BE SOFTWARE ARCHITECTURE WISE · KAPITEL 2

Qualitätsmerkmale und architekturrelevante Anforderungen

Architektur ist dazu da, Qualitäten zu liefern: Geschwindigkeit unter Last, Betrieb auch dann, wenn Teile ausfallen, gefahrlose Änderbarkeit, Geheimhaltung von Daten. Dieses Kapitel zeigt, wie man vage Wünsche wie „schnell“, „zuverlässig“ und „sicher“ in präzise, testbare Anforderungen verwandelt, die Designentscheidungen lenken können. Es behandelt den Unterschied zwischen funktionalen Anforderungen, Qualitätsmerkmalen und Randbedingungen, das Qualitätsmodell nach ISO/IEC 25010, Qualitätsszenarien, die Arithmetik von Performance und Verfügbarkeit, das Auslegen auf Veränderung, Sicherheit, Benutzbarkeit und Kosten als Qualitäten sowie das Erheben, Priorisieren und Abwägen von Qualitätsmerkmalen. Es eignet sich für Entwicklerinnen und Entwickler auf dem Weg zur Architektur und für alle, die Anforderungen schreiben oder prüfen.

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

AUF EINEN BLICK

Das lernen Sie

- Funktionale Anforderungen, Qualitätsmerkmale und Randbedingungen unterscheiden und eine architekturrelevante Anforderung erkennen.
- Das Produktqualitätsmodell ISO/IEC 25010:2023 als Checkliste nutzen, ohne dass es zu Papierkram wird.
- Sechsteilige Qualitätsszenarien mit messbaren Antwortmaßen schreiben und vage Aussagen in testbare verwandeln.
- Mit Perzentilen, Littles Gesetz, einfachen Warteschlangenformeln und dem Amdahlschen Gesetz rechnen, um über Performance und Skalierbarkeit zu urteilen.
- Verfügbarkeit aus den Neunen, aus MTBF und MTTR sowie für in Reihe und parallel geschaltete Komponenten berechnen.
- Erklären, wie Änderbarkeit, Testbarkeit, Deploybarkeit, Sicherheit, Benutzbarkeit, Kosten und Nachhaltigkeit spezifiziert und erreicht werden.
- Qualitätsanforderungen mit einem Utility Tree erheben und priorisieren und die Abwägungen zwischen konkurrierenden Qualitäten erklären.

Bevor Sie beginnen

Kapitel 1, besonders die Abschnitte über Stakeholder und Anliegen sowie über Entscheidungen und Abwägungen. Etwas Sicherheit im einfachen Rechnen (Prozente, Multiplikation und Division).

Dauer

Etwa 3 Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

Qualitätsmerkmal · Randbedingung · architekturrelevante Anforderung · ISO/IEC 25010 · Qualitätsszenario · Antwortmaß · Latenz und Durchsatz · Perzentil · Littles Gesetz · Verfügbarkeit · MTTR · Taktik · Utility Tree · Fitness Function

Warum Qualitäten und nicht Features über die Architektur entscheiden

Fragen Sie eine beliebige Gruppe von Stakeholdern, was ein System tun muss, und Sie erhalten eine Liste von Features: einen Flug suchen, einen Warenkorb bezahlen, einen Termin buchen. Fragen Sie, wie gut es das tun muss, und die Antworten werden meist vager: Es soll schnell sein, es soll nicht ausfallen, es soll sicher sein, es soll leicht zu ändern sein. Diese zweiten Antworten sind Qualitätsmerkmale, und um sie geht es in der Architektur vor allem. Dieselbe Liste von Features lässt sich mit einem einzigen Programm auf einem Server umsetzen oder mit vierzig Services, verteilt über drei Kontinente. Was diese Designs unterscheidet, ist nicht, was das System tut, sondern wie viele Menschen es gleichzeitig bedienen muss, wie viel Ausfallzeit das Unternehmen verkraften kann, wie oft sich der Code ändern wird und wie sensibel die Daten sind.

Kapitel 1 beschrieb Architektur als die Menge der bedeutsamen Entscheidungen, die die Anliegen der Stakeholder durch Abwägungen erfüllen. Dieses Kapitel macht diese Anliegen präzise. Eine Anforderung, die eine messbare Wirkung auf die Struktur eines Systems hat, ist eine architekturrelevante Anforderung, und die meisten davon erweisen sich als Qualitätsmerkmale oder Randbedingungen statt als Funktionen. Sie früh zu finden und so zu formulieren, dass sie sich testen lassen, gehört zum Wertvollsten, was ein Architekt tut.

Wie das Kapitel zusammenhängt

Abschnitt 2.1 trennt funktionale Anforderungen, Qualitätsmerkmale und Randbedingungen, erklärt, warum viele Architekten den Begriff „nichtfunktionale Anforderungen“ meiden, und spielt ein Konzertticketsystem an dem Tag durch, an dem der Vorverkauf für eine große Tournee beginnt. Abschnitt 2.2 stellt das Produkt-Qualitätsmodell nach ISO/IEC 25010:2023 mit seinen neun Merkmalen vor und zeigt, wie man es als Gedächtnisstütze statt als auszufüllendes Formular nutzt. Abschnitt 2.3 vermittelt die nützlichste Technik des Kapitels: das sechsteilige Qualitätsszenario, das „muss schnell sein“ in eine testbare Aussage verwandelt.

Die Abschnitte 2.4 bis 2.7 besuchen dann die Qualitäten, die Architekturen am häufigsten prägen. Abschnitt 2.4 behandelt Performance und Skalierbarkeit mit echter Arithmetik: Latenz und Durchsatz, Perzentile, Tail-Latenz, Littles Gesetz, Warteschlangen und das Amdahlsche Gesetz. Abschnitt 2.5 tut dasselbe für Verfügbarkeit und Zuverlässigkeit: die Neunen, die Mean Time Between Failures und die Mean Time to Repair sowie die Verfügbarkeit von Komponenten in Reihe und parallel. Abschnitt 2.6 behandelt die Qualitäten der Veränderung (Änderbarkeit, Testbarkeit und Deploybarkeit), und Abschnitt 2.7 fasst Sicherheit, Benutzbarkeit und Barrierefreiheit, Interoperabilität, Kosten, Energieeffizienz, Betriebssicherheit und regulatorische Qualitäten zusammen.

Abschnitt 2.8 schließt den Kreis. Er zeigt, woher Qualitätsanforderungen kommen, wie ein Quality Attribute Workshop und ein Utility Tree aus einer Vielzahl von Meinungen eine Rangliste machen, welche Qualitäten mit welchen im Konflikt stehen und wie Fitness Functions die wichtigen Qualitäten nach dem Bau der Architektur weiter automatisch prüfen.

Von vagen Wünschen zu Zahlen

In diesem Kapitel beginnt der Kurs zu rechnen. Sie ermitteln, wie viele Anfragen aus einer Ankunftsrate und einer Antwortzeit gleichzeitig in Bearbeitung sind, warum eine Anfrage an einen Server, der zu 90 % ausgelastet ist, im Mittel das Zehnfache seiner Bedienzeit braucht, warum ein Job, der zu 10 % seriell ist, auf sechzehn Prozessoren verteilt nur eine 6,4-fache Beschleunigung bringt und wie viele Minuten Ausfallzeit pro Jahr ein Versprechen von „vier Neunen“ tatsächlich zulässt. Keine der Rechnungen ist schwerer als Schularithmetik, aber sie

verändern Gespräche: Aus „die Datenbank ist langsam“ wird „der Connection Pool ist für 100 gleichzeitige Anfragen ausgelegt, und laut Littles Gesetz brauchen wir in der Abendspitze 400“.

Zahlen entlarven auch falsche Sicherheit. Eine mittlere Antwortzeit von 120 Millisekunden kann verbergen, dass eine von fünfzig Anfragen länger als eine Sekunde dauert; deshalb spezifizieren Fachleute das 95. oder 99. Perzentil. Eine Anforderung ohne Antwortmaß lässt sich nicht testen, und eine Anforderung, die sich nicht testen lässt, wird stillschweigend geopfert, sobald ein Termin näher rückt.

Abwägungen sind die eigentliche Aufgabe

Kein System kann zugleich maximal schnell, verfügbar, sicher, günstig und leicht änderbar sein. Caching erhöht die Geschwindigkeit, erschwert aber die Konsistenz; Verschlüsselung und zusätzliche Prüfungen schützen Daten, erhöhen aber die Latenz; Redundanz steigert Verfügbarkeit und Kosten zugleich; flexible Plug-in-Designs erleichtern Änderungen und erschweren die Vorhersage der Performance. Das Kapitel behandelt Qualitäten daher als konkurrierende Ziele, die im Kontext gewichtet werden müssen, nicht als abzuhakende Kästchen. Die Kunst des Architekten besteht darin zu wissen, welche zwei oder drei Qualitäten für dieses System am wichtigsten sind, sie präzise zu formulieren und Designs zu wählen, deren Schwächen dort liegen, wo das Unternehmen sie sich leisten kann.

Die Designtechniken, mit denen jede Qualität erreicht wird, die sogenannten Taktiken, werden hier nur in Grundzügen vorgestellt. Spätere Kapitel bauen sie aus: Designprinzipien in Kapitel 3, Architekturstile in den Kapiteln 5 und 6, Daten-, Sicherheits-, Resilienz- und Cloud-Praktiken in den Kapiteln 7 bis 10 und Bewertungsmethoden wie die ATAM und Fitness Functions in Kapitel 11.

Der eine Gedanke, den Sie mitnehmen sollten

Wenn Sie sich nur eines aus diesem Kapitel merken, dann dies: Eine Qualität ist erst dann eine Anforderung, wenn sie einen Stimulus, einen Kontext und ein Maß hat. „Der Bezahlvorgang muss schnell sein“ ist ein Wunsch. „Wenn in der Freitagabendspitze 2000 Kunden pro Minute Bestellungen absenden, werden 99 % der Bezahlvorgänge innerhalb von 1,5 Sekunden abgeschlossen“ ist eine Anforderung. Man kann darauf hin entwerfen, sie testen, überwachen und mit Belegen über sie streiten, und sie sagt allen, vom Product Owner bis zum Betriebsteam, wie Erfolg aussieht.

Funktionale Anforderungen, Qualitätsmerkmale und Randbedingungen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.8 Wie Software entsteht
- 1.3 Warum sie zählt
- 1.5 Stakeholder und Anliegen

Begriffe von früher, die er verwendet: Stakeholder, User Story, Abwägung (Trade-off)

DIE IDEE IN KÜRZE

Denken Sie an die Bestellung einer Hochzeitstorte. „Drei Etagen, Zitronenbiskuit, weiße Glasur“ beschreibt, was die Torte ist: ihre Funktionen. „Fertig bis Samstag 14 Uhr, übersteht eine 40-minütige Autofahrt, unbedenklich für einen Gast mit Nussallergie“ beschreibt, wie gut sie funktionieren muss: ihre Qualitäten. „Die Location erlaubt nur Torten von zugelassenen Konditoreien“ ist eine Randbedingung: eine Entscheidung, die bereits für Sie getroffen wurde. Eine Konditorei kann denselben Zitronenbiskuit auf sehr unterschiedliche Weise herstellen, und es sind die Qualitäten und Randbedingungen, nicht der Geschmack, die bestimmen, wie die Torte gebaut und transportiert wird. Bei Software ist es genauso.

WAS SIE WISSEN MÜSSEN

- Funktionale Anforderungen sagen, was ein System tun muss; Qualitätsmerkmale sagen, wie gut es das tun muss; Randbedingungen sind bereits getroffene Entscheidungen wie Vorschriften, vorgegebene Technologien, Termine und Budgets.
- Viele Architekten meiden den Begriff „nichtfunktionale Anforderungen“, weil er eine große Kategorie durch Ausschluss definiert, nahelegt, die Anforderungen seien von den Funktionen getrennt, und optional klingt.
- Eine architekturrelevante Anforderung (ASR) ist eine, deren Änderung eine Änderung der Systemstruktur erzwingen würde; die meisten ASRs sind Qualitätsmerkmale oder Randbedingungen, aber auch manche Funktionen sind bedeutsam.
- Dieselben Funktionen lassen sich mit sehr unterschiedlichen Architekturen umsetzen; es sind die Qualitätsmerkmale und Randbedingungen, die zwischen ihnen entscheiden.

> - Implizite Anforderungen, etwa nie Bestellungen zu verlieren oder nie doppelt abzubuchen, werden selten aufgeschrieben und müssen aktiv bei Stakeholdern, aus Vorfällen und aus Vorschriften erhoben werden.

Drei Arten von Anforderungen

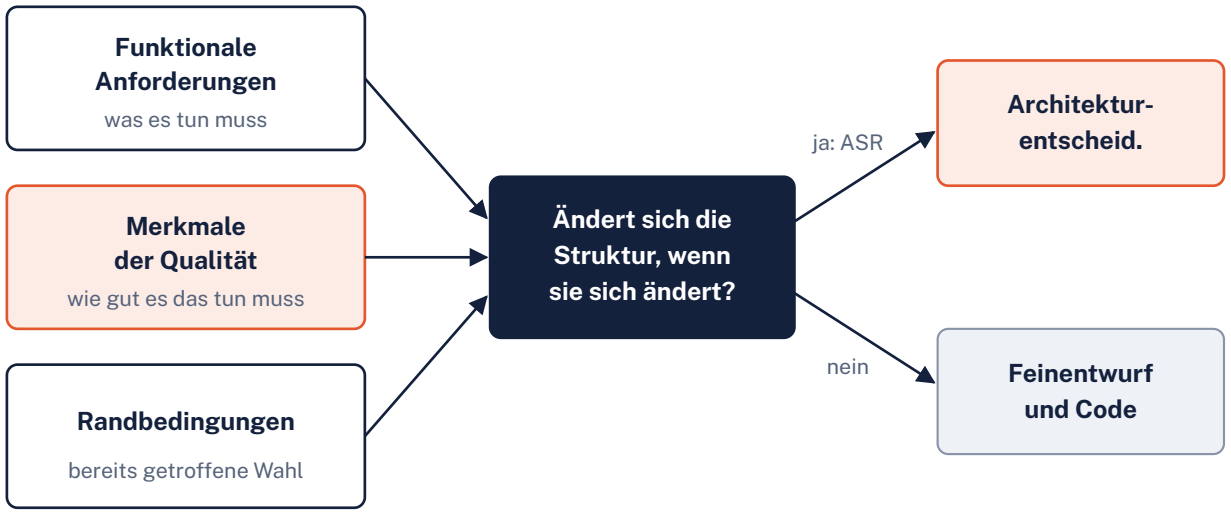
Eine Anforderung ist alles, was ein System erfüllen muss, um für seine Stakeholder akzeptabel zu sein. Architekten finden es nützlich, Anforderungen in drei Arten einzuteilen, weil jede Art das Design auf andere Weise beeinflusst. Eine funktionale Anforderung sagt, was das System tun muss: „Ein Kunde kann einen Artikel in den Warenkorb legen“, „das Apothekensystem prüft jedes neue Rezept gegen die Allergien des Patienten“, „ein Fahrgast kann die Position des Fahrers auf einer Karte sehen“. Funktionale Anforderungen beschreiben Verhalten, Daten und die Regeln, die beide verbinden. Sie sind das, was die meisten User Stories und Use Cases erfassen.

Ein Qualitätsmerkmal sagt, wie gut das System es tun muss. „Suchergebnisse erscheinen innerhalb einer halben Sekunde“, „der Buchungsservice ist zu 99,95 % der Zeit verfügbar“, „ein neuer Zahlungsanbieter lässt sich in weniger als zwei Wochen anbinden“, „Patientenakten sind nur für die behandelnden Ärztinnen und Ärzte lesbar“. Qualitätsmerkmale sind Eigenschaften des Systems als Ganzes oder einer Funktion, und sie ergeben nur mit einem Maß einen Sinn: schnell genug für wen, unter welcher Last, wie gemessen?

Eine Randbedingung ist eine Entscheidung, die bereits für Sie getroffen wurde. „Das System muss im bestehenden Azure-Tenant des Unternehmens laufen“, „alle personenbezogenen Daten bleiben in der Europäischen Union“, „das Team kann Java und hat kein Budget für Umschulungen“, „das erste Release muss live sein, bevor das Steuerjahr am 6. April beginnt“. Randbedingungen entfernen Optionen, statt Verhalten zu beschreiben. Innerhalb des Projekts sind sie nicht verhandelbar, doch ein guter Architekt prüft, ob jede wirklich eine Randbedingung ist und nicht bloß eine Gewohnheit oder Annahme.

Die Grenzen sind nicht immer scharf, und das ist in Ordnung. Der Sinn der Einteilung ist keine bürokratische Ordnung; sie soll sicherstellen, dass niemand fünfzig User Stories sammelt und vergisst zu fragen, wie viele Nutzer es geben wird, wie lange das Unternehmen einen Ausfall überlebt oder welche Aufsichtsbehörde zusieht.

Wie Anforderungen die Architektur erreichen



Alle drei Arten von Anforderungen fließen in die Architektur ein, aber nur diejenigen, deren Änderung eine strukturelle Änderung erzwingen würde, sind architekturrelevant.

Warum viele Architekten den Begriff „nichtfunktionale Anforderungen“ meiden

Anforderungsdokumente teilen traditionell alles in funktionale und nichtfunktionale Anforderungen (NFRs) ein. Der Begriff ist überall, in Verträgen, Ausschreibungsunterlagen und Stellenanzeigen, deshalb müssen Sie ihn kennen. Viele Architekten und Forschende meiden ihn dennoch, aus drei Gründen.

Erstens definiert er eine große und wichtige Kategorie über das, was sie nicht ist. Alles, was keine Funktion ist, landet im selben Topf: Antwortzeiten, Verschlüsselungsregeln, die Wahl des Cloud-Anbieters und die Farbe des Logos. Martin Glinz’ Paper *On Non-Functional Requirements* (IEEE International Requirements Engineering Conference, 2007) zeigte, dass veröffentlichte Definitionen uneins sind, was in die Kategorie gehört, und dass derselbe Bedarf je nach Formulierung so oder so eingeordnet werden kann: „Nur berechtigte Mitarbeitende dürfen Gehälter sehen“ liest sich als Funktion, wenn es als Zugriffsregel formuliert ist, und als Qualität, wenn es als Sicherheitsziel formuliert ist.

Zweitens legt der Name nahe, diese Anforderungen seien von den Funktionen getrennt und könnten später ergänzt werden. Das geht nicht. Ein Qualitätsmerkmal qualifiziert eine Funktion: Es ist der Bezahlvorgang, der schnell sein muss, die Zahlung, die nie verloren gehen darf, die Patientenakte, die vertraulich sein muss. Bass, Clements und Kazman sprechen in *Software Architecture in Practice* deshalb von Qualitätsmerkmal-Anforderungen und behandeln Randbedingungen als eigene Kategorie.

Drittens klingt „nichtfunktional“ optional, als ob ein System, das zwar läuft, aber langsam, unsicher oder nicht änderbar ist, trotzdem „funktioniert“. In der Praxis sind genau das die Systeme, die neu geschrieben werden. Manche Teams sagen lieber „Qualitätsanforderungen“; andere, vor allem bei Thoughtworks, sagen „Cross-Functional Requirements“, um zu betonen, dass sie quer durch alle Features gehen. Dieser Kurs spricht von Qualitätsmerkmalen und Randbedingungen. Wenn Ihre Organisation NFRs sagt, müssen Sie nicht um das Wort streiten; achten Sie nur darauf, dass jede NFR entweder eine messbare Qualität oder eine klar formulierte Randbedingung ist.

Randbedingungen: technisch, organisatorisch, regulatorisch, zeitlich und budgetär

Randbedingungen verdienen mehr Aufmerksamkeit, als sie meist bekommen, weil sie oft mehr von der Architektur bestimmen als jedes Feature. Eine Randbedingung ist einfach eine Anforderung ohne verbleibenden Spielraum. Es hilft, sie nach ihrer Herkunft zu gruppieren.

Arten von Randbedingungen und ihre Wirkung auf die Architektur

Art	Beispiel	Typische Wirkung auf die Architektur
Technisch	Muss über die veröffentlichten Schnittstellen in das bestehende SAP-Finanzsystem integriert werden; muss auf der Kubernetes-Plattform des Unternehmens laufen.	Legt Integrationsstile, Datenformate und Deployment-Modell fest; kann manche Datenbanken oder Sprachen ausschließen.
Organisatorisch	Drei Teams in zwei Zeitzonen; das Betriebsteam unterstützt nur PostgreSQL; das Sicherheitsteam muss jede externe Abhängigkeit freigeben.	Prägt, wie das System aufgeteilt wird (Gesetz von Conway, Kapitel 3) und welche Technologien realistisch sind.
Regulatorisch und rechtlich	Personenbezogene Daten von Personen mit Wohnsitz in der EU werden nach der DSGVO verarbeitet; Kartendaten fallen unter PCI DSS; ein Medizinprodukt muss Sicherheitsvorschriften erfüllen.	Bestimmt Speicherort der Daten, Verschlüsselung, Audit-Logging, Aufbewahrungsregeln und die Trennung sensibler Komponenten.
Zeit	Die neuen Steuerregeln gelten ab einem festen Datum; eine Handelsplattform muss vor der Novemberspitze bereit sein.	Begünstigt bewährte Technologie und einfachere Designs; kann ein schrittweises Release mit bekannten technischen Schulden erzwingen.
Budget	Die Betriebskosten müssen unter einem festen Monatsbetrag bleiben; kein Geld für kommerzielle Lizenzen.	Begrenzt Redundanz, Regionen und Managed Services; macht die Kosten pro Transaktion zu einer ausdrücklichen Qualität.

Zwei Gewohnheiten helfen. Erstens: Notieren Sie die Quelle jeder Randbedingung (eine Vorschrift, einen Vertrag, eine namentlich genannte entscheidende Person), damit sie geprüft und bei veränderten Umständen neu bewertet werden kann. Zweitens: Unterscheiden Sie echte Randbedingungen von Vorlieben, die sich als Randbedingungen ausgeben. „Wir müssen Microservices verwenden“ ist selten eine Randbedingung; es ist eine Designentscheidung, die jemand früh getroffen hat, und sie sollte wie jede andere Entscheidung durch Qualitätsmerkmale begründet werden. Der Abschnitt in Kapitel 1 über Entscheidungen und Abwägungen erklärt, warum frühe, schwer umkehrbare Entscheidungen diese Prüfung verdienen.

Architekturrelevante Anforderungen

Die meisten Anforderungen wirken sich nicht auf die Architektur aus. Ob auf der Rechnung die Postleitzahl des Kunden über oder unter dem Straßennamen steht, ist ein Detail; das lässt sich an einem Nachmittag ändern. Eine architekturrelevante Anforderung (ASR) ist eine, die eine messbare Wirkung auf die Architektur hat: Würde sie sich ändern, müsste sich die Struktur des Systems ändern. ASRs früh zu erkennen ist wichtig, weil es teuer ist, sie spät zu berücksichtigen. Einer Ticketseite eine Woche vor dem Verkauf einen Warteraum hinzuzufügen ist möglich; den Sitzplatzbestand nach dem Start über Server zu partitionieren ist eine Neuentwicklung.

Lianping Chen, Muhammad Ali Babar und Bashar Nuseibeh befragten Praktiker, wie sie ASRs erkennen (*Characterizing Architecturally Significant Requirements*, IEEE Software, 2013). Ihre Ergebnisse decken sich mit der Erfahrung der meisten Architekten. ASRs sind meist schwer zu erfüllen, betreffen oft das System als Ganzes statt eines einzelnen Features, stehen häufig miteinander im Konflikt und werden oft vage oder gar nicht formuliert. Qualitätsmerkmale und Randbedingungen machen die Mehrheit aus, aber auch manche funktionalen Anforderungen sind relevant.

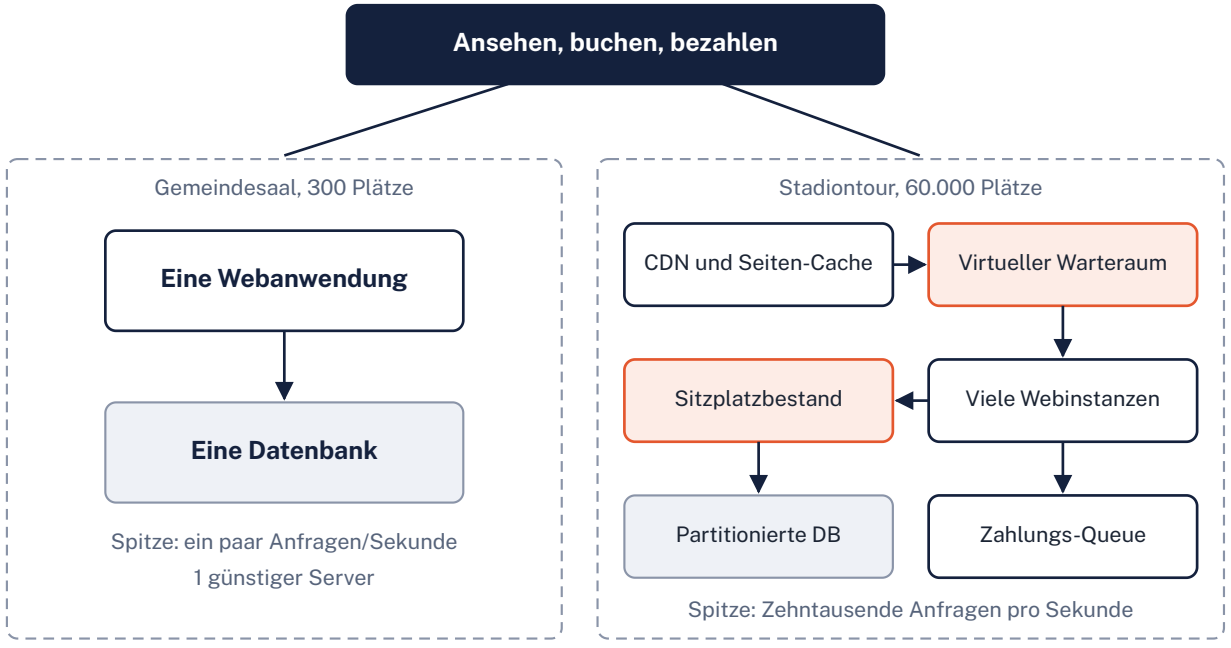
Beispiele für funktionale ASRs: „Jede Änderung an einer Patientenakte muss einer Person und einem Zeitpunkt zuordenbar sein“ (das erfordert einen Audit-Mechanismus über alle Komponenten), „Nutzer können in Echtzeit am selben Dokument zusammenarbeiten“ (das verlangt ein Synchronisationsmodell und wahrscheinlich einen anderen Kommunikationsstil) und „Kunden können eine Überweisung innerhalb von 30 Minuten rückgängig machen“ (das betrifft, wie Geld zwischen Konten bewegt wird). Jede davon erzwingt eine strukturelle Entscheidung, nicht bloß mehr Code.

Warum Qualitäten die Architektur stärker prägen als Funktionen

Hier ist der zentrale Gedanke dieses Kapitels. Käme es nur auf die Funktionalität an, wäre fast jede Struktur recht: Man könnte das ganze System als ein großes Programm schreiben, und es würde trotzdem die richtigen Ergebnisse berechnen. Bass, Clements und Kazman sagen das direkt: Funktionalität bestimmt die Architektur nicht, weil viele verschiedene Architekturen dieselben Funktionen liefern können. Was sie unterscheidet, ist, wie gut jede die Qualitäten erreicht.

Betrachten Sie zwei Ticketsysteme mit identischen Features: Veranstaltungen ansehen, Plätze reservieren, bezahlen. Das eine verkauft Tickets für einen Gemeindesaal mit 300 Plätzen, das andere eine Stadiontournee mit 60.000 Plätzen. Der geschäftigste Moment des Gemeindesaals bringt vielleicht ein paar Anfragen pro Sekunde, und eine einzelne Webanwendung mit einer Datenbank ist die richtige Antwort: günstig, einfach und leicht zu ändern. Die Stadiontournee muss einen tausendfach größeren Ansturm abfangen, darf nie denselben Platz zweimal verkaufen und muss verhindern, dass Bots in der ersten Sekunde alles aufkaufen. Dieses System braucht ein Content Delivery Network, eine Warteschlange vor dem Shop, einen sorgfältig entworfenen Sitzplatzbestand, partitionierte Speicherung und einen Zahlungsablauf, der auf einen überlasteten Anbieter warten kann. Gleiche Funktionen, völlig verschiedene Architekturen, und der Unterschied kommt allein von Qualitätsmerkmalen und Randbedingungen.

Gleiche Funktionen, unterschiedliche Qualitäten



Identische Features können zu radikal verschiedenen Strukturen führen, wenn sich die Anforderungen an Last, Konsistenz und Sicherheit unterscheiden.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: Ist diese Anforderung relevant?

Der schwierigste Gedanke dieses Abschnitts ist, zu erkennen, welche Anforderungen architekturrelevant sind (ASRs). Der Test ist nicht, ob eine Anforderung Funktion oder Qualität heißt, sondern ob ihre Änderung die Struktur des Systems ändern würde. Probieren Sie den Test an drei Anforderungen aus.

- Der Test: Stellen Sie sich vor, die Anforderung ändert sich (die Zahl verdoppelt oder gekürzt, die Regel verschärft oder gestrichen), und fragen Sie, was sich bewegen müsste: Komponenten, Daten oder die Wege zwischen ihnen.
- Eine Rechnung zeigt die Postleitzahl des Kunden unter dem Straßennamen. Verschieben Sie sie nach oben, ändert sich an einem Nachmittag eine Vorlage. Sonst bewegt sich nichts, also ist das Feinentwurf, keine Architektur.
- Die Stadiontournee muss einen tausendfach größeren Ansturm abfangen als die paar Anfragen pro Sekunde des Gemeindesaals. Dieser Ansturm ist ein großer Teil des Grundes, warum sie ein Content Delivery Network, einen Warteraum und partitionierte Speicherung braucht. Senkt man die Last auf die des Gemeindesaals, könnte der größte Teil dieser Maschinerie wegfallen; der Saal kommt mit einer Webanwendung und einer Datenbank aus.
- Die Lastzahl ist also eine ASR. Relevant macht sie ihre Wirkung auf die Struktur, nicht ihre Einordnung als Qualität.
- Auch Funktionen können den Test bestehen. Nehmen Sie die Regel, dass jede Bearbeitung einer Patientenakte zeigen muss, wer sie wann vorgenommen hat. Sie klingt wie ein Feature, braucht aber einen Audit-Mechanismus über alle Komponenten. Fällt sie weg, fällt dieser Mechanismus weg, also ist sie relevant.
- **Die Lehre:** Fragen Sie „Was würde diese Änderung bewegen?“, nicht „Was für eine Anforderung ist das?“. Die meisten Anforderungen jeder Art bleiben innerhalb eines Teils. Die wenigen, die Komponenten, Daten oder die Wege zwischen ihnen bewegen würden, sind diejenigen, um die herum die Architektur gebaut werden muss.

Implizite Qualitäten, die niemand aufschreibt

Einige der wichtigsten Qualitäten werden nie formuliert, weil alle Stakeholder sie voraussetzen. Diese impliziten Anforderungen treten erst zutage, wenn sie verletzt werden, meist im Produktivbetrieb. Niemand schreibt „keine Bestellungen verlieren“, „dem Kunden nicht doppelt abbuchen“, „bei der Zeitumstellung weiterlaufen“, „auf dem Smartphone und der Website denselben Kontostand zeigen“ oder „die Daten eines Kunden nicht an einen anderen

preisgeben“. Doch jede davon ist eine ASR mit schweren Folgen: idempotente Zahlungsverarbeitung, dauerhafte Schreibvorgänge, zeitzonensichere Terminplanung, Konsistenz zwischen den Kanälen und strikte Mandantentrennung.

Implizite Anforderungen bleiben aus sozialen Gründen verborgen. Fachliche Stakeholder halten sie für zu selbstverständlich, um sie zu erwähnen; Entwickler nehmen an, jemand anderes habe sich darum gekümmert; und Anforderungsvorlagen fragen selten danach. Nützliche Fragen, um sie aufzuspüren, sind: „Was würde Sie rasend machen, wenn es passierte?“, „Was würde uns in die Zeitung bringen?“, „Was ist bei dem System, das wir ablösen, schiefgegangen?“ und „Wofür wird das Betriebsteam nachts geweckt?“. Vorfallberichte und Support-Tickets von Vorgängersystemen sind eine reiche Quelle, ebenso die veröffentlichten Leitlinien von Aufsichtsbehörden und die Pflichten aus Kundenverträgen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein Konzerttickets ystem am Verkaufstag

Ein Ticketunternehmen verkauft an einem Freitag um 10:00 Uhr vier Abende in einem Stadion (60.000 Plätze pro Abend). An einem normalen Tag hat die Website etwa 2000 Besucher pro Stunde. Das Marketing erwartet in den ersten zehn Minuten 1,5 Millionen Menschen. Die Features sind die üblichen: ansehen, Plätze wählen, sie zehn Minuten reservieren, bezahlen. Was sind die architekturrelevanten Anforderungen?

Tickets im Verkauf	4 Abende × 60.000 Plätze = 240.000 Tickets
Normale Ankunftsrate	2000 Besucher ÷ 3600 s ≈ 0,56 Besucher pro Sekunde
Ankunftsrate am Verkaufstag	1.500.000 Besucher ÷ 600 s = 2500 Besucher pro Sekunde, etwa das 4500-Fache der normalen Rate
Anfragerate	Wenn der Browser jedes Besuchers in der ersten Minute etwa 8 Anfragen an die Website stellt (Seiten, Sitzpläne, API-Aufrufe): 2500 × 8 = 20.000 Anfragen pro Sekunde
Nachfrage und Angebot	1.500.000 Besucher für 240.000 Tickets: Die meisten werden unabhängig vom Design enttäuscht, daher zählen Fairness und klare Kommunikation ebenso viel wie Geschwindigkeit
Randbedingung Zahlung	Der Vertrag mit dem Zahlungsanbieter erlaubt 300 Zahlungsautorisierungen pro Sekunde. Bei durchschnittlich 2,5 Tickets pro Bestellung ergeben 240.000 ÷ 2,5 = 96.000 Bestellungen; 96.000 ÷ 300 = 320 s, die Zahlungen allein dauern also mindestens 5,3 Minuten, selbst wenn alles andere sofort geschieht
ASR 1: Spitzenlast (Performance)	2500 Ankünfte pro Sekunde abfangen, ohne zusammenzubrechen; statische Seiten und Sitzpläne aus einem Cache oder CDN ausliefern, damit nur echte Käufe den Kern erreichen
ASR 2: kein Platz doppelt verkauft (Konsistenz)	Platzreservierungen müssen atomar sein, auch wenn Tausende gleichzeitig auf denselben Block klicken; das prägt das Design des Bestands und seiner Speicherung
ASR 3: Fairness (Geschäftsregel mit struktureller Wirkung)	Käufer in der Reihenfolge ihrer Ankunft über einen virtuellen Warteraum einlassen, damit das Kernsystem einen kontrollierten Strom statt eines Ansturms sieht
ASR 4: Bot-Resistenz (Sicherheit)	Automatisierte Käufer erkennen und blockieren, bevor sie den Bestand erreichen; Rate Limiting pro Konto und pro Gerät
ASR 5: Graceful Degradation (Verfügbarkeit)	Wenn Zahlungen langsamer werden, Reservierungen gültig halten und Zahlungsversuche in eine Warteschlange stellen, statt ganze Bestellungen scheitern zu lassen

Randbedingung	Fester, vorab angekündigter Termin; Limit des Zahlungsanbieters von 300 pro Sekunde

Keine dieser ASRs steht in der Feature-Liste. Alle stammen aus Last, Konsistenz, Fairness, Sicherheit und einer Randbedingung eines Drittanbieters.

Das Zahlungslimit wird direkt zu einer Designentscheidung: Da nicht alle Käufer gleichzeitig zahlen können, muss das System Plätze reservieren und Zahlungen in eine Warteschlange stellen, was wiederum die Reservierungsdauer bestimmt.

Die Realität liefert eine Mahnung. Im November 2022 litt der Vorverkauf von Ticketmaster für eine große Stadiontournee unter der Nachfrage an langen Verzögerungen und Ausfällen; das Unternehmen meldete insgesamt 3,5 Milliarden Systemanfragen, das Vierfache seiner bisherigen Spitze.

Checkliste: architekturrelevante Anforderungen erkennen

Gehen Sie diese Fragen früh im Projekt für jede Kandidatenanforderung und für das System als Ganzes durch.

- [ ] **Würde ihre Änderung die Struktur ändern?**
  Wenn eine verdoppelte Zahl, eine gelockerte Regel oder eine gestrichene Anforderung Komponenten, Daten oder Kommunikationswege bewegen würde, ist sie relevant.
- [ ] **Gilt sie für viele Features?**
  Anforderungen, die quer durch das ganze System gehen (Audit, Sicherheit, Verfügbarkeit, Internationalisierung), brauchen meist eine strukturelle Antwort.
- [ ] **Ist sie schwierig oder neu?**
  Eine anspruchsvolle Zahl (unter 100 ms, 99,99 %), eine Last weit über allem, was das Team bisher bewältigt hat, oder ein Feature ohne Vorbild ist ein starkes Signal.
- [ ] **Hat sie hohen Geschäftswert oder hohes Risiko?**
  Wenn ihr Scheitern Umsatz kosten, gegen das Gesetz verstoßen oder jemandem schaden würde, behandeln Sie sie als relevant, auch wenn sie einfach aussieht.
- [ ] **Steht sie im Konflikt mit einer anderen Qualität?**
  Anforderungen im Spannungsverhältnis (schnell und stark konsistent, offen und sicher) brauchen bewusste Abwägungen, und das sind Architekturentscheidungen.
- [ ] **Ist sie eine verkappte Randbedingung?**
  Vorgegebene Technologien, Vorschriften, Termine und Budgets entfernen Optionen; halten Sie sie mit ihrer Quelle fest.
- [ ] **Hat sie schon einmal wehgetan?**
  Sehen Sie sich Vorfälle, Support-Tickets und Post-Mortems des abzulösenden Systems und ähnlicher Systeme anderswo an.
- [ ] **Setzt jemand sie stillschweigend voraus?**
  Fragen Sie, was die Stakeholder rasend machen oder das Unternehmen in die Nachrichten bringen würde; diese Antworten decken implizite Anforderungen auf.

ACHTUNG

Häufige Fehler

- Hunderte User Stories sammeln und eine einzige Zeile mit „Das System muss schnell, sicher und skalierbar sein“.
- Jede Anforderung als relevant behandeln. Wenn alles eine ASR ist, ist nichts priorisiert; die meisten Anforderungen gehören in den Feinentwurf.
- Eine Designentscheidung („Kafka verwenden“, „Microservices verwenden“) wie eine Randbedingung hinnehmen, ohne ein Qualitätsmerkmal, das sie begründet.
- Qualitäten ohne Maß formulieren. Auf „hochverfügbar“ kann man weder hin entwerfen noch es testen; Abschnitt 2.3 zeigt, wie man das behebt.
- Annehmen, Qualität lasse sich ergänzen, nachdem die Features gebaut sind. Performance, Sicherheit und Verfügbarkeit hängen von der Struktur ab und sind teuer nachzurüsten.

Wie Fachleute mit ASRs arbeiten

Erfahrene Architekten führen eine kurze, lebendige Liste von ASRs, typischerweise fünf bis fünfzehn Einträge, statt eines langen Anforderungsdokuments. Jeder Eintrag formuliert die Qualität als testbares Szenario (Abschnitt 2.3), nennt den Stakeholder, dem sie wichtig ist, hält ihre Priorität fest und verweist auf die Entscheidungen, die sie ausgelöst hat. Die arc42-Vorlage zum Beispiel verlangt auf der ersten Seite die drei bis fünf wichtigsten Qualitätsziele und später eine ausführlichere Liste von Qualitätsanforderungen. Die Liste wird immer dann überprüft, wenn eine große Entscheidung fällt oder das Unternehmen die Richtung ändert, und die wichtigsten Einträge werden schließlich durch automatisierte Prüfungen abgesichert, sogenannte Fitness Functions, die in Abschnitt 2.8 vorgestellt werden.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Ist „Patienten können online Termine buchen“ eine funktionale Anforderung, ein Qualitätsmerkmal oder eine Randbedingung?

Eine funktionale Anforderung: Sie sagt, was das System tun muss. Eine Qualität würde ergänzen, wie gut, etwa wie schnell Buchungen bestätigt werden oder wie viele sich montags um 8 Uhr bewältigen lassen.

▼ **Warum ist „Das System muss skalierbar sein“ noch keine nützliche Anforderung?**

Sie hat keinen Stimulus, keinen Kontext und kein Maß. Sie sagt nicht, wie viel Last, wie schnell die Last wächst oder welche Antwort erhalten bleiben muss, also kann niemand darauf hin entwerfen oder sie testen.

▼ **Nennen Sie ein Beispiel für eine funktionale Anforderung, die architekturrelevant ist.**

Zum Beispiel ein vollständiger Audit-Trail jeder Änderung an Patientenakten oder kollaboratives Bearbeiten in Echtzeit. Beides braucht einen strukturellen Mechanismus im ganzen System, nicht nur mehr Code an einer Stelle.

▼ **Welche Frage trennt eine relevante Anforderung am besten von einer gewöhnlichen?**

Müsste sich die Struktur des Systems ändern, wenn sich diese Anforderung änderte? Wenn ja, ist sie architekturrelevant.

▼ **Probieren Sie es aus: Eine Ticketing-Aufgabenstellung lautet: Fans können Plätze auswählen; Bestätigung innerhalb von 2 Sekunden zur Spitzenzeit am Vorverkaufstag; das Hosting muss beim aktuellen Cloud-Anbieter bleiben; niemandem wird zweimal etwas berechnet. Ordnen Sie jeden Punkt ein, und sagen Sie, welche architekturrelevant sind.**

Die Platzwahl ist funktional; sie ist wichtig, entscheidet die Struktur aber kaum. Die Bestätigung innerhalb von 2 Sekunden zur Spitzenzeit ist ein Qualitätsmerkmal und klar relevant, da sie Warteschlangen und Kapazität bestimmt. Die Hosting-Regel ist eine Randbedingung, eine bereits getroffene Entscheidung, und sie ist relevant, weil sie die Optionen einschränkt. Dass niemand doppelt belastet wird, ist eine implizite Qualität, meist unausgesprochen, und hochrelevant: Sie prägt die Idempotenz und die Art, wie Zahlungen erfasst werden.

QUELLEN

- Martin Glinz, On Non-Functional Requirements (RE 2007): https://files.ifi.uzh.ch/rerg/amadeus/publications/papers/RE07_Glinz.pdf
- Chen, Ali Babar and Nuseibeh, Characterizing Architecturally Significant Requirements (IEEE Software, 2013): https://pure.itu.dk/en/publications/characterizing-architecturally-significant-requirements/
- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- arc42: Introduction and goals (quality goals): https://docs.arc42.org/section-1/

Qualitätsmodelle: ISO/IEC 25010 und wie man es nutzt

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.5 Stakeholder und Anliegen
- 2.1 Funktionale und Qualitätsanforderungen

Begriffe von früher, die er verwendet: Qualitätsmerkmal, Stakeholder, Anforderung

DIE IDEE IN KÜRZE

Wenn Sie einen Gebrauchtwagen kaufen, sorgt eine gute Checkliste dafür, dass Sie nichts vergessen: Motor, Bremsen, Reifen, Rost, Papiere, Verbrauch. Nicht jeder Punkt muss perfekt sein, aber Sie wollen sich jeden angesehen und entschieden haben, ob er Ihnen wichtig ist. Ein Qualitätsmodell ist diese Checkliste für Software. Die internationale Norm ISO/IEC 25010 nennt neun Bereiche, etwa Performance, Zuverlässigkeit, Sicherheit und wie leicht sich die Software ändern lässt, damit nichts Wichtiges übersehen wird. Die Kunst besteht darin, sie für gute Fragen zu nutzen, nicht zum Ausfüllen von Formularen.

WAS SIE WISSEN MÜSSEN

- Ein Qualitätsmodell bietet ein gemeinsames Vokabular aus Qualitätsmerkmalen und Teilmerkmalen und dient als Checkliste, um die Qualitäten zu finden, auf die es ankommt.
- ISO/IEC 25010:2023 definiert neun Produktmerkmale: funktionale Eignung, Leistungseffizienz, Kompatibilität, Interaktionsfähigkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit, Flexibilität und Betriebssicherheit.
- Gegenüber 2011 wurde aus Benutzbarkeit die Interaktionsfähigkeit, aus Übertragbarkeit die Flexibilität, Betriebssicherheit kam hinzu, und die Nutzungsqualität wanderte in eine eigene Norm, ISO/IEC 25019:2023.
- Andere Listen wie FURPS+, die Qualitäten in Software Architecture in Practice und Q42 überschneiden sich stark mit ISO/IEC 25010; keine ist die einzig richtige.
- Nutzen Sie ein Modell als Anstoß für eine schnelle, ausdrückliche Entscheidung zu jedem Merkmal, und machen Sie dann nur aus den relevanten einige messbare, priorisierte Szenarien.

Wozu ein Qualitätsmodell dient

Abschnitt 2.1 hat dargelegt, dass Qualitätsmerkmale den größten Teil einer Architektur bestimmen. Die naheliegende nächste Frage lautet: welche Qualitäten? Sich selbst überlassen, nennen Stakeholder die zwei oder drei, die ihnen zuletzt wehgetan haben, meist Geschwindigkeit und Verfügbarkeit, und vergessen den Rest, bis es zu spät ist. Ein Qualitätsmodell ist eine strukturierte Liste von Qualitätsmerkmalen, jeweils in engere Teilmerkmale gegliedert, mit vereinbarten Namen und Definitionen. Es erfüllt zwei Aufgaben. Es gibt allen ein gemeinsames Vokabular, sodass „Zuverlässigkeit“ für den Product Owner, das Testteam und den Lieferanten dasselbe bedeutet. Und es dient als Checkliste, die Sie auf Qualitäten aufmerksam macht, die noch niemand angesprochen hat.

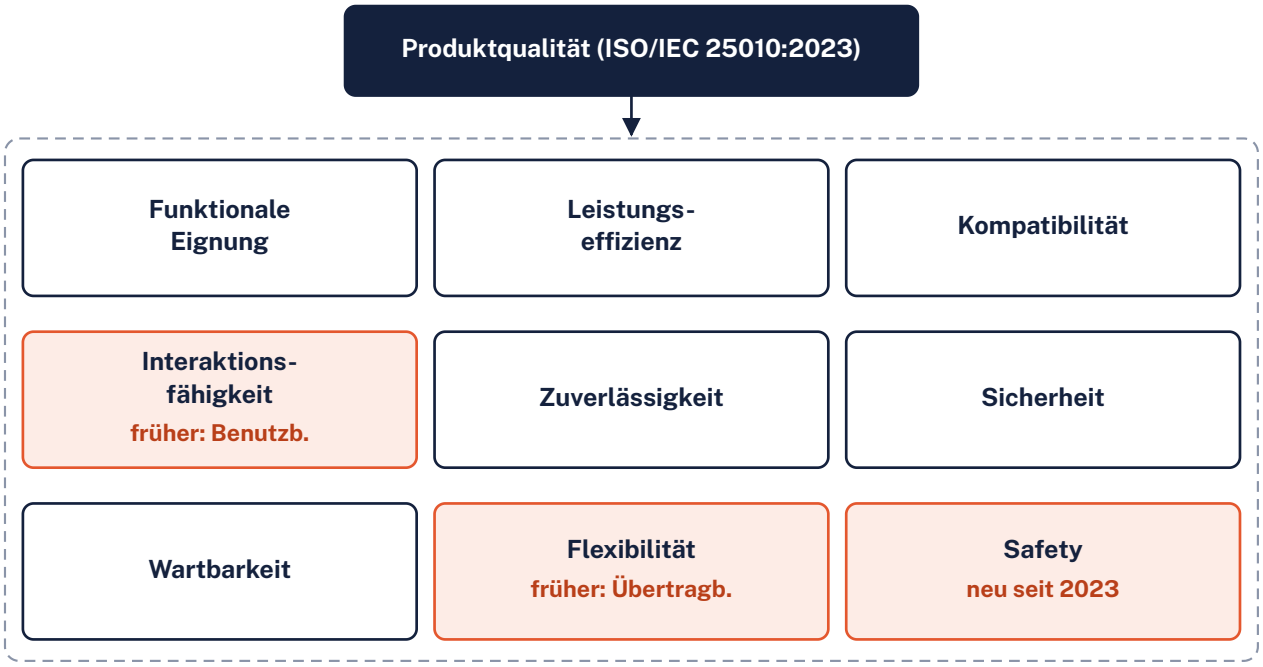
Ein Modell sagt Ihnen nicht, welche Qualitäten für Ihr System wichtig sind oder wie viel Sie von jeder brauchen. Das ergibt sich aus Stakeholdern und Geschäftszielen (Abschnitt 2.8). Es sagt Ihnen auch nicht, wie Sie sie erreichen; das ist die Aufgabe von Taktiken, Mustern und dem Rest dieses Kurses. Es ist eine Karte des Geländes, kein Weg hindurch.

ISO/IEC 25010:2023: das Produktqualitätsmodell

Das am häufigsten zitierte Modell ist ISO/IEC 25010, Teil der Normenfamilie SQuaRE (Systems and software Quality Requirements and Evaluation). Die zweite Ausgabe, veröffentlicht im November 2023, definiert ein Produktqualitätsmodell mit neun Merkmalen. Es gilt für Softwareprodukte und für ICT-Produkte im weiteren Sinn, und die Norm nennt seine Einsatzzwecke: Anforderungen spezifizieren, Anforderungen auf Vollständigkeit prüfen, Design- und Testziele festlegen, Abnahmekriterien definieren und Qualitätsmaße festlegen. Verwandte Normen der Familie legen fest, wie jedes Teilmerkmal gemessen (ISO/IEC 25023) und wie ein Produkt bewertet wird.

Die neun Merkmale sind funktionale Eignung, Leistungseffizienz, Kompatibilität, Interaktionsfähigkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit, Flexibilität und Betriebssicherheit. Das erste beschreibt, ob das System die richtigen Dinge korrekt tut; die anderen acht beschreiben, wie gut es sie tut und wie gut es sich betreiben, schützen und ändern lässt. Das Diagramm unten zeigt die Struktur; die Tabelle danach listet jedes Teilmerkmal mit einer Beispielfrage auf, die ein Architekt stellen könnte.

Die neun Merkmale



ISO/IEC 25010:2023 gliedert die Produktqualität in neun Merkmale; gegenüber der Ausgabe von 2011 wurden zwei umbenannt und eines hinzugefügt.

Merkmale, Teilmerkmale und eine Beispielfrage für eine Mobile-Banking-App

Merkmal	Teilmerkmale	Beispielfrage
Funktionale Eignung	Funktionale Vollständigkeit, funktionale Korrektheit, funktionale Angemessenheit	Berechnet die App die Kreditrate korrekt, und leistet sie alles, was der Kunde braucht, um die Aufgabe zu erledigen?
Leistungseffizienz	Zeitverhalten, Ressourcennutzung, Kapazität	Wie schnell erscheint der Kontostand auf einem Mittelklasse-Smartphone über 4G, und wie viele gleichzeitige Nutzer kann das Backend bedienen?
Kompatibilität	Koexistenz, Interoperabilität	Kann die App Zahlungsdaten mit dem nationalen Echtzeitzahlungsverfahren austauschen und neben anderen Apps laufen, ohne gemeinsame Ressourcen aufzubrauchen?
Interaktionsfähigkeit (früher Benutzbarkeit)	Erkennbarkeit der Angemessenheit, Erlernbarkeit, Bedienbarkeit, Schutz vor Benutzerfehlern, Nutzerbindung, Inklusivität, Benutzerunterstützung, Selbstbeschreibungsfähigkeit	Kann ein Erstnutzer ohne Hilfe einen Dauerauftrag einrichten, und kann jemand mit Screenreader dasselbe?
Zuverlässigkeit	Fehlerfreiheit, Verfügbarkeit, Fehlertoleranz, Wiederherstellbarkeit	Wenn der Kartenabwicklungspartner ausfällt, können Kunden dann weiterhin Kontostände sehen und Geld zwischen eigenen Konten bewegen?
Sicherheit	Vertraulichkeit, Integrität, Nichtabstreitbarkeit, Verantwortlichkeit, Authentizität, Widerstandsfähigkeit	Können wir nachweisen, welcher Kunde eine Überweisung autorisiert hat, und widersteht die App Credential-Stuffing-Angriffen?
Wartbarkeit	Modularität, Wiederverwendbarkeit, Analysierbarkeit, Änderbarkeit, Testbarkeit	Wie lange dauert es, eine neue Kontoart hinzuzufügen, und lassen sich die Überweisungsregeln ohne echtes Kernbankensystem testen?
Flexibilität (früher Übertragbarkeit)	Anpassbarkeit, Skalierbarkeit, Installierbarkeit, Austauschbarkeit	Kann das Backend von 1 auf 5 Millionen Kunden wachsen, und könnten wir den Push-Benachrichtigungsanbieter innerhalb eines Quartals austauschen?

Merkmal	Teilmerkmale	Beispielfrage
Betriebssicherheit (Safety)	Betriebliche Einschränkung, Risikoerkennung, Ausfallsicherheit (Fail Safe), Gefahrenwarnung, sichere Integration	Vor allem für Systeme relevant, die Menschen, Sachen oder der Umwelt schaden können: eine Infusionspumpe, eine Zugsteuerung, ein Industrieroboter.

Einige Definitionen sollte man genau kennen, weil sie vom Alltagsgebrauch abweichen. In ISO/IEC 25010 ist **Verfügbarkeit** ein Teilmerkmal der Zuverlässigkeit: der Grad, in dem ein System bei Bedarf betriebsbereit und zugänglich ist. **Fehlerfreiheit** ist der Grad, in dem ein System seine Funktionen im Normalbetrieb ohne Fehler erfüllt. **Kapazität** unter Leistungseffizienz ist die Höchstgrenze eines Parameters, etwa gleichzeitiger Nutzer oder Transaktionen pro Sekunde, bei der die Anforderungen noch erfüllt werden. **Skalierbarkeit** steht unter Flexibilität; darin zeigt sich eine Sicht auf Skalierung als Fähigkeit, sich an wechselnde Last anzupassen, statt als Frage roher Geschwindigkeit. **Widerstandsfähigkeit**, neu unter Sicherheit, ist die Fähigkeit, den Betrieb unter einem Angriff aufrechtzuerhalten. Die Abschnitte 2.4 bis 2.7 greifen diese Qualitäten vertieft auf; Abschnitt 2.5 zeigt, wie man Verfügbarkeit berechnet.

Was sich gegenüber der Ausgabe von 2011 geändert hat

Die erste Ausgabe, ISO/IEC 25010:2011, löste das ältere Modell ISO/IEC 9126 ab und war ein Jahrzehnt lang die Standardreferenz. Viele Lehrbücher, Ausschreibungsunterlagen und Vorlagen verwenden noch ihre acht Merkmale, daher lohnt es sich zu wissen, was sich verschoben hat.

ISO/IEC 25010: 2011 im Vergleich mit 2023

Thema	Ausgabe 2011	Ausgabe 2023
Anzahl der Merkmale	Acht	Neun
Benutzbarkeit	Benutzbarkeit, einschließlich Ästhetik der Benutzungsoberfläche und Barrierefreiheit	Umbenannt in Interaktionsfähigkeit; Ästhetik ersetzt durch Nutzerbindung; Barrierefreiheit aufgeteilt in Inklusivität und Benutzerunterstützung; Selbstbeschreibungsfähigkeit hinzugefügt
Übertragbarkeit	Übertragbarkeit: Anpassbarkeit, Installierbarkeit, Austauschbarkeit	Umbenannt in Flexibilität; Skalierbarkeit als Teilmerkmal hinzugefügt
Zuverlässigkeit	Reife, Verfügbarkeit, Fehlertoleranz, Wiederherstellbarkeit	Reife ersetzt durch Fehlerfreiheit
Sicherheit	Vertraulichkeit, Integrität, Nichtabstreitbarkeit, Verantwortlichkeit, Authentizität	Widerstandsfähigkeit hinzugefügt (einem Angriff standhalten)
Betriebssicherheit (Safety)	Nur innerhalb der Nutzungsqualität behandelt (Risikofreiheit)	Neues Produktmerkmal mit fünf Teilmerkmalen
Nutzungsqualität	Teil von ISO/IEC 25010 (Effektivität, Effizienz, Zufriedenheit, Risikofreiheit, Kontextabdeckung)	In eine eigene Norm verschoben, ISO/IEC 25019:2023 (Nutzen, Risikofreiheit, Akzeptanz)

Die Umbenennungen sind mehr als Kosmetik. „Interaktionsfähigkeit“ weitet den Blick von der einfachen Bedienung auf den gesamten Informationsaustausch zwischen Nutzern und Produkt in verschiedenen Nutzungskontexten, und das neue Teilmerkmal Inklusivität rückt Barrierefreiheit für Menschen unterschiedlichen Alters und unterschiedlicher Fähigkeiten vom Rand ins Zentrum. „Flexibilität“ bündelt die Qualitäten, mit denen ein Produkt Veränderungen in seiner Umgebung bewältigt: neue Plattformen, neue Installationsziele und mehr oder weniger Last. Dass Betriebssicherheit als Produktmerkmal hinzukam, spiegelt wider, wie viel Software heute physische Dinge steuert, von Autos bis zu Medizinprodukten.

Die zweite strukturelle Änderung: Die Nutzungsqualität, also die Qualität, die Menschen bei der Nutzung des Produkts in einem realen Kontext erleben, hat nun eine eigene Norm, ISO/IEC 25019:2023. Ihre drei Merkmale sind Nutzen (Benutzbarkeit, Barrierefreiheit und Eignung in der Nutzung), Risikofreiheit (wirtschaftliche Risiken, Risiken für Gesundheit und Menschenleben sowie Umwelt- und gesellschaftliche Risiken) und Akzeptanz (Erfahrung,

Vertrauenswürdigkeit und Konformität). Die Trennung verdeutlicht eine für Architekten nützliche Unterscheidung: Produktqualität ist das, was Sie bauen und am Produkt testen können; Nutzungsqualität ist das Ergebnis für echte Menschen, das auch von Schulung, Kontext und der Art abhängt, wie das Produkt eingesetzt wird.

ACHTUNG

Achten Sie auf die Ausgabe

Wenn ein Vertrag oder die Vorlage eines Kunden „ISO 25010“ nennt, prüfen Sie, welche Ausgabe gemeint ist. Eine Anforderung, die „Benutzbarkeit und Übertragbarkeit“ aufführt, beruht fast sicher auf dem Modell von 2011. Das ist an sich kein Problem, aber beide Listen in einem Dokument zu mischen erzeugt Lücken (Betriebssicherheit und Widerstandsfähigkeit werden leicht übersehen) und Dopplungen (Barrierefreiheit taucht zweimal auf). Wählen Sie eine Ausgabe und sagen Sie das.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: was Sie bauen und was Menschen bekommen

Die Idee, die hier die meiste Sorgfalt verlangt, ist der Unterschied zwischen Produktqualität und Nutzungsqualität. Produktqualität ist eine Eigenschaft der Software, die das Team bauen und testen kann. Nutzungsqualität ist das, was echte Menschen tatsächlich davon haben, und das hängt auch davon ab, wer sie sind und in welcher Situation sie sich befinden. Die Mobile-Banking-App zeigt den Unterschied.

- Unter Schutz vor Benutzerfehlern, einem Teil der Interaktionsfähigkeit, schreibt das Team der Bank ein Szenario: Die App warnt den Kunden vor einer Zahlung an einen Empfänger, dessen Name nicht zum Konto passt.
- Das ist Produktqualität. Das Testteam sendet Zahlungen mit passenden und nicht passenden Namen und prüft, dass die Warnung jedes Mal erscheint, wenn sie soll.
- Angenommen, sie besteht jeden Test. Die Bank will trotzdem etwas anderes wissen: Verlieren Kunden tatsächlich weniger Geld durch irrtümliche oder betrügerische Zahlungen? Das ist Risikofreiheit, ein Teil der Nutzungsqualität, die seit 2023 eine eigene Norm hat, ISO/IEC 25019.
- Die Antwort hängt von mehr als der App ab. Ein Kunde in Eile tippt die Warnung vielleicht einfach weg; ein Betrüger am Telefon überredet ihn womöglich, sie zu ignorieren; ein Kunde, der von solchen Betrugsmaschen gehört hat, hält vielleicht inne und prüft. Die Software ist jedes Mal dieselbe; das Ergebnis nicht.
- Deshalb werden die beiden unterschiedlich geprüft: Produktqualität mit Tests am Produkt, Nutzungsqualität durch Beobachtung realer Ergebnisse, etwa wie viele Kunden melden, an die falsche Person gezahlt zu haben.
- **Die Lehre:** Produktqualität kann das Team direkt entwerfen und testen; Nutzungsqualität zeigt sich erst in der realen Nutzung. Gute Produktqualität macht gute Nutzungsqualität wahrscheinlicher, kann sie allein aber nicht garantieren.

Andere Listen, denen Sie begegnen werden

ISO/IEC 25010 ist nicht die einzige Taxonomie, und in der Praxis werden oft einfachere oder anders geschnittene Listen verwendet. FURPS+ ging aus FURPS hervor, einer bei Hewlett-Packard entwickelten Klassifikation, die Robert Grady und Deborah Caswell in *Software Metrics: Establishing a Company-Wide Program* (1987) darlegten. FURPS gruppiert Anforderungen in Funktionalität (Functionality), Benutzbarkeit (Usability), Zuverlässigkeit (Reliability), Performance und Supportfähigkeit (Supportability); die spätere Erweiterung „+“ ergänzt Randbedingungen: Design-, Implementierungs-, Schnittstellen- und physische Randbedingungen sowie in manchen Fassungen Geschäftsregeln. Bekannt wurde FURPS+

durch den Rational Unified Process und seine Nachfolger. Seine Stärke ist die Kürze; seine Schwäche, dass „Supportfähigkeit“ Wartbarkeit, Testbarkeit, Konfigurierbarkeit, Installierbarkeit und Lokalisierung in einen Topf wirft.

Bass, Clements und Kazman erheben in der vierten Auflage von *Software Architecture in Practice* (2021) keinen Anspruch auf Vollständigkeit. Sie widmen je ein Kapitel den Qualitäten, die Architekturen am häufigsten treiben: Verfügbarkeit, Deploybarkeit, Energieeffizienz, Integrierbarkeit, Änderbarkeit, Performance, Betriebssicherheit, Sicherheit, Testbarkeit und Benutzbarkeit, gefolgt von einem Kapitel über den Umgang mit weiteren Qualitäten. Zu jeder geben sie ein allgemeines Szenario und einen Katalog von Taktiken an. Ihre Liste ist am Design statt an der Bewertung ausgerichtet, was sie für Architekten besonders nützlich macht.

Mark Richards und Neal Ford verwenden in *Fundamentals of Software Architecture* den Begriff Architekturmerkmale und ordnen sie in Gruppen wie betriebliche (Verfügbarkeit, Performance, Skalierbarkeit), strukturelle (Änderbarkeit, Erweiterbarkeit, Übertragbarkeit) und übergreifende (Sicherheit, Barrierefreiheit, Recht). Das offene Q42-Modell von Gernot Starke, gepflegt neben der arc42-Vorlage, wählt wieder einen anderen Ansatz: Es versieht weit über hundert benannte Qualitäten mit einer kurzen Liste von Dimensionen (etwa effizient, flexibel, betreibbar, zuverlässig, betriebssicher, sicher, geeignet und benutzbar) und bietet für jede Beispielanforderungen.

Diese Listen unterscheiden sich im Detail, überschneiden sich aber stark. Die Uneinigkeit ist selbst lehrreich: Es gibt keine einzig richtige Zerlegung von „Qualität“, nur für einen Zweck mehr oder weniger nützliche. Ein Standardmodell hilft, wenn Sie mit dem Einkauf, mit Prüfern oder anderen Organisationen kommunizieren müssen; einem Produktteam dient im Alltag vielleicht eine schlankere Liste besser.

Ein Modell als Checkliste nutzen, ohne Bürokratie zu schaffen

Die große Gefahr eines Qualitätsmodells ist, dass es zum Formular wird. Eine Anforderungsvorlage mit einer Überschrift für jedes der 40 Teilmerkmale verleitet dazu, jedes Feld zu füllen, und heraus kommt ein Dokument voller Aussagen wie „Das System soll wartbar sein“, das niemand liest und niemand testet. Der Aufwand verteilt sich gleichmäßig auf Qualitäten, die sehr ungleich wichtig sind, und die zwei oder drei Qualitäten, die die Architektur wirklich entscheiden, gehen im Rest unter.

Besser ist es, das Modell als Anstoß für ein Gespräch zu behandeln. Gehen Sie die Merkmale einmal zügig mit den Stakeholdern durch und fragen Sie bei jedem: Ist das für unser System wichtig, und wenn ja, wie sähe ein Versagen aus? Verwerfen Sie die irrelevanten ausdrücklich (zu schreiben „Betriebssicherheit: nicht zutreffend, die App kann keinen physischen Schaden

verursachen“ ist nützlich, weil es eine Entscheidung festhält). Machen Sie aus den relevanten eine kleine Zahl konkreter, messbarer Szenarien (Abschnitt 2.3). Gewichten Sie sie (Abschnitt 2.8). Dann legen Sie das Modell bis zum nächsten Review beiseite.

Ein schlanker Weg, das Modell anzuwenden

1. **Die neun Merkmale durchgehen**
   Lesen Sie in einer einstündigen Sitzung jedes Merkmal laut vor und fragen Sie die Stakeholder, ob es wichtig ist. Notieren Sie, wem es wichtig ist und warum.

2. **Relevanz markieren**
   Kennzeichnen Sie jedes Merkmal als hoch, mittel, niedrig oder nicht zutreffend, und halten Sie zu jeder Entscheidung einen Satz Begründung fest.

3. **Bei den hohen in die Tiefe gehen**
   Sehen Sie sich nur bei Merkmalen mit hoher Relevanz die Teilmerkmale an, um das genaue Anliegen zu finden, etwa Wiederherstellbarkeit statt Zuverlässigkeit im Allgemeinen.

4. **Szenarien schreiben**
   Formulieren Sie jedes genaue Anliegen als ein oder zwei messbare Szenarien mit Stimulus, Kontext und Antwortmaß.

5. **Priorisieren und verknüpfen**
   Gewichten Sie die Szenarien und verknüpfen Sie jedes mit den Designentscheidungen, die es treibt, damit die Liste kurz und lebendig bleibt.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ISO/IEC 25010 auf eine Mobile-Banking-App anwenden

Eine mittelgroße Bank löst ihre Mobile-App und ihr Backend ab. Sie hat 1,2 Millionen aktive App-Nutzer; am Abend des Zahltags melden sich etwa 25 % davon zwischen 18:00 und 20:00 Uhr an. Das Team geht die neun Merkmale in einer Stunde durch und hält das Ergebnis fest.

Schätzung der Spitzenlast	1.200.000 × 25 % = 300.000 Anmeldungen in 2 Stunden = 300.000 ÷ 7200 s ≈ 42 Anmeldungen pro Sekunde im Mittel; mit dem Faktor 3 für Lastspitzen ≈ 125 pro Sekunde
Funktionale Eignung: hoch	Zinsen, Gebühren und Kontostände müssen auf den Cent genau stimmen. Szenario: Jeder angezeigte Kontostand stimmt innerhalb von 5 Sekunden nach einer gebuchten Transaktion mit dem Hauptbuch des Kernsystems überein.
Leistungseffizienz: hoch	Szenario: Bei 125 Anmeldungen pro Sekunde wird der Startbildschirm mit den Kontoständen für 95 % der Nutzer über eine 4G-Verbindung innerhalb von 2 Sekunden angezeigt. Akku- und Datenverbrauch auf dem Smartphone zählen ebenfalls (Ressourcennutzung).
Kompatibilität: mittel	Muss mit dem nationalen Echtzeitzahlungsverfahren und den gesetzlich vorgeschriebenen Open-Banking-APIs interoperieren; muss auf den beiden neuesten Hauptversionen beider mobiler Betriebssysteme laufen.
Interaktionsfähigkeit: hoch	Inklusivität und Schutz vor Benutzerfehlern sind am wichtigsten. Szenario: Jemand mit Screenreader kann ohne sehende Hilfe eine Zahlung an einen neuen Empfänger abschließen; die App warnt vor einer Zahlung an einen Empfängernamen, der nicht zum Konto passt.
Zuverlässigkeit: hoch	Wiederherstellbarkeit und Fehlertoleranz. Szenario: Fällt die Kartenplattform aus, funktionieren Kontostände und Überweisungen zwischen eigenen Konten weiter; der volle Service ist innerhalb von 30 Minuten wiederhergestellt.
Sicherheit: hoch	Authentizität, Nichtabstreitbarkeit und Widerstandsfähigkeit. Szenario: Jede Zahlung wird mit starker Kundenauthentifizierung autorisiert und so aufgezeichnet, dass die Bank nachweisen kann, wer sie freigegeben hat; 10.000 Credential-Stuffing-Versuche pro Minute werden gedrosselt, ohne echte Nutzer auszusperren.
Wartbarkeit: mittel	Änderbarkeit und Testbarkeit. Szenario: Ein neues Sparprodukt lässt sich in weniger als 6 Wochen einführen, ohne die Zahlungskomponente zu ändern.
Flexibilität: mittel	Skalierbarkeit. Szenario: Das Backend kann durch zusätzliche Instanzen auf 3 Millionen Nutzer wachsen, ohne Neuentwurf.

Betriebssicherheit: nicht zutreffend	Die App kann keinen physischen Schaden verursachen; Risiken für Geld sind unter Sicherheit und Zuverlässigkeit abgedeckt. Ausdrücklich festgehalten, damit die Entscheidung sichtbar ist.
Ergebnis	Fünf Merkmale mit hoch bewertet, drei mit mittel, eines als nicht zutreffend. Insgesamt elf Szenarien, jedes mit einer Zahl. Die drei wichtigsten (Sicherheit, Zuverlässigkeit, Performance in der Zahltagsspitze) werden zu den Qualitätszielen auf der ersten Seite des Architekturdokuments.

Die Rechnung verwandelt „viele Nutzer am Zahltag“ in eine testbare Auslegungsgröße (etwa 125 Anmeldungen pro Sekunde in Lastspitzen).

Die Teilmerkmale haben ihre Aufgabe erfüllt: Aus „Sicherheit“ wurden Authentizität, Nichtabstreitbarkeit und Widerstandsfähigkeit, die jeweils andere Mechanismen nahelegen.

Betriebssicherheit als nicht zutreffend zu markieren ist eine Entscheidung, keine Auslassung. Fügt die Bank später ein Feature hinzu, das Schaden verursachen könnte, zeigt der Eintrag, dass er neu bewertet werden muss.

Häufige Fragen

Muss ich ISO/IEC 25010 zitieren, um Architektur richtig zu betreiben?
Nein. Die Norm ist wertvoll als gemeinsames Vokabular und als Checkliste und unverzichtbar, wenn Verträge oder Aufsichtsbehörden auf sie verweisen. Viele hervorragende Teams nutzen eine schlankere Liste. Entscheidend ist, dass die wichtigen Qualitäten identifiziert, messbar gemacht und priorisiert werden.

Wohin gehört Skalierbarkeit: zur Performance oder zur Flexibilität?
In ISO/IEC 25010:2023 ist sie ein Teilmerkmal der Flexibilität, während Kapazität unter Leistungseffizienz steht. In der Praxis hängen beide eng zusammen, und Abschnitt 2.4 behandelt sie gemeinsam. Lassen Sie nicht zu, dass Streit über die Einordnung das Schreiben des Szenarios verzögert.

Kommt „Observability“ im Modell vor?
Nicht unter diesem Namen. Teile davon entsprechen der Analysierbarkeit (Wartbarkeit) und der Verantwortlichkeit (Sicherheit). Viele Fachleute behandeln sie heute als eigenständige Qualität; Abschnitt 2.7 geht darauf ein.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Wie wurden Benutzbarkeit und Übertragbarkeit in ISO/IEC 25010:2023 umbenannt?**

Aus Benutzbarkeit wurde Interaktionsfähigkeit, aus Übertragbarkeit Flexibilität.

▼ **Unter welches Merkmal ordnet ISO/IEC 25010:2023 die Verfügbarkeit ein?**

Unter Zuverlässigkeit, neben Fehlerfreiheit, Fehlertoleranz und Wiederherstellbarkeit.

▼ **Warum ist es nützlich, bei einem Merkmal „nicht zutreffend“ einzutragen, statt das Feld leer zu lassen?**

Es hält eine bewusste Entscheidung mit ihrer Begründung fest, sodass Prüfende sehen, dass sie bedacht wurde, und die Entscheidung neu bewertet werden kann, wenn sich das System ändert.

▼ **Was ist die größte Gefahr, wenn man ein Qualitätsmodell in eine Anforderungsvorlage verwandelt?**

Man füllt jedes Feld mit vagen Aussagen, der Aufwand verteilt sich gleichmäßig auf Qualitäten sehr unterschiedlicher Bedeutung, und die wenigen Qualitäten, die die Architektur wirklich entscheiden, gehen unter.

▼ **Probieren Sie es aus: Beim Durchgehen von ISO/IEC 25010 für eine Kurier-App markiert ein Kollege alle neun Merkmale „sicherheitshalber“ als hoch. Was ist daran falsch? Nennen Sie zwei, die Sie als hoch markieren würden, und eines, das Sie als niedrig markieren würden, und begründen Sie das.**

Wenn alles hoch ist, leitet nichts das Design, und das Modell ist zur Bürokratie geworden. Zuverlässigkeit ist hoch: Kuriere mit schlechtem Empfang dürfen keine Aufträge verlieren, schauen Sie also auf die Wiederherstellbarkeit. Interaktionsfähigkeit ist hoch: Fahrer brauchen kurze, einfache Schritte, und aus demselben Grund verdient die Betriebssicherheit einen Blick. Kompatibilität könnte niedriger sein, wenn die App nur mit den eigenen Services des Unternehmens spricht. Halten Sie für jedes Merkmal einen Satz Begründung fest, und schreiben Sie dann nur für die hohen Merkmale Szenarien.

QUELLEN

- ISO/IEC 25010:2023 Product quality model (ISO catalogue page): https://www.iso.org/standard/78176.html
- ISO/IEC 25019:2023 Quality-in-use model (ISO catalogue page): https://www.iso.org/standard/78177.html

- Q42: ISO 25010 update 2023 (summary of changes): https://quality.arc42.org/articles/iso-25010-update-2023
- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/

Testbare Qualitätsszenarien schreiben

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.1 Funktionale und Qualitätsanforderungen
- 2.2 Qualitätsmodelle

Begriffe von früher, die er verwendet: Qualitätsmerkmal, Stakeholder, Produktion

DIE IDEE IN KÜRZE

Wer einem Bauunternehmen sagt: „Machen Sie das Dach stabil“, gibt ihm nichts, was sich prüfen ließe. Die Aussage „Das Dach muss bei einem Wintersturm 50 Zentimeter nassen Schnee tragen, ohne undicht zu werden oder sich um mehr als 2 Zentimeter durchzubiegen“ dagegen schon. Sie benennt, was geschieht (Schnee), die Bedingungen (ein Sturm), das betroffene Bauteil (das Dach), was geschehen soll (keine Undichtigkeit) und eine messbare Zahl (2 Zentimeter). Ein Qualitätsszenario leistet dasselbe für Software. Es verwandelt einen Wunsch wie „Die Website muss schnell sein“ in eine Aussage, die präzise genug ist, um dafür zu entwerfen und sie zu testen.

WAS SIE WISSEN MÜSSEN

- Ein Qualitätsszenario hat sechs Teile: Quelle des Stimulus, Stimulus, Artefakt, Umgebung, Antwort und Antwortmaß.
- Allgemeine Szenarien sind systemunabhängige Vorlagen für eine Qualität; konkrete Szenarien setzen spezifische Werte für ein System ein und dienen als Anforderungen.
- Ein Szenario ohne Antwortmaß oder mit einem Mittelwert statt eines Perzentils für Zeitangaben lässt sich nicht zuverlässig testen.
- Typische Fehler sind, mehrere Stimuli in einem Szenario zu vermischen, die Umgebung wegzulassen und statt des Bedarfs eine Lösung wie eine Technologiewahl hinzuschreiben.
- Szenarien lassen sich direkt in Akzeptanztests übertragen und später in Fitness Functions, die die Qualität fortlaufend prüfen.

Warum Szenarien

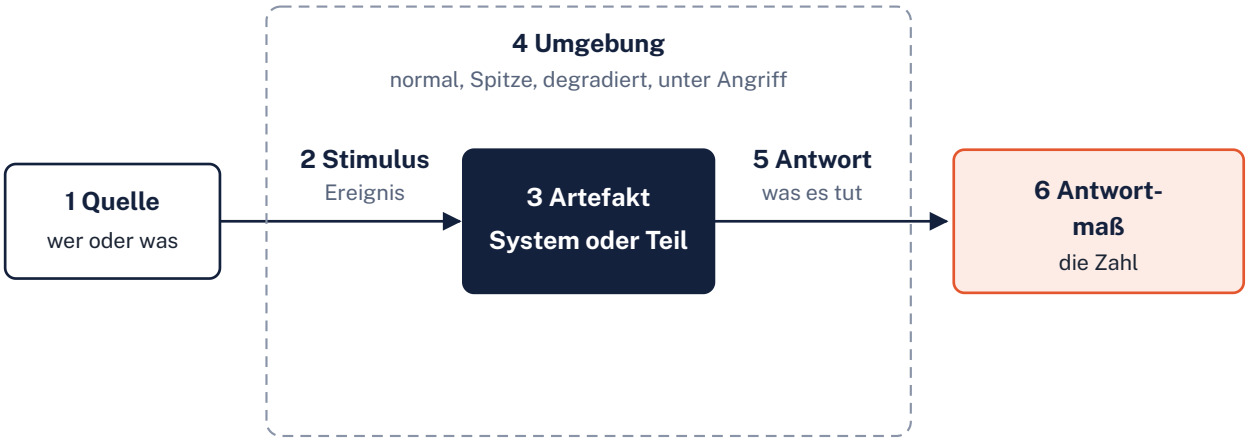
Abschnitt 2.2 hat Ihnen das Vokabular der Qualitäten gegeben; dieser Abschnitt gibt Ihnen die Grammatik. Ein Qualitätsname allein, etwa Performance oder Sicherheit, lässt sich weder entwerfen noch testen. Selbst eine Zahl allein genügt nicht: „2 Sekunden“ sagt nichts, solange Sie nicht wissen, für welche Operation, unter welcher Last, wo gemessen und für welchen Anteil der Anfragen. Das Qualitätsszenario, entwickelt am Software Engineering Institute (SEI) und von Len Bass, Paul Clements und Rick Kazman in *Software Architecture in Practice* beschrieben, löst das, indem es jede Qualitätsanforderung in dieselbe sechsteilige Form zwingt.

Die Idee ist einfach. Jede Qualität zeigt sich, wenn einem System etwas widerfährt: ein Ansturm von Benutzern, ein Absturz, ein Änderungswunsch, ein Angriff. Ein Szenario benennt dieses Ereignis, die Umstände, den betroffenen Teil des Systems, was geschehen soll und woran Sie erkennen, dass es gut genug geschehen ist. Es ist kurz (meist ein bis drei Sätze), konkret und testbar. Und weil jede Qualität dieselbe Form verwendet, können Szenarien für Performance, Sicherheit und Änderbarkeit nebeneinander in einer priorisierten Liste stehen.

Die sechs Teile

Die sechs Teile sind die Quelle des Stimulus, der Stimulus, das Artefakt, die Umgebung, die Antwort und das Antwortmaß. Lernen Sie sie in dieser Reihenfolge, denn zusammen erzählen sie eine kurze Geschichte: *Jemand oder etwas* tut *etwas* mit *einem Teil des Systems* unter *bestimmten Bedingungen*; das System *tut etwas*, und wir *messen es* so. Der Stimulus und das Antwortmaß sind die beiden Teile, die am häufigsten weggelassen werden, und sie sind die beiden wichtigsten.

Aufbau eines Szenarios



Beispiel: 500 Kunden pro Sekunde (Quelle, Stimulus) rufen am Black Friday (Umgebung) den Checkout (Artefakt) auf; Bestellungen werden bestätigt (Antwort), p99 unter 1,5 s, keine verlorenen Bestellungen (Antwortmaß).

Eine Quelle erzeugt einen Stimulus, der ein Artefakt in einer Umgebung erreicht; die Antwort des Artefakts wird an einem Antwortmaß gemessen.

Die sechs Teile im Detail

Teil	Bedeutung	Beispiele
Quelle des Stimulus	Die Person, das System oder das Ereignis, das den Stimulus erzeugt.	Kunden auf Smartphones; ein interner Batch-Job; ein Entwickler; ein Angreifer; eine ausfallende Festplatte.
Stimulus	Das Ereignis, auf das das System reagieren muss.	Eine Anfragespitze; ein Serverabsturz; ein Änderungswunsch; ein Anmeldeversuch mit gestohlenen Zugangsdaten.
Artefakt	Der Teil des Systems, der den Stimulus erfährt: das ganze System oder eine bestimmte Komponente.	Der Checkout-Service; die Bestelldatenbank; das Preismodul; die Deployment-Pipeline.
Umgebung	Die Bedingungen, unter denen der Stimulus eintrifft.	Normalbetrieb; Spitzenlast am Black Friday; ein Rechenzentrum ausgefallen; während eines Deployments; in der Entwicklung.
Antwort	Was das System (oder die Menschen um es herum) daraufhin tut.	Verarbeitet die Bestellung; schaltet per Failover auf das Replikat um; die Änderung wird umgesetzt, getestet und deployt; der Versuch wird blockiert und protokolliert.
Antwortmaß (Response Measure)	Wie die Antwort gemessen wird, damit sich das Szenario testen lässt.	p99 unter 800 ms; Service innerhalb von 60 s wiederhergestellt, ohne verlorene Bestellungen; unter 2 Personentagen; 100 % der Versuche blockiert.

Die Umgebung verdient besondere Sorgfalt, weil derselbe Stimulus unter verschiedenen Bedingungen ganz unterschiedliche Antworten verlangen kann. Ein abgestürzter Server im Normalbetrieb erfordert vielleicht ein automatisches Failover innerhalb einer Minute; derselbe Absturz während eines geplanten Wartungsfensters kann mit manueller Wiederherstellung akzeptabel sein. Ein Änderungswunsch zur Entwurfszeit (vor dem Release) kostet weit weniger als dieselbe Änderung, wenn sich bereits Daten in der Produktion angesammelt haben. Wer die Umgebung ausdrücklich angibt, verhindert, dass Szenarien verglichen werden, die in Wahrheit unterschiedliche Situationen betreffen.

Das Artefakt zählt aus ähnlichem Grund. „Das System antwortet innerhalb von 200 ms“ verbirgt, welcher Teil verantwortlich ist. Wer das Artefakt benennt (den Suchservice, das API-Gateway, den mobilen Client), sagt dem Team, wo die Entwurfsarbeit und die Messung stattfinden müssen. Bei manchen Qualitäten, etwa Änderbarkeit und Deploybarkeit, kann das Artefakt Code, Konfiguration oder die Delivery-Pipeline sein statt einer laufenden Komponente.

Allgemeine und konkrete Szenarien

Bass, Clements und Kazman unterscheiden zwei Arten. Ein allgemeines Szenario ist systemunabhängig: Für ein Qualitätsmerkmal listet es die typischen Werte auf, die jeder Teil annehmen kann. Für Verfügbarkeit etwa können die Quellen intern oder extern sein, Menschen, Hardware, Software oder Infrastruktur; Stimuli können ein Absturz, eine Auslassung, ein falsches Timing oder eine falsche Antwort sein; Antworten können Erkennen, Protokollieren, Wiederherstellen, Maskieren oder eingeschränkter Betrieb sein; und Maße können die Zeit bis zur Erkennung, die Zeit bis zur Reparatur, der Verfügbarkeitsprozentsatz oder der Anteil der bearbeiteten Anfragen sein.

Ein konkretes Szenario wählt spezifische Werte für ein bestimmtes System: „Die primäre Bestelldatenbank stürzt im Normalbetrieb ab; das Replikat übernimmt, und Schreibvorgänge laufen innerhalb von 60 Sekunden wieder, ohne dass eine bestätigte Bestellung verloren geht.“ Allgemeine Szenarien sind Vorlagen und Checklisten; konkrete Szenarien sind Anforderungen. Mit dem allgemeinen Szenario stellen Sie sicher, dass Sie die richtigen Quellen, Stimuli und Maße bedacht haben, und schreiben dann die Handvoll konkreter Szenarien, auf die es für Ihr System ankommt.

Von vagen Aussagen zu testbaren Szenarien

Die meisten Qualitätsanforderungen beginnen ihr Leben als Adjektive. Die Kunst besteht darin, die Fragen zu stellen, die jedes Adjektiv in ein Szenario verwandeln: Welche Operation? Von wem ausgelöst? Wie oft oder wie viel? Unter welchen Bedingungen? Was soll geschehen? Welche Zahl zeigt uns, dass es funktioniert hat? Die Tabelle zeigt typische Vorher-nachher-Paare. Beachten Sie, dass die „Nachher“-Fassungen manchmal offenlegen, dass der Stakeholder mehrere verschiedene Dinge gemeint hat, die dann zu mehreren Szenarien werden.

Vorher und nachher

Vage Aussage	Testbares Szenario
Die Website muss schnell sein.	Während der abendlichen Spitzenlast an Werktagen von 300 Suchanfragen pro Sekunde liefert die Produktsuche Ergebnisse innerhalb von 400 ms für 95 % der Anfragen und innerhalb von 1 s für 99 %, gemessen am Load Balancer.
Das System muss hochverfügbar sein.	Fällt einer der drei Anwendungsserver im Normalbetrieb aus, nimmt der Buchungsservice weiterhin Buchungen an; keine bestätigte Buchung geht verloren, und Benutzer sehen höchstens 30 Sekunden lang Fehler.
Das System muss sicher sein.	Sendet ein Angreifer mit geleakten Zugangsdaten 10.000 Anmeldeversuche pro Minute, sperrt das System die Quelle nach 20 Fehlversuchen, alarmiert das Sicherheitsteam innerhalb von 5 Minuten, und echte Benutzer an anderen Adressen können sich weiterhin anmelden.
Der Code muss leicht änderbar sein.	Wünscht sich der Fachbereich eine neue Lieferoption, kann ein mit dem Code vertrauter Entwickler sie in höchstens 3 Tagen umsetzen, testen und deployen und ändert dabei nur das Liefermodul und dessen Konfiguration.
Die App muss benutzerfreundlich sein.	Ein Erstbenutzer ohne Schulung kann sich in weniger als 4 Minuten registrieren und eine Bestellung aufgeben; 90 % der Testpersonen gelingt das beim ersten Versuch.
Deployments müssen einfach sein.	Ein Entwickler kann eine Änderung am Bestellservice weniger als 30 Minuten nach dem Merge ohne Ausfallzeit in Produktion ausliefern und sie innerhalb von 5 Minuten zurückrollen, wenn die Fehlerraten steigen.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist: Woher kommen die Zahlen?

Viele zögern, eine Zahl hinzuschreiben, weil sie die „richtige“ nicht kennen. Beginnen Sie mit Belegen: aktuellen Traffic-Logs, dem Monitoring des alten Systems, dem Verhalten der Wettbewerber, Geschäftszielen („Wir wollen die Bestellungen bis nächstes Jahr verdoppeln“) und Forschung zur Geduld von Benutzern. Schlagen Sie dann einen Wert vor und lassen Sie die Stakeholder reagieren. „Sind 400 Millisekunden für 95 % der Suchanfragen akzeptabel?“ führt zu einem weit besseren Gespräch als „Wie schnell soll es sein?“. Eine heute vereinbarte Zahl lässt sich morgen ändern; über eine Anforderung ohne Zahl kann man nicht einmal diskutieren.

Szenariobeispiele für sechs Qualitäten

Die folgenden Beispiele verwenden einen einzigen Onlineshop, damit Sie sehen, wie verschiedene Qualitäten im selben Format aussehen. Jedes ist kompakt geschrieben, mit allen sechs Teilen beschriftet; in der Praxis schreiben Teams sie oft als ein oder zwei Sätze.

Konkrete Szenarien für einen Onlineshop

Qualität	Szenario (Quelle, Stimulus, Artefakt, Umgebung, Antwort, Maß)
Performance	Quelle: angemeldete Kunden. Stimulus: 2000 Bestellungen pro Minute. Artefakt: Checkout-Service. Umgebung: Spitzenlast am Freitagabend. Antwort: Bestellungen validiert, gespeichert und bestätigt. Maß: p95 ≤ 1,5 s, p99 ≤ 3 s, keine verlorenen Bestellungen.
Verfügbarkeit	Quelle: interne Hardware. Stimulus: Der primäre Datenbankknoten stürzt ab. Artefakt: Bestelldatenbank. Umgebung: Normalbetrieb. Antwort: Replikat automatisch hochgestuft, laufende Anfragen wiederholt. Maß: Schreibvorgänge laufen innerhalb von 60 s wieder; keine bestätigte Bestellung verloren (RPO = 0).
Änderbarkeit	Quelle: Product Owner. Stimulus: Wunsch, eine neue Zahlungsart hinzuzufügen. Artefakt: Zahlungskomponente. Umgebung: Entwurfszeit, nach dem ersten Release. Antwort: Änderung umgesetzt, getestet und deployt. Maß: ≤ 5 Personentage; keine Änderung außerhalb der Zahlungskomponente; keine Regression bei bestehenden Zahlungsarten.
Sicherheit	Quelle: ein angemeldeter Kunde mit böswilliger Absicht. Stimulus: versucht, die Bestellungen eines anderen Kunden zu lesen, indem er eine ID in der URL ändert. Artefakt: Bestell-API. Umgebung: Normalbetrieb. Antwort: Anfrage abgelehnt und mit der Identität des Aufrufers protokolliert. Maß: 100 % solcher Anfragen abgewiesen; Alarm innerhalb von 1 Minute nach 5 Versuchen.
Benutzbarkeit	Quelle: Erstkunde. Stimulus: möchte nach der Bestellung eine Lieferadresse ändern. Artefakt: Web- und mobile Frontends. Umgebung: Laufzeit, keine Hilfe verfügbar. Antwort: schließt die Änderung ohne Hilfe ab. Maß: 90 % der Teilnehmer am Usability-Test schaffen es innerhalb von 2 Minuten.
Deploybarkeit	Quelle: Entwickler. Stimulus: eine gemergte Änderung am Katalogservice. Artefakt: Deployment-Pipeline und Katalogservice. Umgebung: Geschäftszeiten, normale Last. Antwort: gebaut, getestet und schrittweise ausgerollt. Maß: innerhalb von 45 Minuten in Produktion; keine Ausfallzeit; automatischer Rollback innerhalb von 5 Minuten, wenn die Fehlerrate 1 % übersteigt.

Zwei Merkmale dieser Beispiele verdienen Beachtung. Erstens geht es in Szenarien nicht nur um laufende Software. Szenarien zu Änderbarkeit und Deploybarkeit haben Entwickler und Product Owner als Quellen, und ihre Maße sind Aufwand, verstrichene Zeit und Umfang der

Änderung. Zweitens verbinden gute Maße oft eine Geschwindigkeitszahl mit einer Korrektheitszahl: „innerhalb von 60 Sekunden“ und „keine bestätigte Bestellung verloren“. Ein schnelles Failover, das Daten verliert, ist nicht das, was das Unternehmen wollte.

Häufige Fehler

Wer Hunderte von Teams geschriebene Szenarien prüft, findet immer wieder dieselben Fehler. Jeder macht ein Szenario untestbar oder irreführend.

Fehler und wie man sie behebt

Fehler	Beispiel	Behebung
Kein Antwortmaß	„Unter Spitzenlast antwortet der Checkout schnell.“	Ergänzen Sie eine Zahl und eine Statistik: „p95 ≤ 1,5 s, p99 ≤ 3 s“.
Mittelwert statt Perzentil	„Durchschnittliche Antwortzeit unter 300 ms.“	Mittelwerte verbergen langsame Ausreißer; geben Sie Perzentile an (Abschnitt 2.4).
Mehrere Stimuli in einem Szenario	„Wenn sich die Last verdoppelt oder ein Server ausfällt oder ein Deployment läuft, bleibt das System schnell.“	In drei Szenarien aufteilen, jedes mit eigener Umgebung und eigenem Maß.
Als Anforderung getarnte Lösung	„Das System verwendet Redis, um Produktseiten zu cachen.“	Nennen Sie den Bedarf („Produktseiten laden bei 1000 Aufrufen pro Sekunde für 95 % der Aufrufe in unter 300 ms“) und lassen Sie den Cache als Entwurfsoption offen.
Fehlende Umgebung	„Der Service erholt sich innerhalb von 5 Minuten.“	Sagen Sie, wovon und wann: „nach dem Ausfall einer einzelnen Zone im Normalbetrieb“.
Nicht messbare Wörter	„Elegant“, „nahtlos“, „robust“, „benutzerfreundlich“.	Ersetzen Sie jedes durch ein beobachtbares Verhalten und eine Zahl.
Unmögliche Absolutheiten	„Das System fällt nie aus.“ „100 % der Angriffe werden blockiert.“	Verwenden Sie realistische Ziele und sagen Sie, was geschieht, wenn sie verfehlt werden.

ACHTUNG

Als Anforderungen getarnte Lösungen

Das ist der schädlichste Fehler, weil er wie Fortschritt aussieht. „Kubernetes verwenden“, „Microservices verwenden“, „eine Message Queue verwenden“ sind Entwurfsentscheidungen. Wer sie als Anforderungen formuliert, überspringt den Schritt, in dem das Team fragt, ob sie der beste Weg zur gewünschten Qualität sind, und beseitigt die Belege, die man braucht, um die Wahl später zu überprüfen. Besteht ein Stakeholder auf einer Technologie, fragen Sie, welches Problem sie löst; die Antwort ist das eigentliche Szenario. Ist die Technologie tatsächlich durch eine Richtlinie vorgegeben, halten Sie sie als Randbedingung mit ihrer Quelle fest.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Ist dieses Szenario fertig?

Prüfen Sie jeden Szenarioentwurf mit diesen Fragen, bevor er ins Architekturdokument oder ins Backlog wandert.

Q1 **Beschreibt es einen einzigen Stimulus mit einer benannten Quelle?**

- **Ja** Weiter.
- **Nein** Teilen Sie es auf oder benennen Sie, wer oder was das Ereignis auslöst.

Q2 **Nennt es die Umgebung (normal, Spitzenlast, degradiert, während eines Deployments, zur Entwurfszeit)?**

- **Ja** Weiter.
- **Nein** Ergänzen Sie die Bedingungen; sonst lässt sich das Maß nicht deuten.

Q3 **Benennt es das Artefakt, also den verantwortlichen Teil des Systems?**

- **Ja** Weiter.
- **Nein** Benennen Sie die Komponente, die Schnittstelle oder die Pipeline, damit das Team weiß, wo entworfen und gemessen werden muss.

Q4 **Ist die Antwort ein beobachtbares Verhalten statt einer Technologiewahl?**

- **Ja** Weiter.
- **Nein** Formulieren Sie den Bedarf neu und verschieben Sie die Technologie in eine Entwurfsentscheidung oder eine Randbedingung.

Q5 **Enthält das Antwortmaß eine Zahl, eine Einheit und, bei Zeitangaben, ein Perzentil?**

- **Ja** Das Szenario ist testbar: Ergänzen Sie eine Priorität und einen Verantwortlichen.
- **Nein** Ergänzen Sie das Maß; ohne es ist das Szenario nur ein Wunsch.

Von Szenarien zu Akzeptanztests und Fitness Functions

Ein gut geschriebenes Szenario ist schon der größte Teil eines Tests. Quelle und Stimulus sagen Ihnen, welche Last oder welches Ereignis Sie erzeugen müssen, die Umgebung, wie Sie den Test aufsetzen, das Artefakt, was Sie beobachten müssen, und das Antwortmaß liefert die Bestehensbedingung. Ein Performance-Szenario wird zum Lasttest; ein Verfügbarkeitsszenario wird zur Failover-Übung oder zum Chaos-Experiment; ein Änderbarkeitsszenario wird zur Überprüfung, wie viel Code eine Beispieländerung berührt hat und wie lange sie dauerte; ein Sicherheitsszenario wird zum Testfall eines Penetrationstests.

Der nächste Schritt besteht darin, weiter zu prüfen. Eine Fitness Function, ein Begriff, den Neal Ford, Rebecca Parsons und Patrick Kua in *Building Evolutionary Architectures* bekannt gemacht haben, ist jede automatisierte oder wiederholbare Prüfung, wie gut das System ein Architekturziel erfüllt. In der Delivery-Pipeline oder gegen Telemetriedaten aus der Produktion ausgeführt, macht sie aus einem Szenario statt eines einmaligen Akzeptanztests einen dauerhaften Wächter: Der Build schlägt fehl oder ein Alarm wird ausgelöst, wenn die Qualität abdriftet. Kapitel 11 behandelt Fitness Functions ausführlich in Fitness Functions und evolutionäre Architektur.

Der Lebenslauf eines Szenarios



Ein Szenario steuert einen Akzeptanztest, dann eine Fitness Function, die bei jeder Änderung läuft, und Belege aus der Produktion fließen zurück, um das Szenario zu verfeinern.

```
# A performance scenario expressed as an automated check (illustrative)
# Scenario: at 300 searches/s, p95 ⩽ 400 ms and p99 ⩽ 1000 ms.
import statistics

def check_search_latency(latencies_ms, p95_target=400, p99_target=1000):
    cuts = statistics.quantiles(latencies_ms, n=100)  # 99 cut points
    p95, p99 = cuts[94], cuts[98]
    assert p95 ⩽ p95_target, f"p95 {p95:.0f} ms exceeds {p95_target} ms"
    assert p99 ⩽ p99_target, f"p99 {p99:.0f} ms exceeds {p99_target} ms"
    return p95, p99

# latencies_ms comes from a load test that replays 300 searches per second
# for ten minutes against a production-like environment.
```

Ein Performance-Szenario als einfache automatisierte Prüfung der p95- und p99-Latenz.

Wie Fachleute Szenarien einsetzen

In der Praxis schreiben Teams selten lange Tabellen sechsteiliger Szenarien für alles. Sie schreiben eine kleine Zahl, bei einem größeren System oft zehn bis dreißig, für die architekturrelevanten Qualitäten, und bewahren sie dort auf, wo alle sie sehen: im Architekturdokument (arc42 Abschnitt 10 verlangt genau das), im Backlog als

Akzeptanzkriterien oder auf einem gemeinsamen Qualitätsboard. Manche Teams verwenden eine kürzere Form; das Projekt Q42 von Gernot Starke etwa empfiehlt eine kompakte Struktur aus Kontext, Auslöser und Akzeptanzkriterien. Die Form ist weniger wichtig als die Disziplin: Jede wichtige Qualität hat einen Stimulus, einen Kontext und eine Zahl.

Szenarien treiben auch Bewertungsmethoden an. Der Quality Attribute Workshop und die Architecture Tradeoff Analysis Method (ATAM), beide vom SEI, sammeln und priorisieren Szenarien mit den Stakeholdern und prüfen die Architektur dann gegen die wichtigsten. Abschnitt 2.8 zeigt, wie Szenarien in einem Utility Tree gesammelt und gewichtet werden.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Nennen Sie die sechs Teile eines Qualitätsszenarios in der richtigen Reihenfolge.**

Quelle des Stimulus, Stimulus, Artefakt, Umgebung, Antwort und Antwortmaß.

▼ **Was ist falsch an „Das System verwendet ein CDN, damit Seiten schnell laden“?**

Es nennt eine Lösung, keinen Bedarf, und hat kein Maß. Die Anforderung sollte Last, Bedingungen und Ziel-Antwortzeit beschreiben; das CDN ist ein möglicher Entwurf, um sie zu erfüllen.

▼ **Wer ist in einem Änderbarkeitsszenario typischerweise die Quelle des Stimulus?**

Eine Person, die eine Änderung wünscht, etwa ein Product Owner, ein Entwickler oder eine Aufsichtsbehörde; der Stimulus ist der Änderungswunsch.

▼ **Worin unterscheidet sich ein Szenario von einer Fitness Function?**

Ein Szenario ist die Formulierung einer Anforderung; eine Fitness Function ist eine automatisierte oder wiederholbare Prüfung, die fortlaufend misst, ob das System sie erfüllt.

▼ **Probieren Sie es aus: Machen Sie aus „Die Ticket-Website muss am Vorverkaufstag schnell bleiben“ ein sechsteiliges Szenario für ein Qualitätsmerkmal, und erfinden Sie dafür sinnvolle Werte. Benennen Sie jeden Teil. Nennen Sie dann einen Grund, warum sich der ursprüngliche Satz nie hätte testen lassen.**

Quelle: die Browser der Fans. Stimulus: 2500 Personen kommen pro Sekunde an. Artefakt: der Ablauf für Platzwahl und Kauf. Umgebung: Spitzenlast beim Vorverkaufsstart, alle Zonen fehlerfrei. Antwort: Jede Seitenanfrage wird bedient, oder der Fan kommt in einen Warteraum. Antwortmaß: p99 der Seitenantwort unter 2 Sekunden. Das Original ließ sich nicht testen, weil „schnell“ keine Zahl hat und weder Last noch Bedingungen genannt sind, sodass niemand sagen könnte, ob es bestanden wurde.

QUELLEN

- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- arc42 section 10: Quality requirements and quality scenarios: https://docs.arc42.org/section-10/
- Q42: How to specify quality requirements: https://quality.arc42.org/articles/specify-quality-requirements

Performance und Skalierbarkeit: Latenz, Durchsatz, Perzentile und Last

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.2 Daten und Algorithmen
- 0.4 Wie Computer kommunizieren
- 0.7 Server und die Cloud

Begriffe von früher, die er verwendet: Server, Cache, Datenbank

DIE IDEE IN KÜRZE

Stellen Sie sich einen Supermarkt vor. Wie lange Sie an der Kasse verbringen, ist die Latenz; wie viele Kunden die Kassen in einer Stunde bedienen, ist der Durchsatz. Ihre Zeit an der Kasse besteht teils aus dem Scannen (der eigentlichen Arbeit) und teils aus der Schlange vor Ihnen. Sind die Kassen nur halb ausgelastet, stehen Sie kaum an; sind sie fast immer belegt, werden die Schlangen sehr schnell sehr lang, obwohl jeder Scan gleich lange dauert. Mehr Kassen zu öffnen hilft, bis sich alle einen einzigen Filialleiter für Erstattungen teilen müssen. Bei Performance und Skalierbarkeit geht es in Software um genau diese Effekte.

WAS SIE WISSEN MÜSSEN

- Latenz ist, wie lange eine Arbeitseinheit dauert; Durchsatz ist, wie viel Arbeit pro Zeiteinheit erledigt wird. Eine Performance-Anforderung sollte beides angeben, zusammen mit der Last, bei der es gilt.
- Die Antwortzeit ist Bedienzeit plus Wartezeit (plus Netzwerkzeit); unter Last überwiegt meist die Wartezeit.
- Legen Sie Perzentile wie p95 und p99 fest und überwachen Sie sie, nicht Mittelwerte; in Fan-out-Systemen wird eine seltene langsame Antwort eines Servers für Benutzer zur Regel (Dean und Barroso, 2013).
- Littles Gesetz, $L = \lambda \times W$, verknüpft Anfragerate, Antwortzeit und Nebenläufigkeit und ist die Grundlage für die Dimensionierung von Pools und Limits.
- Im einfachsten Warteschlangenmodell, M/M/1, beträgt die mittlere Antwortzeit $S \div (1 - \rho)$; sie verdoppelt sich also zwischen 80 % und 90 % Auslastung. Andere Warteschlangen unterscheiden sich im Detail, zeigen aber dieselbe Auslastungsklippe; halten Sie kritische Ressourcen daher deutlich unterhalb der Sättigung.

- Amdahls Gesetz begrenzt die Beschleunigung bei einem seriellen Anteil s auf 1 ÷ s; das Universal Scalability Law ergänzt Kohärenzkosten, durch die die Kapazität einen Höchstwert erreicht und dann sinkt.

Zwei verschiedene Fragen: wie schnell und wie viel

Performance ist die Qualität, die zuerst genannt und am schlechtesten spezifiziert wird. Sie umfasst zwei verschiedene Fragen. Latenz fragt, wie lange eine Arbeitseinheit dauert: wie viele Millisekunden zwischen dem Tippen auf „Bezahlen“ und der Meldung „Zahlung akzeptiert“ liegen. Durchsatz fragt, wie viel Arbeit das System pro Zeiteinheit erledigt: 1200 Zahlungen pro Sekunde, 40.000 Bestellungen pro Stunde. Beide hängen zusammen, sind aber nicht austauschbar. Ein Reisebus, der 50 Menschen in 100 Minuten von London nach Oxford bringt, hat eine schlechtere Latenz und einen viel besseren Durchsatz als ein Sportwagen, der zwei Menschen in 60 Minuten befördert.

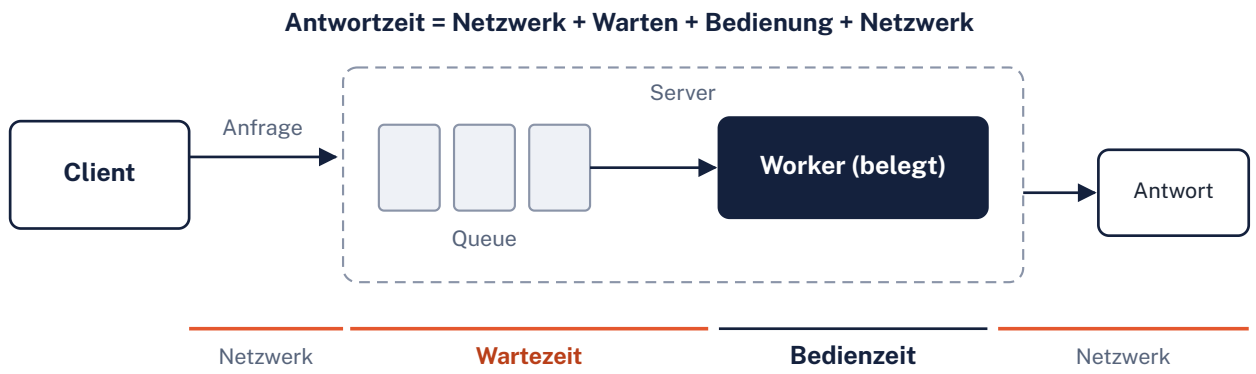
In Software zeigt sich dieselbe Spannung überall. Datenbank-Schreibvorgänge zu bündeln verbessert den Durchsatz, lässt aber jeden einzelnen Schreibvorgang auf seinen Stapel warten. Ein Cache verbessert beides für wiederholte Lesezugriffe, hilft aber nicht beim ersten. Eine gute Performance-Anforderung sagt deshalb, um welche Frage es geht, und nennt meist beides: „bei 1200 Zahlungen pro Sekunde (Durchsatz) werden 99 % innerhalb von 800 ms abgeschlossen (Latenz)“. ISO/IEC 25010 ordnet dies unter Leistungseffizienz als Zeitverhalten, Ressourcennutzung und Kapazität ein (Abschnitt 2.2).

Antwortzeit = Bedienzeit + Wartezeit

Wenn Sie die Latenz auf der Seite des Clients messen, erhalten Sie die Antwortzeit: alles, was zwischen dem Senden der Anfrage und dem Empfang der Antwort geschah. Es lohnt sich, sie in Teile zu zerlegen. Die Bedienzeit ist die Zeit, die eine Ressource tatsächlich an Ihrer Anfrage arbeitet: sie parst, die Abfrage ausführt, das Ergebnis aufbereitet. Die Wartezeit ist die Zeit, die Ihre Anfrage in einer Warteschlange verbringt, weil die Ressource mit der Anfrage eines anderen beschäftigt ist. Hinzu kommt die Netzwerk-Übertragungszeit, in beiden Richtungen.

Die Unterscheidung zählt, weil die beiden Teile auf unterschiedliche Mittel ansprechen. Die Bedienzeit sinkt, wenn Sie die Arbeit selbst billiger machen: ein besserer Index, eine kleinere Nutzlast, ein schnellerer Algorithmus. Die Wartezeit sinkt, wenn Sie Kapazität hinzufügen oder die Last senken, und sie ist bei einem ausgelasteten System oft viel größer als die Bedienzeit. Viele Untersuchungen einer „langsamen Datenbank“ enden mit der Entdeckung, dass jede Abfrage 5 ms dauert, aber 300 ms auf eine freie Verbindung im Pool wartet.

Wohin die Zeit geht



Die Antwortzeit, die ein Client sieht, umfasst die Netzwerkübertragung, das Warten in einer Queue auf einen belegten Worker und die Bedienzeit selbst.

Perzentile: Warum Mittelwerte irreführen

Antwortzeiten sind nie eine einzelne Zahl; sie bilden eine Verteilung, und die ist fast immer schief. Die meisten Anfragen sind schnell, und eine Minderheit ist langsam, weil sie auf einen kalten Cache, eine Garbage-Collection-Pause, eine Sperre, eine Wiederholung oder einen überlasteten Nachbarn trifft. Das arithmetische Mittel vermischt beide Gruppen zu einem Wert, der niemanden beschreibt. Fachleute verwenden stattdessen Perzentile. Das 50. Perzentil (p50, der Median) ist der Wert, den die Hälfte der Anfragen unterbietet; p95 ist der Wert, den 95 % der Anfragen unterbieten; p99 und p99,9 beschreiben die langsamsten 1 % und 0,1 %.

Das langsame Ende der Verteilung heißt Tail, und es ist wichtiger, als es scheint. Die Benutzer im Tail sind oft Ihre besten Kunden: die mit den größten Warenkörben, der längsten Historie und den meisten Daten. Ein Benutzer, der 40 Ressourcen lädt, um eine Seite anzusehen, erlebt die langsamste dieser 40, nicht den Durchschnitt. Und die Wahrscheinlichkeit, dass eine Seite mit 40 Ressourcen mindestens eine p99-Anfrage trifft, beträgt $1 - 0{,}99^{40} \approx 33$ %: Ein „seltenes“ 1-%-Ereignis betrifft ein Drittel der Seitenaufrufe.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: der Mittelwert, der das Problem verbirgt

Eine Produktsuch-API protokolliert in einer Minute 1000 Antwortzeiten: 900 mit 80 ms, 80 mit 150 ms, 15 mit 1200 ms (kalter Cache) und 5 mit 4000 ms (Timeout und Wiederholung).

Summe aller Zeiten	900 × 80 + 80 × 150 + 15 × 1200 + 5 × 4000 = 72.000 + 12.000 + 18.000 + 20.000 = 122.000 ms
Mittelwert	122.000 ÷ 1000 = 122 ms. Sieht gegenüber einem Ziel von 200 ms gesund aus.
p50 (Median)	Die 500.-schnellste Anfrage: 80 ms
p95	Die 950.-schnellste Anfrage liegt in der 150-ms-Gruppe (Anfragen 901 bis 980): 150 ms
p99	Die 990. liegt in der 1200-ms-Gruppe (Anfragen 981 bis 995): 1200 ms
p99.9	Die 999. liegt in der 4000-ms-Gruppe (996 bis 1000): 4000 ms
Wer betroffen ist	20 von 1000 Anfragen (2 %) dauerten über eine Sekunde. Bei 300 Suchanfragen pro Sekunde sind das 6 Benutzer pro Sekunde oder etwa 21.600 pro Stunde.

Der Mittelwert von 122 ms besteht ein Ziel von 200 ms, während 2 % der Benutzer länger als eine Sekunde warten. Ein Szenario mit der Formulierung „Durchschnitt unter 200 ms“ würde dieses System durchlassen.

Die Anforderung „p95 ≤ 200 ms und p99 ≤ 500 ms“ lässt es zu Recht durchfallen und lenkt das Team auf die Pfade mit kaltem Cache und Wiederholung.

Perzentile lassen sich nicht mitteln. Das p99 über zehn Server ist nicht der Mittelwert ihrer zehn p99-Werte; es muss aus den zusammengeführten Daten berechnet oder aus zusammengeführten Histogrammen angenähert werden.

```
# Why the mean misleads: 1,000 response times from one minute of traffic
import statistics

times_ms = [80] * 900 + [150] * 80 + [1200] * 15 + [4000] * 5

mean = statistics.mean(times_ms)
cuts = statistics.quantiles(times_ms, n=1000, method="inclusive")
p50, p95, p99, p999 = cuts[499], cuts[949], cuts[989], cuts[998]
print(f"mean {mean:.0f} ms, p50 {p50:.0f}, p95 {p95:.0f}, p99 {p99:.0f}, p99.9
{p999:.0f}")
# mean 122 ms, p50 80, p95 150, p99 1200, p99.9 4000
```

Dieselbe Berechnung in wenigen Zeilen Python; der Mittelwert sieht gut aus, während p99 und p99,9 das Problem aufdecken.

Tail-Latenz im großen Maßstab

Der Tail wird zum Hauptereignis in großen Systemen mit Fan-out, bei denen eine Benutzeranfrage in viele parallele Anfragen an andere Server aufgeteilt wird und die Antwort auf alle warten muss. Eine Suchmaschine, ein Social-Feed oder eine Empfehlungsseite befragt für jede Abfrage womöglich Dutzende oder Hunderte von Backend-Shards. Jeffrey Dean und Luiz André Barroso von Google haben das in *The Tail at Scale* (Communications of the ACM, 2013) analysiert.

Ihre Veranschaulichung ist einfach. Angenommen, jeder Server antwortet typischerweise in 10 ms, braucht aber für eine von 100 Anfragen 1 Sekunde. Eine Benutzeranfrage, die auf einen Server wartet, ist in 1 % der Fälle langsam. Eine Benutzeranfrage, die auf 100 solcher Server wartet, ist immer dann langsam, wenn irgendeiner von ihnen langsam ist: in $1 - 0{,}99^{100} \approx 63$ % der Fälle. Ein seltenes Ereignis an einem Server wird für den Benutzer zum Normalfall. Dean und Barroso berichten Messungen aus einem Google-Service, bei dem die Latenz im 99. Perzentil für eine einzelne Leaf-Anfrage 10 ms betrug, das 99. Perzentil für das Warten auf alle Leaf-Anfragen aber 140 ms.

Weil sich nicht jede Quelle der Schwankung beseitigen lässt (geteilte Hardware, Hintergrundjobs, Garbage Collection, Netzwerk-Queues), plädieren sie für tail-tolerante Entwürfe, so wie fehlertolerante Entwürfe akzeptieren, dass Hardware ausfällt. Zu ihren Techniken gehören Hedged Requests (eine zweite Kopie an ein anderes Replikat senden, wenn die erste nicht innerhalb etwa der Zeit des 95. Perzentils geantwortet hat, und die zuerst eintreffende Antwort verwenden), verknüpfte Anfragen, die sich gegenseitig abbrechen, feiner granulierte Partitionen, die sich umverteilen lassen, das vorübergehende Ausschließen langsamer Maschinen und die Rückgabe eines ausreichend guten Ergebnisses, wenn die

meisten, aber nicht alle Shards geantwortet haben. In einem von ihnen beschriebenen Benchmark senkte Hedging nach 10 ms die Latenz im 99,9. Perzentil von 1800 ms auf 74 ms, bei nur etwa 2 % mehr Anfragen.

Littles Gesetz: wie viele Anfragen gleichzeitig in Bearbeitung sind

Littles Gesetz, 1961 von John Little bewiesen, ist die nützlichste einzelne Formel der Kapazitätsplanung. Für jedes stabile System gilt: Die durchschnittliche Zahl der Elemente darin (L) ist gleich der durchschnittlichen Ankunftsrate (λ, Lambda) multipliziert mit der durchschnittlichen Verweildauer jedes Elements (W): **L = λ × W**. Es gilt unabhängig vom Ankunftsmuster oder der Reihenfolge der Bedienung, sofern das System stabil ist (im Mittel verlassen es so viele Elemente, wie ankommen) und Sie einheitliche Einheiten verwenden.

In Software ist L die Zahl der gleichzeitig laufenden Anfragen (die Nebenläufigkeit), λ die Anfragerate und W die Antwortzeit. Erhält ein Service 200 Anfragen pro Sekunde und verbringt jede 0,25 Sekunden darin, sind im Mittel jederzeit 200 × 0,25 = 50 Anfragen in Bearbeitung. Diese Zahl bestimmt Thread-Pools, Connection-Pools, Arbeitsspeicher und Lizenzanzahlen. Das Gesetz funktioniert auch rückwärts: Können Sie höchstens 50 Anfragen gleichzeitig halten und dauert jede 0,25 s, schaffen Sie dauerhaft nicht mehr als 50 ÷ 0,25 = 200 Anfragen pro Sekunde.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: einen Datenbank-Connection-Pool mit Littles Gesetz dimensionieren

Der Fahrtenservice einer Fahrdienst-App verarbeitet zur abendlichen Spitzenzeit 1800 Anfragen pro Sekunde. Jede Anfrage macht Datenbankaufrufe, die eine Verbindung insgesamt 40 ms lang belegen. Jede der 6 Service-Instanzen hat einen Pool von 10 Verbindungen. Reicht das?

Ankunftsrate λ	1800 Anfragen pro Sekunde
Belegungsdauer einer Verbindung W	40 ms = 0,040 s
Belegte Verbindungen L = λ × W	1800 × 0,040 = im Mittel 72 belegte Verbindungen
Verfügbare Verbindungen	6 Instanzen × 10 = 60
Ergebnis	72 > 60. Anfragen werden auf Verbindungen warten, die Wartezeit wächst zur Spitzenzeit unbegrenzt, und die Antwortzeiten steigen, bis Timeouts einsetzen.
Ziel mit Kapazitätsreserve	Den Pool höchstens zu etwa 70 % auslasten: 72 ÷ 0,7 ≈ 103 Verbindungen, also etwa 18 pro Instanz
Datenbank prüfen	103 Verbindungen müssen auch für den Datenbankserver tragbar sein; falls nicht, W senken (schnellere Abfragen, weniger Roundtrips) oder die Zahl der Datenbankaufrufe pro Anfrage verringern
Was, wenn die Abfragen schneller werden?	Wird W auf 25 ms gesenkt, ergibt sich L = 1800 × 0,025 = 45 Verbindungen; das passt bei 75 % Auslastung in die vorhandenen 60

Littles Gesetz rechnet ein Durchsatzziel und eine Latenz in eine Nebenläufigkeitszahl um, die sich direkt mit einem konfigurierten Limit vergleichen lässt.

Es liefert Mittelwerte. Lastspitzen treiben die momentane Zahl höher; deshalb plant das Beispiel eine Reserve ein, statt genau auf 72 zu dimensionieren.

Auslastung und Warteschlangen: Warum Systeme von der Klippe stürzen

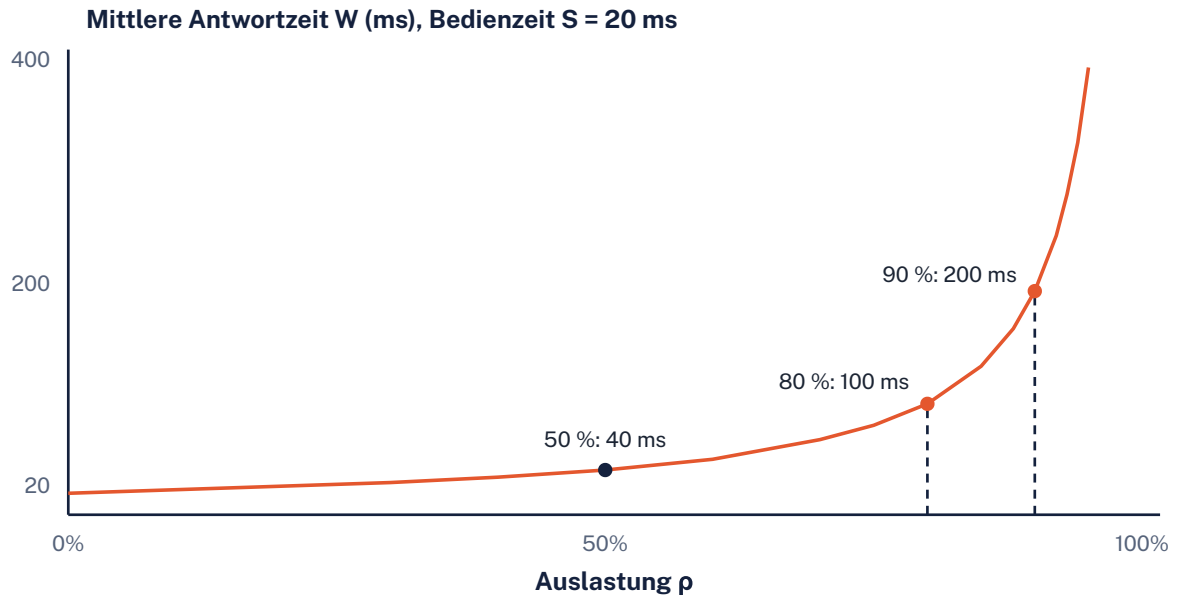
Auslastung (ρ, Rho) ist der Anteil der Zeit, in dem eine Ressource belegt ist. Ein Server, der 40 Anfragen pro Sekunde erhält, die jeweils 20 ms Bedienzeit brauchen, ist 40 × 0,020 = 0,8 der Zeit belegt: zu 80 % ausgelastet. Die Intuition sagt, eine zu 80 % belegte Ressource sollte nur wenig langsamer sein als eine zu 50 % belegte. Die Theorie der Warteschlangenbildung sagt etwas anderes, denn Anfragen treffen nicht gleichmäßig ein; sie ballen sich, und wenn sie sich ballen, warten die späteren.

Das einfachste Modell ist die M/M/1-Warteschlange: zufällige Ankünfte, zufällige Bedienzeiten und ein einzelner Server. Ihre mittlere Antwortzeit ist **W = S ÷ (1 − ρ)**, wobei S die mittlere Bedienzeit ist. Bei 50 % Auslastung ist W = 2S. Bei 80 % ist W = 5S. Bei 90 % ist W = 10S. Bei 99 % ist W = 100S. Der Nenner schrumpft gegen null, wenn sich die Auslastung 100 % nähert, sodass die Antwortzeit unbegrenzt wächst. Reale Systeme sind nicht genau M/M/1, und Pools mit mehreren Servern verhalten sich bei gleicher Auslastung besser, aber die Form ist universell: flach, dann ein Knick, dann eine Klippe.

M/M/1-Warteschlange: mittlere Antwortzeit bei 20 ms Bedienzeit

Auslastung ρ	W = S ÷ (1 − ρ)	Mittlere Antwortzeit	Davon Wartezeit
50%	20 ÷ 0.5	40 ms	20 ms
70%	20 ÷ 0.3	67 ms	47 ms
80%	20 ÷ 0.2	100 ms	80 ms
90%	20 ÷ 0.1	200 ms	180 ms
95%	20 ÷ 0.05	400 ms	380 ms
99%	20 ÷ 0.01	2000 ms	1980 ms

Die Auslastungsklippe



In einer M/M/1-Warteschlange verdoppelt sich die mittlere Antwortzeit zwischen 80 % und 90 % Auslastung und steigt in Richtung 100 % weiter steil an.

Die praktischen Lehren sind direkt. Planen Sie nicht, kritische Ressourcen nahe 100 % Belegung zu betreiben; lassen Sie Reserve, damit Lastspitzen Sie nicht über den Knick schieben. Beobachten Sie Auslastung und Queue-Länge gemeinsam (die USE-Methode von Brendan Gregg prüft für jede Ressource Auslastung, Sättigung und Fehler). Und denken Sie daran, dass die Klippe kleine Änderungen gefährlich macht: Bei 90 % Auslastung bringt Sie ein Anstieg des Traffics um nur etwa 6 % auf 95 % und verdoppelt die mittlere Antwortzeit. Kapitel 9 behandelt Queues, Backpressure und Kapazitätsplanung in Asynchronität, Queues und Backpressure.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: Warum ein ausgelasteter Server Anfragen warten lässt

Der Punkt, an dem hier die meisten stolpern, ist, warum das Warten so steil zunimmt, je stärker eine Ressource belegt ist, obwohl jede Anfrage nicht mehr Arbeit braucht als vorher. Verfolgen Sie einen Server, während sein Traffic steigt.

- Jede Anfrage braucht im Mittel 20 ms Arbeit: ihre Bedienzeit. Bei 40 Anfragen pro Sekunde ist der Server 40 × 0,020 = 0,8 der Zeit belegt; seine Auslastung beträgt also 80 %.
- Kämen die Anfragen gleichmäßig an, eine alle 25 ms, fände jede den Server frei vor und bräuchte genau 20 ms.
- Echte Anfragen kommen zufällig an und ballen sich deshalb. Kommen drei gleichzeitig, wird die erste sofort bedient, die zweite wartet etwa 20 ms und die dritte etwa 40 ms. Danach folgt vielleicht eine Lücke, in der der Server untätig ist, und Leerlaufzeit lässt sich nicht für die nächste Ballung aufsparen.
- Bei 50 % Belegung gibt es reichlich Lücken, sodass sich Schlangen bald auflösen. Bei 80 % gibt es weniger, und die Schlange einer Ballung ist oft noch da, wenn die nächste eintrifft. Nahe 100 % bleibt fast keine freie Zeit, und die Schlangen werden sehr lang.
- Die Tabelle des Abschnitts für einen einzelnen Server mit zufälligen Ankünften (die M/M/1-Warteschlange) zeigt den Effekt: eine mittlere Antwortzeit von 40 ms bei 50 % Belegung, 100 ms bei 80 % und 200 ms bei 90 %. Der Schritt von 40 auf 45 Anfragen pro Sekunde bringt den Server von 80 % auf 90 % Belegung und verdoppelt die mittlere Antwortzeit.
- **Die Lehre:** Die Arbeit selbst ist nicht langsamer geworden; die zusätzliche Zeit ist reine Warteschlangenbildung. Halten Sie kritische Ressourcen deutlich unter Volllast, denn nahe Volllast führt ein kleiner Anstieg des Traffics zu einem großen Anstieg der Antwortzeit.

Skalierbarkeit: Last, Elastizität, Scale-up und Scale-out

Skalierbarkeit ist die Fähigkeit, durch zusätzliche Ressourcen mehr Last zu bewältigen, idealerweise mit proportional wachsender Kapazität und ohne Neuentwurf. Sie bedeutet erst etwas, wenn Sie sagen, was „Last“ ist. Lastparameter unterscheiden sich von System zu System: Anfragen pro Sekunde bei einer API, gleichzeitige Verbindungen bei einem Chatdienst, das Verhältnis von Lese- zu Schreibzugriffen bei einem sozialen Netzwerk, das Datenvolumen bei einer Analyseplattform oder die Zahl der Follower des meistgefolgten Kontos bei einem Feed. Ein Skalierbarkeitsszenario nennt den Lastparameter, das Wachstum und die Antwort,

die erhalten bleiben muss: „Wenn die täglichen Bestellungen von 50.000 auf 500.000 steigen, bleibt die p95-Latenz des Checkouts durch zusätzliche Instanzen unter 1,5 s, bei höchstens 20 % höheren Kosten pro Bestellung.“

Elastizität fügt die Zeit hinzu: die Fähigkeit, Kapazität automatisch hinzuzufügen und zu entfernen, wenn die Last steigt und fällt, sodass eine Ticketplattform für einen Vorverkauf wachsen und danach wieder schrumpfen kann. Sie ist ein großer Teil des wirtschaftlichen Arguments für Cloud-Plattformen. Es gibt zwei grundlegende Wege, Kapazität hinzuzufügen. Vertikale Skalierung (Scale-up) gibt einem Rechner mehr CPU, mehr Arbeitsspeicher oder schnelleren Speicher; sie ist einfach, erfordert keine Änderung an der Software und stößt an harte Grenzen und steigende Preise. Horizontale Skalierung (Scale-out) fügt weitere Rechner hinzu und verteilt die Arbeit; sie hat keine feste Obergrenze, verlangt aber, dass die Software dafür entworfen ist, mit zustandslosen Instanzen, Lastverteilung und partitionierten Daten. Kapitel 9 behandelt die Strategien ausführlich in Skalierungsstrategien: vertikal, horizontal und der Scale Cube.

Amdahls Gesetz: die Grenze der Parallelität

Mehr Prozessoren oder Knoten helfen nur bei Arbeit, die sich parallel erledigen lässt. Das Amdahlsche Gesetz, aus Gene Amdahls Aufsatz von 1967 über die Grenzen des parallelen Rechnens, erfasst das. Ist ein Anteil s der Arbeit seriell (er muss Schritt für Schritt erledigt werden) und lässt sich der Rest, 1 − s, perfekt auf N Prozessoren aufteilen, beträgt die bestmögliche Beschleunigung **speed-up(N) = 1 ÷ (s + (1 − s) ÷ N)**. Wächst N unbegrenzt, nähert sich die Beschleunigung 1 ÷ s. Bei 10 % serieller Arbeit kann keine Zahl von Prozessoren den Job mehr als zehnmal schneller machen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein nächtlicher Abrechnungslauf

Der nächtliche Abrechnungslauf eines Telekommunikationsunternehmens dauert auf einem Server 8 Stunden. Profiling zeigt, dass 10 % der Zeit seriell sind (Tarife laden, dann eine einzige Zusammenfassungsdatei schreiben); die anderen 90 % bewerten Anrufe unabhängig pro Kunde. Der Lauf muss in unter 1,5 Stunden fertig sein. Wie viele Server werden gebraucht?

Serieller Anteil s	0,10; paralleler Anteil 0,90
Benötigte Beschleunigung	8 h ÷ 1.5 h ≈ 5.33
N = 4	1 ÷ (0,10 + 0,90 ÷ 4) = 1 ÷ 0,325 ≈ 3,08; Laufzeit 8 ÷ 3,08 ≈ 2,6 h
N = 8	1 ÷ (0,10 + 0,1125) = 1 ÷ 0,2125 ≈ 4,71; Laufzeit ≈ 1,70 h
N = 16	1 ÷ (0,10 + 0,05625) = 1 ÷ 0,15625 = 6,4; Laufzeit = 1,25 h
Kleinstes N, das das Ziel erreicht	Nötig ist 0,10 + 0,90 ÷ N ≤ 1 ÷ 5,33 = 0,1875, also N ≥ 0,90 ÷ 0,0875 ≈ 10,3: 11 Server ergeben 1 ÷ (0,10 + 0,0818) ≈ 5,50, Laufzeit ≈ 1,45 h
N = 64	1 ÷ (0,10 + 0,0141) ≈ 8,77; Laufzeit ≈ 0,91 h
Grenzwert für wachsendes N	1 ÷ 0,10 = 10; Laufzeit nie unter 0,8 h
Alternative	Den seriellen Teil halbieren (s = 0,05), etwa indem die Tarife parallel geladen werden. Dann ergeben 8 Server 1 ÷ (0,05 + 0,11875) ≈ 5,93, Laufzeit ≈ 1,35 h: Das Ziel wird mit 8 statt 11 Servern erreicht.

Der Schritt von 16 auf 64 Server (die vierfachen Kosten) verkürzt den Lauf nur von 1,25 h auf 0,91 h. Der Ertrag nimmt schnell ab.

Den seriellen Anteil zu verringern ist oft weit billiger, als Hardware hinzuzufügen. Architekten suchen nach den versteckten seriellen Teilen: einer einzelnen Sperre, einer Datenbankzeile, die alle aktualisieren, einem zentralen Koordinator.

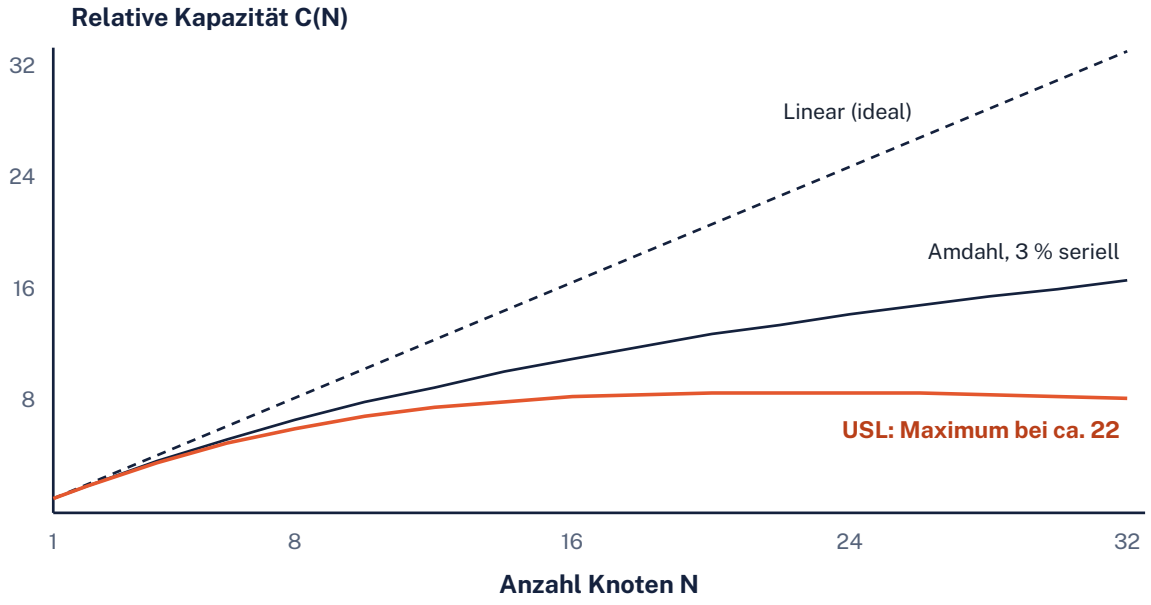
Das Universal Scalability Law

Amdahls Gesetz nimmt an, dass zusätzliche Prozessoren die Lage nie verschlechtern; die Beschleunigung flacht nur ab. Reale verteilte Systeme werden oft tatsächlich schlechter: Ab einem gewissen Punkt senken weitere Knoten den Durchsatz. Neil Gunthers Universal Scalability Law (USL) modelliert das mit zwei Abzügen. Konkurrenz (α, Alpha) sind die Kosten

des Wartens auf geteilte Ressourcen, derselbe Effekt wie Amdahls serieller Anteil. Kohärenz (β, Beta) sind die Kosten, Daten zwischen Knoten konsistent zu halten: Jeder Knoten muss mit jedem anderen Informationen austauschen, sodass diese Kosten ungefähr mit dem Quadrat der Knotenzahl wachsen.

Die relative Kapazität ist **C(N) = N ÷ (1 + α(N − 1) + βN(N − 1))**. Mit β = 0 reduziert sie sich auf Amdahls Gesetz. Mit jedem β über null erreicht die Kapazität bei etwa N = √((1 − α) ÷ β) einen Höchstwert und fällt dann. Für α = 0,03 und β = 0,002 liegt der Höchstwert bei etwa √(0,97 ÷ 0,002) ≈ 22 Knoten, mit einer Kapazität von etwa dem 8,6-Fachen eines Knotens; bei 32 Knoten ist die Kapazität wieder auf etwa 8,2 gesunken. Gunther beschreibt das Modell in *Guerrilla Capacity Planning* (2007). Sein praktischer Wert liegt darin, dass sich α und β an eine Handvoll Lasttest-Messungen anpassen lassen, sodass Sie vorhersagen können, wo sich Skalierung nicht mehr lohnt, bevor Sie die Hardware kaufen.

Lineare, Amdahl- und USL-Skalierung



Ideale Skalierung ist linear; Amdahls Gesetz flacht wegen serieller Arbeit ab; das USL berücksichtigt zusätzlich Kohärenzkosten, sodass die Kapazität einen Höchstwert erreicht und dann sinkt.

Taktiken für Performance: ein Überblick

Bass, Clements und Kazman ordnen Performance-Taktiken in zwei Familien. Die erste ist, den Ressourcenbedarf zu steuern: zu verringern, wie viel Arbeit ankommt oder wie viel jede Einheit kostet. Die zweite ist, Ressourcen zu verwalten: die verfügbaren Ressourcen stärker zu nutzen oder mehr davon hinzuzufügen. Die meisten realen Entwürfe kombinieren mehrere Taktiken; jede hat auch anderswo Kosten, weshalb es sich um Architekturentscheidungen handelt.

Performance-Taktiken und wo der Kurs sie behandelt

Taktik	Was er tut	Typische Kosten oder Risiken	Mehr dazu in
Bedarf verringern: Rate Limiting, Zugangskontrolle, Warteräume	Begrenzt oder glättet die eingehende Last, sodass der Kern unterhalb seines Knicks bleibt	Manche Benutzer werden verzögert oder abgewiesen	Kapitel 9: Queues und Backpressure
Arbeit pro Anfrage verringern: effiziente Algorithmen, kleinere Nutzlasten, weniger Roundtrips	Senkt die Bedienzeit	Entwicklungsaufwand; manchmal weniger flexibler Code	Kapitel 0: Daten und Algorithmen
Priorisieren und einplanen	Bedient wichtige Arbeit zuerst (bezahlte Bestellungen vor Berichten)	Niedrig priorisierte Arbeit kann verhungern	Kapitel 9: Resilienzmuster
Caching	Hält Ergebnisse nahe beim Benutzer, um wiederholte Arbeit zu vermeiden	Veraltete Daten, aufwendige Invalidierung, Speicherkosten	Kapitel 7: Caching
Nebenläufigkeit und Asynchronität	Verarbeitet unabhängige Arbeit parallel; nimmt langsame Arbeit aus dem Anfragepfad	Schwierigeres Nachvollziehen, Probleme mit Reihenfolge und Konsistenz	Kapitel 6: Kommunikation
Replikation	Fügt Kopien von Services oder Daten hinzu, um Lesezugriffe und Last zu verteilen	Konsistenz zwischen den Kopien; Kosten	Kapitel 7: Replikation und Partitionierung
Partitionierung (Sharding)	Teilt Daten und Arbeit auf, sodass jeder Knoten einen Anteil übernimmt	Partitionsübergreifende Abfragen und Umverteilung sind schwierig	Kapitel 7: Replikation und Partitionierung
Ressourcen hinzufügen: Scale-up oder Scale-out	Erhöht die Kapazität	Kosten; begrenzt durch serielle Arbeit und Kohärenzaufwand	Kapitel 9: Skalierungsstrategien

ACHTUNG

Häufige Fehler

- Mittelwerte statt Perzentile festzulegen oder zu überwachen und Perzentile über Server hinweg zu mitteln.
- Mit konstanter Rate Lasttests zu fahren, obwohl der reale Traffic in Schüben ankommt; die Queues entstehen in den Schüben.
- Zu planen, eine kritische Ressource mit 90 % Auslastung oder mehr zu betreiben, ohne Spielraum für Spitzen.
- Anzunehmen, dass mehr Server immer mehr Kapazität bedeuten, ohne die Kosten für serielle Arbeit und Kohärenz zu messen.
- Die Bedienzeit zu optimieren und dabei die Wartezeit zu ignorieren, die unter Last meist der größere Teil ist.
- Nur auf der Serverseite zu messen, obwohl Benutzer auch die Netzwerkzeit und das Rendering im Client erleben.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Ein Service erhält 500 Anfragen pro Sekunde, und jede dauert 80 ms. Wie viele sind im Mittel gleichzeitig in Bearbeitung?

Nach Littles Gesetz ist L = 500 × 0,08 = 40 Anfragen in Bearbeitung.

▼ Wie hoch ist nach der M/M/1-Formel die mittlere Antwortzeit eines Servers mit 10 ms Bedienzeit bei 75 % Auslastung?

W = 10 ÷ (1 − 0,75) = 40 ms, davon 30 ms Wartezeit.

▼ Warum ist p99 als Anforderungsmaß meist besser als der Mittelwert?

Verteilungen von Antwortzeiten sind schief; der Mittelwert kann gesund aussehen, während ein nennenswerter Teil der Benutzer sehr langsame Antworten erleidet, und p99 macht diesen Tail sichtbar.

▼ Wenn 20 % eines Jobs seriell sind, wie groß ist die maximale Beschleunigung durch zusätzliche Prozessoren?

1 ÷ 0,2 = das 5-Fache, egal wie viele Prozessoren hinzukommen.

▼ **Probieren Sie es aus: Ein Bestellservice verarbeitet 2400 Anfragen pro Sekunde, von denen jede eine Datenbankverbindung 30 ms lang belegt. Er hat 8 Instanzen mit je 8 Verbindungen. Reicht das? Welche Poolgröße würden Sie pro Instanz festlegen, wenn Sie 70 % Auslastung anstreben?**

Nach Littles Gesetz sind im Schnitt 2400 x 0,030 = 72 Verbindungen belegt, und es gibt nur 8 x 8 = 64. Anfragen warten also auf Verbindungen, und die Wartezeit steigt zur Spitzenzeit. Für 70 % Auslastung brauchen Sie 72 / 0,7 = etwa 103 Verbindungen, also 103 / 8 = etwa 13 pro Instanz, insgesamt 104. Prüfen Sie, ob die Datenbank so viele verkraftet, und senken Sie die 30 ms, falls nicht.

QUELLEN

- Dean and Barroso, The Tail at Scale (Google Research): https://research.google/pubs/pub40801/
- Neil Gunther: How to quantify scalability (the USL): https://www.perfdynamics.com/Manifesto/USLscalability.html
- Google SRE book: Monitoring distributed systems (latency and percentiles): https://sre.google/sre-book/monitoring-distributed-systems/
- Brendan Gregg: The USE method: https://brendangregg.com/usemethod.html

Verfügbarkeit und Zuverlässigkeit: Neunen, MTBF, MTTR und Fehlertoleranz

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.7 Server und die Cloud
- 2.2 Qualitätsmodelle
- 2.3 Qualitätsszenarien

Begriffe von früher, die er verwendet: Server, Load Balancer, Komponente, Datenbank, Rechenzentrum

DIE IDEE IN KÜRZE

Denken Sie an einen Laden an der Ecke. Verfügbarkeit bedeutet, wie oft die Tür offen ist, wenn Sie ankommen. Zuverlässigkeit bedeutet, ob die Kasse, sobald Sie drinnen sind, Ihren ganzen Einkauf lang funktioniert. Läden bleiben offen, weil sie eine Ersatzkasse, eine zweite Arbeitskraft und einen schnellen Weg haben, Probleme zu bemerken und zu beheben. Bei Software ist es genauso: Architekten entscheiden, wie viel Reservekapazität sie vorhalten, wie sie Ausfälle schnell erkennen und wie schnell sie sich erholen, und sie drücken das Ziel als Zahl aus, etwa 99,9 % der Zeit.

WAS SIE WISSEN MÜSSEN

- Verfügbarkeit ist der Anteil der Zeit oder der Anfragen, in dem der Dienst bereit ist; Zuverlässigkeit ist die Wahrscheinlichkeit, über einen festgelegten Zeitraum ohne Ausfall zu laufen; beides ist verschieden, und ein System kann beim einen gut und beim anderen schlecht abschneiden.
- Eine Fehlerursache (Fault) ist eine Ursache, ein Fehler (Error) ein falscher interner Zustand und ein Ausfall (Failure) eine für Benutzer sichtbare Abweichung; Taktiken verhindern Fehlerursachen, erkennen Fehler oder verhindern, dass Fehler zu Ausfällen werden.
- Zulässige Ausfallzeit = (1 - Verfügbarkeit) x Zeitraum: 99,9 % sind etwa 8,76 Stunden pro Jahr oder 43,2 Minuten pro Monat; 99,99 % sind etwa 52,6 Minuten pro Jahr.
- A = MTBF / (MTBF + MTTR); daher ist es oft günstiger und wirksamer, die Reparaturzeit durch Automatisierung und Redundanz zu verkürzen, als Ausfälle seltener zu machen.

- In Reihe multiplizieren sich die Verfügbarkeiten; parallel multiplizieren sich die Nichtverfügbarkeiten, aber nur, wenn die Kopien unabhängig voneinander ausfallen und das Failover funktioniert.
- RTO (wie lange die Wiederherstellung dauern darf) und RPO (wie viele Daten Sie verlieren dürfen) kommen vom Fachbereich und bestimmen Design und Kosten der Disaster Recovery.
- SLIs messen, SLOs setzen interne Ziele und SLAs sind Verträge; ein SLO macht aus Verfügbarkeit ein Error Budget, das Teams für Änderungen ausgeben können.

Vier Begriffe, die oft verwechselt werden

Benutzer erleben ein System entweder als funktionierend oder nicht. Architekten brauchen genauere Begriffe, denn jeder Begriff verweist auf eine andere Antwort im Design. Verfügbarkeit ist der Anteil der Zeit (oder der Anfragen), in dem das System bereit ist, seinen Dienst auf Anforderung zu erbringen. Zuverlässigkeit ist die Wahrscheinlichkeit, dass das System über einen festgelegten Zeitraum unter festgelegten Bedingungen korrekt und ohne Ausfall arbeitet. Fehlertoleranz ist die Fähigkeit, den Dienst weiter zu erbringen, obwohl ein Teil ausgefallen ist. Resilienz ist die umfassendere Fähigkeit, Störungen aufzufangen, den wesentlichen Dienst (vielleicht eingeschränkt) aufrechtzuerhalten und sich danach wieder auf den Normalzustand zu erholen.

ISO/IEC 25010:2023 ordnet Verfügbarkeit, Fehlertoleranz und Wiederherstellbarkeit zusammen mit Fehlerfreiheit (die den Begriff Reife von 2011 ersetzt hat) als Teilmerkmale der Zuverlässigkeit ein. Bass, Clements und Kazman behandeln Verfügbarkeit in *Software Architecture in Practice* als eigenes Qualitätsmerkmal, weil sie so viele Architekturentscheidungen bestimmt. Keine der beiden Sichtweisen ist falsch; nutzen Sie die, die dem Gespräch hilft, aber sagen Sie ausdrücklich, welchen Begriff Sie meinen.

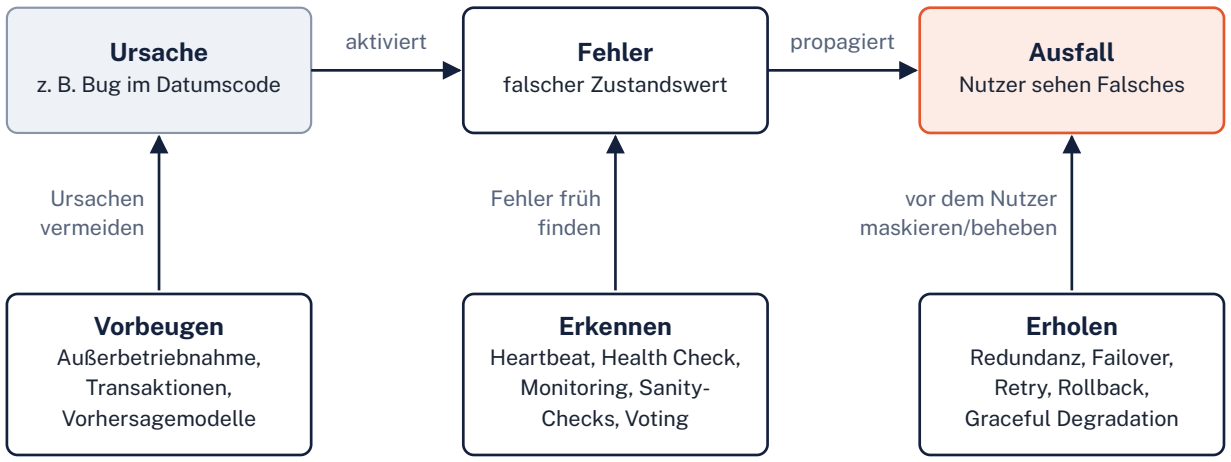
Der Unterschied zwischen Verfügbarkeit und Zuverlässigkeit ist in der Praxis wichtig. Stellen Sie sich einen Dienst vor, der einmal pro Stunde abstürzt und in einer Sekunde neu startet. Seine Verfügbarkeit beträgt 3599 / 3600 = 99,97 %, was hervorragend aussieht. Seine Zuverlässigkeit über einen zweistündigen Batch-Job ist schlecht: Treten Ausfälle zufällig mit einer Rate von einem pro Stunde auf, liegt die Wahrscheinlichkeit, zwei Stunden ohne Ausfall zu laufen, bei e^{-2}, etwa 13,5 %. Für ein Zahlungsterminal zählt die Verfügbarkeit; für eine lange wissenschaftliche Berechnung, einen Operationsroboter oder ein Flugzeug zählt die Zuverlässigkeit über die gesamte Mission.

Fehlerursache, Fehler und Ausfall

Die Dependability-Forschung (Avižienis, Laprie, Randell und Landwehr, 2004) trennt drei Begriffe, die die Alltagssprache vermischt. Eine Fehlerursache (Fault) ist die Ursache: ein Bug, eine defekte Festplatte, eine Fehlkonfiguration, ein abgelaufenes Zertifikat. Eine Fehlerursache kann jahrelang schlummern. Wird sie aktiviert, erzeugt sie einen Fehler (Error): einen Teil des internen Systemzustands, der falsch ist, etwa einen beschädigten Datensatz oder einen falschen Wert im Speicher. Erreicht der Fehler die Grenze des Dienstes, ist das Ergebnis ein Ausfall (Failure): Der Dienst, den die Benutzer erhalten, weicht davon ab, wie er sein sollte.

Bass, Clements und Kazman verwenden dieselbe Idee einfacher: Ein Ausfall ist eine Abweichung von der Spezifikation, die die Benutzer des Systems beobachten können, und die Ursache eines Ausfalls ist eine Fehlerursache. Für Architekten zählt, dass jedes Glied der Kette ein Ansatzpunkt ist. Sie können verhindern, dass Fehlerursachen entstehen (Code-Reviews, eine Komponente außer Betrieb nehmen, bevor sie verschleißt), Fehler erkennen, bevor sie sich ausbreiten (Validierung, Health Checks), und Fehler eindämmen oder maskieren, sodass sie nie zu Ausfällen werden (Redundanz, Wiederholungen, Transaktionen).

Die Kette und wo Taktiken ansetzen



Taktikfamilien nach Bass, Clements und Kazman, Software Architecture in Practice (4. Auflage, 2021)

Vorbeugende Taktiken verhindern, dass Fehlerursachen entstehen, erkennende Taktiken bemerken Fehler, und Taktiken zur Erholung verhindern, dass Fehler zu sichtbaren Ausfällen werden.

Die Neunen zählen

Verfügbarkeitsziele werden meist als Prozentsatz mit einer Anzahl von Neunen angegeben: 99,9 % sind drei Neunen. Es hilft, jedes Ziel in zulässige Ausfallzeit umzurechnen, denn ein Prozentsatz verschleiert, wie wenig Zeit Sie wirklich haben. Die Rechnung ist einfach:

Ausfallzeit = (1 - Verfügbarkeit) x Zeitraum. Ein Jahr mit 365 Tagen hat 525.600 Minuten, also erlauben 99,9 % 0,001 x 525.600 = 525,6 Minuten, etwa 8,76 Stunden. Für einen Monat mit 30 Tagen (43.200 Minuten) sind es 43,2 Minuten.

Jede weitere Neun teilt das Kontingent durch zehn, während die Entwicklungskosten meist viel schneller steigen. Bei 99,99 % haben Sie etwa 4,3 Minuten pro Monat: weniger als die Zeit, die vergeht, bis jemand alarmiert ist, aufwacht, sich anmeldet und versteht, was nicht stimmt. Oberhalb von drei oder vier Neunen muss die Wiederherstellung daher automatisch erfolgen; Menschen sind zu langsam, um Teil der Schleife zu sein.

Zulässige Ausfallzeit für gängige Verfügbarkeitsziele (Jahr mit 365 Tagen, Monat mit 30 Tagen)

Verfügbarkeit	Pro Jahr	Pro Monat	Pro Woche	Pro Tag
99 % (zwei Neunen)	3,65 Tage (87,6 h)	7,2 Stunden	1,68 Stunden (100,8 min)	14,4 Minuten
99,9 % (drei Neunen)	8,76 Stunden	43,2 Minuten	10,1 Minuten	1,44 Minuten
99.95%	4,38 Stunden	21,6 Minuten	5,04 Minuten	43,2 Sekunden
99,99 % (vier Neunen)	52,6 Minuten	4,32 Minuten	1,01 Minuten	8,64 Sekunden
99,999 % (fünf Neunen)	5,26 Minuten	25,9 Sekunden	6,05 Sekunden	0,86 Sekunden

ACHTUNG

Zeitbasiert oder anfragebasiert?

Die Tabelle setzt voraus, dass Verfügbarkeit als Betriebszeit über die Uhrzeit gemessen wird. Viele Onlinedienste messen sie stattdessen als Anteil erfolgreicher Anfragen: 999.000 gute Antworten von 1.000.000 sind 99,9 %. Die anfragebasierte Sicht erfasst Teilausfälle ganz natürlich (eine Region ist ausgefallen oder nur der Bezahlvorgang schlägt fehl) und gewichtet Stoßzeiten stärker als ruhige Stunden. Wofür Sie sich auch entscheiden, halten Sie genau fest, was als Erfolg zählt (Statuscodes, eine Latenzgrenze?), wo gemessen wird (am Load Balancer oder im Browser des Benutzers) und über welches Zeitfenster. Zwei Teams können beide 99,9 % für sich beanspruchen und damit sehr Verschiedenes meinen.

MTBF, MTTF und MTTR

Drei Mittelwerte verbinden Ausfälle und Reparaturen. Mean Time to Failure (MTTF) ist die durchschnittliche Zeit, die eine Komponente läuft, bevor sie ausfällt. Mean Time to Repair (MTTR) ist die durchschnittliche Zeit vom Ausfall bis zum wiederhergestellten Dienst (manche Organisationen lesen das R als „Recover“ oder „Restore“; die Idee ist dieselbe, aber sagen Sie, welche Uhr Sie verwenden: ab dem Ausfall, ab der Erkennung oder ab dem ersten Alarm). Mean Time Between Failures (MTBF) ist die durchschnittliche Zeit von einem Ausfall bis zum nächsten.

Die klassische Formel für den stationären Zustand lautet **A = MTBF / (MTBF + MTTR)**. Genau genommen gilt für ein reparierbares System MTBF = MTTF + MTTR, und der exakte Ausdruck ist MTTF / (MTTF + MTTR): Betriebszeit geteilt durch Betriebszeit plus Ausfallzeit. Lehrbücher schreiben oft MTBF, wo sie die mittlere Betriebszeit meinen; wenn Reparaturen im Vergleich zur Zeit zwischen Ausfällen kurz sind, wie es meist der Fall ist, ist der Unterschied vernachlässigbar. Die Formel setzt einen langfristigen Mittelwert voraus und sagt nichts darüber, wie Ausfälle verteilt sind; deshalb passt sie gut zu einem Zuverlässigkeitsmaß und zu Perzentilen der Reparaturzeit.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Ist es günstiger, seltener auszufallen oder sich schneller zu erholen?

Ein Bestellservice einer Essensliefer-App fällt im Durchschnitt einmal alle 30 Tage aus (MTBF = 720 Stunden). Jeder Ausfall braucht 4 Stunden bis zur Reparatur, weil ein Engineer alarmiert werden, das Problem diagnostizieren und eine Datenbank von Hand wiederherstellen muss. Das Unternehmen will besser als 99,9 % sein.

Aktuelle Verfügbarkeit	720 / (720 + 4) = 720 / 724 = 0,99448, also 99,45 %
Aktuelle Ausfallzeit pro Jahr	(1 - 0,99448) x 8760 h = etwa 48,4 Stunden
Option 1: MTBF verdoppeln	Bessere Hardware und Tests; Ausfälle alle 60 Tage. A = 1440 / 1444 = 0,99723 (99,72 %), etwa 24,3 Stunden pro Jahr
Option 2: MTTR auf 15 Minuten senken	Automatische Erkennung, ein Hot Standby und skriptgesteuertes Failover. A = 720 / 720,25 = 0,99965 (99,97 %), etwa 3,0 Stunden pro Jahr
Abgleich mit dem Ziel	Nur Option 2 übertrifft 99,9 % (8,76 Stunden pro Jahr)

Ausfälle halb so oft halbierten die Ausfallzeit. Die Reparaturzeit um den Faktor 16 zu verkürzen, senkte die Ausfallzeit um etwa denselben Faktor und erreichte das Ziel.

Eine höhere MTBF kämpft gegen Physik und Komplexität: Festplatten, Netzwerke und Menschen lassen sich nicht perfekt machen. Eine niedrigere MTTR liegt größtenteils in der Hand des Architekten: Erkennung, Automatisierung, Redundanz, kleine umkehrbare Deployments. Deshalb konzentriert sich der moderne Betrieb auf die Wiederherstellung.

MTTR ist ein Mittelwert und verbirgt lange Ausläufer. Ein achtstündiger Ausfall unter vielen fünfminütigen bestimmt das ganze Jahr; verfolgen Sie daher die Verteilung der Incident-Dauern, nicht nur den Mittelwert.

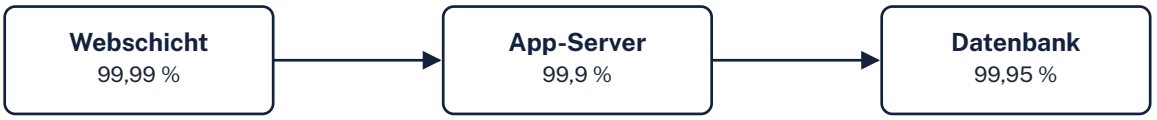
Komponenten in Reihe und parallel

Reale Systeme sind Ketten von Teilen. Braucht eine Anfrage die Webschicht, den Anwendungsserver und die Datenbank, müssen alle drei funktionieren: Die Komponenten sind *in Reihe* geschaltet, und ihre Verfügbarkeiten multiplizieren sich. Da jeder Faktor unter eins liegt, ist das Produkt immer niedriger als das schwächste Glied. Zehn Komponenten mit je 99,9 % ergeben $0{,}999^{10}$ = 0,990, nur 99,0 %.

Genügt dagegen eine beliebige von mehreren identischen Kopien, sind die Kopien *parallel* geschaltet (Redundanz). Das Paar fällt nur aus, wenn alle Kopien gleichzeitig ausfallen; daher multipliziert man die Nichtverfügbarkeiten: $A_{parallel} = 1 - (1 - A)^n$ für n identische Kopien. Zwei Server mit je 99 % ergeben 1 - 0,01 x 0,01 = 99,99 %.

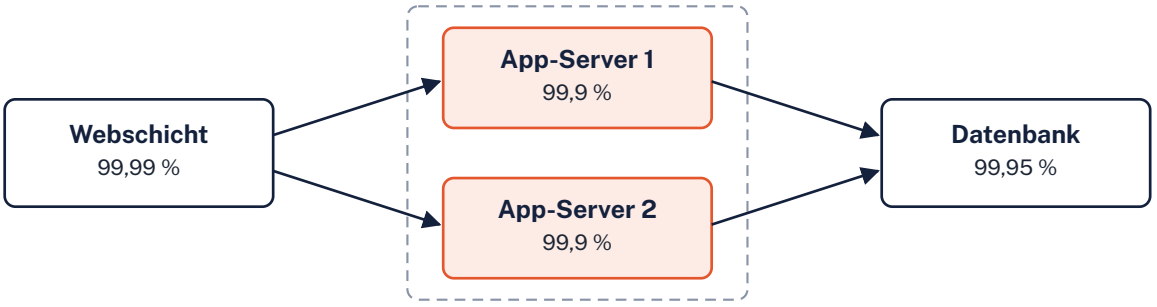
Reihe und parallel in Zahlen

A. Drei Komponenten in Reihe: jede muss funktionieren



0,9999 x 0,999 x 0,9995 = 0,9984 (99,84 %, etwa 14 h Ausfall pro Jahr)

B. Dieselbe Kette mit zwei parallelen App-Servern: einer genügt



Paar: 1 - (0,001 x 0,001) = 0,999999; Kette: 0,9994 (99,94 %, etwa 5,3 h pro Jahr)

Die schwächste Stufe zu verdoppeln, senkt die Ausfallzeit der ganzen Kette von etwa 14 Stunden pro Jahr auf etwa 5,3.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Wofür sollte das nächste Pfund ausgegeben werden?

Fortsetzung des Diagramms. Die Kette in Teil A erreicht 99,84 %. Welche Änderung bringt sie am nächsten an 99,99 %?

Einen zweiten App-Server hinzufügen	0,9999 x 0,999999 x 0,9995 = 0,99940 (99,94 %): etwa 5,3 Stunden pro Jahr
Dann ein Datenbankreplikat mit Failover hinzufügen	Datenbankpaar: $1 - 0{,}0005^2 = 0{,}99999975$. Kette: 0,9999 x 0,999999 x 0,99999975 = 0,99990 (99,99 %): etwa 53 Minuten pro Jahr
Was das System jetzt begrenzt	Die einzelne Webschicht (99,99 %) verursacht fast die gesamte verbleibende Ausfallzeit

Die Rechnung setzt voraus, dass die Kopien unabhängig voneinander ausfallen. Zwei Server im selben Rack, an derselben Stromversorgung, mit demselben fehlerhaften Release oder beide abhängig von einem einzigen DNS-Anbieter teilen Common-Mode-Fehlerursachen und können gemeinsam ausfallen. Die realen Gewinne sind immer kleiner, als die Formel nahelegt.

Sie setzt außerdem voraus, dass das Failover sofort erfolgt und immer funktioniert. Ein Standby, der nie in einem echten Failover getestet wurde, ist eine Hoffnung, keine Komponente.

Eine nützliche Regel der SRE-Autoren von Google (Treynor, Dahlin, Rau und Beyer, 2017): Jede kritische Abhängigkeit sollte ungefähr eine Neun mehr bieten als der Dienst, der auf ihr beruht, oder der Dienst braucht einen Weg, ohne sie zurechtzukommen.

Verfügbarkeitstaktiken: erkennen, erholen, vorbeugen

Bass, Clements und Kazman gliedern Verfügbarkeitstaktiken in drei Familien. Sie sind Designoptionen, keine Checkliste, die vollständig abzuarbeiten ist; jede hat ihren Preis.

Die drei Familien von Verfügbarkeitstaktiken

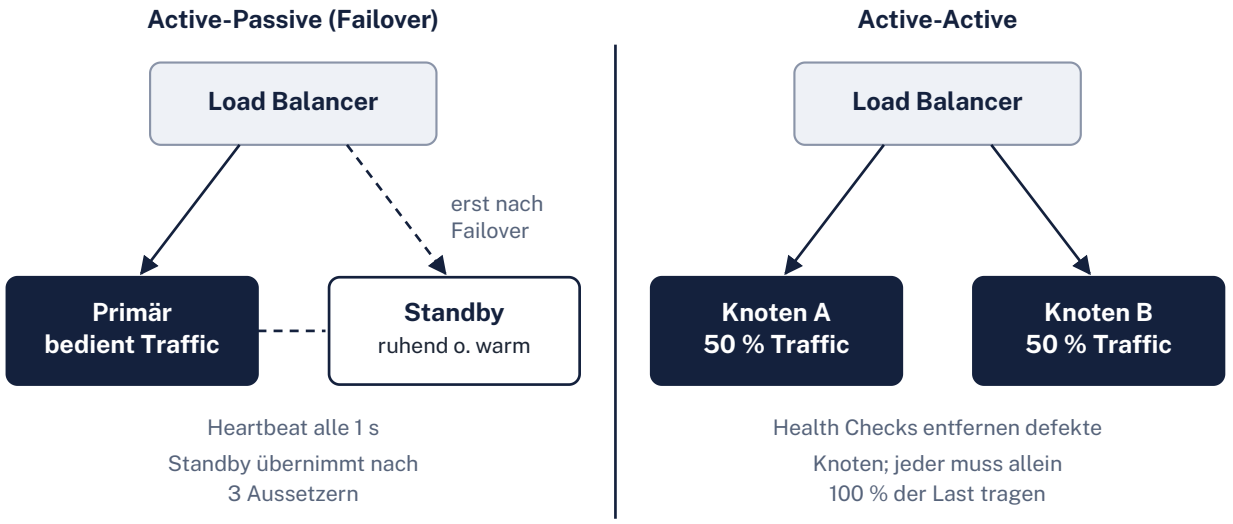
1. **Fehlerursachen erkennen**
   Heartbeat: Eine Komponente sendet regelmäßig eine Nachricht „Ich lebe“, und Stille löst eine Aktion aus. Ping/Echo: Ein Monitor fragt an und erwartet eine Antwort. Health Checks: ein Endpoint wie `/health`, den ein Load Balancer oder Orchestrator abfragt (ein Liveness-Check fragt „Läuft der Prozess?“, ein Readiness-Check fragt „Kann er jetzt Traffic bedienen?“). Monitoring von Fehlerraten und Latenz, Plausibilitätsprüfungen von Werten und Abstimmung (Voting) zwischen Replikaten, die dasselbe Ergebnis berechnen.

2. **Erholen: Vorbereitung und Reparatur**
   Redundante Reserven in drei Temperaturen: *Active-Active* (alle Kopien bedienen Live-Traffic), *Hot oder Warm Standby* (eine synchron gehaltene, bereite Kopie) und *Cold Spare* (wird erst bei Bedarf gestartet).
   Failover: leitet den Traffic auf eine gesunde Kopie um. Wiederholung (Retry) behandelt vorübergehende Fehlerursachen (mit Grenzen und Backoff, sonst werden Wiederholungen zur selbst verursachten Überlast). Rollback kehrt zu einem bekannten guten Zustand oder Release zurück.
   Graceful Degradation: hält den Kerndienst ohne eine ausgefallene Funktion am Laufen: Der Shop nimmt weiter Bestellungen an, auch wenn die Empfehlungen nicht verfügbar sind.

3. **Erholen: Wiedereingliederung**
   Eine reparierte Komponente sicher zurückholen: im Schattenmodus betreiben, ihren Zustand neu synchronisieren, in eskalierenden Stufen neu starten (erst ein Thread, dann ein Prozess, dann eine Maschine).

4. **Fehlerursachen vorbeugen**
   Außerbetriebnahme (einen Knoten für Wartung oder einen geplanten Neustart aus der Rotation nehmen, bevor ein bekanntes Speicherleck zuschlägt), Transaktionen (Alles-oder-nichts-Aktualisierungen, damit ein Absturz keinen halb geschriebenen Zustand hinterlassen kann), Vorhersagemodelle (eine Festplatte austauschen, wenn ihre Fehlerzähler steigen) und Ausnahmevermeidung (strengere Typen, Eingabevalidierung).

Zwei Formen von Redundanz



Active-Passive bezahlt für ungenutzte Kapazität und eine Failover-Lücke; Active-Active nutzt die ganze Kapazität, aber jeder Knoten muss den Verlust des anderen überstehen.

Die Wahl zwischen Active-Passive und Active-Active

Active-Passive ist leichter zu durchschauen, besonders bei zustandsbehafteten Komponenten wie einer relationalen Datenbank mit einem einzigen beschreibbaren Primärknoten. Die Kosten sind der ungenutzte Standby, eine Failover-Lücke (Erkennung plus Hochstufung, oft einige zehn Sekunden) und das Risiko, dass der Standby unbemerkt aus dem Takt geraten ist. Eine klassische Falle ist *Split Brain*: Das Netzwerk zwischen beiden bricht ab, jeder hält den anderen für tot, und beide nehmen Schreibvorgänge an. Dagegen hilft ein Quorum oder ein externer Schiedsrichter.

Active-Active nutzt die gesamte Kapazität und hat keinen Hochstufungsschritt, aber jeder Knoten muss so dimensioniert sein, dass er die volle Last tragen kann, wenn sein Partner ausfällt. Zwei Knoten mit je 70 % Auslastung werden bei Ausfall des einen mit 140 % belastet, und der Überlebende bricht zusammen: ein redundantes Design, das trotzdem ausfällt. Active-Active zwingt Sie außerdem, die Datenkonsistenz über Kopien hinweg zu lösen, was ein großes Thema für sich ist (siehe Konsistenzmodelle, CAP und PACELC).

Disaster Recovery: RTO und RPO

Redundanz innerhalb eines Rechenzentrums schützt Sie nicht davor, das ganze Rechenzentrum, eine Region, ein Cloud-Konto oder Ihre Daten durch Ransomware zu verlieren. Die Disaster-Recovery-Planung arbeitet mit zwei Zahlen, die vom Fachbereich kommen sollten, nicht von den Engineers. Das Recovery Time Objective (RTO) ist die längste akzeptable Zeit, um

den Dienst nach einer Katastrophe wiederherzustellen. Das Recovery Point Objective (RPO) ist der größte akzeptable Datenverlust, gemessen in Zeit: Ein RPO von 15 Minuten bedeutet, dass Sie höchstens die Änderungen der letzten 15 Minuten verlieren dürfen.

Die beiden Zahlen führen zu sehr unterschiedlichen Designs und Kosten. Nächtliche Backups, die in eine andere Region kopiert werden, ergeben ein RPO von bis zu 24 Stunden und ein RTO von vielen Stunden (Terabytes wiederherzustellen dauert). Kontinuierliche asynchrone Replikation zu einem Warm Standby in einer anderen Region kann ein RPO von Sekunden und ein RTO von Minuten ergeben, bei vielleicht doppelten Infrastrukturkosten. Synchrone Replikation kann das RPO nahe null bringen, fügt aber jedem Schreibvorgang die Round-Trip-Zeit zwischen den Standorten hinzu. Cloud-Anbieter beschreiben eine Stufenleiter von Strategien: Backup and Restore, Pilot Light, Warm Standby und Multi-Site Active-Active, jede schneller und teurer als die vorige. Die Einzelheiten gehören zu Multi-Region und Disaster Recovery.

Unabhängig von der Strategie gilt: Ein Backup, das Sie nie wiederhergestellt haben, ist kein Backup. Testen Sie Wiederherstellungen regelmäßig und messen Sie die Zeit; die gemessene Wiederherstellungszeit ist Ihr echtes RTO.

Von Zielen zu SLIs, SLOs und SLAs

Der moderne Betrieb drückt Verfügbarkeit mit drei verwandten Begriffen aus, die in SLIs, SLOs und Error Budgets ausführlich behandelt werden. Ein Service Level Indicator (SLI) ist die Messung, zum Beispiel der Anteil der Checkout-Anfragen, die in weniger als 800 ms erfolgreich sind. Ein Service Level Objective (SLO) ist das interne Ziel für diese Messung über ein Zeitfenster, zum Beispiel 99,9 % über 30 Tage. Ein Service Level Agreement (SLA) ist ein Vertrag mit Kunden, der Folgen benennt, etwa Gutschriften, wenn ein Niveau verfehlt wird; es wird normalerweise lockerer angesetzt als das SLO, damit das Team gewarnt ist, bevor der Vertrag verletzt wird.

Das SLO macht aus Verfügbarkeit ein Budget. Ein monatliches SLO von 99,9 % erlaubt, dass 0,1 % der Anfragen fehlschlagen; bei 2 Millionen Anfragen pro Monat sind das 2000 fehlgeschlagene Anfragen oder etwa 43 Minuten Totalausfall. Teams geben dieses Error Budget für Risiken aus: Releases, Experimente, Migrationen. Ist es aufgebraucht, treten sie kürzer und investieren in Zuverlässigkeit. So verbinden sich die abstrakten Neunen dieses Abschnitts mit alltäglichen Entscheidungen.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Sie müssen die Tabelle der Neunen nicht auswendig lernen. Merken Sie sich die Methode: eins minus den Prozentsatz, mal den Zeitraum. 99,9 % lassen 0,1 % eines Monats mit 30 Tagen übrig, also 0,001 x 43.200 Minuten = 43,2 Minuten. Jede andere Zeile ist dieselbe Rechnung.

Häufige Fehler

Neunen versprechen, ohne die Abhängigkeiten mitzuzählen: Ein Dienst mit 99,99 %, der auf einer einzigen Datenbank mit 99,9 % aufbaut, kann sein Ziel nicht erreichen. Redundanz hinzufügen, ohne gemeinsame Ursachen zu beseitigen (dasselbe Release, derselbe Konfigurations-Push, dasselbe Zertifikat, das um Mitternacht auf jedem Knoten abläuft). Verfügbarkeit innerhalb des Systems messen, wo alles grün aussieht, während Benutzer es wegen DNS oder eines Content Delivery Network nicht erreichen. Failover planen, aber nie proben; Chaos Engineering gibt es genau dafür, diese Pfade bewusst zu üben (Chaos Engineering). Schließlich: einer Zahl um ihrer selbst willen nachjagen. Ein Dienstplansystem, das während der Bürozeiten genutzt wird, braucht keine fünf Neunen, und das Geld ist anderswo besser angelegt.

Um Verfügbarkeit testbar zu machen, schreiben Sie sie als Szenario (Abschnitt 2.3): „Fällt der primäre Datenbankserver während der Stoßzeiten aus, erkennt das System dies innerhalb von 10 Sekunden und nimmt innerhalb von 60 Sekunden wieder Bestellungen an, ohne eine bestätigte Bestellung zu verlieren.“ Jeder Teil dieses Satzes wird zu einer Designentscheidung und zu einem Test.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Wie viele Minuten Ausfallzeit erlaubt ein Ziel von 99,95 % in einem Monat mit 30 Tagen?

0,0005 x 43.200 = 21,6 Minuten.

▼ Zwei unabhängige Load Balancer haben je 99,5 % Verfügbarkeit, und jeder kann den gesamten Traffic tragen. Wie hoch ist die Verfügbarkeit des Paars?

1 - (0.005 x 0.005) = 1 - 0.000025 = 99.9975%.

▼ **Warum bringt ein zweiter, identischer Server womöglich viel weniger, als die Parallelformel vorhersagt?**

Weil die Kopien gemeinsame Ausfallursachen haben können, etwa dasselbe Release, dieselbe Konfiguration, Stromversorgung, dasselbe Netzwerk oder dieselbe Abhängigkeit, sodass sie nicht unabhängig voneinander ausfallen; außerdem kann das Failover selbst langsam oder defekt sein.

▼ **Die Backups eines Teams laufen alle 6 Stunden, und eine vollständige Wiederherstellung dauert 3 Stunden. Wie hoch sind ihr effektives RPO und RTO im ungünstigsten Fall?**

Ein RPO von bis zu 6 Stunden verlorener Daten und ein RTO von mindestens 3 Stunden plus der Zeit, um die Katastrophe zu erkennen und die Wiederherstellung zu beschließen.

▼ **Probieren Sie es aus: Eine Kette besteht aus Gateway mit 99,95 %, Bestellservice mit 99,9 % und Zahlung mit 99,9 %. Wie hoch ist ihre Verfügbarkeit, und wie hoch die Ausfallzeit pro Jahr? Fügen Sie dann eine zweite, unabhängige Bestellinstanz mit funktionierendem Failover hinzu. Rechnen Sie neu, und nennen Sie die Annahme, die das Ergebnis zunichtemachen könnte.**

In Reihe: 0,9995 x 0,999 x 0,999 = 0,99750, etwa 99,75 %, also 21,9 Stunden im Jahr. Zwei Bestellinstanzen ergeben 1 - 0,001 x 0,001 = 0,999999, die Kette also 0,9995 x 0,999999 x 0,999 = 0,99850, etwa 13,1 Stunden. Zahlung und Gateway dominieren jetzt. Die Annahme ist Unabhängigkeit: Ein gemeinsames fehlerhaftes Release, eine gemeinsame Stromversorgung oder Datenbank bedeutet, dass beide Kopien zusammen ausfallen, und der Failover muss wirklich funktionieren.

QUELLEN

- Google SRE book: Embracing Risk (time-based and request-based availability): https://sre.google/sre-book/embracing-risk/
- Google SRE book: Service Level Objectives: https://sre.google/sre-book/service-level-objectives/
- SEI blog: Tactics and Patterns for Software Robustness (availability tactics): https://www.sei.cmu.edu/blog/tactics-and-patterns-for-software-robustness/
- AWS whitepaper: Disaster recovery options in the cloud: https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-options-in-the-cloud.html
- Avižienis, Laprie, Randell and Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing (IEEE TDSC, 2004): https://doi.org/10.1109/TDSC.2004.2

Änderbarkeit, Testbarkeit und Deploybarkeit: auf Veränderung auslegen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.3 Code organisieren
- 0.8 Wie Software entsteht
- 2.3 Qualitätsszenarien

Begriffe von früher, die er verwendet: Feature-Flag, Schnittstelle, Modul, Unit-Test, Produktion

DIE IDEE IN KÜRZE

Denken Sie an eine Küche. Ist der Herd in die Wand eingebaut und sind die Leitungen in Beton verlegt, muss man zum Austauschen die Wand einreißen. Wird er an eine Standardsteckdose angeschlossen, tauschen Sie ihn an einem Nachmittag aus. Bei Software ist es genauso: Manche Teile werden sich mit Sicherheit ändern, deshalb geben Architekten ihnen Standardanschlüsse, machen es leicht zu prüfen, ob nach einem Austausch noch alles funktioniert, und machen die Auslieferung der neuen Version zur Routine statt zu einem riskanten Ereignis.

WAS SIE WISSEN MÜSSEN

- Änderbarkeit beschreibt Kosten und Risiko von Änderungen; spezifizieren Sie sie, indem Sie angeben, was sich wahrscheinlich ändert, wer die Änderung vornimmt, wann sie gebunden wird und was die Änderung kosten darf.
- Taktiken für Änderbarkeit erhöhen die Kohäsion, verringern die Kopplung (kapseln, einen Vermittler nutzen, Abhängigkeiten beschränken) und verschieben die Bindung, doch jeder spätere Bindungszeitpunkt bringt zusätzliche Mechanik mit sich.
- Testbarkeit hängt von Steuerbarkeit und Beobachtbarkeit ab; isolieren Sie Abhängigkeiten hinter Schnittstellen und injizieren Sie sie, damit Tests Test Doubles verwenden können.
- Deploybarkeit beruht auf kleinen Batches, Automatisierung, unabhängig deploybaren Einheiten, abwärtskompatiblen Änderungen, schrittweisem Rollout und schnellem Rollback.

- Die aktuellen DORA-Metriken für die Softwareauslieferung sind Durchlaufzeit von Änderungen, Deployment-Häufigkeit, Wiederherstellungszeit nach fehlgeschlagenem Deployment, Änderungsfehlerrate und Deployment-Nacharbeitsrate.
- Wartbarkeit hat in ISO/IEC 25010:2023 fünf Teilmerkmale: Modularität, Wiederverwendbarkeit, Analysierbarkeit, Änderbarkeit und Testbarkeit.

Drei Qualitäten rund um Veränderung

Der größte Teil der Kosten eines erfolgreichen Systems entsteht nach seinem ersten Release, wenn es wieder und wieder geändert wird. Drei verwandte Qualitäten beschreiben, wie gut eine Architektur das unterstützt. Bei Änderbarkeit geht es um Kosten und Risiko einer Änderung. Bei Testbarkeit geht es darum, wie leicht Sie durch Tests zeigen können, dass sich das System (oder ein Teil davon) wie beabsichtigt verhält, und wie bereitwillig sich Fehlerursachen zeigen. Bei Deploybarkeit geht es darum, wie schnell, sicher und vorhersehbar eine Änderung in Produktion gebracht und wieder zurückgenommen werden kann.

Sie hängen in einem Kreislauf zusammen. Eine Änderung ist nur dann günstig, wenn Sie sie an einer Stelle vornehmen können (Änderbarkeit), schnell wissen, dass Sie nichts kaputt gemacht haben (Testbarkeit), und sie ohne großes Zeremoniell ausliefern können (Deploybarkeit). Ein modulares System ohne schnelle Tests ändert sich trotzdem langsam, weil jede Änderung wochenlange manuelle Regressionstests braucht. Ein System mit guten Tests, aber einem vierteljährlichen Release-Fenster liefert trotzdem langsam Nutzen.

Änderbarkeit: vier Fragen zu jeder Änderung

Bass, Clements und Kazman legen nahe, dass Änderbarkeit bedeutungslos ist, solange Sie nicht sagen, was sich voraussichtlich ändern wird. Wer für jede denkbare Änderung plant, erhält ein überdimensioniertes System voller ungenutzter Erweiterungspunkte; wer für keine plant, erhält ein System, in dem jede Änderung eine Neuentwicklung ist. Vier Fragen schärfen die Anforderung.

Was kann sich ändern? Funktionen, Plattformen (eine neue Datenbankversion, der Umzug in eine andere Cloud), die Umgebung (eine neue Partner-API, neue Vorschriften), Qualitäten (die doppelte Last) oder die Kapazität. **Wie wahrscheinlich ist die Änderung?** Investieren Sie Designaufwand dort, wo Änderungen wahrscheinlich sind: Zahlungsanbieter, Steuerregeln und Benutzeroberflächen ändern sich oft; das Konzept einer Bestellung selten. **Wer nimmt die Änderung vor, und wann?** Ein Entwickler beim Programmieren, ein Release-Engineer beim Deployment, ein Administrator über die Konfiguration oder ein Endbenutzer über Einstellungen. Der Moment, in dem eine Entscheidung festgelegt wird, heißt ihr

Bindungszeitpunkt. **Was kostet die Änderung?** Aufwand, Dauer und Geld, um sie umzusetzen, zu testen und zu deployen, dazu die Zahl der anderen betroffenen Elemente und das Risiko, Defekte einzuführen.

Die Antworten auf diese Fragen ergeben ein Szenario der Art, wie sie in Abschnitt 2.3 eingeführt wurde: „Ein Entwickler fügt die Unterstützung für einen neuen Zahlungsanbieter hinzu, ändert außer der Konfiguration keine bestehenden Module, braucht dafür einschließlich Tests höchstens 3 Personentage, und die Fehler im Bezahlvorgang nehmen nicht zu.“

Änderbarkeitsszenarien mit unterschiedlichen Quellen und Bindungszeitpunkten

Wer ändert	Was sich ändert	Wann es bindet	Antwortmaß (Response Measure)
Entwickler	Einen neuen Lieferpartner zu einer Essenslieferplattform hinzufügen	Programmierzeit	Nur der Partner-Adapter ändert sich; umgesetzt, getestet und deployt innerhalb von 5 Tagen
Produktmanager	Einen neuen Checkout-Ablauf für 10 % der Benutzer einschalten	Laufzeit (Feature-Flag)	Wirkt innerhalb von 1 Minute, ohne Deployment
Operations-Engineer	Das System auf einen neuen Datenbankhost umstellen	Deployment (Konfiguration)	Keine Codeänderung; ein Pipeline-Lauf von unter 15 Minuten
Administrator beim Kunden	Ein benutzerdefiniertes Feld zu einer Rechnung hinzufügen	Laufzeit (Einstellungen)	Ohne Beteiligung des Anbieters; sofort verfügbar
Plattformteam	Das Web-Framework auf eine neue Hauptversion aktualisieren	Build-Zeit	Weniger als 20 Dateien ändern sich außerhalb eines Anti-Corruption Layer; unter 2 Wochen

Taktiken für Änderbarkeit

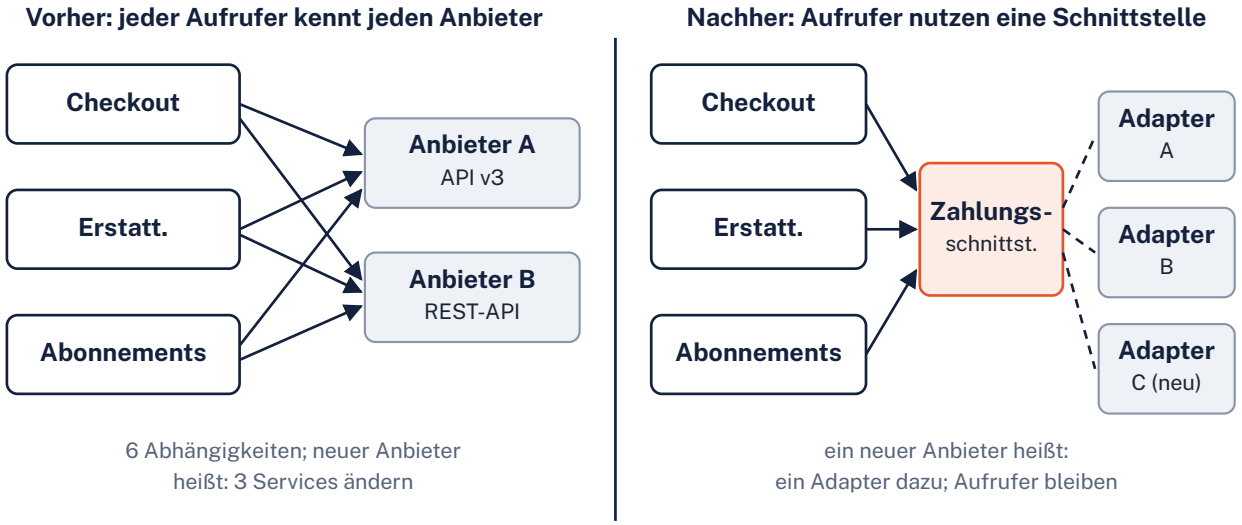
Die Taktiken in *Software Architecture in Practice* senken die Kosten von Änderungen, indem sie begrenzen, wie weit eine Änderung ausstrahlt. Sie beruhen auf zwei alten Ideen, die in Kapitel 3 ausführlich behandelt werden. Kohäsion misst, wie stark die Verantwortlichkeiten innerhalb eines Moduls zusammengehören; Kopplung misst, wie stark ein Modul von einem anderen abhängt. Eine wahrscheinliche Änderung sollte ein einziges kohäsives Modul betreffen, und dieses Modul sollte lose an den Rest gekoppelt sein.

Kohäsion erhöhen. Teilen Sie ein Modul auf, das zwei unzusammenhängende Aufgaben erledigt (ein `CustomerService`, der sowohl die Anmeldung als auch Treuepunkte verwaltet), und verschieben Sie Verantwortlichkeiten dorthin, wo sie hingehören, damit eine wahrscheinliche Änderung an einer Stelle landet.

Kopplung verringern. *Kapseln*: Verbergen Sie die Interna eines Moduls hinter einer expliziten Schnittstelle, nach David Parnas' Prinzip des Information Hiding (1972), damit Aufrufer nicht von dem abhängen können, was sich ändern kann. *Einen Vermittler nutzen*: Setzen Sie etwas zwischen zwei Beteiligte, sodass keiner den anderen direkt kennt: einen Adapter, einen Message Broker, einen Publish-Subscribe-Kanal, ein Repository vor einer Datenbank oder ein API-Gateway. *Abhängigkeiten beschränken*: Legen Sie fest, welche Module welche aufrufen dürfen (Schichtregeln, ein Modul, das nur im eigenen Paket sichtbar ist), und setzen Sie das mit Werkzeugen durch. *Gemeinsame Dienste abstrahieren*: Stellen Sie gemeinsames Verhalten einmal hinter einer Schnittstelle bereit, statt es zu kopieren.

Bindung verschieben. Legen Sie eine Entscheidung so spät fest, wie es nützlich ist, damit die Änderung später und von jemandem weiter weg vom Code vorgenommen werden kann: Konfigurationsdateien, Umgebungsvariablen, beim Start erkannte Plug-ins, Service Discovery und Feature-Flags. Jeder spätere Bindungszeitpunkt bringt zusätzliche Mechanik mit sich (zu validierende Konfiguration, zu entfernende Flags, versionierte Plug-in-Schnittstellen); verschieben Sie daher nur Entscheidungen, die wirklich variieren.

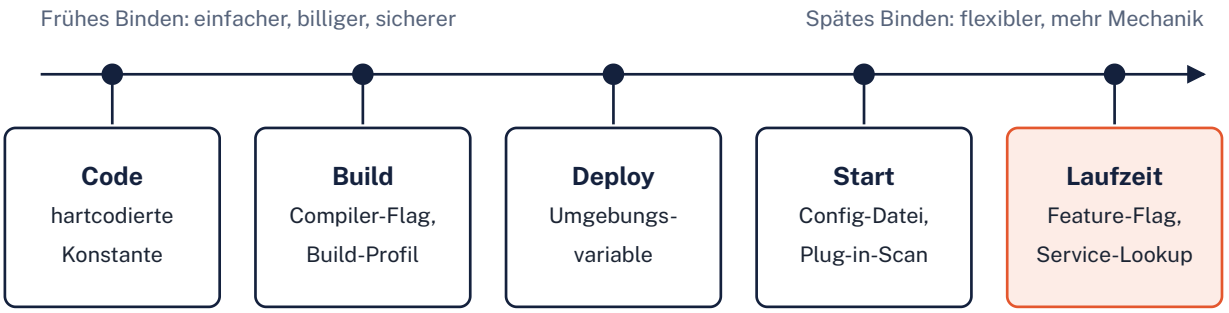
Mit einem Vermittler Änderungen eindämmen



Gestrichelt: Adapter realisieren die Schnittstelle

Eine Schnittstelle mit einem Adapter pro Anbieter macht aus einer Änderung, die drei Services betraf, das Hinzufügen eines einzigen Adapters.

Bindungszeitpunkt



Jeder Schritt nach rechts erlaubt eine Änderung später, durch jemanden weiter weg vom Code, ohne neues Release

Wird eine Entscheidung nach rechts verschoben, kann sie sich später und ohne Release ändern, doch jeder Schritt bringt zusätzliche Mechanik, die verwaltet werden muss.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Wie spät sollte diese Entscheidung gebunden werden?

Nutzen Sie sie, wenn ein Stakeholder sagt: „Das könnte sich ändern.“ Beginnen Sie oben und hören Sie beim ersten Ja auf.

Q1 **Ist die Änderung unwahrscheinlich, oder bräuchte sie ohnehin neuen Code und neue Tests?**

Ja Binden Sie zur Programmierzeit. Halten Sie es einfach und gut gekapselt, damit die Änderung lokal bleibt, wenn sie kommt.

Nein Weiter zur nächsten Frage.

Q2 **Unterscheidet sich der Wert nur zwischen Umgebungen (Test, Staging, Produktion) oder zwischen den Installationen verschiedener Kunden?**

Ja Binden Sie beim Deployment: Umgebungsvariablen oder von der Pipeline eingespeiste Konfiguration, versioniert in der Versionsverwaltung.

Nein Weiter zur nächsten Frage.

Q3 **Müssen Dritte oder andere Teams Verhalten hinzufügen, ohne Ihren Code zu ändern?**

Ja Binden Sie beim Start oder zur Laufzeit mit einer Plug-in-oder Erweiterungsschnittstelle, und versionieren Sie diese Schnittstelle sorgfältig.

Nein Weiter zur nächsten Frage.

Q4 **Müssen Nicht-Entwickler Verhalten schnell ein- oder ausschalten können, oder müssen Sie zuerst für einen Teil der Benutzer freigeben?**

Ja Binden Sie zur Laufzeit mit einem Feature-Flag, und geben Sie jedem Flag einen Verantwortlichen und ein Entfernungsdatum.

Nein Binden Sie zur Build-oder Deployment-Zeit; späteres Binden würde Mechanik hinzufügen, die niemand braucht.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Was ist das Änderbarkeitsszenario wert?

Ein Essenslieferdienst bindet die Kassensysteme von Restaurants an. Heute erfordert jeder neue Kassensystem-Anbieter Änderungen an den Services für Speisekarte, Bestellungen und Erstattungen: etwa 15 Personentage Arbeit plus einen Regressionstestzyklus von 5 Tagen. Der Vertrieb erwartet in den nächsten 12 Monaten 8 neue Anbieter. Der Architekt schlägt eine Adapter-Schnittstelle für Anbieter vor (wie im Diagramm oben). Nehmen Sie Vollkosten von £600 pro Personentag an.

Kosten pro Anbieter heute	15 + 5 = 20 Personentage, also 20 x £600 = £12.000
Kosten für 8 Anbieter heute	8 x £12,000 = £96,000
Einmalige Kosten des Refactorings	Schnittstelle entwerfen und bauen und die 3 bestehenden Anbieter migrieren: 30 Personentage = £18.000
Kosten pro Anbieter danach	Einen Adapter schreiben (3 Tage) plus Contract Tests gegen die Schnittstelle (1 Tag): 4 Personentage = £2400
Kosten für 8 Anbieter danach	£18,000 + 8 x £2,400 = £37,200
Einsparung im ersten Jahr	£96,000 -£37,200 = £58,800
Break-even	Jeder Anbieter spart £12.000 -£2400 = £9.600, also sind die £18.000 nach £18.000 / £9600 = 1,9 zurückverdient, das heißt mit dem zweiten neuen Anbieter

Die Rechnung ist nur so gut wie die Vorhersage der Änderungen. Gewinnt der Vertrieb keine neuen Anbieter, ist das Refactoring reine Kosten. Deshalb müssen Änderbarkeitsanforderungen sagen, was sich wahrscheinlich ändert.

Die Durchlaufzeit zählt oft mehr als die Kosten: Wird der Start eines Anbieters von etwa 4 Wochen auf unter 1 Woche verkürzt, kann das Aufträge gewinnen, was keine Personentage-Rechnung erfasst.

Die neue Adapter-Schnittstelle ist selbst eine Festlegung. Sie später zu ändern, betrifft jeden Adapter; daher verdient sie sorgfältiges Design und Versionierung.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: entscheiden, wann eine Wahl festgelegt wird

Die schwierigste Idee dieses Abschnitts ist der Bindungszeitpunkt: der Moment, in dem eine Entscheidung festgelegt wird, der bestimmt, wer sie später ändern kann und wie leicht. Verfolgen Sie eine Entscheidung durch drei Bindungszeitpunkte anhand des Diagramms dieses Abschnitts, in dem der Checkout die Zahlungsanbieter über eine einzige Zahlungsschnittstelle mit einem Adapter pro Anbieter erreicht.

- Adapter A und Adapter B verbinden die Zahlungsschnittstelle jeweils mit einem Anbieter. Angenommen, die Entscheidung lautet, welchen Adapter der Checkout verwendet.
- Programmierzeit: Ein Entwickler schreibt „verwende Adapter A“ in den Code. Das ist die einfachste Wahl, aber der Wechsel zu B bedeutet eine Codeänderung, Tests und ein neues Release, und nur ein Entwickler kann ihn vornehmen.
- Deployment-Zeit: Die Wahl wandert in die Konfiguration, etwa in eine Umgebungsvariable, die die Deployment-Pipeline liefert. Test und Produktion können nun verschiedene Anbieter nutzen, und ein Operations-Engineer kann den Anbieter mit einem Pipeline-Lauf und ohne Codeänderung wechseln.
- Laufzeit: Ein Feature-Flag leitet 10 % der Zahlungen über Adapter B, und ein Produktmanager kann diesen Anteil innerhalb einer Minute erhöhen oder senken, ohne Deployment.
- Jeder spätere Schritt bringt zusätzliche Mechanik mit sich. Die Konfiguration muss geprüft werden, damit ein vertippter Wert keine Zahlungen stoppen kann; das Flag braucht einen Verantwortlichen und ein Entfernungsdatum; und solange beide Adapter live sein können, müssen beide getestet bleiben.
- Einen neuen Adapter C hinzuzufügen, bedeutet weiterhin neuen Code und neue Tests, egal bei welchem Bindungszeitpunkt. Das Verschieben der Bindung ändert, wann und von wem eine Wahl zwischen bestehenden Optionen getroffen wird.
- **Die Lehre:** Fragen Sie, wer eine Entscheidung ändern muss und wie oft, und wählen Sie dann den frühesten Bindungszeitpunkt, der das ermöglicht. Später zu binden, bringt nur Mechanik, die niemand nutzt.

Testbarkeit: Steuerbarkeit und Beobachtbarkeit

Eine Komponente ist testbar, wenn ein Test sie in einen bekannten Zustand versetzen, ihr gewählte Eingaben geben und sehen kann, was sie getan hat. Zwei Eigenschaften erfassen das. **Steuerbarkeit** ist die Fähigkeit, die Eingaben und den internen Zustand einer Komponente aus einem Test heraus festzulegen. **Beobachtbarkeit** (im Sinne des Testens) ist die Fähigkeit,

ihre Ausgaben und den relevanten internen Zustand zu sehen. Code, der die Systemuhr direkt liest, einen echten Zahlungsdienst aufruft, Zufallszahlen wählt oder in eine gemeinsame Datenbank schreibt, ist schwer zu steuern; Code, der seine Ergebnisse in privatem Zustand oder nur im Logtext verbirgt, ist schwer zu beobachten.

Architekturtaktiken für Testbarkeit zielen daher darauf, Tests Kontrolle und Einblick zu geben und die Komplexität zu begrenzen. **Isolieren Sie Abhängigkeiten** hinter Schnittstellen und nutzen Sie dann Dependency Injection: Die Komponente erhält ihre Kollaborateure (eine Uhr, ein Payment-Gateway, ein Repository) von außen, statt sie selbst zu erzeugen, sodass ein Test einen Ersatz übergeben kann. Diese Ersatzobjekte sind Test Doubles, ein von Gerard Meszaros bekannt gemachter Begriff: Ein *Stub* liefert vorgefertigte Antworten, ein *Fake* ist eine funktionierende, leichtgewichtige Implementierung (ein In-Memory-Repository), ein *Mock* prüft, dass erwartete Aufrufe erfolgt sind, und ein *Spy* zeichnet Aufrufe für spätere Assertions auf. **Stellen Sie Testschnittstellen bereit**: Hooks, um Zustand zu setzen, Traffic aufzuzeichnen und wiederzugeben oder in einer Sandbox-Umgebung zu laufen. **Begrenzen Sie Nichtdeterminismus**: Vermeiden Sie versteckte Nebenläufigkeit und globalen Zustand, und machen Sie Zeit und Zufall injizierbar. **Begrenzen Sie strukturelle Komplexität**: Weniger Abhängigkeiten und keine Zyklen bedeuten kleinere Test-Setups.

Die Architektur prägt die Teststrategie auch auf höheren Ebenen. In einem System aus Services braucht jede Grenze Contract Tests, die prüfen, dass beide Seiten sich über die API einig sind, und einige End-to-End-Tests, die den echten Pfad durchlaufen. Ein Design mit klaren Grenzen lässt die meisten Tests schnell und isoliert laufen, während ein verworrenes Design langsame, fragile End-to-End-Tests für alles erzwingt.

Dependency Injection macht eine Komponente steuerbar



Derselbe OrderService läuft in Produktion gegen das echte Gateway und in Tests gegen ein skriptgesteuertes Fake, weil er nur von der Schnittstelle abhängt.

```
class OrderService:
    def __init__(self, gateway, clock):
        self.gateway = gateway   # anything with charge(amount, card)
        self.clock = clock       # anything with now()

    def place(self, order):
        if self.clock.now().hour ⩾ 23:
            return "kitchen closed"
        return self.gateway.charge(order.total, order.card)

# In a test: full control, no network, no waiting until 11 pm
class FakeClock:
    def __init__(self, t): self.t = t
    def now(self): return self.t

class FakeGateway:
    def __init__(self): self.calls = []
    def charge(self, amount, card):
        self.calls.append(amount)
        return "approved"
```

Konstruktorinjektion: Der Service erhält seine Uhr und sein Gateway, sodass ein Test Fakes übergeben kann.

Deploybarkeit

Deploybarkeit misst, wie vorhersehbar und günstig eine Änderung vom Commit eines Entwicklers in die Produktion gelangt und wie schnell sie rückgängig gemacht werden kann. Zu einer vollwertigen Qualität wurde sie erst im letzten Jahrzehnt, als Continuous Delivery zeigte, dass Release-Häufigkeit und Stabilität keine Gegensätze sind. Die vierte Auflage von *Software Architecture in Practice* (2021) hat ihr ein Kapitel hinzugefügt.

Die zentrale Idee sind **kleine Batches**. Ein Release mit einer einzigen Änderung ist leicht zu testen, leicht zu durchschauen und leicht zurückzurollen; ein Release mit drei Monaten Arbeit enthält Hunderte zusammenwirkender Änderungen, sodass Ausfälle wahrscheinlich und schwer zu diagnostizieren sind. Kleine Batches erfordern **Automatisierung**: eine Deployment-Pipeline, die jede Änderung auf dieselbe Weise baut, testet und deployt, skriptgesteuerte Infrastruktur und keine manuellen Schritte, die jemand vergessen kann. Zu den Architekturtaktiken gehören unabhängig deploybare Einheiten (damit das Release eines Teams nicht auf das eines anderen wartet), abwärtskompatible Schnittstellen und Datenbankmigrationen, Mechanismen für Rollback und Roll-forward, schrittweise Rollouts (Canary Releases zunächst für einen kleinen Teil des Traffics, Blue-Green-Umschalten zwischen zwei Umgebungen) und Feature-Flags, die das *Deployen* von Code vom *Freigeben* einer Funktion für Benutzer trennen. Pipelines und Strategien werden in CI/CD-Pipelines und Deployment-Strategien behandelt.

Die Kosten, Deploybarkeit zu vernachlässigen, können gravierend sein. Im August 2012 deployte das Handelsunternehmen Knight Capital neuen Code manuell auf seine acht Order-Routing-Server, übersah aber einen. Ein wiederverwendetes Konfigurations-Flag aktivierte dann auf diesem Server veralteten Code, der den Markt mit Orders überflutete, in rund 45 Minuten etwa 4 Millionen Ausführungen erzeugte und das Unternehmen etwa 460 Millionen US-Dollar kostete. Ein automatisiertes, verifiziertes Deployment und das Entfernen toten Codes hätten es jeweils verhindert.

Auslieferung messen: die DORA-Metriken

Das DORA-Forschungsprogramm (DevOps Research and Assessment, beschrieben in *Accelerate* von Nicole Forsgren, Jez Humble und Gene Kim, 2018, und fortgeführt in Jahresberichten, die heute innerhalb von Google Cloud erstellt werden) fand heraus, dass eine kleine Zahl von Kennzahlen die Leistung der Softwareauslieferung vorhersagt und dass die Besten zugleich schnell und stabil sind. Das aktuelle Modell hat fünf DORA-Metriken für die Softwareauslieferung, gegliedert in Durchsatz und Instabilität. Ihre Namen haben sich entwickelt: Aus der alten „Mean Time to Restore“ wurde 2023 die Wiederherstellungszeit nach fehlgeschlagenem Deployment, und die Deployment-Nacharbeitsrate kam 2024 hinzu.

Zwei Vorbehalte. Erstens beschreiben diese Metriken das Auslieferungssystem eines Teams, nicht Einzelpersonen; wer damit Menschen in eine Rangfolge bringt, lädt zum Schönen der Zahlen ein. Zweitens sind sie Ergebnisse. Die Architektur beeinflusst sie stark: Teams, die ihren Teil unabhängig deployen, isoliert testen und schnell zurückrollen können, schneiden besser ab; das ist ein Grund, warum die DORA-Forschung lose gekoppelte Architektur mit höherer Auslieferungsleistung verbindet.

DORA-Metriken für die Softwareauslieferung (dora.dev, aktuelles Modell)

Metrik	Gruppe	Was sie misst	Architektonische Hebel
Durchlaufzeit von Änderungen	Durchsatz	Zeit vom Commit einer Änderung in die Versionsverwaltung bis zu ihrem Lauf in Produktion	Schnelle automatisierte Tests, kleine unabhängig deploybare Einheiten, keine Übergaben
Deployment-Häufigkeit	Durchsatz	Wie oft Änderungen in Produktion deployt werden (oder die Zeit zwischen Deployments)	Unabhängige Deploybarkeit, automatisierte Pipeline, Feature-Flags
Wiederherstellungszeit nach Fehl-Deployment	Durchsatz	Zeit bis zur Erholung von einem Deployment, das fehlschlägt und sofortiges Eingreifen erfordert	Automatisiertes Rollback, Canary Releases, gutes Monitoring, kleine Änderungen
Änderungsfehlerrate	Instabilität	Anteil der Deployments, die sofortiges Eingreifen erfordern, etwa ein Rollback oder einen Hotfix	Testbarkeit, Contract Tests, abwärtskompatible Änderungen
Deployment-Nacharbeitsrate	Instabilität	Anteil der Deployments, die ungeplant sind und wegen eines Incidents in Produktion erfolgen	Qualität von Tests und Reviews; Observability, die Probleme früh erkennt

Wartbarkeit in ISO/IEC 25010

ISO/IEC 25010:2023 fasst diese Anliegen unter dem Merkmal Wartbarkeit zusammen, mit fünf Teilmerkmalen: **Modularität** (das System besteht aus getrennten Komponenten, sodass eine Änderung an einer möglichst wenig Auswirkung auf andere hat), **Wiederverwendbarkeit** (ein Bestandteil kann in mehr als einem System oder beim Bau anderer Bestandteile verwendet werden), **Analysierbarkeit** (wie leicht sich die Auswirkung einer Änderung beurteilen, ein Mangel diagnostizieren oder die zu ändernden Teile finden lassen), **Änderbarkeit** (das System kann geändert werden, ohne Defekte einzuführen oder die Qualität zu mindern) und **Testbarkeit** (Testkriterien können festgelegt und Tests durchgeführt werden, um zu bestimmen, ob sie erfüllt sind). Deploybarkeit hat im Standard kein eigenes Teilmerkmal; Installierbarkeit und Austauschbarkeit gehören zur Flexibilität. Das Modell ist eine Checkliste, wie Abschnitt 2.2 erklärt; erst Szenarien machen jeden Punkt konkret.

ACHTUNG

Häufige Fehler

- Für eingebildete Änderungen bauen: Plug-in-Frameworks und Abstraktionsschichten für Varianten, nach denen niemand gefragt hat. Jeder Erweiterungspunkt kostet Komplexität und Testaufwand. Fragen Sie, wie wahrscheinlich die Änderung ist, bevor Sie die Bindung verschieben.
- Feature-Flags ohne Ausstieg: Hunderte veralteter Flags vervielfachen die Pfade durch den Code, die niemand testet. Geben Sie jedem Flag einen Verantwortlichen und ein Entfernungsdatum.
- Microservices als Heilmittel für langsame Auslieferung: Ein System in Services aufzuteilen, die trotzdem gemeinsam ausgeliefert werden müssen (ein verteilter Monolith), verschlechtert die Deploybarkeit, statt sie zu verbessern.
- Mock-lastige Tests, die Implementierungsdetails prüfen: Sie bestehen, während das System kaputt ist, und schlagen bei jedem Refactoring fehl. Bevorzugen Sie Fakes und Contract Tests an Architekturgrenzen.
- Die Durchlaufzeit in einem Team ab Beginn eines Tickets und in einem anderen ab dem Commit messen und beide dann vergleichen. Definieren Sie jede Metrik einmal und wenden Sie sie einheitlich an.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum ist „Das System soll leicht änderbar sein“ keine brauchbare Anforderung?

Sie sagt nicht, was sich ändern wird, wer es ändert, wann oder welche Kosten akzeptabel sind; daher kann niemand dafür entwerfen oder prüfen, ob sie erfüllt ist.

▼ Was ist der Unterschied zwischen einem Stub und einem Mock?

Ein Stub liefert vorgefertigte Antworten, damit der Test Eingaben steuern kann; ein Mock prüft zusätzlich, dass die erwarteten Aufrufe erfolgt sind, sodass der Test Interaktionen verifiziert.

▼ Warum verbessern Feature-Flags die Deploybarkeit?

Sie trennen das Deployen von Code vom Freigeben einer Funktion, sodass unfertige oder riskante Funktionen ausgeschaltet deployt, schrittweise freigegeben und ohne neues Deployment wieder ausgeschaltet werden können.

▼ **Ein Team deployt monatlich, und jedes Release enthält etwa 200 Änderungen. Welche DORA-Metriken leiden wahrscheinlich, und warum?**

Die Durchlaufzeit von Änderungen (Änderungen warten wochenlang auf das Release), die Deployment-Häufigkeit und wahrscheinlich auch Änderungsfehlerrate und Wiederherstellungszeit, weil große Batches eher fehlschlagen und schwerer zu diagnostizieren und zurückzurollen sind.

▼ **Probieren Sie es aus: Jeder neue Kurierpartner kostet heute 10 Personentage. Ein Refactoring von 15 Tagen würde das auf 4 Tage senken. In diesem Jahr werden fünf Partner erwartet, zu £600 pro Tag. Lohnt sich das Refactoring, und wann ist die Gewinnschwelle erreicht?**

Ohne: 5 x 10 = 50 Tage, £30.000. Mit: 15 + (5 x 4) = 35 Tage, £21.000, eine Ersparnis von 15 Tagen oder £9000. Jeder Partner spart 6 Tage, die Gewinnschwelle liegt also bei 15 / 6 = 2,5, das heißt beim dritten Partner. Es lohnt sich, wenn mindestens drei Partner wahrscheinlich sind. Außerdem wird jede Integration weniger riskant, prüfen Sie aber zuerst die Prognose, denn die Ersparnis hängt von ihr ab.

QUELLEN

- DORA: software delivery performance metrics: https://dora.dev/guides/dora-metrics/
- DORA: history of the software delivery metrics: https://dora.dev/guides/dora-metrics/history/
- Martin Fowler: Test Double: https://martinfowler.com/bliki/TestDouble.html
- Pete Hodgson: Feature Toggles (aka Feature Flags), martinfowler.com: https://martinfowler.com/articles/feature-toggles.html
- Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern: https://martinfowler.com/articles/injection.html

Sicherheit, Benutzbarkeit, Interoperabilität, Kosten und Nachhaltigkeit als Qualitäten

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.2 Qualitätsmodelle
- 2.3 Qualitätsszenarien
- 2.5 Verfügbarkeit und Zuverlässigkeit

Begriffe von früher, die er verwendet: ISO/IEC 25010, Qualitätsszenario, Taktik, Randbedingung, API (Programmierschnittstelle)

DIE IDEE IN KÜRZE

Wenn Sie eine Bank wählen, ist Ihnen wichtig, dass sie Ihr Geld sicher verwahrt, dass ihre App leicht zu bedienen ist, dass sie mit anderen Banken zusammenarbeitet, dass ihre Gebühren fair sind und vielleicht auch, dass sie keine Energie verschwendet. Nichts davon ist eine Funktion, auf die Sie auf einem Bildschirm zeigen können; jedes ist eine Qualität des gesamten Dienstes. Architekten müssen jede davon so konkret fassen, dass man darauf hin entwerfen und sie prüfen kann, und oft müssen sie die eine gegen die andere abwägen.

WAS SIE WISSEN MÜSSEN

- Sicherheit beginnt mit Vertraulichkeit, Integrität und Verfügbarkeit; ISO/IEC 25010:2023 nennt Vertraulichkeit, Integrität, Nichtabstreitbarkeit, Verantwortlichkeit, Authentizität und Widerstandsfähigkeit, wobei Verfügbarkeit unter Zuverlässigkeit fällt.
- Sicherheitsszenarien benennen den Angreifer, den Angriff, das Ziel und eine messbare Antwort wie die Erkennungszeit oder die offengelegten Daten; Taktiken widerstehen Angriffen, erkennen sie, reagieren darauf und erholen sich davon.
- Interaktionsfähigkeit (früher Benutzbarkeit) beeinflusst die Architektur über Reaktionsfähigkeit, Rückgängigmachen, Abbrechen und Offline-Arbeit; Barrierefreiheit nach WCAG 2.2 Stufe AA wird zunehmend zur rechtlichen Randbedingung.
- Interoperabilität erfordert Einigkeit über die Bedeutung, nicht nur über das Format; Branchenstandards und Adapter senken die Kosten.
- Kosten und CO2 drückt man am besten pro Arbeitseinheit aus; die SCI-Spezifikation (ISO/IEC 21031:2024) berechnet ((E x I) + M) pro R und zählt nur echte Reduktionen.

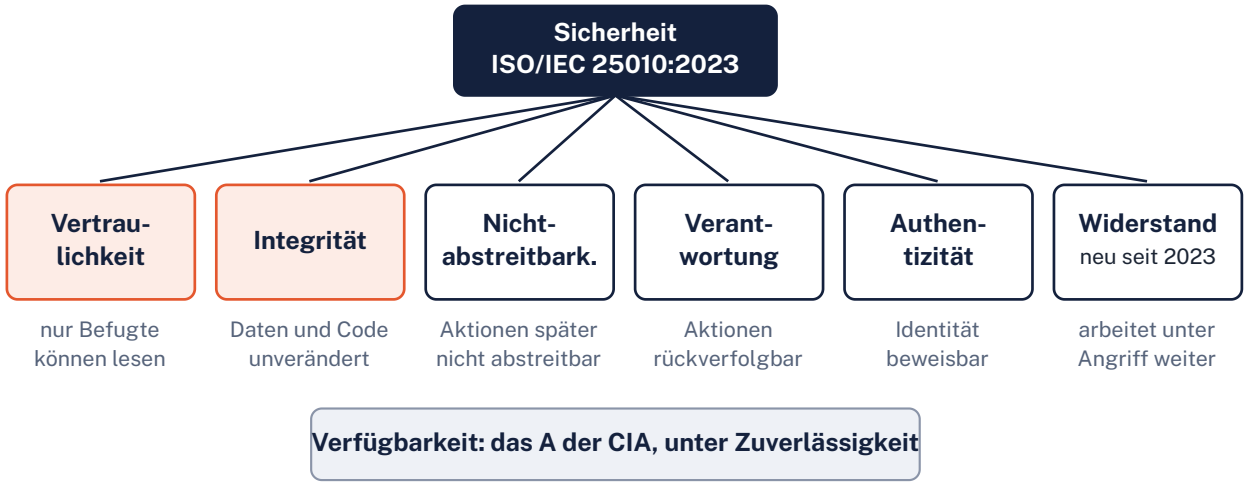
- Auch Betriebssicherheit, Observability, Datenresidenz, Auditierbarkeit und Aufbewahrung sind Qualitäten und sollten früh als testbare Randbedingungen erfasst werden.

Sicherheit als Qualitätsmerkmal

Sicherheit ist die Fähigkeit des Systems, Daten und Funktionen vor unbefugtem Zugriff und unbefugter Nutzung zu schützen und dabei weiterhin die Befugten zu bedienen. Architekten gehen oft von der CIA-Triade aus: **Vertraulichkeit** (nur Befugte können Daten lesen), **Integrität** (Daten und Code werden nicht ohne Befugnis verändert, und Veränderungen sind erkennbar) und **Verfügbarkeit** (befugte Nutzer können den Dienst nutzen, wenn sie ihn brauchen). Angriffe lassen sich den drei zuordnen: Wer Kundendaten stiehlt, verletzt die Vertraulichkeit; wer die Bankverbindung eines Zahlungsempfängers ändert, verletzt die Integrität; und ein Denial-of-Service-Angriff verletzt die Verfügbarkeit.

ISO/IEC 25010:2023 verfeinert das Bild. Sein Merkmal Sicherheit hat sechs Teilmerkmale: Vertraulichkeit, Integrität, **Nichtabstreitbarkeit** (Aktionen oder Ereignisse lassen sich als geschehen nachweisen, sodass eine Partei sie später nicht leugnen kann), **Verantwortlichkeit** (Aktionen lassen sich eindeutig der Entität zuordnen, die sie ausgeführt hat), **Authentizität** (die Identität eines Subjekts oder einer Ressource lässt sich als die behauptete nachweisen) und, neu in der Ausgabe 2023, **Widerstandsfähigkeit** (das Produkt arbeitet weiter wie vorgesehen, während es von einem böswilligen Akteur angegriffen wird). Verfügbarkeit steht in diesem Modell unter Zuverlässigkeit, weshalb sie in Abschnitt 2.5 behandelt wurde.

Die Teilmerkmale der Sicherheit



Vertraulichkeit und Integrität stammen aus der CIA-Triade; die anderen vier machen Verantwortlichkeit, Identitätsnachweis und Verhalten unter Angriff ausdrücklich.

Sicherheitsszenarien schreiben

„Das System muss sicher sein“ lässt sich weder entwerfen noch testen. Ein Sicherheitsszenario benennt eine Quelle (wer angreift, mit welchem Zugang), einen Stimulus (was versucht wird), ein Artefakt (was angegriffen wird), eine Umgebung (online, während eines Vorfalls, mit dem Angreifer bereits im Netzwerk), eine Antwort und ein Maß. Maße für Sicherheit betreffen oft die Erkennungszeit, das Ausmaß des Schadens oder die Zeit bis zur Wiederherstellung, weil sich „kein Angriff ist je erfolgreich“ nicht überprüfen lässt.

Drei Beispiele für eine Onlinebank. *Credential Stuffing:* Ein externer Angreifer probiert geleakte Kombinationen aus Benutzername und Passwort mit 50 Versuchen pro Sekunde von vielen Adressen aus an der Login-API; das System erkennt das Muster innerhalb von 2 Minuten, fordert die Quelladressen heraus oder sperrt sie, und Konten mit Multi-Faktor-Authentifizierung werden nicht übernommen. *Manipulation durch Insider:* Ein Support-Mitarbeiter mit legitimem Zugang ändert die Empfängerdaten eines Kunden; die Änderung erfordert die Freigabe durch eine zweite Person, und das Audit-Log hält wer, was und wann in einer Form fest, die der Mitarbeiter nicht verändern kann. *Gestohlenes Backup:* Ein Angreifer gelangt an eine Kopie des Datenbank-Backups der letzten Woche; alle personenbezogenen Daten darin sind mit getrennt gespeicherten Schlüsseln verschlüsselt, sodass keine Kundendaten lesbar sind.

Beachten Sie, dass jedes Szenario eine andere Architektur nach sich zieht: Rate Limiting und Anomalieerkennung am Rand des Systems, einen Freigabe-Workflow nach dem Vier-Augen-Prinzip und manipulationssichere Logs sowie Verschlüsselung mit getrenntem Schlüsselmanagement. Solche Szenarien systematisch zu finden ist Aufgabe der Bedrohungsmodellierung, die in Kapitel 8 behandelt wird.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: die Sicherheitsbegriffe auseinanderhalten

Die Idee, die hier die meiste Sorgfalt verlangt, ist es, die ähnlich klingenden Teilmerkmale der Sicherheit auseinanderzuhalten, besonders Authentizität, Verantwortlichkeit und Nichtabstreitbarkeit. Nehmen Sie das Insider-Szenario von oben, in dem ein Support-Mitarbeiter die Empfängerdaten eines Kunden ändert, und ordnen Sie jede Schutzmaßnahme darin einem Teilmerkmal zu.

- Die Empfängerdaten dürfen sich nicht ohne Befugnis ändern, und jede Änderung muss erkennbar sein; im Szenario muss eine zweite Person die Änderung freigeben. Das schützt die **Integrität**.
- Das Audit-Log hält fest, wer die Änderung vorgenommen hat, worin sie bestand und wann, sodass sie sich genau diesem einen Mitarbeiter zuordnen lässt. Das ist **Verantwortlichkeit**: zu wissen, wer es war.
- Das Log kann nur dann die richtige Person nennen, wenn die Bank sicher sein kann, dass, wer sich als der Mitarbeiter angemeldet hat, wirklich dieser Mitarbeiter ist und nicht jemand mit einem gestohlenen Passwort. Nachweisen zu können, dass eine Person oder ein System die ist, die sie zu sein behauptet, ist **Authentizität**.
- Das Log wird in einer Form aufbewahrt, die der Mitarbeiter nicht verändern kann; sagt er später ‚das war ich nicht', kann die Bank zeigen, dass er es war. Das ist **Nichtabstreitbarkeit**: der Nachweis, dass die Aktion stattgefunden hat, sodass sie sich nicht leugnen lässt.
- Die anderen beiden gehören zu anderen Szenarien. **Vertraulichkeit** (nur Befugte können Daten lesen) schützt das Szenario mit dem gestohlenen Backup durch Verschlüsselung. **Widerstandsfähigkeit**, also unter Angriff wie vorgesehen weiterzuarbeiten, prüft das Credential-Stuffing-Szenario.
- **Die Lehre:** Vertraulichkeit und Integrität schützen die Daten; Authentizität, Verantwortlichkeit und Nichtabstreitbarkeit betreffen die Menschen, die mit den Daten handeln: nachweisen, wer sie sind, nachverfolgen, was sie getan haben, und es später belegen. Jedes verlangt einen anderen Mechanismus, also benennen Sie in jedem Szenario das gemeinte.

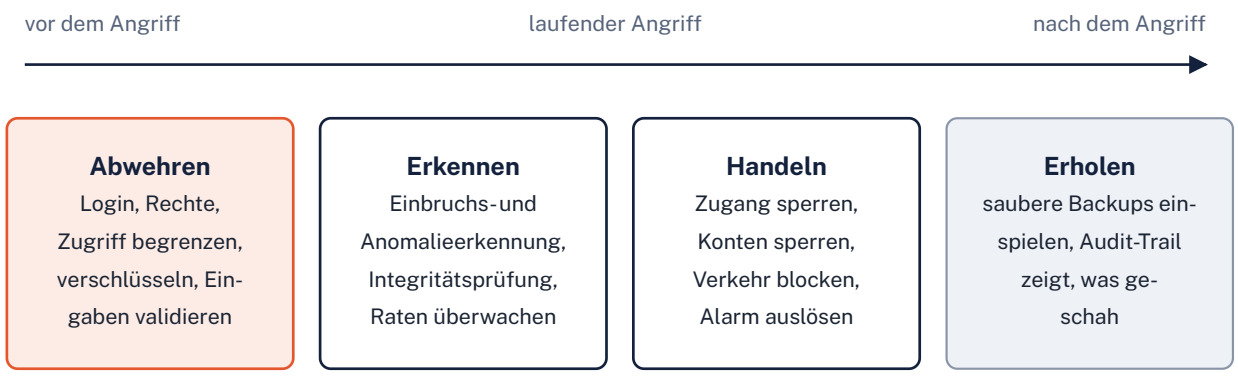
Sicherheitstaktiken: ein Überblick

Bass, Clements und Kazman gruppieren Sicherheitstaktiken danach, wann sie wirken. **Angriffen widerstehen**: Akteure identifizieren, authentifizieren und autorisieren, Zugriff und Angriffsfläche begrenzen (weniger offene Ports, weniger öffentliche Endpoints, Least Privilege), Daten bei der Übertragung und im Ruhezustand verschlüsseln, Entitäten trennen (sensible Daten in einem eigenen Speicher oder Netzwerksegment), Standardeinstellungen

ändern und Eingaben validieren. **Angriffe erkennen**: Angriffserkennung (Intrusion Detection), Denial of Service erkennen, die Integrität von Nachrichten prüfen und ungewöhnliche Verzögerungen oder Muster bemerken. **Auf Angriffe reagieren**: Zugriff entziehen, Konten nach fehlgeschlagenen Anmeldungen sperren, Betrieb und Nutzer informieren. **Sich von Angriffen erholen**: einen bekannten guten Zustand wiederherstellen (wieder die Taktiken der Verfügbarkeit) und einen Audit-Trail führen, um zu verstehen, was geschehen ist, und die Nichtabstreitbarkeit zu stützen.

Dieser Abschnitt stellt nur die Familien vor. Prinzipien wie Defense in Depth und Least Privilege, Identitätsprotokolle, Verschlüsselung, Zero Trust und die Sicherheit der Lieferkette sind Thema von Kapitel 8. Der architektonische Punkt hier ist, dass Sicherheit eine Eigenschaft der gesamten Struktur ist und keine Komponente, die man am Ende hinzufügt: Wo Vertrauensgrenzen verlaufen, welche Komponente welche Daten hält und wie Services einander authentifizieren, sind frühe und teuer zu ändernde Entscheidungen.

Taktiken vor, während und nach einem Angriff



Gruppierung nach Bass, Clements und Kazman; Details in Kapitel 8

Abwehrtaktiken verringern die Erfolgsaussichten eines Angriffs; Taktiken zum Erkennen, Reagieren und Erholen begrenzen den Schaden, wenn die Abwehr versagt.

Benutzbarkeit, Interaktionsfähigkeit und Barrierefreiheit

ISO/IEC 25010:2023 hat Benutzbarkeit in *Interaktionsfähigkeit* umbenannt: das Maß, in dem festgelegte Nutzer über die Benutzeroberfläche mit einem Produkt interagieren können, um Informationen auszutauschen und bestimmte Aufgaben zu erledigen. Ihre Teilmerkmale sind Erkennbarkeit der Eignung, Erlernbarkeit, Bedienbarkeit, Schutz vor Benutzerfehlern, Nutzerbindung (die die Ästhetik der Benutzeroberfläche ersetzt hat), Inklusivität, Benutzerunterstützung und Selbstbeschreibungsfähigkeit. Vieles davon ist Interaktionsdesign statt Architektur, doch mehrere Anliegen der Benutzbarkeit reichen tief in die Struktur hinein.

Reaktionsfähigkeit ist das offensichtliche. Bewährte Richtwerte der Ergonomie, bekannt gemacht durch Jakob Nielsen, nennen drei Grenzen: Etwa 0,1 Sekunden wirken unmittelbar, etwa 1 Sekunde erhält den Gedankenfluss des Nutzers, und etwa 10 Sekunden sind die Grenze der Aufmerksamkeit. Um sie einzuhalten, braucht es unter Umständen Caching, asynchrone Verarbeitung mit Fortschrittsanzeige oder optimistische Aktualisierungen. Rückgängigmachen und Abbrechen sind architektonisch: „die letzten fünf Aktionen rückgängig machen“ erfordert umkehrbare Befehle, und das Abbrechen eines langen Uploads erfordert eine Operation, die sich sauber unterbrechen lässt. Offline-Arbeit in einer mobilen App erfordert lokale Speicherung und Synchronisation mit Konfliktbehandlung. All das ist viel billiger von Anfang an einzuplanen als nachzurüsten.

Barrierefreiheit, also die Möglichkeit für Menschen mit Behinderungen, das Produkt zu nutzen, ist zunehmend eine rechtliche Randbedingung und keine bloße Vorliebe mehr. Die Web Content Accessibility Guidelines in Version 2.2 (WCAG 2.2), seit Oktober 2023 eine W3C-Empfehlung, definieren testbare Erfolgskriterien auf den Stufen A, AA und AAA; Stufe AA ist der übliche Maßstab im Recht und in der Beschaffung. WCAG 2.2 hat neun Kriterien ergänzt, darunter Mindestgrößen für Zielflächen bei Zeigereingabe und barrierefreie Authentifizierung (kein Schritt der Anmeldung darf von einem kognitiven Funktionstest abhängen, etwa dem Merken eines Passworts oder dem Abschreiben eines Codes, es sei denn, dem Nutzer wird eine alternative Methode oder Hilfe angeboten, etwa das Einfügen aus einem Passwortmanager; einige weitere enge Ausnahmen gelten). In der Europäischen Union gilt der European Accessibility Act (Richtlinie (EU) 2019/882) seit dem 28. Juni 2025 für viele Verbraucherprodukte und -dienstleistungen, darunter E-Commerce, Bankdienstleistungen, E-Books und Personenbeförderungsdienste. Für Architekten ist Barrierefreiheit daher eine früh zu erfassende Randbedingung: Sie beeinflusst die Wahl des Frontend-Frameworks, das Designsystem, den Authentifizierungsablauf und die Test-Pipeline (automatisierte Barrierefreiheitsprüfungen in der Continuous Integration sowie manuelle Tests mit assistiven Technologien).

Interoperabilität

Interoperabilität ist das Maß, in dem zwei oder mehr Systeme Informationen austauschen und die ausgetauschten Informationen nutzen können. ISO/IEC 25010 ordnet sie der Kompatibilität zu, neben der Koexistenz. Sie hat zwei Ebenen. *Syntaktische* Interoperabilität bedeutet, dass sich die Systeme über Format und Protokoll einig sind: Beide sprechen HTTP und JSON. *Semantische* Interoperabilität bedeutet, dass sie sich über die Bedeutung einig sind: `"amount": 1050` sind Pence, nicht Pfund; `status: "closed"` bedeutet bezahlt, nicht storniert. Die meisten Integrationsfehler sind semantisch.

Branchen senken die Kosten semantischer Einigung mit gemeinsamen Standards: HL7 FHIR für den Austausch von Gesundheitsdaten, ISO 20022 für Finanznachrichten, GTFS für Fahrpläne im öffentlichen Verkehr. Zu den architektonischen Taktiken gehören, Services über eine Registry zu finden, Schnittstellen mit Adaptern und Übersetzern anzupassen und Versionen ausdrücklich zu verwalten, damit ältere Partner weiter funktionieren. Ein Interoperabilitätsszenario könnte lauten: „Ein neues Krankenhaus tritt dem Netzwerk bei und tauscht innerhalb von 4 Wochen Patientenzusammenfassungen über FHIR-R4-Ressourcen aus, ohne Änderungen am zentralen Patientenaktenservice.“

Kosten als Qualität

Kosten tauchen in Qualitätsmodellen selten auf, doch sie begrenzen jede Architekturentscheidung. Unterscheiden Sie **Erstellungskosten** (der Aufwand, das System zu bauen), **Betriebskosten** (Infrastruktur, Lizenzen, Dienste Dritter, Support) und **Änderungskosten** (der Aufwand für jede künftige Änderung, um den es bei der Änderbarkeit geht). In der Cloud sind die Betriebskosten variabel und unmittelbar von der Architektur geprägt: Gesprächige Services zahlen für Netzwerkübertragung, überdimensionierte Cluster zahlen für ungenutzte Kapazität, und ein unbedachtes Abfragemuster kann eine Datenbankrechnung vervielfachen.

Ein nützlicher Weg, Kosten zum Qualitätsmerkmal zu machen, sind Stückkosten: Kosten pro Transaktion, pro aktivem Nutzer oder pro Bestellung. Anders als eine monatliche Gesamtrechnung zeigen sie, ob die Architektur mit wachsendem Geschäft effizienter oder weniger effizient wird, und sie lassen sich als Szenario formulieren: „Bei 3 Millionen Bestellungen im Monat bleiben die Infrastrukturkosten unter 1 Penny pro Bestellung.“ FinOps-Praktiken, um dies zu steuern, werden in Kapitel 10 behandelt.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Stückkosten und CO2 pro Anfrage für eine API einer Essenslieferplattform

Eine Bestell-API läuft auf einem Cluster, der £24.000 im Monat kostet, und verarbeitet 3.000.000 Bestellungen im Monat. Eine Analyse schätzt ihren Energieverbrauch auf 1200 kWh im Monat, in einem Stromnetz mit durchschnittlich 250 g CO_2e pro kWh, und ihr werden monatlich 60 kg CO_2e an grauen Emissionen der Hardware zugerechnet. Die Formel der Software Carbon Intensity lautet SCI = ((E x I) + M) pro R.

Kosten pro Bestellung	£24.000 / 3.000.000 = £0,008, also 0,8 Pence pro Bestellung
Betriebsemissionen (E x I)	1200 kWh x 250 g = 300.000 g = 300 kg CO_2e
Graue Emissionen addieren (M)	300 + 60 = 360 kg CO_2e
Durch die funktionale Einheit teilen (R = eine Bestellung)	360.000 g / 3.000.000 = 0,12 g CO_2e pro Bestellung
Änderung: Speisekarten cachen, Energie um 25 % senken	E = 900 kWh: (900 x 250 + 60.000) / 3.000.000 = 0,095 g pro Bestellung
Änderung: Batch-Jobs in einer Region mit 50 g pro kWh ausführen	Senkt I für diesen Anteil der Last; CO2-bewusste Planung (Carbon-Aware Scheduling) senkt den Wert, ohne den Code zu ändern
Wachstum auf 6.000.000 Bestellungen auf demselben Cluster	Die Kosten pro Bestellung halbieren sich auf 0,4 Pence; auch das CO2 pro Bestellung sinkt, solange der Energieverbrauch unterproportional wächst (Server im Leerlauf ziehen weiterhin Strom, daher ist höhere Auslastung effizienter)

Beide Kennzahlen sind Raten pro Einheit nützlicher Arbeit; sie belohnen daher Effizienz, statt Wachstum zu bestrafen.

Die SCI-Spezifikation zählt nur echte Reduktionen: Der Kauf von Kompensationszertifikaten oder Herkunftsnachweisen für erneuerbare Energie senkt den Wert nicht.

Energie- und Netzwerte sind Schätzungen; entscheidend ist, sie über die Zeit auf dieselbe Weise zu messen, damit sich Änderungen vergleichen lassen.

Energieeffizienz und Nachhaltigkeit

Software stößt selbst kein CO2 aus, doch die Hardware, auf der sie läuft, verbraucht Strom und trägt die Emissionen ihrer Herstellung in sich. Die Spezifikation Software Carbon Intensity (SCI) der Green Software Foundation, von der ISO als ISO/IEC 21031:2024 veröffentlicht, definiert eine Rate: Betriebsemissionen (Energie E mal CO2-Intensität I des Stroms) plus graue Emissionen M, geteilt durch eine funktionale Einheit R wie einen Nutzer, einen API-Aufruf oder eine Bestellung. Sie nennt drei Hebel: **Energieeffizienz** (für dieselbe Arbeit weniger Strom verbrauchen), **Hardwareeffizienz** (weniger Geräte nutzen und diese stärker und länger auslasten) und **CO2-Bewusstsein** (flexible Arbeit dann und dort ausführen, wo der Strom sauberer ist).

Die Architektur beeinflusst alle drei. Passende Dimensionierung (Right-Sizing) und höhere Auslastung verbessern die Hardwareeffizienz; Caching, effiziente Datenformate und das Vermeiden unnötiger Arbeit verbessern die Energieeffizienz; Queues und Batch-Verarbeitung ermöglichen es, Arbeit zeitlich oder räumlich zu verlagern. Nachhaltigkeit geht oft mit den Kosten einher, aber nicht immer: Ein Warm-Standby in einer zweiten Region verbessert die Verfügbarkeit und verdoppelt die grauen Emissionen. Große KI-Modelle machen die Frage dringlicher, wie Kapitel 12 erörtert.

Betriebssicherheit (Safety)

Betriebssicherheit ist die Freiheit von unvertretbaren Risiken für Menschen, Sachwerte oder die Umwelt. Sie unterscheidet sich von Sicherheit: Sicherheit schützt das System vor Menschen mit schädlicher Absicht; Betriebssicherheit schützt Menschen vor dem System, auch wenn niemand böswillig handelt. ISO/IEC 25010:2023 hat Betriebssicherheit als neuntes Merkmal ergänzt, mit den Teilmerkmalen betriebliche Beschränkung, Risikoidentifikation, Ausfallsicherheit (Fail-safe), Gefahrenwarnung und sichere Integration.

Sicherheitskritische Bereiche haben eigene Normen, die Architektur und Prozess vorgeben: IEC 61508 für funktionale Sicherheit allgemein, ISO 26262 für Straßenfahrzeuge, IEC 62304 für Software von Medizinprodukten und DO-178C für Software in der Luftfahrt. Typische Taktiken sind ein ausfallsicherer Standardzustand (ein Zug hält an, wenn die Signale ausfallen), unabhängige Überwachungskanäle, die die Hauptsteuerung übersteuern können, Redundanz mit Mehrheitsentscheid und eine strikte Trennung sicherheitskritischer Komponenten von allem anderen, damit ein Fehler im Infotainmentsystem nicht die Bremsen erreichen kann. Beachten Sie, dass ausfallsicher und hochverfügbar im Konflikt stehen können: Ein ausfallsicheres System hält oft an, was sicher, aber nicht verfügbar ist.

Observability, eine aufkommende Qualität

Observability (Beobachtbarkeit) ist das Maß, in dem Sie den inneren Zustand eines Systems von außen anhand der Daten verstehen können, die es ausgibt, und zwar gut genug, um Fragen zu beantworten, die beim Bau nicht vorhergesehen wurden. Sie ist in ISO/IEC 25010 nicht aufgeführt, auch wenn Analysierbarkeit und Verantwortlichkeit sie berühren. In verteilten Systemen ist sie zu einer eigenständigen Qualität geworden, weil die Wiederherstellungszeit und damit die Verfügbarkeit davon abhängt, wie schnell Menschen herausfinden, was nicht stimmt. Architektonisch erfordert sie strukturierte Logs, Metriken und verteilte Traces mit Korrelations-IDs, die einer Anfrage über Services hinweg folgen, und zwar von Anfang an eingeplant. Ein Szenario könnte lauten: „Bei jedem fehlgeschlagenen Bezahlvorgang kann ein Bereitschaftsingenieur innerhalb von 5 Minuten den fehlerhaften Service und den Weg der Anfrage ermitteln.“ Siehe Observability in Kapitel 9.

Rechtliche und regulatorische Qualitäten

Gesetze und Vorschriften erzeugen Qualitäten, die man bis zu einem Audit leicht übersieht. Datenresidenz verlangt, dass bestimmte Daten in einem bestimmten Land oder einer bestimmten Region gespeichert oder verarbeitet werden; sie kann aus Branchenregeln, Verträgen mit der öffentlichen Hand oder Kundenanforderungen stammen. Die DSGVO selbst verlangt nicht, dass Daten in der EU bleiben, aber Übermittlungen außerhalb der EU brauchen eine Rechtsgrundlage und Garantien, was in der Praxis bestimmt, wo Services und Backups laufen. **Auditierbarkeit** bedeutet, später zeigen zu können, wer was wann getan hat: manipulationssichere Logs, für einen festgelegten Zeitraum aufbewahrt, mit synchronisierten Uhren. **Aufbewahrungs- und Löschregeln** verlangen, Daten mindestens eine bestimmte Zeit aufzubewahren (Finanzunterlagen) und nach einer Höchstdauer oder auf Antrag zu löschen (personenbezogene Daten), was in Systemen mit vielen Kopien, Caches und Backups schwierig ist. **Erklärbarkeit** und menschliche Aufsicht sind für manche automatisierten Entscheidungen vorgeschrieben, wie das Kapitel zur KI-Verordnung der EU erläutert.

Behandeln Sie diese als Randbedingungen mit Akzeptanzkriterien: „Kundendaten deutscher Auftraggeber der öffentlichen Hand werden nur in EU-Regionen gespeichert und verarbeitet, einschließlich Backups und Logs“ ist testbar und schließt manche Managed Services sofort aus. Datenschutzanforderungen werden in Privacy by Design und die DSGVO behandelt.

Zusammenfassung: die Qualitäten dieses Abschnitts und Beispielmaße

Qualität	Beispiel für ein Antwortmaß	Typischer architektonischer Hebel
Sicherheit (Vertraulichkeit)	Keine personenbezogenen Daten aus einem gestohlenen Backup lesbar	Verschlüsselung im Ruhezustand mit getrennt verwalteten Schlüsseln
Sicherheit (Verantwortlichkeit)	Jede Änderung an Empfängerdaten einer Person zuordenbar, Logs manipulationssicher	Zentrales Audit-Log, nur einmal beschreibbarer Speicher (Write-once)
Interaktionsfähigkeit	95 % der Erstnutzer schließen den Bezahlvorgang ohne Hilfe in unter 3 Minuten ab	Einfache Abläufe, schnelle Antworten, Rückgängigmachen
Barrierefreiheit	Alle Customer Journeys erfüllen WCAG 2.2 Stufe AA	Barrierefreies Designsystem, automatisierte Prüfungen in der CI
Interoperabilität	Neuer Partner nach dem veröffentlichten API-Standard innerhalb von 4 Wochen angebunden	Standardformate, Adapter, versionierte APIs
Kosten	Infrastruktur unter 1 Penny pro Bestellung bei 3 Millionen Bestellungen im Monat	Right-Sizing, Caching, Autoscaling
Nachhaltigkeit	SCI unter 0,1 g CO_2e pro Bestellung	Effizienz, Auslastung, CO2-bewusste Planung
Observability	Fehlerhafter Service innerhalb von 5 Minuten für jede fehlgeschlagene Anfrage ermittelt	Strukturierte Logs, Metriken, Distributed Tracing
Datenresidenz	Alle Daten von EU-Kunden, einschließlich Backups, bleiben in EU-Regionen	An Regionen gebundene Speicherung und Verarbeitung

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Was ist der Unterschied zwischen Verantwortlichkeit und Nichtabstreitbarkeit?**

Verantwortlichkeit bedeutet, dass sich Aktionen der Entität zuordnen lassen, die sie ausgeführt hat; Nichtabstreitbarkeit bedeutet, dass es einen so starken Nachweis gibt, dass die Entität die Aktion später nicht leugnen kann.

▼ **Warum zählt „die letzten fünf Aktionen rückgängig machen“ als architektonische Anforderung und nicht nur als Anforderung an die Benutzeroberfläche?**

Weil Operationen dafür als umkehrbare Befehle mit gespeicherter Historie modelliert werden müssen, was beeinflusst, wie Zustandsänderungen im gesamten System strukturiert sind.

▼ **Eine API verbraucht 800 kWh im Monat in einem Stromnetz mit 200 g CO2e pro kWh, ihr sind 40 kg graue Emissionen zugerechnet, und sie bedient 2.000.000 Anfragen. Wie hoch ist ihre SCI pro Anfrage?**

(800 x 200 g + 40.000 g) / 2.000.000 = 200.000 / 2.000.000 = 0,1 g CO2e pro Anfrage.

▼ **Wie unterscheidet sich Betriebssicherheit von Sicherheit?**

Sicherheit schützt das System und seine Daten vor Menschen mit schädlicher Absicht; Betriebssicherheit schützt Menschen und Umwelt vor Schaden, den das System verursachen könnte, auch ohne böswilligen Akteur.

▼ **Probieren Sie es aus: Eine Ticketing-API kostet £18.000 im Monat für 1.500.000 Tickets und verbraucht 800 kWh bei 300 g CO2e pro kWh, mit 40 kg grauen Emissionen. Berechnen Sie Kosten und SCI pro Ticket, dann den SCI, wenn Caching den Energieverbrauch um 25 % senkt.**

Kosten: £18.000 / 1.500.000 = 1,2 Pence pro Ticket. Betriebsemissionen: 800 x 300 g = 240 kg; mit den zusätzlichen 40 kg sind es 280 kg, und 280.000 g / 1.500.000 = etwa 0,187 g pro Ticket. Mit Caching: 600 kWh ergeben 180 kg, plus 40 kg sind 220 kg oder etwa 0,147 g pro Ticket. Werte pro Einheit bleiben bei wachsendem Volumen aussagekräftig, verfolgen Sie also diese statt der Gesamtrechnung.

QUELLEN

- W3C: Web Content Accessibility Guidelines (WCAG) 2.2: https://www.w3.org/TR/WCAG22/
- W3C WAI: What's new in WCAG 2.2: https://www.w3.org/WAI/standards-guidelines/wcag/new-in-22/
- Green Software Foundation: Software Carbon Intensity (SCI) specification: https://sci.greensoftware.foundation/
- Nielsen Norman Group: Response Times, the 3 Important Limits: https://www.nngroup.com/articles/response-times-3-important-limits/
- OWASP Application Security Verification Standard: https://owasp.org/www-project-application-security-verification-standard/

ABSCHNITT 2.8 VON 2.8 · MITTELSTUFE · 22 MIN. LESEZEIT · ZULETZT GEPRÜFT IM NOVEMBER 2026

Qualitätsmerkmale erheben, priorisieren und abwägen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.5 Stakeholder und Anliegen
- 1.6 Entscheidungen und Abwägungen
- 2.3 Qualitätsszenarien

Begriffe von früher, die er verwendet: Stakeholder, Anliegen, Qualitätsszenario, Perzentil (p95, p99), Architecture Decision Record (ADR)

DIE IDEE IN KÜRZE

Bei der Planung eines Familienurlaubs will jeder etwas anderes: Einer will es günstig, eine andere will einen Pool, ein Dritter will kurze Reisezeiten. Sie können nicht alles haben, also setzen Sie sich zusammen, notieren, was jede Person wirklich braucht, stimmen darüber ab, was am wichtigsten ist, und einigen sich, worauf Sie verzichten. Architekten machen dasselbe mit den Qualitäten eines Systems: Sie holen alle Beteiligten zusammen, machen aus Wünschen konkrete, messbare Ziele, bringen sie in eine Rangfolge und halten die gewählten Abwägungen schriftlich fest.

WAS SIE WISSEN MÜSSEN

- Qualitätsanforderungen müssen bei Stakeholdern, aus Geschäftszielen, Vorschriften, früheren Vorfällen, Wettbewerbern und Messungen am bestehenden System erhoben werden, weil sie selten ungefragt genannt werden.
- Der Quality Attribute Workshop des SEI hat acht Schritte; seine Ergebnisse sind Architekturtreiber und priorisierte, verfeinerte Szenarien, gewichtet durch eine Abstimmung mit festem Stimmenbudget.
- Ein Utility Tree verfeinert den Nutzen zu Qualitätsmerkmalen, Verfeinerungen und konkreten Szenarien, die jeweils bewertet werden (geschäftliche Bedeutung, technische Schwierigkeit); (H,H)-Szenarien erhalten als Erste Aufmerksamkeit im Design.
- Qualitäten stehen im Konflikt (Performance und Sicherheit, Verfügbarkeit und Konsistenz, Änderbarkeit und Performance, Kosten und fast alles, Benutzbarkeit und Sicherheit), und die richtige Lösung hängt vom Kontext und vom Geschäftswert ab.

- Halten Sie wesentliche Abwägungen in Entscheidungsdokumenten fest, machen Sie jedes Ziel mit einer Skala, einem Messverfahren und Stufen messbar und verfolgen Sie es über die Zeit.
- Fitness Functions machen aus den Antwortmaßen der Szenarien automatisierte Prüfungen, die laufen, während sich das System weiterentwickelt.

Woher Qualitätsanforderungen kommen

Funktionale Anforderungen landen meist in einem Backlog. Qualitätsanforderungen selten: Stakeholder setzen sie voraus („natürlich muss es schnell und sicher sein“) oder wissen nicht, dass sie sie nennen müssen. Architekten müssen sie deshalb aktiv erheben, und zwar aus mehreren Quellen.

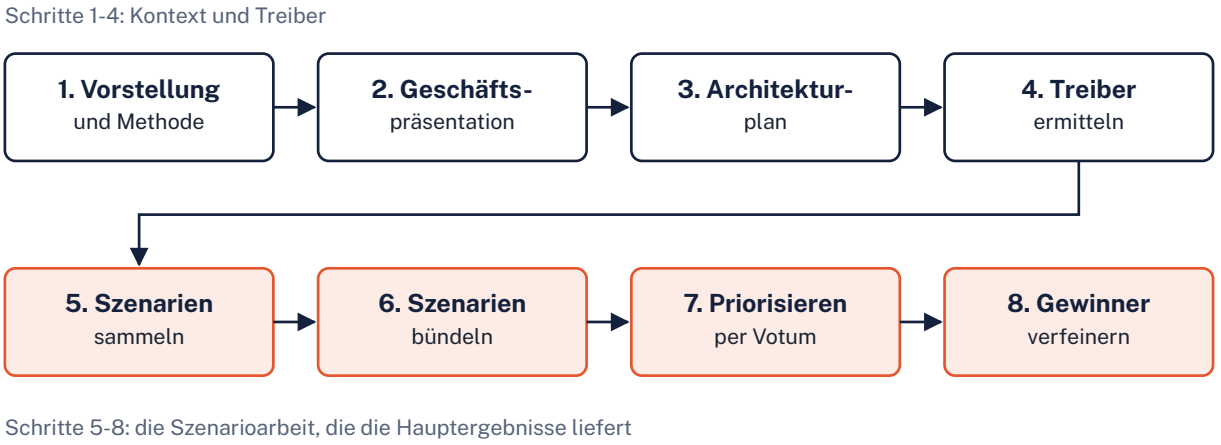
Stakeholder. Jede Gruppe hat andere Anliegen: Kunden geht es um Tempo und einfache Bedienung, dem Betrieb um Monitoring und Wiederherstellung, der Finanzabteilung um Betriebskosten, dem Support um die Diagnose von Problemen, der Compliance um Audit-Trails. Die Stakeholder-Landkarte aus Kapitel 1 ist der Ausgangspunkt. **Geschäftsziele.** „Nächstes Jahr in drei neue Länder expandieren“ setzt Lokalisierung, Datenresidenz und Skalierbarkeit voraus; „die Kosten des Kundensupports um 20 % senken“ setzt Benutzbarkeit und Self-Service voraus. **Vorschriften und Verträge.** Datenschutz, Barrierefreiheitsrecht, Branchenregeln und Kunden-SLAs schaffen nicht verhandelbare Randbedingungen. **Vorfälle.** Jeder frühere Ausfall, jede Datenpanne und jeder langsame Tag zeigt eine Qualität, die zu ungenau spezifiziert war; Post-Incident-Reviews sind eine ergiebige Quelle. **Wettbewerber und Markt.** Wenn die Konkurrenz in einer Sekunde lädt, sind zwei Sekunden ein Qualitätsproblem. **Das bestehende System.** Sein heute gemessenes Verhalten (tatsächliche Last, tatsächliche Latenzperzentile, tatsächliche Durchlaufzeit von Änderungen) liefert eine Basislinie, die aus „schneller“ eine Zahl macht.

Interviews und Fragebögen funktionieren, doch Qualitätsmerkmale zeigen sich am besten, wenn Stakeholder einander zuhören. Ein Product Owner, der von der Betriebsleitung erfährt, was eine Stunde Ausfall zur Abendessenszeit am Freitag kostet, priorisiert anders. Das ist die Idee hinter strukturierten Workshops.

Der Quality Attribute Workshop

Der Quality Attribute Workshop (QAW), entwickelt am Software Engineering Institute (Barbacci und Kollegen; die dritte Ausgabe des Methodenberichts erschien 2003), ist eine moderierte Sitzung, typischerweise einen Tag lang, die früh stattfindet, bevor die Architektur feststeht. Er bringt 5 bis 30 Stakeholder zusammen, um priorisierte, verfeinerte Qualitätsszenarien zu erarbeiten. Er hat acht Schritte.

Die acht Schritte des QAW



Die ersten vier Schritte schaffen Kontext und Treiber; die letzten vier erarbeiten, bündeln, gewichten und verfeinern die Szenarien.

Was in jedem Schritt geschieht

1. **1. Präsentation und Vorstellung**
   Die Moderation erklärt Zweck und Methode; die Teilnehmenden stellen sich und ihre Anliegen vor.

2. **2. Präsentation von Geschäft oder Auftrag**
   Eine Vertretung des Geschäfts beschreibt Kontext, Ziele und übergeordnete Anforderungen: warum das System existiert und worauf es am meisten ankommt.

3. **3. Präsentation des Architekturplans**
   Eine technische Leitung stellt aktuelle Pläne, bestehende Systeme, bekannte Randbedingungen und technologische Festlegungen vor, so früh sie auch sein mögen.

4. **4. Ermittlung der Architekturtreiber**
   Die Moderation verdichtet die bisher gehörten wichtigsten Architekturtreiber (Ziele, Qualitäten, Randbedingungen) und prüft sie mit der Gruppe.

5. **5. Szenario-Brainstorming**
   Die Teilnehmenden schlagen reihum Szenarien vor; die Moderation sorgt dafür, dass jeder Treiber mindestens eines hat und dass Szenarien für normale Nutzung, Wachstum und Erkundung (Stress) alle vertreten sind.

6 **6. Bündelung der Szenarien**
Fast gleiche Szenarien werden zusammengeführt, aber nur, wenn die Vorschlagenden zustimmen, damit sich die Stimmen nicht aufteilen.

7 **7. Priorisierung der Szenarien**
Alle Teilnehmenden erhalten ein Stimmenbudget von etwa 30 % der Anzahl der Szenarien und verteilen es in zwei Runden, mit beliebig vielen Stimmen pro Szenario. Die Auszählung ergibt die Rangfolge der Szenarien.

8 **8. Verfeinerung der Szenarien**
Die vier oder fünf wichtigsten Szenarien werden vollständig ausformuliert: Quelle, Stimulus, Artefakt, Umgebung, Antwort und Antwortmaß, mit den Geschäftszielen, denen sie dienen, und offenen Fragen.

Teams, die keinen ganzen Tag erübrigen können, führen oft einen kürzeren „Mini-QAW“ durch, eine schlankere Variante, die Praktiker wie Michael Keeling, Will Chaparro und Thijmen de Gooijer beschrieben haben: mit einer Taxonomie von Qualitätsmerkmalen als Denkanstoß und ein paar Stunden in einer kleineren Gruppe. Das Wesentliche bleibt: Stakeholder gemeinsam, Szenarien statt Adjektiven, ausdrückliche Abstimmung und Verfeinerung der Gewinner.

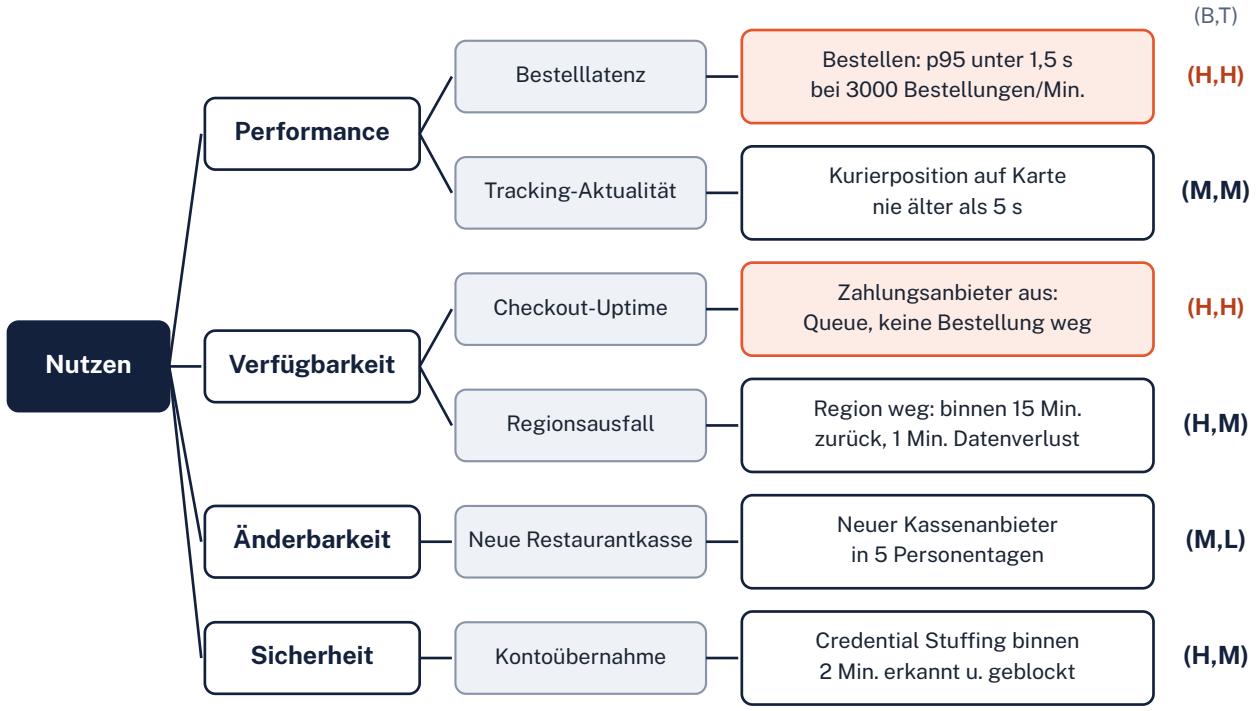
Der Utility Tree

Ein Utility Tree ist eine Methode, Qualitätsanforderungen zu ordnen und zu priorisieren; er wurde als Teil der Architecture Tradeoff Analysis Method (ATAM) eingeführt, die Rick Kazman, Mark Klein und Paul Clements am SEI entwickelt haben; die Bewertungsmethode selbst wird in Kapitel 11 behandelt. Die Wurzel ist der *Nutzen* (Utility), die Gesamtgüte des Systems. Die zweite Ebene enthält Qualitätsmerkmale (Performance, Verfügbarkeit und so weiter). Die dritte Ebene verfeinert jedes zu spezifischeren Anliegen (für Performance: Bestelllatenz, Aktualität der Sendungsverfolgung). Die Blätter sind konkrete, messbare Szenarien.

Jedes Blatt wird mit einem Paar bewertet: **geschäftliche Bedeutung** (wie wichtig das Szenario für das Geschäft ist: High, Medium oder Low) und **technische Schwierigkeit** oder Risiko (wie schwer es mit der Architektur zu erreichen ist: High, Medium oder Low). Das Paar wird (B,T) geschrieben, zum Beispiel (H,M). Mit (H,H) bewertete Szenarien sind die obersten Prioritäten der Architektur: Sie sind am wichtigsten und am schwierigsten und verdienen daher frühen Entwurfsaufwand, Prototypen und Analyse. (H,L)-Szenarien sind wichtig, aber leicht, sodass ein Standardansatz genügt; (L,H)-Szenarien sind Kandidaten zum Streichen oder Verschieben, weil sie schwer, aber wenig wert sind.

Der Baum ist auch außerhalb einer formalen Bewertung nützlich. Er macht Abwägungen auf einer Seite sichtbar, zeigt Lücken (eine Qualität ohne Szenarien) und verhindert, dass der lauteste Stakeholder die Prioritäten allein festlegt.

Ein Utility Tree für eine Essenslieferplattform



Zwei mit (H,H) bewertete, hervorgehobene Szenarien sind die obersten Prioritäten der Architektur; das (M,L)-Szenario zur Änderbarkeit lässt sich mit gewöhnlichem gutem Design erfüllen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: der Utility Tree für eine Essenslieferplattform

Eine Essenslieferplattform verbindet Kunden, 4000 Restaurants und 9000 Kuriere in einem Land. Das Unternehmen plant, das Bestellvolumen innerhalb von 18 Monaten zu verdoppeln. Die Spitze am Freitagabend liegt heute bei 3000 Bestellungen pro Minute. Ein zweistündiger QAW ergab 26 Szenarien; bei 12 Teilnehmenden hatte jede Person 8 Stimmen (30 % von 26 sind 7,8, aufgerundet auf eine gerade 8). Die Szenarien mit den meisten Stimmen wurden Blätter des Baums.

Performance: Bestelllatenz (H,H)	Kunden geben Bestellungen mit der Spitzenrate von 3000 pro Minute auf; 95 % der Bestellbestätigungen kommen innerhalb von 1,5 s zurück und 99 % innerhalb von 3 s. Hohe Bedeutung: abgebrochene Warenkörbe sind entgangener Umsatz. Hohe Schwierigkeit: Die Anfrage durchläuft die Services für Warenkorb, Preisgestaltung, Zahlung und Restaurant
Kapazitätsprüfung für dieses Szenario	3000 Bestellungen/Min. = 50 Bestellungen/s. Verdoppeltes Volumen ergibt 100 Bestellungen/s. Littles Gesetz braucht die mittlere Zeit, kein Perzentil (Abschnitt 2.4), daher liefert das Ziel von 1,5 s nur eine vorsichtige Planungsgröße: Läge der Mittelwert tatsächlich bei 1,5 s, wären etwa 100 x 1,5 = 150 Bestellungen gleichzeitig in Bearbeitung; der tatsächliche Durchschnitt ist niedriger, weil der Mittelwert deutlich unter dem p95 liegt
Verfügbarkeit: Checkout-Uptime (H,H)	Wenn beim Zahlungsanbieter während der Spitze ein Timeout auftritt, werden Bestellungen angenommen und zur Zahlung in eine Queue gestellt, keine bestätigte Bestellung geht verloren, und Kunden werden innerhalb von 10 s informiert. Schwierig, weil es asynchrone Zahlung, idempotente Wiederholungen und eine Nutzererfahrung für „ausstehend“ erfordert
Verfügbarkeit: Regionsausfall (H,M)	Fällt die Cloud-Region aus, läuft der Dienst innerhalb von 15 Minuten (RTO) in einer zweiten Region weiter, mit höchstens 1 Minute verlorener Bestelldaten (RPO). Wichtig, aber technisch gut verstanden
Sicherheit: Kontoübernahme (H,M)	Ein Credential-Stuffing-Angriff wird innerhalb von 2 Minuten erkannt und blockiert; kein durch Multi-Faktor-Authentifizierung geschütztes Konto wird übernommen
Performance: Aktualität der Sendungsverfolgung (M,M)	Die Position eines Kuriers auf der Karte des Kunden ist unter Spitzenlast nie älter als 5 s
Änderbarkeit: neuer Kassenanbieter (M,L)	Ein neuer Anbieter von Restaurant-Kassensystemen wird in 5 Personentagen integriert, indem ein Adapter hinzugefügt wird (Abschnitt 2.6)
Daraus folgende Prioritäten	Zuerst den Bestellpfad und den Entwurf mit Zahlungs-Queue prototypisch umsetzen und unter Last testen; für Regionsausfälle bewährte Multi-Region-Muster nutzen; die Adapterschnittstelle einfach halten

Bewertungen sind von der Gruppe vereinbarte Einschätzungen, keine Berechnungen. Halten Sie fest, wer was wie bewertet hat und warum, damit sich die Bewertungen später hinterfragen lassen.

Der Baum steuert den Arbeitsplan: Die beiden (H,H)-Szenarien erhalten in den ersten Sprints architektonische Spikes, während das (M,L)-Szenario keine besondere Aufmerksamkeit braucht.

Die Sendungsverfolgung der Kuriere wurde nur als mittel wichtig bewertet, was die Entwickler überraschte; der Product Owner erklärte, dass Kunden meist die geschätzte Ankunftszeit prüfen, nicht die Karte.

Priorisieren, wenn alles „kritisch“ ist

Jeder Stakeholder wird sein eigenes Anliegen hoch bewerten. Techniken, die echte Entscheidungen erzwingen, helfen. Ein festes Stimmenbudget (wie im QAW) zwingt dazu, knappe Stimmen gezielt einzusetzen. Paarweiser Vergleich („Wenn Sie nur eines haben könnten: Suche unter einer Sekunde oder null Datenverlust beim Failover?“) legt die wahre Reihenfolge offen. Geld oder Zeit neben jedes Ziel zu stellen („vier Neunen kosten zusätzlich £9000 im Monat und zwei Monate Arbeit“) ändert viele Meinungen. Und die Unterscheidung von *Randbedingungen* (gesetzliche Anforderungen an Barrierefreiheit, Datenresidenz) und *Präferenzen* verhindert, dass Nicht-Verhandelbares versehentlich weggehandelt wird.

Prioritäten sind nicht fest. Ein Start-up, das seine ersten Kunden sucht, stellt Time-to-Market und Änderbarkeit zu Recht über eine Verfügbarkeit von fünf Neunen; dasselbe Produkt muss drei Jahre später, mit Unternehmenskunden und vertraglichen SLAs, neu gewichten. Überprüfen Sie den Utility Tree, wenn sich das Geschäft ändert.

Konflikte zwischen Qualitäten

Qualitäten wirken aufeinander. Manche verstärken sich gegenseitig: Modularität hilft der Testbarkeit, die wiederum der Deploybarkeit hilft. Viele stehen im Konflikt, weil die Taktik, die einer Qualität hilft, einer anderen schadet. Die ATAM, beschrieben von Bass, Clements und Kazman, nennt die Stellen, an denen eine Architekturentscheidung mehrere Qualitäten beeinflusst, Trade-off-Punkte. Wer die üblichen Konflikte kennt, kann sie ansprechen, bevor sie jemanden überraschen.

Häufige Konflikte zwischen Qualitäten und wie der Kontext sie auflöst

Konflikt	Warum sie auseinanderstreben	Wege, ihn im Kontext aufzulösen
Performance vs. Sicherheit	Verschlüsselung, Authentifizierungsprüfungen, Eingabevalidierung und Audit-Logging erzeugen bei jeder Anfrage Mehrarbeit	Den tatsächlichen Overhead messen (mit modernem TLS und moderner Hardware oft gering); Autorisierungsentscheidungen kurz cachen; teure Prüfungen an den Rand verlagern; nie Sicherheit bei Daten opfern, deren Offenlegung ernsten Schaden anrichten würde
Verfügbarkeit vs. Konsistenz	Während einer Netzwerkpartition muss ein repliziertes System zwischen Antworten (womöglich mit veralteten Daten) und Verweigern wählen	Pro Operation entscheiden: Eine Speisekarte oder Kurierposition darf leicht veraltet sein, eine Zahlung oder Bestandsreservierung nicht. Siehe CAP und PACELC in Kapitel 6
Änderbarkeit vs. Performance	Schichten, Schnittstellen, Vermittler und spätes Binden fügen Indirektion und zusätzliche Hops hinzu	Die Indirektion dort beibehalten, wo Änderungen wahrscheinlich sind; den gemessenen Hot Path optimieren; einige bewusste Abkürzungen akzeptieren, dokumentiert in einem Entscheidungsdokument
Kosten vs. fast alles	Redundanz, zusätzliche Regionen, mehr Tests und schnellere Hardware kosten alle Geld	Jedes Qualitätsziel an seinen Geschäftswert knüpfen; Error Budgets und Stückkosten nutzen; Neunen nur dort kaufen, wo ein Ausfall mehr kostet als seine Verhinderung
Benutzbarkeit vs. Sicherheit	Starke Authentifizierung, Timeouts und Bestätigungen erzeugen Reibung	Risikobasierte Authentifizierung (Step-up nur bei riskanten Aktionen), Passkeys und Single Sign-on, sinnvolle Sitzungsdauern; Nutzer in Tests einbeziehen
Time-to-Market vs. Wartbarkeit	Abkürzungen beschleunigen das erste Release und bremsen spätere	Technische Schulden bewusst aufnehmen, festhalten und ihre Tilgung einplanen (Kapitel 1)

Die richtige Lösung hängt immer vom Kontext ab. „Sicherheit schlägt Performance“ ist ein Slogan, keine Entscheidung: 2 ms zusätzlicher TLS-Handshake auf einer öffentlichen Website sind belanglos, während synchrone Betrugsprüfungen bei jeder Kartenzahlung in einem vollen Stadion ein echtes Problem sein können, das eine architektonische Antwort verdient (ein Modell „erst genehmigen, dann prüfen“ mit Limits). Fragen Sie: Welchem Geschäftsziel dient jede Seite, was kostet jede Option, und wer muss zustimmen?

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: ein Konflikt, zwei Antworten

Der Punkt, an dem hier die meisten stolpern, ist, dass ein Konflikt zwischen zwei Qualitäten nicht ein für alle Mal für das ganze System entschieden wird. Er wird Operation für Operation entschieden, nach dem, was jede dem Geschäft wert ist. Verfolgen Sie die Tabellenzeile Verfügbarkeit gegen Konsistenz durch die Essenslieferplattform.

- Angenommen, die Plattform hält Kopien einiger ihrer Daten auf mehr als einer Maschine, um einen Ausfall zu überstehen, und gibt jede Änderung von einer Kopie an die anderen weiter.
- Nehmen Sie nun an, die Netzwerkverbindung zwischen zwei Kopien bricht eine Weile ab: eine Netzwerkpartition. Eine Kopie, die von den neuesten Änderungen nichts erfährt, kann entweder mit dem weiterantworten, was sie hat, was veraltet sein kann, oder die Antwort verweigern, bis die Verbindung zurück ist. Beides zugleich kann sie nicht zusagen. Kapitel 6 kommt auf diese Wahl zurück.
- Kurierposition: Ein Punkt auf der Karte des Kunden, der ein paar Sekunden alt ist, richtet wenig Schaden an, und Kunden prüfen ohnehin meist die geschätzte Ankunftszeit. Verfügbarkeit gewinnt: weiter antworten.
- Zahlung: Auf veraltete Informationen hin zu handeln könnte bedeuten, einen Kunden doppelt zu belasten oder eine Bestellung als bezahlt zu behandeln, obwohl die Zahlung nie durchging. Konsistenz gewinnt: die Zahlung zurückhalten und die Bestellung als ausstehend anzeigen, statt zu raten.
- Dieselbe Plattform gibt also auf denselben Konflikt entgegengesetzte Antworten. Keine davon stammt aus einem Slogan wie ‚Konsistenz schlägt Verfügbarkeit‘; jede folgt daraus, was eine falsche Antwort bei dieser Operation kosten würde.
- **Die Lehre:** Wenn zwei Qualitäten im Konflikt stehen, fragen Sie, welchem Geschäftsziel jede Seite bei genau dieser Operation dient, und entscheiden Sie dort. Halten Sie dann die Wahl und ihre Gründe fest, wie der nächste Teil dieses Abschnitts erklärt, damit niemand sie später rückgängig macht, ohne zu wissen, warum.

Entscheidungen festhalten und Qualitäten verfolgen

Eine Abwägung, die nicht aufgeschrieben ist, wird erneut diskutiert, oft von Leuten, die nicht dabei waren. Halten Sie wesentliche Abwägungen als Architecture Decision Records fest (Kapitel 4): den Kontext, die Qualitäten im Spannungsfeld, die erwogenen Optionen, die Entscheidung und ihre Konsequenzen, einschließlich dessen, worauf Sie bewusst verzichtet haben.

Machen Sie dann jedes Qualitätsziel messbar und sichtbar. Tom Gilbs Planguage bietet ein diszipliniertes Format: eine *Skala* (was gemessen wird, in welcher Einheit), ein *Messverfahren* (wie gemessen wird) und Stufen wie *Fail* (die Schwelle, unterhalb derer die Anforderung nicht erfüllt ist), *Goal* (das Ziel) und *Stretch* oder *Wish* (ehrgeizigere Stufen). Angewandt auf das Szenario zur Bestelllatenz: Skala, Sekunden vom Absenden bis zur Bestätigung beim 95. Perzentil; Messverfahren, Real-User-Monitoring während der Freitagsspitze; Fail über 3 s; Goal 1,5 s; Stretch 1 s.

Verfolgen Sie die Maße schließlich über die Zeit auf einem Dashboard, das die Stakeholder einsehen können, neben den SLOs (Kapitel 9), und überprüfen Sie sie, sobald sich Geschäft, Last oder Team ändern.

Von Szenarien zu Fitness Functions

Der stärkste Weg, eine Qualitätsanforderung dauerhaft zu erfüllen, ist, sie bei jeder Änderung des Systems automatisch zu prüfen. Neal Ford, Rebecca Parsons und Patrick Kua nennen eine solche Prüfung in *Building Evolutionary Architectures* (2017; zweite Ausgabe mit Pramod Sadalage, 2022) eine architektonische Fitness Function: eine objektive Bewertung, wie gut das System ein Architekturmerkmal erfüllt. Ein Lasttest in der Pipeline, der den Build scheitern lässt, wenn die p95-Bestelllatenz bei 100 Bestellungen pro Sekunde 1,5 s übersteigt, ist eine Fitness Function. Ebenso ein Test, der fehlschlägt, wenn irgendein Modul außerhalb von `payments` die Bibliothek des Zahlungsanbieters importiert (zum Schutz des Änderbarkeitsszenarios), eine geplante Failover-Übung, die die Wiederherstellungszeit misst, oder ein Abhängigkeitsscan, der bekannte verwundbare Bibliotheken blockiert.

Fitness Functions sind der natürliche Endpunkt der Arbeit in diesem Kapitel: Ein vager Wunsch wird zu einem Szenario (Abschnitt 2.3), das Szenario erhält eine Priorität im Utility Tree, und das Antwortmaß des Szenarios wird zu einer automatisierten Prüfung. Nicht jede Qualität lässt sich günstig automatisieren (Benutzbarkeit braucht weiterhin Menschen), doch jede, die Sie automatisieren, verhindert eine schleichende Erosion. Kapitel 11 behandelt ihren Entwurf ausführlich: Fitness Functions und evolutionäre Architektur.

Checkliste: einen Quality Attribute Workshop durchführen

Angepasst für eine halb- oder ganztägige Sitzung; ein kürzerer „Mini-QAW“ kann mit einer kleineren Gruppe dasselbe abdecken.

- [ ] **Die richtige Mischung einladen**
  5 bis 30 Personen: Fachverantwortliche, Nutzer oder deren Vertretungen, Betrieb, Sicherheit, Support, Entwicklung und alle, die Compliance abzeichnen. Am häufigsten fehlt der Betrieb.
- [ ] **Eine kurze Vorablektüre verschicken**
  Die Geschäftsziele, bekannte Randbedingungen und ein Entwurf der Liste der Qualitätsmerkmale (ISO/IEC 25010 als Denkanstoß), damit alle schon nachdenkend ankommen.
- [ ] **Mit Geschäftszielen beginnen, nicht mit Technologie**
  Ein ranghoher Stakeholder erklärt, warum das System existiert, wie Erfolg aussieht und was ein Scheitern kosten würde.
- [ ] **Die Architekturtreiber vereinbaren**
  Eine kurze Liste der Ziele, Qualitäten und Randbedingungen, die die Architektur prägen werden, mit der Gruppe geprüft, bevor die Szenarien beginnen.
- [ ] **Szenarien in Runden sammeln**
  Alle tragen reihum mindestens zwei bei, damit auch leise Stimmen gehört werden. Berücksichtigen Sie normale Nutzung, Wachstum („in zwei Jahren“) und Stress („wenn der Zahlungsanbieter ausfällt“).
- [ ] **Duplikate nur mit Zustimmung zusammenführen**
  Das Bündeln verhindert, dass sich Stimmen aufteilen, doch die vorschlagende Person muss zustimmen, dass zwei Szenarien wirklich dasselbe meinen.
- [ ] **Mit festem Budget abstimmen**
  Jede Person erhält Stimmen in Höhe von etwa 30 % der Szenarienzahl, verteilt auf zwei Runden. Das legt echte Prioritäten offen und vermeidet „alles ist kritisch“.
- [ ] **Die wichtigsten wenigen vollständig verfeinern**
  Schreiben Sie die vier oder fünf wichtigsten als sechsteilige Szenarien mit messbaren Antwortmaßen aus und notieren Sie das Geschäftsziel, dem jedes dient.
- [ ] **Offene Fragen und Konflikte festhalten**
  Meinungsverschiedenheiten über Priorität oder Machbarkeit sind wertvolle Ergebnisse; listen Sie sie mit einer verantwortlichen Person auf.

- [ ] **Das Ergebnis innerhalb weniger Tage veröffentlichen**
  Teilen Sie die priorisierten Szenarien, nehmen Sie sie in die Architekturdokumentation und das Backlog auf und planen Sie, wann jedes geprüft wird.

ACHTUNG

Häufige Fehler

- Qualitäten nur beim geschäftlichen Auftraggeber erheben: Anliegen von Betrieb, Sicherheit und Support tauchen dann nach dem Go-live auf, wenn sie teuer sind.
- Adjektive akzeptieren: „skalierbar“ und „robust“ auf einer Anforderungsliste, ohne ein Szenario dahinter.
- Jedes Blatt mit (H,H) bewerten: Wenn alles oberste Priorität hat, hat nichts Priorität. Haken Sie nach, bis die Bewertungen unterscheiden.
- Den Utility Tree als einmaliges Artefakt behandeln: Er sollte überprüft werden, wenn sich Ziele oder Last ändern.
- Konflikte stillschweigend im Code lösen: Wer still das Audit-Logging weglässt, um ein Latenzziel zu erreichen, hat eine Architekturentscheidung getroffen, die niemand festgehalten hat.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Wie viele Stimmen erhält jede teilnehmende Person in einem QAW mit 20 gebündelten Szenarien ungefähr?

Etwa 30 % von 20, also 6 Stimmen, verteilt auf zwei Runden zu je 3.

▼ Ein Blatt des Utility Tree ist mit (L,H) bewertet. Was legt das nahe?

Es ist für das Geschäft wenig wichtig, aber schwer zu erreichen; es ist also ein Kandidat zum Streichen, Verschieben oder Neuverhandeln, statt die Architektur zu bestimmen.

▼ Warum ist „Sicherheit schlägt immer Performance“ ein schlechter Weg, einen Konflikt zu lösen?

Weil die richtige Antwort vom tatsächlichen Overhead, vom Schaden, den die Sicherheitsmaßnahme verhindert, und vom Geschäftswert der Performance abhängt; der Konflikt muss im Einzelfall mit Zahlen gelöst und festgehalten werden.

▼ **Nennen Sie ein Beispiel für eine Fitness Function für eine Anforderung an die Änderbarkeit.**

Ein automatisierter Test im Build, der fehlschlägt, wenn irgendein Modul außer dem Zahlungsadapter die Bibliothek des Zahlungsanbieters importiert, sodass der Anbieter austauschbar bleibt.

▼ **Probieren Sie es aus: Stakeholder bezeichnen alles als kritisch. Bewerten Sie diese Punkte (Wichtigkeit, Schwierigkeit) für den Utility Tree einer Essenslieferplattform: Bestellbestätigung innerhalb von 1,5 s zur Spitzenzeit; Export des Monatsberichts innerhalb einer Stunde; Hinzufügen eines Zahlungsanbieters innerhalb eines Monats. Was bekommt als Erstes Entwurfsaufmerksamkeit?**

Die Bestellbestätigung ist (H,H): Verlorene Warenkörbe sind verlorener Umsatz, und die Anfrage durchläuft unter Spitzenlast mehrere Services, sie bekommt also zuerst Designaufmerksamkeit. Ein neuer Zahlungsanbieter könnte (M,M) sein: wertvoll und mäßig schwierig, sofern nicht schon ein Adapter existiert. Der Berichtsexport ist (L,L) oder (M,L): leicht über Nacht auszuführen. Die Bewertungen erzwingen eine Rangfolge, ohne behaupten zu müssen, eine Qualität sei immer mehr wert. Dokumentieren Sie die Gründe, und prüfen Sie sie per Abstimmung mit den Stakeholdern.

QUELLEN

- SEI: Quality Attribute Workshop collection: https://www.sei.cmu.edu/library/quality-attribute-workshop-collection/
- SEI: Architecture Tradeoff Analysis Method (ATAM) collection: https://www.sei.cmu.edu/library/architecture-tradeoff-analysis-method-collection/
- IREB RE Magazine: Discover Quality Requirements with the Mini-QAW: https://re-magazine.ireb.org/articles/discover-quality-requirements-with-the-mini-qaw
- Thoughtworks Technology Radar: Architectural fitness function: https://www.thoughtworks.com/radar/techniques/architectural-fitness-function

Weiterführende Literatur

- *Software Architecture in Practice (4th edition)*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. Die Quelle der Qualitätsszenarien und Taktiken, mit je einem Kapitel zu jeder wichtigen Qualität und einem zu architekturrelevanten Anforderungen.
  https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- *ISO/IEC 25010:2023 Systems and software engineering: Product quality model*, ISO/IEC JTC 1/SC 7 (2023), International Organization for Standardization. Das Standard-Produktqualitätsmodell mit neun Merkmalen; die Begleitnorm ISO/IEC 25019:2023 behandelt die Nutzungsqualität.
  https://www.iso.org/standard/78176.html
- *Characterizing Architecturally Significant Requirements*, Lianping Chen, Muhammad Ali Babar and Bashar Nuseibeh (2013), IEEE Software, volume 30, issue 2. Eine empirische Studie darüber, wie Praktiker die Anforderungen erkennen, die eine Architektur prägen.
  https://pure.itu.dk/en/publications/characterizing-architecturally-significant-requirements/
- *The Tail at Scale*, Jeffrey Dean and Luiz André Barroso (2013), Communications of the ACM, volume 56, issue 2. Ein kurzes, gut lesbares Paper darüber, warum seltene langsame Antworten große Fan-out-Systeme dominieren, und über die Techniken, mit denen Google sie bändigt.
  https://research.google/pubs/pub40801/
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. Klare Darstellung von Last, Perzentilen, Zuverlässigkeit und Skalierbarkeit in Datensystemen, mit vielen Praxisbeispielen.
- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. Die Sicht eines Praktikers darauf, wie man Architekturmerkmale identifiziert, priorisiert und mit Fitness Functions schützt.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (eds) (2016), O'Reilly Media. Online frei lesbar; die Kapitel über Service Level Objectives und Monitoring zeigen, wie Qualitäten im Produktivbetrieb gemessen werden.
  https://sre.google/books/
- *Systems Performance: Enterprise and the Cloud (2nd edition)*, Brendan Gregg (2020), Addison-Wesley. Das Standardwerk zur Performance-Analyse, von Warteschlangen und Auslastung bis zur USE-Methode.

Glossar

Allgemeines Szenario
Eine systemunabhängige Vorlage, die die typischen Quellen, Stimuli, Umgebungen, Reaktionen und Maße für ein Qualitätsmerkmal auflistet.

Amdahlsches Gesetz
Eine Formel (Gene Amdahl, 1967), die zeigt, dass der Geschwindigkeitsgewinn durch zusätzliche Prozessoren durch den Teil der Arbeit begrenzt ist, der seriell laufen muss.

Änderbarkeit
Wie kostengünstig und sicher sich ein System ändern lässt: die Kosten, der Aufwand und das Risiko einer Änderung und wie weit sie über die Stelle hinaus wirkt, an der sie vorgenommen wird.

Anforderungserhebung
Das aktive Herausarbeiten von Anforderungen aus Stakeholdern, Dokumenten, Vorschriften und bestehenden Systemen, statt darauf zu warten, dass sie genannt werden.

Antwortmaß (Response Measure)
Die Zahl, die ein Szenario testbar macht, etwa „99 % innerhalb von 800 ms“, „Service innerhalb von 2 Minuten wiederhergestellt“ oder „Änderung in 3 Personentagen abgeschlossen“.

Antwortzeit
Die Gesamtzeit zwischen dem Senden einer Anfrage durch einen Client und dem Empfang der Antwort: die Bedienzeit plus etwaige Wartezeiten in Warteschlangen plus Netzwerkverzögerungen.

Architekturrelevante Anforderung (ASR)
Eine Anforderung, die messbare Auswirkungen auf die Architektur eines Systems hat: Würde sie sich ändern, müsste sich die Struktur ändern. Die meisten ASRs sind Qualitätsmerkmale oder Randbedingungen.

Architekturtreiber
Ein Ziel, eine Qualitätsanforderung oder eine Randbedingung, die die Architektur wesentlich prägt.

Ausfall (Failure)
Ein Ereignis, bei dem der von einem System erbrachte Dienst für seine Benutzer erkennbar von dem abweicht, was er sein sollte.

Auslastung
Der Anteil der Zeit, in der eine Ressource beschäftigt ist, meist als ρ (rho) geschrieben. Ein Server, der 40 Anfragen pro Sekunde mit je 20 ms Arbeit bearbeitet, ist zu 80 % ausgelastet.

Barrierefreiheit
Das Maß, in dem Menschen mit Behinderungen, auch solche, die assistive Technologien wie Screenreader nutzen, ein Produkt verwenden können.

Bedienzeit (Service Time)
Die Zeit, die eine Ressource tatsächlich mit der Bearbeitung einer Anfrage verbringt, ohne die Zeit, die die Anfrage auf diese Ressource wartet.

Betriebssicherheit (Safety)
Freiheit von unvertretbaren Risiken eines Schadens für Menschen, Sachwerte oder die Umwelt, der durch das System verursacht wird, unabhängig davon, ob jemand böswillig handelt.

Bindungszeitpunkt
Der Zeitpunkt, zu dem eine Entscheidung festgelegt wird: beim Schreiben des Codes, beim Build, beim Deployment, beim Start oder während das System läuft.

CIA-Triade
Die drei klassischen Ziele der Informationssicherheit: Vertraulichkeit (nur Berechtigte können lesen), Integrität (keine unbefugten Änderungen) und Verfügbarkeit (nutzbar, wenn benötigt).

Datenresidenz
Eine Anforderung, dass bestimmte Daten nur in einem bestimmten Land oder einer bestimmten Region gespeichert und manchmal auch verarbeitet werden.

Dependency Injection
Einer Komponente die benötigten Kollaborateure (eine Datenbank, eine Uhr, ein Zahlungs-Gateway) von außen übergeben, statt sie diese selbst erzeugen zu lassen, damit sie sich austauschen lassen, zum Beispiel gegen Fakes in Tests.

Deploybarkeit
Wie schnell, sicher und vorhersehbar eine Änderung in Produktion gebracht und bei Bedarf wieder zurückgenommen werden kann.

DORA-Metriken
Die Kennzahlen der Softwareauslieferung aus dem DORA-Forschungsprogramm: Change Lead Time, Deployment-Häufigkeit, Wiederherstellungszeit nach fehlgeschlagenem Deployment, Change Fail Rate und Deployment Rework Rate.

Durchsatz
Die Arbeitsmenge, die ein System pro Zeiteinheit erledigt, etwa 1200 Anfragen pro Sekunde oder 40.000 Bestellungen pro Stunde.

Elastizität
Die Fähigkeit, Ressourcen automatisch hinzuzufügen und zu entfernen, wenn die Last steigt und fällt, sodass die Kapazität dem Bedarf folgt und Sie nur für das zahlen, was Sie nutzen.

Failover
Das Umschalten der Arbeit von einer ausgefallenen Komponente auf einen funktionsfähigen Standby oder ein Replikat, automatisch oder von Hand.

Fan-out
Ein Muster, bei dem eine eingehende Anfrage mehrere parallele Anfragen an andere Services oder Server auslöst, deren Antworten dann zusammengeführt werden.

Fehler (Error)
Ein Teil des internen Zustands eines Systems, der falsch ist, weil eine Fehlerursache (Fault) aktiviert wurde, etwa ein beschädigter Datensatz. Ein Fehler kann zu einem Ausfall (Failure) führen, muss es aber nicht.

Fehlertoleranz
Die Fähigkeit eines Systems, seinen Dienst weiter zu erbringen, obwohl einer seiner Teile ausgefallen ist, meist durch Redundanz.

Fehlerursache (Fault)
Die Ursache eines Problems, etwa ein Bug, eine defekte Festplatte oder eine Fehlkonfiguration. Eine Fehlerursache kann ruhen, bis etwas sie aktiviert.

Funktionale Anforderung
Eine Aussage darüber, was ein System tun muss: ein Verhalten, eine Berechnung oder eine Information, die es bereitstellen muss, etwa „ein Kunde kann eine Bestellung stornieren, bevor sie versandt wird“.

FURPS+
Eine Qualitätsklassifikation, die bei Hewlett-Packard als FURPS (Functionality, Usability, Reliability, Performance und Supportability) entstand und später um „+“ für Randbedingungen wie Design, Implementierung, Schnittstellen und physische Grenzen erweitert wurde.

Graceful Degradation
Weiterhin die wichtigsten Funktionen bereitstellen, wenn ein Teil eines Systems ausfällt, statt vollständig auszufallen, zum Beispiel ein Shop, der noch Bestellungen annimmt, während die Empfehlungen offline sind.

Heartbeat
Eine regelmäßige „Ich lebe noch“-Nachricht einer Komponente; bleiben die Nachrichten aus, nimmt ein Monitor an, dass die Komponente ausgefallen ist, und handelt.

Implizite Anforderung
Eine Qualität, die alle Stakeholder voraussetzen, aber niemand aufschreibt, etwa „Bestellungen gehen nie verloren“ oder „das System funktioniert auch bei einer Zeitumstellung weiter“.

Interoperabilität
Das Maß, in dem Systeme Informationen austauschen und das Ausgetauschte korrekt verwenden können, was Einigkeit über die Bedeutung ebenso wie über das Format erfordert.

ISO/IEC 25010
Das internationale Standardmodell für Produktqualität. Die Ausgabe von 2023 definiert neun Merkmale, von funktionaler Eignung bis Betriebssicherheit, jedes in Teilmerkmale gegliedert.

Konkretes Szenario
Ein Szenario, das für ein bestimmtes System formuliert ist, mit konkreten Akteuren, Ereignissen, Bedingungen und Zahlen, bereit für Entwurf und Test.

Littles Gesetz
Die Regel $L = \lambda \times W$: Die durchschnittliche Anzahl der Elemente in einem stabilen System ist gleich der durchschnittlichen Ankunftsrate multipliziert mit der durchschnittlichen Verweildauer jedes Elements darin.

Mean Time Between Failures (MTBF)
Die durchschnittliche Zeit von einem Ausfall einer reparierbaren Komponente oder eines Systems bis zum nächsten.

Mean Time to Failure (MTTF)
Die durchschnittliche Zeit, die eine Komponente fehlerfrei läuft, bevor sie ausfällt.

Mean Time to Repair (MTTR)
Die durchschnittliche Zeit von einem Ausfall bis zur Wiederherstellung des Service. Manche Organisationen lesen das R als „recover“ oder „restore“; geben Sie immer an, welcher Start- und Endpunkt gemessen wird.

Nichtfunktionale Anforderung (NFR)
Eine weit verbreitete, aber unscharfe Bezeichnung für Qualitätsanforderungen und Randbedingungen. Viele Architekten vermeiden sie, weil diese Anforderungen Funktionen näher bestimmen, statt losgelöst von ihnen zu stehen.

Nutzungsqualität
Die Qualität, die ein Produkt Menschen in einem bestimmten Nutzungskontext bietet (ob es nützlich, risikofrei und akzeptabel ist). Seit 2023 hat sie eine eigene Norm, ISO/IEC 25019.

Observability
Wie gut Menschen anhand der Daten, die ein laufendes System ausgibt (Logs, Metriken, Traces), verstehen können, was darin vorgeht, auch bei Problemen, mit denen niemand gerechnet hat.

Perzentil (p95, p99)
Der Wert, unter dem ein bestimmter Anteil der Messungen liegt. Ein p99 von 400 ms bedeutet, dass 99 % der Anfragen 400 ms oder weniger und 1 % länger gedauert haben.

Qualitätsmodell
Eine strukturierte Liste von Qualitätsmerkmalen und ihren Teilmerkmalen, die als gemeinsames Vokabular und als Checkliste beim Festlegen oder Bewerten von Qualität dient.

Qualitätsszenario
Eine kurze, testbare Formulierung einer Qualitätsanforderung aus sechs Teilen: Quelle des Stimulus, Stimulus, Artefakt, Umgebung, Antwort und Antwortmaß.

Quality Attribute Workshop (QAW)
Eine moderierte Methode des SEI, bei der Stakeholder zu Beginn eines Projekts Qualitätsszenarien sammeln, zusammenführen, darüber abstimmen und sie verfeinern.

Randbedingung
Eine Designentscheidung, die bereits für Sie getroffen wurde und keinen oder nur sehr wenig Spielraum lässt, etwa eine vorgeschriebene Programmiersprache, eine Vorschrift, eine feste Frist oder ein Budget.

Recovery Point Objective (RPO)
Die größte Datenmenge, gemessen als Zeitspanne, deren Verlust das Unternehmen in einem Katastrophenfall hinnehmen kann, zum Beispiel die Änderungen der letzten 15 Minuten.

Recovery Time Objective (RTO)
Die längste Zeit, die das Unternehmen zwischen einer Katastrophe und der Wiederherstellung des Service hinnehmen kann.

Redundanz
Das Vorhalten mehrerer Exemplare einer Komponente, damit ein anderes übernehmen kann, wenn eines ausfällt.

Resilienz
Die umfassendere Fähigkeit eines Systems, Störungen aufzufangen, seinen wesentlichen Dienst (gegebenenfalls eingeschränkt) aufrechtzuerhalten und sich danach in den Normalzustand zurückzuversetzen.

Rollback
Die Rückführung eines Systems auf einen früheren, als gut bekannten Zustand oder ein früheres Release, nachdem eine Änderung Probleme verursacht hat.

Service Level Agreement (SLA)
Ein Vertrag mit Kunden, der ein Service-Level und die Folgen bei Nichteinhaltung festlegt, etwa Service-Gutschriften.

Service Level Indicator (SLI)
Eine sorgfältig definierte Messgröße für das Verhalten eines Service, etwa der Anteil der Anfragen, die in weniger als 300 ms erfolgreich sind.

Service Level Objective (SLO)
Ein internes Ziel für einen SLI über ein Zeitfenster, etwa 99,9 % erfolgreiche Anfragen über 30 Tage.

Skalierbarkeit
Die Fähigkeit eines Systems, durch zusätzliche Ressourcen mehr Last zu bewältigen, idealerweise mit einem vorhersehbaren und annähernd proportionalen Anstieg von Kapazität und Kosten.

Software Carbon Intensity (SCI)
Eine Spezifikation der Green Software Foundation, veröffentlicht als ISO/IEC 21031:2024, die die CO_2-Emissionen von Software als Rate ausdrückt: ((Energie x CO_2-Intensität des Stromnetzes) + graue Emissionen) pro funktionaler Einheit.

Stimulus
Das Ereignis, um das es in einem Qualitätsszenario geht, etwa ein Anfrageschub, ein Serverabsturz, ein Änderungswunsch oder ein Angriff.

Stückkosten
Die Betriebskosten eines Systems geteilt durch eine Einheit nützlicher Arbeit, etwa Kosten pro Bestellung oder pro aktivem Nutzer, damit sich die Effizienz verfolgen lässt, während das Unternehmen wächst.

Tail-Latenz
Die Antwortzeiten der langsamsten Anfragen, meist durch hohe Perzentile wie p99 oder p99,9 beschrieben. Bei großer Skalierung entscheidet oft das Ende der Verteilung darüber, was Nutzer erleben.

Taktik
Eine Entwurfsentscheidung, die beeinflusst, wie ein System auf einen Qualitätsstimulus reagiert, etwa Caching zur Senkung der Latenz oder Redundanz, um einen Absturz zu überstehen. Taktiken sind die Bausteine von Mustern.

Teilmerkmal
Eine spezifischere Qualität innerhalb eines Merkmals eines Qualitätsmodells; so sind Fehlertoleranz und Wiederherstellbarkeit in ISO/IEC 25010 Teilmerkmale der Zuverlässigkeit.

Test Double
Ein Ersatz für einen echten Kollaborateur in einem Test, etwa ein Stub mit vorgefertigten Antworten, ein Fake mit einer einfachen funktionierenden Implementierung oder ein Mock, der die empfangenen Aufrufe prüft.

Testbarkeit
Wie leicht sich durch Tests zeigen lässt, dass sich ein System oder eine Komponente wie vorgesehen verhält; das hängt davon ab, ob sich Eingaben und Zustand steuern und Ausgaben beobachten lassen.

Trade-off-Punkt
Eine Architekturentscheidung, die mehr als ein Qualitätsmerkmal beeinflusst und eines verbessert, während sie einem anderen schadet.

Universal Scalability Law (USL)

Neil Gunthers Modell dafür, wie die Kapazität mit zusätzlichen Knoten wächst, unter Berücksichtigung der Konkurrenz um gemeinsame Ressourcen und der Kosten, Daten kohärent zu halten. Es sagt ein Maximum voraus, ab dem weitere Knoten die Kapazität verringern.

Utility Tree

Ein Baum aus der ATAM, der den Gesamtnutzen in Qualitätsmerkmale und dann in konkrete Szenarien verfeinert, die jeweils nach geschäftlicher Bedeutung und technischer Schwierigkeit bewertet werden.

Verfügbarkeit

Der Anteil der Zeit oder der Anfragen, in dem ein System bereit ist, seinen Dienst auf Anforderung zu erbringen, meist als Prozentsatz angegeben, etwa 99,9 %.

Wartbarkeit

In ISO/IEC 25010, wie wirksam und effizient sich ein Produkt ändern lässt, mit den Teilmerkmalen Modularität, Wiederverwendbarkeit, Analysierbarkeit, Änderbarkeit und Testbarkeit.

Warteschlangenbildung (Queueing)

Das Warten auf eine ausgelastete Ressource. Die Warteschlangentheorie untersucht, wie die Wartezeit mit der Auslastung wächst; sie erklärt, warum die Antwortzeit steil ansteigt, wenn eine Ressource sich 100 % Auslastung nähert.

WCAG

Die Web Content Accessibility Guidelines des W3C. Version 2.2 (2023) legt prüfbare Erfolgskriterien auf den Stufen A, AA und AAA fest; Stufe AA ist der übliche rechtliche Maßstab.

Zuverlässigkeit

Die Wahrscheinlichkeit, dass ein System für einen festgelegten Zeitraum unter festgelegten Bedingungen korrekt und ohne Ausfall arbeitet. Ein System kann hochverfügbar und dennoch unzuverlässig sein, wenn es oft ausfällt, sich aber schnell erholt.