-Design, Datenplatzierung und Failover sind Thema von Abschnitt 10.6.

DNS-basiertes Load Balancing und Anycast



DNS lenkt Nutzer, indem es mit unterschiedlichen Adressen antwortet; Anycast gibt allen dieselbe Adresse und lässt das Internet-Routing sie zum nächstgelegenen Standort bringen.

ACHTUNG

Häufige Fehler

- gRPC- oder HTTP/2-Traffic hinter einen Layer-4-Balancer stellen und sich wundern, warum neue Instanzen keine Anfragen erhalten.
- Round Robin für Services verwenden, deren Anfragen sich in den Kosten stark unterscheiden oder bei denen manche Instanzen langsamer sind als andere.
- Die Datenbank aus Liveness Probes aufrufen, sodass ein kurzer Datenbankaussetzer die ganze Flotte neu startet.
- Instanzen ohne Draining beenden, sodass jedes Deployment und jedes Herunterskalieren eine Welle von Fehlern erzeugt.
- Sich für die Korrektheit auf Sticky Sessions verlassen und dann Warenkörbe und Anmeldungen verlieren, wann immer eine Instanz ausfällt oder ersetzt wird.
- Annehmen, DNS-Failover geschehe sofort; Clients und Resolver können eine alte Antwort minutenlang weiterverwenden.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum erhält ein neu hinzugefügter gRPC-Server hinter einem Layer-4-Load-Balancer womöglich keinen Traffic?

Layer-4-Balancing wählt ein Backend pro Verbindung, und gRPC-Clients halten eine langlebige HTTP/2-Verbindung; vorhandene Clients schicken ihm also keine Anfragen, bis sie sich neu verbinden.

▼ Was ist der Unterschied in der Wirkung zwischen einer fehlgeschlagenen Liveness Probe und einer fehlgeschlagenen Readiness Probe?

Eine fehlgeschlagene Liveness Probe lässt den Orchestrator die Instanz neu starten; eine fehlgeschlagene Readiness Probe nimmt sie nur aus dem Load Balancer, bis sie wieder besteht.

▼ Warum schneiden viele unabhängige Load Balancer schlecht ab, die jeweils die am wenigsten belastete Instanz wählen?

Sie handeln auf Basis derselben veralteten Sicht und wählen alle dieselbe Instanz, die sie nacheinander überlasten; die bessere von zwei zufälligen Instanzen zu wählen, vermeidet diesen Herdentrieb.

▼ **Nennen Sie zwei Probleme, die entstehen, wenn Sessions mit Sticky Sessions im Arbeitsspeicher der Instanz gehalten werden.**

Nutzer verlieren ihre Sessions, wenn die Instanz ausfällt oder ersetzt wird, und die Last wird ungleichmäßig, weil intensive Nutzer an einer Instanz festhängen.

QUELLEN

- Michael Mitzenmacher: The Power of Two Choices in Randomized Load Balancing (IEEE TPDS, 2001): https://www.eecs.harvard.edu/~michaelm/postscripts/tpds2001.pdf
- Kubernetes documentation: Liveness, Readiness, and Startup Probes: https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/
- Envoy documentation: Supported load balancers: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancers
- NGINX documentation: HTTP load balancing: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/

ABSCHNITT 9.3 VON 9.8 · MITTELSTUFE · 29 MIN. LESEZEIT

Asynchronität, Queues und Backpressure

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.4 Performance und Skalierbarkeit
- 6.4 Ereignisgesteuert
- 6.5 Kommunikation und APIs

Begriffe von früher, die er verwendet: Littles Gesetz, Idempotent, Zeitliche Kopplung, Durchsatz

DIE IDEE IN KÜRZE

Eine gut besuchte Bäckerei verteilt zur Öffnungszeit Nummernzettel: Die Kundschaft wartet in einer geordneten Reihe, statt sich an der Theke zu drängen, und die Bäcker arbeiten in gleichmäßigem Tempo. Das ist eine Queue. Doch eine Schlange kann nur so lang werden, bis das Brot ausgeht oder die Leute aufgeben. Deshalb hört eine gute Bäckerei auch auf, Nummern auszugeben, wenn die Schlange zu lang ist, bedient Vorbestellungen zuerst, begrenzt jede Person auf eine vernünftige Zahl von Broten und sagt Nachzüglern ehrlich, sie sollen in zehn Minuten wiederkommen, statt sie vergeblich warten zu lassen.

WAS SIE WISSEN MÜSSEN

- Eine Queue zwischen Producer und Consumer verwandelt eine Lastspitze in einen Rückstand, den die Consumer in ihrem eigenen Tempo abarbeiten, sodass sie für den Durchschnitt dimensioniert werden können; der Preis ist Verzögerung für jedes Element hinter dem Rückstand.
- Competing Consumers skalieren Hintergrundarbeit durch zusätzliche Worker; die Zustellung erfolgt at least once, daher müssen Consumer idempotent sein, und Visibility Timeouts und Prefetch müssen zu den Verarbeitungszeiten passen.
- Eine Queue kann keinen dauerhaften Überschuss aufnehmen; begrenzen Sie jede Queue mit Littles Gesetz (maximale Länge = Verarbeitungsrate × akzeptable Wartezeit) und überwachen Sie neben der Länge auch das Alter der ältesten Nachricht.
- Backpressure bringt schnellere Stufen dazu, langsamer zu werden; ist eine begrenzte Queue voll, muss das System den Producer blockieren, neue Arbeit ablehnen oder Arbeit verwerfen, und es sollte diese Wahl bewusst treffen.
- Load Shedding lehnt einen Teil der Arbeit früh und kostengünstig ab, nach Priorität und Alter, damit der Goodput unter Überlast hoch bleibt.

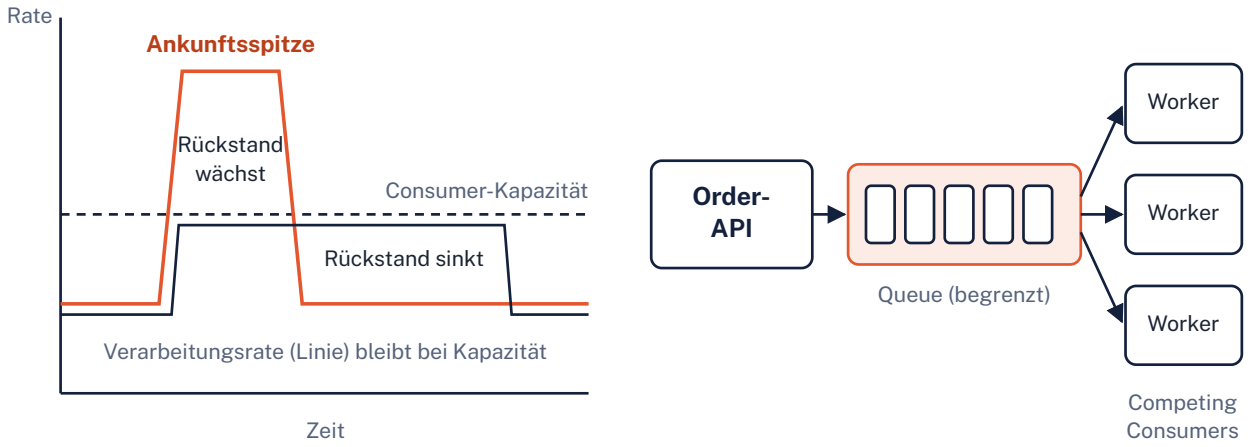
- Rate Limiting teilt Kapazität pro Client auf: Der Token Bucket erlaubt Bursts bis zu seiner Kapazität und eine gleichmäßige Nachfüllrate, der Leaky Bucket glättet die Ausgabe, und feste Fenster erlauben doppelte Bursts an den Fenstergrenzen.
- Beantworten Sie Limits mit 429 (oder 503, wenn der ganze Server überlastet ist) plus Retry-After, und leiten Sie wiederholt fehlschlagende Nachrichten in eine Dead Letter Queue, die jemandem gehört und überwacht wird.

Queues verwandeln Lastspitzen in Rückstände

Last kommt nie gleichmäßig an. Mittagsbestellungen ballen sich um 12:30, Ticketkäufer drücken alle um 10:00:00 auf den Knopf, und ein Batch-Job kippt um Mitternacht eine Million Events ab. Ein synchroner Service muss entweder Kapazität für die höchste Spitze haben oder während dieser Spitze ausfallen. Eine Queue zwischen dem Producer der Arbeit und dem Service, der sie erledigt, ändert die Rechnung: Während einer Spitze wartet die zusätzliche Arbeit in der Queue, und die Consumer verarbeiten sie in ihrem eigenen gleichmäßigen Tempo und holen auf, wenn die Spitze vorbei ist. Microsofts Katalog von Cloud-Mustern nennt das Queue-Based Load Leveling. Abschnitt 2.4 hat gezeigt, warum sich Warteschlangen bilden, sobald Ankünfte sich ballen; in diesem Abschnitt geht es darum, sie gezielt einzusetzen.

Der Nutzen ist real: Consumer können für den Durchschnitt statt für die Spitze dimensioniert werden, ein langsamer oder neu startender Consumer verliert keine Arbeit, und der Producer ist nicht mehr zeitlich an den Consumer gekoppelt (Abschnitt 6.5). Der Preis ist, dass die Arbeit nun später erledigt wird. Eine Queue verwandelt eine Spitze in einen *Rückstand*, und ein Rückstand ist schlicht eine Verzögerung, die jedes Element dahinter erfährt. Für den Versand von Bestätigungs-E-Mails oder die Aktualisierung eines Suchindex ist das in Ordnung, für die Zuweisung eines Kuriers zu einer Bestellung fragwürdig und für eine Zahlung, deren Bestätigung die Kundin oder der Kunde erwartet, inakzeptabel. Die erste Entwurfsfrage für jede Queue lautet daher: Wie lange darf ein Element warten, und was geschieht mit Elementen, die länger warten würden?

Eine Queue fängt eine Lastspitze ab



Während der Spitze nimmt die Queue die Ankünfte über der Consumer-Kapazität auf; danach bauen die Consumer den Rückstand in ihrem gleichmäßigen Tempo ab.

Work Queues und Competing Consumers

Am häufigsten ist eine Work Queue mit Competing Consumers: Mehrere identische Worker lesen aus einer Queue, jede Nachricht geht an genau einen von ihnen, und der Durchsatz wächst, indem man Worker hinzufügt. Das ist die X-Achse des Scale Cube, angewandt auf Hintergrundarbeit, und sie skaliert natürlich anhand des richtigen Signals: Rückstand pro Worker oder Alter der ältesten Nachricht.

Für die Korrektheit kommt es auf die Mechanik an. Ein Worker nimmt eine Nachricht, verarbeitet sie und bestätigt sie dann; stürzt der Worker vorher ab, macht der Broker die Nachricht nach einem Timeout wieder sichtbar (SQS nennt das den Visibility Timeout), und ein anderer Worker verarbeitet sie. Die Zustellung erfolgt daher at least once, also müssen Consumer idempotent sein (Abschnitt 6.4). Ein Visibility Timeout, der kürzer als die tatsächliche Verarbeitungszeit ist, führt dazu, dass dieselbe Nachricht zweimal parallel verarbeitet wird. Die Reihenfolge geht verloren, sobald es mehr als einen Consumer gibt, es sei denn, die Nachrichten werden nach Schlüssel partitioniert; dann begrenzt die Zahl der Partitionen die Zahl sinnvoller Consumer. Und Prefetching (ein Worker nimmt mehrere Nachrichten auf einmal, um Roundtrips zu sparen) sollte bei langsamen Aufgaben niedrig bleiben, sonst hortet ein Worker Nachrichten, die untätige Worker hätten verarbeiten können.

Eine Queue schafft keine Kapazität

Eine Queue fängt einen vorübergehenden Überschuss auf, aber keinen dauerhaften. Kommt Arbeit lange genug schneller an, als sie erledigt wird, wächst der Rückstand ohne Grenze, und Littles Gesetz ($L = \lambda \times W$) besagt, dass die Wartezeit mit ihm wächst. Eine unbegrenzte In-Memory-Queue erschöpft irgendwann den Speicher und bringt den Prozess zum Absturz, wobei alles darin verloren geht; eine unbegrenzte persistente Queue füllt ihre Festplatte oder hält Arbeit noch Stunden, nachdem sich niemand mehr dafür interessiert. Schlimmer noch: Die Queue verbirgt das Problem. Producer erhalten schnelle Bestätigungen, die Fehlerraten bleiben bei null, und das einzige Symptom ist, dass Ergebnisse immer später eintreffen. Fachleute bei Amazon beschreiben solche Rückstände als potenziell unüberwindbar, weil ein Service, der weit genug zurückfällt, seine gesamte Kapazität für Arbeit aufwenden kann, die seine Clients längst aufgegeben haben.

Daraus folgen zwei Praktiken. Erstens: Messen Sie neben der Queue-Länge auch das Alter der ältesten Nachricht; das Alter ist das, was Benutzer erleben, und es reagiert sofort, wenn Consumer ins Stocken geraten. Zweitens: Geben Sie jeder Queue eine Grenze, die Sie mit Littles Gesetz aus der akzeptablen Wartezeit ableiten, und legen Sie im Voraus fest, was passiert, wenn die Grenze erreicht ist. Um diese Entscheidung geht es in den nächsten drei Teilen dieses Abschnitts.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Bestell-Queue für einen Ticketverkaufsstart dimensionieren

Tickets für ein Stadionkonzert gehen um 10:00 in den Verkauf. Bestellungen landen in einer Queue; Worker autorisieren die Zahlung und weisen Plätze zu, jeweils in 250 ms. Die Prognose lautet 600 Bestellungen pro Sekunde in den ersten zwei Minuten. Den Kunden ist eine Antwort innerhalb von zwei Minuten versprochen. Es gibt 25.000 Tickets, und eine Bestellung umfasst im Durchschnitt 2,5 Tickets.

Worker-Kapazität mit 40 Workern	40 ÷ 0,25 s = 160 Bestellungen pro Sekunde
Rückstand nach der zweiminütigen Spitze	(600 − 160) × 120 s = 52.800 wartende Bestellungen
Wartezeit der letzten Bestellung (Littles Gesetz, W = L ÷ λ)	52.800 ÷ 160 = 330 s, fünfeinhalb Minuten: Das Versprechen ist gebrochen
Längste Queue, die das Versprechen hält	L = λ × W = 160 × 120 = 19.200 Bestellungen
Mehr Worker?	Der Vertrag mit dem Zahlungsanbieter erlaubt 300 Anfragen pro Sekunde, also höchstens 75 Worker (300 Bestellungen pro Sekunde). Der Rückstand wird (600 − 300) × 120 = 36.000 und die Wartezeit 36.000 ÷ 300 = 120 s: Das Versprechen wird gerade so gehalten, ohne Reserve
Was der Bestand zulässt	25.000 ÷ 2,5 = 10.000 Bestellungen können erfolgreich sein. Bei 600 pro Sekunde treffen so viele in weniger als 17 Sekunden ein; die übrigen rund 62.000 Bestellungen in die Queue zu stellen, verzögert nur das unvermeidliche „ausverkauft“
Entscheidung	75 Worker; ein Warteraum vor der Bestellseite lässt 300 Käufer pro Sekunde ein; die Bestell-Queue ist auf 12.000 begrenzt (der Bestand plus 20 % für fehlgeschlagene Zahlungen); sobald die Reservierungen den Bestand abdecken, erhalten neue Bestellungen sofort die Antwort „ausverkauft, tragen Sie sich in die Warteliste ein“, statt in die Queue zu gelangen

Littles Gesetz verwandelt ein Versprechen über die Wartezeit in eine maximale Queue-Länge. Jede Queue, die darüber hinaus wachsen darf, ist ein gebrochenes Versprechen, das nur noch nicht verkündet wurde.

Das Rate Limit des nachgelagerten Systems, nicht die Zahl der Worker, die die Plattform starten könnte, bestimmte die tatsächliche Kapazität; zusätzliche Consumer stießen daher an eine harte Obergrenze.

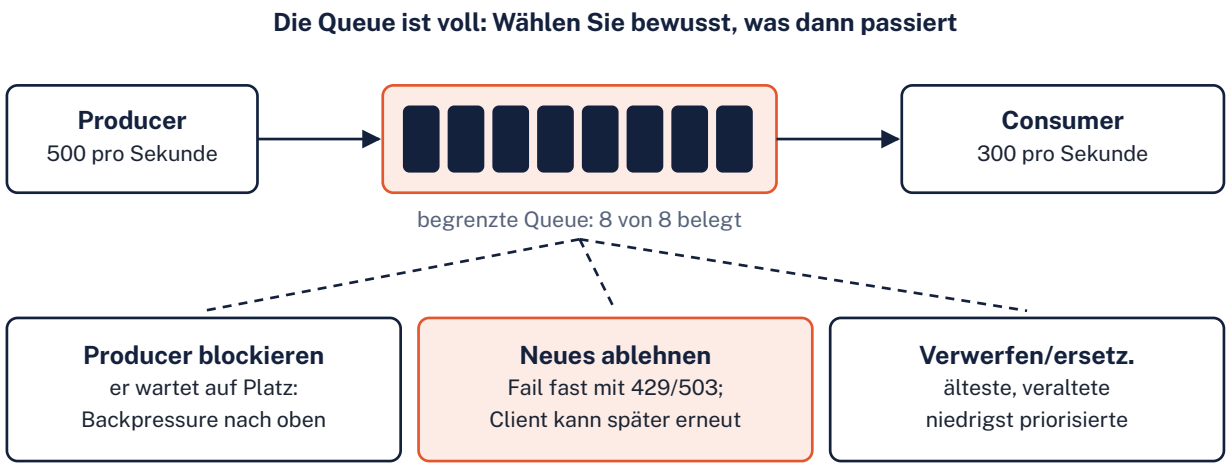
Die beste Grenze für eine Queue ist oft eine geschäftliche: Es hat keinen Sinn, Arbeit vorzuhalten, die unmöglich erfolgreich sein kann.

Backpressure: Producer zum Verlangsamen bringen

Backpressure ist jeder Mechanismus, mit dem eine langsamere nachgelagerte Stufe eine schnellere vorgelagerte Stufe zum Verlangsamen bringt, statt Arbeit dazwischen auflaufen zu lassen. Der Begriff stammt von Flüssigkeiten in Rohren, und die Idee ist uns bei Pipes and Filters (Abschnitt 5.4) begegnet. Sie ist in viele Schichten des Stacks eingebaut, den Sie bereits nutzen. Die Flusskontrolle von TCP stoppt einen Sender, wenn der Puffer des Empfängers voll ist. Eine begrenzte blockierende Queue lässt das `put` des Producers auf freien Platz warten. Pull-basierte Consumer wie bei Kafka holen Arbeit erst, wenn sie bereit sind, sodass das Log des Brokers statt des Speichers des Consumers den Rückstand hält, und ein Kafka-Producer blockiert oder schlägt fehl, wenn sein eigener Sendepuffer voll ist. Die Reactive-Streams-Spezifikation lässt einen Subscriber eine bestimmte Anzahl von Elementen anfordern, sodass ein Stream nie mehr liefert, als der Subscriber verlangt hat.

Backpressure funktioniert gut innerhalb eines Prozesses oder entlang einer Pipeline. Über eine Request-Response-Grenze mit menschlichen Benutzern wird daraus meist Ablehnung: Ein überlasteter HTTP-Service kann ein Smartphone nicht höflich warten lassen, also antwortet er mit `429 Too Many Requests` oder `503 Service Unavailable`, idealerweise mit einem `Retry-After`-Header, und verlässt sich darauf, dass der Client zurückweicht. Ist eine begrenzte Queue voll, gibt es nur drei Möglichkeiten: den Producer blockieren, die neue Arbeit ablehnen oder etwas bereits Angenommenes verwerfen. Jede ist irgendwo richtig; der Fehler ist, keine Wahl zu treffen.

Drei Reaktionen auf eine volle Queue



Ist eine begrenzte Queue voll, muss das System blockieren, ablehnen oder verwerfen; eine bewusste Wahl hält die Überlast in Grenzen.

Was passiert, wenn eine Queue voll ist: die Wahl

Strategie	Verhalten	Geeignet für	Risiko
Producer blockieren	Der Producer wartet, bis Platz frei ist	Pipelines und Batch-Jobs, bei denen der Producer pausieren kann	Blockierte Producer halten Threads und Verbindungen; über Services hinweg pflanzt sich die Verlangsamung so nach oben fort
Neue Arbeit ablehnen	Fail fast mit einem Fehler wie 429 oder 503	Interaktive Anfragen; der Client kann es erneut versuchen, warten oder eine Meldung anzeigen	Clients, die sofort erneut anfragen, verwandeln Ablehnung in mehr Last (Abschnitt 9.4)
Älteste verwerfen	Das Element verwerfen, das am längsten gewartet hat	Telemetrie, Standortupdates, Live-Preise: Die neuesten Daten sind am wertvollsten	Stiller Verlust; muss gezählt und sichtbar gemacht werden
Nach Priorität verwerfen	Die unwichtigsten Elemente zuerst verwerfen	Gemischte Lasten: Checkouts behalten, Analytics-Events verwerfen	Niedrige Prioritäten können verhungern; Prioritäten müssen ehrlich vergeben werden
In persistenten Speicher auslagern	Überlauf auf Festplatte oder in eine persistente Queue	Arbeit, die irgendwann erledigt werden muss, aber Stunden warten kann	Nur eine größere Queue: braucht trotzdem eine Grenze und eine Schätzung der Abbauzeit

Load Shedding und Priorisierung

Ein Server unter mehr Last, als er bewältigen kann, schneidet schlechter ab, als wenn er seine Kapazität bedienen und den Rest ablehnen würde. Er nimmt alles an, jede Anfrage wartet in seinen internen Queues, die Antwortzeiten steigen über die Timeouts der Clients, und am Ende verwendet er seine gesamte Kapazität auf Anfragen, deren Clients bereits aufgegeben haben. Der Durchsatz bleibt hoch, während der *Goodput*, die Rate der Anfragen, die rechtzeitig genug abgeschlossen werden, um nützlich zu sein, gegen null einbricht. Load Shedding verhindert das, indem es einige Anfragen bewusst ablehnt, damit die übrigen erfolgreich sind.

Gutes Load Shedding hat drei Eigenschaften. Es ist **früh und kostengünstig**: Eine Anfrage abzulehnen muss weit weniger kosten, als sie zu bedienen; es geschieht daher an der Eingangstür, vor aufwendiger Authentifizierung oder Datenbankarbeit, anhand eines billigen Signals wie laufender Anfragen oder des Queue-Alters. Es ist **priorisiert**: Googles SRE-Buch beschreibt, jede Anfrage mit einer Kritikalität zu kennzeichnen (von CRITICAL_PLUS bis hinunter zu SHEDDABLE), damit Batch- und Prefetch-Verkehr vor nutzerseitigem Verkehr verworfen wird. In der Essenslieferplattform haben Checkouts und Standortupdates der Kuriere Vorrang vor Empfehlungen, die wiederum Vorrang vor Analytics-Events haben. Und es ist **zeitbewusst**: Eine Anfrage, die bereits länger gewartet hat als der Timeout ihres Aufrufers, sollte verworfen statt verarbeitet werden. Fachleute bei Facebook haben zwei Verfeinerungen beschrieben: eine Controlled-Delay-Queue, die nur eine kurze Wartezeit zulässt, wenn eine Queue in letzter Zeit nicht leer war, und den Wechsel zu Last-in-first-out unter Last, sodass die frischesten Anfragen, deren Benutzer noch warten, zuerst bedient werden.

Ein weiterer Schritt besteht darin, das Limit selbst anzupassen. Feste Nebenläufigkeitslimits sind an einem guten Tag entweder zu niedrig oder an einem schlechten zu hoch; adaptive Nebenläufigkeitsbegrenzer erhöhen die erlaubte Zahl laufender Anfragen, solange die Latenz niedrig bleibt, und senken sie, wenn die Latenz steigt – eine Idee, die der TCP-Überlastkontrolle entlehnt ist. Ganz am Eingang eines Systems kann Zugangskontrolle die Form eines virtuellen Warteraums annehmen, wie beim Ticketverkaufsstart oben: Benutzer werden am Rand in einer Warteschlange gehalten und in dem Tempo eingelassen, das der Kern bedienen kann – Load Shedding, präsentiert als Fairness.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: beschäftigt, aber nutzlos

Der schwierigste Gedanke dieses Abschnitts ist, dass ein überlasteter Server, der jede Anfrage annimmt, am Ende fast keine nützliche Arbeit leisten kann, obwohl er nie aufhört zu arbeiten. Einige Anfragen absichtlich abzulehnen, Load Shedding, ist das, was den Rest nützlich hält.

- Angenommen, der Checkout der Essenslieferplattform kann 100 Anfragen pro Sekunde abschließen, und die mobile App gibt jede Anfrage auf, die nicht innerhalb von 2 Sekunden beantwortet ist.
- Während einer Werbeaktion kommen 150 Anfragen pro Sekunde an. Der Service nimmt sie alle an, also wächst seine interne Queue jede Sekunde um 50.
- Nach 4 Sekunden warten etwa 200. Bei 100 pro Sekunde wartet eine neue Anfrage etwa 2 Sekunden, bevor die Arbeit an ihr beginnt; bis dahin hat ihre App bereits aufgegeben.
- Die Queue wächst weiter, also wird von da an fast jede Anfrage zu spät fertig. Der Service schließt weiterhin 100 pro Sekunde ab, sein Durchsatz sieht also gesund aus, aber sein Goodput, die rechtzeitig genug abgeschlossenen Anfragen, ist gegen null gefallen.
- Fügen Sie nun Load Shedding an der Eingangstür hinzu: Sobald die laufenden Anfragen ein Limit erreichen, das so gewählt ist, dass angenommene Arbeit rechtzeitig fertig wird, werden neue sofort mit einem 503-Fehler („Service Unavailable“) abgelehnt, was weit weniger kostet, als sie zu bedienen. Etwa 50 pro Sekunde werden abgelehnt; die 100 angenommenen werden rechtzeitig beantwortet.
- Gutes Load Shedding ist außerdem priorisiert: Ein Server, der mehrere Arten von Anfragen bearbeitet, lehnt Empfehlungen und Analytics-Events vor Checkouts ab. Er verwirft auch jede Anfrage, die bereits länger gewartet hat als der Timeout ihres Aufrufers.
- **Die Lehre:** Unter Überlast besteht die eigentliche Wahl nicht zwischen „alle bedienen“ und „einige bedienen“, sondern zwischen „einige absichtlich ablehnen“ und „fast niemanden rechtzeitig bedienen“. Lehnen Sie früh, kostengünstig und nach Priorität ab.

Algorithmen für Rate Limiting

Rate Limiting beschränkt, wie viel Arbeit ein bestimmter Client, API-Schlüssel, Mandant oder eine IP-Adresse in einem Zeitraum einreichen darf. Es unterscheidet sich im Zweck von Load Shedding: Load Shedding schützt den Server vor völliger Überlast, egal wer sie verursacht,

während Rate Limiting Kapazität fair verteilt und vor jedem einzelnen Client schützt, ob fehlerhaft oder böswillig. Beide antworten auf dieselbe Weise, meist mit HTTP 429. Vier Algorithmen dominieren.

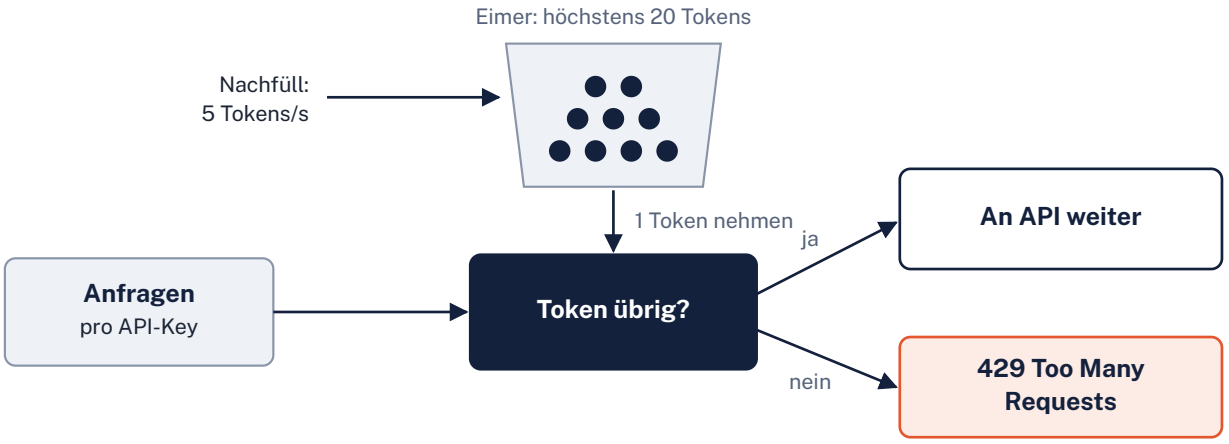
Rate-Limiting-Algorithmen im Vergleich

Algorithmus	So funktioniert es	Bursts	Zustand pro Schlüssel	Hinweise
Token Bucket	Tokens werden mit Rate r bis zur Kapazität b hinzugefügt; jede Anfrage verbraucht eines	Erlaubt Bursts bis b, dann im Mittel r	Zwei Zahlen: Tokens und Zeitpunkt der letzten Nachfüllung	Am weitesten verbreitet; Burst und Dauerrate werden getrennt eingestellt
Leaky Bucket	Anfragen gelangen in einen Eimer, der sich mit konstanter Rate leert; ist er voll, werden neue Anfragen abgelehnt	Die Ausgabe ist völlig gleichmäßig; Bursts warten oder werden abgelehnt	Eine Queue oder ein Zähler mit Zeitstempel	Für Traffic Shaping verwendet; das Request Limiting von NGINX basiert darauf
Fixed Window Counter	Anfragen pro Kalenderfenster zählen (pro Minute); Zurücksetzen an der Grenze	Bis zum Doppelten des Limits über eine Grenze hinweg	Ein Zähler	Am einfachsten; 100 Anfragen um 12:00:59 und 100 um 12:01:00 kommen beide durch
Sliding Window Log	Den Zeitstempel jeder Anfrage im letzten Fenster speichern; diese zählen	Exakt	Ein Eintrag pro Anfrage	Präzise, aber speicherhungrig bei hohen Raten
Sliding Window Counter	Den Zählerstand des vorigen Fensters danach gewichten, wie viel davon noch überlappt, und den aktuellen Zählerstand addieren	Gute Näherung	Zwei Zähler	Nach 30 s der Minute, 80 in der Vorminute, 40 aktuell: 80 × 0,5 + 40 = 80

Der Token Bucket verdient einen genaueren Blick, weil er der übliche Standard ist. Der Eimer fasst bis zu b Tokens und erhält r Tokens pro Sekunde. Jede Anfrage nimmt ein Token; ist keines übrig, wird sie abgelehnt (oder in der Shaping-Variante verzögert). Ein Client, der ruhig war, kann daher einen Burst von b Anfragen auf einmal senden, was zu echten Clients passt, etwa einer mobilen App, die einen Bildschirm öffnet, während er über jeden längeren Zeitraum

im Mittel r nicht überschreiten kann. Der Leaky Bucket ist das Spiegelbild: Er glättet die Ausgabe auf eine konstante Rate, was nützlich ist, um ein nachgelagertes System zu schützen, das überhaupt keine Bursts verkraftet.

Der Token Bucket



Jede Anfrage muss ein Token nehmen; die Nachfüllrate bestimmt die Dauerrate, die Kapazität des Eimers den größten Burst.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein Token Bucket für eine Partner-API

Ein API-Schlüssel für Restaurantpartner ist durch einen Token Bucket mit Kapazität b = 20 und Nachfüllrate r = 5 Tokens pro Sekunde begrenzt (im Mittel 300 pro Minute). Der Eimer ist bei t = 0 voll.

t = 0: ein Burst von 30 Anfragen	Die ersten 20 nehmen die 20 Tokens und werden weitergeleitet; 10 werden mit 429 abgelehnt
t = 0 bis 1 s	5 neue Tokens kommen hinzu (5 pro Sekunde)
t = 1 bis 10 s: gleichmäßig 8 Anfragen pro Sekunde	Etwa 5 pro Sekunde werden angenommen und 3 abgelehnt; der Eimer bleibt nahezu leer
t = 10 bis 14 s: keine Anfragen	Der Eimer füllt sich mit 5 pro Sekunde und ist bei t = 14 wieder voll (20)
Obergrenze über einen beliebigen Zeitraum T	Höchstens b + r × T Anfragen: in beliebigen 60 Sekunden 20 + 5 × 60 = 320, sodass der langfristige Mittelwert nie 5 pro Sekunde übersteigen kann
Retry-After für eine abgelehnte Anfrage	Das nächste Token kommt in 1 ÷ r = 0,2 s, aber der Header zählt ganze Sekunden, also lautet die Antwort Retry-After: 1

Die Kapazität bestimmt den Burst, den ein wohlerzogener Client nach einer ruhigen Phase senden darf; die Nachfüllrate bestimmt die Dauerrate. Wählen Sie beide getrennt.

Ein Client, der Retry-After beachtet, mit etwas Jitter, pendelt sich bei der Nachfüllrate ein; einer, der sofort erneut anfragt, sammelt nur weitere 429er.

```
import math, time

class TokenBucket:
    def __init__(self, rate, capacity):
        self.rate = rate              # tokens added per second
        self.capacity = capacity      # largest burst allowed
        self.tokens = capacity        # start full
        self.updated = time.monotonic()

    def allow(self, cost=1):
        now = time.monotonic()
        elapsed = now - self.updated
        self.tokens = min(self.capacity, self.tokens + elapsed * self.rate)
        self.updated = now
        if self.tokens ⩾ cost:
            self.tokens -= cost
            return True, 0
        wait = (cost - self.tokens) / self.rate   # seconds until enough tokens
        return False, max(1, math.ceil(wait))      # for the Retry-After header

bucket = TokenBucket(rate=5, capacity=20)        # one bucket per API key
allowed, retry_after = bucket.allow()
```

Ein Token Bucket mit verzögertem Nachfüllen (Lazy Refill): Tokens werden für die seit dem letzten Aufruf vergangene Zeit aufgefüllt, sodass kein Timer nötig ist. In Produktion liegt der Zustand in einem gemeinsamen Speicher wie Redis und wird atomar aktualisiert.

HTTP 429, Retry-After und wo Limits durchgesetzt werden

HTTP hat Standardsignale für beide Situationen. `429 Too Many Requests`, definiert in RFC 6585 (2012), teilt einem Client mit, dass er in einem bestimmten Zeitraum zu viele Anfragen gesendet hat. `503 Service Unavailable` besagt, dass der Server als Ganzes vorübergehend überlastet ist oder gewartet wird. Beide können einen `Retry-After`-Header (RFC 9110) tragen, der eine Anzahl von Sekunden oder ein Datum angibt, nach dem ein erneuter Versuch sinnvoll ist. Viele APIs geben außerdem Header zurück, die das Limit, das verbleibende Kontingent und den Rücksetzzeitpunkt ankündigen, traditionell `X-RateLimit-*`; ein IETF-Entwurf standardisiert die Felder `RateLimit` und `RateLimit-Policy`. Wohlerzogene Clients beachten `Retry-After`, fügen Jitter hinzu, damit abgelehnte Clients nicht alle im selben Augenblick zurückkommen, und behandeln wiederholte 429er als Grund, langsamer zu werden, statt es heftiger zu versuchen (Abschnitt 9.4).

Limits lassen sich auf mehreren Schichten durchsetzen, und ausgereifte Systeme nutzen mehr als eine: am Edge oder API-Gateway pro IP-Adresse und API-Schlüssel, um Missbrauch und außer Kontrolle geratene Clients zu stoppen, bevor sie etwas kosten; pro Mandant, damit ein großer Kunde die anderen nicht aushungern kann; und innerhalb jedes Service, um die eigene Kapazität und die eigenen Abhängigkeiten zu schützen. Ein globales Limit über viele Gateway-

Instanzen braucht gemeinsamen Zustand, meist atomare Zähler in Redis, was einen Netzwerkaufruf pro Anfrage hinzufügt; ein gängiger Kompromiss ist, jeder von N Instanzen ein lokales Limit von etwa 1 ÷ N des Gesamtwerts zu geben und eine gewisse Ungenauigkeit hinzunehmen. Limits sollten anhand gemessener Kapazität festgelegt und veröffentlicht werden, damit Partner darauf hin entwerfen können, und die Zahl der Ablehnungen sollte überwacht werden, denn ein plötzlicher Anstieg ist entweder ein Angriff, ein Fehler in einem Client oder ein Limit, das nicht mehr zur Realität passt.

Dead Letter Queues

Manche Nachrichten scheitern jedes Mal: eine fehlerhafte Nutzlast, eine Bestellung für ein Restaurant, das es nicht mehr gibt, ein Bug, den eine bestimmte Eingabe auslöst. Endlos wiederholt, blockiert eine solche Poison Message eine Partition oder verschwendet für immer einen Worker. Eine Dead Letter Queue (eingeführt in Abschnitt 6.4) nimmt Nachrichten auf, die eine festgelegte Anzahl von Malen gescheitert sind (SQS nennt das den Maximum Receive Count in einer Redrive Policy), damit die übrige Arbeit weiterlaufen kann.

Eine Dead Letter Queue ist ein Werkzeug für Defekte, nicht für Überlast: Landen Nachrichten dort, weil Consumer zu langsam sind, liegt die Lösung in Kapazität oder Backpressure, nicht in einer zweiten Queue. Jede Nachricht in einer DLQ bedeutet, dass etwas, das das Geschäft erwartet hat, nicht passiert ist. Die DLQ braucht daher einen Verantwortlichen, einen Alarm auf ihre Tiefe oder ihr Alter, Werkzeuge, um Nachrichten mit ihrem Fehler und ihrer Versuchszahl zu untersuchen, und einen Weg, sie erneut abzuspielen, sobald die Ursache behoben ist. Legen Sie im Voraus fest, was ein verspätetes Replay für die Reihenfolge bedeutet und für Nachrichten, deren geschäftlicher Moment vorbei ist: Eine Erstattung kann einen Tag später nachgeholt werden, eine Kurierzuweisung nicht.

ACHTUNG

Häufige Fehler

- Unbegrenzte Queues, im Speicher oder auf der Festplatte, die eine vorübergehende Überlast in einen Absturz oder in stundenlange veraltete Arbeit verwandeln.
- Die Queue-Länge überwachen, aber nicht das Alter der ältesten Nachricht, sodass festhängende Consumer unbemerkt bleiben, während die Länge normal aussieht.
- Load Shedding erst nach dem teuren Teil der Arbeit, sodass eine Ablehnung fast so viel kostet wie ein Erfolg.
- Rate Limits mit festen Fenstern, die Clients über eine Fenstergrenze hinweg das Doppelte des Limits senden lassen.
- Clients, die 429 und Retry-After ignorieren und sofort erneut anfragen, wodurch aus einem Limit eine Flut wird.
- Visibility Timeouts, die kürzer als die Verarbeitungszeit sind, sodass eine Nachricht von zwei Workern gleichzeitig verarbeitet wird.
- Eine Dead Letter Queue als Überlauftank nutzen, statt die Kapazität zu korrigieren, und sie ohne Verantwortlichen und ohne Überwachung lassen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Worker verarbeiten 200 Nachrichten pro Sekunde, und Nachrichten dürfen höchstens 30 Sekunden warten. Was ist die größte akzeptable Queue-Länge?

Nach Littles Gesetz 200 × 30 = 6000 Nachrichten; eine längere Queue bedeutet, dass manche Nachrichten länger als 30 Sekunden warten.

▼ Warum ist das Alter der ältesten Nachricht ein besserer Alarm als die Queue-Länge?

Das Alter misst die Verzögerung, die Benutzer erleben, und steigt sofort, wenn Consumer stocken, während eine normal wirkende Länge eine festhängende Queue oder einen langsamen Abbau verbergen kann.

▼ Ein Token Bucket hat die Kapazität 50 und füllt sich mit 10 pro Sekunde. Wie viele Anfragen kann ein Client in der ersten Sekunde nach einer langen ruhigen Phase stellen?

Etwa 60: die 50 Tokens, die bereits im Eimer liegen, plus die 10, die in dieser Sekunde hinzukommen.

▼ **Was ist der Unterschied zwischen Load Shedding und Rate Limiting?**

Load Shedding schützt den Server vor völliger Überlast, egal wer sie verursacht; Rate Limiting begrenzt, was jeder Client senden darf, damit Kapazität fair geteilt wird.

QUELLEN

- Microsoft Azure Architecture Center: Queue-Based Load Leveling pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling
- Google SRE book: Handling Overload: https://sre.google/sre-book/handling-overload/
- RFC 6585: Additional HTTP Status Codes (429 Too Many Requests): https://www.rfc-editor.org/rfc/rfc6585
- Ben Maurer: Fail at Scale (ACM Queue, 2015): https://queue.acm.org/detail.cfm?id=2839461

ABSCHNITT 9.4 VON 9.8 · FORTGESCHRITTEN · 32 MIN. LESEZEIT

Timeouts, Wiederholungen, Circuit Breaker, Bulkheads und Graceful Degradation

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.4 Performance und Skalierbarkeit
- 6.1 Warum verteilen
- 6.5 Kommunikation und APIs

Begriffe von früher, die er verwendet: Teilausfall, Perzentil (p95, p99), Tail-Latenz, Littles Gesetz, Idempotenzschlüssel

DIE IDEE IN KÜRZE

Stellen Sie sich vor, Sie rufen einen Lieferanten an, der nie abnimmt. Wenn Sie den ganzen Tag in der Leitung bleiben, erledigen Sie nichts anderes, und wenn jede Kollegin und jeder Kollege dasselbe tut, steht das ganze Büro still. Vernünftige Büros legen nach ein paar Klingeltönen auf (ein Timeout), versuchen es etwas später erneut, aber nicht endlos (eine Wiederholung mit Pause), rufen einen Lieferanten, der offensichtlich geschlossen hat, nicht mehr an und prüfen in einer Stunde erneut (ein Circuit Breaker), sorgen dafür, dass ein Lieferant nicht alle Telefone belegen kann (ein Bulkhead), und haben einen Plan B, etwa die Preisliste von gestern (ein Fallback). Softwareservices brauchen genau dieselben Gewohnheiten.

WAS SIE WISSEN MÜSSEN

- Kaskadierende Ausfälle entstehen, wenn Aufrufer auf eine langsame Abhängigkeit warten, bis ihnen selbst Threads und Verbindungen ausgehen; jeder Integrationspunkt braucht gezielten Schutz (Nygard, Release It!).
- Setzen Sie explizite Connect- und Request-Timeouts anhand gemessener Latenzperzentile, die entlang der Aufrufkette kleiner werden, und geben Sie die verbleibende Deadline weiter, damit kein Service an einer aufgegebenen Anfrage arbeitet.
- Wiederholen Sie nur vorübergehende Fehler bei idempotenten Operationen, wenige Male, mit begrenztem exponentiellem Backoff und Full Jitter (Brooker, 2015).
- Wiederholungen multiplizieren sich über Schichten (drei Versuche auf drei Schichten ergeben 27 Aufrufe); wiederholen Sie nur auf einer Schicht und begrenzen Sie Wiederholungen mit einem Budget von etwa 10 % der Anfragen.

- Ein Circuit Breaker schlägt schnell fehl, solange eine Abhängigkeit ausgefallen ist: Er wechselt bei Fehlern von geschlossen zu offen und über halboffene Testaufrufe zurück zu geschlossen.
- Bulkheads geben jeder Abhängigkeit oder jedem Aufrufer einen eigenen begrenzten Pool, und Fallbacks halten die Kernabläufe am Laufen; Fallbacks müssen aber einfach, unabhängig vom ausgefallenen Teil und getestet sein und bei Sicherheitsentscheidungen geschlossen fehlschlagen (Fail-closed).
- Idempotenz macht Wiederholungen, Hedging und Replays sicher; Hedged Requests senken die Tail-Latenz für idempotente Lesezugriffe zu geringen, budgetierten Kosten an zusätzlicher Last.

Ausfälle sind normal; ihre Ausbreitung ist es nicht

In einem verteilten System ist immer irgendeine Abhängigkeit langsam oder fällt aus: Ein Zahlungsanbieter hat eine schlechte Minute, eine Datenbank macht ein Failover, eine Netzwerkverbindung verliert Pakete, ein Deployment geht schief. Abschnitt 6.1 hat gezeigt, warum sich solche Teilausfälle nicht verhindern lassen und warum ein Aufrufer oft nicht einmal erkennen kann, was passiert ist. Die architektonische Frage lautet daher nicht, wie man Ausfälle vermeidet, sondern wie man verhindert, dass ein lokaler Ausfall zu einem globalen wird. Ein kaskadierender Ausfall ist die typische Art, wie Systeme sterben: Eine langsame Abhängigkeit macht ihre Aufrufer langsam, deren Ressourcen füllen sich mit wartenden Anfragen, sie werden ihrerseits für ihre eigenen Aufrufer langsam, und binnen Sekunden hat ein Problem in einer unbedeutenden Komponente die ganze Plattform lahmgelegt.

Michael Nygard hat die Muster, die solche Ketten unterbrechen, in *Release It!* (2007; zweite Auflage 2018) katalogisiert und sich dabei auf Ausfälle gestützt, die er untersucht hatte. Seine zentrale Beobachtung ist, dass Integrationspunkte, also jede Stelle, an der ein System ein anderes aufruft, die Hauptquelle von Instabilität sind und dass jeder einzelne gezielten Schutz braucht. Die Muster in diesem Abschnitt (Timeouts, Deadlines, Wiederholungen mit Backoff, Circuit Breaker, Bulkheads, Fallbacks, Idempotenz und Hedging) sind der Standardwerkzeugkasten. Sie sind Taktiken im Sinne von Abschnitt 2.5: Jede ist eine einzelne Entscheidung, die auf Verfügbarkeit zielt, und jede hat ihren Preis.

Timeouts: das erste und wichtigste Muster

Ein Timeout begrenzt, wie lange ein Aufrufer wartet. Ohne Timeout hält ein Aufruf an eine Abhängigkeit, die nie antwortet, einen Thread, eine Verbindung und Speicher so lange, wie das Betriebssystem die Verbindung offen hält, und das können viele Minuten sein. Viele Client-Bibliotheken haben standardmäßig keinen Timeout (die weit verbreitete Python-Bibliothek

`requests` wartet unbegrenzt, wenn man nichts anderes angibt); das Fehlen eines Timeouts ist also eine Entwurfsentscheidung durch Unterlassen. Das ausgearbeitete Beispiel unten zeigt die Folgen.

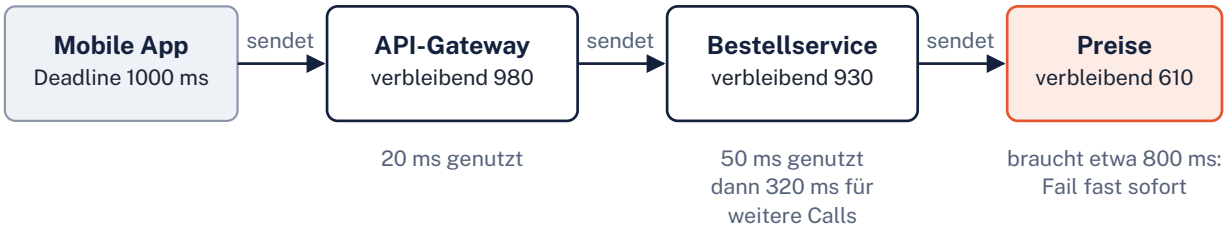
Meist kommt es auf zwei Timeouts an. Der **Connect-Timeout** begrenzt den Verbindungsaufbau und kann kurz sein, innerhalb eines Rechenzentrums wenige hundert Millisekunden, weil ein gesunder Server Verbindungen fast sofort annimmt. Der **Request-Timeout** begrenzt den gesamten Austausch und sollte aus Messungen abgeleitet werden: knapp über der hohen Perzentil-Latenz der Abhängigkeit (p99 oder p99.9) unter normaler Last, sodass er krankhafte Wartezeiten abschneidet, ohne gewöhnliche langsame Anfragen scheitern zu lassen. Er muss außerdem in das eigene Budget des Aufrufers passen. Gibt das Gateway nach 2 Sekunden auf, wartet der Bestellservice aber 10 Sekunden auf die Preisgestaltung, arbeitet der Bestellservice 8 Sekunden lang an einer Anfrage, auf die niemand wartet. Timeouts sollten daher entlang einer Aufrufkette kleiner werden, und das ist die Idee hinter Deadlines.

Deadlines und ihre Weitergabe

Eine Deadline ist ein absoluter Zeitpunkt, bis zu dem die gesamte Anfrage abgeschlossen sein muss. Deadline-Weitergabe gibt die verbleibende Zeit an jeden nachgelagerten Aufruf weiter, sodass jeder Service weiß, wie viel Budget noch übrig ist, statt einen eigenen festen Timeout anzuwenden. gRPC hat das eingebaut: Ein Client setzt eine Deadline, die verbleibende Zeit reist mit dem Aufruf mit, und ein Server kann prüfen, ob die Deadline überschritten ist, und die Arbeit einstellen. Über HTTP geben Teams einen Header mit den verbleibenden Millisekunden oder der absoluten Deadline weiter, idealerweise über dieselbe Middleware, die auch den Tracing-Kontext weitergibt (Abschnitt 9.6).

Die Weitergabe bringt zwei Vorteile. Ein Service kann schnell fehlschlagen, wenn die verbleibende Zeit offensichtlich zu kurz für die Arbeit ist, statt eine Datenbankabfrage zu starten, die nicht rechtzeitig fertig werden kann. Und wenn der Benutzer aufgibt, stoppt auch die Arbeit darunter, statt weiter Kapazität zu verbrauchen, und das genau während der Überlast, die die Verzögerung verursacht hat. Das ist Load Shedding toter Anfragen, einer der wirksamsten Schutzmechanismen, die ein System unter Stress haben kann.

Eine Deadline, die durch eine Aufrufkette weitergegeben wird



20 ms genutzt

50 ms genutzt
dann 320 ms für
weitere Calls

braucht etwa 800 ms:
Fail fast sofort

Jeder Hop sendet die Restzeit, keinen eigenen neuen Timeout

Jeder Hop gibt die verbleibende Zeit weiter; ein Service, der in der Restzeit nicht fertig werden kann, schlägt sofort fehl, statt Arbeit zu leisten, auf die niemand wartet.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: wie ein fehlender Timeout zum Totalausfall wird

Der Bestellservice der Essenslieferplattform läuft mit 6 Instanzen zu je 200 Worker-Threads: 1200 Threads. Zur Mittagszeit erhält er 400 Anfragen pro Sekunde: 100 Checkouts, die den Zahlungsanbieter aufrufen (normalerweise 300 ms, sodass ein Checkout seinen Thread insgesamt etwa 350 ms hält), und 300 andere Anfragen (Bestellhistorie, Warenkorb, Sendungsverfolgung), die 50 ms dauern. Für den Zahlungsclient ist kein Timeout konfiguriert. Um 12:31 antwortet der Zahlungsanbieter nicht mehr, hält die Verbindungen aber offen.

Normal belegte Threads (Littles Gesetz)	Checkout 100 × 0,35 s = 35; übriger Verkehr 300 × 0,05 s = 15; insgesamt 50 von 1200 (etwa 4 %)
Nach 12:31	Jeder Checkout hält seinen Thread nun unbegrenzt: Jede Sekunde hängen 100 weitere Threads fest
Zeit bis zur Erschöpfung	(1200 − 15 − 35) ÷ 100 ≈ 11,5 Sekunden: Innerhalb von etwa 12 Sekunden wartet jeder Thread auf den Zahlungsanbieter
Folge	Bestellhistorie, Warenkorb und Sendungsverfolgung, die Zahlungen nie aufrufen, warten jetzt auf Threads und laufen am Gateway in den Timeout. Ein Ausfall in 25 % des Verkehrs hat 100 % lahmgelegt
Und weiter oben	Die Verbindungen des Gateways zum Bestellservice stauen sich auf dieselbe Weise; Health Checks laufen in den Timeout, Instanzen werden als ungesund markiert, und wenn die Liveness-Probe betroffen ist, werden sie neu gestartet, kalt, in dieselben Bedingungen hinein
Mit einem Timeout von 2 Sekunden	Belegte Checkout-Threads: 100 × 2 s = 200 über die ganze Flotte (Littles Gesetz). Etwa 985 Threads bleiben für den übrigen Verkehr, also scheitert nur der Checkout, und zwar mit einem klaren Fehler nach 2 s
Ein Circuit Breaker kommt hinzu	Sobald etwa die Hälfte der letzten 20 Aufrufe gescheitert ist, öffnet der Breaker: Checkouts scheitern in Millisekunden und belegen fast keine Threads; ein Testaufruf alle 30 s erkennt die Erholung
Ein Bulkhead kommt hinzu	Die Begrenzung der Zahlungsaufrufe auf 40 Threads pro Instanz (240 insgesamt) garantiert mindestens 960 Threads für alles andere, unabhängig von den gesetzten Timeouts
Ein Fallback kommt hinzu	Der Checkout bietet dem Kunden eine andere Zahlungsart an oder erklärt, dass Kartenzahlungen verzögert sind, statt eines allgemeinen Fehlers

Die Rechnung ist wieder Littles Gesetz: belegte Threads = Anfragerate × Zeit, die jede Anfrage einen Thread hält. Ohne Timeout ist die Haltezeit unbegrenzt, also auch die Zahl der benötigten Threads.

Timeouts begrenzen, wie lange; Bulkheads begrenzen, wie viele; Circuit Breaker beenden die Aufrufe ganz. Jedes Muster schließt eine andere Lücke, deshalb werden sie zusammen eingesetzt.

Nygards *Release It!* beschreibt echte Ausfälle, die genau diesem Muster folgten, oft ausgelöst durch einen Integrationspunkt, den niemand für kritisch hielt.

Erschöpfung des Thread-Pools

Ohne Timeout hält ein langsamer Zahlungsanbieter jeden Thread des Bestellservice, sodass selbst Anfragen scheitern, die Zahlungen nie berühren.

Wiederholungen mit exponentiellem Backoff und Jitter

Viele Fehler sind vorübergehend: ein verlorenes Paket, ein Verbindungsabbruch während eines Failovers, ein kurzes 503 von einer überlasteten Instanz. Eine Wiederholung (Retry) verwandelt sie in Erfolge, die der Benutzer nie bemerkt. Doch Wiederholungen sind auch Last, und schlecht entworfene Wiederholungen gehören zu den häufigsten Ursachen von Ausfällen. Vier Regeln machen sie sicher. Wiederholen Sie nur **vorübergehende** Fehler (Timeouts, Verbindungsfehler, 502, 503, 504 und 429 nach der angegebenen Wartezeit), niemals Fehler, die erneut auftreten werden, etwa 400 oder 404. Wiederholen Sie nur **idempotente** Operationen oder solche mit einem Idempotenzschlüssel, denn eine Anfrage mit Timeout kann trotzdem erfolgreich gewesen sein. Wiederholen Sie nur **wenige Male**, typischerweise zwei- oder dreimal. Und **warten** Sie zwischen den Versuchen, mit exponentiellem Backoff: einer Wartezeit, die sich mit jedem Versuch verdoppelt, bis zu einer Obergrenze.

Backoff allein reicht nicht. Scheitern tausend Clients im selben Moment, wie es bei einem kurzen Aussetzer einer Abhängigkeit geschieht, wiederholen sie alle nach 100 ms, dann alle nach 200 ms und treffen in synchronisierten Wellen ein, die die Abhängigkeit unten halten

können. Marc Brookers Artikel *Exponential Backoff And Jitter* (AWS Architecture Blog, 2015) hat das simuliert und gezeigt, dass Zufall, Jitter, sowohl die Gesamtarbeit als auch die Zeit bis zum Abschluss stark verringert. Sein empfohlenes „Full Jitter“ wartet eine zufällige Zeit zwischen null und der exponentiellen Schranke: `sleep = random(0, min(cap, base × 2^attempt))`. Jitter gehört zu allem, was viele Clients gleichzeitig tun: Wiederholungen, Neuverbindungen, Ablaufzeiten von Cache-Einträgen und geplante Jobs.

```
import random, time

class TransientError(Exception):
    """Timeouts, connection resets, 502, 503 and 504 responses."""

def call_with_retries(fn, attempts=3, base=0.1, cap=2.0):
    for attempt in range(attempts):
        try:
            return fn()
        except TransientError:
            if attempt == attempts - 1:
                raise                       # give up: let the caller decide
            ceiling = min(cap, base * 2 ** attempt)
            time.sleep(random.uniform(0, ceiling))   # full jitter
```

Wiederholung eines vorübergehenden Fehlers mit begrenztem exponentiellem Backoff und Full Jitter (Brooker, 2015). Nur idempotente Operationen oder solche mit einem Idempotenzschlüssel sollten wiederholt werden.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: warum Wiederholungen Zufall brauchen

Der schwierigste Gedanke dieses Abschnitts ist, dass exponentielles Backoff allein Wiederholungen nicht verteilt: Wenn viele Aufrufer im selben Moment scheitern, warten sie auch gleich lange und kommen gemeinsam zurück. Erst Zufall, Jitter, löst die Menge auf.

- Angenommen, 1000 Zahlungsaufrufe, jeder mit einem Idempotenzschlüssel, sodass Wiederholungen sicher sind, laufen gerade, als der Anbieter einen kurzen Aussetzer hat. Alle 1000 scheitern im selben Augenblick.
- Jeder Aufrufer folgt einfachem exponentiellem Backoff: 100 ms warten, erneut versuchen; scheitert das, 200 ms warten, dann 400 ms. Weil alle gemeinsam gescheitert sind, kommen alle 1000 Wiederholungen gemeinsam an, 100 ms später.
- Ein Anbieter, der sich gerade erholt, kann nicht 1000 Aufrufe in einem Augenblick aufnehmen, also scheitern viele erneut, und die nächste Welle kommt gemeinsam 200 ms später. Zwischen den Wellen ist der Anbieter untätig, während der Wellen überflutet, und die Wellen können ihn unten halten.
- Mit Full Jitter wartet jeder Aufrufer eine zufällige Zeit zwischen null und der Backoff-Wartezeit. Die ersten Wiederholungen verteilen sich über die 100 ms: etwa 10 pro Millisekunde statt 1000 auf einmal.
- Aufrufer, die erneut scheitern, warten eine zufällige Zeit bis zu 200 ms, dann bis zu 400 ms, sodass die Last verteilt ankommt, was ein sich erholender Anbieter weit leichter aufnehmen kann. Brookers Simulationen ergaben, dass das sowohl die Gesamtarbeit als auch die Zeit bis zum Abschluss verringert.
- Dieselbe Ballung tritt bei allem auf, was viele Maschinen gleichzeitig tun, etwa Neuverbinden, ablaufende Cache-Einträge oder geplante Jobs; auch dorthin gehört Jitter.
- **Die Lehre:** Backoff lässt Aufrufer jedes Mal länger warten; Jitter sorgt dafür, dass nicht alle gleich lange warten. Höflich zu warten reicht nicht, wenn alle im Gleichschritt warten.

Retry-Stürme und Retry-Budgets

Wiederholungen multiplizieren sich über Schichten. Angenommen, ein Tippen in der mobilen App läuft durch das API-Gateway, den Bestellservice und den Zahlungsclient, und jede der drei Schichten unternimmt bis zu drei Versuche. Ist der Zahlungsanbieter ausgefallen, erzeugt eine Benutzeraktion 3 × 3 × 3 = 27 Aufrufe an ihn, und ein Ausfall, der eine gleichmäßige Fehlerrate

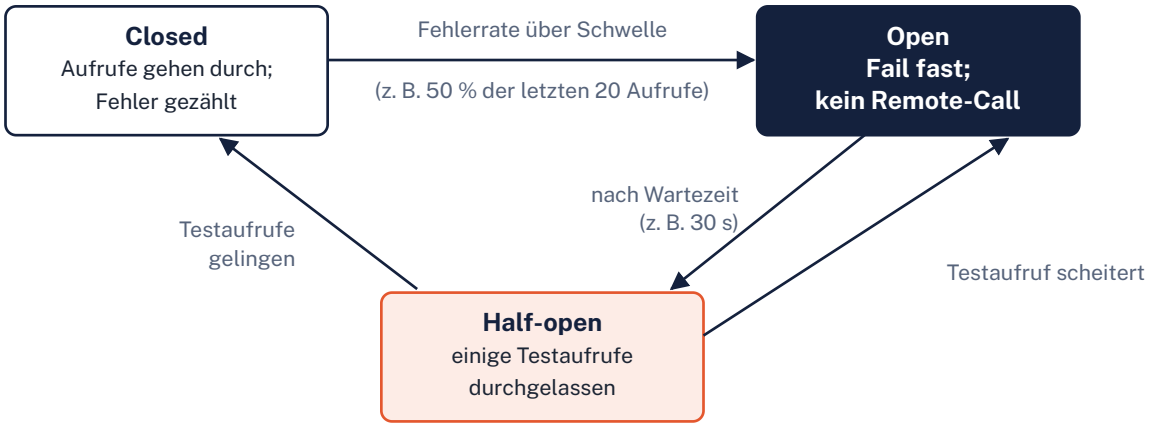
hätte bedeuten sollen, wird zu einem Retry-Sturm mit der 27-fachen normalen Last, der den Anbieter auch dann noch unten hält, wenn die ursprüngliche Ursache behoben ist. Benutzer, die erneut tippen, fügen eine vierte Schicht hinzu.

Drei Abwehrmaßnahmen wirken zusammen. Erstens: **Wiederholen Sie nur auf einer Schicht**, meist der, die der fehlschlagenden Abhängigkeit am nächsten ist, und lassen Sie die anderen Schichten den Fehler nach oben weiterreichen; Googles SRE-Buch empfiehlt, dass ein überlasteter Server das ausdrücklich meldet, damit Aufrufer weiter oben nicht wiederholen. Zweitens: Nutzen Sie ein Retry-Budget: Das SRE-Buch beschreibt, Wiederholungen nur zuzulassen, solange sie unter 10 % der Anfragen eines Clients liegen, was die zusätzliche Last durch Wiederholungen auf etwa das 1,1-Fache des Normalen begrenzt statt auf bis zum Dreifachen. AWS-SDKs implementieren ein ähnliches Kontingent an Retry-Tokens, und die Retry-Policy von gRPC enthält Throttling zum selben Zweck. Drittens: Kombinieren Sie Wiederholungen mit einem Circuit Breaker, damit eine Abhängigkeit, von der bekannt ist, dass sie ausgefallen ist, gar nicht erst erneut aufgerufen wird.

Circuit Breaker

Ein Circuit Breaker, von Nygard nach dem elektrischen Schutzschalter benannt, umhüllt Aufrufe an eine Abhängigkeit und beobachtet deren Ergebnisse. Solange die Abhängigkeit gesund ist, ist der Breaker **geschlossen** und Aufrufe gehen durch, während Fehler und Timeouts gezählt werden. Überschreiten die Fehler eine Schwelle, zum Beispiel die Hälfte der letzten 20 Aufrufe, **öffnet** der Breaker: Für einen festgelegten Zeitraum scheitert jeder Aufruf sofort, ohne das Netzwerk zu berühren, was die Threads des Aufrufers schützt und der Abhängigkeit Raum zur Erholung gibt. Läuft der Zeitraum ab, wird der Breaker **halboffen** und lässt einige Testaufrufe durch. Gelingen sie, schließt er; scheitert einer, öffnet er erneut für einen weiteren Zeitraum. Martin Fowlers Beschreibung von 2014 machte das Muster weithin bekannt, und Netflix’ Bibliothek Hystrix (heute im Wartungsmodus) machte es bei Microservices populär; Resilience4j für Java und Polly für .NET sind heute die gängigen Bibliotheken, und Service Meshes bieten verwandte Schutzmechanismen wie Nebenläufigkeitslimits und Outlier Detection (in Envoy heißen sie Circuit Breaking, obwohl sie anders funktionieren).

Der Zustandsautomat des Circuit Breaker



Closed zählt Fehler, Open schlägt schnell fehl, und Half-open prüft die Erholung mit einigen Testaufrufen.

Die Einstellungen verdienen Überlegung. Was zählt als Fehler? Timeouts und 5xx-Antworten sollten zählen; 4xx-Antworten meist nicht, weil sie die Anfrage des Aufrufers widerspiegeln, nicht die Gesundheit der Abhängigkeit. Auch langsame Erfolge können zählen, sodass eine Abhängigkeit, die in 5 Sekunden antwortet, als fehlerhaft behandelt wird. Das Fenster muss groß genug sein, um nicht bei ein paar unglücklichen Aufrufen auszulösen (Bibliotheken verlangen eine Mindestzahl an Aufrufen), und der offene Zeitraum lang genug für die Erholung, aber kurz genug, um einen Ausfall nicht zu verlängern. Breaker sollten pro Abhängigkeit, manchmal pro Endpoint, eingerichtet werden, damit ein fehlerhafter Reporting-Endpoint nicht das Bestellen abschneidet. Wiederholungen sollten einen offenen Breaker als endgültige Antwort behandeln, nicht als etwas, das man wiederholt. Und jeder Zustandswechsel sollte geloggt und grafisch dargestellt werden, denn ein offener Breaker ist ein Symptom, das jemand sehen muss.

```
import time

class CircuitOpen(Exception):
    pass

class CircuitBreaker:
    def __init__(self, failure_threshold=5, reset_after=30.0):
        self.failure_threshold = failure_threshold  # failures in a row that trip it
        self.reset_after = reset_after              # seconds to stay open
        self.failures, self.state, self.opened_at = 0, "closed", 0.0

    def call(self, fn, *args, **kwargs):
        if self.state == "open":
            if time.monotonic() - self.opened_at < self.reset_after:
                raise CircuitOpen("failing fast")   # no remote call at all
            self.state = "half-open"                # let a trial call through
        try:
            result = fn(*args, **kwargs)
        except Exception:
            self.failures += 1
            if self.state == "half-open" or self.failures >= self.failure_threshold:
                self.state, self.opened_at = "open", time.monotonic()
            raise
        self.failures, self.state = 0, "closed"    # success closes the circuit
        return result

payments = CircuitBreaker()
# payments.call(provider.authorise, order_id, amount, timeout=2.0)
```

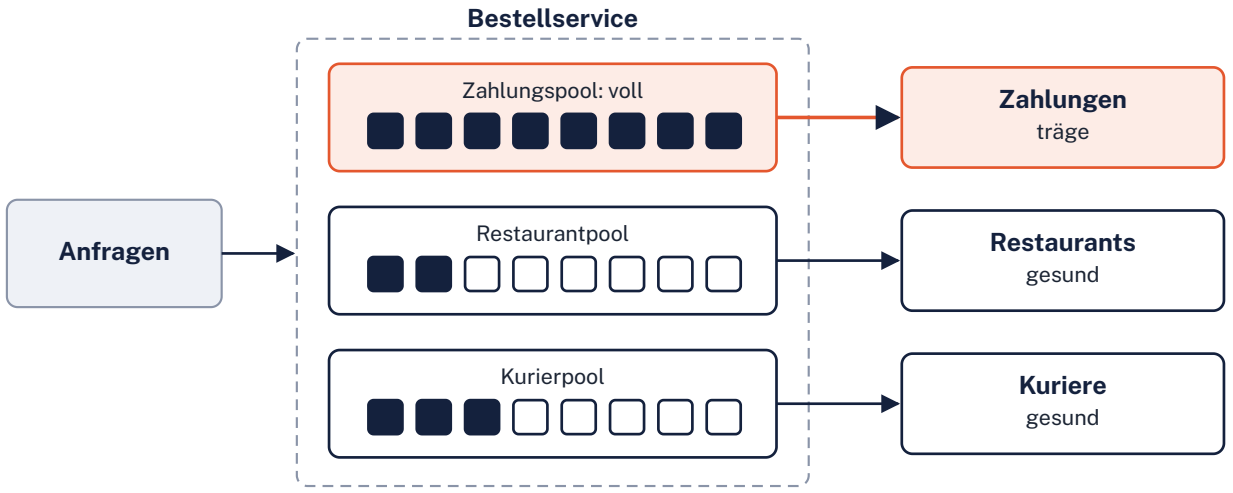
Ein minimaler Circuit Breaker. Produktionsbibliotheken wie Resilience4j (Java) und Polly (.NET) ergänzen Fehlerratenfenster, Erkennung langsamer Aufrufe, Limits für gleichzeitige Testaufrufe, Thread-Sicherheit und Metriken.

Bulkheads

Der Rumpf eines Schiffs ist durch Schotten (Bulkheads) in wasserdichte Abteilungen geteilt, sodass ein Leck eine Abteilung flutet, statt das Schiff zu versenken. In Software partitioniert ein Bulkhead Ressourcen, sodass ein fehlerhafter oder überlasteter Teil nicht alles verbrauchen kann. Die häufigste Form gibt jeder Abhängigkeit einen eigenen begrenzten Thread-Pool, Connection-Pool oder Semaphor: Zahlungen dürfen höchstens 40 gleichzeitige Aufrufe pro Instanz nutzen, Restaurantabfragen 40, Kurierzuweisung 40. Werden Zahlungen langsam, füllt sich ihr Pool, und weitere Zahlungsaufrufe werden sofort abgelehnt, aber Restaurant- und Kurieraufrufe laufen weiter. Ein Timeout begrenzt, wie lange jeder Aufruf Ressourcen halten kann; ein Bulkhead begrenzt, wie viele gleichzeitig gehalten werden können, und schützt daher selbst dann, wenn Timeouts zu lang gesetzt sind.

Bulkheads wirken auch im größeren Maßstab. Getrennte Deployments desselben Service für verschiedene Aufrufer (ein Pool von Bestellservice-Instanzen für die Kunden-App, ein anderer für die Restaurantpartner-API) verhindern, dass eine fehlerhafte Partnerintegration die Kunden aushungert. Zellbasierte Architekturen, die Z-Achse des Scale Cube, machen jede Zelle zu einem Bulkhead für einen Teil der Kunden, und Shuffle Sharding, von Fachleuten bei AWS beschrieben, weist jedem Kunden eine zufällige kleine Kombination von Instanzen zu, sodass der giftige Verkehr eines Kunden nur sehr wenige andere trifft. Der Preis jedes Bulkheads ist Auslastung: Für eine Abteilung reservierte Kapazität kann nicht an eine andere verliehen werden, daher müssen Pools anhand des gemessenen Bedarfs dimensioniert werden.

Bulkheads innerhalb eines Service



Mit einem eigenen Pool pro Abhängigkeit füllt ein langsamer Zahlungsanbieter nur den Zahlungspool; Restaurant- und Kurieraufrufe laufen normal weiter.

Fallbacks und Graceful Degradation

Scheitert ein Aufruf oder ist ein Breaker offen, muss der Aufrufer trotzdem antworten. Ein Fallback legt fest, was er zurückgibt. Die Optionen, von der besten zur schlechtesten für den Benutzer, sind: ein **zwischengespeicherter oder veralteter Wert** (die zuletzt bekannte Speisekarte, mit Preisen, die im Bezahlvorgang bestätigt werden); ein **sinnvoller Standardwert** (eine geschätzte Lieferzeit von 30 bis 45 Minuten statt einer Live-Schätzung); eine **eingeschränkte Funktion** (beliebte Gerichte statt personalisierter Empfehlungen); **die Arbeit aufschieben** (eine Bewertung annehmen und veröffentlichen, wenn sich der Bewertungsservice erholt hat); und ein **klarer, schneller Fehler**, der dem Benutzer sagt, was zu tun ist. Diese pro Feature gemeinsam mit dem Product Owner zu entwerfen, ist das, was Graceful Degradation in der Praxis bedeutet: Die Kernabläufe (stöbern, bestellen, bezahlen,

verfolgen) funktionieren weiter, während nicht wesentliche Features zurücktreten. Dasselbe Denken leitet Load Shedding (Abschnitt 9.3): Wissen Sie im Voraus, welche Features Sie zuerst abschalten.

Sicherheitsentscheidungen sind die Ausnahme: Ist der Autorisierungsservice nicht erreichbar, ist der sichere Fallback, zu verweigern (Fail-closed), nicht zu erlauben. Fachleute sind sich bei Fallbacks auch allgemein uneins. Nygard und die Fachleute bei Netflix befürworten sie breit. Fachleute bei Amazon haben im Builders'-Library-Artikel *Avoiding fallback in distributed systems* argumentiert, dass Fallback-Pfade selten durchlaufen werden, schwer zu testen sind und selbst Ausfälle auslösen können, indem sie eine große Flotte im ungünstigsten Moment in einen ungetesteten Modus schalten; sie ziehen es vor, den primären Pfad zuverlässiger zu machen und schnell fehlzuschlagen. Beide Positionen sind sich über die praktische Regel einig: Ein Fallback muss einfacher sein als das, was er ersetzt, darf nicht von der ausgefallenen Komponente abhängen und muss regelmäßig durchlaufen werden, zum Beispiel an Game Days (Abschnitt 9.8), sonst sollte es ihn nicht geben.

Idempotenz: was Wiederholungen und Hedging sicher macht

Idempotenz ist die Eigenschaft, dass etwas zweimal zu tun dieselbe Wirkung hat wie es einmal zu tun. Jedes Muster in diesem Abschnitt, das Arbeit wiederholt, hängt davon ab: Wiederholungen nach einem Timeout, Hedged Requests, erneute Zustellung von Nachrichten nach dem Absturz eines Consumers und Replays aus einer Dead Letter Queue. Abschnitt 6.5 hat gezeigt, wie ein Idempotenzschlüssel `POST /orders` sicher wiederholbar macht, und Abschnitt 6.4, wie idempotente Consumer Nachrichten deduplizieren. Die architektonische Regel ist einfach: Prüfen Sie vor dem Hinzufügen einer Wiederholung, ob die Operation idempotent ist, und wenn nicht, fügen Sie zuerst einen Schlüssel oder eine Duplikatprüfung hinzu. Eine Retry-Policy für einen nicht idempotenten Zahlungsaufruf ist kein Resilienzmuster, sondern ein Generator für Doppelabbuchungen.

Hedged Requests

Wiederholungen behandeln Fehler; Hedged Requests behandeln Langsamkeit. Abschnitt 2.4 hat sie nach Dean und Barrosos *The Tail at Scale* (2013) eingeführt: Senden Sie eine Anfrage an ein Replikat, und wenn es nicht innerhalb, sagen wir, der Latenz des 95. Perzentils geantwortet hat, senden Sie eine Kopie an ein zweites Replikat, verwenden Sie die Antwort, die zuerst kommt, und brechen Sie die andere ab. Weil nur die langsamsten paar Prozent der Anfragen dupliziert werden, ist die zusätzliche Last gering, während der Tail drastisch schrumpft, wie der in Abschnitt 2.4 zitierte Benchmark gezeigt hat.

Hedging hat Voraussetzungen. Die Operation muss idempotent sein, normalerweise ein Lesezugriff. Es muss unabhängige Replikate geben, damit die zweite Kopie wahrscheinlich nicht aus demselben Grund langsam ist. Die zusätzlichen Anfragen müssen budgetiert und unter Überlast abgeschaltet werden, denn wenn jedes Replikat langsam ist, verdoppelt Hedging die Last auf einem System, das ohnehin kämpft, genau wie eine unbudgetierte Wiederholung. Sorgfältig eingesetzt auf leselastigen Fan-out-Pfaden wie Suche und Speisekartenabfragen, ist es eines der wirksamsten verfügbaren Werkzeuge gegen Tail-Latenz.

Die Muster auf einen Blick

Muster	Schützt vor	Hauptkosten oder -risiko
Timeout	Unbegrenztes Warten; Ressourcen, die von toten Aufrufen gehalten werden	Zu kurz lässt gesunde langsame Aufrufe scheitern; zu lang schützt nichts
Deadline-Weitergabe (Deadline Propagation)	Arbeit, die weiterläuft, nachdem der Aufrufer aufgegeben hat	Braucht Unterstützung in jedem Service und jeder Bibliothek auf dem Pfad
Wiederholung mit Backoff und Jitter	Vorübergehende Fehler	Zusätzliche Last; Duplikate, sofern nicht idempotent
Retry-Budget	Retry-Stürme, die die Last vervielfachen	Manche vorübergehenden Fehler werden in schlechten Phasen nicht wiederholt
Circuit Breaker	Wiederholte Aufrufe einer ausgefallenen Abhängigkeit	Feinabstimmung; kann eine nur teilweise fehlerhafte Abhängigkeit abschneiden
Bulkhead	Eine Abhängigkeit oder ein Aufrufer verbraucht alle Ressourcen	Reservierte Kapazität kann nicht geteilt werden
Fallback und Degradation	Totalausfall eines Features	Ungetestete Codepfade; veraltete oder ungefähre Ergebnisse
Hedged Request	Tail-Latenz durch langsame Replikate	Zusätzliche Last; nur Lesezugriffe

Checkliste: jeder Remote-Aufruf

Prüfen Sie jeden Integrationspunkt, einschließlich Datenbanken, Caches, Broker und Drittanbieter-APIs, anhand dieser Liste.

- [ ] **Connect- und Request-Timeouts sind explizit gesetzt**
  Gewählt anhand der gemessenen hohen Perzentile der Abhängigkeit und der eigenen Deadline des Aufrufers, nie auf einem Bibliotheksstandard belassen.
- [ ] **Die verbleibende Deadline wird weitergegeben**
  Nachgelagerte Aufrufe erhalten die Restzeit, und die Arbeit stoppt, wenn sie abläuft.
- [ ] **Wiederholungen sind begrenzt und sicher**
  Nur vorübergehende Fehler, nur idempotente Operationen oder solche mit Idempotenzschlüssel, auf einer Schicht, mit begrenztem exponentiellem Backoff und Jitter.
- [ ] **Ein Retry-Budget begrenzt die zusätzliche Last**
  Wiederholungen bleiben ein kleiner Anteil des Verkehrs (zum Beispiel 10 %), sodass ein Ausfall die Last nicht vervielfachen kann.
- [ ] **Ein Circuit Breaker schützt die Abhängigkeit**
  Schwellen, Wartezeit und was als Fehler zählt, sind abgestimmt, und sein Zustand wird als Metrik exportiert.
- [ ] **Die Nebenläufigkeit zur Abhängigkeit ist begrenzt**
  Ein Bulkhead (eigener Pool oder Semaphor) verhindert, dass diese Abhängigkeit jeden Thread oder jede Verbindung verbraucht.
- [ ] **Das Verhalten im Fehlerfall ist entschieden**
  Ein Fallback, eine eingeschränkte Antwort oder ein klarer Fehler, mit dem Product Owner abgestimmt, und Fail-closed bei Sicherheitsentscheidungen.
- [ ] **Timeouts, Wiederholungen, ausgelöste Breaker und Fallbacks sind beobachtbar**
  Gezählt, grafisch dargestellt und mit Alarmen versehen, damit stille Degradation bemerkt wird (Abschnitt 9.6).
- [ ] **Der Fehlerpfad ist getestet**
  Fault Injection oder ein Game Day zeigt, dass Timeouts, Breaker und Fallback tatsächlich funktionieren (Abschnitt 9.8).

ACHTUNG

Häufige Fehler

- Timeouts auf Bibliotheksstandards belassen, die oft unendlich oder minutenlang sind.
- Nachgelagerte Timeouts, die länger sind als vorgelagerte, sodass Arbeit weiterläuft, nachdem der Aufrufer weg ist.
- Auf jeder Schicht wiederholen, nicht idempotente Operationen wiederholen oder ohne Jitter wiederholen.
- 4xx-Antworten in einem Circuit Breaker als Fehler zählen, sodass ein fehlerhafter Client ihn für alle auslöst.
- Ein gemeinsamer Thread- oder Connection-Pool für alle Abhängigkeiten, sodass die langsamste über alle entscheidet.
- Fallbacks, die nie durchlaufen werden und beim ersten Bedarf scheitern oder etwas anderes überlasten.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Ein Service erhält 50 Anfragen pro Sekunde, die jeweils eine Abhängigkeit mit einem Timeout von 4 Sekunden aufrufen. Wie viele Threads können diese Aufrufe belegen, wenn die Abhängigkeit hängt?

Nach Littles Gesetz etwa 50 × 4 = 200 Threads; deshalb müssen der Timeout und ein Bulkhead-Limit zur Poolgröße passen.

▼ Warum ist Full Jitter besser als einfaches exponentielles Backoff, wenn viele Clients gleichzeitig scheitern?

Einfaches Backoff lässt alle Clients in denselben Momenten wiederholen, in synchronisierten Wellen; zufällige Wartezeiten verteilen die Wiederholungen, sodass die Abhängigkeit eine gleichmäßige, geringere Last sieht.

▼ In welchem Zustand macht ein Circuit Breaker keine Remote-Aufrufe, und was beendet diesen Zustand?

Im offenen Zustand; wenn die Wartezeit abläuft, wird er halboffen und lässt Testaufrufe durch.

▼ **Warum sollte eine Autorisierungsprüfung nicht darauf zurückfallen, den Zugriff zu erlauben, wenn der Autorisierungsservice ausgefallen ist?**

Sicherheitsentscheidungen müssen geschlossen fehlschlagen (Fail-closed): Zugriff während eines Ausfalls zu erlauben, würde jeden hereinlassen, sobald der Service ausfällt oder angegriffen wird.

QUELLEN

- Marc Brooker: Exponential Backoff And Jitter (AWS Architecture Blog, 2015): https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- Martin Fowler: CircuitBreaker: https://martinfowler.com/bliki/CircuitBreaker.html
- Google SRE book: Addressing Cascading Failures: https://sre.google/sre-book/addressing-cascading-failures/
- Microsoft Azure Architecture Center: Bulkhead pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead
- Dean and Barroso: The Tail at Scale (Google Research): https://research.google/pubs/pub40801/

ABSCHNITT 9.5 VON 9.8 · MITTELSTUFE · 24 MIN. LESEZEIT

SLIs, SLOs, SLAs und Error Budgets

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.4 Performance und Skalierbarkeit
- 2.5 Verfügbarkeit und Zuverlässigkeit

Begriffe von früher, die er verwendet: Verfügbarkeit, Zuverlässigkeit, Perzentil (p95, p99), Latenz

DIE IDEE IN KÜRZE

Stellen Sie sich ein Busunternehmen vor, das „fast alle Busse pünktlich“ verspricht. Damit das nützlich ist, muss es sagen, was als pünktlich gilt (innerhalb von zwei Minuten), wie es das misst (an jeder Haltestelle) und was „fast alle“ bedeutet (99 von 100 über einen Monat). Verfehlt es das, hört das Fahrplanteam auf, neue Linien auszuprobieren, und kümmert sich zuerst um die Pünktlichkeit. Weil Perfektion unmöglich und sehr teuer ist, ist der eine verspätete Bus von hundert ein Spielraum, den es für Baustellen und neue Linien ausgeben kann. Software-Zuverlässigkeit funktioniert genau so.

WAS SIE WISSEN MÜSSEN

- Ein SLI ist eine nutzerorientierte Messung, am besten ausgedrückt als gute Ereignisse geteilt durch gültige Ereignisse; ein SLO ist ein Ziel dafür über ein Zeitfenster; ein SLA ist ein Vertrag mit Konsequenzen, lockerer angesetzt als das SLO.
- Nützliche SLIs decken für anfragegetriebene Services Verfügbarkeit, Latenz und Qualität ab, für Pipelines Aktualität, Korrektheit und Abdeckung und für Speicher die Dauerhaftigkeit, gemessen so nah am Nutzer, wie es praktikabel ist.
- Das Error Budget ist eins minus das SLO: 99,9 % über 30 Tage erlauben 43,2 Minuten Totalausfall oder 0,1 % der Anfragen, und eine schriftliche Error-Budget-Policy legt fest, was geschieht, wenn es aufgebraucht ist.
- Die Burn Rate ist die beobachtete Fehlerquote geteilt durch die erlaubte Quote; das verbrauchte Budget ist Burn Rate mal Alerting-Fenster geteilt durch SLO-Fenster, sodass eine Burn Rate von 14,4 über eine Stunde 2 % eines 30-Tage-Budgets verbraucht.
- Die Multi-Window-, Multi-Burn-Rate-Alerts des SRE Workbook alarmieren bei 2 % des Budgets in einer Stunde oder 5 % in sechs Stunden und erzeugen ein Ticket bei 10 % in drei Tagen, jeweils bestätigt durch ein kurzes Fenster von einem Zwölftel der Länge.

- Typische Fehler sind Ziele von 100 %, zu viele SLOs, SLIs, die Nutzer nicht erleben, Messen an der falschen Stelle und SLOs ohne Policy oder Alerts dahinter.

Drei Begriffe, eine Idee

Kapitel 2 hat Verfügbarkeitsziele und die Neunen eingeführt (Abschnitt 2.5) und gezeigt, dass 99,9 % über 30 Tage 43,2 Minuten übrig lassen. Dieser Abschnitt macht aus dieser Rechnung eine Betriebsdisziplin. Das Vokabular stammt aus Googles Buch *Site Reliability Engineering* (Beyer, Jones, Petoff und Murphy, O'Reilly, 2016) und seinem Begleitband *The Site Reliability Workbook* (2018) und wird heute weit über Google hinaus verwendet.

Ein Service Level Indicator (SLI) ist ein sorgfältig definiertes, quantitatives Maß für einen Aspekt des Service, den Nutzer erhalten. Die nützlichste Form ist ein Verhältnis: gute Ereignisse geteilt durch gültige Ereignisse. „Der Anteil der Checkout-Anfragen, die innerhalb von 800 ms eine erfolgreiche Antwort liefern, gemessen am Load Balancer" ist ein SLI. Es reicht von 0 % (nichts funktioniert) bis 100 % (nichts ist kaputt), und die Verhältnisform bedeutet, dass sich jedes SLI auf dieselbe Weise lesen, vergleichen und mit Alerts versehen lässt.

Ein Service Level Objective (SLO) ist ein Ziel für ein SLI über ein festgelegtes Zeitfenster: „99,9 % der Checkout-Anfragen sind gut, gemessen über gleitende 28 Tage." Es ist ein internes Versprechen, auf das sich das Team, der Product Owner und die Menschen einigen, die von dem Service abhängen. Ein Service Level Agreement (SLA) ist ein Vertrag mit Kunden, der Konsequenzen benennt, meist Service-Gutschriften, wenn ein Niveau verfehlt wird. Viele Services haben SLOs, aber kein SLA; interne Plattformen haben fast nie SLAs.

Die Idee, die alle drei verbindet, ist das Error Budget. Liegt das Ziel bei 99,9 %, dürfen 0,1 % der Ereignisse schlecht sein, ohne dass jemand versagt hat. Diese 0,1 % sind keine Verschwendung; sie sind der Spielraum, den das Team hat, um Risiken einzugehen: Features ausliefern, Migrationen durchführen, mit der Konfiguration experimentieren. Das Budget macht aus einer Diskussion über Gefühle („Ist der Service zuverlässig genug?") eine Diskussion über Zahlen („Haben wir unser Budget verbraucht?") und gibt Produkt und Engineering eine gemeinsame Regel dafür, wann man schnell vorangeht und wann man bremst.

SLI, SLO und SLA auf einer Skala

Das SLI ist die Messung, das SLO das interne Ziel und das SLA eine lockerere vertragliche Untergrenze; die Lücke zwischen SLO und SLA ist die Warnzone.

Gute SLIs wählen

Ein SLI ist nur so nützlich, wie es nah an dem liegt, was Nutzer erleben. Das Workbook bietet eine kurze Auswahl, die die meisten Services abdeckt. Anfragegetriebene Services (APIs, Webseiten) brauchen **Verfügbarkeit** (der Anteil gültiger Anfragen, die erfolgreich bedient werden), **Latenz** (der Anteil, der schneller als ein Schwellenwert bedient wird) und manchmal **Qualität** (der Anteil, der ohne Einschränkung bedient wird, etwa eine Speisekarte, die ohne personalisierte Empfehlungen angezeigt wird, weil diese Abhängigkeit ausgefallen war). Datenpipelines brauchen **Aktualität** (der Anteil der Daten, die vor kürzerer Zeit als ein Schwellenwert aktualisiert wurden), **Korrektheit** (der Anteil der Ausgabe, der stimmt) und **Abdeckung** (der Anteil der verarbeiteten Eingabe). Speichersysteme ergänzen **Dauerhaftigkeit** (der Anteil geschriebener Daten, der noch gelesen werden kann).

Latenz-SLIs werden als Anteile geschrieben, nicht als Mittelwerte: „99 % der Speisekarten-Anfragen schneller als 300 ms“. Das knüpft direkt an die Perzentile aus Abschnitt 2.4 an: Zu sagen, 99 % der Anfragen müssen unter 300 ms liegen, ist dasselbe wie zu sagen, p99 muss unter 300 ms liegen, aber die Anteilsform lässt sich sauber über Zeit und Server addieren (man zählt die guten und die gültigen), Perzentile dagegen nicht. Üblich sind zwei Latenzschwellen, zum Beispiel 90 % unter 200 ms und 99 % unter 800 ms, um sowohl das typische als auch das Tail-Erlebnis zu erfassen.

Wo Sie messen, ist ebenso wichtig wie was Sie messen. Das Workbook trennt die *SLI-Spezifikation* (das für Nutzer relevante Ergebnis, etwa „erfolgreich aufgegebene Bestellungen“) von der *SLI-Implementierung* (wie Sie es zählen: Serverlogs, Load-Balancer-Metriken, synthetische Probes oder Instrumentierung in der Mobil-App). Wer im Anwendungsserver misst, verpasst Anfragen, die ihn nie erreicht haben, weil DNS, das Content

Delivery Network oder der Load Balancer ausgefallen ist. Wer im Client misst, sieht alles, erfasst aber auch schlechte Mobilverbindungen der Nutzer und kommt verspätet an. Die meisten Teams beginnen mit Load-Balancer-Metriken, die günstig und nah am Rand sind, und ergänzen synthetische Probes für Pfade mit wenig Traffic.

Legen Sie genau fest, was als *gültig* und was als *gut* zählt, denn die Grenzfälle entscheiden, ob die Zahl etwas bedeutet. Ist ein 404 für eine vertippte URL schlecht? Meist nicht: Es ist eine korrekte Antwort. Ist eine 429-Antwort wegen Rate Limiting schlecht? Das hängt davon ab, ob der Client innerhalb seines vereinbarten Kontingents lag. Sind Health Checks und Lasttests gültige Ereignisse? Nein; schließen Sie sie aus, sonst verwässern sie das Verhältnis. Schreiben Sie diese Regeln neben das SLO.

SLIs für die durchgehenden Beispiele

Nutzerpfad	SLI-Typ	SLI-Spezifikation (gut / gültig)	Beispiel-SLO
Essenslieferung: Speisekarten durchsuchen	Latenz	Speisekarten-Anfragen, beantwortet in unter 300 ms / alle gültigen Speisekarten-Anfragen	99 % über 28 Tage
Essenslieferung: Bestellung aufgeben	Verfügbarkeit	Bestellanfragen, beantwortet mit Erfolg oder einer korrekten fachlichen Ablehnung / alle gültigen Bestellanfragen	99,9 % über 28 Tage
Essenslieferung: Kurierstandort	Aktualität	Kartenansichten, deren Kurierposition jünger als 30 s ist / alle Kartenansichten	99 % über 28 Tage
Essenslieferung: Auszahlungen an Restaurants	Korrektheit	Auszahlungsläufe, ohne Abweichung abgeglichen / alle Auszahlungsläufe	99,99 % über ein Quartal
Ticketing: Bezahlvorgang beim Vorverkaufsstart	Latenz und Verfügbarkeit	Sitzplatzreservierungen, bestätigt innerhalb von 2 s / alle Reservierungsversuche zugelassener Fans	99,5 % bei jedem Vorverkaufsstart
Ticketing: Zustellung der E-Tickets	Aktualität	Tickets in der App des Fans innerhalb von 5 Minuten nach der Zahlung / alle bezahlten Bestellungen	99,9 % über 28 Tage

Das Ziel festlegen

Ein Ziel ist eine geschäftliche Entscheidung, keine technische. Die Kosten jeder weiteren Neun steigen steil, während Nutzer ab einem gewissen Punkt keine Verbesserungen mehr bemerken, weil ihr eigenes Mobilfunknetz, ihr WLAN und ihre Geräte häufiger ausfallen. Googles SRE-Buch sagt es unverblümt: 100 % anzustreben ist für fast alles falsch, weil es kein Budget für

Änderungen lässt, ein Vermögen kostet und ohnehin nicht erreichbar ist. Ein Ziel von 100 % macht außerdem jeden einzelnen Ausfall zu einem Policy-Verstoß, was Menschen lehrt, das SLO zu ignorieren.

Gute Ausgangspunkte sind die bisherige Leistung (was hat der Service im letzten Quartal erreicht, und waren die Nutzer zufrieden?), die Ziele der Services, von denen Sie abhängen (Sie können keine 99,99 % auf einer Datenbank mit 99,9 % versprechen, wie Abschnitt 6.1 für Aufrufketten gezeigt hat), und die geschäftlichen Kosten mangelnder Zuverlässigkeit. Für die Essenslieferplattform kosten fehlgeschlagene Bestellungen zur Mittagszeit Umsatz und Kundenvertrauen, daher sind 99,9 % bei der Bestellaufgabe vertretbar; Aktualisierungen der Kurierposition sind weniger kritisch, und 99 % genügen. Beginnen Sie etwas unterhalb der aktuellen Leistung, damit das Team nicht sofort im Verstoß ist, und verschärfen Sie später anhand von Belegen.

Wählen Sie das Zeitfenster bewusst. Ein gleitendes Zeitfenster wie die letzten 28 Tage entspricht der Art, wie Nutzer sich an Ausfälle erinnern (sie vergessen sie nicht am Monatsersten), und 28 Tage enthalten genau vier von jedem Wochentag, was wöchentliche Muster fair hält. Kalendermonate eignen sich für Berichte und Verträge. Ereignisbasierte Services wie ein Ticket-Vorverkaufsstart brauchen unter Umständen ein SLO pro Ereignis, weil eine Stunde Ausfall während des Vorverkaufsstarts weit mehr zählt als eine ruhige Woche.

Error Budgets in Zahlen

Das Error Budget lässt sich in Zeit oder in Ereignissen zählen. Zeitbasierte Budgets passen zu Services, die entweder laufen oder ausgefallen sind; anfragebasierte Budgets sind fairer für Services mit ungleichmäßigem Traffic, weil ein Ausfall um 03:00 weniger Nutzer trifft als einer zur Mittagszeit.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: das Budget für die Bestellaufgabe

Die Bestell-API der Essenslieferplattform hat ein SLO von 99,9 % guten Anfragen über 30 Tage. Sie verarbeitet in diesem Zeitraum etwa 3 Millionen gültige Bestellanfragen, stark konzentriert auf Mittag- und Abendessen.

Budget als Anteil	1 - 0,999 = 0,001 bzw. 0,1 %
Budget in Zeit	0,001 × 30 Tage × 1440 Minuten = 43,2 Minuten Totalausfall
Budget in Anfragen	0,001 × 3.000.000 = 3000 schlechte Anfragen
Incident 1: fehlerhaftes Deployment um 15:00	12 Minuten × 800 Anfragen pro Minute = 9600 Anfragen; 40 % fehlgeschlagen = 3840 schlechte Anfragen
Ergebnis	3840 ist mehr als das gesamte Budget von 3000 Anfragen: Ein Incident erschöpft den Monat
Derselbe Incident um 03:00	12 × 50 Anfragen pro Minute × 0,4 = 240 schlechte Anfragen, 8 % des Budgets

Anfragebasierte Budgets gewichten Ausfälle danach, wie viele Nutzer sie treffen; deshalb können dieselben 12 Minuten 8 % oder mehr als 100 % des Budgets kosten.

Die zeitbasierte Sicht (43,2 Minuten) hätte beide Incidents gleich bewertet, mit 12 × 0,4 = 4,8 Minuten äquivalentem Ausfall, etwa 11 % des Budgets.

Error-Budget-Policies

Ein Budget bedeutet nichts, wenn nichts passiert, sobald es aufgebraucht ist. Eine Error-Budget-Policy ist ein kurzes Dokument, das Engineering, Produkt und Leitung im Voraus vereinbaren und das festlegt, was das Team tun wird. Die Beispiel-Policy im Workbook misst das Budget über die zurückliegenden vier Wochen; überschreitet der Service sein Budget, stoppen alle Änderungen und Releases außer dringenden Fixes (P0-Probleme) und Sicherheitsfixes, bis der Service wieder innerhalb seines SLO liegt. Sie verlangt außerdem ein Postmortem mit mindestens einer Maßnahme höchster Priorität, wann immer ein einzelner Incident mehr als 20 % des Vier-Wochen-Budgets verbraucht, und eskaliert Meinungsverschiedenheiten an den Chief Technology Officer.

Der Wert einer Policy liegt darin, dass sie vereinbart ist, bevor sie greift. Mitten im Release-Endspurt will niemand einfrieren; mit einer unterzeichneten Policy ist der Freeze eine vorab vereinbarte Regel statt eines Streits zwischen einer SRE und einem Produktmanager. Gute

Policies legen auch fest, was *nicht* zulasten des Teams geht: ein unternehmensweiter Netzwerkausfall, eine Abhängigkeit, die selbst eingefroren ist, oder Traffic außerhalb des Geltungsbereichs des SLO, etwa ein Lasttest.

Fachleute sind uneins darüber, wie mechanisch die Policy sein sollte. Manche Teams wenden sie streng an, weil Ausnahmen das Vertrauen in das ganze System untergraben. Andere sehen das Budget als Information für ein Gespräch und argumentieren, dass ein harter Freeze genau die Fixes verzögern kann, die die Zuverlässigkeit wiederherstellen würden. Ein Mittelweg ist verbreitet: riskante Feature-Releases automatisch einfrieren, Zuverlässigkeitsfixes weiter ausliefern und einer benannten Führungskraft erlauben, dokumentierte Ausnahmen zu gewähren. Auch der umgekehrte Fall zählt: Ein Team, das sein Budget nie ausgibt, ist wahrscheinlich zu vorsichtig und könnte schneller ausliefern, mehr Experimente durchführen oder seine Kosten senken.

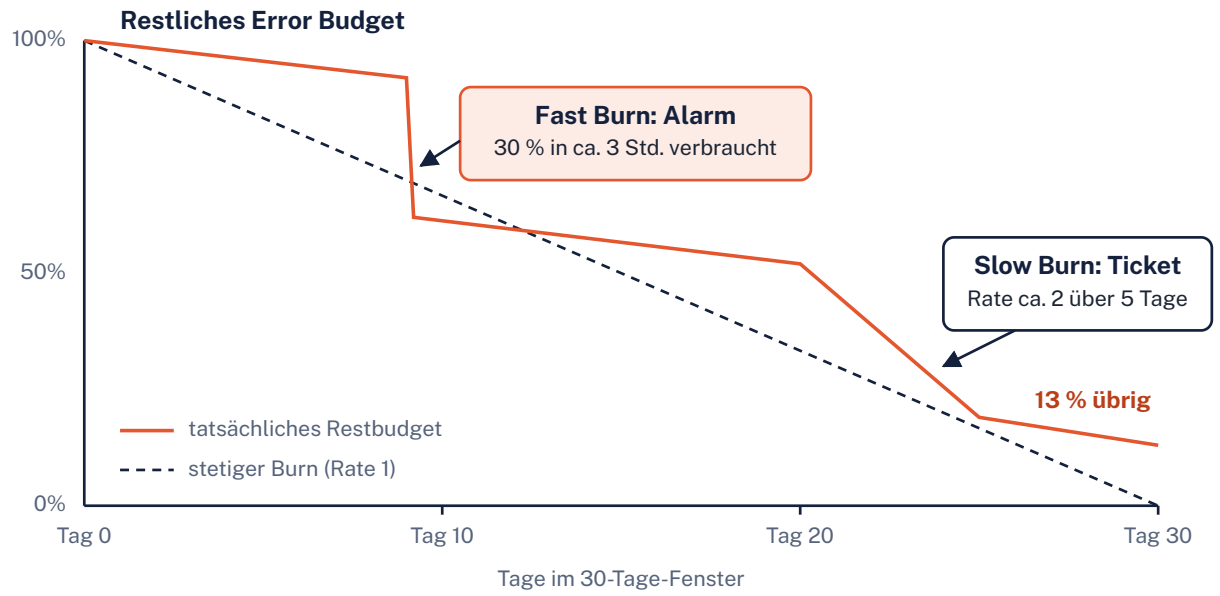
Burn Rate und Alerting auf SLOs

Alerting auf rohe Fehlerraten („alarmieren, wenn mehr als 1 % der Anfragen 5 Minuten lang fehlschlagen“) ist laut: Kurze Ausreißer wecken nachts Menschen für Probleme, die das SLO nie gefährden würden, während eine stetige Fehlerrate von 0,5 % still den Monat aufbrauchen kann, ohne je die Schwelle zu überschreiten. Die Antwort des Workbook ist, auf die Burn Rate zu alarmieren: wie schnell das Budget verbraucht wird, verglichen mit der Rate, die es genau am Ende des Zeitfensters aufbrauchen würde.

Burn Rate = beobachtete Fehlerquote / erlaubte Fehlerquote. Bei einem SLO von 99,9 % liegt die erlaubte Quote bei 0,1 %, also entspricht eine Fehlerquote von 0,1 % einer Burn Rate von 1 (das Budget reicht genau für das Zeitfenster), 1 % einer Burn Rate von 10 und 100 % (ein Totalausfall) einer Burn Rate von 1000. Der Anteil des Budgets im Zeitfenster, der während eines Alerting-Fensters verbraucht wird, ist Burn Rate x Alerting-Fenster / SLO-Fenster. Bei einem Fenster von 30 Tagen (720 Stunden) verbraucht eine Burn Rate von 14,4 über eine Stunde 14,4 × 1 / 720 = 0,02, also 2 % des Monatsbudgets; bei dieser Rate wäre das ganze Budget nach 720 / 14,4 = 50 Stunden aufgebraucht.

Das Workbook empfiehlt einen Multi-Window-, Multi-Burn-Rate-Alert als sinnvollen Standard für ein SLO von 99,9 %: alarmieren, wenn 2 % des Budgets in einer Stunde (Burn Rate 14,4) oder 5 % in sechs Stunden (Burn Rate 6) verbraucht werden, und ein Ticket eröffnen, wenn 10 % in drei Tagen (Burn Rate 1) verbraucht werden. Jede Bedingung muss über ein langes und ein kurzes Fenster gelten, wobei das kurze Fenster ein Zwölftel des langen ist (jeweils 5 Minuten, 30 Minuten und 6 Stunden). Das lange Fenster stellt sicher, dass das Problem erheblich ist; das kurze stellt sicher, dass es noch andauert, sodass der Alert wenige Minuten nach einem Fix aufhört auszulösen, statt eine Stunde lang rot zu bleiben.

Fast Burn und Slow Burn



Ein Fast Burn verbraucht einen großen Teil des Budgets in Stunden und sollte jemanden wecken; ein Slow Burn verbraucht es über Tage und gehört in ein Ticket.

```
SLO = 0.999                # 99.9% of requests good
PERIOD_H = 30 * 24         # 720-hour SLO window

def burn_rate(bad, total, slo=SLO):
    """Observed error ratio divided by the ratio the SLO allows."""
    if total == 0:
        return 0.0
    return (bad / total) / (1 - slo)

def budget_spent(rate, window_h, period_h=PERIOD_H):
    """Fraction of the whole window's budget used in window_h hours."""
    return rate * window_h / period_h

def page(w1h, w5m, w6h, w30m):
    """Each argument is (bad, total) for that window."""
    fast = burn_rate(*w1h) >= 14.4 and burn_rate(*w5m) >= 14.4
    slow = burn_rate(*w6h) >= 6 and burn_rate(*w30m) >= 6
    return fast or slow

print(round(burn_rate(bad=288, total=20_000), 1))   # 14.4
print(budget_spent(14.4, window_h=1))               # 0.02, i.e. 2%
```

Burn Rate, verbrauchtes Budget und eine Alarmierungsregel mit zwei Fenstern. In der Produktion wird dieselbe Logik als Alerting-Regeln im Monitoring-System geschrieben (zum Beispiel als Recording-und Alerting-Regeln in Prometheus).

Die Schwellen legen auch die Erkennungszeit fest. Fällt die Bestell-API vollständig aus (Fehlerquote 100 %, Burn Rate 1000), überschreitet die Ein-Stunden-Fehlerquote 1,44 % nach 0,0144 × 60 = etwa 0,9 Minuten, die Alarmierung erfolgt also in unter einer Minute. Schlagen 2 % der Anfragen fehl (Burn Rate 20), erreicht die Ein-Stunden-Quote 1,44 % nach 0,72 × 60 = etwa 43 Minuten. Schwere Probleme alarmieren schnell; leichte nur, wenn sie anhalten, und genau dieses Verhalten wünscht man sich um drei Uhr morgens.

Services mit wenig Traffic sprengen diese Rechnung: Bei 10 Anfragen pro Stunde ist ein einziger Fehlschlag eine Fehlerquote von 10 % und eine Burn Rate von 100. Die Abhilfen des Workbook sind synthetischer Traffic, der Signal hinzufügt, das Zusammenfassen verwandter Services mit wenig Traffic unter einem SLO, einzelne Fehlschläge durch Wiederholungen und Fallbacks weniger ins Gewicht fallen zu lassen oder ein längeres Zeitfenster zu akzeptieren. Für das Ticketsystem, das zwischen Vorverkaufsstarts fast leerläuft und dann extrem ausgelastet ist, funktionieren SLOs pro Ereignis mit Burn-Rate-Alerts während des Ereignisfensters besser als eine Monatszahl.

SLAs und Service-Gutschriften

Ein SLA verlagert das Ziel vom Engineering ins Vertragsrecht. Es definiert den abgedeckten Service, wie Verfügbarkeit gemessen wird, Ausschlüsse (geplante Wartung, Missbrauch durch Kunden, höhere Gewalt), die Abhilfe und wie ein Kunde sie geltend macht. Die übliche Abhilfe ist eine Service-Gutschrift: ein Prozentsatz der Monatsgebühr, der erstattet wird, gestaffelt danach, wie deutlich das Niveau verfehlt wurde. Gutschriften sind typischerweise klein im Vergleich zum tatsächlichen Schaden des Kunden; deshalb versteht man SLAs besser als Signal für das Vertrauen des Anbieters denn als Versicherung; Kunden, deren Schäden groß wären, brauchen eigene Resilienz, keine größere Gutschrift.

Betrachten Sie ein illustratives SLA, das die Essenslieferplattform Restaurantpartnern für ihre API zur Bestellannahme anbieten könnte: 99,5 % monatliche Verfügbarkeit, mit einer Gutschrift von 10 % auf die Plattformgebühr dieses Monats unter 99,5 % und 25 % unter 99,0 %. Das interne SLO liegt bei 99,9 %. Die Lücke dazwischen ist gewollt: Das Team sollte sein SLO-Budget erschöpfen, seine Policy anwenden und sich erholen, lange bevor der Vertrag gefährdet ist. Setzen Sie das SLO nie gleich dem SLA; Sie würden erst von Problemen erfahren, wenn Sie bereits Geld schulden.

Daraus folgen zwei architektonische Konsequenzen. Erstens kann ein SLA nur ehrlich sein, wenn Sie die SLAs Ihrer eigenen Abhängigkeiten kennen (Cloud-Anbieter, Zahlungsdienstleister, Kartenanbieter) und wissen, wie sie sich kombinieren. Zweitens brauchen SLAs Belege: Die Messmethode im Vertrag muss eine sein, die Sie aus Ihrer eigenen Telemetrie liefern können; das ist eine Anforderung an das Design der Observability.

Häufige Fehler

Zu viele SLOs. Ein Service mit dreißig SLOs hat keines, weil niemand über sie nachdenken oder danach handeln kann. Beginnen Sie mit zwei oder drei pro kritischem Nutzerpfad: meist einem Verfügbarkeits- und einem oder zwei Latenz-SLOs. **Das Falsche messen.** CPU-Auslastung, Queue-Tiefe und Replikationsverzögerung sind nützliche Ursachen zum Beobachten, aber keine SLIs, weil Nutzer sie nicht direkt erleben. **An der falschen Stelle messen.** Serverseitige Erfolgsquoten sehen perfekt aus, während das CDN Fehler liefert. **Mittelwerte bilden.** Ein SLO auf die mittlere Latenz verbirgt den langsamen Tail, über den Nutzer sich beschweren.

Ziele ohne Konsequenzen. Ein SLO ohne Budget-Policy und ohne Burn-Rate-Alerts ist Dashboard-Dekoration. **Ziele, die nur Engineers festlegen.** Der Product Owner muss zustimmen, weil das SLO Features gegen Zuverlässigkeit abwägt. **Nie überarbeiten.** Überprüfen Sie SLOs jedes oder jedes zweite Quartal: Beschweren sich Nutzer, während das SLO grün ist, ist das SLI oder das Ziel falsch; wird das Budget nie angetastet, ist das Ziel vielleicht zu locker oder das Team zu vorsichtig. **Ein SLO für sehr unterschiedliche Nutzerpfade.** Wer das Durchsuchen der Speisekarte und die Zahlung in einem Verhältnis zusammenwirft, lässt reichliche, günstige Anfragen die Fehlschläge bei den seltenen, wertvollen verdecken.

SLOs sind mit dem Rest dieses Kapitels verbunden. Burn-Rate-Alerts sind die symptombasierten Alerts, die unter Observability empfohlen werden; Incidents, die das Budget verbrauchen, fließen in Post-Incident-Reviews ein; und das Antwortmaß eines SLO ist eine natürliche Fitness Function für die Architektur.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Stellen Sie sich das Error Budget als monatliches Taschengeld für Fehler vor. Das SLI ist der Kontoauszug, das SLO das Taschengeld, das Sie mit Ihrer Familie vereinbart haben, und das SLA der Kreditvertrag mit der Bank, die Ihnen Gebühren berechnet, wenn Sie Ihr Konto stark überziehen. Das Taschengeld auszugeben ist in Ordnung; dafür ist es da. Ist es aufgebraucht, kaufen Sie bis zum nächsten Monat keine Leckereien (neue Features) mehr.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum wird ein SLI meist als gute Ereignisse geteilt durch gültige Ereignisse geschrieben?

Weil das Verhältnis immer zwischen 0 % und 100 % liegt, sich durch Zählen sauber über Server und Zeit addieren lässt und jedes SLI auf dieselbe Weise gelesen, verglichen und in ein Error Budget umgewandelt werden kann.

▼ Ein SLO von 99,9 % über 30 Tage verzeichnet sechs Stunden lang eine Burn Rate von 6. Welcher Anteil des Budgets wird verbraucht?

6 × 6 / 720 = 0,05, also 5 % des 30-Tage-Budgets.

▼ Warum sollte ein SLA lockerer sein als das interne SLO?

Damit das Team ein aufgebrauchtes Error Budget erkennt und darauf reagiert, lange bevor der Vertrag verletzt wird und Service-Gutschriften fällig werden.

▼ Was leistet das kurze Fenster in einem Multi-Window-Alert?

Es bestätigt, dass das Problem noch andauert, sodass sich der Alert wenige Minuten nach einem Fix zurücksetzt, statt das ganze lange Fenster über aktiv zu bleiben.

QUELLEN

- Google SRE book: Service Level Objectives (2016): https://sre.google/sre-book/service-level-objectives/
- The Site Reliability Workbook: Implementing SLOs (2018): https://sre.google/workbook/implementing-slos/
- The Site Reliability Workbook: Alerting on SLOs: https://sre.google/workbook/alerting-on-slos/
- The Site Reliability Workbook: Example error budget policy: https://sre.google/workbook/error-budget-policy/

Observability: Logs, Metriken und Traces

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.4 Performance und Skalierbarkeit
- 6.1 Warum verteilen
- 9.5 SLOs und Error Budgets

Begriffe von früher, die er verwendet: Perzentil (p95, p99), Service Level Objective (SLO), Service Level Indicator (SLI), Auslastung, Personenbezogene Daten

DIE IDEE IN KÜRZE

Ein Auto-Armaturenbrett zeigt ein paar Anzeigen und Warnleuchten: Geschwindigkeit, Tankfüllung, Motortemperatur. Das ist Monitoring: Es informiert Sie über Probleme, die jemand vorhergesehen hat. Macht das Auto ein seltsames Geräusch, das keine Leuchte erklärt, schließt eine Mechanikerin oder ein Mechaniker einen Diagnosecomputer an und liest detaillierte Aufzeichnungen darüber, was jedes Teil getan hat, um einen Fehler zu finden, den niemand vorhergesehen hat. Observability bedeutet, Software so zu bauen, dass diese Art der Untersuchung immer möglich ist: Jedes Teil führt klare Aufzeichnungen, und die Aufzeichnungen einer Fahrt lassen sich von Teil zu Teil verfolgen.

WAS SIE WISSEN MÜSSEN

- Monitoring prüft vordefinierte Bedingungen; Observability ist die Fähigkeit, neues, nicht vorhergesehenes Verhalten aus vorhandener Telemetrie zu erklären, und muss durch konsistente Instrumentierung und gemeinsame Identifikatoren von Anfang an eingeplant werden.
- Metriken sind günstige Aggregate für Trends und Alerts, strukturierte Logs zeichnen einzelne Ereignisse auf, Traces zeigen den Weg einer Anfrage durch Services, und Profile zeigen, wo Code Ressourcen verbraucht.
- Counter, Gauges und Histogramme sind die zentralen Metriktypen; Perzentile lassen sich nicht über Server oder Zeit mitteln, Histogramm-Buckets aber addieren, und das Perzentil wird aus dem zusammengeführten Ergebnis abgelesen.
- RED (Rate, Errors, Duration) beschreibt Services, USE (Utilisation, Saturation, Errors) beschreibt Ressourcen, und die Four Golden Signals sind Latenz, Traffic, Fehler und Sättigung.

- Distributed Tracing (Dapper, 2010) verknüpft Spans durch Kontextweitergabe, standardisiert durch den W3C-Header traceparent; OpenTelemetry, ein 2026 graduiertes CNCF-Projekt, ist der herstellerneutrale Standard für die Erzeugung von Traces, Metriken und Logs.
- Unbegrenzte Label-Werte lassen die Kardinalität von Metriken und die Kosten explodieren; Details pro Nutzer oder pro Bestellung gehören daher in Traces und Logs, und Tail-based Sampling behält die Traces, auf die es ankommt.
- Alarmieren Sie bei Symptomen, die Nutzer spüren, etwa SLO-Burn-Rates, nicht bei Ursachen wie der CPU, und bauen Sie jedes Dashboard so, dass es eine Frage für ein Publikum beantwortet.

Monitoring und Observability

Monitoring bedeutet, vordefinierte Messwerte und Prüfungen zu erfassen und Alarm zu schlagen, wenn sie eine Schwelle überschreiten: Festplatte zu 90 % voll, Fehlerrate über 1 %, Health Check schlägt fehl. Es beantwortet Fragen, die man vorher zu stellen wusste. Observability (Beobachtbarkeit) ist eine Eigenschaft des Systems: wie gut Sie seinen inneren Zustand verstehen können, auch Zustände, die niemand vorhergesehen hat, und zwar aus der Telemetrie, die es bereits ausgibt, ohne neuen Code auszuliefern, um es herauszufinden. Der Begriff stammt aus der Regelungstechnik, wo Rudolf Kálmán ihn 1960 als die Fähigkeit definierte, den inneren Zustand eines Systems aus seinen Ausgaben zu erschließen.

Die Unterscheidung ist wichtig, weil verteilte Systeme meist auf neue Weise ausfallen. Eine Essensbestellung kann langsam sein, weil die Tablet-App eines Restaurants aggressiv wiederholt, weil der Endpoint eines Zahlungsanbieters in einer Region beeinträchtigt ist oder weil ein neues Speisekarten-Feature eine Abfrage nur für Restaurants mit mehr als 500 Gerichten verlangsamt. Für keinen dieser Fälle hat jemand ein Dashboard gebaut. Monitoring sagt Ihnen, *dass* der Bezahlvorgang langsam ist; Observability lässt Sie fragen, *für wen, wo und warum*, indem Sie die Telemetrie nach beliebigen Attributen aufschlüsseln, bis das Muster sichtbar wird.

Fachleute sind uneins darüber, wie weit die Unterscheidung reicht. Eine Sicht, verbreitet bei Tool-Anbietern, versteht Observability als Dach über drei Säulen: Logs, Metriken und Traces. Charity Majors, Liz Fong-Jones und George Miranda (*Observability Engineering*, O'Reilly, 2022) argumentieren, dass das Säulenmodell den Kern verfehlt und Observability reichhaltige, strukturierte Ereignisse mit hoher Kardinalität erfordert, die sich in beliebigen Kombinationen abfragen lassen. Beide Lager sind sich über die architektonische Konsequenz einig: Observability ist ein Qualitätsmerkmal, das durch konsistente Instrumentierung, gemeinsame Identifikatoren und Kontextweitergabe von Anfang an eingeplant werden muss, weil es sich während eines Incident nicht nachrüsten lässt.

Arten von Telemetrie

Signal	Was er ist	Am besten für	Hauptkosten oder Grenze
Metriken	Numerische Zeitreihen, pro Intervall aggregiert und mit wenigen Dimensionen gelabelt	Günstige Langzeittrends, Dashboards, Alerting auf SLIs	Voraggregiert: kann keine Fragen zu einzelnen Anfragen beantworten; Labels müssen niedrige Kardinalität haben
Logs	Aufzeichnungen einzelner Ereignisse mit Zeitstempel, idealerweise strukturiert	Details zu einem Ereignis: ein Fehler, eine Entscheidung, ein Audit-Eintrag	Volumen und Kosten wachsen mit dem Traffic; Freitext ist schwer abzufragen
Traces	Bäume zeitlich erfasster Spans, die den Weg einer Anfrage durch Services zeigen	Herausfinden, wo die Zeit bleibt und welcher Hop in einem verteilten Aufruf fehlschlug	Müssen bei großem Volumen gesampelt werden; nutzlos, wenn ein Hop den Kontext verliert
Profile	Stichproben, wo Code CPU-Zeit verbraucht oder Speicher belegt	Auf Code-Ebene erklären, warum ein Service langsam oder teuer ist	Neuere Werkzeuge; kontinuierliches Profiling verursacht etwas Overhead
Wide Events	Ein Datensatz mit vielen Attributen pro Anfrage und pro Service	Ad-hoc-Aufschlüsselung nach beliebigen Attributen (Kunde, Version, Region)	Braucht einen Speicher, der für Abfragen mit hoher Kardinalität gebaut ist

Strukturiertes Logging

Eine Logzeile wie `Payment failed for order 81F3 after 3001ms` ist leicht zu schreiben und schwer zu nutzen: Jede Abfrage braucht einen fragilen regulären Ausdruck, und jeder Entwickler formuliert anders. Strukturiertes Logging schreibt jeden Eintrag als Schlüssel-Wert-Felder, meist JSON, mit vereinbarten Namen: Zeitstempel, Level, Service, Ereignisname, fachliche Identifikatoren (Bestellung, Restaurant, Kurier), Ergebnis, Dauer sowie die Trace- und Span-Identifikatoren, die die Zeile mit ihrem Trace verknüpfen. Abfragen werden präzise („alle card_authorisation_failed-Ereignisse für den Anbieter cardco in der letzten Stunde, gruppiert nach http_status“), und Logs verschiedener Services lassen sich verknüpfen.

Ein paar Regeln halten Logs nützlich und bezahlbar. Loggen Sie Ereignisse, keine Prosa; verwenden Sie einen festen Ereignisnamen pro Art von Ereignis. Verwenden Sie Levels konsistent, und loggen Sie nicht auf Error-Level für Zustände, bei denen niemand handeln muss. Loggen Sie nie Secrets, Tokens, vollständige Kartennummern oder unnötige personenbezogene Daten; Telemetrie-Pipelines werden kopiert, an Dritte übermittelt und

monatelang aufbewahrt, sie sind daher eine häufige Quelle von Datenlecks, und personenbezogene Daten in Logs fallen wie jede andere Kopie unter die DSGVO (Abschnitt 8.7). Security Logging und Erkennung, ein verwandter, aber anderer Zweck mit eigenen Anforderungen an Aufbewahrung und Integrität, werden in Abschnitt 8.8 behandelt.

```
traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
             version-trace id (16 bytes)-parent span id (8 bytes)-flags (01 = sampled)

{"ts": "2026-10-22T12:31:07.412Z", "level": "error",
 "service": "payment-service", "event": "card_authorisation_failed",
 "order_id": "ord_81F3", "provider": "cardco", "http_status": 504,
 "duration_ms": 3001, "attempt": 2,
 "trace_id": "4bf92f3577b34da6a3ce929d0e0e4736", "span_id": "00f067aa0ba902b7"}
```

Ein W3C-traceparent-Header und ein strukturierter Logeintrag des Zahlungsservice. Die gemeinsame trace_id lässt eine Abfrage von dieser Logzeile zum ganzen Trace springen.

Metriken und warum sich Perzentile nicht mitteln lassen

Die meisten Metriksysteme, etwa Prometheus (ein graduiertes CNCF-Projekt) und das Metrikmodell von OpenTelemetry, bieten dieselben wenigen Typen. Ein Counter steigt nur, etwa die Gesamtzahl bedienter Anfragen; Sie lesen ihn als Rate (Anfragen pro Sekunde über die letzten fünf Minuten), was auch Neustarts übersteht. Ein Gauge ist ein aktueller Wert, der steigt und fällt: Queue-Länge, belegte Verbindungen, Speicher. Ein Histogramm zählt Beobachtungen in Buckets (zum Beispiel Anfragen unter 50 ms, unter 100 ms, unter 250 ms und so weiter) plus eine Gesamtzahl und eine Summe, aus denen Perzentile geschätzt werden. Prometheus hat außerdem einen Summary-Typ, der Quantile in jedem Prozess berechnet.

Die Unterscheidung zwischen Histogrammen und Summaries ist kein Detail. Wie Abschnitt 2.4 festgestellt hat, ist ein Perzentil eine Eigenschaft einer ganzen Verteilung, und das Perzentil einer kombinierten Verteilung lässt sich nicht aus den Perzentilen ihrer Teile berechnen. Wer p99-Werte über Server oder über Minuten mittelt, erhält eine Zahl, die kein Perzentil von irgendetwas ist. Bucket-Zählungen lassen sich dagegen einfach addieren: Summieren Sie jeden Bucket über alle Server und schätzen Sie dann das Perzentil aus dem zusammengeführten Histogramm. Deshalb sollten flottenweite Latenz-SLIs auf Histogrammen aufbauen (oder auf Zählungen von Anfragen unter einer Schwelle, wie in Abschnitt 9.5), nie auf Summaries pro Instanz.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: zwei p99-Werte mitteln

Der Speisekartenservice läuft auf zwei Instanzen, die in einem Fünf-Minuten-Intervall jeweils 10.000 Anfragen verarbeiten. Instanz A ist gesund: Alle ihre Anfragen dauern 20 bis 300 ms, und ihr p99 liegt bei 180 ms. Instanz B hat einen lauten Nachbarn: 7000 Anfragen dauern 20 bis 300 ms, aber 3000 verteilen sich gleichmäßig zwischen 500 und 1100 ms.

p99 von B (die langsamsten 100 von 10.000)	1100 - 600 × (100 / 3000) = 1080 ms
Mittelwert der beiden p99	(180 + 1080) / 2 = 630 ms
Wahres p99 aller 20.000 Anfragen	Die langsamsten 200 liegen alle auf B: 1100 - 600 × (200 / 3000) = 1060 ms
Fehler durch Mittelung	630 ms gemeldet gegenüber 1060 ms erlebt: Das Dashboard unterschätzt den Tail um 430 ms
Mit Histogrammen	Addieren Sie die Bucket-Zählungen von A und B und lesen Sie das 99. Perzentil aus den zusammengeführten Buckets ab: etwa 1060 ms, begrenzt nur durch die Bucket-Breite

Gemittelte Perzentile können in beide Richtungen falsch sein, und der Fehler wächst, wenn Instanzen unterschiedlichen Traffic tragen.

Dasselbe gilt über die Zeit: Das p99 einer Stunde ist nicht der Mittelwert von zwölf p99-Werten über je fünf Minuten.

Was messen: RED, USE und die Golden Signals

Drei kurze Checklisten decken das meiste ab, was ein Service bereitstellen sollte. Die RED-Methode, 2015 von Tom Wilkie vorgeschlagen, gilt für alles, was Anfragen bedient: **Rate** (Anfragen pro Sekunde), **Errors** (fehlgeschlagene Anfragen pro Sekunde) und **Duration** (die Verteilung der Antwortzeiten). Die USE-Methode von Brendan Gregg gilt für Ressourcen: Prüfen Sie für jede CPU, jede Festplatte, jede Netzwerkverbindung, jeden Connection-Pool und jede Queue **Utilisation** (wie ausgelastet sie ist), **Saturation** (wie viel Arbeit wartet) und **Errors**. Die Four Golden Signals aus dem Kapitel des Google-SRE-Buchs über das Monitoring verteilter Systeme sind **Latenz**, **Traffic**, **Fehler** und **Sättigung**: im Grunde RED plus Sättigung.

In der Praxis beschreibt RED, was Nutzer erleben, und USE erklärt, warum. Steigt die Dauer beim Zahlungsservice (RED), könnte USE auf seinen Ressourcen einen Connection-Pool mit 100 % Auslastung und 40 wartenden Anfragen zeigen (Sättigung), was wiederum auf den langsamen Kartenanbieter verweist, der Verbindungen offen hält. Messen Sie die Latenz für Erfolge und Fehlschläge getrennt: Ein schneller Fehler ist keine gute Latenz, und ein Timeout ist sowohl ein Fehler als auch eine langsame Antwort.

Drei Checklisten im Vergleich

Checkliste	Gilt für	Signale	Typische Frage, die sie beantwortet
RED (Wilkie)	Anfragegetriebene Services	Rate, Errors, Duration	Bedient der Bestellservice die Nutzer gerade gut?
USE (Gregg)	Ressourcen: CPU, Speicher, Festplatten, Pools, Queues	Utilisation, Saturation, Errors	Welche Ressource ist der Engpass?
Four Golden Signals (SRE-Buch)	Nutzerseitige Systeme	Latenz, Traffic, Fehler, Sättigung	Ist das System gesund, und wie nah ist es an seinen Grenzen?

Distributed Tracing

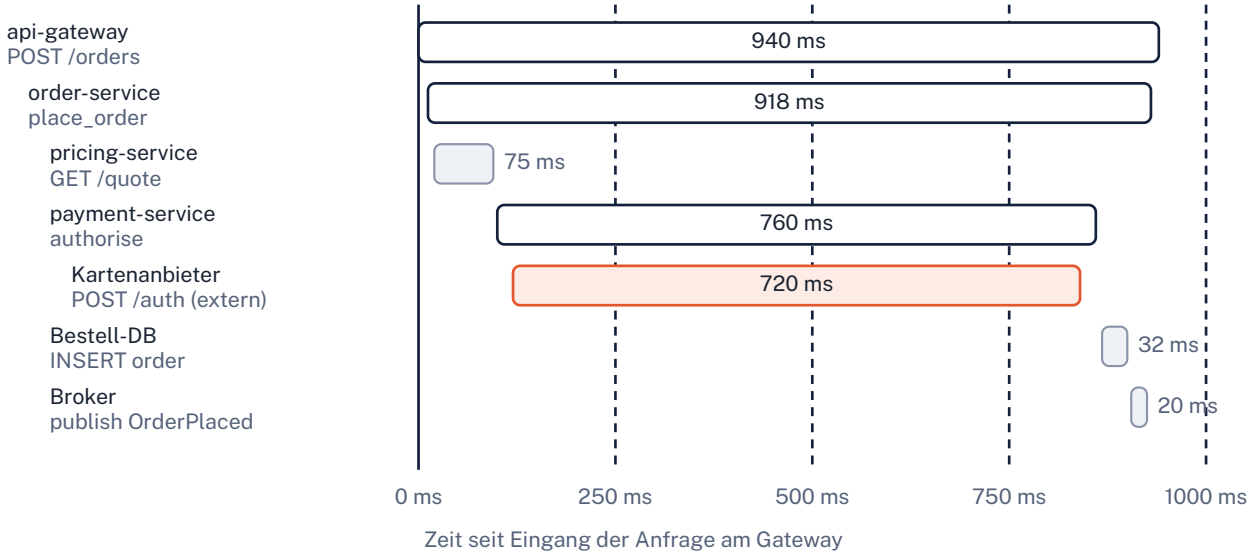
In einem Monolithen zeigt ein Stack Trace, wo eine langsame Anfrage ihre Zeit verbracht hat. Über Services hinweg gibt es keinen einzelnen Stack. Distributed Tracing baut einen nach: Jede Anfrage erhält am Rand einen Trace-Identifikator, jede Arbeitseinheit wird zu einem Span (mit Name, Startzeit, Dauer, Status, Attributen und dem Identifikator ihres Eltern-Spans), und ein Tracing-Backend setzt die Spans zu einem Baum zusammen. Googles Dapper-Paper (Sigelman und Kollegen, 2010) beschrieb den Ansatz in großem Maßstab: geringer Overhead, Instrumentierung in gemeinsamen Bibliotheken statt im Anwendungscode und Sampling, sodass nur ein Bruchteil der Anfragen aufgezeichnet wird.

Tracing hängt vollständig von der Kontextweitergabe (Context Propagation) ab: Jeder Service muss den Trace-Kontext an den nächsten weitergeben. Die Spezifikation W3C Trace Context (seit 2020 eine W3C Recommendation) standardisiert das für HTTP mit einem `traceparent`-Header, der eine Version, eine 16-Byte-Trace-ID, die 8-Byte-ID des aufrufenden Spans und Flags enthält, etwa ob der Trace gesampelt wird, plus einem optionalen `tracestate`-Header für herstellerspezifische Daten. Beim Messaging reisen dieselben Werte in Nachrichtenheadern, und der Span des Consumers verweist auf den des Producers. Ein einziger Service, der den Header verliert (eine alte Bibliothek, ein handgeschriebener HTTP-Client, ein Batch-Job), zerbricht jeden Trace, der durch ihn läuft; deshalb gehört die Weitergabe in gemeinsame Plattformbibliotheken, nicht in den Code jedes Teams.

Jeden Span jeder Anfrage aufzuzeichnen ist bei großem Volumen teuer, daher werden Traces gesampelt. Head-based Sampling entscheidet zu Beginn der Anfrage (1 von 100 behalten) und ist günstig, verwirft aber die meisten der seltenen langsamen und fehlgeschlagenen Anfragen, auf die es Ihnen ankommt. Tail-based Sampling puffert Spans, bis die Anfrage abgeschlossen ist, und behält dann alle Fehler, alle Anfragen über einer Schwelle und einen kleinen Teil des Rests; es braucht mehr Speicher im Collector, behält aber die interessanten Traces.

Ein Trace über die Services der Essenslieferplattform

Trace 4bf92f35…: eine Anfrage „Bestellen“ (jeder Balken ist ein Span)



Der Wasserfall zeigt auf einen Blick, dass 720 der 940 ms auf das Warten auf den externen Kartenanbieter entfallen, nicht auf die eigenen Services der Plattform.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: eine Bestellung durch die Services verfolgen

Die schwierigste Idee dieses Abschnitts ist, wie ein Trace aus Teilen zusammengesetzt wird, die verschiedene Services aufzeichnen, von denen keiner die ganze Anfrage sieht. Der Klebstoff sind zwei Identifikatoren, die bei jedem Aufruf mitgegeben werden.

- Ein Kunde gibt eine Bestellung auf. Das API-Gateway empfängt POST /orders, gibt der Anfrage eine neue Trace-ID und startet seinen eigenen Span: einen Datensatz mit Namen, Startzeit und, sobald er endet, einer Dauer.
- Ruft das Gateway den Bestellservice auf, fügt es einen traceparent-Header hinzu, der die Trace-ID und die ID seines eigenen Spans trägt. Der Bestellservice startet seinen place_order-Span und vermerkt den Span des Gateways als dessen Eltern-Span.
- Der Bestellservice macht dasselbe bei jedem Aufruf, den er tätigt, an Preisgestaltung und Zahlung, und legt den Trace-Kontext in die Header des Ereignisses OrderPlaced. Jeder neue Span vermerkt die Trace-ID und seinen Eltern-Span.
- Jeder Service sendet seine abgeschlossenen Spans separat an das Tracing-Backend. Die gemeinsame Trace-ID lässt das Backend sie einsammeln, und die Eltern-Verweise lassen es sie zu dem Wasserfall im Diagramm anordnen.
- Angenommen nun, der Bestellservice hätte die Preisgestaltung über einen handgeschriebenen HTTP-Client aufgerufen, der den Header nicht hinzufügt. Die Preisgestaltung würde einen eigenen neuen Trace starten, und ihre Spans würden im Wasserfall der Bestellung fehlen.
- Deshalb gehört die Kontextweitergabe in gemeinsame Bibliotheken, die jedes Team nutzt: Ein Hop, der sie vergisst, zerbricht jeden Trace, der durch ihn läuft.
- **Die Lehre:** Jeder Service zeichnet nur seinen eigenen Teil einer Anfrage auf. Die Trace-ID und die Eltern-Verweise, die bei jedem Aufruf mitreisen, machen aus diesen Teilen eine Geschichte; ein Trace ist also nur so vollständig wie sein schwächster Hop.

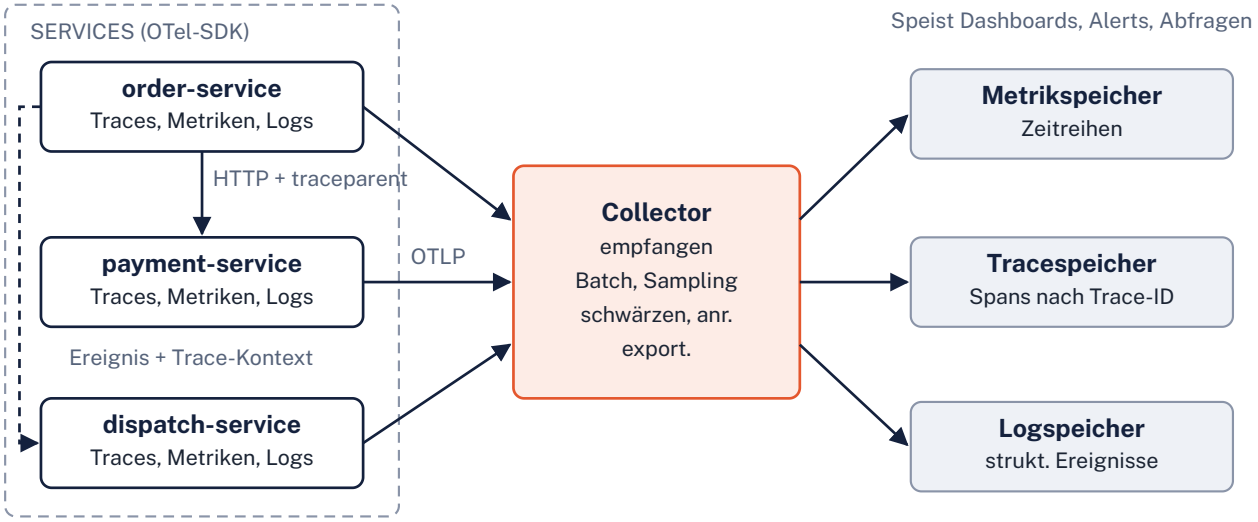
OpenTelemetry: ein Standard für die Instrumentierung

Bis etwa 2019 hatte jeder Tracing- und Monitoring-Anbieter eigene Agents und Bibliotheken, ein Anbieterwechsel bedeutete also, jeden Service neu zu instrumentieren. OpenTelemetry (OTel) entstand 2019 durch die Zusammenführung zweier früherer Open-Source-Projekte, OpenTracing und OpenCensus, unter der Cloud Native Computing Foundation (CNCF) und wurde im Mai 2026 als CNCF-Projekt graduiert. Es bietet Sprach-APIs und SDKs, automatische Instrumentierung für gängige Frameworks und Clients, das Wire-Protokoll OTLP, den Collector und semantische Konventionen, die Attributnamen wie `http.request.method` und

`http.response.status_code` festlegen, damit Telemetrie verschiedener Teams und Sprachen zusammenpasst. Traces, Metriken und Logs sind stabil; Profile kamen als viertes Signal hinzu und gingen 2026 in eine öffentliche Alphaphase.

Der architektonische Wert liegt in der Trennung von Instrumentierung und Backend. Services senden OTLP an einen Collector, der meist neben jedem Knoten oder als zentrales Gateway läuft; der Collector bündelt, sampelt, entfernt sensible Attribute, ergänzt Metadaten wie die Deployment-Version und exportiert in die Speicher, die die Organisation nutzt. Ein Anbieterwechsel oder das Senden derselben Daten an zwei Backends während einer Migration wird zu einer Konfigurationsänderung am Collector statt zu einer Codeänderung in zweihundert Services. Ein Plattformteam stellt typischerweise SDK-Einrichtung, Weitergabe und Collector als Paved Road bereit, sodass Produktteams nur fachliche Attribute ergänzen.

Telemetrie-Pipeline



Die Instrumentierung liegt in den Services, die Verarbeitung im Collector und die Speicherung in austauschbaren Backends.

```
from opentelemetry import trace
import structlog

tracer = trace.get_tracer("order-service")
log = structlog.get_logger()

def place_order(order):
    with tracer.start_as_current_span("place_order") as span:
        span.set_attribute("restaurant.id", order.restaurant_id)
        span.set_attribute("order.item_count", len(order.items))
        quote = pricing.quote(order)       # instrumented HTTP client adds
        payment.authorise(order, quote)    # the traceparent header itself
        log.info("order_placed", order_id=order.id,
                 total_pence=quote.total_pence)
        return order.id
```

Manuelle Instrumentierung mit der OpenTelemetry-Python-API und einem strukturierten Logger. Automatisch instrumentierte HTTP-Clients erzeugen Kind-Spans und geben den Kontext ohne zusätzlichen Code weiter.

Kardinalität und Kosten

Telemetrie kann leicht mehr kosten als das System, das sie beobachtet. Der Haupttreiber bei Metriken ist die Kardinalität: Jede eindeutige Kombination von Label-Werten ist eine eigene Zeitreihe, die gespeichert und indiziert werden muss. Labels mit wenigen Werten (HTTP-Methode, Statusklasse, Region) sind unproblematisch. Labels mit unbegrenzten Werten (Nutzer-ID, Bestell-ID, vollständige URL mit Identifikatoren darin, Text von Fehlermeldungen) vervielfachen die Zeitreihen ohne Grenze und können ein Metriksystem lahmlegen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Zeitreihen zählen

Der Bestellservice exportiert ein Histogramm der Anfragedauer mit 12 Buckets. Prometheus speichert jeden Bucket als Zeitreihe, plus eine Summe und eine Anzahl, also 14 Zeitreihen pro Label-Kombination.

Paare aus Route und Methode	60
Statusklassen (1xx bis 5xx)	5
Instanzen	30
Zeitreihen für diese eine Metrik	60 × 5 × 30 × 14 = 126,000
Ein Region-Label hinzufügen (5 Regionen)	126.000 × 5 = 630.000 Zeitreihen
restaurant_id hinzufügen (40.000 Restaurants)	126.000 × 40.000 = etwa 5 Milliarden mögliche Zeitreihen: nicht machbar

Fragen pro Restaurant oder pro Kunde gehören in Traces, Logs oder Wide Events, wo hohe Kardinalität erwartet wird, nicht in Metrik-Labels.

Exemplars, die einem Histogramm-Bucket eine Beispiel-Trace-ID anhängen, verknüpfen eine Latenzspitze in einer günstigen Metrik mit einem teuren, detaillierten Trace.

Weitere Kostenhebel sind das Sampling von Traces (Tail-based Sampling behält die nützlichen), kürzere Aufbewahrung für Detaildaten und längere für Aggregate, standardmäßig keine Debug-Logs in der Produktion sowie die Weiterbelastung der Telemetrie an die Teams, die sie erzeugen, damit ihre Kosten sichtbar werden. Behandeln Sie das Telemetrievolumen wie jede andere Ressource als Designparameter mit einem Budget.

Alerting auf Symptome, nicht auf Ursachen

Die Empfehlung des SRE-Buchs lautet, dass Alarmierungen Symptome betreffen sollten, die Nutzer spüren, nicht die vielen möglichen Ursachen. Hohe CPU-Last, eine volle Festplatte auf einem Replikat oder ein neu gestarteter Container sind Ursachen: oft harmlos, manchmal schon von der Automatisierung behandelt und nie eine vollständige Liste dessen, was schiefgehen

kann. Ein brennendes SLO, fehlschlagende Bezahlvorgänge oder eine stockende Pipeline für Kurierpositionen sind Symptome: Sie zählen, was auch immer ihre Ursache ist. Symptombasierte Alerts sind weniger zahlreich, selten falsch und decken Ausfälle ab, die niemand vorhergesehen hat; ursachenbasierte Signale gehören auf Dashboards, die helfen, ein Symptom zu diagnostizieren, sobald es alarmiert wurde.

Jede Alarmierung sollte dringend, umsetzbar und echt sein. Alert-Müdigkeit ist an sich schon ein Zuverlässigkeitsrisiko: Engineers in Rufbereitschaft, die von bedeutungslosen Alerts geweckt werden, lernen, sie abzutun, und verpassen den einen, der zählt. Eine praktische Regel ist, bei SLO-Burn-Rates zu alarmieren (Abschnitt 9.5), langsamere Burns und Ressourcenwarnungen an eine Ticket-Queue zu senden und jede Alarmierung mit einem Runbook und einem Dashboard zu verknüpfen.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Sollte dieser Alert einen Menschen alarmieren?

Stellen Sie diese Fragen zu jeder Alert-Regel, bevor sie jemanden wecken darf, und erneut, wann immer sie auslöst, ohne dass jemand handeln muss.

Q1 **Bedeutet die Bedingung, dass Nutzer jetzt oder bald zu Schaden kommen (ein schnell brennendes SLO, ein fehlschlagender Nutzerpfad)?**

- **Ja** Weiter zur nächsten Frage.
- **Nein** Nicht alarmieren. Stellen Sie es zur Diagnose auf ein Dashboard oder eröffnen Sie ein Ticket, wenn sich jemand während der Arbeitszeit darum kümmern sollte.

Q2 **Muss ein Mensch handeln, statt einer automatischen Reaktion wie einem Neustart oder einer Skalierung nach außen?**

- **Ja** Weiter zur nächsten Frage.
- **Nein** Automatisieren Sie die Reaktion und alarmieren Sie nur, wenn die Automatisierung fehlschlägt.

Q3 **Muss die Maßnahme vor dem nächsten Arbeitstag erfolgen?**

- **Ja** Alarmieren Sie und verknüpfen Sie den Alert mit einem Runbook und dem passenden Dashboard.
- **Nein** Senden Sie ihn statt an den Pager an die Ticket-Queue des Teams.

Q4 **Hat dieser Alert im letzten Monat alarmiert, ohne dass jemand etwas tun musste?**

- **Ja** Korrigieren Sie die Schwelle oder das Signal oder löschen Sie den Alert: Eine Alarmierung, auf die niemand reagiert, lehrt Menschen, Alarmierungen zu ignorieren.
- **Nein** Behalten Sie ihn und überprüfen Sie ihn bei der nächsten Übergabe der Rufbereitschaft.

Dashboards, die Fragen beantworten

Eine Wand aus fünfzig Graphen beantwortet keine Frage schnell. Gute Dashboards werden wie jede Benutzeroberfläche gestaltet: für ein benanntes Publikum und eine konkrete Frage. Eine Service-Übersicht beantwortet „Ist dieser Service gesund?“ mit seinen SLOs, dem verbleibenden Error Budget und den RED-Metriken oben und den USE-Metriken seiner wichtigsten Ressourcen darunter. Ein Dashboard für den Nutzerpfad beim Vorverkaufsstart

beantwortet „Bekommen die Fans Tickets?“ mit Zulassungen, Sitzplatzreservierungen, Zahlungen und Fehlschlägen pro Minute von Ende zu Ende. Diagnose-Dashboards erreicht man durch Drill-down; sie werden nicht zuerst gezeigt.

Halten Sie Dashboards als Code in der Versionsverwaltung, geben Sie jedem einen Verantwortlichen, zeigen Sie Deployments und Änderungen an Feature-Flags als Annotationen in den Graphen (die meisten Incidents folgen auf eine Änderung) und löschen Sie Dashboards, die seit Monaten niemand geöffnet hat. Während eines Incident ist die erste nützliche Ansicht meist der nutzerseitige Pfad, die zweite der Trace einer fehlschlagenden Anfrage und erst dann die Ressourcenmetriken.

ACHTUNG

Häufige Fehler bei der Observability

- Tracing nur in einigen Services einführen: Ein Hop ohne Weitergabe zerbricht jeden Trace, der durch ihn läuft.
- Nutzer-IDs, Bestell-IDs oder rohe URLs in Metrik-Labels packen.
- Perzentile über Instanzen oder Zeitfenster mitteln.
- Nur auf dem Server messen, sodass Ausfälle bei DNS, CDN oder Load Balancer unsichtbar bleiben.
- Personenbezogene Daten und Secrets loggen und so den Logspeicher zur am schlechtesten geschützten Kopie der Kundendaten machen.
- Auf jede Ursache alarmieren, sodass echte Symptome im Rauschen untergehen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum muss das p99 für eine ganze Flotte aus zusammengeführten Histogrammen berechnet werden statt aus dem p99 jeder Instanz?

Weil ein Perzentil kombinierter Daten keine Funktion der Perzentile der Teile ist; Bucket-Zählungen addieren sich exakt, sodass das zusammengeführte Histogramm eine echte (durch die Buckets begrenzte) Schätzung liefert, während gemittelte p99-Werte stark falsch sein können.

▼ Was geht kaputt, wenn ein Service in einer Aufrufkette den traceparent-Header nicht weitergibt?

Jeder Trace, der durch ihn läuft, wird geteilt: Spans nach diesem Hop beginnen einen neuen Trace, sodass das Gesamtbild von Ende zu Ende und die Verbindung zwischen Ursache und Symptom verloren gehen.

▼ **Warum ist order_id ein schlechtes Metrik-Label, aber ein gutes Span-Attribut?**

Als Label erzeugt es eine neue Zeitreihe pro Bestellung und lässt Kardinalität und Kosten explodieren; als Span- oder Log-Attribut wird es einmal pro Ereignis gespeichert, wo hohe Kardinalität erwartet wird und für Abfragen nützlich ist.

▼ **Was davon sollte jemanden alarmieren: CPU bei 95 % auf einer Instanz oder ein Checkout-SLO, das mit Rate 20 brennt?**

Das brennende SLO, weil es ein Symptom ist, das Nutzer spüren; der CPU-Wert ist eine mögliche Ursache, die auf ein Diagnose-Dashboard gehört.

QUELLEN

- Google SRE book: Monitoring Distributed Systems: https://sre.google/sre-book/monitoring-distributed-systems/
- Sigelman et al.: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure (Google, 2010): https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/
- W3C Trace Context: https://www.w3.org/TR/trace-context/
- OpenTelemetry documentation: Signals: https://opentelemetry.io/docs/concepts/signals/
- Prometheus documentation: Histograms and summaries: https://prometheus.io/docs/practices/histograms/

ABSCHNITT 9.7 VON 9.8 · FORTGESCHRITTEN · 25 MIN. LESEZEIT

Kapazitätsplanung und Performancetests

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.4 Performance und Skalierbarkeit
- 9.1 Skalierung
- 9.3 Queues und Backpressure

Begriffe von früher, die er verwendet: Kapazitätsreserve (Headroom), Littles Gesetz, Perzentil (p95, p99), Autoscaling, Fitness Function

DIE IDEE IN KÜRZE

Vor einer großen Hochzeit schätzt ein Caterer, wie viele Gäste kommen werden, rechnet aus, wie viele Teller jeder Koch in einer Stunde anrichten kann, plant ein paar zusätzliche Kräfte ein, falls jemand krank wird, und bestellt die zusätzlichen Öfen Wochen im Voraus, weil man sie am Tag selbst nicht kaufen kann. Dann hält er eine Probe in voller Größe ab, um zu prüfen, ob die Küche es wirklich schafft. Softwareteams machen es genauso: Sie prognostizieren die Nachfrage, messen, was jeder Server bewältigen kann, planen Reservekapazität ein, bestellen, was Zeit braucht, und proben mit realistisch simulierten Kunden.

WAS SIE WISSEN MÜSSEN

- Die Kapazitätsplanung prognostiziert die Spitzennachfrage, misst die Kapazität pro Einheit unter realistischen Bedingungen, plant Reserven und Redundanz ein und bestellt alles mit langer Vorlaufzeit, einschließlich Kontingenten und Limits von Drittanbietern.
- Last-, Stress-, Soak-, Spike- und Breakpoint-Tests beantworten unterschiedliche Fragen: ob die Ziele bei Spitzenlast erreicht werden, wie das System darüber hinaus ausfällt, ob es mit der Zeit nachlässt, wie es auf plötzliche Lastsprünge reagiert und wo die tatsächliche Grenze liegt.
- Geschlossene Lastmodelle werden langsamer, wenn das System langsamer wird, während bei offenen Modellen weiterhin Anfragen mit fester Rate eintreffen; öffentlicher Traffic und Vorverkaufsstarts sind offen, daher können Tests mit geschlossenem Modell gefährlich optimistisch sein (Schroeder, Wierman und Harchol-Balter, 2006).
- Coordinated Omission (Gil Tene) verbirgt langsame Antworten, wenn ein Testwerkzeug vor dem Senden auf Antworten wartet; messen Sie ab dem geplanten Sendezeitpunkt, mit Lastgeneratoren nach dem offenen Modell und vollständigen Histogrammen.

- k6, Gatling, JMeter und Locust sind die wichtigsten Open-Source-Werkzeuge; realistische Szenarien, Datenmengen, Hot Keys, die Latenz von Abhängigkeiten und ein nicht ausgelasteter Generator sind wichtiger als das Werkzeug.
- Performance-Budgets, etwa die Schwellenwerte der Core Web Vitals oder eine Latenzzuteilung pro Hop, werden in der Delivery-Pipeline automatisch als Fitness Functions geprüft.
- Die knappste gemeinsam genutzte Ressource, oft ein Drittanbieter, bestimmt die sichere Einlassrate für ein Ereignis wie einen Ticket-Vorverkaufsstart.

Kapazität ist ein Plan, keine Hoffnung

Die Kapazitätsplanung beantwortet eine Frage im Voraus: Werden wir von allem genug haben, wenn die Nachfrage ihren Höhepunkt erreicht? „Alles“ umfasst mehr als Server. Dazu gehören Datenbankverbindungen und Schreibdurchsatz, Cache-Speicher, Partitionen des Message Brokers, Kontingente des Cloud-Kontos, Rate Limits von Drittanbietern, Lizenzen und die Personen in Rufbereitschaft. Die Elastizität der Cloud macht manche Kapazität schnell verfügbar, aber nicht jede: Eine primäre Datenbank lässt sich nicht in Sekunden verdoppeln, ein Zahlungsanbieter braucht unter Umständen Wochen Vorlauf, um ein Limit anzuheben, und Autoscaling (behandelt in Abschnitt 9.1) reagiert in Minuten auf einen Lastsprung, der in Sekunden eintrifft.

Die Methode hat vier Schritte, die in einem regelmäßigen Zyklus wiederholt werden (vierteljährlich bei stetigem Wachstum, pro Ereignis bei Produktstarts und Vorverkaufsstarts):

Der Zyklus der Kapazitätsplanung

1. **Nachfrage prognostizieren**
   Kombinieren Sie organisches Wachstum (Bestellungen pro Tag steigen um 4 % im Monat), Saisonalität (Mittags- und Abendspitzen, Freitagabende, Weihnachten) und bekannte Ereignisse (eine Marketingkampagne, ein Vorverkaufsstart für ein Stadion). Drücken Sie die Nachfrage in den Einheiten aus, nach denen das System skaliert: Anfragen pro Sekunde pro Journey, Schreibvorgänge pro Sekunde, Nachrichten pro Sekunde, und zwar zur Spitze, nicht im Tagesdurchschnitt.

2. **Kapazität pro Einheit messen**
   Ermitteln Sie durch Tests, wie viel Last eine Instanz, eine Datenbank oder eine Partition aufnehmen kann, während sie ihr Latenz-SLO noch einhält. Der Knick der Antwortzeitkurve, ab dem die Warteschlangenbildung zu dominieren beginnt (Abschnitt 2.4), ist die praktische Grenze, nicht der Punkt, an dem das System abstürzt.

3 **Reserven und Redundanz einplanen**
Dimensionieren Sie für die prognostizierte Spitze plus eine Marge für Prognosefehler, Lastspitzen und Wiederholungen, und so, dass beim Verlust einer Instanz oder einer ganzen Availability Zone noch genug Kapazität bleibt (N+1 oder Redundanz auf Zonenebene).

4 **Vorlaufzeiten planen und überprüfen**
Listen Sie auf, was Zeit braucht (Erhöhungen von Kontingenten, reservierte Kapazität, Datenbank-Upgrades, Limits von Drittanbietern, Hardware für On-Premises-Systeme), und bestellen Sie es rechtzeitig. Vergleichen Sie anschließend die tatsächliche Nachfrage mit der Prognose und verbessern Sie das Modell.

Die Dimensionierung mit Littles Gesetz wurde in Abschnitt 9.1 durchgerechnet; dieser Abschnitt konzentriert sich darauf, wie man vertrauenswürdige Zahlen dafür gewinnt. Der häufigste Fehler in der Kapazitätsplanung ist nicht falsche Arithmetik, sondern eine Kapazität pro Instanz, die unter unrealistischen Bedingungen gemessen wurde, und der zweithäufigste ist eine vergessene gemeinsame Beschränkung, etwa ein Limit eines Drittanbieters, die keine noch so große Kapazitätsreserve auf den eigenen Servern beheben kann.

Arten von Performancetests

Performancetests sind der einzige Weg, Kapazität zu messen, Engpässe zu finden, bevor Kunden sie finden, und zu prüfen, dass ein Release nichts verschlechtert hat. Die Tests unterscheiden sich in der Lastform, die sie anlegen, und in der Frage, die sie beantworten. Ein Lasttest prüft die erwartete Spitze; ein Stresstest geht darüber hinaus, um zu sehen, wie das System ausfällt; ein Soak-Test läuft über Stunden, um schleichende Lecks aufzudecken; ein Spike-Test prüft die Reaktion auf plötzliche Sprünge; ein Breakpoint-Test steigert die Last, bis etwas bricht, um die tatsächliche Grenze zu finden.

Arten von Performancetests

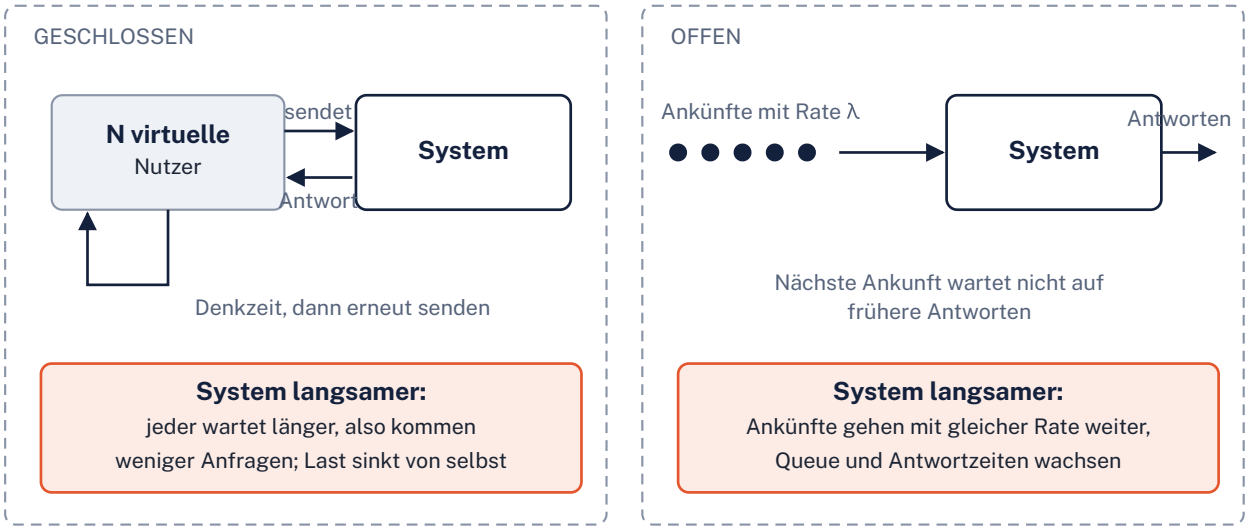
Test	Beantwortete Frage	Lastform	Typische Dauer	Achten auf
Smoke	Funktionieren Testskript und Umgebung überhaupt?	Wenige Benutzer	Minuten	Skriptfehler, falsche Umgebung
Last	Erreichen wir unsere Latenz- und Fehlerziele bei der erwarteten Spitze?	Anstieg auf die prognostizierte Spitze, halten	30–60 Minuten	p99-Latenz, Fehlerrate, Ressourcensättigung
Stress	Wie fällt das System jenseits der Spitze aus, und erholt es sich?	Anstieg auf das 1,5- bis 3-Fache der Spitze, dann zurück	30–60 Minuten	Graceful Degradation, Load Shedding, Erholungszeit
Soak (Dauertest)	Verschlechtert sich etwas mit der Zeit?	Normale oder Spitzenlast, gehalten	Stunden bis Tage	Speicherlecks, wachsende Queues, Verbindungslecks, volllaufende Festplatten
Spike	Können wir einen plötzlichen Ansturm abfangen?	Sprung von niedrig auf sehr hoch in Sekunden	Minuten	Verzögerung des Autoscalings, Queue-Überlauf, Verhalten des Rate Limiters
Breakpoint (Kapazität)	Wo liegt die Grenze, und was ist der erste Engpass?	Stetiger Anstieg bis zum Ausfall	Bis es bricht	Der Knick in der Latenz, die erste gesättigte Ressource

Offene und geschlossene Lastmodelle

Lastgeneratoren simulieren Benutzer auf eine von zwei Arten, und die falsche Wahl liefert Ergebnisse, die nicht nur ungenau, sondern gefährlich optimistisch sind. In einem geschlossenen Lastmodell betreibt das Werkzeug eine feste Zahl virtueller Benutzer; jeder sendet eine Anfrage, wartet auf die Antwort, pausiert für eine Denkzeit und wiederholt das. Wird das System langsamer, sendet jeder virtuelle Benutzer weniger Anfragen, sodass die Last von selbst sinkt. In einem offenen Lastmodell treffen Anfragen mit einer festen Rate ein, unabhängig davon, wie schnell frühere beantwortet werden, so wie echte Besucher einer öffentlichen Website: Tausende Fans, die um 10:00 auf „Kaufen“ drücken, warten nicht aufeinander.

Bianca Schroeder, Adam Wierman und Mor Harchol-Balter zeigten („Open Versus Closed: A Cautionary Tale“, NSDI 2006), dass die beiden Modelle für dasselbe System und dieselbe durchschnittliche Last sehr unterschiedliche Antwortzeiten liefern können und dass Scheduling- oder Kapazitätsentscheidungen, die unter dem einen Modell optimiert wurden, unter dem anderen falsch sein können. Interne Systeme mit einer festen Population von Clients, etwa eine Callcenter-Anwendung mit 300 Mitarbeitenden, sind einigermaßen geschlossen. Öffentlicher Internet-Traffic, ereignisgesteuerte Consumer, die aus einer Queue gespeist werden, und Vorverkaufsstarts sind offen oder nahezu offen. Für solche Systeme ist ein Test nach dem offenen Modell der realistische; die Executors `constant-arrival-rate` und `ramping-arrival-rate` von k6 zum Beispiel setzen ihn um, während der Executor `constant-vus` geschlossen ist.

Geschlossen und offen im Vergleich



Ein Test mit geschlossenem Modell zieht sich höflich zurück, wenn das System kämpft; echter öffentlicher Traffic tut das nicht.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: ein Test, der sich höflich zurückzieht

Der schwierigste Gedanke dieses Abschnitts ist, dass die Art, wie ein Lastgenerator Anfragen sendet, Überlast verbergen kann. In einem geschlossenen Modell warten simulierte Benutzer auf jede Antwort, sodass ein kämpfendes System automatisch weniger Last erhält; echte Fans bei einem Vorverkaufsstart warten nicht aufeinander.

- Ein Team testet den Saalplan für den Vorverkaufsstart mit 300 virtuellen Benutzern. Jeder ruft den Plan ab, wartet auf die Antwort, pausiert 2,5 Sekunden und wiederholt das. Bei Antworten in 0,5 Sekunden dauert jeder Zyklus 3 Sekunden: 300 ÷ 3 = 100 Anfragen pro Sekunde.
- Nach der Hälfte der Zeit sorgt eine langsame Datenbankabfrage dafür, dass jede Antwort 5 Sekunden dauert. Jeder Zyklus dauert jetzt 7,5 Sekunden, also senden dieselben 300 Benutzer nur noch 300 ÷ 7,5 = 40 Anfragen pro Sekunde.
- Die Last hat sich mehr als halbiert, genau dann, wenn das System kämpft, eine Atempause, die echter Traffic nie gewährt, sodass der Bericht weit besser aussieht, als der Vorverkaufsstart verlaufen wird.
- Echte Fans folgen einem offenen Modell: 100 pro Sekunde treffen weiter ein, egal, was mit früheren Anfragen geschah. Bei je 5 Sekunden ergibt Littles Gesetz 100 × 5 = 500 laufende Anfragen, zehnmal so viele wie die normalen 50, und wenn das System keine 100 pro Sekunde abschließen kann, wächst die Queue immer weiter.
- Die Arrival-Rate-Executors von k6 bilden das nach: Sie starten neue Anfragen mit fester Rate, egal wie langsam frühere beantwortet werden, sofern der Test genügend virtuelle Benutzer zulässt.
- Geschlossene Tests passen zu einer wirklich festen Population, etwa einer Callcenter-Anwendung, die von 300 Mitarbeitenden genutzt wird, die während des Wartens nichts senden.
- **Die Lehre:** Wählen Sie das Modell passend dazu, wer den Traffic sendet. Testen Sie bei öffentlichen Websites, Queue-Consumern und Vorverkaufsstarts mit fester Ankunftsrate, sonst ist der Test genau dort am optimistischsten, wo es am meisten darauf ankommt.

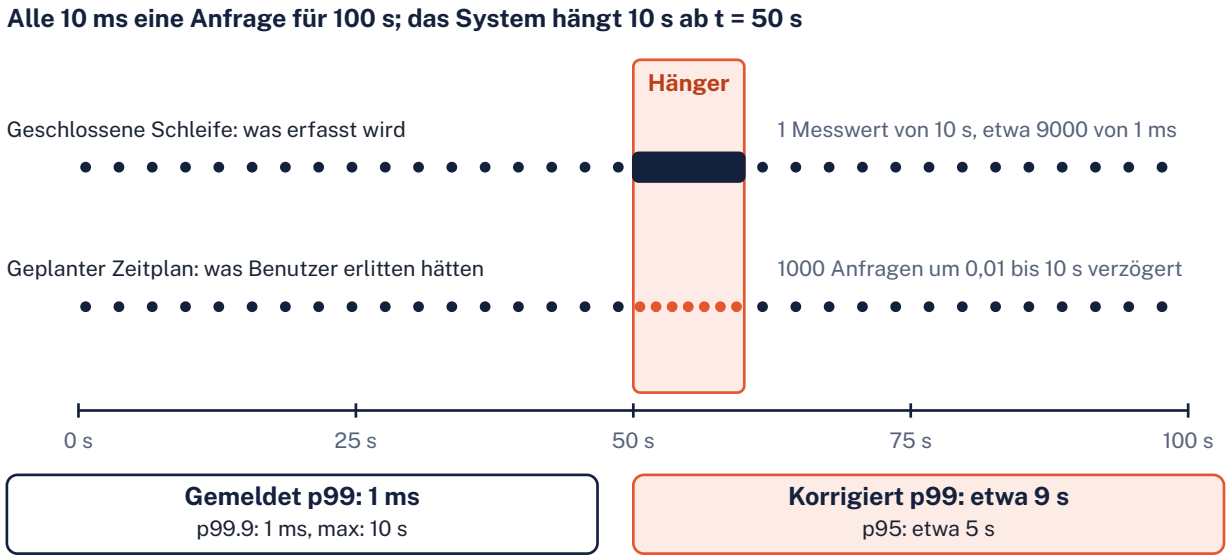
Coordinated Omission

Gil Tene von Azul Systems, Autor der Bibliothek HdrHistogram, gab einem Messfehler einen Namen, der viele Latenztests und Monitoring-Werkzeuge betrifft: Coordinated Omission. Ein Lastgenerator mit geschlossener Schleife wartet auf jede Antwort, bevor er die nächste Anfrage sendet. Wenn das System hängt, hängt der Generator mit, sodass die Anfragen, die

echte Benutzer während des Hängers gesendet hätten und die alle langsam gewesen wären, nie gesendet und nie gemessen werden. Das Testwerkzeug „koordiniert“ sich mit dem System, das es testet, und lässt genau die schlechten Messwerte aus.

Betrachten Sie einen Test, der 100 Sekunden lang alle 10 ms eine Anfrage an einen Dienst senden soll, der normalerweise in 1 ms antwortet. Bei 50 Sekunden friert der Dienst für 10 Sekunden ein (etwa durch eine lange Garbage-Collection-Pause). Das Testwerkzeug mit geschlossener Schleife zeichnet etwa 9000 schnelle Messwerte und einen 10-Sekunden-Messwert auf und meldet daher ein p99 von 1 ms und sogar ein p99.9 von 1 ms. Nach dem geplanten Zeitplan wären während des Einfrierens jedoch etwa 1000 Anfragen eingetroffen, die zwischen 10 Sekunden und 10 ms gewartet hätten, im Durchschnitt etwa 5 Sekunden. Zehn Prozent aller Anfragen wurden stark verzögert: Das wahre p95 liegt bei etwa 5 Sekunden und das wahre p99 bei etwa 9 Sekunden.

Was das Testwerkzeug übersieht



Der eine langsame Messwert des Testwerkzeugs verbirgt tausend langsame Benutzererfahrungen.

Die Abhilfe besteht darin, für offene Systeme Generatoren nach dem offenen Modell (mit Ankunftsrate) zu verwenden, die Latenz ab dem Zeitpunkt zu messen, zu dem eine Anfrage gesendet werden *sollte*, und nicht ab dem tatsächlichen Senden, vollständige Histogramme statt Durchschnittswerten aufzuzeichnen und jedem Latenzbericht zu misstrauen, dessen Maximum weit über seinem p99.9 liegt, mit wenigen Messwerten dazwischen. Tenes modifiziertes HTTP-Benchmarking-Werkzeug wrk2 und die Korrekturoptionen von HdrHistogram gibt es genau dafür. Derselbe Fehler tritt im Produktions-Monitoring auf, wenn eine Client-Bibliothek nur die Anfragen misst, die sie senden konnte.

Werkzeuge

Vier Open-Source-Werkzeuge decken die meisten Anforderungen ab. **Apache JMeter** ist der Veteran: Java-basiert, mit einem grafischen Editor für Testpläne, vielen Protokoll-Plug-ins und Thread-Gruppen, die standardmäßig geschlossen sind. **Gatling** schreibt Tests als Code (Scala, Java und Kotlin, in jüngerer Zeit auch JavaScript und TypeScript), ist bei hoher Last effizient und bietet offene und geschlossene Injektionsprofile. **k6**, gepflegt von Grafana Labs, führt JavaScript-Testskripte auf einer in Go geschriebenen Engine aus, hat explizite Executors für das offene und das geschlossene Modell sowie Bestanden-oder-nicht-Schwellenwerte, die sich für Continuous Integration eignen. **Locust** beschreibt das Benutzerverhalten in Python und verteilt die Last auf Worker-Prozesse; seine Benutzer arbeiten mit geschlossener Schleife. Kommerzielle und in der Cloud gehostete Dienste ergänzen eine verteilte Lasterzeugung in großem Maßstab und Reporting.

Das Szenario ist wichtiger als das Werkzeug. Ein gutes Skript bildet den echten Mix der Journeys nach (aus der Produktionsanalyse: vielleicht 70 % Stöbern, 20 % Änderungen am Warenkorb, 10 % Bezahlvorgang), realistische Denkzeiten, realistische Daten (verschiedene Benutzer, Restaurants und Plätze, nicht 10.000-mal derselbe) und das richtige Lastmodell, und es prüft die Antworten auf Korrektheit, denn eine schnelle Fehlerseite ist keine erfolgreiche Anfrage.

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  scenarios: {
    onsale: {
      executor: 'ramping-arrival-rate',   // open model: arrivals per second
      startRate: 100, timeUnit: '1s',
      preAllocatedVUs: 2000, maxVUs: 20000,
      stages: [
        { target: 4900, duration: '2m' },  // ramp to 1.5x design load
        { target: 4900, duration: '20m' }, // hold
      ],
    },
  },
  thresholds: {                           // fail the run if missed
    http_req_failed: ['rate<0.005'],
    http_req_duration: ['p(99)<500'],
  },
};

export default function () {
  const res = http.get('https://perf.tickets.example/api/events/42/seats');
  check(res, { 'status is 200': (r) => r.status === 200 });
}
```

Ein k6-Test nach dem offenen Modell für den Saalplan beim Vorverkaufsstart, der auf 4900 Anfragen pro Sekunde ansteigt und den Lauf scheitern lässt, wenn p99 über 500 ms liegt oder mehr als 0,5 % der Anfragen fehlschlagen.

Realistische Umgebungen und Daten

Ein Performancetest ist nur so vertrauenswürdig wie seine Umgebung. Eine Testdatenbank mit 10.000 Zeilen, deren Indizes in den Arbeitsspeicher passen, sagt nichts über die Produktion mit 200 Millionen Zeilen aus, bei denen das nicht der Fall ist. Die folgende Checkliste erfasst die üblichen Unterschiede.

Größe und Form der Daten: Datenmengen in Produktionsgröße und realistische Verteilungen, einschließlich Hot Keys wie der beliebtesten Restaurants oder der besten Plätze, die Konkurrenzsituationen erzeugen, die ein gleichmäßiger Test nie sieht. **Dieselben Bausteine:** dieselben Instanztypen, Container-Limits, Datenbankkonfiguration, Größen der Verbindungspools und Netzwerktopologie (einschließlich Hops zwischen Zonen). **Cache-Zustand:** Testen Sie sowohl warme als auch kalte Caches, denn ein Deployment oder Failover leert sie (Abschnitt 7.6). **Abhängigkeiten:** Dienste von Drittanbietern werden durch Stubs ersetzt, die deren echte Latenzverteilung und Rate Limits nachbilden, nicht durch sofortige

Antworten. **Isolation:** keine Tests anderer Teams in derselben Umgebung, sonst sind die Ergebnisse Rauschen. **Der Generator selbst:** genug Kapazität für die Lastgeneratoren an den richtigen Standorten; ein Generator, der mit 100 % CPU läuft, misst sich selbst.

Manche Organisationen testen vorsichtig in der Produktion, weil keine Kopie ihr je ganz gleicht: synthetische Benutzer und Testereignisse, die aus dem Geschäftsreporting ausgeschlossen werden, Traffic Shadowing (echte Anfragen werden an eine neue Version kopiert und deren Antworten verworfen) und schrittweise Steigerungen mit automatischen Abbruchbedingungen. Das erfordert dieselben Schutzmaßnahmen wie Chaos-Experimente: einen kleinen Blast Radius, einen Notausschalter und eine genaue Beobachtung der SLOs.

Performance-Budgets

Ein Performance-Budget ist eine Grenze, die eine Änderung nicht überschreiten darf, im Voraus vereinbart und automatisch geprüft, damit die Performance nicht Regression für Regression erodiert. Bei Web-Frontends decken Budgets das Seitengewicht, die Skriptgröße und die Core Web Vitals ab; Googles Schwellenwerte für eine gute Erfahrung, gemessen am 75. Perzentil der Seitenaufrufe, sind dort ein Largest Contentful Paint von höchstens 2,5 Sekunden, eine Interaction to Next Paint von höchstens 200 ms und ein Cumulative Layout Shift von höchstens 0,1.

Bei Backends teilt ein Latenzbudget ein Ende-zu-Ende-SLO auf die Hops einer Journey auf. Der Checkout der Essenslieferplattform hat ein p99-Ziel von 800 ms; das Team weist 20 ms dem Gateway zu, 60 ms dem Bestellservice, 80 ms der Preisgestaltung, 450 ms der Zahlung (das meiste davon für den externen Kartenanbieter), 40 ms der Datenbank, 100 ms Netzwerk und Serialisierung und behält 50 ms in Reserve. Jedes Team kennt nun sein eigenes Ziel, und eine Änderung, die die Preisgestaltung auf 150 ms treibt, wird als Budgetverletzung sichtbar, bevor sie zur SLO-Verletzung wird. Streng genommen addieren sich die Perzentile der Teile nicht zum Perzentil des Ganzen; behandeln Sie die Aufteilung daher als technische Zuteilung und prüfen Sie den Ende-zu-Ende-Wert gesondert.

Budgets werden durchgesetzt, indem in der Delivery-Pipeline ein kurzer, stabiler Performancetest läuft und der Build fehlschlägt, wenn ein Schwellenwert überschritten wird, wie es die k6-Schwellenwerte oben tun. Das ist eine Fitness Function für die Performance. Tests in der Pipeline müssen eher wiederholbar als realistisch sein: dieselbe Umgebung, dieselben Daten, dieselbe Last, damit eine Änderung um 10 % eine echte Regression bedeutet und kein Rauschen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein Kapazitätsplan für einen Ticket-Vorverkaufsstart

Dies ist ein anderes, größeres Ereignis als der Vorverkaufsstart in Abschnitt 9.3, mit eigenen Zahlen. Eine Stadiontour umfasst drei Abende mit je 60.000 Plätzen, insgesamt 180.000 Tickets, die an einem Freitag um 10:00 in den Verkauf gehen. Fans betreten einen virtuellen Warteraum und werden mit kontrollierter Rate in den Kaufprozess eingelassen. Lasttests haben den Knick jeder API-Instanz bei 520 Anfragen pro Sekunde gemessen; das Team plant mit 400, um p99 unter 500 ms zu halten.

Nachfrageprognose	520.000 Vorab-Registrierungen, von denen etwa 75 % voraussichtlich erscheinen: Planen Sie mit 400.000 Fans im Warteraum. Bei durchschnittlich 2,5 Tickets pro Bestellung ist die Tour mit 72.000 Bestellungen ausverkauft.
Bindende Beschränkung	Das vereinbarte Limit des Zahlungsanbieters liegt bei 120 Zahlungen pro Sekunde. Wenn 80 % der eingelassenen Fans kaufen und ein Drittel des Limits für Wiederholungen und Fehler zurückbehalten wird, lassen Sie 100 Fans pro Sekunde ein (6000 pro Minute): 80 Zahlungen pro Sekunde.
Aktive Fans (Littles Gesetz)	100 Fans/s × 240 s im Kaufprozess = 24.000 gleichzeitig aktive Fans
API-Last	100 Fans/s × 25 Anfragen pro Journey = 2500 Anfragen/s; × 1,3 für gehäufte Saalplan-Aufrufe = 3250 Anfragen/s Auslegungslast
Instanzen	3250 / 400 = 8,1, also 9 Instanzen; um den Verlust einer von drei Zonen zu überstehen, müssen 9 in zwei Zonen passen: 5 pro Zone × 3 = 15 Instanzen, gestartet vor 09:30
Datenbank für Platzreservierungen	1,5 Reservierungsversuche pro Fan = 150 Schreibvorgänge/s bei gemessenen 600 Schreibvorgängen/s mit p99 unter 50 ms: 25 % Auslastung, aber testen Sie die Konkurrenz um die beliebtesten Plätze
Warteraum	400.000 Fans, die alle 20 s abfragen = 20.000 Anfragen/s, ausgeliefert am CDN-Edge, ohne die API-Schicht je zu erreichen
Ergebnis	72.000 Bestellungen / 80 pro Sekunde = 900 s: ausverkauft in etwa 15 Minuten, nachdem etwa 90.000 Fans eingelassen wurden; der Warteraum muss alle anderen umgehend informieren
Vorlaufzeiten	Zahlungslimit 3 Wochen vorher bestätigt; Cloud-Kontingent 1 Woche vorher erhöht; Lasttest mit dem 1,5-Fachen (etwa 4900 Anfragen/s), Spike- und 2-stündiger Soak-Test 2 Wochen vorher in einer Umgebung in Produktionsgröße

Die Einlassrate wird von der knappsten gemeinsam genutzten Ressource bestimmt, hier dem Zahlungsanbieter, nicht davon, wie viele Server sich das Team leisten kann; doppelt so viele Server würden die Tickets nicht schneller verkaufen.

Autoscaling ist hier ein Sicherheitsnetz, nicht der Plan: Ein Sprung von null auf 3250 Anfragen pro Sekunde um 10:00:00 kommt schneller, als Instanzen starten können.

ACHTUNG

Fallstricke, die Performancetests lügen lassen

- Generatoren nach dem geschlossenen Modell für offenen Traffic und Latenz, die ab dem tatsächlichen statt ab dem geplanten Sendezeitpunkt gemessen wird (Coordinated Omission).
- Durchschnittswerte statt Perzentilen und Histogrammen berichten.
- Winzige, gleichförmige Testdaten, die in den Arbeitsspeicher passen und keine Hot Keys haben.
- Stubs, die sofort antworten, wo die echte Abhängigkeit 400 ms braucht und ein Rate Limit hat.
- Schnelle Fehlerantworten als erfolgreiche Anfragen zählen.
- Ein Lastgenerator, der selbst der Engpass ist.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum unterschätzt ein Lasttest nach dem geschlossenen Modell bei einer öffentlichen Website tendenziell die Antwortzeiten?

Weil seine virtuellen Benutzer auf jede Antwort warten, bevor sie mehr senden, sodass die angebotene Last sinkt, wenn das System langsamer wird, während echte Besucher weiter eintreffen und die Queues wachsen.

▼ Ein Testwerkzeug zeigt ein p99 von 2 ms und ein Maximum von 8 s, mit fast keinen Messwerten dazwischen. Was sollten Sie vermuten?

Coordinated Omission: Das Testwerkzeug hing wahrscheinlich mit dem System und hat die vielen Anfragen nicht erfasst, die während der Pause verzögert worden wären.

▼ Warum ermöglichen im Vorverkaufsbeispiel zusätzliche API-Instanzen keinen schnelleren Ausverkauf?

Weil das Limit des Zahlungsanbieters von 120 Zahlungen pro Sekunde, nicht die API-Schicht, begrenzt, wie viele Fans pro Sekunde kaufen können.

▼ Wofür ist ein Soak-Test gedacht?

Um Probleme zu finden, die erst mit der Zeit unter gleichbleibender Last auftreten, etwa Speicher- oder Verbindungslecks, wachsende Queues und volllaufende Festplatten.

QUELLEN

- Schroeder, Wierman and Harchol-Balter: Open Versus Closed: A Cautionary Tale (NSDI 2006): https://www.usenix.org/conference/nsdi-06/open-versus-closed-cautionary-tale
- Grafana k6 documentation: Open and closed models: https://grafana.com/docs/k6/latest/using-k6/scenarios/concepts/open-vs-closed/
- web.dev: Web Vitals: https://web.dev/articles/vitals
- Google SRE book: Handling Overload: https://sre.google/sre-book/handling-overload/

ABSCHNITT 9.8 VON 9.8 · MITTELSTUFE · 25 MIN. LESEZEIT

Chaos Engineering, Incidents und schuldfreies Lernen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.5 Verfügbarkeit und Zuverlässigkeit
- 9.4 Resilienzmuster
- 9.5 SLOs und Error Budgets

Begriffe von früher, die er verwendet: Mean Time to Repair (MTTR), Error Budget, Service Level Objective (SLO), Runbook, Failover

DIE IDEE IN KÜRZE

Die Feuerwehr wartet nicht auf einen echten Brand, um herauszufinden, ob die Schläuche funktionieren. Sie hält Übungen ab, legt kleine kontrollierte Feuer, um zu üben, und setzt sich nach jedem echten Brand zusammen, um herauszuarbeiten, was passiert ist, ohne nach einem Schuldigen zu suchen, damit der nächste Einsatz besser läuft. Softwareteams machen es genauso. Sie legen absichtlich kleine Teile ihrer Systeme lahm, um zu prüfen, ob die Sicherheitsnetze funktionieren, folgen einem klaren Plan mit benannten Rollen, wenn wirklich etwas schiefgeht, und untersuchen danach jeden Incident ehrlich, damit das System besser wird und nicht nur die Menschen.

WAS SIE WISSEN MÜSSEN

- Chaos Engineering, das aus Chaos Monkey von Netflix hervorging (2011 beschrieben), experimentiert mit einem System, um Vertrauen aufzubauen, dass es turbulenten Bedingungen standhält; es ist ein kontrollierter Test einer Hypothese, kein willkürliches Kaputtmachen.
- Die Principles of Chaos Engineering sind: eine Hypothese über das Verhalten im Steady State, reale Ereignisse, Experimente in der Produktion, kontinuierliche Automatisierung und ein minimaler Blast Radius mit Abbruchbedingungen.
- Game Days proben erhebliche Ausfälle mit dem ganzen Team und testen neben der Software auch Menschen, Zugänge, Runbooks, Backups und Kommunikation.
- Das Incident-Management trennt Rollen: einen Incident Commander, der koordiniert statt zu debuggen, Responder aus dem Betrieb, eine Kommunikationsleitung, die für die Statusseite zuständig ist, und einen Protokollführer; Schweregrade werden nach den Auswirkungen auf Benutzer definiert, und Eindämmung kommt vor Diagnose.

- Schuldfreie Post-Incident-Reviews (Google-SRE-Buch; John Allspaw, 2012) suchen nach zweiten Geschichten und mehreren beitragenden Faktoren statt nach einer einzigen Grundursache oder menschlichem Versagen.
- MTTR über viele Incidents hinweg ist eine irreführende Kennzahl, weil die Dauern stark schief verteilt sind; SLO-Auswirkungen, wiederkehrende Themen und abgeschlossene Systemänderungen sind bessere Anhaltspunkte.
- Starke Maßnahmen beseitigen Gefahren oder schaffen Leitplanken, haben Verantwortliche und Fälligkeitstermine und werden über die Error-Budget-Policy priorisiert; Ermahnungen zur Vorsicht ändern nichts.

Ausfälle absichtlich üben

Alles in diesem Kapitel war bisher ein Versprechen, wie sich das System verhält, wenn etwas schiefgeht: Timeouts werden auslösen, Circuit Breaker werden öffnen, Last wird abgeworfen, Replikate werden übernehmen, Alerts werden jemanden erreichen, der weiß, was zu tun ist. Kapitel 2 hat gewarnt, dass ein Failover, das nie geübt wurde, oft genau dann versagt, wenn es endlich gebraucht wird (Abschnitt 2.5). In diesem Abschnitt geht es darum, diese Versprechen in Belege zu verwandeln, und darum, wie Organisationen auf die Ausfälle reagieren, die trotzdem passieren, und aus ihnen lernen.

Zwei Praktiken gehören zusammen. Vor Ausfällen verursachen *Chaos Engineering* und *Game Days* absichtlich kleine, kontrollierte Ausfälle, um Schwachstellen zu finden und Menschen zu trainieren. Nach Ausfällen begrenzt das *Incident-Management* den Schaden, und *schuldfreie Post-Incident-Reviews* machen aus jedem Incident eine dauerhafte Verbesserung. Beide beruhen auf derselben Überzeugung: Komplexe Systeme sind immer teilweise kaputt, und Resilienz entsteht dadurch, früh und gefahrlos herauszufinden, wie.

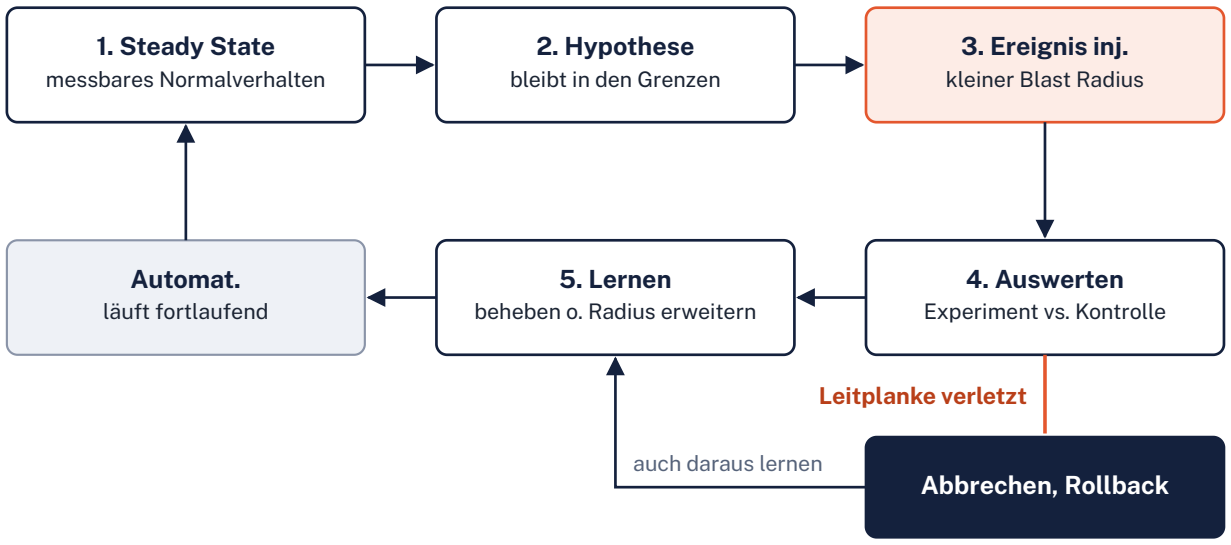
Chaos Engineering

Chaos Engineering entstand aus der Migration von Netflix aus eigenen Rechenzentren zu Amazon Web Services, die nach einer schweren Datenbankbeschädigung im Jahr 2008 begann. Einzelne Cloud-Instanzen konnten jederzeit verschwinden, also baute Netflix Chaos Monkey, ein Werkzeug, das während der Geschäftszeiten zufällig Produktionsinstanzen beendete, wenn die Engineers an ihren Schreibtischen saßen, um daraus entstehende Probleme zu bemerken und zu beheben. Netflix beschrieb es zusammen mit einer Familie verwandter Werkzeuge namens Simian Army in einem Blogbeitrag im Juli 2011. Spätere Werkzeuge gingen weiter: Chaos Kong simulierte den Verlust einer ganzen AWS-Region, indem der Traffic daraus abgezogen wurde.

Engineers von Netflix und andere hielten die Disziplin später als *Principles of Chaos Engineering* fest (principlesofchaos.org, zuletzt 2019 überarbeitet), die sie definieren als das Experimentieren mit einem System, um Vertrauen in seine Fähigkeit aufzubauen, turbulenten Bedingungen in der Produktion standzuhalten. Das entscheidende Wort ist *Experimentieren*: Chaos Engineering heißt nicht, Dinge willkürlich kaputt zu machen, um zu sehen, was passiert, sondern die wissenschaftliche Methode auf Resilienz anzuwenden. Die Prinzipien nennen fünf Praktiken:

Eine Hypothese über das Verhalten im Steady State aufstellen. Wählen Sie eine messbare Ausgabe, die normalen Betrieb anzeigt, etwa aufgegebene Bestellungen pro Minute oder den Checkout-SLI, statt interner Metriken wie CPU. Die Steady-State-Hypothese sagt voraus, dass diese Ausgabe innerhalb ihrer Grenzen bleibt, während der Fehler injiziert wird. **Reale Ereignisse variieren.** Injizieren Sie die Fehler, die tatsächlich vorkommen: sterbende Instanzen, eine langsamer werdende Abhängigkeit, Paketverlust, eine volle Festplatte, ein abgelaufenes Zertifikat, Uhrenabweichung, eine Traffic-Spitze. **Experimente in der Produktion durchführen.** Nur die Produktion hat den echten Traffic, die echten Daten und die echte Konfiguration. **Experimente automatisieren, damit sie kontinuierlich laufen.** Ein einmaliges Experiment beweist Resilienz einmal; Systeme ändern sich täglich. **Den Blast Radius minimieren.** Beginnen Sie mit dem kleinsten Experiment, aus dem sich etwas lernen lässt (eine Instanz, ein kleiner Prozentsatz des Traffics), und halten Sie einen Abbruchschalter bereit.

Der Kreislauf eines Chaos-Experiments



Ein Chaos-Experiment ist ein kontrollierter Test einer Hypothese, mit Leitplanken, die es stoppen, bevor Benutzer ernsthaft Schaden nehmen.

Experimente in der Produktion durchzuführen ist das umstrittenste Prinzip. Seine Befürworter argumentieren, dass sich Staging-Umgebungen in Traffic, Daten, Größe und Konfiguration unterscheiden, sodass nur Experimente in der Produktion echte Schwachstellen aufdecken, und dass ein kleines, kontrolliertes Experiment um 15:00 für Benutzer weit schonender ist als derselbe Ausfall, der unkontrolliert um 03:00 eintritt. Kritiker weisen darauf hin, dass vielen Organisationen die Observability, die Automatisierung und das organisatorische Vertrauen fehlen, um das sicher zu tun. Der praktische Konsens ist ein schrittweises Vorgehen: in einer Staging-Umgebung beginnen, dann in der Produktion mit winzigem Blast Radius und automatischen, an SLOs geknüpften Abbruchbedingungen laufen lassen und nur mit wachsendem Vertrauen erweitern. Beginnen Sie nie mit einem System, von dem Sie bereits wissen, dass es fragil ist; beheben Sie zuerst die bekannten Schwachstellen, denn ein Experiment, dessen Ergebnis Sie vorhersagen können, lehrt nichts.

Werkzeuge sind inzwischen breit verfügbar: Cloud-Anbieter bieten verwaltete Fault Injection an (AWS Fault Injection Service, Azure Chaos Studio), Chaos Mesh und LitmusChaos sind Open-Source-Projekte der CNCF für Kubernetes, und kommerzielle Plattformen wie Gremlin bieten Bibliotheken von Angriffen mit Sicherheitskontrollen. Service Meshes und Proxys können Latenz und Fehler in einen Prozentsatz der Aufrufe injizieren, ohne den Anwendungscode anzufassen. Die Werkzeuge sind der leichte Teil; der schwere Teil ist, aussagekräftige Hypothesen zu wählen und auf die Ergebnisse zu reagieren.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein Latenzexperiment am Bestellservice

Der Bestellservice der Essenslieferplattform ruft den Speisekartenservice auf, um Artikelpreise zu bestätigen. Laut Entwurf (nach Abschnitt 9.4) hat der Aufruf ein Timeout von 300 ms und fällt auf zwischengespeicherte Preise zurück. Jede Instanz des Bestellservice hat 50 Worker-Threads und erhält etwa 40 Bestellungen pro Sekunde, von denen jede normalerweise einen Thread 0,2 s lang belegt.

Steady State	Erfolgsquote der Bestellungen mindestens 99,9 %; Checkout-p99 unter 800 ms; Bestellungen pro Minute innerhalb von 5 % derselben Minute der Vorwoche
Hypothese	Wenn Aufrufe des Speisekartenservice von einer Instanz des Bestellservice zusätzlich 2000 ms dauern, bleiben die Steady-State-Metriken dieser Instanz innerhalb der Grenzen, weil Timeout und Fallback die Verzögerung abfangen
Blast Radius	1 von 20 Instanzen (5 % des Bestell-Traffics) in einer Region, 15:00 an einem Werktag, 15 Minuten, die anderen 19 Instanzen als Kontrollgruppe
Abbruchbedingungen	Erfolgsquote der Experimentinstanz 2 Minuten lang unter 99 % oder Burn Rate des regionalen Checkout-SLO über 14,4: automatisch stoppen
Erwartete Thread-Nutzung	Mit dem Timeout von 300 ms: 40/s × (0,2 + 0,3) s = 20 belegte Threads von 50
Was geschah	Die Erfolgsquote fiel innerhalb von 3 Minuten auf 97 %, und der Abbruch griff. Traces zeigten Aufrufe, die 2,0 s warteten: Das Timeout betrug in Wirklichkeit 3000 ms, ein Bibliotheks-Standardwert, der die beabsichtigte Einstellung überschrieb
Warum es scheiterte	40/s × (0,2 + 2,0) s = 88 benötigte Threads, aber es gibt nur 50: Anfragen stauten sich und liefen am Gateway in ein Timeout
Maßnahmen	Das Timeout in der gemeinsamen Client-Konfiguration setzen, einen automatisierten Test ergänzen, der den Build scheitern lässt, wenn einem ausgehenden Aufruf ein explizites Timeout fehlt, das Experiment wiederholen und dann auf eine ganze Zone erweitern

Das Experiment fand bei 5 % des Traffics einer Region und in 3 Minuten eine Schwachstelle, die eine echte Verlangsamung des Speisekartenservice zur Mittagszeit zu einem regionalen Ausfall gemacht hätte.

Der Entwurf war richtig; die Konfiguration nicht. Experimente testen, was deployt ist, nicht, was dokumentiert ist.

Game Days

Ein Game Day ist eine geplante Übung, bei der ein Team absichtlich einen erheblichen Ausfall verursacht oder simuliert und die Reaktion darauf übt. Jesse Robbins machte die Praxis bei Amazon bekannt, wo seine Berufsbezeichnung Berichten zufolge Master of Disaster lautete. Google führt große Übungen namens Disaster Recovery Testing (DiRT) durch, und sein SRE-Buch beschreibt das Wheel of Misfortune, ein Rollenspiel, in dem Engineers in Rufbereitschaft einen vergangenen Incident durchspielen, während ein Spielleiter enthüllt, was das Monitoring gezeigt hätte.

Game Days testen mehr als Software. Sie testen, ob Alerts die richtigen Personen erreichen, ob die Menschen um 03:00 die nötigen Zugänge haben, ob Runbooks aktuell sind, ob das Failover-Verfahren tatsächlich funktioniert, ob sich Backups wiederherstellen lassen und ob der Kommunikationsplan die Kunden erreicht. Ein typischer Game Day bei der Essenslieferplattform könnte an einem ruhigen Nachmittag die Bestelldatenbank auf ihren Standby umschalten, während die gesamte Rufbereitschaftsrotation zusieht, ein Moderator Überraschungen einstreut („die Person, die das Runbook kennt, ist im Urlaub“) und ein Beobachter jede Verzögerung und Verwirrung für das anschließende Review notiert. Vor einem großen Ticket-Vorverkaufsstart könnte dasselbe Team proben, wie der Zahlungsanbieter ausfällt und mitten im Verkauf auf den Ersatzanbieter umgeschaltet wird.

Incident-Management

Ein Incident ist ein ungeplantes Ereignis, das einen Dienst stört oder zu stören droht und eine koordinierte Reaktion erfordert. Incident-Management ist die Praxis, diese Reaktion so zu führen, dass sie schnell, ruhig und gut kommuniziert ist. Die meiste moderne Praxis lehnt sich an das Incident Command System an, das in den 1970er-Jahren in Kalifornien für die Bekämpfung von Waldbränden entwickelt und im Kapitel des Google-SRE-Buchs über das Management von Incidents für Software angepasst wurde.

Der zentrale Gedanke ist die Trennung der Rollen. Der Incident Commander ist für den Incident verantwortlich: Er setzt Prioritäten, vergibt Rollen, trifft Entscheidungen und behält den Überblick, und er debuggt bewusst *nicht*, denn ein Commander, der in einer Logdatei versunken ist, führt nicht. Die Betriebsleitung und die Responder diagnostizieren und dämmen ein. Die Kommunikationsleitung ist die einzige Stimme gegenüber allen außerhalb der Reaktion: Support-Teams, Führungskräften und Kunden über die Statusseite, mit Updates im zugesagten Rhythmus, auch wenn es nichts Neues gibt. Ein Protokollführer hält Zeitablauf, Entscheidungen und Hypothesen fest. In einem kleinen Unternehmen kann eine Person mehrere Rollen innehaben, aber die Rollen sollten trotzdem benannt werden. Bei Schichtwechseln werden Rollen ausdrücklich übergeben.

Rollen im Incident

Ein Incident-Kanal; der Commander debuggt nicht

Die Trennung von Koordination, technischer Arbeit und Kommunikation hält jede Rolle fokussiert.

Einige Gewohnheiten machen den Unterschied. Rufen Sie Incidents früh und ohne großen Aufwand aus: Es ist besser, einen Fehlalarm abzublasen, als nach einer Stunde festzustellen, dass drei Teams getrennt voneinander dasselbe Problem debuggt haben. Ordnen Sie jedem Incident einen Schweregrad zu, der nach den Auswirkungen auf Benutzer definiert ist, nicht nach der Ursache, damit alle wissen, wen sie rufen und wie breit sie kommunizieren müssen. Dämmen Sie ein, bevor Sie diagnostizieren: Rollen Sie zuerst das letzte Deployment zurück, schalten Sie per Failover um, schalten Sie das Feature-Flag ab oder werfen Sie Last ab, und finden Sie das Warum später heraus. Nutzen Sie einen einzigen Incident-Kanal. Und denken Sie an externe Pflichten: Eine Verletzung des Schutzes personenbezogener Daten muss nach der DSGVO unter Umständen binnen 72 Stunden der Aufsichtsbehörde gemeldet werden, weshalb Sicherheitsvorfälle dem zugehörigen Prozess in Abschnitt 8.8 folgen.

Beispielhafte Schweregrade für die Essenslieferplattform

Niveau	Beispielhafte Auswirkung	Reaktion	Kommunikation
SEV1 (kritisch)	Bestellen oder Bezahlen für die meisten Benutzer einer Region nicht verfügbar; Verdacht auf Datenschutzverletzung	Sofort alarmieren; Incident Commander und Kommunikationsleitung umgehend benannt	Statusseite innerhalb von 15 Minuten; Updates alle 30 Minuten; Führungskräfte informiert
SEV2 (schwer)	Eine zentrale Journey für viele Benutzer stark beeinträchtigt, etwa Checkout-p99 über 4 s in einer Stadt; schnelles Verbrennen des SLO-Budgets	Die Bereitschaft alarmieren; Commander benannt	Statusseite, falls Kunden es bemerken können; interne Updates stündlich
SEV3 (gering)	Teilweise Beeinträchtigung mit Workaround; ein internes Werkzeug ausgefallen	Während der Arbeitszeit vom zuständigen Team bearbeitet	Team-Kanal; Ticket
SEV4 (niedrig)	Kosmetisches Problem oder Beinahe-Vorfall ohne Auswirkungen auf Benutzer	Backlog-Eintrag	Optionales Review, wenn es etwas lehrt

Schuldfreie Post-Incident-Reviews

Nach dem Incident kommt der wertvollste Teil: das Lernen. Ein Post-Incident-Review (oft Postmortem genannt) rekonstruiert, was passiert ist, warum es den Beteiligten damals sinnvoll erschien und was sich ändern wird. Das Kapitel des Google-SRE-Buchs über Postmortem-Kultur legt die Erwartungen fest: eine schriftliche Aufzeichnung für jeden erheblichen Incident, die Auswirkungen, Zeitablauf, beitragende Ursachen und Folgemaßnahmen abdeckt, geprüft und geteilt wird und ohne Schuldzuweisung verfasst ist.

John Allspaws Beitrag „Blameless PostMortems and a Just Culture“ aus dem Jahr 2012, geschrieben, als er bei Etsy den technischen Betrieb leitete, erklärt, warum Blameless Culture (Schuldfreiheit) praktisch und nicht nur freundlich ist. Engineers, die mit Bestrafung rechnen, hören auf, vollständig zu berichten, was sie gesehen und getan haben, sodass die Organisation genau die Informationen verliert, die sie braucht, um den nächsten Incident zu verhindern. Allspaw stützt sich auf die Sicherheitswissenschaft, darunter Sidney Dekkers Arbeiten zur Just Culture, und auf die Idee der *zweiten Geschichten*: Die erste Geschichte sagt „ein Engineer hat den falschen Befehl ausgeführt“; die zweite fragt, warum der Befehl so leicht auszuführen war, warum das Werkzeug nicht gewarnt hat, warum das Vorgehen unter Zeitdruck vernünftig erschien, und findet Bedingungen, die auch die nächste Person in die Falle locken werden.

Schuldfrei heißt nicht folgenlos: Von den Menschen wird weiterhin erwartet, vollständig Rechenschaft abzulegen, und die Organisation ist dafür verantwortlich, die Bedingungen zu beheben.

Gute Reviews vermeiden den Rückschaufehler (das Ergebnis war damals nicht offensichtlich), kontrafaktische Aussagen („hätte sollen“, „hat es versäumt“) und die Suche nach einer einzigen Grundursache. Richard Cooks kurzer Aufsatz „How Complex Systems Fail“ (1998) argumentiert, dass bei schweren Ausfällen in komplexen Systemen mehrere Mängel zusammenkommen müssen, sodass es keine einzelne Grundursache gibt, nur beitragende Faktoren. Eine Five-Whys-Kette, die bei „menschlichem Versagen“ endet, hat zu früh aufgehört. Halten Sie auch fest, was gut lief: Die Expertise und Improvisation, die den Schaden begrenzt haben, gehören dazu, wie das System wirklich in Betrieb bleibt, und sie bleiben unsichtbar, wenn niemand sie festhält.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: vom Wer zum Wie

Der schwierigste Gedanke dieses Abschnitts ist, dass ein gutes Post-Incident-Review über die Person hinausblickt, die den letzten Fehler gemacht hat, hin zu den Bedingungen, die ihn leicht gemacht haben, denn diese Bedingungen werden auch die nächste Person in die Falle locken. Stellen Sie sich vor, das Timeout aus dem ausgearbeiteten Beispiel wäre nicht durch ein Experiment entdeckt worden.

- Eines Mittags wird der Speisekartenservice langsam. Die Aufrufe des Bestellservice an ihn warten bis zu 3 Sekunden statt der vorgesehenen 300 ms, seine Threads gehen aus, und Bezahlvorgänge schlagen in der ganzen Region fehl.
- Die erste Geschichte ist schnell geschrieben: Ein Entwickler hat den Speisekarten-Client mit einem Timeout von 3 Sekunden ausgeliefert. Ihre Maßnahme ergibt sich von selbst: bei der Konfiguration sorgfältiger sein.
- Die zweite Geschichte fragt, warum der Fehler leicht zu machen und schwer zu bemerken war. Der Entwickler hatte 300 ms eingetragen, aber ein Bibliotheks-Standardwert von 3000 ms überschrieb das stillschweigend; nichts im Build prüfte auf explizite Timeouts; und im Entwurf stand 300 ms, sodass die Reviewer nichts Falsches sahen.
- Das sind beitragende Faktoren: mehrere Bedingungen, die zusammenkamen. Jeder Entwickler könnte in dieselbe Falle tappen, daher ändert es nichts, diesem einen die Schuld zu geben.
- Weil das Review im Sinne der Blameless Culture schuldfrei ist, kann der Entwickler offen sagen, was er konfiguriert hat und warum, was das Team zum Bibliotheks-Standardwert führt.
- Die Maßnahmen ändern, wie im ausgearbeiteten Beispiel, das System: das Timeout in der gemeinsamen Client-Konfiguration setzen und eine Build-Prüfung ergänzen, die fehlschlägt, wenn einem ausgehenden Aufruf ein explizites Timeout fehlt, jeweils mit einer verantwortlichen Person und einem Fälligkeitstermin.
- **Die Lehre:** ‚Menschliches Versagen' ist der Ausgangspunkt eines Reviews, nicht sein Ende. Fragen Sie, warum jede Handlung damals sinnvoll erschien, und beheben Sie die Bedingungen; eine Ermahnung zur Vorsicht lässt die Falle scharf.

Aus Incidents lernen, und die Grenzen von MTTR

Organisationen wollen natürlich eine Zahl, die zeigt, ob sich die Incident Response verbessert, und die Mean Time to Recovery (MTTR) ist der übliche Kandidat. Kapitel 2 verwendete MTTR in A = MTBF / (MTBF + MTTR), wo sie für eine einzelne Komponente mit wiederholbaren Ausfällen

aussagekräftig ist. Als organisatorische Kennzahl über viele verschiedene Incidents hinweg ist sie viel schwächer. Die Dauern von Incidents sind stark schief verteilt: Die meisten sind kurz und einige wenige sehr lang, sodass der Mittelwert von einer Handvoll Ausreißern dominiert wird. Courtney Nashs Analysen Tausender öffentlicher Incident-Berichte in der Verica Open Incident Database (VOID-Berichte ab 2021) fanden genau diese Schiefe, und ein Google-Bericht von Štěpán Davidovič aus dem Jahr 2021, *Incident Metrics in SRE*, zeigte mit Simulationen, dass Veränderungen der MTTR bei realistischen Incident-Zahlen überwiegend Rauschen sind und nicht verlässlich zeigen können, ob eine Verbesserung gewirkt hat.

Bessere Signale sind die Auswirkungen auf SLOs und Error Budgets (die Incidents danach gewichten, was Benutzer gespürt haben), wiederkehrende Themen über Reviews hinweg (dieselbe Art von Konfigurationsänderung, die dreimal Ärger macht), Beinahe-Vorfälle, wie schnell Incidents erkannt werden, wie oft Reviews zu abgeschlossenen Systemänderungen führen, und qualitative Belege, etwa ob Engineers Reviews nützlich finden. Viele Organisationen veröffentlichen Reviews intern in einer durchsuchbaren Bibliothek und halten regelmäßige Incident-Review-Meetings ab, damit die Lehre eines Teams auch andere erreicht.

Maßnahmen, die das System tatsächlich ändern

Ein Review ist nur so viel wert wie die Änderungen, die es bewirkt. Maßnahmen unterscheiden sich stark in ihrer Wirkung. Am stärksten sind solche, die die Gefahr beseitigen (den gefährlichen Befehl löschen, den unsicheren Zustand unmöglich machen), dann Leitplanken, die verhindern, dass sich der Fehler ausbreitet (Validierung der Konfiguration, gestaffelte Rollouts und Canaries, Rate Limits für Änderungen, automatischer Rollback bei SLO-Verbrauch), dann schnellere Erkennung und Eindämmung (ein neuer SLO-Alert, ein getestetes Runbook, ein automatisiertes Failover). Am schwächsten sind Schulungen, Dokumentation und Ermahnungen. „Sorgfältiger sein“ ist keine Maßnahme; es ändert nichts an den Bedingungen, auf die die nächste Person treffen wird.

Jede Maßnahme braucht eine verantwortliche Person, eine Priorität und einen Fälligkeitstermin und muss im normalen Backlog des Teams stehen statt in einem vergessenen Dokument. Die Error-Budget-Policy aus Abschnitt 9.5 liefert den Hebel: Ein Incident, der einen großen Teil des Budgets verbraucht hat, verschafft seinen Korrekturen Vorrang vor Features. Halten Sie die Liste kurz; fünf erledigte Maßnahmen sind besser als zwanzig offene. Wo eine Maßnahme die Architektur ändert, etwa durch ein zusätzliches Bulkhead oder das Entfernen einer synchronen Abhängigkeit, halten Sie sie in einem Architecture Decision Record fest, der den Incident nennt, und verlinken Sie die relevanten Runbooks aus den Alerts, die beim nächsten Mal auslösen.

Checkliste: ein Post-Incident-Review durchführen

Nutzen Sie sie für jeden Incident ab einem vereinbarten Schweregrad, für jeden Incident, der mehr als einen festgelegten Anteil eines Error Budgets verbraucht, und für interessante Beinahe-Vorfälle.

- [ ] **Innerhalb weniger Arbeitstage ansetzen**
  Erinnerungen verblassen schnell; Logs und Chatverläufe laufen womöglich ab. Laden Sie die Beteiligten ein, einen Moderator, der nicht beteiligt war, und die Verantwortlichen der betroffenen Services.
- [ ] **Den Zeitablauf zuerst aus Belegen aufbauen**
  Fügen Sie Alerts, Deployments, Chatnachrichten, Dashboards und Traces vor dem Meeting zu einem Zeitablauf zusammen, damit die Diskussion von Fakten statt von Erinnerungen ausgeht.
- [ ] **Die Auswirkungen aus Sicht der Benutzer beschreiben**
  Wer war betroffen, wie viele, wie lange, welche SLOs und wie viel Error Budget; nehmen Sie, wo relevant, Geld, Daten und regulatorische Risiken auf.
- [ ] **Den schuldfreien Rahmen ausdrücklich setzen**
  Sagen Sie zu Beginn, dass es darum geht zu verstehen, wie das System und die Arbeit wirklich ablaufen, und dass niemand dafür bestraft wird, zu schildern, was er getan hat und warum.
- [ ] **Fragen, warum es damals sinnvoll erschien**
  Fragen Sie bei jeder wichtigen Entscheidung, was die Person wusste, was sie erreichen wollte und was die Werkzeuge ihr angezeigt haben. Vermeiden Sie kontrafaktische Formulierungen wie ‚hätte sollen' und ‚hat es versäumt'.
- [ ] **Nach mehreren beitragenden Faktoren suchen**
  Technische, organisatorische und prozedurale Bedingungen zusammen; widerstehen Sie der Versuchung, bei der ersten einzelnen ‚Grundursache' oder bei menschlichem Versagen stehen zu bleiben.
- [ ] **Festhalten, was gut lief**
  Halten Sie die Anpassungen, die Expertise und das Glück fest, die den Schaden begrenzt haben; sie gehören dazu, wie das System wirklich in Betrieb bleibt.
- [ ] **Wenige Maßnahmen vereinbaren, die das System ändern**
  Ziehen Sie das Beseitigen von Gefahren und das Schaffen von Leitplanken Schulungen und Ermahnungen vor; geben Sie jeder Maßnahme eine verantwortliche Person, eine Priorität und einen Fälligkeitstermin, und verfolgen Sie sie im normalen Backlog.

- [ ] **Veröffentlichen und teilen**
  Schreiben Sie das Review in einer Standardvorlage auf, teilen Sie es breit, fügen Sie es einer durchsuchbaren Bibliothek hinzu und gehen Sie offene Maßnahmen in einem regelmäßigen Review-Meeting erneut durch.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum braucht ein Chaos-Experiment eine Steady-State-Hypothese, statt einfach ‚zu sehen, was kaputtgeht'?

Weil eine messbare Vorhersage über das normale, für Benutzer sichtbare Verhalten zeigt, ob das System zurechtkam, Abbruchbedingungen liefert und die Übung zu einem Beleg statt zu einer Anekdote macht.

▼ Warum sollte der Incident Commander nicht debuggen?

Weil die Rolle Koordination ist: Prioritäten, Rollen, Entscheidungen und Überblick. Ein Commander, der in technischen Details versinkt, lässt niemanden die Reaktion steuern.

▼ Was ist eine ‚zweite Geschichte' in einem Post-Incident-Review?

Die Darstellung, die über die Frage hinausgeht, wer einen Fehler gemacht hat, hin zu den Bedingungen, Werkzeugen, dem Druck und dem Design, die die Handlung vernünftig erscheinen ließen und auch die nächste Person in die Falle locken werden.

▼ Warum ist die durchschnittliche MTTR über ein Jahr voller Incidents ein schwaches Maß für Verbesserung?

Die Dauern von Incidents sind stark schief verteilt, sodass einige lange Incidents den Mittelwert dominieren, und bei realistischen Incident-Zahlen sind Veränderungen daran überwiegend Rauschen.

QUELLEN

- Principles of Chaos Engineering: https://principlesofchaos.org/
- Netflix Technology Blog: The Netflix Simian Army (2011): https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116
- John Allspaw: Blameless PostMortems and a Just Culture (Etsy, 2012): https://www.etsy.com/codeascraft/blameless-postmortems
- Google SRE book: Postmortem Culture: Learning from Failure: https://sre.google/sre-book/postmortem-culture/

- Google SRE: Incident Metrics in SRE (Štěpán Davidovič): https://sre.google/resources/practices-and-processes/incident-metrics-in-sre/

Weiterführende Literatur

- *Release It! Design and Deploy Production-Ready Software (2nd edition)*, Michael T. Nygard (2018), Pragmatic Bookshelf. Das Buch, das den Mustern Circuit Breaker und Bulkhead ihre Namen gab; seine Stabilitätsmuster und Anti-Patterns, illustriert mit realen Ausfällen, sind die Grundlage von Abschnitt 9.4.
- *The Art of Scalability (2nd edition)*, Martin L. Abbott and Michael T. Fisher (2015), Addison-Wesley. Führt den Scale Cube ein und verbindet technische Skalierungsentscheidungen mit Organisation und Prozessen.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (editors) (2016), O'Reilly Media. Kostenlos online; die Quelle von SLOs und Error Budgets, wie sie heute praktiziert werden, und hervorragender Kapitel über Überlast, kaskadierende Ausfälle und Post-Mortems.
  https://sre.google/sre-book/table-of-contents/
- *The Site Reliability Workbook: Practical Ways to Implement SRE*, Betsy Beyer, Niall Richard Murphy, David K. Rensin, Kent Kawahara and Stephen Thorne (editors) (2018), O'Reilly Media. Das praktische Begleitbuch: SLOs umsetzen, Burn-Rate-Alerting über mehrere Zeitfenster und Rufbereitschaft, mit ausgearbeiteten Beispielen.
  https://sre.google/workbook/table-of-contents/
- *Exponential Backoff And Jitter*, Marc Brooker (2015), AWS Architecture Blog. Ein kurzer, auf Simulationen gestützter Artikel, der zeigt, warum Wiederholungen zufällige Verzögerungen brauchen; der Ursprung der weit verbreiteten Full-Jitter-Formel.
  https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- *Systems Performance: Enterprise and the Cloud (2nd edition)*, Brendan Gregg (2020), Addison-Wesley. Das Standardwerk zur Performanceanalyse, einschließlich der USE-Methode, Observability-Werkzeugen und Fallstricken beim Benchmarking.
- *Chaos Engineering: System Resiliency in Practice*, Casey Rosenthal and Nora Jones (2020), O'Reilly Media. Geschrieben von Praktikern von Netflix und anderen; erklärt die Prinzipien des Chaos Engineering und wie Organisationen es einführen.
- *Fail at Scale: Reliability in the face of rapid change*, Ben Maurer (2015), ACM Queue, vol. 13, no. 8. Wie Facebook Überlast und langsame Abhängigkeiten mit Queues mit kontrollierter Verzögerung, adaptivem LIFO und clientseitigen Nebenläufigkeitslimits bewältigt.
  https://queue.acm.org/detail.cfm?id=2839461

Glossar

Anycast
Die Bekanntgabe derselben IP-Adresse von vielen Standorten aus, sodass das Routing des Internets die Pakete jedes Benutzers an den nächstgelegenen Standort liefert, der sie bekannt gibt.

Autoscaling
Die automatische Änderung der Zahl der Instanzen bei wechselnder Last: reaktiv anhand einer Metrik, nach Zeitplan oder vorausschauend anhand einer Prognose.

Beitragender Faktor
Eine von mehreren Bedingungen, die einen Incident gemeinsam möglich gemacht oder verschlimmert haben; Reviews suchen nach vielen beitragenden Faktoren statt nach einer einzigen Grundursache.

Blameless Culture
Ein Ansatz, bei dem Menschen, die an einem Incident beteiligt waren, ohne Angst vor Bestrafung beschreiben können, was sie getan haben und warum, damit die Organisation lernt, wie das System wirklich funktioniert.

Blast Radius
Wie viel vom System und wie viele Benutzer ein Ausfall, eine Änderung oder ein Experiment betreffen kann; Chaos-Experimente halten ihn bewusst klein.

Breakpoint-Test
Ein Performance-Test, der die Last stetig steigert, bis das System zusammenbricht, um seine maximale Kapazität und seinen ersten Engpass zu finden.

Bulkhead
Die Aufteilung von Ressourcen wie Thread-Pools, Connection-Pools oder Instanzen, damit ein Ausfall oder eine Überlast in einem Teil nicht die Ressourcen der anderen aufbrauchen kann.

Burn Rate
Wie schnell ein Error Budget verbraucht wird, im Verhältnis zu der Rate, die es genau am Ende des Zeitfensters aufbrauchen würde; eine Burn Rate von 1 verbraucht es in genau einem Zeitfenster.

Chaos Engineering
Experimente an einem System, idealerweise in Produktion und unter Kontrolle, bei denen realistische Ausfälle eingespeist werden, um Vertrauen zu gewinnen, dass es turbulenten Bedingungen standhält.

Circuit Breaker
Eine Hülle um Aufrufe an eine Abhängigkeit, die nach genügend Fehlschlägen eine Weile keine Aufrufe mehr durchlässt und sofort fehlschlägt und dann Testaufrufe durchlässt, um die Erholung zu prüfen (Michael Nygard, 2007).

Competing Consumers
Ein Muster, bei dem mehrere Worker aus derselben Queue lesen und jede Nachricht nur von einem von ihnen verarbeitet wird, sodass der Durchsatz durch zusätzliche Worker steigt.

Connection Draining
Das Stoppen neuer Anfragen an eine Instanz, die entfernt wird, während bereits laufende Anfragen zu Ende bearbeitet werden, sodass beim Herunterskalieren und bei Deployments keine Arbeit verloren geht.

Cooldown (Stabilisierungsfenster)
Ein Zeitraum nach einer Skalierungsaktion, in dem der Autoscaler wartet, bevor er erneut handelt, damit die Wirkung der letzten Änderung gemessen werden kann.

Coordinated Omission
Ein von Gil Tene benannter Messfehler, bei dem ein Lastgenerator auf langsame Antworten wartet, bevor er weitere Anfragen sendet, und so die Verzögerungen nie erfasst, die echte Benutzer erlitten hätten.

Counter
Eine Metrik, deren Wert nur steigt (oder bei einem Neustart auf null zurückgesetzt wird), etwa die Gesamtzahl bedienter Anfragen; meist betrachtet man ihre Zuwachsrate.

Deadline-Weitergabe (Deadline Propagation)
Die Weitergabe der für eine gesamte Anfrage verbleibenden Zeit an jeden nachgelagerten Aufruf, damit kein Service an einer Anfrage weiterarbeitet, die sein Aufrufer bereits aufgegeben hat.

Distributed Tracing
Das Aufzeichnen von Weg und Zeitverhalten einer Anfrage auf ihrem Weg durch viele Services, damit sich der langsame oder fehlschlagende Schritt finden lässt.

Error Budget
Das Maß an Unzuverlässigkeit, das ein SLO zulässt: eins minus Zielwert, bezogen auf das Zeitfenster. Ein SLO von 99,9 % bei 1.000.000 Anfragen erlaubt 1000 fehlerhafte Anfragen.

Error-Budget-Policy
Eine vereinbarte, schriftliche Festlegung, was ein Team tut, wenn sein Error Budget zu schnell verbraucht wird oder erschöpft ist, etwa riskante Releases einfrieren.

Exponentielles Backoff
Zwischen aufeinanderfolgenden Wiederholungen immer länger warten, typischerweise mit Verdopplung der Wartezeit bis zu einer Obergrenze, damit ein überlasteter Service Zeit zur Erholung bekommt.

Flapping
Ein Autoscaler fügt wiederholt Kapazität hinzu und entfernt sie wieder, weil er auf kurzlebige Änderungen oder auf die Auswirkungen seiner eigenen Aktionen reagiert.

Four Golden Signals
Latenz, Traffic, Fehler und Sättigung: die vier Messgrößen, die das Google-SRE-Buch für jedes benutzerseitige System zur Überwachung empfiehlt.

Game Day
Eine geplante Übung, bei der ein Team absichtlich einen Ausfall herbeiführt oder simuliert und dessen Erkennung, die Reaktion darauf und die Wiederherstellung übt.

Gauge
Eine Metrik, die einen aktuellen Wert festhält, der steigen oder fallen kann, etwa die Länge einer Queue, den belegten Speicher oder die Temperatur.

Geschlossenes Lastmodell
Ein Lastmodell mit einer festen Zahl von Benutzern, von denen jeder erst dann eine neue Anfrage sendet, wenn die vorige beantwortet ist (plus Denkzeit), sodass die Last sinkt, wenn das System langsamer wird.

Gleitendes Zeitfenster
Ein Messzeitraum, der sich kontinuierlich weiterbewegt, etwa die letzten 28 Tage, im Gegensatz zu einem Kalendermonat, der am Ersten neu beginnt.

Globales Load Balancing
Die Verteilung von Benutzern auf Rechenzentren oder Regionen, meist durch DNS-Antworten, die sich je nach Standort und Zustand unterscheiden, oder durch Anycast-Routing.

Health Check
Ein regelmäßiger Test, etwa eine HTTP-Anfrage an einen Status-Endpoint, anhand dessen ein Load Balancer oder Orchestrator entscheidet, ob eine Instanz Traffic erhalten oder neu gestartet werden soll.

Hedged Request
Nach kurzer Verzögerung eine zweite Kopie einer langsamen Anfrage an ein anderes Replikat senden und die Antwort verwenden, die zuerst eintrifft, um die Tail-Latenz zu senken.

Histogramm
Eine Metrik, die Beobachtungen wie Anfragedauern in Wertebereiche, sogenannte Buckets, einsortiert, sodass sich Perzentile schätzen und Histogramme vieler Server addieren lassen.

Incident Commander
Die Person, die die Reaktion auf einen Incident koordiniert: Sie setzt Prioritäten, verteilt Rollen und trifft Entscheidungen, statt selbst Dinge zu reparieren.

Jitter
Zufall, der Wartezeiten für Wiederholungen oder Zeitpläne beigemischt wird, damit nicht viele Clients im selben Moment handeln und synchronisierte Lastspitzen erzeugen.

Kapazitätsplanung
Die Prognose des künftigen Bedarfs, die Messung, wie viel jede Infrastruktureinheit bewältigen kann, und die rechtzeitige Bereitstellung ausreichender Kapazität mit Reserve, bevor sie gebraucht wird.

Kapazitätsreserve (Headroom)
Freie Kapazität, die bewusst über der erwarteten Spitzenlast vorgehalten wird, um Lastspitzen, Ausfälle und die Zeit bis zur Bereitstellung weiterer Kapazität abzufangen.

Kardinalität
Die Zahl der unterschiedlichen Werte, die ein Feld annehmen kann; bei Metriken die Zahl der unterschiedlichen Label-Kombinationen, von denen jede zu einer eigenen zu speichernden Zeitreihe wird.

Kontextweitergabe (Context Propagation)
Die Weitergabe von Trace-IDs (und anderem Anfragekontext) von einem Service zum nächsten mit jedem Aufruf oder jeder Nachricht, zum Beispiel im W3C-Header traceparent.

Lasttest
Ein Performance-Test, der die erwartete (oder prognostizierte Spitzen-)Last über einen Zeitraum anlegt, um zu prüfen, ob die Ziele für Latenz und Fehlerrate eingehalten werden.

Layer-4-Load-Balancer
Ein Load Balancer, der auf der Transportebene (TCP oder UDP) arbeitet und Verbindungen oder Pakete weiterleitet, ohne die darin enthaltene HTTP-Anfrage zu lesen.

Layer-7-Load-Balancer
Ein Load Balancer, der das Anwendungsprotokoll versteht, meist HTTP, und daher jede Anfrage nach Pfad, Header oder Cookie routen, TLS terminieren und Anfragen wiederholen kann.

Leaky Bucket
Ein Algorithmus für Rate Limiting oder Traffic Shaping, bei dem Anfragen in einen Eimer gelangen und ihn mit konstanter Rate verlassen; ist der Eimer voll, werden neue Anfragen abgewiesen.

Least Connections
Ein Load-Balancing-Algorithmus, der jede neue Anfrage an die Instanz schickt, die gerade die wenigsten Anfragen oder Verbindungen in Bearbeitung hat.

Liveness Probe
Ein Health Check, der prüft, ob ein Prozess unrettbar hängt; schlägt er fehl, startet der Orchestrator die Instanz neu.

Load Shedding
Das bewusste, frühe und kostengünstige Abweisen einiger Anfragen, wenn ein System überlastet ist, damit die angenommenen Anfragen weiterhin rechtzeitig erfolgreich sind.

Monitoring
Das Erfassen und Beobachten vordefinierter Messwerte und Prüfungen, damit bekannte Arten von Ausfällen erkannt und gemeldet werden.

Multi-Window-, Multi-Burn-Rate-Alert
Ein SLO-Alert, der nur auslöst, wenn die Burn Rate sowohl über ein langes als auch über ein kurzes Zeitfenster über einem Schwellenwert liegt, und so schnelle Erkennung mit schnellem Zurücksetzen und wenigen Fehlalarmen verbindet.

Offenes Lastmodell
Ein Lastmodell, bei dem neue Anfragen mit einer festgelegten Rate eintreffen, unabhängig davon, wie schnell frühere beantwortet werden, so wie Kunden, die eine Website aufrufen.

OpenTelemetry
Ein herstellerneutrales Open-Source-Projekt der CNCF, das APIs, SDKs, ein Protokoll (OTLP) und einen Collector bereitstellt, um Traces, Metriken und Logs zu erzeugen und zu exportieren.

Performance-Budget
Eine vereinbarte Obergrenze für eine Performance-Kennzahl, etwa das Seitengewicht oder die p99-Latenz pro Service-Aufruf, die eine Änderung nicht überschreiten darf, idealerweise automatisch geprüft.

Post-Incident-Review
Ein strukturiertes Meeting mit schriftlichem Bericht nach einem Incident, das rekonstruiert, was geschehen ist, warum es den Beteiligten damals sinnvoll erschien und was sich ändern wird; auch Postmortem genannt.

Power of Two Random Choices
Ein Load-Balancing-Verfahren, das zwei Instanzen zufällig auswählt und die Anfrage an die weniger ausgelastete schickt; Mitzenmacher zeigte, dass es fast so gut verteilt, wie wenn man jede Instanz prüfen würde.

Rate Limiting
Die Begrenzung, wie viele Anfragen ein Client, ein Schlüssel oder ein Mandant in einem Zeitraum stellen darf, um Kapazität zu schützen und fair zu verteilen; überzählige Anfragen werden meist mit HTTP 429 beantwortet.

Readiness Probe
Ein Health Check, der prüft, ob eine Instanz gerade Datenverkehr bedienen kann; schlägt er fehl, wird die Instanz aus dem Load Balancer genommen, ohne neu gestartet zu werden.

RED-Methode
Tom Wilkies Checkliste für das Monitoring anfragegesteuerter Services: Rate, Errors und Duration (Rate, Fehler und Dauer) der Anfragen.

Retry-Budget
Eine Obergrenze für Wiederholungen, ausgedrückt als Anteil des normalen Datenverkehrs (zum Beispiel 10 %), damit Wiederholungen bei gelegentlichen Fehlern helfen, die Last während eines Ausfalls aber nicht vervielfachen können.

Retry-Sturm
Eine Lastwelle, die entsteht, wenn viele Clients, oft auf mehreren Ebenen, fehlschlagende Anfragen gleichzeitig wiederholen, wodurch ein überlasteter Service überlastet bleibt.

Schweregrad
Eine vordefinierte Einstufung, wie schwerwiegend ein Incident ist, etwa SEV1 bis SEV4, die festlegt, wer alarmiert wird, wie schnell und wie breit kommuniziert wird.

Service-Gutschrift
Eine Erstattung oder ein Rabatt, der einem Kunden nach einem SLA zusteht, wenn der Anbieter ein zugesagtes Service-Level verfehlt.

Soak-Test
Ein langer Performance-Test unter normaler Last, der über Stunden oder Tage läuft, um schleichende Probleme wie Speicherlecks oder wachsende Warteschlangen aufzudecken; auch Dauertest (Endurance Test) genannt.

Span
Eine zeitlich gemessene Operation innerhalb eines Trace, etwa ein HTTP-Aufruf oder eine Datenbankabfrage, mit Name, Startzeit, Dauer, Attributen und einer Verknüpfung zu ihrem übergeordneten Span.

Spike-Test
Ein Performance-Test, der die Last plötzlich erhöht, zum Beispiel innerhalb von Sekunden auf das Zehnfache, um Autoscaling, Warteschlangen und Schutzmechanismen zu prüfen.

Statusseite
Eine öffentliche oder für Kunden bestimmte Seite, die über aktuelle Incidents und den Zustand der Services informiert und von der für die Kommunikation zuständigen Person des Incidents aktualisiert wird.

Steady-State-Hypothese
In einem Chaos-Experiment die Vorhersage, dass ein messbares Anzeichen für normales Verhalten, etwa Bestellungen pro Minute, innerhalb festgelegter Grenzen bleibt, während ein Fehler eingespeist wird.

Sticky Session (Session-Affinität)
Eine Load-Balancer-Einstellung, die alle Anfragen eines Clients an dieselbe Instanz schickt, meist damit Sitzungsdaten im Speicher dieser Instanz gefunden werden.

Stresstest
Ein Performance-Test, der die Last über die erwartete Spitze hinaus steigert, um zu sehen, wie das System versagt und ob es sich erholt.

Strukturiertes Logging
Das Schreiben von Logeinträgen als maschinenlesbare Schlüssel-Wert-Datensätze (oft JSON) mit einheitlichen Feldnamen statt als Freitextsätze.

Telemetrie
Daten, die ein laufendes System über sein eigenes Verhalten ausgibt: Logs, Metriken, Traces, Profile und Ereignisse.

Timeout
Die maximale Zeit, die ein Aufrufer auf eine Operation wartet, bevor er aufgibt und sie als fehlgeschlagen behandelt; jeder entfernte Aufruf braucht einen.

Token Bucket
Ein Algorithmus für Rate Limiting, bei dem einem Eimer mit fester Rate Tokens bis zu einer Kapazität hinzugefügt werden und jede Anfrage ein Token verbraucht; er erlaubt Lastspitzen bis zur Kapazität und eine gleichmäßige Durchschnittsrate.

Trace-Sampling
Das Aufbewahren nur eines Teils der Traces, um die Kosten zu begrenzen: Head-based Sampling entscheidet zu Beginn einer Anfrage, Tail-based Sampling nach ihrem Ende, sodass es langsame und fehlgeschlagene Anfragen behalten kann.

USE-Methode
Brendan Greggs Checkliste zur Analyse von Ressourcen: Prüfen Sie für jede Ressource Utilisation, Saturation und Errors (Auslastung, Sättigung und Fehler).

Virtueller Benutzer
Ein simulierter Benutzer in einem Lasttestwerkzeug, der ein Skript von Anfragen ausführt, meist in einer Schleife.

Zustandsloser Service
Eine Service-Instanz, die zwischen Anfragen keine Daten in ihrem eigenen Speicher oder auf ihrer Festplatte hält, sodass jede Instanz jede Anfrage bearbeiten kann und Instanzen beliebig hinzugefügt oder entfernt werden können.