BE SOFTWARE ARCHITECTURE WISE · KAPITEL 11

Bewertung, Entscheidungsfindung und Governance

In diesem Kapitel geht es darum, Architekturen über die Zeit zu bewerten, zu steuern und weiterzuentwickeln. Es behandelt die formale Bewertung mit ATAM, leichtgewichtige Reviews wie Risk-Storming und Pre-Mortems, Fitness Functions und evolutionäre Architektur, Governance in agilen Organisationen, Team Topologies und soziotechnisches Design, die Modernisierung von Altsystemen mit dem Strangler Fig, Führung und Einfluss sowie die wirtschaftliche und ethische Begründung von Architekturarbeit. Es führt alle vorherigen Kapitel zusammen, nutzt die Essenslieferplattform als durchgehendes Beispiel und richtet sich an erfahrene Entwicklerinnen und Entwickler, technische Leads und Architekten, die wesentliche Entscheidungen treffen oder begleiten.

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

AUF EINEN BLICK

Das lernen Sie

- Erklären, wann und warum eine Architektur bewertet werden sollte, und die neun Schritte von ATAM durchführen, um Risiken, Nicht-Risiken, Sensitivitätspunkte, Trade-off-Punkte und Risikothemen zu ermitteln.
- Leichtgewichtige Reviews wie Risk-Storming und Pre-Mortems moderieren und ihre Ergebnisse in ein Risikoregister mit Verantwortlichen überführen.
- Fitness Functions entwerfen, die Qualitätsszenarien in automatisierte Prüfungen von Struktur, Performance, Sicherheit, Resilienz und Daten verwandeln.
- Architekturprinzipien formulieren und für unterschiedliche Arten von Entscheidungen zwischen Guardrails, Gates und dem Advice Process wählen.
- Teamtypen, Interaktionsmodi und kognitive Last aus Team Topologies anwenden, um Teams auf die angestrebte Architektur auszurichten.
- Eine schrittweise Modernisierung von Altsystemen mit Strangler Fig, Branch by Abstraction und Parallelbetrieb planen.
- Architekturentscheidungen ohne formale Autorität führen und beeinflussen, auch in der Kommunikation mit der Geschäftsleitung.
- Kapitalwert und Amortisationszeit für einen Business Case berechnen und eine Architektur auf ethische und Nachhaltigkeitsanliegen prüfen.

Bevor Sie beginnen

Kapitel 1 bis 4, besonders Entscheidungen und Abwägungen (Abschnitt 1.6), technische Schulden (Abschnitt 1.7), Qualitätsszenarien und Utility Trees (Abschnitte 2.3 und 2.8), das Gesetz von Conway (Abschnitt 3.7) und Entscheidungsdokumente (Abschnitt 4.5). Die Kapitel 5 bis 10 helfen, sind aber nicht zwingend nötig.

Dauer

Etwa 3½ Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

ATAM · Utility Tree · Sensitivitätspunkt · Trade-off-Punkt · Risk-Storming · Pre-Mortem · Fitness Function · Evolutionäre Architektur · Advice Process · Technologieradar · Team Topologies · Strangler Fig · Kapitalwert · Software Carbon Intensity

Architektur bewerten, steuern und weiterentwickeln

Die vorherigen Kapitel haben gezeigt, wie man beschreibt, was ein System leisten muss, wie man es strukturiert und wie man es schnell, zuverlässig, sicher und wirtschaftlich macht. In diesem Kapitel geht es um die Arbeit, die diese Entscheidungen über die gesamte Lebensdauer eines Systems begleitet: prüfen, ob sie tragfähig sind, bevor Geld gebunden wird; sie gültig halten, während jeden Monat Hunderte von Änderungen eingehen; festlegen, wer in einer Organisation mit vielen Teams welche Entscheidungen treffen darf; Teams und Altsysteme

umgestalten; Menschen überzeugen; und die Kosten der Architekturarbeit wirtschaftlich und ethisch begründen. Hier kommt der Kurs zusammen, daher verweist das Kapitel oft zurück: auf Qualitätsszenarien und Utility Trees in Kapitel 2, auf Prinzipien und Metriken in Kapitel 3, auf Entscheidungsdokumente in Kapitel 4, auf das Gesetz von Conway in Abschnitt 3.7 und auf technische Schulden in Abschnitt 1.7.

Das Niveau ist fortgeschritten, aber jeder Abschnitt beginnt mit einer kurzen Erklärung für Neulinge, und das durchgehende Beispiel ist die seit Kapitel 2 verwendete Essenslieferplattform: Kunden, Restaurants und Kuriere, Services für Bestellungen, Disposition und Benachrichtigungen, ein Event Broker und ein externer Zahlungsanbieter. Sie sehen, wie ihre Architektur bewertet, per Risk-Storming untersucht, durch Fitness Functions geschützt, über Prinzipien und ein Technologieradar gesteuert, um Teams herum neu organisiert und teilweise modernisiert wird.

Entscheidungen bewerten und schützen: Abschnitte 11.1 bis 11.3

Abschnitt 11.1 behandelt die formale Architekturbewertung und die ATAM des SEI: warum und wann bewertet wird, ihre neun Schritte in zwei Sitzungen, wer teilnimmt, und ihre charakteristischen Ergebnisse: Risiken, Nicht-Risiken mit ihren Annahmen, Sensitivitätspunkte, Trade-off-Punkte und Risikothemen. Der Utility Tree aus Abschnitt 2.8 kehrt in seinen ursprünglichen Zusammenhang zurück, und CBAM zeigt, wie man den Nutzen jeder Abhilfe gegen ihre Kosten abwägt. Eine verdichtete ATAM der Essenslieferplattform zeigt die Tabellen, die eine Bewertung tatsächlich hervorbringt.

Abschnitt 11.2 erklärt, warum monatliche Review-Gremien so oft scheitern und was stattdessen funktioniert: Simon Browns Risk-Storming, Gary Kleins Pre-Mortem, Ted Newards Architektur-Katas, Eoin Woods' TARA für bestehende Systeme, schriftliche Design-Reviews und Checklisten. Am Ende stehen ein Schritt-für-Schritt-Format für ein 90-minütiges Review und eine Hilfe zur Wahl zwischen den Techniken.

Abschnitt 11.3 macht aus der einmaligen Bewertung einen dauerhaften Schutz. Eine Fitness Function ist eine objektive Prüfung eines Architekturmerkmals, idealerweise automatisiert: eine Abhängigkeitsregel, ein Lasttest mit Perzentil-Schwellenwerten, ein Schwachstellenscan, ein Chaos-Experiment oder eine Prüfung der Schemakompatibilität. Zusammen mit kleinen, häufigen und sicheren Änderungen ermöglichen Fitness Functions eine evolutionäre Architektur. Drei Szenarien aus Kapitel 2 werden zu funktionierenden Prüfungen, mit Code.

Steuern, organisieren und modernisieren: Abschnitte 11.4 bis 11.6

Abschnitt 11.4 fragt, wie eine Organisation mit vielen autonomen Teams ihre Architektur stimmig hält. Er unterscheidet Prinzipien, Standards, Guardrails und Gates; zeigt, wie man Prinzipien mit Name, Aussage, Begründung und Auswirkungen formuliert; und beschreibt dezentrale Entscheidungsfindung über Andrew Harmel-Laws Advice Process, Architektur-Beratungsforen und Technologieradare, wobei die Einhaltung über Fitness Functions und Policy as Code automatisiert wird.

Abschnitt 11.5 nimmt das Gesetz von Conway als Entwurfswerkzeug ernst. Er stellt Team Topologies von Matthew Skelton und Manuel Pais vor: vier Teamtypen, drei Interaktionsmodi, kognitive Last des Teams, Team-APIs und Bruchlinien, und zeigt, wie man die Teams der Essenslieferplattform entlang von Änderungsströmen neu gestaltet. Abschnitt 11.6 befasst sich mit den Systemen, die bereits existieren: warum Big-Bang-Neuentwicklungen so oft scheitern, welche Migrationsstrategien es gibt und welche schrittweisen Techniken funktionieren, darunter Strangler Fig, Branch by Abstraction, Parallelbetrieb, Change Data Capture und Anti-Corruption Layer, durchgespielt am Herauslösen eines Bestellmoduls aus einem Monolithen.

Führen, begründen und verantwortungsvoll handeln: Abschnitte 11.7 und 11.8

Abschnitt 11.7 behandelt die menschliche Seite der Rolle: führen ohne formale Autorität, Vertrauen aufbauen, Entscheidungen per Konsent oder Konsens moderieren, mit Konflikten und Organisationspolitik umgehen, mit der Geschäftsleitung kommunizieren, Mentoring, Staff-plus-Laufbahnen und die Entscheidungsfallen, in die auch Erfahrene tappen, mit einer umstrittenen Event-Bus-Entscheidung, die Schritt für Schritt bewältigt wird.

Abschnitt 11.8 schließt das Kapitel mit den beiden Argumenten ab, die ein Architekt am dringendsten gut vorbringen muss. Der Business Case: Kosten, Nutzen, Risiken und Optionen, mit Kapitalwert und Amortisationszeit in einem ausgearbeiteten Beispiel berechnet, und technische Schulden in Geld und Zeit ausgedrückt. Und die ethische Begründung: Berufskodizes, vorhersehbare Schäden wie Verlust der Privatsphäre, Verzerrungen (Bias), fehlende Barrierefreiheit und manipulatives Design sowie Nachhaltigkeit, gemessen mit der Spezifikation Software Carbon Intensity.

Der eine Gedanke, den Sie mitnehmen sollten

Architektur ist nicht fertig, wenn das Diagramm gezeichnet ist. Ihre Entscheidungen müssen an dem geprüft werden, was das Unternehmen braucht, geschützt werden, während sich das System verändert, von den richtigen Personen mit dem richtigen Rat getroffen, von Teams getragen werden, deren Struktur zu ihnen passt, und in einer Weise begründet werden, die

Auftraggeber und Gesellschaft akzeptieren können. Die wirksamsten Architekten machen diese Arbeit leicht und kontinuierlich statt schwer und gelegentlich: kurze Risiko-Reviews statt jährlicher Audits, automatisierte Guardrails statt Genehmigungswarteschlangen, Rat statt Erlaubnis, schrittweise Migration statt Neuentwicklung. Die Disziplin, die all das zusammenhält, ist die, mit der dieser Kurs begonnen hat: die benötigte Qualität benennen, sie messbar machen, bewusst entscheiden, das Warum aufschreiben und weiter prüfen.

Architekturbewertung und ATAM

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.6 Entscheidungen und Abwägungen
- 2.3 Qualitätsszenarien
- 2.8 Erheben und priorisieren

Begriffe von früher, die er verwendet: Qualitätsszenario, Utility Tree, Qualitätsmerkmal, Stakeholder

DIE IDEE IN KÜRZE

Bevor eine Stadt eine neue Brücke baut, prüfen unabhängige Ingenieure den Entwurf: Sie fragen, was bei Hochwasser, unter den schwersten Lkw und bei Sturm geschieht, und sie zeigen, wo eine Entscheidung, die Geld spart, eine andere Gefahr verschärft. Sie entwerfen die Brücke nicht neu und geben ihr auch keine Note von eins bis zehn; sie sagen den Eigentümern, wo die echten Risiken liegen, damit diese entscheiden können, was zu tun ist, solange Änderungen noch günstig sind. Die Architekturbewertung leistet dasselbe für Software. Ihre bekannteste Methode, ATAM, bringt Eigentümer, Entwickler und Nutzer des Systems zusammen, um den Entwurf an den Situationen zu prüfen, auf die es am meisten ankommt.

WAS SIE WISSEN MÜSSEN

- Die Architekturbewertung prüft, ob die wesentlichen Entscheidungen eines Systems es ihm ermöglichen, seine Geschäftsziele und Qualitätsanforderungen zu erfüllen, und ist vor größeren oder unumkehrbaren Festlegungen am wertvollsten.
- ATAM (Kazman, Klein und Clements, SEI, 2000) umfasst neun Schritte über Vorbereitung, zwei Bewertungssitzungen und Nachbereitung, mit einem unabhängigen Bewertungsteam, den Entscheidungsträgern des Projekts und einem größeren Kreis von Stakeholdern.
- Ihre Ergebnisse sind Risiken, Nicht-Risiken (mit ihren Annahmen), Sensitivitätspunkte, Trade-off-Punkte und Risikothemen, die mit den geschäftlichen Treibern verknüpft sind, die sie gefährden; eine Bestehensgrenze vergibt sie nicht.

- Der Utility Tree, den die Entscheidungsträger in Schritt 5 aufbauen, wird mit den Szenarien verglichen, die die Stakeholder in Schritt 7 sammeln und bewerten; Lücken zwischen beiden decken blinde Flecken auf.
- CBAM baut auf ATAM auf: Es gewichtet den Nutzen, den jede Architekturstrategie priorisierten Szenarien hinzufügt, und teilt ihn durch ihre Kosten, um das Kosten-Nutzen-Verhältnis zu vergleichen, wobei Randbedingungen das Verhältnis weiterhin übersteuern können.
- Eine vollständige ATAM kostet Dutzende Personentage, daher sollte die Methode an der Tragweite ausgerichtet werden, und ihre Ergebnisse müssen zu Entscheidungsdokumenten, Backlog-Einträgen und Fitness Functions mit Verantwortlichen werden.

Warum eine Architektur bewerten

Jede Architektur ist eine Wette. Wenn ein Team einen Event Broker, eine Datenbank pro Service oder eine einzige Cloud-Region wählt, sagt es voraus, dass diese Entscheidungen die Performance, Verfügbarkeit, Sicherheit und Kosten liefern, die das Unternehmen braucht. Abschnitt 1.3 hat gezeigt, warum solche Wetten zählen: Die Kosten, eine strukturelle Entscheidung zu ändern, wachsen mit allem, was darauf aufgebaut ist, und einige der teuersten Fehlschläge der Softwaregeschichte waren Fehler der Anordnung, nicht einzelner Codezeilen. Die Architekturbewertung ist die bewusste Prüfung dieser Wette, bevor oder während das Geld ausgegeben wird.

Eine Bewertung stellt eine enge, praktische Frage: Welche Architekturentscheidungen gefährden, gemessen an dem, was dieses System leisten muss, diese Ziele? Sie ist weder ein Code-Review noch ein Schönheitswettbewerb zwischen Stilen noch ein Audit der Dokumentationsqualität. Sie arbeitet mit Entscheidungen und deren Folgen für Qualitätsmerkmale und kann daher lange vor dem ersten Code stattfinden, an einem Whiteboard-Entwurf oder an einer Reihe von Entscheidungsdokumenten, und sie lässt sich bei einem System wiederholen, das seit Jahren läuft.

Eine Bewertung hat auch Nutzen über die Ergebnisse hinaus. Stakeholder, die sich selten begegnen (Product Owner, Betrieb, Sicherheit, Finanzen, Entwicklung), müssen sich einigen, was am wichtigsten ist, und das legt oft widersprüchliche Annahmen offen. Der Architekt muss den Entwurf so klar erklären, dass Außenstehende ihn hinterfragen können, was die Dokumentation verbessert. Und die Organisation erhält eine Aufzeichnung bekannter Risiken, die sie bewusst akzeptiert hat, was etwas ganz anderes ist als Risiken, die niemand bemerkt hat.

Wann bewerten

Der beste Zeitpunkt für eine Bewertung ist, wenn genug entschieden ist, damit es etwas zu prüfen gibt, aber bevor diese Entscheidungen teuer rückgängig zu machen sind. In der Praxis gibt es mehrere natürliche Zeitpunkte, und reife Organisationen nutzen für jeden einen anderen Umfang der Bewertung.

Zeitpunkte für eine Bewertung

Auslöser	Was geprüft wird	Typischer Umfang
Eine Kandidatenarchitektur liegt vor, vor dem Bau im großen Maßstab	Ob die wichtigsten Stile und Taktiken die priorisierten Qualitätsszenarien erfüllen können	Vollständige oder reduzierte ATAM bei großen oder langlebigen Systemen; sonst ein leichtgewichtiges Review
Vor einer großen Festlegung	Ein Anbietervertrag, eine Cloud-Migration, eine Finanzierungsentscheidung oder ein Plattformwechsel, die teuer rückgängig zu machen wären	Formal, oft mit externen Bewertern, weil die Entscheidung eine Einwegtür ist
Deutliche Änderung der geschäftlichen Treiber	Neue Märkte, eine Fusion, ein zehnfaches Wachstumsziel, neue Regulierung	Den Utility Tree neu durchgehen und die geänderten Szenarien analysieren
Angesammelte Änderungen und Drift	Ob das gebaute System noch der angestrebten Architektur und ihren Gründen entspricht	Regelmäßiges leichtgewichtiges Review plus automatisierte Fitness Functions
Übernahme oder Due Diligence	Ob ein bestehendes System die Pläne des Käufers tragen kann	Zeitlich begrenzte Bewertung eines bestehenden Systems, oft kombiniert mit Codeanalyse
Nach einem schweren Vorfall	Ob der Vorfall eine architektonische Schwäche offengelegt hat und nicht nur einen einmaligen Fehler	Gezielte Analyse der betroffenen Szenarien

Zu frühes Bewerten verschwendet Aufwand an Entscheidungen, die noch nicht real sind; zu spätes Bewerten macht aus Ergebnissen bloßes Bedauern. Eine nützliche Regel aus Fairbanks' risikogetriebenem Modell (Abschnitt 1.8) gilt auch hier: Investieren Sie in die Bewertung im Verhältnis zum Risiko. Ein internes Werkzeug zweier Teams braucht keinen zweitägigen Workshop; eine nationale Zahlungsplattform schon.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist: was eine Bewertung als Eingabe braucht

Eine Bewertung ist nur so gut wie das, was man ihr gibt. Mindestens braucht sie die Geschäftsziele, eine kurze Liste priorisierter Qualitätsszenarien (Abschnitt 2.3) und eine Beschreibung der Architektur: ein Kontext- und ein Containerdiagramm (Abschnitt 4.3), die wichtigsten Laufzeitabläufe und die wesentlichen Entscheidungen. Fehlen diese, ist ihre Erstellung der erste und oft wertvollste Teil der Bewertung.

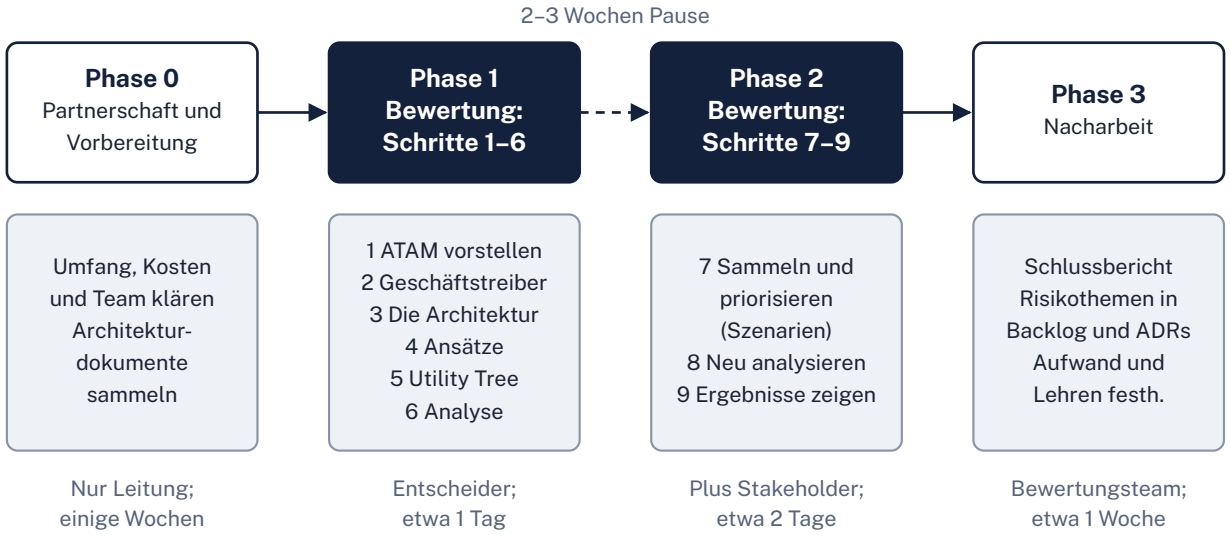
ATAM: Ursprung, Zweck und Beteiligte

Die Architecture Tradeoff Analysis Method (ATAM) ist die bekannteste Bewertungsmethode. Rick Kazman, Mark Klein und Paul Clements vom Software Engineering Institute (SEI) der Carnegie Mellon University beschrieben sie im technischen Bericht *ATAM: Method for Architecture Evaluation* (CMU/SEI-2000-TR-004, August 2000). Sie ging aus der früheren Software Architecture Analysis Method (SAAM) hervor, die Kazman und Kollegen 1994 veröffentlichten und die sich vor allem auf Änderbarkeit konzentrierte. ATAM wird auch in *Software Architecture in Practice* von Bass, Clements und Kazman beschrieben, inzwischen in der vierten Auflage (2021).

Ihr Zweck wird oft missverstanden. ATAM vergibt keine Punktzahl und keine Bestehensgrenze für eine Architektur, und sie beweist nicht, dass ein System seine Anforderungen erfüllen wird. Sie zeigt, wo Architekturentscheidungen die Geschäftsziele gefährden und wo sie Abwägungen zwischen Qualitäten erzwingen, damit die Verantwortlichen entscheiden können, was zu tun ist. Der Name der Methode betont Abwägungen, weil die meisten interessanten Risiken dort liegen, wo eine Qualität auf Kosten einer anderen erkauft wird.

Drei Gruppen nehmen teil. Das **Bewertungsteam**, meist drei bis fünf Personen von außerhalb des Projekts, führt die Methode durch: eine Teamleitung, eine Bewertungsleitung, die moderiert, Protokollführende, die Szenarien und Abläufe festhalten, und Fragende mit Fachwissen zu bestimmten Qualitäten. Die **Entscheidungsträger des Projekts** sind der Architekt, die Projekt- oder Produktleitung und der Auftraggeber oder Kunde, der auf die Ergebnisse reagieren kann. Die **Architektur-Stakeholder** sind alle anderen Beteiligten: Entwicklung, Test, Betrieb, Sicherheit, Integration, Support und Vertretungen der Nutzer. Unabhängigkeit zählt: Bewerter, die das System entworfen haben oder dem Architekten unterstellt sind, finden weniger Risiken.

Die Phasen von ATAM



Zwischen den beiden Bewertungssitzungen liegt eine Pause, in der das Team die ersten Ergebnisse verarbeitet und der größere Kreis der Stakeholder zusammengestellt wird.

Die neun Schritte

1

1. ATAM vorstellen

Die Bewertungsleitung erklärt die Methode, die Rollen der Beteiligten und die Ergebnisse, damit niemand die Sitzung für ein Audit oder ein Design-Review seiner persönlichen Arbeit hält.

2

2. Die geschäftlichen Treiber vorstellen

Eine leitende Vertretung des Geschäfts erläutert in geschäftlichen Begriffen die Ziele des Systems, die Randbedingungen, die Stakeholder und die wichtigsten Qualitäten. Diese geschäftlichen Treiber

sind der Maßstab für alles Weitere.

3

3. Die Architektur vorstellen

Der Architekt beschreibt die Architektur auf angemessener Ebene: Kontext, Hauptstrukturen, Laufzeitverhalten, Deployment und wie sie die Treiber adressiert. Ein typisches Zeitlimit von etwa einer Stunde erzwingt Fokus.

4

4. Architekturansätze identifizieren

Der Architekt benennt die verwendeten Architekturansätze

(Stile, Muster und Taktiken wie ein Broker, Replikation oder Caching), und das Bewertungsteam katalogisiert sie, ohne sie schon zu analysieren.

5 **5. Den Utility Tree der Qualitätsmerkmale erstellen**
Die Entscheidungsträger erstellen oder verfeinern einen Utility Tree und bewerten jedes Blattszenario nach geschäftlicher Bedeutung und technischer Schwierigkeit.

6 **6. Die Architekturansätze analysieren**
Für die Szenarien mit der höchsten Priorität fragen die Bewerter, welche Ansätze jedes davon adressieren, und hinterfragen sie mit qualitätsspezifischen Fragen; dabei halten sie Risiken, Nicht-Risiken, Sensitivitätspunkte und Trade-off-Punkte fest.

7 **7. Szenarien sammeln und priorisieren**
In Phase 2 schlägt der größere Kreis der Stakeholder eigene Szenarien vor und stimmt über sie ab. Das Ergebnis wird mit dem Utility Tree verglichen; wichtige Szenarien, die im Baum fehlen, sind bereits für sich ein Befund.

8 **8. Die Architekturansätze erneut analysieren**
Die am höchsten bewerteten neuen Szenarien werden wie in Schritt 6 analysiert, nun mit dem Wissen der anwesenden Stakeholder.

9 **9. Die Ergebnisse vorstellen**
Das Bewertungsteam stellt Ansätze, Szenarien, Utility Tree, Risiken, Nicht-Risiken, Sensitivitätspunkte, Trade-off-Punkte und Risikothemen vor und setzt jedes Thema in Bezug zu den geschäftlichen Treibern, die es gefährdet.

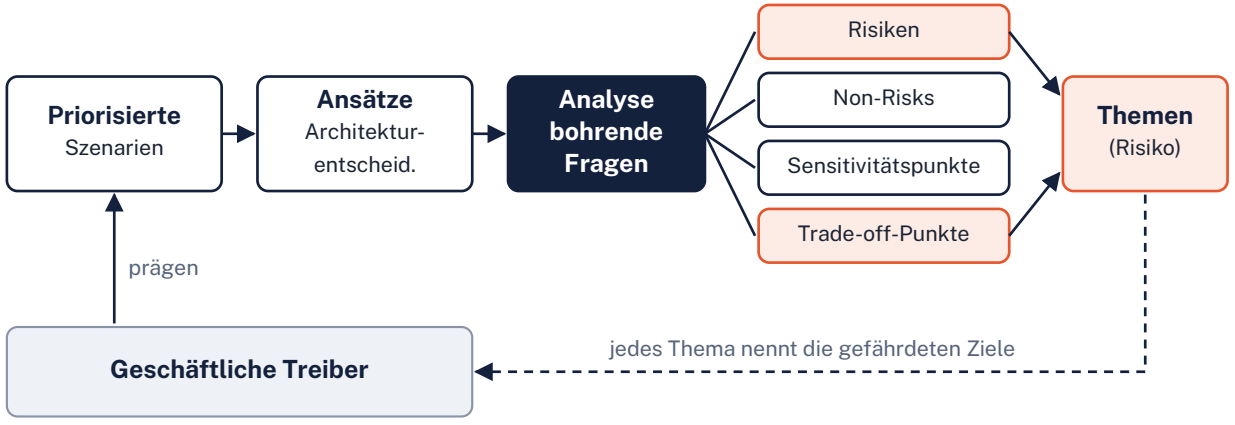
Das Vokabular der Analyse

Das Herzstück der Methode sind die Schritte 6 und 8. Für jedes wichtige Szenario bitten die Bewerter den Architekten, durchzuspielen, wie die Architektur reagiert, und sie bohren mit Fragen nach, die aus jeder Qualität stammen: Wie viele synchrone Aufrufe liegen auf diesem Pfad, was passiert mit laufenden Anfragen, wenn dieser Knoten ausfällt, wo ist diese Zugangsberechtigung gespeichert, was begrenzt die Queue? Jede Antwort wird mit fünf Begriffen eingeordnet, die weit über ATAM hinaus zum Standardvokabular geworden sind.

Die Ergebnisse der Analyse

Ergebnis	Bedeutung	Beispiel einer Essenslieferplattform
Risiko	Eine Entscheidung oder eine fehlende Entscheidung, die verhindern kann, dass ein wichtiges Szenario oder Geschäftsziel erreicht wird	Die Kartenautorisierung liegt im synchronen Bestellpfad, daher verlangsamt ein langsamer Zahlungsanbieter jede Bestellung
Nicht-Risiko	Eine Entscheidung, die für das Szenario als tragfähig beurteilt wird, festgehalten mit den Annahmen, von denen sie abhängt	Transactional Outbox und idempotente Consumer erfüllen das Szenario „keine verlorenen Bestellungen“, sofern der Broker jede Partition dreifach repliziert
Sensitivitätspunkt	Eine Eigenschaft von Komponenten oder Verbindungen, die für eine Qualitätsantwort entscheidend ist	Die Bestelllatenz reagiert empfindlich auf die Zahl synchroner Hops und auf die p99-Antwortzeit des Anbieters
Trade-off-Punkt	Eine Eigenschaft, die für mehr als eine Qualität ein Sensitivitätspunkt ist, sodass die Verbesserung der einen die andere verschlechtert	Das Intervall der Kurier-Standortmeldungen: Kürzer verbessert die Aktualität der Karte, leert aber die Handyakkus und erhöht mobile Daten- und Erfassungskosten
Risikothema	Eine Gruppe von Risiken mit einer gemeinsamen Ursache, verknüpft mit den geschäftlichen Treibern, die sie gefährdet	Drittanbieterdienste liegen auf synchronen kritischen Pfaden und gefährden das Ziel, das Bestellvolumen ohne Konversionsverluste zu verdoppeln

Wie die Analyse ihre Ergebnisse erzeugt



Die Analyse setzt priorisierte Szenarien in Bezug zu den Entscheidungen, die sie adressieren; die daraus entstehenden Risiken werden zu Themen gruppiert, die auf die gefährdeten Geschäftsziele zurückverweisen.

Nicht-Risiken verdienen mehr Aufmerksamkeit, als sie gewöhnlich bekommen. Festzuhalten, dass eine Entscheidung *wegen* einer bestimmten Annahme tragfähig ist (drei Replikate, eine vertraglich zugesicherte Antwortzeit eines Lieferanten, Traffic unterhalb einer bestimmten Grenze), macht aus einer vagen Beruhigung etwas, das sich später prüfen lässt. Ändert sich die Annahme, wird das Nicht-Risiko zum Risikokandidaten. Sensitivitäts- und Trade-off-Punkte sind nach der Bewertung ähnlich wertvoll: Sie sagen künftigen Maintainern, welche Parameter sie nur mit Vorsicht ändern sollten, und sie sind natürliche Orte für die automatisierten Prüfungen aus Abschnitt 11.3.

Der Utility Tree noch einmal

Abschnitt 2.8 hat den Utility Tree als Werkzeug zur Priorisierung von Qualitätsanforderungen eingeführt. Er entstand in ATAM und spielt innerhalb der Methode zwei unterschiedliche Rollen. In Schritt 5 bauen die Entscheidungsträger den Baum von oben nach unten auf: Nutzen, dann Qualitätsmerkmale, dann Verfeinerungen, dann konkrete Szenarien mit Bewertung (geschäftliche Bedeutung, technische Schwierigkeit). Die (H,H)-Blätter werden zur Tagesordnung für Schritt 6, weil sie am wichtigsten und am schwersten zu erreichen sind; dort verstecken sich Risiken am ehesten.

In Schritt 7 sammelt der größere Kreis der Stakeholder Szenarien von unten nach oben und stimmt über sie ab, ähnlich wie in einem Quality Attribute Workshop. Der Vergleich beider Sichten ist einer der aufschlussreichsten Momente einer Bewertung. Tauchen die wichtigsten Szenarien der Stakeholder bereits im Baum auf, verstehen die Entscheidungsträger den Zweck ihres Systems. Fehlt ein Szenario, das viele Stimmen erhält, etwa ein Betriebsszenario über Restaurant-Tablets, die offline gehen, haben die Entscheidungsträger einen blinden Fleck, und die Architektur womöglich auch. Solche Szenarien werden in den Baum aufgenommen und in Schritt 8 analysiert.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine verdichtete ATAM der Essenslieferplattform

Die Plattform aus den Kapiteln 2 und 4 verbindet Kunden, 4000 Restaurants und 9000 Kuriere. Die geschäftlichen Treiber sind, das Bestellvolumen innerhalb von 18 Monaten zu verdoppeln (von 50 auf 100 Bestellungen pro Sekunde zur Freitagsspitze), ohne Konversionen zu verlieren, und nationale Restaurantketten zu gewinnen, die vertraglich zugesicherte Verfügbarkeit verlangen. Die Architektur: Microservices pro Bounded Context (Bestellungen, Restaurant, Disposition, Benachrichtigung), jeweils mit eigener PostgreSQL-Datenbank; ein API-Gateway; synchrones REST für die Bestellaufgabe, einschließlich der Kartenautorisierung bei einem externen Zahlungsanbieter; ein Kafka-Broker mit Transactional Outbox für Bestellereignisse; Kurierpositionen, die alle 4 Sekunden gesendet und in Redis gehalten werden; eine Cloud-Region über drei Availability Zones, mit nächtlichen Backups, die in eine zweite Region kopiert werden. Der Utility Tree der Phase 1 übernahm die sechs Szenarien aus Abschnitt 2.8.

Bestelllatenz (H,H)	p95 unter 1,5 s und p99 unter 3 s bei 100 Bestellungen/s. Ansätze: Gateway, vier synchrone Hops, Kartenautorisierung in der Anfrage. Gemessene Latenz des Anbieters: p95: 900 ms, p99: 2,5 s
Analyse	Allein der Anbieter verbraucht den größten Teil des p99-Budgets, sodass 500 ms für vier Hops und Datenbank-Schreibvorgänge bleiben. Littles Gesetz verwendet die mittlere Zeit, nicht p99 (Abschnitt 2.4), aber das Risiko ist klar: Treibt eine Verlangsamung beim Anbieter den Mittelwert auf 2,5 s, bedeuten 100 Bestellungen/s 250 gleichzeitig laufende Anfragen, und der Pool von 200 Workern des Bestellservice ist gesättigt. **Risiko R1**; **Sensitivitätspunkt S1**: Zahl der synchronen Hops und p99 des Anbieters
Verfügbarkeit des Checkouts (H,H)	Zahlungsanbieter läuft zur Spitzenzeit in Timeouts: Bestellungen werden angenommen und in die Queue gestellt, keine geht verloren, Kunde wird innerhalb von 10 s informiert. Der aktuelle Entwurf lässt die Bestellung scheitern, wenn die Autorisierung in ein Timeout läuft. **Risiko R2**
Vorgeschlagene Option	Asynchron autorisieren: die Bestellung als ausstehend annehmen, aus einer Queue autorisieren. Verbessert Latenz und Verfügbarkeit, aber manche angenommenen Bestellungen scheitern später bei der Zahlung, und Restaurants dürfen keine unbezahlten Bestellungen zubereiten. **Trade-off-Punkt T1**: wo im Bestellablauf die Zahlung bestätigt wird
Keine verlorenen Bestellungen (Teil von H,H)	Outbox in derselben lokalen Transaktion wie die Bestellung, Kafka mit drei Replikaten, idempotente Consumer. **Nicht-Risiko N1**, sofern Replikationsfaktor 3 und Producer-Bestätigungen von allen synchronen Replikaten (In-Sync Replicas) gelten

Ausfall einer Region (H,M)	RTO 15 min, RPO 1 min. Nächtliche Backups ergeben ein RPO von bis zu 24 Stunden und eine Wiederherstellung, die in Stunden gemessen wird. **Risiko R3**
Aktualität der Verfolgung (M,M)	Kartenposition nie älter als 5 s. 9000 Kuriere alle 4 s ergeben 2250 Schreibvorgänge/s, für Redis problemlos. **Trade-off-Punkt T2**: das Aktualisierungsintervall (Aktualität gegen Akku, Daten und Kosten)
Kontoübernahme (H,M)	Rate Limiting nach IP-Adresse am Gateway erkennt kein Credential Stuffing, das über Tausende von Adressen verteilt ist. **Risiko R4**
Brainstorming in Phase 2	Stakeholder aus Restaurants und Betrieb stimmten stark für ein Szenario, das im Baum fehlte: Ein Restaurant-Tablet, das zur Spitzenzeit 10 Minuten offline ist, muss beim erneuten Verbinden jede Bestellung erhalten. Push-Benachrichtigungen werden einmal und ohne Bestätigung gesendet. **Risiko R5** und ein blinder Fleck im Baum
Risikothemen	**Thema A** (R1, R2, R4): Abhängigkeiten von Drittanbietern und am Rand des Systems liegen ohne Isolation auf synchronen kritischen Pfaden; gefährdet die Verdopplung des Volumens ohne Konversionsverluste. **Thema B** (R3, R5): Wiederherstellungs- und Zustellgarantien hinken der Verfügbarkeit hinterher, die Restaurantketten versprochen wird; gefährdet die Verträge mit den Ketten
Was danach geschah	Entscheidungsdokumente zur asynchronen Autorisierung mit einem Status „ausstehend“ (T1 wird akzeptiert, Restaurants werden erst nach der Autorisierung benachrichtigt), zu einer Warm-Standby-Region, zur bestätigten Zustellung von Bestellungen an Tablets und zur Erkennung von Credential Stuffing beim Identity Provider. Die Annahmen von N1 wurden zu einer automatisierten Konfigurationsprüfung

Die Ergebnisse betreffen Entscheidungen, nicht Personen oder Codequalität, und jedes lässt sich auf ein Szenario und einen geschäftlichen Treiber zurückführen.

Zwei der fünf Risiken stammten aus Szenarien, die der Architekt nicht priorisiert oder nicht bedacht hatte; genau dafür gibt es den Schritt mit den Stakeholdern.

Trade-off-Punkte sind keine Probleme, die beseitigt werden müssen: T1 wurde bewusst akzeptiert und dokumentiert, mit seinen Folgen für Restaurants und Kunden.

CBAM: Geld auf den Tisch legen

ATAM sagt Ihnen, wo die Risiken liegen; sie sagt Ihnen nicht, welche Abhilfen ihr Geld wert sind. Die Cost Benefit Analysis Method (CBAM), am SEI von Rick Kazman, Jai Asundi und Mark Klein entwickelt (erstmals 2001 veröffentlicht), baut auf den Szenarien einer ATAM auf, um diese wirtschaftliche Begründung zu liefern.

In groben Zügen nehmen die Stakeholder die priorisierten Szenarien und ordnen für jedes mehreren Antwortniveaus einen Nutzen (auf einer Skala von 0 bis 100) zu: schlechtester Fall, aktueller Stand, Ziel und bester Fall. Die Architekten schlagen Architekturstrategien vor und schätzen die Antwort, die jede erreichen würde. Der Nutzengewinn jeder Strategie wird an der Nutzen-Antwort-Kurve des Szenarios abgelesen und mit der Priorität des Szenarios gewichtet; über alle Szenarien summiert ergibt das den Nutzen der Strategie. Nutzen geteilt durch Kosten ergibt ein Kosten-Nutzen-Verhältnis (Value for Cost, VFC), und die Strategien werden im Rahmen des verfügbaren Budgets in der Reihenfolge ihres VFC gewählt und dann gegen Intuition und Randbedingungen auf Plausibilität geprüft.

Eine Mini-CBAM für das Risiko eines Regionsausfalls

Strategie	Erwartete Antwort	Nutzengewinn x Gewicht	Jährliche Kosten	Kosten-Nutzen-Verhältnis
Aktuell: nächtliche Backups	RTO etwa 6 h, RPO bis zu 24 h	Nutzen 10 (Basiswert)	bereits bezahlt	n/a
A: Warm-Standby-Region, asynchrone Datenbankreplikation	RTO 10 min, RPO unter 1 min	(90 - 10) x 0.3 = 24	£180k	24 / 180 = 0.13
B: Pilot Light (Daten repliziert, Rechenleistung bei Bedarf gestartet)	RTO 40 min, RPO unter 1 min	(60 - 10) x 0.3 = 15	£70k	15 / 70 = 0.21

Strategie B bringt mehr Nutzen pro Pfund, verfehlt aber die Wiederherstellungszeit von 15 Minuten, die die Restaurantverträge verlangen. Die Zahlen informieren die Entscheidung; sie treffen sie nicht. Der eigentliche Wert von CBAM liegt darin, dass es explizite, vergleichbare Schätzungen von Nutzen und Kosten erzwingt und offenlegt, wann eine Randbedingung und nicht ein Verhältnis entscheiden sollte. Seine Schwäche ist, dass Nutzen und Kosten Einschätzungen sind; Teams sollten sie variieren und prüfen, ob sich die Rangfolge ändert, genau wie Abschnitt 1.6 es für gewichtete Entscheidungsmatrizen empfiehlt. Abschnitt 11.8 zeigt, wie man aus solchen Schätzungen einen Business Case macht.

Kosten und Praxis

Eine vollständige ATAM kostet viel Arbeitszeit. Als grobe Veranschaulichung: Ein vierköpfiges Bewertungsteam, das jeweils zwei Tage in die Vorbereitung steckt (8 Personentage), eine eintägige Phase 1 mit drei Entscheidungsträgern (7), eine zweitägige Phase 2 mit zehn zusätzlichen Stakeholdern (34) und eine Woche Nacharbeit durch zwei Personen (10) ergeben

rund 60 Personentage, bevor irgendetwas behoben ist. Für ein System, dessen Ausfall Millionen kosten würde oder das ein Jahrzehnt laufen soll, ist das eine günstige Versicherung. Für ein kleines Produktteam ist es unverhältnismäßig.

Mehrere Praktiken halten Bewertungen lohnend. Skalieren Sie die Methode: Bass, Clements und Kazman beschreiben eine leichtgewichtige Architekturbewertung, eine reduzierte ATAM für interne Teams, die sich in einem Tag oder weniger durchführen lässt, und manche Organisationen führen nur die Schritte 2 bis 6 mit einer Handvoll Personen durch. Bereiten Sie gründlich vor: Der größte Teil der Kosten einer schlechten Bewertung entsteht dadurch, dass man erst im Raum feststellt, dass niemand die Architektur beschreiben kann. Halten Sie die Bewerter unabhängig, aber informiert. Vereinbaren Sie vorab, wem die Ergebnisse gehören und wie sie weiterverfolgt werden, idealerweise als Entscheidungsdokumente und Backlog-Einträge mit Verantwortlichen. Und wiederholen Sie die Bewertung: Die Ergebnisse sind eine Momentaufnahme, und die Risiken ändern sich mit dem System und dem Geschäft.

Fachleute sind sich uneinig, welchen Platz die Methode heute hat. Kritiker sehen in ihr ein schwergewichtiges Relikt großer, vorab geplanter Projekte, das schlecht zu Teams passt, die viele Male am Tag deployen. Befürworter entgegnen, dass die Konzepte (Szenarien, Sensitivitäts- und Trade-off-Punkte, Risikothemen) genau das sind, was agilen Teams fehlt, wenn sie die Bewertung auslassen, und dass sich die Methode verkleinern lässt. Beide haben einen Punkt: Das volle Zeremoniell passt zu Systemen mit hoher Tragweite und langer Lebensdauer und zu großen Festlegungen, während das Vokabular und die Gewohnheit szenariogetriebenen Fragens überall hingehören. Abschnitt 11.2 behandelt das leichtere Ende.

ACHTUNG

Häufige Fehler bei Architekturbewertungen

- Die Bewertung als Prüfung mit Bestehen oder Durchfallen behandeln, sodass der Architekt verteidigt statt erkundet und Risiken verborgen bleiben.
- Ohne geschäftliche Treiber bewerten, was eine Liste technischer Sorgen ohne jede Möglichkeit zur Rangfolge ergibt.
- Die Analyse in Codestil, Werkzeugvorlieben oder persönlichen Geschmack abschweifen lassen statt szenariogetriebene Fragen zu stellen.
- Nicht-Risiken ohne ihre Annahmen festhalten, sodass niemand bemerkt, wenn eine Annahme nicht mehr gilt.
- Eine lange Risikoliste ohne Risikothemen erstellen, sodass das Management die wenigen zugrunde liegenden Ursachen nicht erkennen kann.
- Die Bewertung erst nach Vertragsunterzeichnung oder Plattformkauf durchführen, wenn man die Ergebnisse nur noch bedauern kann.
- Keine Nachverfolgung: Ergebnisse, die nicht zu Entscheidungsdokumenten, Backlog-Einträgen und Verantwortlichen werden, sind innerhalb eines Quartals vergessen.

Häufige Fragen

Lässt sich ATAM auf ein bereits bestehendes System anwenden?
Ja. Die Schritte sind dieselben; die Architektur wird aus Dokumentation, Code und den Menschen, die das System betreiben, rekonstruiert, und Messungen am laufenden System ersetzen Schätzungen. Ein bestehendes System zu bewerten ist vor einer großen Erweiterung, einer Migration oder einer Übernahme üblich.

Bedeutet eine ATAM ohne Befund, dass das System seine Anforderungen erfüllen wird?
Nein. ATAM untersucht eine Stichprobe von Szenarien auf der Ebene von Designentscheidungen. Sie verringert die Wahrscheinlichkeit unangenehmer Überraschungen, kann aber Messung, Tests und Betriebserfahrung nicht ersetzen. Ihre Ergebnisse sollten nach Möglichkeit in Tests und Fitness Functions überführt werden.

Wer sollte die Architektur vorstellen?
Der Architekt oder Lead Engineer, der sie am besten kennt. Aufgabe des Bewertungsteams ist es, Fragen zu stellen, nicht vorzutragen, und Aufgabe des Architekten ist es, zu erklären und zu erkunden, nicht zu verteidigen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Was ist der Unterschied zwischen einem Sensitivitätspunkt und einem Trade-off-Punkt?

Ein Sensitivitätspunkt ist eine Eigenschaft, die für eine Qualitätsantwort entscheidend ist. Ein Trade-off-Punkt ist eine Eigenschaft, die zugleich für mehrere Qualitäten ein Sensitivitätspunkt ist, sodass ihre Änderung der einen hilft und der anderen schadet.

▼ Warum hält ATAM Nicht-Risiken fest, und was muss mit ihnen festgehalten werden?

Um zu zeigen, welche Entscheidungen untersucht und als tragfähig beurteilt wurden. Auch die Annahmen, von denen sie abhängen, müssen festgehalten werden, damit das Nicht-Risiko erneut geprüft werden kann, wenn sich eine Annahme ändert.

▼ Was bedeutet es, wenn Stakeholder in Phase 2 stark für ein Szenario stimmen, das im Utility Tree fehlt?

Die Entscheidungsträger haben einen blinden Fleck in Bezug auf das, was zählt, und die Architektur adressiert dieses Szenario womöglich nicht. Es wird in den Baum aufgenommen und in Schritt 8 analysiert.

▼ Wie wird in CBAM das Kosten-Nutzen-Verhältnis einer Strategie berechnet?

Ihr Nutzen, also der nach Priorität gewichtete Nutzengewinn über die betroffenen Szenarien, wird durch ihre Kosten geteilt.

▼ Probieren Sie es aus: Ordnen Sie diese ATAM-Ergebnisse ein: Der Bestelldurchsatz hängt direkt von der Anzahl der Broker-Partitionen ab; Verschlüsselung auf einer Verbindung verbessert die Sicherheit, erhöht aber die Latenz; ein einziger Zahlungsanbieter ohne Ausweichlösung auf dem Bestellpfad; eine ausgefallene Zone wird mit einem getesteten Failover abgefangen.

Die Partitionsanzahl ist ein Sensitivitätspunkt: Eine Eigenschaft beeinflusst eine Qualität stark. Die Verschlüsselung ist ein Trade-off-Punkt, weil dieselbe Entscheidung der Sicherheit hilft und der Performance schadet. Der einzelne Zahlungsanbieter ist ein Risiko, das den Geschäftstreiber Bestellverfügbarkeit bedroht. Der getestete Failover ist ein Nicht-Risiko, unter genannten Annahmen wie genügend Kapazität in den anderen Zonen. Gruppieren Sie die Risiken in Themen, und machen Sie dann aus jedem Befund ein Entscheidungsdokument oder einen Backlog-Eintrag mit einer verantwortlichen Person.

QUELLEN

- SEI: ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004): https://www.sei.cmu.edu/library/atam-method-for-architecture-evaluation/

- SEI: Making Architecture Design Decisions: An Economic Approach (the CBAM): https://www.sei.cmu.edu/library/making-architecture-design-decisions-an-economic-approach

Leichtgewichtige Reviews und Risk-Storming

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 4.3 Das C4-Modell
- 4.8 Kommunizieren
- 11.1 Bewertung mit ATAM

Begriffe von früher, die er verwendet: Risiko (in der Architekturbewertung), C4-Modell, Systemkontextdiagramm, Containerdiagramm, Architecture Decision Record (ADR)

DIE IDEE IN KÜRZE

Vor einer langen Wanderung breitet eine umsichtige Gruppe die Karte auf dem Tisch aus, und alle markieren still und jeder für sich die Stellen, die ihnen Sorgen machen: die Flussquerung, das steile Geröllfeld, den Abschnitt ohne Handyempfang. Dann vergleichen sie. Stellen, die viele markiert haben, bekommen einen Plan; eine Stelle, die nur eine Person markiert hat, bekommt eine Frage, denn diese Person weiß vielleicht etwas, das die anderen nicht wissen. Leichtgewichtige Architektur-Reviews funktionieren genauso. Statt dass ein fernes Gremium einen fertigen Plan genehmigt, betrachten die Menschen, die das System kennen, gemeinsam seine Diagramme, früh und oft, und einigen sich, welche Risiken zuerst angegangen werden.

WAS SIE WISSEN MÜSSEN

- Review-Gremien, die zu festen Zeitpunkten tagen, sehen Entwürfe meist zu spät, haben zu wenig Kontext, laden zu defensiven Präsentationen ein und verursachen Wartezeiten; unabhängige Reviews wirken besser, wenn sie häufig, früh, gemeinsam und risikoorientiert sind.
- Risk-Storming (Simon Brown) hat vier Schritte: Diagramme zeichnen, Risiken einzeln und schweigend identifizieren, sie auf den Diagrammen zusammenführen, dann besprechen und zusammenfassen, bewertet als Wahrscheinlichkeit mal Auswirkung von 1 bis 9.
- Ein Pre-Mortem (Gary Klein, Harvard Business Review, 2007) bittet die Beteiligten, sich vorzustellen, das Vorhaben sei bereits gescheitert, und aufzuschreiben, warum; das bringt Zweifel an Plänen, Fähigkeiten und Zeitplänen ans Licht, die Diagramm-Reviews übersehen.

- Architektur-Katas (Ted Neward) schulen Entwurfs- und Review-Fähigkeiten, während TARA (Eoin Woods) bestehende Systeme prüft, indem es Expertenurteil mit automatisierter Analyse von Code und Laufzeitdaten verbindet.
- Schriftliche Design- und RFC-Reviews funktionieren am besten mit einer benannten Entscheidung, einer Kommentarfrist, verpflichtenden Reviewern und Kommentaren, die als Frage, Vorschlag oder blockierender Einwand gekennzeichnet sind.
- Ein gutes Review liefert ein Risikoregister mit Verantwortlichen, Terminen, akzeptierten Risiken und Nicht-Risiken samt Annahmen, innerhalb weniger Tage veröffentlicht und im normalen Backlog verfolgt.

Warum schwergewichtige Review-Gremien oft scheitern

Viele Organisationen steuern Architektur über ein Architecture Review Board: ein Gremium erfahrener Architekten, das monatlich tagt, einen Foliensatz oder ein langes Dokument erhält und Entwürfe genehmigt oder ablehnt. Die Absicht ist vernünftig: unabhängige, erfahrene Augen auf wesentliche Entscheidungen. Die Ergebnisse sind oft schlecht, aus Gründen, die strukturell und nicht persönlich sind.

Das erste Problem ist der Zeitpunkt. Ein Gremium, das zu festen Zeitpunkten tagt, sieht einen Entwurf entweder zu früh (noch ist nichts entschieden) oder, weit häufiger, zu spät, wenn das Team schon einen Prototyp gebaut, sich an einen Anbieter gebunden und einen Termin zugesagt hat. Eine Ablehnung ist dann so teuer, dass das Gremium unter Auflagen genehmigt, die niemand nachverfolgt. Das zweite ist der Kontext: Reviewer, die eine Stunde mit einem System verbringen, das sie nie gesehen haben, stellen allgemeine Fragen und übersehen die konkreten Risiken, die die eigenen Entwickler, Betriebs- und Sicherheitsfachleute des Teams in Minuten nennen könnten. Das dritte sind die Anreize: Wenn ein Review eine Prüfung ist, präsentieren Teams den Entwurf, den sie genehmigt haben wollen, nicht ihre Zweifel, und das Gremium erfährt wenig. Das vierte ist die Warteschlange: Eine monatliche Sitzung mit voller Tagesordnung fügt jeder wesentlichen Entscheidung Wochen des Wartens hinzu und verleitet Teams dazu, das Gremium zu umgehen, indem sie Entscheidungen als geringfügig deklarieren.

Das alles heißt nicht, dass unabhängige Reviews wertlos sind. Regulierte Branchen (Banken, Gesundheitswesen, Luftfahrt) brauchen oft eine dokumentierte Freigabe, und manche Entscheidungen brauchen tatsächlich die Autorität eines leitenden Gremiums. Die Lehre ist, dass Reviews *häufig, früh, gemeinsam und risikoorientiert* sein sollten, wobei eine formale Genehmigung den wenigen Entscheidungen vorbehalten bleibt, die sie wirklich erfordern. Abschnitt 11.4 greift auf, wie Governance ohne langsame Gates funktionieren kann.

Was ein gutes leichtgewichtiges Review leistet

Ein leichtgewichtiges Review ist eine kurze, strukturierte Sitzung, typischerweise ein bis zwei Stunden, in der die Menschen, die einen Entwurf am besten kennen, und einige, die ihn mit frischem Blick sehen, gemeinsam nach Risiken suchen. Es zielt auf vier Dinge: Risiken entdecken, solange sie günstig zu beheben sind, ein gemeinsames Verständnis des Entwurfs im Team aufbauen, die Annahmen des Architekten unabhängig hinterfragen und eine Aufzeichnung der akzeptierten, abgemilderten oder verworfenen Risiken hinterlassen. Sein Ergebnis ist eine priorisierte Risikoliste mit Verantwortlichen, kein Urteil.

Mehrere Techniken passen zu dieser Beschreibung. Sie unterscheiden sich darin, was sie untersuchen (ein Diagramm, einen Plan, ein Dokument oder ein laufendes System) und worin sie am besten sind, und sie lassen sich gut kombinieren.

Leichtgewichtige Review-Techniken im Vergleich

Technik	Untersucht	Am besten für	Typische Dauer
Risk-Storming (Simon Brown)	Architekturdiagramme	Technische Risiken schnell mit dem ganzen Team finden	45 bis 90 Minuten
Pre-Mortem (Gary Klein)	Einen Plan oder eine Entscheidung	Zweifel am gesamten Vorhaben aufdecken, einschließlich Personen und Zeitplan	30 bis 60 Minuten
Architektur-Kata (Ted Neward)	Eine kurze, erfundene oder reale Aufgabenstellung	Entwurfskompetenz aufbauen und Alternativen erkunden	1 bis 3 Stunden
TARA (Eoin Woods)	Ein bestehendes System: Personen, Dokumente, Code und Laufzeitdaten	Ein laufendes System auf Drift und Risiken prüfen	Einige Tage Expertenaufwand
Design- oder RFC-Review	Einen schriftlichen Vorschlag	Asynchrone, offene Kommentierung vor dem Bau	Ein Kommentarfenster von Tagen plus eine kurze Besprechung
Checklisten-Review	Jeden Entwurf	Sicherstellen, dass die üblichen blinden Flecken bedacht wurden	Teil jeder der obigen Techniken

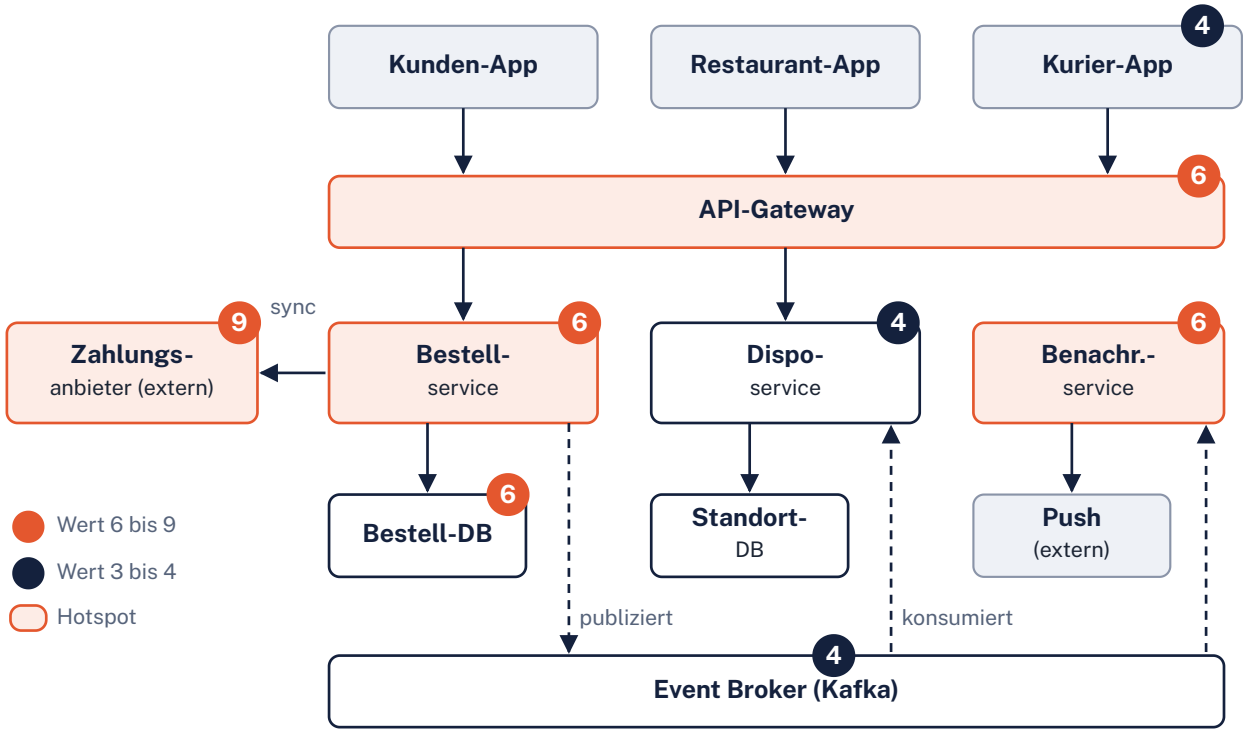
Risk-Storming

Risk-Storming ist eine visuelle, kollaborative Technik von Simon Brown, dem Urheber des C4-Modells (Abschnitt 4.3). Sie beruht auf einer einfachen Beobachtung: Verschiedene Menschen sehen verschiedene Risiken, und am besten sammelt man sie, indem man alle vor dieselben Diagramme stellt und sie bittet, unabhängig hinzusehen, bevor diskutiert wird. Sie hat vier Schritte.

1. Die Architektur zeichnen. Hängen Sie Diagramme dessen auf, was Sie bauen oder ändern wollen, idealerweise auf mehr als einer Ebene: Ein Systemkontextdiagramm und ein Containerdiagramm genügen meist, dazu ein Komponentendiagramm für den riskantesten Container. **2. Risiken einzeln identifizieren.** Alle Beteiligten schreiben etwa zehn Minuten lang schweigend je ein Risiko pro Haftnotiz, mit kurzer Beschreibung und Priorität. Das Schweigen ist wichtig: Es verhindert, dass die erste oder ranghöchste Stimme alle anderen verankert, und genau so verlieren Gruppendiskussionen am häufigsten Information. **3. Die Risiken auf den Diagrammen zusammenführen.** Alle kleben ihre Notizen auf den Teil des Diagramms, den das Risiko betrifft. Sofort entstehen Häufungen, und das Diagramm wird zur Heatmap. **4. Besprechen und zusammenfassen.** Die Gruppe bespricht die Häufungen und achtet dann besonders auf zwei Arten von Notizen: Risiken, die nur eine Person genannt hat (oft jemand mit Wissen, das anderen fehlt, etwa der Support-Engineer), und Risiken, denen verschiedene Personen sehr unterschiedliche Prioritäten gegeben haben (ein Zeichen für ein Missverständnis). Vereinbarte Risiken kommen mit Verantwortlichen in ein Risikoregister; Punkte, die man übereinstimmend nicht als Risiko einstuft, werden ebenfalls festgehalten.

Die Priorität wird meist als Wahrscheinlichkeit mal Auswirkung bewertet, jeweils niedrig (1), mittel (2) oder hoch (3), was Werte von 1 bis 9 ergibt. Browns Leitfaden teilt sie ein in hoch (6 oder 9), mittel (3 oder 4) und niedrig (1 oder 2), üblicherweise dargestellt mit rosa oder roten, gelben oder orangefarbenen und grünen Notizen. Eine Liste von Denkanstößen hilft, über das eigene Fachgebiet hinauszudenken: Skalierbarkeit, Verfügbarkeit, Performance, Sicherheit, Datenkonsistenz, Abhängigkeiten von Drittanbietern, Kosten, technologische Reife, Fähigkeiten des Teams sowie Auslieferung und Betrieb.

Eine Risk-Storming-Heatmap auf einem Containerdiagramm



Nach dem Zusammenführen häufen sich die Werte auf dem Bestell- und Zahlungspfad, dem gemeinsamen Gateway und den Restaurant-Benachrichtigungen; diese Container sind die Hotspots, die zuerst besprochen werden.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Risk-Storming am Containerdiagramm der Essenslieferplattform

Sechs Personen (der Architekt, zwei Entwickler, ein Betriebsingenieur, ein Sicherheitsingenieur und ein Support-Engineer, der Beschwerden von Restaurants bearbeitet) verbrachten 75 Minuten mit dem obigen Containerdiagramm, das Systemkontextdiagramm daneben. Alle schrieben zehn Minuten lang schweigend Risiken auf; 31 Notizen kamen an die Wand, die sich zu 8 unterschiedlichen Risiken zusammenfassen ließen.

Zahlungsanbieter auf dem synchronen Bestellpfad (9)	Von vier Personen genannt. Verlangsamungen des Anbieters zur Spitzenzeit blockieren die Bestellaufgabe. Maßnahme: Entscheidungsdokument zur asynchronen Autorisierung mit einem Bestellstatus „ausstehend“; verantwortlich: Architekt; fällig in zwei Wochen
Restaurant-Benachrichtigungen werden nicht bestätigt (6)	Nur vom Support-Engineer genannt, der jeden Freitag Beschwerden über verpasste Bestellungen sieht. Sonst wusste es niemand. Maßnahme: bestätigte Zustellung mit erneutem Senden und einem Alarm, wenn ein Tablet keine Bestätigungen mehr sendet; verantwortlich: Benachrichtigungsteam
Gemeinsame Gateway-Konfiguration (6)	Eine fehlerhafte Routenänderung kann alle drei Apps gleichzeitig lahmlegen. Maßnahme: Gateway-Konfiguration pro App und Canary-Rollout von Gateway-Änderungen
Failover der Bestelldatenbank nie getestet (6)	Ein Replikat existiert, aber die Hochstufung wurde nie geprobt. Maßnahme: vierteljährliche Failover-Übung, gemessen am Wiederherstellungsszenario
Kafka-Kenntnisse (von einer Person mit 9 bewertet, von den anderen mit 2)	Der einzige Ingenieur, der Kafka schon betrieben hatte, erklärte Partition-Rebalancing während Deployments, Consumer-Lag und Fehler bei der Aufbewahrung. Nach der Diskussion einigte sich die Gruppe auf 4: real, aber beherrschbar. Maßnahme: verwalteter Kafka-Dienst und ein Runbook
Kurier-Standortverkehr über das Gateway (4)	Etwa 2250 Updates pro Sekunde teilen sich das Gateway mit dem Bestellverkehr. Maßnahme: im nächsten Lasttest messen; separater Erfassungs-Endpoint, falls die Bestelllatenz leidet
Consumer-Lag der Disposition zur Spitzenzeit (4)	Maßnahme: Alarm bei Consumer-Lag; eine Fitness Function in den Lasttest aufnehmen
Standortspeicher verliert beim Neustart Daten (2)	Übereinstimmend kein Risiko: Positionen werden alle 4 Sekunden aktualisiert. Als solches festgehalten, mit der Annahme

Der wertvollste Befund (Restaurant-Benachrichtigungen) kam von der Person, die in einer offenen Diskussion am wenigsten wahrscheinlich als Erste gesprochen hätte; genau darum geht es bei der schweigenden, individuellen Identifikation.

Die Uneinigkeit über Kafka wurde durch Wissen gelöst, nicht durch Mittelung der Werte.

Jedes vereinbarte Risiko verließ den Raum mit einer verantwortlichen Person und einer Maßnahme; das Register wurde noch am selben Tag veröffentlicht.

Pre-Mortems

Risk-Storming betrachtet eine Struktur. Ein Pre-Mortem betrachtet einen Plan. Gary Klein beschrieb es im September 2007 in der *Harvard Business Review*. Eine Obduktion erklärt, warum ein Patient gestorben ist, was allen hilft außer dem Patienten; ein Pre-Mortem findet statt, bevor das Projekt beginnt, wenn die Antworten seinen Kurs noch ändern können.

Das Vorgehen ist kurz. Die Leitung erläutert der Gruppe den Plan oder die Entscheidung und verkündet dann, es sei jetzt einige Monate später und das Vorhaben sei gescheitert, und zwar gründlich. Alle schreiben einige Minuten lang unabhängig voneinander jeden Grund auf, der ihnen für das Scheitern einfällt. Dann geht die Leitung reihum und sammelt jeweils einen Grund, bis die Listen erschöpft sind, und die verantwortliche Person des Plans prüft sie auf Möglichkeiten, ihn zu stärken. Klein zitiert Forschung von Deborah Mitchell, Jay Russo und Nancy Pennington (1989), wonach die Vorstellung, ein Ereignis sei bereits eingetreten (prospektive Rückschau), die Fähigkeit, Gründe für künftige Ergebnisse zu erkennen, um etwa 30 % steigert. Die Rahmung verändert auch die soziale Dynamik: Die Frage ist nicht mehr, ob der Plan scheitern könnte, was illoyal klingt, sondern warum er gescheitert ist, was zur Kreativität einlädt.

In der Architektur sind Pre-Mortems besonders vor großen Festlegungen nützlich: bei der Einführung eines ereignisgesteuerten Stils, der Migration in eine neue Cloud, der Ablösung eines Altsystems (Abschnitt 11.6) oder der Einführung einer Plattform. Typische Ergebnisse, die ein diagrammbasiertes Review übersehen würde, sind ein Team ohne Erfahrung mit der neuen Technologie, eine Migration, die auf halbem Weg stecken bleibt und zwei Systeme zu betreiben hinterlässt, ein wichtiger Lieferant, der seine Preise ändert, und ein Geschäft, das die Geduld verliert, bevor sich Nutzen zeigt.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: zuerst das Scheitern vorstellen

Die Idee, die hier die meiste Sorgfalt verlangt, ist, warum ein Pre-Mortem funktioniert: Sich vorzustellen, ein Plan sei bereits gescheitert, bringt Zweifel ans Licht, die eine schlichte Diskussion über „Was könnte schiefgehen?“ oft unausgesprochen lässt. Verfolgen Sie einen Plan aus der obigen Risk-Storming-Sitzung.

- Angenommen, das Team beschließt, die Kartenautorisierung aus dem synchronen Bestellpfad zu nehmen und Bestellungen im Status „ausstehend“ anzunehmen. Bevor die Arbeit beginnt, treffen sich das Team und Leute aus Nachbarteams für eine halbe Stunde.
- Die Leitung erklärt den Plan und verkündet dann, es sei jetzt sechs Monate später und die Änderung sei gründlich gescheitert. Alle schreiben allein und schweigend jeden Grund auf, der ihnen einfällt.
- Die Rahmung leistet die Arbeit. Fragt man, ob der Plan scheitern könnte, klingt jemand mit Zweifeln illoyal, wenn er sie äußert. Fragt man, warum er gescheitert ist, wird jeder Zweifel zu einer nützlichen Antwort.
- Die Leitung sammelt jeweils einen Grund, bis die Listen erschöpft sind: Niemand im Bestellteam hatte so einen Entwurf schon gebaut; die Änderung blieb auf halbem Weg stecken, sodass alter und neuer Zahlungspfad parallel liefen; der Zahlungsanbieter änderte seine Preise; das Geschäft verlor die Geduld, bevor sich der Nutzen zeigte.
- Kaum einer dieser Punkte würde als Haftnotiz auf dem Containerdiagramm auftauchen: Sie betreffen eher Zeitplan, Lieferanten und Geduld als Kästen und Pfeile. Die verantwortliche Person des Plans stärkt ihn dann, etwa indem sie zuerst Schulungen organisiert und einen Termin für das Abschalten des alten Pfads festlegt.
- **Die Lehre:** Risk-Storming untersucht eine Struktur; ein Pre-Mortem untersucht einen Plan. Sich vorzustellen, das Scheitern sei bereits eingetreten, macht Gründe leichter zu finden und sicherer auszusprechen; halten Sie daher eines vor einer großen Festlegung ab, solange die Antworten den Plan noch ändern können.

Architektur-Katas

Architektur-Katas wurden von Ted Neward, inspiriert von Dave Thomas’ Code-Katas, als Übungen für Architekten geschaffen. Eine Kata ist eine kurze Aufgabenstellung für ein fiktives System (einige Absätze zu Anforderungen, Nutzern und Randbedingungen, bewusst unvollständig). Kleine Teams von drei bis fünf Personen bearbeiten sie für eine feste Zeit, befragen eine Moderation, die den Kunden spielt, um fehlende Anforderungen aufzudecken,

skizzieren dann eine Architektur und stellen sie den anderen Teams vor, die sie hart, aber fair hinterfragen, bevor der Raum abstimmt. Neal Ford pflegt eine öffentliche Sammlung von Katas, und O'Reilly hat Wettbewerbe mit Architektur-Katas veranstaltet.

Katas sind keine Reviews eines realen Entwurfs, tragen aber auf zwei Weisen zur Review-Kultur bei. Sie schulen die Fähigkeit, einen Entwurf unter Befragung zu erklären und zu verteidigen und gute Fragen zum Entwurf anderer zu stellen, genau das, was leichtgewichtige Reviews brauchen. Und eine interne Kata zu einem realen anstehenden Problem, bei der zwei oder drei Teams unabhängig entwerfen, ist ein günstiger Weg, den Lösungsraum vor einer Festlegung zu erkunden: Wenn drei Teams drei verschiedene Entwürfe liefern, zeigen die Unterschiede, wo die echten Abwägungen liegen.

TARA und checklistenbasierte Reviews

Reviews bestehender Systeme stehen vor einem anderen Problem: Die gebaute Architektur kann von der beschriebenen abweichen, und die Belege sind über Personen, Dokumente, Code und Produktionsdaten verstreut. Eoin Woods, Mitautor von *Software Systems Architecture*, beschrieb 2011 den Tiny Architectural Review Approach (TARA) als bewusst leichtgewichtige, praxistaugliche Alternative zu szenariolastigen Methoden. Eine kleine Zahl von Reviewern erhebt die wichtigsten Anforderungen und den Kontext des Systems, ermittelt seine tatsächliche Struktur, beurteilt seine Implementierung mit automatisierter Analyse (Modulabhängigkeiten, Größe, Code-Metriken, Testabdeckung) neben Laufzeitinformationen wie Logs, verknüpft das Gefundene mit Expertenurteil mit den Anforderungen und präsentiert Ergebnisse und Empfehlungen in einer vorab festgelegten Form. Seine Stärke sind Tempo und Belege; seine Schwäche, die Woods selbst einräumte, ist die Abhängigkeit von der Expertise der Reviewer.

Checklisten ergänzen jede Technik. Eine Checkliste findet allein keine Risiken, verhindert aber, dass ein Review eine ganze Kategorie vergisst (Wiederherstellung aus Backups, Secrets, Kosten im großen Maßstab), nur weil niemand im Raum zufällig daran gedacht hat. Halten Sie Checklisten kurz, zugeschnitten auf die Vorfallgeschichte Ihrer Organisation, und formulieren Sie sie als Fragen, die Belege verlangen statt Ja-oder-Nein-Antworten.

Well-Architected-Reviews

Die großen Cloud-Anbieter veröffentlichen Frameworks, die als fertige Review-Checklisten dienen können. AWS und Google Cloud beschreiben je sechs Säulen, Azure fünf; sie decken Betrieb, Sicherheit, Zuverlässigkeit, Performance und Kosten ab, bei den ersten beiden kommt Nachhaltigkeit hinzu. Ein Review führt einen Workload durch strukturierte Fragen. Das AWS Well-Architected Tool kostet nichts extra, hält Ihre Antworten fest und markiert Probleme mit hohem Risiko, und mit benutzerdefinierten Lenses lassen sich eigene Regeln ergänzen; Azure

bietet ein Online-Well-Architected-Review. Für den Bestell-Workload der Essenslieferplattform ergibt eine halbtägige Sitzung eine breite Abdeckung und eine Liste von Risiken, die zu beheben oder zu akzeptieren sind. Die Grenzen sind real: Die Fragen spiegeln die Sicht eines Anbieters auf gute Praxis wider und wägen Ihre eigenen Qualitätsszenarien nicht gegeneinander ab, wie es Abschnitt 11.1 tut. Prüfen Sie die aktuelle Version, denn Säulen und Fragen ändern sich.

Designdokumente und RFC-Reviews

Abschnitt 4.8 hat Designdokumente und RFCs als Weg beschrieben, eine Änderung schriftlich vorzuschlagen, bevor sie gebaut wird. Sie gut zu reviewen ist eine eigene Fähigkeit. Ein schriftliches Review hat echte Vorteile gegenüber einer Besprechung: Reviewer kommentieren, wenn sie Zeit zum Nachdenken haben, Kolleginnen und Kollegen aus der Ferne und stillere Teammitglieder tragen gleichberechtigt bei, und die Kommentare werden zur Aufzeichnung. Typische Fehlschläge sind Dokumente, die niemand liest, Kommentar-Threads, die nie abgeschlossen werden, und Reviews, die zu einem inoffiziellen Gate werden.

Gute Praxis ist, oben die gesuchte Entscheidung und die Kommentarfrist zu nennen; die Reviewer zu benennen, deren Beitrag erforderlich ist (Sicherheit, Betrieb, die Verantwortlichen betroffener Services), und alle anderen einzuladen; die Reviewer zu bitten, jeden Kommentar als Frage, Vorschlag oder blockierenden Einwand zu kennzeichnen, damit der Autor weiß, welche gelöst werden müssen; gegen die Qualitätsszenarien und die Checkliste unten zu reviewen, nicht nach persönlichem Geschmack; nur für ungelöste Einwände eine kurze Besprechung anzusetzen; und die wesentlichen Entscheidungen als ADRs festzuhalten. In Organisationen mit Advice Process (Abschnitt 11.4) ist der RFC das Mittel, um Rat einzuholen, und die Entscheidung bleibt beim Autor.

Ein 90-minütiges Architektur-Review durchführen, Schritt für Schritt

1. **Vorher: vorbereiten (einige Tage im Voraus)**
   Der Architekt verteilt eine einseitige Zusammenfassung, das Kontext- und das Containerdiagramm, die drei bis fünf wichtigsten Qualitätsszenarien mit Zahlen und die offenen Fragen. Laden Sie fünf bis acht Personen ein: das Team, Betrieb, Sicherheit, jemanden aus jedem Nachbarteam und eine erfahrene Person von außen. Wählen Sie eine Moderation, die nicht der Architekt ist.

2. **0 bis 10 Minuten: Ziele und Regeln**
   Die Moderation nennt den Zweck (Risiken finden, nicht genehmigen oder ablehnen), den Zeitrahmen und das Ergebnis. Eine Vertretung des Geschäfts oder der Product Owner fasst in zwei Minuten die geschäftlichen Treiber zusammen.

3. **10 bis 25 Minuten: die Architektur anhand von Szenarien durchgehen**
   Der Architekt spielt zwei oder drei der wichtigsten Szenarien anhand der Diagramme durch, etwa eine Bestellung zur Spitzenzeit und einen Ausfall des Zahlungsanbieters. Nur Verständnisfragen.

4. **25 bis 35 Minuten: schweigende Risikoidentifikation**
   Alle schreiben Risiken auf Notizen, eines pro Notiz, mit Wahrscheinlichkeit und Auswirkung. Keine Diskussion.

5. **35 bis 50 Minuten: zusammenführen und gruppieren**
   Die Notizen kommen auf die Diagramme. Die Moderation fasst Duplikate laut zusammen und hebt Häufungen, einzelne Notizen und unterschiedliche Prioritäten hervor.

6. **50 bis 75 Minuten: die wichtigsten Risiken besprechen**
   Beginnen Sie mit den am höchsten bewerteten Häufungen. Klären Sie für jede, was das Risiko wirklich ist, wie es bewertet wird und ob es ein Risiko, ein Nicht-Risiko mit Annahme oder eine Frage ist, die einen Spike erfordert.

7. **75 bis 85 Minuten: Maßnahmen und Verantwortliche**
   Jedes vereinbarte hohe oder mittlere Risiko erhält eine Gegenmaßnahme, eine verantwortliche Person und einen Termin. Entscheidungen, die als ADRs festgehalten werden sollen, werden aufgelistet.

8. **85 bis 90 Minuten: Abschluss**
   Lesen Sie die Risiken, die akzeptierten Risiken und die Nicht-Risiken mit ihren Annahmen vor. Vereinbaren Sie, wann der Entwurf erneut betrachtet wird.

9. **Danach: innerhalb von zwei Tagen veröffentlichen**
   Veröffentlichen Sie das Risikoregister, Fotos der kommentierten Diagramme und die Maßnahmenliste dort, wo das Team seine Architekturdokumentation führt, und verfolgen Sie die Maßnahmen im normalen Backlog.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: welche Art von Review?

Beginnen Sie oben; Techniken lassen sich kombinieren.

Q1 **Steht eine teure, schwer umkehrbare Festlegung bevor, etwa ein mehrjähriger Vertrag, ein Plattformkauf oder eine Finanzierungsentscheidung, mit vielen Stakeholdern?**

Ja Nutzen Sie eine ATAM oder eine reduzierte ATAM mit unabhängigen Bewertern (Abschnitt 11.1) und ergänzen Sie ein Pre-Mortem zum Plan.

Nein Weiter.

Q2 **Gibt es einen schriftlichen Vorschlag für eine wesentliche Änderung?**

Ja Führen Sie ein schriftliches Design- oder RFC-Review mit Kommentarfrist durch, dann eine kurze Besprechung für blockierende Einwände.

Nein Weiter.

Q3 **Gibt es Diagramme der geplanten oder bestehenden Architektur?**

Ja Führen Sie darauf ein Risk-Storming im 90-Minuten-Format durch.

Nein Zeichnen Sie zuerst ein Kontext- und ein Containerdiagramm; schon das Zeichnen legt oft Risiken offen.

Q4 **Ist die Hauptsorge eher der Erfolg des gesamten Vorhabens (Fähigkeiten, Zeitplan, Migration, Akzeptanz) als die Struktur?**

Ja Ergänzen Sie ein Pre-Mortem.

Nein Weiter.

Q5 **Läuft das System bereits, mit vermuteter Drift oder wachsenden Schulden?**

Ja Nutzen Sie ein Review im Stil von TARA, das Interviews mit automatisierter Code- und Laufzeitanalyse verbindet, und machen Sie aus den Ergebnissen Fitness Functions.

Nein Verlassen Sie sich zwischen den Reviews auf Fitness Functions und reviewen Sie erneut bei der nächsten wesentlichen Entscheidung.

Checkliste: Fragen für jedes leichtgewichtige Architektur-Review

Verlangen Sie Belege, keine Beruhigung. Passen Sie die Liste an die Vorfallgeschichte Ihrer Organisation an.

- [ ] **Welche drei bis fünf Qualitätsszenarien sind am wichtigsten, und wie lauten ihre Zahlen?**
  Ohne messbare Treiber kann das Review Risiken nicht gewichten (Abschnitt 2.3).
- [ ] **Was liegt auf dem synchronen Pfad der wichtigsten User Journey?**
  Zählen Sie die Hops, listen Sie Timeouts und Wiederholungen auf und identifizieren Sie jeden Drittanbieter auf dem Pfad.
- [ ] **Was passiert, wenn eine Abhängigkeit langsam ist oder ausfällt?**
  Langsam ist meist schlimmer als ausgefallen; fragen Sie nach beidem und danach, wie Nutzer informiert werden.
- [ ] **Wem gehören die wichtigen Daten, und wurde eine Wiederherstellung getestet?**
  Ein einziger Schreiber, Erwartungen an die Konsistenz, Aufbewahrung und Nachweis einer kürzlich erfolgreichen Wiederherstellung (Kapitel 7).
- [ ] **Wo liegen die Vertrauensgrenzen, Identitäten und Secrets?**
  Achten Sie auf ein aktuelles Bedrohungsmodell und auf Secrets in Code oder Konfiguration (Kapitel 8).
- [ ] **Was bricht zuerst beim Doppelten der erwarteten Spitzenlast?**
  Verlangen Sie das Kapazitätsmodell oder Lasttestergebnisse, nicht Intuition.
- [ ] **Wie wird das System beobachtet, deployt und zurückgerollt?**
  Dashboards, an SLOs gekoppelte Alarme, Deployment-Strategie und die Dauer eines Rollbacks.
- [ ] **Was wird sich im nächsten Jahr wahrscheinlich ändern, und was würde diese Änderung berühren?**
  Prüfen Sie, dass wahrscheinliche Änderungen innerhalb eines Moduls oder Service bleiben.
- [ ] **Was wird der Betrieb beim erwarteten Volumen kosten?**
  Kosten pro Bestellung, pro Nutzer oder pro Transaktion, mit den wichtigsten Kostentreibern.
- [ ] **Sind die wesentlichen Entscheidungen mit Alternativen und Konsequenzen festgehalten?**
  ADRs oder ein Designdokument, einschließlich der akzeptierten Abwägungen.

- [ ] **Passt die Teamstruktur zum Entwurf?**
  Jeder Teil hat ein verantwortliches Team mit den Fähigkeiten, ihn zu betreiben (Abschnitt 3.7 und Abschnitt 11.5).

ACHTUNG

Wie leichtgewichtige Reviews schiefgehen

- Offene Diskussion vor der schweigenden Identifikation, sodass die erste Stimme die Gruppe verankert.
- Uneinigkeit wegmitteln, statt zu fragen, warum eine Person ein Risiko so anders bewertet hat.
- Die Notation des Diagramms reviewen statt der Risiken der Architektur.
- Keine Verantwortlichen und Termine, sodass das Risikoregister zur Liste guter Vorsätze wird.
- Der Architekt als Moderation, was es schwer macht, Kritik anzunehmen und gleichzeitig die Sitzung zu leiten.
- Reviews nur durchführen, wenn schon etwas in Schwierigkeiten ist; regelmäßige Routine-Reviews sind günstiger und weniger defensiv.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum verlangt Risk-Storming, Risiken schweigend zu identifizieren, bevor diskutiert wird?

Damit die erste oder ranghöchste Stimme die Gruppe nicht verankert und das unabhängige Wissen jeder Person auf das Diagramm gelangt.

▼ Welchen zwei Arten von Notizen gebührt beim Risk-Storming im Besprechungsschritt besondere Aufmerksamkeit?

Risiken, die nur eine Person genannt hat, die vielleicht etwas weiß, was andere nicht wissen, und Risiken mit sehr unterschiedlichen Prioritäten, die auf ein Missverständnis hindeuten.

▼ Was untersucht ein Pre-Mortem, das eine Risk-Storming-Sitzung an einem Containerdiagramm meist nicht untersucht?

Den Plan und seinen Kontext als Ganzes: Fähigkeiten, Zeitplan, Migration, Lieferanten und die Geduld der Organisation statt der technischen Struktur.

▼ **Nennen Sie zwei strukturelle Gründe, warum monatliche Architektur-Review-Gremien oft scheitern.**

Zwei beliebige von: Sie sehen Entwürfe zu spät, Reviewern fehlt der Kontext, Teams präsentieren defensiv, und die Warteschlange fügt Wochen Verzögerung hinzu, die zum Umgehen verleiten.

▼ **Probieren Sie es aus: Ein Risk-Storming bewertet (Wahrscheinlichkeit, Auswirkung): Ausfall des Zahlungsanbieters (3, 3); Fehlkonfiguration des Brokers (2, 3); veralteter Suchindex (3, 1); Brand im Rechenzentrum (1, 3). Berechnen Sie die Werte, und bringen Sie die Risiken in eine Rangfolge. Warum wurden zuerst schweigend Notizen geschrieben?**

Zahlungsanbieter 9, hoch. Broker 6, hoch. Suchindex 3 und Feuer 3, beide mittel, wobei das Feuer wegen seiner hohen Auswirkung eine Zeile mit seinen Annahmen verdient. Gehen Sie die beiden obersten zuerst an, mit Verantwortlichen und Terminen im normalen Backlog. Stilles Schreiben verhindert, dass die erste oder ranghöchste Stimme die Gruppe lenkt. Wenn vier Personen unabhängig voneinander dasselbe Risiko nennen, ist das ein nützlicher Beleg, und Meinungsverschiedenheiten zeigen abweichende Annahmen.

QUELLEN

- Simon Brown: Risk-storming: https://riskstorming.com/
- Gary Klein: Performing a Project Premortem, Harvard Business Review (2007): https://hbr.org/2007/09/performing-a-project-premortem
- Neal Ford: Architectural Katas: https://nealford.com/katas/about.html
- Experiences with ALMA: Architecture-Level Modifiability Analysis: https://research.vu.nl/en/publications/experiences-with-alma-architecture-level-modifiability-analysis/
- The pillars of the AWS Well-Architected Framework: https://docs.aws.amazon.com/wellarchitected/latest/framework/the-pillars-of-the-framework.html
- What is AWS Well-Architected Tool?: https://docs.aws.amazon.com/wellarchitected/latest/userguide/intro.html
- Workloads (AWS Well-Architected Tool): https://docs.aws.amazon.com/wellarchitected/latest/userguide/workloads.html
- AWS Well-Architected Tool pricing: https://aws.amazon.com/well-architected-tool/pricing/
- Azure Well-Architected Framework pillars (Microsoft): https://learn.microsoft.com/en-us/azure/well-architected/pillars
- Google Cloud Well-Architected Framework: https://docs.cloud.google.com/architecture/framework

Fitness Functions und evolutionäre Architektur

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.3 Qualitätsszenarien
- 3.8 Einfachheit und Messung
- 10.5 CI/CD und Deployment

Begriffe von früher, die er verwendet: Qualitätsszenario, Antwortmaß (Response Measure), Architecture Decision Record (ADR), Utility Tree, Last Responsible Moment

DIE IDEE IN KÜRZE

Das Armaturenbrett eines Autos hat Warnleuchten für die Dinge, die nie unbemerkt schiefgehen dürfen: Öldruck, Motortemperatur, Bremsen. Sie warten nicht auf die jährliche Inspektion, um herauszufinden, dass der Motor im März überhitzt ist. Eine Fitness Function ist eine Warnleuchte für eine Softwarearchitektur. Das Team legt fest, welche Qualitäten wirklich zählen, etwa wie schnell eine Bestellung bestätigt wird oder welche Teile des Codes miteinander sprechen dürfen, schreibt für jede eine automatische Prüfung und führt diese Prüfungen bei jeder Änderung der Software aus. Schlägt eine Prüfung fehl, wird die Änderung gestoppt oder jemand alarmiert, solange das Problem noch klein ist.

WAS SIE WISSEN MÜSSEN

- Eine architektonische Fitness Function ist jeder Mechanismus, der ein Architekturmerkmal objektiv bewertet, idealerweise automatisiert und rückverfolgbar auf ein Qualitätsszenario, dessen Antwortmaß zu ihrem Schwellenwert wird.
- Fitness Functions werden eingeteilt in atomar oder holistisch, ausgelöst, kontinuierlich oder zeitgesteuert, statisch oder dynamisch, automatisiert oder manuell sowie geplant oder emergent; ein gesunder Satz mischt sie.
- Typische Beispiele sind Abhängigkeitsregeln für die Struktur, Lasttests und Latenzbudgets für die Performance, Schwachstellenscans für die Sicherheit, Chaos-Experimente für die Resilienz und Prüfungen der Schemakompatibilität für Daten.
- Evolutionäre Architektur (Ford, Parsons, Kua und Sadalage) unterstützt gelenkte, inkrementelle Änderung über mehrere Dimensionen; Fitness Functions liefern die Lenkung, kleine deploybare Einheiten und Pipelines liefern die Inkremente.

- Jede Fitness Function braucht einen Verantwortlichen, eine dokumentierte Begründung und einen Schwellenwert, und ein Fehlschlag erzwingt eine Entscheidung: den Code korrigieren oder die Regel ändern und festhalten, warum.
- Beginnen Sie mit wenigen zuverlässigen Prüfungen für die Merkmale mit der höchsten Priorität, nutzen Sie Baselines, um bestehende Systeme schrittweise zu verschärfen, und halten Sie langsame holistische Prüfungen aus jedem einzelnen Build heraus.

Von der einmaligen Bewertung zur kontinuierlichen Prüfung

Eine Bewertung wie die ATAM (Abschnitt 11.1) ist eine Momentaufnahme. Am Tag nach ihrem Ende beginnen Entwickler, das System zu ändern, und jede Änderung kann still eine Entscheidung rückgängig machen, auf die sich die Bewertung gestützt hat: ein bequemer Import über eine Modulgrenze hinweg, ein zusätzlicher synchroner Aufruf auf dem Bestellpfad, eine Bibliothek mit einer bekannten Schwachstelle. Abschnitt 1.7 hat das Ergebnis beschrieben: Erosion, wenn die Architektur verletzt wird, und Drift, wenn sie ignoriert wird. Regelmäßige Reviews fangen einiges davon ab, Monate zu spät. Die Idee dieses Abschnitts ist, die wichtigsten Architekturentscheidungen automatisch und kontinuierlich zu prüfen, so wie Unit-Tests das Verhalten prüfen.

Der Begriff stammt aus *Building Evolutionary Architectures* (O'Reilly, 2017) von Neal Ford, Rebecca Parsons und Patrick Kua, dessen zweite Ausgabe, mit Pramod Sadalage, im November 2022 erschien. Sie haben ihn aus der evolutionären Informatik entlehnt, wo eine Fitness Function bewertet, wie nah eine Kandidatenlösung am Ziel ist, damit der Algorithmus die besseren Varianten behalten kann. In der Architektur ist das Ziel die Menge der Merkmale, die das System behalten muss, während sich alles andere ändert.

Was eine Fitness Function ist

Die zweite Ausgabe definiert eine architektonische Fitness Function als jeden Mechanismus, der eine objektive Integritätsbewertung eines oder mehrerer Architekturmerkmale liefert. Drei Wörter in dieser Definition tragen die Last. *Jeder Mechanismus*: Ein Unit-Test, eine Regel der statischen Analyse, ein Lasttest, ein Monitor, ein Chaos-Experiment oder sogar ein geplantes manuelles Review kommen alle infrage. *Objektiv*: Das Ergebnis darf nicht davon abhängen, wer die Prüfung ausführt; „der Code fühlt sich gut strukturiert an“ ist keine Fitness Function, „kein Paket in der Domänenschicht importiert das Web-Framework“ schon. *Architekturmerkmal*: Die Prüfung schützt eine Qualität, für die die Architektur da ist (Performance, Verfügbarkeit, Änderbarkeit, Sicherheit, Datenkompatibilität), keine Geschäftsregel; das ist die Aufgabe gewöhnlicher Tests.

In der Praxis hat eine nützliche Fitness Function noch einige weitere Eigenschaften. Sie ist auf eine Anforderung rückführbar, idealerweise auf ein Qualitätsszenario, dessen Antwortmaß zu ihrem Schwellenwert wird (Abschnitt 2.3). Sie hat einen Verantwortlichen, der entscheidet, was bei einem Fehlschlag passiert. Sie schlägt laut fehl, indem sie einen Build abbricht oder jemanden alarmiert, statt ein Dashboard zu füllen, das niemand liest. Und ihr Schwellenwert samt Begründung ist dokumentiert, meist in einem ADR, damit ein späteres Team eine Regel, die etwas Wichtiges schützt, von einer willkürlichen Zahl unterscheiden kann.

Abschnitt 3.8 hat die wichtige Abgrenzung zu Metriken gezogen: Eine Metrik beschreibt, eine Fitness Function entscheidet. Eine Kopplungsmetrik auf einem Dashboard zeigt Ihnen, dass es schlechter wird; eine Fitness Function kodiert eine Entscheidung, die das Team bereits getroffen hat, sodass ihr Bruch eine bewusste Wahl erzwingt: den Code korrigieren oder die Regel ändern und festhalten, warum.

Anatomie einer Fitness Function



Eine Fitness Function macht aus dem Antwortmaß eines Szenarios einen Schwellenwert, eine automatische Prüfung und eine Aktion, mit einem Verantwortlichen und einer dokumentierten Begründung.

Kategorien von Fitness Functions

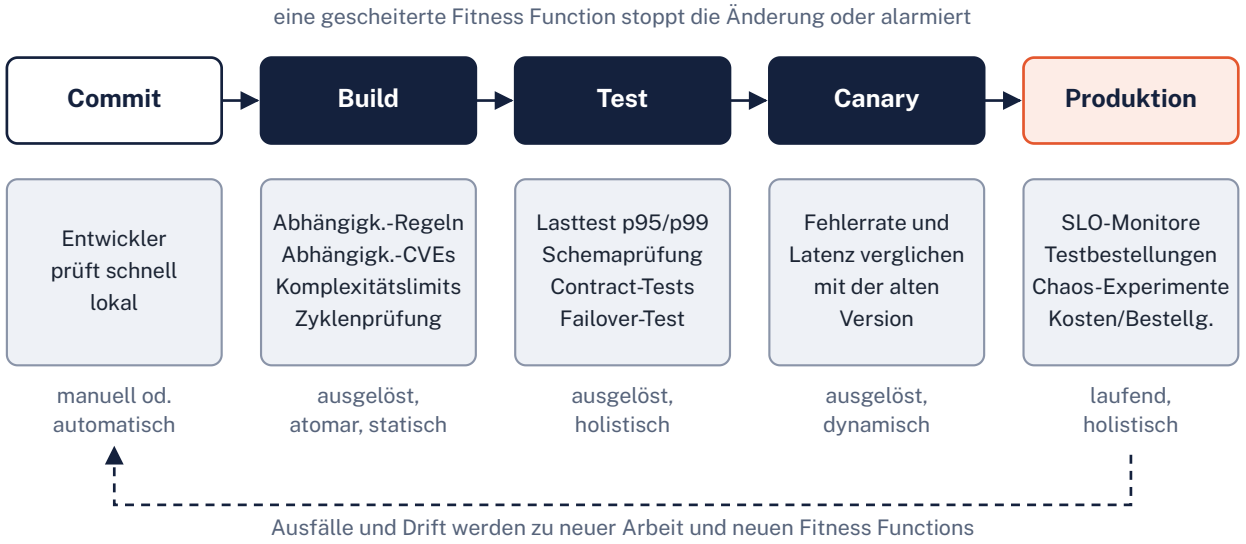
Ford und seine Mitautoren ordnen Fitness Functions entlang mehrerer unabhängiger Dimensionen ein. Die Einteilung ist weniger als Taxonomie nützlich denn als Denkanstoß beim Entwurf: Die Frage „Sollte das atomar oder holistisch, ausgelöst oder kontinuierlich sein?“ führt oft zu einer besseren Prüfung als der, die einem zuerst eingefallen ist.

Dimensionen von Fitness Functions

Dimension	Optionen	Was sie bedeutet	Beispiel
Bereich	Atomar oder holistisch	Atomar prüft ein Merkmal in einem Kontext; holistisch prüft mehrere Merkmale, die in einem gemeinsamen Kontext zusammenwirken	Atomar: keine Zyklen zwischen Paketen. Holistisch: Bestelllatenz, gemessen mit eingeschalteter Verschlüsselung, Audit-Logging und eingeschaltetem Caching
Wann sie läuft	Ausgelöst, kontinuierlich oder zeitgesteuert	Ausgelöst läuft bei einem Ereignis wie einem Commit, Build oder Deployment; kontinuierlich läuft ständig gegen das Live-System; zeitgesteuert schlägt nach Ablauf einer Frist an, als Erinnerung, etwas erneut zu prüfen	Ausgelöst: Abhängigkeitsscan bei jedem Build. Kontinuierlich: SLO-Burn-Rate-Alarm in Produktion. Zeitgesteuert: Build abbrechen, wenn eine fixierte Framework-Version älter als 12 Monate ist
Art des Ergebnisses	Statisch oder dynamisch	Statisch vergleicht mit einem festen Schwellenwert; dynamisch passt den Schwellenwert an den Kontext an, etwa an die Last	Statisch: p95 unter 1,5 s. Dynamisch: p95 darf über 150 Bestellungen/s auf 2 s steigen, wie für extreme Spitzen vereinbart
Wie sie ausgeführt wird	Automatisiert oder manuell	Automatisiert ist vorzuziehen; manche Merkmale, etwa rechtliche oder Usability-Anforderungen, brauchen weiterhin Menschen, idealerweise nach Zeitplan	Manuell: vierteljährliches Barrierefreiheits-Review gegen WCAG 2.2 AA
Planung	Geplant oder emergent	Geplante Prüfungen werden im Voraus aus bekannten Anforderungen entworfen; emergente kommen hinzu, während das Team lernt, was wichtig ist	Emergent: eine Prüfung auf Consumer-Lag, eingeführt nach einem Incident

Holistische Prüfungen verdienen Aufmerksamkeit, weil Qualitäten zusammenwirken: Sicherheitsmaßnahmen kosten Latenz, Caching erschwert die Aktualität der Daten, und ein atomarer Test jeder einzelnen kann bestehen, während die Kombination scheitert. Kontinuierliche Prüfungen sind wichtig, weil manche Merkmale nur in Produktion existieren: echte Nutzerlatenz, echte Ausfallarten, echte Kosten. Ein gesundes Portfolio kombiniert schnelle, atomare, ausgelöste Prüfungen im Build mit einer kleineren Zahl langsamerer holistischer Prüfungen vor dem Release und kontinuierlichen Prüfungen in Produktion.

Wo Fitness Functions laufen



Günstige Strukturprüfungen laufen bei jedem Build, holistische Prüfungen vor dem Release und kontinuierliche Prüfungen in Produktion, und Fehlschläge fließen als neue Arbeit zurück.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: wenn jede kleine Prüfung besteht

Der Punkt, an dem hier die meisten stolpern, ist, dass Prüfungen, die jede für sich bestehen, einen Fehler übersehen können, der erst auftritt, wenn alles zusammen läuft; deshalb braucht ein Team sowohl atomare als auch holistische Fitness Functions. Verfolgen Sie das Bestelllatenz-Szenario durch beide.

- Das Szenario der Essenslieferplattform besagt, dass bei der Freitagsspitze von 100 Bestellungen pro Sekunde 95 % der Bestellbestätigungen innerhalb von 1,5 Sekunden zurückkommen müssen.
- Angenommen, das Team schaltet über einige Monate Verschlüsselung und Audit-Logging auf dem Bestellpfad ein. Jede Änderung hat ihre eigene atomare Prüfung, die bestätigt, dass sie funktioniert und für sich gemessen nur wenig Zeit kostet. Jede Prüfung besteht.
- Aber die Zeit summiert sich. Mit beidem eingeschaltet unter Spitzenlast addieren sich ihre Kosten zu allem anderen auf dem Pfad, und, sagen wir, 6 % der Bestätigungen dauern jetzt länger als 1,5 Sekunden, mehr als die 5 %, die das Szenario erlaubt. Keine einzelne Prüfung war falsch; keine hat das Ganze betrachtet.
- Eine holistische Prüfung erkennt das: ein Lasttest, der 100 Bestellungen pro Sekunde an eine produktionsnahe Umgebung schickt, mit eingeschalteter Verschlüsselung, Audit-Logging und eingeschaltetem Caching, und der fehlschlägt, wenn das 95. Perzentil 1,5 Sekunden übersteigt.
- Holistische Prüfungen haben ihren Preis. Ein zehnminütiger Lasttest ist für jeden Build zu langsam, also läuft er nächtlich und vor jedem Release. Und ein Fehlschlag sagt, dass das Ganze zu langsam ist, nicht, welcher Teil das verursacht hat; das ausgearbeitete Beispiel weiter unten in diesem Abschnitt zeigt einen Weg darum herum.
- **Die Lehre:** Schnelle atomare Prüfungen bei jedem Build fangen die meisten Fehler günstig ab, sehen aber nicht, wie Qualitäten zusammenwirken. Ergänzen Sie weniger, langsamere holistische Prüfungen vor dem Release und kontinuierliche Prüfungen in Produktion, wo sich die Erfahrung echter Nutzer messen lässt.

Fitness Functions über die Qualitäten hinweg

Die meisten Teams beginnen mit der Struktur, weil Strukturprüfungen günstig, schnell und deterministisch sind, und ergänzen dann Prüfungen für Performance, Sicherheit, Resilienz und Daten, während die Pipeline reift. Die Tabelle nennt typische Beispiele; Werkzeuge ändern sich, die Muster nicht.

Beispiele nach Qualität

Qualität	Fitness Function	Art	Typische Werkzeuge
Struktur	Schichten- und Abhängigkeitsregeln, keine Zyklen, Module nutzen einander nur über öffentliche APIs (Abschnitt 3.4)	Atomar, ausgelöst, statisch	ArchUnit, import-linter, dependency-cruiser, Spring-Modulith-Verifikation
Performance	End-to-End-Lasttest mit Perzentil-Schwellenwerten; Latenzbudgets pro Service in der eigenen Pipeline jedes Service	Holistisch, ausgelöst	k6, Gatling, JMeter, Locust; Schwellenwerte, die den Job scheitern lassen
Sicherheit	Build abbrechen bei Abhängigkeiten mit bekannten hohen oder kritischen Schwachstellen; keine Secrets im Code; Container-Images nur aus der freigegebenen Registry	Atomar, ausgelöst	pip-audit, npm audit, OWASP Dependency-Check, Trivy, Secret-Scanner; Policy as Code (Abschnitt 11.4)
Resilienz	Chaos-Experimente, die Instanzen beenden oder Abhängigkeiten verlangsamen und prüfen, dass der stabile Zustand erhalten bleibt (Abschnitt 9.8)	Holistisch, ausgelöst oder kontinuierlich	Toxiproxy, Chaos Toolkit, Fault-Injection-Dienste der Cloud-Anbieter
Daten	Prüfungen der Schemakompatibilität für Events und APIs; Migrationen, an Daten in Produktionsgröße geprobt (Abschnitt 7.7)	Atomar, ausgelöst	Kompatibilitätsprüfungen der Schema Registry, buf breaking, Consumer-Driven-Contract-Tests
Betreibbarkeit und Kosten	Jeder Service stellt Health Checks und Standardmetriken bereit; die Kosten pro Bestellung bleiben unter einem Zielwert	Atomar oder kontinuierlich	Pipeline-Prüfungen für Service-Templates; Kosten-Dashboards mit Alarmen

Evolutionäre Architektur und inkrementelle Änderung

Fitness Functions sind die eine Hälfte dessen, was Ford, Parsons, Kua und Sadalage evolutionäre Architektur nennen: eine Architektur, die gelenkte, inkrementelle Änderung über mehrere Dimensionen unterstützt. Jeder Teil dieser Formulierung ist eine Entwurfsanforderung.

Inkrementelle Änderung hat eine Entwicklungs- und eine Betriebsseite. Entwickler müssen kleine Änderungen sicher vornehmen können; das hängt von Modularität und angemessener Kopplung ab (Kapitel 3) und davon, jeden unabhängig deploybaren Teil, das

Architekturquantum aus Abschnitt 5.1, klein genug zu halten, um ihn ohne Abstimmung mit anderen zu ändern. Der Betrieb muss kleine Änderungen häufig und sicher ausliefern können; das hängt von einer Deployment-Pipeline, schrittweisem Rollout und schnellem Rollback ab (Abschnitt 10.5). *Gelenkt* ist der Punkt, an dem Fitness Functions ins Spiel kommen: Sie definieren, was „fit“ bedeutet, und verhindern, dass Inkremente es verschlechtern. *Mehrere Dimensionen* erinnert daran, dass Architektur nicht nur Codestruktur ist: Daten, Sicherheit, Betrieb, Teamstruktur und Kosten entwickeln sich alle weiter und müssen alle geschützt werden.

Der Ansatz verändert, wie Architekten über die Zukunft denken. Statt jede Anforderung vorherzusagen und im Voraus Flexibilität dafür einzubauen, was, wie Abschnitt 3.8 gezeigt hat, teuer und oft falsch ist, hält das Team das System leicht änderbar und schützt nur die Merkmale, die zählen, während es andere Entscheidungen auf den Last Responsible Moment verschiebt (Abschnitt 1.6). Kritiker weisen auf echte Grenzen hin. Nicht jede Qualität lässt sich automatisch oder günstig prüfen; schlecht gewählte Schwellenwerte erzeugen Rauschen, das Teams zu ignorieren lernen; und manche Entscheidungen, etwa für eine Datenbank oder einen Cloud-Anbieter, bleiben Einwegtüren, wie gut die Pipeline auch ist. Evolutionäre Architektur ist eine Disziplin, die umkehrbaren Teile umkehrbar und die wichtigen Teile geschützt zu halten, kein Versprechen, dass sich später alles ändern lässt.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: drei Szenarien aus Kapitel 2 als Fitness Functions

Abschnitt 2.8 hat einen Utility Tree für die Essenslieferplattform erstellt. Das Team macht aus drei seiner Blätter Fitness Functions, eine ausgelöst und holistisch, eine ein Chaos-Experiment, eine strukturell, nachdem die ATAM in Abschnitt 11.1 zur asynchronen Zahlungsautorisierung geführt hat.

Szenario 1: Bestelllatenz (H,H)	Bei der Freitagsspitze von 100 Bestellungen/s kommen 95 % der Bestellbestätigungen innerhalb von 1,5 s und 99 % innerhalb von 3 s zurück
Fitness Function 1	Holistisch, ausgelöst: ein k6-Lasttest gegen eine produktionsnahe Umgebung, nächtlich und vor jedem Release, der fehlschlägt, wenn p95 oder p99 den Schwellenwert übersteigt oder mehr als 0,1 % der Anfragen fehlschlagen. Ergänzt durch kontinuierliches Monitoring derselben Perzentile aus echtem Verkehr in Produktion
Latenzbudgets	Die 1,5 s werden in Budgets aufgeteilt: Mobilfunknetz und TLS 400 ms, Gateway 50 ms, Bestellungen 250 ms, Preisgestaltung 100 ms, Datenbank-Schreibvorgänge 150 ms; bleiben 550 ms Kapazitätsreserve, seit die Kartenautorisierung nicht mehr auf dem Pfad liegt. Die Pipeline jedes Service führt einen atomaren Komponenten-Lasttest gegen das eigene Budget aus, sodass eine Regression in dem Service erkannt wird, der sie verursacht hat
Szenario 2: Verfügbarkeit des Checkouts (H,H)	Wenn beim Zahlungsanbieter während der Spitze ein Timeout auftritt, werden Bestellungen angenommen und in eine Queue gestellt, keine bestätigte Bestellung geht verloren, und die Kunden werden innerhalb von 10 s informiert
Fitness Function 2	Holistisches Chaos-Experiment, wöchentlich im Staging und vierteljährlich als Game Day in Produktion mit kleinem Blast Radius: Der Anbieter wird zum Hängen gebracht; Bestellungen müssen innerhalb von 10 s als ausstehend angenommen werden und nach der Wiederherstellung alle abgeschlossen werden. Verantwortlich: Bestellteam
Szenario 3: neuer Kassensystem-Anbieter (M,L)	Ein neuer Anbieter von Restaurant-Kassensystemen wird in 5 Personentagen integriert, indem ein Adapter hinzugefügt wird
Fitness Function 3	Atomar, ausgelöst, statisch: Ein import-linter-Contract verbietet den Restaurant-Domänen-und API-Modulen sowie dem Bestellpaket, ein Anbieter-SDK zu importieren, und ein zweiter Contract hindert Adapter daran, einander zu importieren, sodass die Änderung lokal bleibt. Ergänzt durch eine manuelle Prüfung: Der Aufwand jeder echten Anbieterintegration wird erfasst und mit dem Ziel von 5 Tagen verglichen
Zu jeder dokumentiert	Quellszenario, Schwellenwert, Verantwortlicher, das erklärende ADR und ein Review-Termin

Jedes Antwortmaß wurde ohne Umdeutung zu einem Schwellenwert; das ist der Ertrag, wenn man Szenarien von Anfang an gut formuliert.

Das Änderbarkeitsszenario lässt sich nicht vollständig automatisieren: Die Strukturregel schützt die Voraussetzungen für eine Änderung in 5 Tagen, und die manuelle Messung prüft das Ergebnis.

Latenzbudgets machen aus einer holistischen Zahl atomare Prüfungen, die das richtige Team über eine Regression informieren.

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

export const options = {
  scenarios: {
    friday_peak: { executor: 'constant-arrival-rate', rate: 100, timeUnit: '1s',
                   duration: '10m', preAllocatedVUs: 300 },   // 100 orders per second
  },
  thresholds: {   // the scenario's response measure, as pass/fail rules
    'http_req_duration{name:place_order}': ['p(95)<1500', 'p(99)<3000'],
    'http_req_failed{name:place_order}': ['rate<0.001'],
  },
};

const order = { restaurantId: 'r-42', items: [{ sku: 'pad-thai', qty: 1 }] };

export default function () {
  const res = http.post(`${__ENV.BASE_URL}/orders`, JSON.stringify(order), {
    headers: { 'Content-Type': 'application/json' }, tags: { name: 'place_order' } });
  check(res, { 'order accepted': (r) => r.status === 201 || r.status === 202 });
}
```

Fitness Function 1: ein k6-Lasttest, dessen Schwellenwerte das Antwortmaß des Szenarios sind. k6 beendet sich mit einem Fehler, wenn ein Schwellenwert verletzt ist, und lässt damit den Pipeline-Job scheitern.

```
# Fitness function: checkout keeps accepting orders when payments hang (illustrative)
# Runs weekly in staging; toxiproxy, api and ledger are test fixtures.
def test_orders_survive_payment_provider_outage(toxiproxy, api, ledger):
    toxiproxy.add_latency("payment-provider", latency_ms=30_000)   # provider hangs
    placed = [api.place_order(sample_order()) for _ in range(200)]

    assert all(r.status in (201, 202) for r in placed)           # still accepted
    assert all(r.json()["state"] == "PENDING_PAYMENT" for r in placed)
    assert max(r.elapsed_seconds for r in placed) < 10           # told within 10 s

    toxiproxy.remove_toxics("payment-provider")                   # provider recovers
    ledger.wait_until_settled(timeout_seconds=300)
    assert ledger.lost_orders(placed) == []                       # none lost
```

Fitness Function 2: ein Chaos-Experiment, als Test geschrieben (zur Veranschaulichung; die Fixtures kapseln einen Fault-Injection-Proxy, die API und das Zahlungsjournal).

```
# .importlinter: only the POS adapters may use vendor SDKs
[importlinter]
root_package = fooddelivery
include_external_packages = True

[importlinter:contract:pos-vendors-isolated]
name = Vendor POS SDKs are used only inside the POS adapters
type = forbidden
source_modules =
    fooddelivery.restaurant.domain
    fooddelivery.restaurant.api
    fooddelivery.ordering
forbidden_modules =
    tillpro_sdk
    kitchenlink

[importlinter:contract:adapters-independent]
name = POS adapters do not import each other
type = independence
modules =
    fooddelivery.restaurant.pos_adapters.tillpro
    fooddelivery.restaurant.pos_adapters.kitchenlink
```

Fitness Function 3: import-linter-Contracts, die anbieterspezifischen Code in den Adaptern halten, bei jedem Build mit lint-imports ausgeführt.

Die Datenkompatibilität wird auf dieselbe Weise geschützt. Bevor eine Änderung am Event `OrderPlaced` gemergt werden kann, fragt die Pipeline die Schema Registry, ob das neue Schema nach der Kompatibilitätsregel des Subjects mit der zuletzt registrierten Version kompatibel ist, und schlägt fehl, wenn nicht:

```
# Data fitness function: block the merge if the new event schema would break consumers
curl -s -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
  --data @build/order-placed-v7.request.json \
  "$SCHEMA_REGISTRY_URL/compatibility/subjects/order-placed-value/versions/latest" \
  | jq -e '.is_compatible == true'
```

Eine Daten-Fitness-Function, die den Kompatibilitäts-Endpoint einer Schema Registry nutzt; jq -e liefert einen Exit-Code ungleich null und lässt den Job scheitern, wenn das Schema inkompatibel ist.

Fitness Functions entwerfen, die Bestand haben

Beginnen Sie klein und mit dem Wichtigen. Wählen Sie die drei bis fünf Merkmale, die der Utility Tree am höchsten bewertet, schreiben Sie für jedes eine Fitness Function und sorgen Sie dafür, dass sie schnell und zuverlässig sind, bevor Sie weitere hinzufügen. Ein Dutzend vertrauenswürdiger Prüfungen ist mehr wert als hundert, die zufällig fehlschlagen.

Nutzen Sie bei bestehenden Systemen eine Ratsche. Verletzt eine Codebasis eine Regel bereits, erfassen Sie die aktuellen Verstöße als Baseline und lassen Sie nur neue scheitern; verschärfen Sie die Baseline, während das Team die Schulden abbaut. Dasselbe funktioniert für die Performance: Bei Regressionen über eine vereinbarte Marge gegenüber dem letzten Release scheitern, während man auf das Ziel des Szenarios hinarbeitet.

Machen Sie Schwellenwerte explizit und geben Sie ihnen Verantwortliche. Halten Sie Schwellenwerte in versionierter Konfiguration neben einem Verweis auf das ADR, das sie begründet, mit einem namentlich genannten Verantwortlichen. Schlägt eine Fitness Function fehl, entscheidet der Verantwortliche zwischen Codekorrektur und Regeländerung, und die Entscheidung wird dokumentiert. Behandeln Sie eine Fitness Function, die häufig übergangen wird, als Befund: Entweder ist die Regel falsch, oder die Architektur steht unter echtem Druck.

Halten Sie die Pipeline schnell. Atomare Prüfungen gehören in jeden Build; schwere holistische Prüfungen wie zehnminütige Lasttests laufen nächtlich, vor dem Release oder auf dem Release Candidate. Kontinuierliche Prüfungen in Produktion brauchen dieselbe Sorgfalt wie jeder Alarm: an Auswirkungen auf Nutzer gekoppelt, mit einem Runbook und still, wenn nichts falsch ist.

ACHTUNG

Anti-Patterns bei Fitness Functions

- Unzuverlässige Prüfungen: Ein Lasttest auf geteilter, verrauschter Infrastruktur schlägt zufällig fehl, also lernen Teams, ihn so lange neu zu starten, bis er besteht, und glauben ihm nicht mehr.
- Die Messgröße austricksen: Wird eine Zahl zum Ziel, optimieren Menschen die Zahl (Goodharts Gesetz, Abschnitt 3.8); ziehen Sie Prüfungen, die an echte Szenarien gebunden sind, abstrakten Metriken vor.
- Schwellenwerte ohne Begründung: Niemand weiß, warum das Limit 15 ist, also traut sich niemand, es zu ändern, oder alle ignorieren es.
- Prüfungen, die nur berichten: Ein Dashboard, das niemand beobachtet, ist keine Fitness Function.
- Das Einfache automatisieren und das Wichtige ignorieren: Dutzende Stilregeln, aber keine Prüfung der Latenz auf dem Bestellpfad oder des Zahlungs-Fallbacks.
- Fitness Functions als Ersatz für Nachdenken: Sie schützen bereits getroffene Entscheidungen; sie sagen Ihnen nicht, welche Entscheidungen Sie treffen sollen.

Häufige Fragen

Sind Fitness Functions einfach Tests?
Viele sind als Tests umgesetzt, aber ihr Gegenstand ist ein anderer. Gewöhnliche Tests prüfen, dass sich Features korrekt verhalten; Fitness Functions prüfen Architekturmerkmale wie Struktur, Latenz, Resilienz oder Kompatibilität. Manche, etwa Monitore in Produktion und manuelle Reviews, sind überhaupt keine Tests.

Wer schreibt Fitness Functions: Architekten oder Entwickler?
Beide. Architekten bestimmen meist die Merkmale und Schwellenwerte und schreiben oft die ersten Prüfungen, was ein guter Weg ist, praktisch beteiligt zu bleiben (Abschnitt 1.4). Die Teams besitzen und pflegen sie wie jeden anderen Code.

Wie hängen Fitness Functions mit SLOs zusammen?
Ein SLO mit einem Alarm auf sein Error Budget ist eine kontinuierliche, holistische Fitness Function für Verfügbarkeit oder Latenz in Produktion (Abschnitt 9.5). Fitness Functions in der Pipeline versuchen, dieselben Regressionen abzufangen, bevor sie die Nutzer erreichen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Was ist der Unterschied zwischen einer Metrik und einer Fitness Function?

Eine Metrik beschreibt eine Eigenschaft; eine Fitness Function kodiert eine Entscheidung mit einem Schwellenwert und einer Aktion, sodass ihr Bruch einen Build scheitern lässt oder einen Alarm auslöst und eine bewusste Wahl erzwingt.

▼ Nennen Sie je ein Beispiel für eine atomare und eine holistische Fitness Function.

Atomar: eine Regel, die Importe aus der Web-Schicht in die Domäne verbietet. Holistisch: ein Lasttest, der die Bestelllatenz mit aktivierter Sicherheit, aktiviertem Logging und Caching misst.

▼ Was fügt ein Latenzbudget einer End-to-End-Latenz-Fitness-Function hinzu?

Es teilt das End-to-End-Ziel in Limits pro Service auf, die jeder Service in seiner eigenen Pipeline prüft, sodass eine Regression in dem Service erkannt wird, der sie verursacht hat.

▼ Was sind die drei Teile der Definition von evolutionärer Architektur?

Sie unterstützt gelenkte (durch Fitness Functions), inkrementelle (kleine, sichere Änderungen und Releases) Änderung über mehrere Dimensionen (Code, Daten, Sicherheit, Betrieb, Teams und mehr).

▼ Probieren Sie es aus: Machen Sie aus beiden Aussagen jeweils eine Fitness Function mit Auslöser und Schwellenwert: „95 % der Bestellbestätigungen innerhalb von 1,5 s bei 100 Bestellungen/s“ und „Bestellungen darf niemals die Interna von Zahlungen importieren“. Wo würde jede laufen, und was passiert, wenn eine fehlschlägt?

Latenz: ein holistischer, dynamischer Lasttest mit 100 Bestellungen pro Sekunde, der fehlschlägt, wenn p95 1,5 s übersteigt, und nächtlich oder vor dem Release läuft, weil er langsam ist. Abhängigkeitsregel: eine atomare, statische Prüfung mit ArchUnit oder import-linter, die bei jedem verbotenen Import fehlschlägt und bei jedem Build läuft. Ein Fehlschlag erzwingt eine Entscheidung: den Code beheben oder die Regel ändern und den Grund dokumentieren. Jede braucht eine verantwortliche Person und einen Verweis auf ihr Szenario.

QUELLEN

- Building Evolutionary Architectures, 2nd edition (O'Reilly): https://oreilly.com/library/view/building-evolutionary-architectures/9781492097532
- Evolutionary Architecture: book site with fitness function katas: https://evolutionaryarchitecture.com/

- Grafana k6 documentation: Thresholds: https://grafana.com/docs/k6/latest/using-k6/thresholds/
- import-linter documentation: Forbidden contracts: https://import-linter.readthedocs.io/en/stable/contract_types/forbidden/

Architektur-Governance in agilen Organisationen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.6 Entscheidungen und Abwägungen
- 4.5 Entscheidungsdokumentation
- 11.3 Fitness Functions

Begriffe von früher, die er verwendet: Architecture Decision Record (ADR), Fitness Function, Unternehmensarchitektur

DIE IDEE IN KÜRZE

Eine Stadt kann nicht jede neue Küche vom Bürgermeister genehmigen lassen. Stattdessen legt sie einige Prinzipien fest (die Straßen bleiben fußgängerfreundlich), einige Bauvorschriften, die Prüfer automatisch kontrollieren, und eine kleine Zahl von Entscheidungen, die wirklich einen Ausschuss brauchen, etwa eine neue Brücke. Die Bewohner können das meiste selbst tun, aber bei allem, was die Nachbarn betrifft, wird erwartet, dass sie zuerst die Nachbarn fragen. Architektur-Governance in agilen Organisationen funktioniert genauso: Die Teams entscheiden das meiste selbst, automatische Prüfungen setzen die Regeln durch, die eine Maschine prüfen kann, und größere Entscheidungen werden getroffen, nachdem die Betroffenen gefragt wurden.

WAS SIE WISSEN MÜSSEN

- Architektur-Governance stellt sicher, dass bedeutende Entscheidungen gut, im Einklang mit den Zielen der Organisation getroffen und tatsächlich befolgt werden; zu ihren Zwecken gehören Ausrichtung, Risiko und Compliance, die Begrenzung unnötiger Vielfalt, der Schutz gemeinsamer Qualitäten und die Verbreitung von Erfahrungen.
- Prinzipien geben dauerhafte Richtung, Standards machen bestimmte Entscheidungen verbindlich, Guardrails halten die Arbeit automatisch innerhalb von Grenzen, und Gates halten die Arbeit für eine Freigabe an; bevorzugen Sie Guardrails, wo Regeln objektiv sind, und halten Sie Gates früh und selten.
- Prinzipien im TOGAF-Stil haben einen Namen, eine Aussage, eine Begründung und Auswirkungen; ein gutes Prinzip könnte vernünftigerweise auch anders gewählt werden und entscheidet echte Fragen.

- Im Advice Process (Andrew Harmel-Law, 2021) darf jede Person eine Architekturentscheidung treffen, nachdem sie Rat bei den Betroffenen und bei Fachleuten eingeholt hat, dokumentiert in einem ADR; ein Advisory Forum schafft Sichtbarkeit, entscheidet aber nicht.
- Ein Technologieradar ordnet Technologien Ringen wie Adopt, Trial, Assess und Hold zu, mit schriftlicher Begründung; ein internes Radar hält die eigenen Erfahrungen der Organisation fest und lenkt neue Entscheidungen.
- Policy as Code (zum Beispiel Open Policy Agent mit Rego) und Fitness Functions automatisieren die Compliance und liefern Prüfnachweise als Nebenprodukt.

Wozu Architektur-Governance dient

Architektur-Governance ist die Gesamtheit der Praktiken, mit denen eine Organisation sicherstellt, dass ihre bedeutenden technischen Entscheidungen gut getroffen werden, mit ihren Zielen übereinstimmen und tatsächlich befolgt werden. Sie beantwortet vier Fragen: Wer darf welche Entscheidungen treffen, wie sollen sie getroffen werden, welche Richtung sollen sie nehmen, und woran erkennt man, ob das Ergebnis konform ist?

Die Zwecke sind praktisch. Governance richtet die Technologie an der Strategie aus, damit fünfzig Teams nicht fünfzig Arten wählen, Kundendaten zu speichern. Sie steuert Risiko und Compliance: Datenschutz, Sicherheit, Barrierefreiheit und Branchenregulierung verlangen alle Nachweise, dass Regeln befolgt werden. Sie begrenzt unnötige Vielfalt, denn jede zusätzliche Datenbank, Sprache oder Messaging-Lösung braucht Fachwissen, Patches und Rufbereitschaft. Sie schützt Qualitäten, die keinem einzelnen Team gehören, etwa End-to-End-Latenz, Sicherheit und Kosten. Und sie verbreitet Erfahrungen, damit eine Lektion, die ein Team teuer bezahlt hat, nicht vom nächsten noch einmal bezahlt wird.

Agile Organisationen spüren hier ein Spannungsfeld. Das Prinzip des Agilen Manifests, dass die besten Architekturen aus selbstorganisierten Teams hervorgehen (Abschnitt 1.8), zieht in Richtung Autonomie; Größe, Regulierung und gemeinsame Infrastruktur ziehen in Richtung Ausrichtung. Henrik Knibergs Beschreibungen der Engineering-Kultur von Spotify haben für dieses Ziel den Ausdruck *Aligned Autonomy* bekannt gemacht: Teams, die frei über das Wie entscheiden, innerhalb eines gemeinsamen Verständnisses von Was und Warum. Governance ist die Maschinerie, die das möglich macht. Schlecht gemacht, wird sie zum langsamen Genehmigungsprozess aus Abschnitt 11.2; gut gemacht, bemerken die Teams sie kaum, weil das Richtige zugleich das Einfache ist.

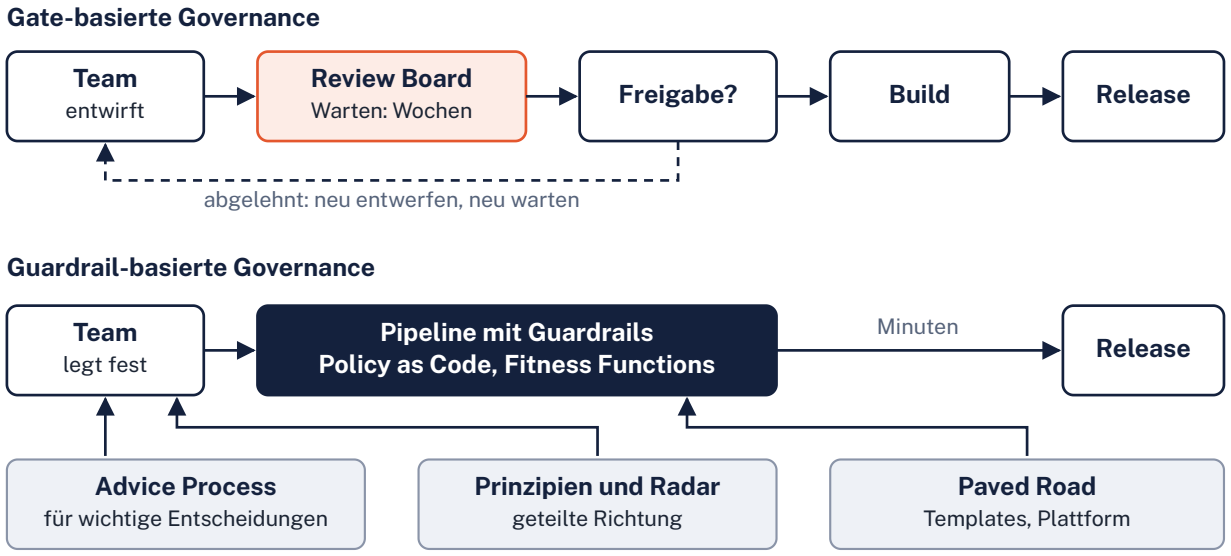
Prinzipien, Standards, Guardrails und Gates

Governance nutzt vier Arten von Instrumenten, die sich darin unterscheiden, wie stark sie einschränken und wann sie wirken. Prinzipien drücken eine dauerhafte Richtung aus und lassen Raum für Urteilsvermögen. Standards sind konkrete, verbindliche Festlegungen (alle Services liefern OpenTelemetry-Traces; alle öffentlichen APIs sind in OpenAPI beschrieben). Guardrails halten Teams während der Arbeit innerhalb sicherer Grenzen, meist automatisch und kontinuierlich. Gates halten die Arbeit an, bis jemand sie freigibt.

Instrumente der Governance

Instrument	Beispiel	Stärke	Risiko
Prinzip	Jede Art von Geschäftsdaten hat genau einen besitzenden Service	Leitet viele Entscheidungen, auch unvorhergesehene	Zu vage, um danach zu handeln, oder in der Praxis ignoriert
Standard	Alle Services liefern OpenTelemetry-Traces und strukturierte JSON-Logs	Konsistenz, gemeinsame Werkzeuge, leichtes Onboarding	Veraltet; blockiert bessere Optionen; Ausnahmen häufen sich
Guardrail	Die Pipeline weist Container-Images aus nicht freigegebenen Registries zurück	Kontinuierlich, schnell, objektiv, keine Warteschlange	Funktioniert nur für Regeln, die eine Maschine prüfen kann; lässt sich umgehen, wenn niemand verantwortlich ist
Gate	Sicherheitsfreigabe, bevor eine neue Kategorie personenbezogener Daten verarbeitet wird	Verantwortliches menschliches Urteil für folgenreiche, irreversible Entscheidungen	Verzögerung, spätes Feedback, Umgehung und Abnicken

Gates und Guardrails im Vergleich



Ein Gate lässt jeden bedeutenden Entwurf auf eine Freigabe warten; Guardrails prüfen objektive Regeln kontinuierlich und lassen den Menschen Zeit für die Entscheidungen, die Urteilsvermögen brauchen.

Der heutige Konsens, wenn auch kein allgemeiner, ist, Guardrails gegenüber Gates zu bevorzugen, wo immer sich eine Regel objektiv formulieren lässt, Gates für die wenigen Entscheidungen vorzubehalten, bei denen wirklich ein verantwortliches menschliches Urteil nötig ist, und diese Gates früh und zeitlich begrenzt abzuhalten. Eine Paved Road (der Begriff von Netflix; Spotify spricht von einem Golden Path) ergänzt beides: Wenn der unterstützte Weg, einen Service zu bauen, bereits freigegebene Images, Logging, Sicherheitsscans und Deployment enthält, halten sich die meisten Teams an die Regeln, ohne dass man es ihnen sagen muss, und die Governance kann sich auf bewusste Abweichungen konzentrieren. Abschnitt 10.7 behandelt die Plattformen, die solche Wege bereitstellen.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: eine Regel als Gate und als Guardrail

Die schwierigste Idee dieses Abschnitts ist, für jede Regel das richtige Instrument zu wählen: warum eine Regel, die eine Maschine prüfen kann, in einen Guardrail gehört und warum einige Entscheidungen trotzdem ein Gate brauchen. Verfolgen Sie zwei Regeln aus der Tabelle oben.

- Nehmen Sie die Regel, dass jedes Container-Image aus der gescannten Registry der Organisation stammen muss. Angenommen, sie wird durch ein Gate durchgesetzt: Jedes Team listet seine Images auf, damit das monatliche Review Board sie freigibt.
- Das Team wartet wochenlang auf einen Termin. Das Board prüft die Liste einmal, sodass ein im Folgemonat geändertes Image nie geprüft wird. Und weil die Warteschlange langsam ist, lernen die Teams, Änderungen als geringfügig auszugeben, um sie zu umgehen, wie Abschnitt 11.2 beschrieben hat.
- Setzen Sie dieselbe Regel nun als Guardrail durch: Die Pipeline weist jedes Deployment zurück, dessen Image von anderswo stammt. Sie prüft jedes Release, in Minuten, ohne Warteschlange, und ihr Log ist ein Nachweis für Prüfer.
- Das funktioniert, weil die Regel objektiv ist: Eine Maschine kann mit Ja oder Nein antworten. Der Guardrail braucht trotzdem einen Verantwortlichen und einen Weg für Ausnahmen, sonst finden die Leute Wege, ihn zu umgehen.
- Nehmen Sie eine zweite Regel: eine Sicherheitsfreigabe, bevor die Plattform eine neue Kategorie personenbezogener Daten verarbeitet. Keine Maschine kann beurteilen, ob Zweck und Risiken vertretbar sind, also muss das eine verantwortliche Person tun, und ein Gate ist richtig. Halten Sie es früh ab, begrenzen Sie es zeitlich und legen Sie im Voraus fest, welche Nachweise es braucht.
- **Die Lehre:** Wenn eine Maschine eine Regel prüfen kann, machen Sie daraus einen Guardrail; behalten Sie Gates den wenigen folgenreichen Entscheidungen vor, die ein verantwortliches menschliches Urteil brauchen, und halten Sie sie früh ab. Entscheidungen, die Urteilsvermögen, aber keine Freigabe brauchen, laufen über den Advice Process, weiter unten in diesem Abschnitt.

Architekturprinzipien formulieren

TOGAF, das Enterprise-Architecture-Framework der Open Group (the TOGAF Standard, 10th Edition, 2022), bietet eine weit verbreitete Vorlage für Prinzipien mit vier Teilen. Der **Name** ist kurz und einprägsam. Die **Aussage** formuliert die Regel in ein oder zwei eindeutigen Sätzen. Die **Begründung** erklärt den geschäftlichen Nutzen so, dass ein nicht technischer Auftraggeber ihn akzeptieren könnte. Die **Auswirkungen** sagen, was das Prinzip in der Praxis

verlangt: was getan werden muss, was es kostet und was es ausschließt. TOGAF nennt außerdem Kriterien für einen guten Satz von Prinzipien: verständlich, robust (präzise genug, um schwierige Fälle zu entscheiden), vollständig, konsistent und stabil.

Zwei Tests entlarven die meisten schwachen Prinzipien. Erstens: Könnte eine vernünftige Organisation das Gegenteil wählen? „Systeme sollen sicher sein“ fällt durch; niemand würde Unsicherheit wählen. „Für generische Fähigkeiten Kaufen dem Selbstbauen vorziehen“ besteht, weil manche Organisationen zu Recht lieber selbst bauen. Zweitens: Ändert das Prinzip eine echte Entscheidung? Kann niemand eine Wahl nennen, die es entscheiden würde, ist es ein Slogan. Andrew Harmel-Law fordert einen Satz, der weder zu klein ist, um die wichtigsten Bedürfnisse abzudecken, noch zu groß, als dass die Teams ihn sich merken könnten, aus den Teams selbst stammend und an die Geschäftsstrategie gebunden, statt allgemeiner Programmiermaximen oder Praktiken. Weniger, schärfere Prinzipien werden behalten und genutzt; lange Listen nicht.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: drei Prinzipien für die Essenslieferplattform

Von den Architekten mit den Teamleitungen verfasst, im Advisory Forum diskutiert und zusammen mit dem Radar veröffentlicht. Jedes folgt der TOGAF-Struktur.

P1 Name	**Ein Besitzer für jede Tatsache**
Aussage	Jede Art von Geschäftsdaten (Bestellungen, Speisekarten, Kurierpositionen, Zahlungen) hat genau einen besitzenden Service, der als einziger schreibt; andere nutzen seine API oder seine Events.
Begründung	Gemeinsam beschreibbare Daten koppeln die Releases der Teams und machen Fehler unauffindbar (Abschnitt 7.1). Klare Zuständigkeit lässt Teams ihre Schemas ändern und unabhängig deployen, und nur so liefert das Unternehmen wöchentlich aus.
Auswirkungen	Kein Service liest oder schreibt die Tabellen eines anderen. Das Reporting nutzt veröffentlichte Events oder ein Data Warehouse. Neue Daten brauchen einen benannten Besitzer, bevor sie gespeichert werden. Eine Fitness Function prüft die Datenbank-Zugangsdaten pro Service.
P2 Name	**Asynchron zwischen Domänen**
Aussage	Domänen integrieren sich standardmäßig über veröffentlichte Events; synchrone Aufrufe über Domänengrenzen werden nur dort genutzt, wo der Nutzer auf die Antwort wartet.
Begründung	Synchrone Ketten vervielfachen die Nichtverfügbarkeit und erhöhen die Latenz (Abschnitt 6.1); die Freitagsspitze und Ausfälle bei Partnern sind die größten betrieblichen Risiken des Unternehmens.
Auswirkungen	Teams entwerfen für Eventual Consistency und idempotente Consumer. Jeder synchrone domänenübergreifende Aufruf hat ein Timeout, einen Fallback und ein Latenzbudget. Mehr betriebliche Investition in Broker und Tracing.
P3 Name	**Generisches kaufen, Kern bauen, portabel bleiben**
Aussage	Generische Fähigkeiten (Identität, Zahlungen, E-Mail, Monitoring) werden gekauft oder als Managed Services genutzt; differenzierende Fähigkeiten (Optimierung der Kurierdisposition, Restaurant-Werkzeuge) werden selbst gebaut. Die Geschäftslogik bleibt hinter Ports, nicht in proprietären Diensten eines Anbieters.
Begründung	Entwicklungszeit ist die knappste Ressource und sollte dorthin fließen, wo das Unternehmen im Wettbewerb steht (Kern-Subdomänen, Abschnitt 5.7), unter Vermeidung von Lock-in, der die Verhandlungsposition schwächen würde (Abschnitt 10.8).

Auswirkungen	Kaufen oder Selbstbauen ist eine ausdrückliche Frage in jedem relevanten ADR. Auf Managed Services wird über Adapter zugegriffen. Manche Managed Features mit starkem Lock-in werden abgelehnt, auch wenn sie bequem wären.

Jedes Prinzip entscheidet echte, wiederkehrende Fragen, und eine vernünftige Organisation könnte anders wählen.

Die Auswirkungen umfassen Kosten, nicht nur Nutzen.

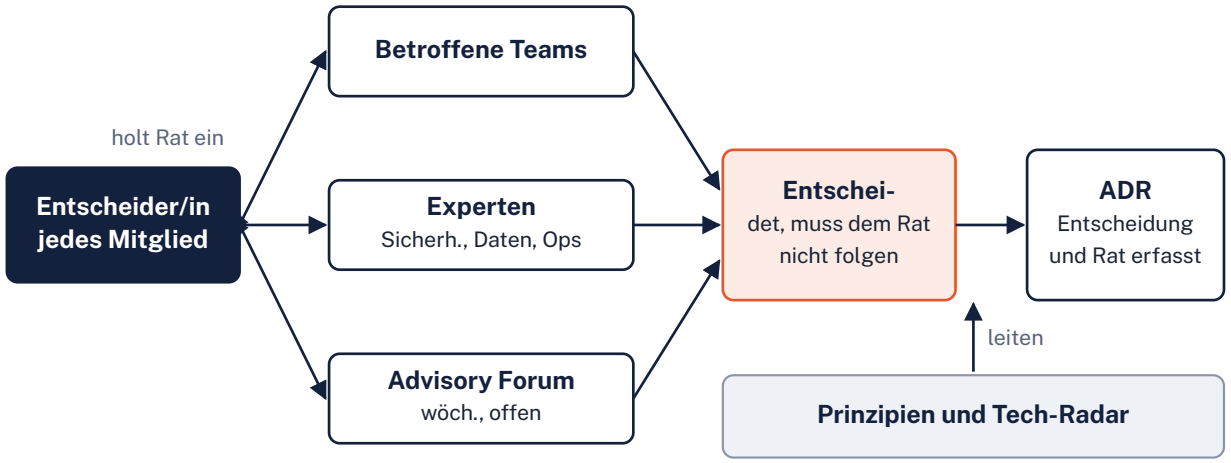
P1 und P2 lassen sich teilweise als Guardrails durchsetzen; P3 braucht menschliches Urteil und wird daher über den Advice Process angewandt.

Dezentrale Entscheidungsfindung: der Advice Process

Wenn Entscheidungen nicht mehr über ein Board laufen, wer trifft sie dann? Andrew Harmel-Laws Artikel *Scaling the Practice of Architecture, Conversationally* (martinfowler.com, Dezember 2021) schlug eine Antwort vor, die von selbstorganisierten Unternehmen entlehnt ist: den Advice Process. Seine einzige Regel lautet, dass jede Person eine Architekturentscheidung treffen darf, sofern sie vor der Entscheidung Rat bei allen einholt, die spürbar betroffen sein werden, und bei Menschen mit einschlägiger Expertise. Wer entscheidet, muss den Rat anhören und dokumentieren, ihm aber nicht folgen; das Ziel ist breiter Input, kein Konsens. Die Verantwortung bleibt bei der Person oder dem Team, die entscheiden, und genau das lässt die Regel funktionieren: Wer weiß, dass er die Folgen trägt, holt ernsthaft Rat ein.

Harmel-Law beschreibt vier unterstützende Elemente. Entscheidungsdokumente (ADRs, Abschnitt 4.5) sind das Werkzeug zum Nachdenken und Festhalten und enthalten den erhaltenen Rat, von wem und wann. Ein Architecture Advisory Forum, meist wöchentlich für etwa eine Stunde und offen für alle, macht neue Spikes, vorgeschlagene und erneut aufgegriffene Entscheidungen sichtbar; entscheidend ist, dass es berät, aber nicht entscheidet. Von den Teams erarbeitete Architekturprinzipien geben eine gemeinsame Richtung. Und ein internes Technologieradar zeigt die aktuelle Technologielandschaft der Organisation und ihre Haltung dazu.

Der Advice Process



Rat fließt von den Betroffenen und von Fachleuten ein; Entscheidung und Rat werden zusammen dokumentiert, und die Verantwortung bleibt bei der entscheidenden Person.

Der Advice Process hat Schwachstellen, die man kennen sollte. Wer entscheidet, fragt vielleicht nur die, die wahrscheinlich zustimmen; eine veröffentlichte Liste, wer bei welcher Art von Entscheidung konsultiert werden muss (die Sicherheit bei allem, was personenbezogene Daten berührt, das Plattformteam bei neuer Infrastruktur), hilft. Rat wird vielleicht zu spät eingeholt, wenn der Code schon geschrieben ist; es hilft, den vorgeschlagenen ADR zum Anfang der Arbeit zu machen, nicht zu ihrem Ende. Und manche finden es anfangs unbequem, dass das Forum nicht entscheidet. In regulierten Umgebungen kann der Prozess neben einer kleinen Zahl formaler Gates bestehen, die nur dort angewandt werden, wo das Gesetz oder die Risikobereitschaft der Organisation sie verlangt.

Technologieradare

Thoughtworks veröffentlicht sein Technology Radar seit 2010 zweimal im Jahr. Jede Ausgabe ordnet rund hundert Blips (Techniken, Werkzeuge, Plattformen sowie Sprachen und Frameworks: die vier Quadranten) Ringen zu, die eine Empfehlung ausdrücken. Adopt bedeutet bewährt und empfohlen; Trial bedeutet, es lohnt sich, das in einem Projekt zu verfolgen, das mit dem Risiko umgehen kann; Assess bedeutet, es lohnt sich zu erkunden, um die Wirkung zu verstehen; und der äußerste Ring, der den größten Teil der Geschichte des Radars Hold hieß und in den jüngsten Ausgaben Caution heißt, bedeutet: mit Vorsicht vorgehen. Jeder Blip hat eine kurze schriftliche Begründung, und die ist der wertvollste Teil.

Viele Organisationen bauen ihr eigenes Radar, für sich selbst statt für die Branche. Thoughtworks stellt mit Build Your Own Radar ein Open-Source-Werkzeug bereit, das ein Radar aus einer Tabelle oder CSV-Datei mit den Spalten name, ring, quadrant, isNew und

description erzeugt. Der Wert liegt weniger im Bild als im Prozess: Teams schlagen Blips vor, eine kleine Gruppe mit Vertretern vieler Teams diskutiert sie, die Ringe werden mit schriftlicher Begründung vergeben, und das Radar wird jedes Quartal oder Halbjahr aktualisiert. Ein Hold-Ring ist ein sanftes, aber wirksames Governance-Werkzeug: Er sagt „keine neuen Einsätze“, ohne zu verbieten, was bereits existiert. Ein Radar unterscheidet sich von einer Liste freigegebener Technologien dadurch, dass es ausdrücklich vorläufig und begründet ist und festhält, was die Organisation gelernt hat, einschließlich ihrer Fehler.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein Radareintrag und das Entscheidungsdokument dahinter

Das Bestellteam will sein pollendes Outbox-Relay (Abschnitt 6.7) durch Change Data Capture mit Debezium ersetzen, das das Write-Ahead-Log der Datenbank liest und Änderungen an Kafka veröffentlicht. Kein Team hat Debezium bisher eingesetzt, daher betrifft die Entscheidung das Plattformteam, das Kafka Connect betreibt, und das Datenteam, das für den Datenbankbetrieb zuständig ist.

1. Entwurf	Das Team schreibt ADR-0031 mit dem Status Proposed und stellt es im wöchentlichen Advisory Forum vor
2. Rat einholen	Plattform-, Daten- und Sicherheitsteam kommentieren den Pull Request; das Forum regt an, das verwaltete CDC des Cloud-Anbieters zu prüfen
3. Entscheiden	Das Bestellteam nimmt den meisten Rat an, lehnt die verwaltete Option wegen Prinzip P3 (Portabilität) ab und hält fest, warum
4. Absichern	Eine Fitness Function für die Event-Verzögerung (p99 unter 2 s) und ein Alarm auf den Rückstand des Replication Slots kommen hinzu
5. Radar	Debezium kommt im Ring Trial ins Radar, beschränkt auf Bestellungen; pollende Relays wandern nach Hold
6. Review	Nach einem Quartal prüft die Radargruppe die Erfahrungen; fallen sie positiv aus, wandert Debezium nach Adopt, und der Review-Vermerk des ADR wird geschlossen

Der Radareintrag ist die kurze öffentliche Zusammenfassung; das ADR enthält die Begründung und den Rat. Beide verweisen aufeinander.

Trial ist im Umfang begrenzt, was die Entscheidung umkehrbar hält, während die Organisation lernt.

Nichts wartete in einer Warteschlange: Die Entscheidung dauerte etwa eine Woche, die meiste Zeit davon wurde mit dem Sammeln wirklich nützlichen Rats verbracht.

```
name,ring,quadrant,isNew,description
Debezium for outbox relay,Trial,Tools,TRUE,"Change data capture from the order database's
write-ahead log to Kafka, replacing polling relays. Trial in ordering only; see ADR-0031.
Owner: ordering team. Review: Q3."
Polling outbox relay,Hold,Techniques,FALSE,"Adds load and latency at peak; no new uses.
Existing relays move when Debezium leaves Trial."
```

Die Radareinträge im CSV-Format von Build Your Own Radar (name, ring, quadrant, isNew, description).

```
# ADR-0031: Use Debezium change data capture for the ordering outbox relay

Status: Accepted (2026-05-14)   Decision-taker: Priya N., ordering team
Context: The polling relay queries the outbox table every 500 ms. At 100 orders/s
it adds measurable database load and up to 1 s of event delay to dispatch.
Options: (1) tune polling; (2) Debezium CDC on the write-ahead log; (3) managed CDC
from the cloud provider.
Decision: Option 2, for the ordering service only, as a Trial on the radar.
Advice received:
- Platform team (2026-05-06): supports; will run Kafka Connect, needs runbook.
- Data team (2026-05-07): concern about replication slot growth if Connect stops;
  asks for an alert on slot lag. Accepted.
- Security (2026-05-08): CDC user must have replication rights only. Accepted.
- Advisory forum (2026-05-09): suggested option 3; rejected for portability (P3).
Consequences: lower load and delay; a new component to operate; slot-lag alert
and a fitness function on event delay (p99 under 2 s) added.
Review: after one quarter, or before any second service adopts it.
```

ADR-0031 in einem kompakten Format im Stil von Nygard, mit dem Rat dokumentiert, wie es der Advice Process verlangt.

Die Rolle der Unternehmensarchitektur heute

Unternehmensarchitektur (Abschnitt 1.2) hatte in agilen Organisationen einen zwiespältigen Ruf: zentrale Teams, die Frameworks, Standards und Genehmigungsprozesse produzierten, die die Delivery-Teams als Hindernisse erlebten. Die Rolle hat sich gewandelt. In Organisationen, in denen es gut funktioniert, kuratieren Unternehmensarchitekten die Prinzipien und das Radar, statt sie vorzuschreiben, pflegen die Landkarte der Geschäftsfähigkeiten und Systeme, die kein einzelnes Team überblicken kann, treffen oder moderieren die wirklich organisationsweiten Einwegtür-Entscheidungen (Identität, Kerndaten, Cloud-Strategie, wichtige Anbieter), leiten das Advisory Forum und gestalten mit den Plattformteams die Plattform und die Paved Road.

Gregor Hohpes Bild vom Architekten, der mit dem Aufzug zwischen Vorstandsetage und Maschinenraum pendelt (Abschnitt 11.7), beschreibt die Fähigkeit, die das erfordert: Strategie in technische Richtung zu übersetzen und technische Zwänge in die Sprache des Geschäfts. Fachleute sind sich weiterhin uneinig, wie viel zentrale Autorität gesund ist. Manche

argumentieren, eine starke zentrale Funktion sei in regulierten, von Übernahmen geprägten oder stark integrierten Unternehmen unverzichtbar; andere meinen, der größte Teil ihres Werts lasse sich über Plattformen, Prinzipien und Beratung erbringen. Die Antwort hängt von der Größe der Organisation ab, von ihrer Regulierung und davon, wie viel ihrer Systemlandschaft wirklich gemeinsam genutzt wird.

Compliance durch Automatisierung

Wie auch immer das Organisationsmodell aussieht: Die günstigste Compliance ist automatisiert. Fitness Functions (Abschnitt 11.3) prüfen Architekturmerkmale. Policy as Code tut dasselbe für organisatorische Regeln: Die Regeln werden in einer maschinenlesbaren Sprache geschrieben, in der Versionsverwaltung gehalten, wie Code reviewt und getestet und automatisch gegen Infrastrukturdefinitionen, Deployment-Manifeste oder API-Anfragen ausgewertet. Open Policy Agent, ein Graduated-Projekt der Cloud Native Computing Foundation, ist eine weit verbreitete Allzweck-Engine mit eigener Policy-Sprache, Rego; Conftest wendet Rego-Policies in einer Pipeline auf Konfigurationsdateien an, und Admission Controller wie OPA Gatekeeper oder Kyverno setzen Policies innerhalb von Kubernetes-Clustern durch.

Automatisierte Compliance hat einen weiteren Vorteil, den regulierte Organisationen sehr schätzen: Nachweise. Ein Pipeline-Log, das zeigt, dass jedes Release dieselben Prüfungen bestanden hat, ist für Prüfer ein weit stärkerer Nachweis als ein unterschriebenes Formular, und es entsteht ohne Zusatzkosten.

```
package main

import rego.v1

# Guardrail: images must come from the organisation's scanned registry
deny contains msg if {
    input.kind == "Deployment"
    some c in input.spec.template.spec.containers
    not startswith(c.image, "registry.fooddelivery.example/")
    msg := sprintf("%s: image %s is not from the approved registry",
[input.metadata.name, c.image])
}

# Guardrail: every container declares a memory limit
deny contains msg if {
    input.kind == "Deployment"
    some c in input.spec.template.spec.containers
    not c.resources.limits.memory
    msg := sprintf("%s: container %s has no memory limit", [input.metadata.name, c.name])
}
```

Zwei Guardrails als Rego-Policies, bei jedem Pipeline-Lauf mit Conftest (conftest test deploy/) gegen Kubernetes-Deployment-Manifeste geprüft.

Autonomie und Ausrichtung ausbalancieren

Die praktische Frage ist, welche Entscheidungen wie gesteuert werden. Eine nützliche Regel folgt aus Abschnitt 1.6: Je umkehrbarer und lokaler eine Entscheidung ist, desto näher am Team sollte sie getroffen werden; je irreversibler und weitreichender, desto mehr Rat und gelegentlich Freigabe braucht sie. Die Tabelle zeigt eine typische Verteilung; jede Organisation passt ihre eigene an.

Wer was entscheidet

Art der Entscheidung	Wer entscheidet	Mechanismus	Beispiel
Lokal und umkehrbar	Das Team	Urteil des Teams; ADR, wenn bedeutend	Eine Bibliothek innerhalb eines Service; interne Modulstruktur
Betrifft andere Teams, mit moderaten Kosten umkehrbar	Das Team, nach Beratung	Advice Process, Advisory Forum, Radar	Einführung von Debezium; ein neues, für andere veröffentlichtes Event
Organisationsweit oder schwer umkehrbar	Benannte leitende Entscheidungsperson nach breiter Beratung	Advice Process mit Unternehmensarchitekten; Änderung von Prinzip oder Radar	Wahl des Identity Providers; primärer Cloud-Anbieter
Objektive, prüfbare Regeln	Niemand im Einzelfall	Guardrails: Policy as Code und Fitness Functions	Freigegebene Image-Registries; keine öffentlichen Storage-Buckets
Reguliert oder folgenreich	Verantwortliche freigebende Person	Frühes, zeitlich begrenztes Gate mit Nachweisen	Verarbeitung von Gesundheitsdaten; sicherheitsrelevante Funktionen

Governance sollte selbst gemessen werden. Nützliche Signale sind, wie lange bedeutende Entscheidungen vom Vorschlag bis zur Annahme brauchen, wie viele Ausnahmen von Standards beantragt und gewährt werden, welcher Anteil der Services auf der Paved Road läuft, wie viele Hold-Technologien sich noch ausbreiten und ob Incidents auf Entscheidungen zurückgehen, bei denen kein Rat eingeholt wurde. Sind Entscheidungen langsam, ist die Governance zu schwer; gehen Incidents immer wieder auf nicht abgestimmte Entscheidungen zurück, ist sie zu leicht.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Gate, Guardrail oder Advice Process?

Nutzen Sie diese Hilfe, wenn Sie entscheiden, wie eine bestimmte Regel oder Art von Entscheidung gesteuert werden soll.

Q1 **Lässt sich die Regel objektiv formulieren und von einer Maschine prüfen?**

- **Ja** Machen Sie daraus einen Guardrail: Policy as Code oder eine Fitness Function in der Pipeline, mit einem Verantwortlichen und einem Weg für Ausnahmen.
- **Nein** Weiter.

Q2 **Ist die Entscheidung auf ein Team beschränkt und günstig umkehrbar?**

- **Ja** Lassen Sie das Team entscheiden; schreiben Sie ein ADR, wenn sie bedeutend ist.
- **Nein** Weiter.

Q3 **Verlangen Gesetz, Regulierung oder die Risikobereitschaft der Organisation eine verantwortliche Freigabe?**

- **Ja** Nutzen Sie ein Gate, aber halten Sie es früh ab, begrenzen Sie es zeitlich und machen Sie die nötigen Nachweise explizit.
- **Nein** Weiter.

Q4 **Ist es eine Einwegtür, die viele Teams betrifft?**

- **Ja** Nutzen Sie den Advice Process mit breiter Konsultation, auch der Unternehmensarchitekten, und einer benannten leitenden Entscheidungsperson.
- **Nein** Nutzen Sie den Advice Process: Konsultieren Sie die Betroffenen und die Fachleute, entscheiden und dokumentieren Sie dann.

ACHTUNG

Häufige Fehler bei der Governance

- Prinzipien, denen niemand widersprechen könnte und die deshalb nichts entscheiden.
- Standards ohne Weg für Ausnahmen, die die Teams dann stillschweigend ignorieren.
- Ein Advisory Forum, das sich langsam in ein Genehmigungsgremium verwandelt.
- Ein Radar, das eine einzige Person pflegt und das Meinungen statt der Erfahrung der Organisation festhält.
- Gates spät in der Lieferung, wo die einzigen realistischen Ergebnisse Freigabe oder teure Nacharbeit sind.
- Regeln automatisieren, für die niemand verantwortlich ist, sodass Fehlalarme umgangen statt behoben werden.

PRÜFEN SIE IHR VERSTÄNDNIS

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

Ein Guardrail hält die Arbeit kontinuierlich innerhalb von Grenzen, meist durch automatische Prüfungen, ohne sie anzuhalten; ein Gate hält die Arbeit an, bis jemand sie freigibt.

▼ Was sind die vier Teile eines Architekturprinzips im TOGAF-Stil?

Name, Aussage, Begründung und Auswirkungen.

▼ Wer trifft im Advice Process die Entscheidung, und muss diese Person dem Rat folgen?

Die Person oder das Team, die entscheiden; sie müssen Rat bei den Betroffenen und bei Fachleuten einholen und dokumentieren, ihm aber nicht folgen.

▼ Was bedeutet es für bestehende Einsätze meist, wenn eine Technologie im Hold-Ring eines Radars landet?

Es bedeutet, dass keine neuen Einsätze beginnen sollen; bestehende Einsätze sind nicht verboten, können aber mit der Zeit migriert werden.

▼ **Probieren Sie es aus: Jede neue Datenbank braucht die Freigabe eines monatlich tagenden Architekturgremiums, und die Warteschlangen erreichen sechs Wochen. Die Teams vermeiden es inzwischen, zu fragen. Die Sicherheit muss dennoch gewährleistet bleiben. Wie würden Sie diese Governance mit Guardrails und dem Advice Process neu gestalten?**

Ersetzen Sie das Gate durch Guardrails, wo Regeln objektiv sind: Policy as Code kann Verschlüsselung, Backups und genehmigte Engines automatisch prüfen und Audit-Nachweise liefern. Veröffentlichen Sie ein Radar, das sagt, welche Datenbanken auf Adopt, Trial oder Hold stehen. Für alles Ungewöhnliche nutzen Sie den Advice Process: Das Team entscheidet, nachdem es Betroffene und Fachleute konsultiert hat, und dokumentiert ein ADR. Das Forum schafft Sichtbarkeit, genehmigt aber nicht. Behalten Sie allenfalls ein frühes Gate, nur für wirklich hochriskante Änderungen.

QUELLEN

- Andrew Harmel-Law: Scaling the Practice of Architecture, Conversationally (martinfowler.com, 2021): https://martinfowler.com/articles/scaling-architecture-conversationally.html
- Thoughtworks Technology Radar: FAQ: https://www.thoughtworks.com/radar/faq
- Thoughtworks: Build Your Own Radar: https://www.thoughtworks.com/radar/byor
- Open Policy Agent documentation: https://www.openpolicyagent.org/docs/latest/

Team Topologies und soziotechnisches Design

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 3.7 Gesetz von Conway
- 5.7 Domain-Driven Design
- 10.7 Platform Engineering

Begriffe von früher, die er verwendet: Gesetz von Conway, Bounded Context, Internal Developer Platform (IDP)

DIE IDEE IN KÜRZE

Denken Sie an eine Restaurantgruppe. Die Küche jedes Restaurants bedient ihre eigenen Gäste von Anfang bis Ende. Ein zentraler Lieferant beliefert jede Küche aus einem einfachen Katalog mit Zutaten und Ausstattung, damit die Köche keinen Bauernhof betreiben müssen. Eine spezialisierte Konditorei stellt das eine Produkt her, das zu schwierig ist, als dass jede Küche es beherrschen könnte. Und ein Gastküchenchef bringt einer Küche ein paar Wochen lang eine neue Technik bei und zieht dann weiter. Team Topologies gibt Softwareorganisationen dieselben vier Arten von Teams und drei Arten der Zusammenarbeit, damit jedes Team eine Aufgabe verantwortet, die es tatsächlich im Kopf behalten kann.

WAS SIE WISSEN MÜSSEN

- Soziotechnische Architektur gestaltet Softwarestruktur und Teamstruktur gemeinsam, weil sich beide nach dem Gesetz von Conway gegenseitig prägen.
- Team Topologies (Skelton und Pais, 2019; zweite Auflage 2025) definiert vier Teamtypen: Stream-aligned (der Standard), Plattform, Enabling und Complicated Subsystem.
- Teams arbeiten in drei bewusst gewählten Modi zusammen: Collaboration (zeitlich begrenztes gemeinsames Erkunden), X-as-a-Service (eine klare Schnittstelle nutzen) und Facilitating (einem anderen Team helfen, eine Fähigkeit aufzubauen).
- Die kognitive Last des Teams, abgeleitet aus John Swellers Cognitive Load Theory (1988), begrenzt, wie viel ein Team verantworten kann; verringern Sie die extrinsische Last mit Plattformen und Automatisierung, und begrenzen Sie die intrinsische Last durch gut gesetzte Grenzen.

- Eine Team-API dokumentiert, was ein Team anbietet und erwartet, sowohl technische Schnittstellen als auch Arbeitsweisen; eine Thinnest Viable Platform ist die kleinste Plattform, die ihre Nutzer tatsächlich schneller macht.
- Bruchlinien wie Bounded Contexts, regulatorischer Geltungsbereich, Änderungstakt, Risiko und Performance-Isolation zeigen, wo Software und Teams aufgeteilt werden sollten; die Technologie allein ist meist die schwächste Naht.
- Eine Neuorganisation der Teams strukturiert gekoppelten Code nicht um; Teamänderungen und Architekturänderungen sollten daher gemeinsam geplant und weiterentwickelt werden.

Teamdesign ist Architekturarbeit

Kapitel 3 hat gezeigt, dass das Gesetz von Conway keine Kuriosität ist, sondern eine Kraft: Das System, das eine Gruppe von Teams baut, bildet tendenziell ab, wie diese Teams kommunizieren. Wenn das stimmt, dann ist die Entscheidung, wie Teams gebildet werden, was jedes verantwortet und wie sie zusammenarbeiten, eine Architekturentscheidung mit ebenso dauerhaften Folgen wie die Wahl einer Datenbank oder eines Integrationsstils. Die Disziplin, Softwarestruktur und Teamstruktur gemeinsam zu gestalten, heißt soziotechnische Architektur.

Der einflussreichste praktische Ansatz dafür ist Team Topologies von Matthew Skelton und Manuel Pais (IT Revolution, 2019; zweite Auflage 2025). Er geht von einem einfachen Ziel aus, dem *schnellen Fluss* (Fast Flow): einem stetigen, nachhaltigen Strom kleiner Änderungen von der Idee bis in die Produktion, mit wenig Warten zwischen Teams. Dann bietet er ein kleines Vokabular, um Organisationen zu gestalten, die das erreichen: vier Teamtypen, drei Interaktionsmodi, eine Grenze namens kognitive Last des Teams und eine Technik, um die Nähte zu finden, entlang derer sich Software und Teams aufteilen lassen.

Mehrere Grundannahmen ziehen sich durch den Ansatz. Das Team, nicht der Einzelne, ist die Liefereinheit: Arbeit wird an Teams vergeben, Verantwortung liegt bei Teams, und Teams sind langlebig, damit sie Wissen über ihre Domäne und ihren Code ansammeln, statt für ein Projekt zusammengestellt und danach wieder verstreut zu werden. Teams sind klein, meist fünf bis neun Personen, weil Vertrauen und gemeinsames Verständnis jenseits dieser Größe schnell abnehmen (die Autoren stützen sich auf Robin Dunbars Forschung zu Gruppengrößen). Und Teamgrenzen sollten für den Fluss von Änderungen gestaltet werden, nicht nach der Bequemlichkeit von Berichtswegen oder der Geschichte, wer was gebaut hat.

Für Architekten verändert das die Aufgabe. Eine vorgeschlagene Service-Grenze ist nur gut, wenn ein Team sie vollständig verantworten kann; ein vorgeschlagenes Team ist nur tragfähig, wenn der Code, den es verantwortet, ihm erlaubt zu arbeiten, ohne drei andere Teams um

Erlaubnis zu fragen. Wer Kästchen zeichnet, ohne zu fragen, wer jedes davon verantworten wird, entwirft nur das halbe System.

> **WENN DAS NEU FÜR SIE IST**
>
> ### Wenn das neu für Sie ist
>
> Ein *Wertstrom* (Value Stream) ist die Abfolge von Arbeit, die einen Bedarf in etwas verwandelt, von dem Kunden oder Nutzer profitieren, zum Beispiel alles, was nötig ist, damit Kunden eine Bestellung für später planen können. *Cross-funktional* bedeutet, dass das Team alle Fähigkeiten besitzt, die diese Arbeit braucht (Design, Frontend, Backend, Test, Betrieb), statt Arbeit an andere Teams weiterzugeben. Abschnitt 3.7 dieses Kurses erklärt das Gesetz von Conway und die Spiegelung zwischen Organisation und System, auf der dieser Abschnitt aufbaut.

Die vier grundlegenden Teamtypen

Team Topologies vertritt die Auffassung, dass fast jedes Team in einer Softwareorganisation einem von vier Typen angehören sollte. Der Sinn so weniger Typen ist Klarheit: Jeder Typ hat einen anderen Zweck, eine andere Art, Erfolg zu messen, und eine andere Beziehung zu den übrigen Teams; wer den Typ eines Teams kennt, weiß also, was er von ihm erwarten kann.

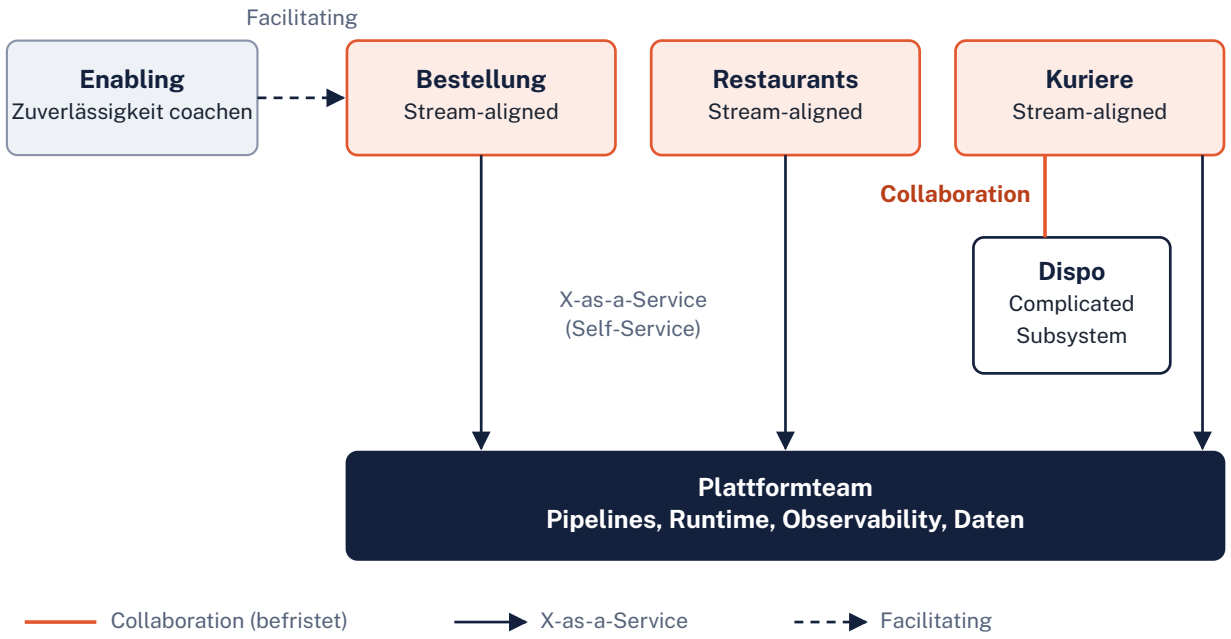
Ein Stream-aligned Team ist auf einen Strom von Änderungen ausgerichtet: ein Produkt, eine Customer Journey, eine Nutzergruppe oder eine fachliche Fähigkeit. Es verantwortet seinen Teil durchgängig, von der Erkundung über den Bau bis zum Betrieb in der Produktion, und sollte die meisten Änderungen liefern können, ohne auf jemanden zu warten. Stream-aligned Teams sind der Standard und sollten die große Mehrheit bilden; die anderen drei Typen gibt es, um ihre Last zu verringern.

Ein Plattformteam stellt interne Dienste (Deployment-Pipelines, eine Laufzeitumgebung, Observability, Dateninfrastruktur) als Produkt bereit, das Stream-aligned Teams per Self-Service nutzen. Eine gute Plattform wird an der Geschwindigkeit ihrer Nutzer gemessen, nicht an ihrer eigenen Feature-Liste. Ein Enabling Team besteht aus Fachleuten, die anderen Teams helfen, eine fehlende Fähigkeit zu erwerben, etwa Testautomatisierung, Observability-Praktiken oder ein neues Framework, und dann weiterziehen; sein Erfolg zeigt sich daran, dass das andere Team es nicht mehr braucht. Ein Complicated-Subsystem-Team verantwortet einen Teil, der seltenes Fachwissen erfordert, etwa eine Preis-Engine auf Basis versicherungsmathematischer Modelle, einen Solver zur Routenoptimierung oder einen Video-Codec, damit Stream-aligned Teams ihn nutzen können, ohne ihn beherrschen zu müssen. Das Buch betont, dass dieser Typ sparsam und nur bei echtem Spezialwissen eingesetzt werden sollte.

Die vier Teamtypen im Vergleich

Teamtyp	Zweck	Verantwortet	Gesunde Anzeichen	Beispiel Essenslieferung
Stream-aligned	Einen kontinuierlichen Strom von Änderungen für einen Teil des Geschäfts liefern	Features, Code, Daten und Betrieb seines Stroms, durchgängig	Released unabhängig; die meisten Änderungen brauchen kein anderes Team; die Bereitschaftslast ist tragbar	Team Kundenbestellung: Speisekarten für Kunden, Warenkorb, Checkout, Bestellverfolgung
Plattform	Die kognitive Last der Stream-aligned Teams durch interne Self-Service-Produkte verringern	Interne Plattformdienste, ihre Dokumentation und ihre Zuverlässigkeit	Teams nutzen sie freiwillig; das Onboarding eines neuen Service dauert Stunden, nicht Wochen	Team Lieferplattform: Pipelines, Container-Runtime, Logging und Metriken, Event Broker
Enabling	Fähigkeitslücken in anderen Teams schließen und dann gehen	Know-how, Leitfäden, Beispiele; nicht den Produktionscode anderer Teams	Einsätze sind zeitlich begrenzt und enden damit, dass das andere Team selbstständig ist	Enabling Team für Zuverlässigkeit, das Teams zu Service Level Objectives und Incident-Praxis coacht
Complicated Subsystem	Ein Spezialgebiet kapseln, das für jedes Stream-aligned Team zu tief ist	Eine Spezialkomponente hinter einer klaren Schnittstelle	Nutzer verwenden sie als Service, ohne die Fachleute im Raum zu brauchen	Team Dispositionsoptimierung: der Algorithmus, der Kuriere Bestellungen zuweist

Teamtypen und Interaktionsmodi



Die meisten Teams sind Stream-aligned; eine Plattform bedient sie als Service, ein Enabling Team unterstützt (Facilitating), und ein Complicated-Subsystem-Team arbeitet für begrenzte Zeit eng mit einem Strom zusammen.

Drei Interaktionsmodi

Teamtypen sagen, was Teams sind; Interaktionsmodi sagen, wie sie zusammenarbeiten. Skelton und Pais argumentieren, dass unklare, spontane Zusammenarbeit zwischen Teams eine Hauptursache für Verzögerungen ist, weil jedes ungeplante Gespräch eine Abhängigkeit darstellt. Sie empfehlen daher, dass jedes Paar von Teams, das zusammenarbeiten muss, bewusst einen von drei Modi wählt, für einen festgelegten Zeitraum.

Bei **Collaboration** arbeiten zwei Teams eng an einem gemeinsamen Ziel zusammen, mit verschwimmenden Zuständigkeiten, zum Beispiel während sie herausfinden, wie eine neue Fähigkeit aufgeteilt werden sollte. Das bringt schnelles Lernen, kostet aber beide Teams Kapazität und kognitive Last und sollte daher zeitlich begrenzt sein. Bei X-as-a-Service stellt ein Team etwas bereit (eine API, eine Plattformfähigkeit, eine Bibliothek), das ein anderes mit wenig oder ganz ohne Abstimmung nutzt; das skaliert gut, funktioniert aber erst, wenn die Grenze gut verstanden und die Schnittstelle gut ist. Bei **Facilitating** hilft ein Team, typischerweise ein Enabling Team, einem anderen, etwas zu lernen oder ein Hindernis zu beseitigen.

Die Modi sollen sich ändern. Ein häufiger Ablauf ist erst erkunden, dann festigen: Ein Plattformteam arbeitet einige Wochen mit einem Stream-aligned Team zusammen, um eine neue Fähigkeit zu entwerfen, macht daraus dann ein Self-Service-Angebot und wechselt zu X-as-a-Service. Die Modi dienen auch als Sensor. Wenn ein Team, das einen Service als Service nutzen soll, ständig Meetings braucht, sind die Schnittstelle oder die Grenze wahrscheinlich falsch; wenn zwei Teams seit einem Jahr zusammenarbeiten, sind sie vielleicht in Wahrheit ein Team, oder die Trennung zwischen ihnen liegt an der falschen Stelle.

Die drei Interaktionsmodi

Modus	Wie er aussieht	Einsetzen, wenn	Kosten oder Warnsignal
Collaboration	Zwei Teams arbeiten eine Zeit lang wie eines an einem gemeinsamen Problem; gemeinsame Design-Sessions, gemeinsame Backlog-Einträge	Eine neue Grenze, Technologie oder Fähigkeit erkunden, bei der noch niemand die richtige Schnittstelle kennt	Hohe kognitive Last für beide Teams; dauert es unbegrenzt an, ist die Grenze wahrscheinlich falsch
X-as-a-Service	Ein Team nutzt eine API, Plattform oder Komponente, die ein anderes bereitstellt, über Dokumentation und Self-Service	Grenze und Schnittstelle sind gut verstanden und stabil	Passt sich langsam an, wenn die Schnittstelle schlecht ist; ständige Supportanfragen zeigen, dass es noch kein echter Service ist
Facilitating	Coaching, Pairing, Beispiele und kurze Workshops, die einem Team helfen, eine Fähigkeit aufzubauen	Einem Team fehlt eine Fähigkeit, oder es wird von einem Hindernis blockiert, das es selbst beseitigen lernen kann	Wird dazu, die Arbeit des anderen Teams zu erledigen, was Abhängigkeit statt Fähigkeit aufbaut

Kognitive Last als Entwurfsrandbedingung

Die Idee der kognitiven Last stammt aus der pädagogischen Psychologie. John Swellers Cognitive Load Theory, erstmals 1988 veröffentlicht, ging von der Beobachtung aus, dass das menschliche Arbeitsgedächtnis nur wenige neue Elemente gleichzeitig halten und verarbeiten kann, sodass Unterricht so gestaltet werden sollte, dass er es nicht überlastet. Die Theorie unterscheidet intrinsische Last (die inhärente Schwierigkeit des Stoffs), extrinsische Last (Schwierigkeit, die durch die Art der Aufgabenstellung entsteht) und, in späteren Arbeiten, lernbezogene Last (germane Load: Anstrengung, die dauerhaftes Verständnis aufbaut). Die Forschung diskutiert noch, ob die lernbezogene Last eine eigene Kategorie ist, und Team Topologies nutzt alle drei als praktische Heuristik, nicht als Messgröße.

Abschnitt 3.7 hat die Teamversion dieser Idee eingeführt: alles, was ein Team im Kopf behalten muss, um das zu bauen, zu betreiben und zu ändern, was es verantwortet. Hier geht es darum, sie als Entwurfsrandbedingung zu nutzen. Wenn eine Architektur einem Team mehr zuweist, als es verstehen kann, sind die Folgen vorhersehbar: langsamere Lieferung, oberflächliche Fixes, eine fragile Rufbereitschaft, Wissen, das sich auf ein oder zwei Personen konzentriert, und schließlich Kündigungen. Es sind dieselben Symptome, die oft schlechten Entwicklern oder schlechtem Code angelastet werden.

Das Buch bietet Faustregeln, um die Verantwortlichkeiten eines Teams nach der Komplexität der Domäne zu bemessen: Geben Sie jede Domäne einem einzigen Team; ein Team kann meist zwei oder drei einfache Domänen bewältigen; ein Team, das eine komplexe Domäne verantwortet, sollte nichts anderes verantworten; und kein Team sollte zwei komplizierte Domänen verantworten. Man kann Teams auch direkt fragen, zum Beispiel mit einer kurzen Umfrage dazu, wie gut sie ihre Systeme verstehen, wie sicher sie sich in der Rufbereitschaft fühlen und wie viel Zeit in Werkzeuge statt in die Domäne fließt. Wachsende Ticket-Warteschlangen, lange Einarbeitungszeiten und häufige Übergaben an die eine Person, die eine Komponente versteht, sind nützliche indirekte Signale.

Die architektonischen Antworten spiegeln die drei Arten von Last. Verringern Sie die extrinsische Last mit Plattformen, Automatisierung, Paved Roads und guter Dokumentation. Begrenzen Sie die intrinsische Last, indem Sie Service- und Modulgrenzen so ziehen, dass der Anteil jedes Teams an der Domäne zusammenhängend und von beherrschbarer Größe ist, und indem Sie tiefe Spezialgebiete hinter Schnittstellen von Complicated Subsystems legen. Schützen Sie die lernbezogene Last, indem Sie Zeit zum Lernen lassen; hier helfen Enabling Teams.

Team-APIs und die Thinnest Viable Platform

Wenn Teams in bewusst gewählten Modi zusammenarbeiten, braucht jedes Team ein klares Gesicht gegenüber dem Rest der Organisation. Team Topologies nennt das die Team-API: alles, was das Team anbietet und erwartet. Dazu gehört die technische API (Endpoints, Events, Bibliotheken sowie ihre Regeln für Versionierung und Abkündigung), aber auch die menschliche Schnittstelle: Dokumentation und Onboarding-Leitfäden, die Roadmap und aktuellen Prioritäten des Teams, seine Arbeitsweisen, die Kanäle, über die es erreichbar ist, wie man Arbeit bei ihm anfragt und mit welchen Antwortzeiten zu rechnen ist.

Eine Team-API aufzuschreiben ist eine günstige und aufschlussreiche Übung. Sie zwingt ein Team zu sagen, was es verantwortet, was meist Services ohne Verantwortliche und umstrittene Daten offenlegt, und sie gibt anderen Teams eine Möglichkeit zur Zusammenarbeit ohne Meeting. Viele Organisationen führen sie als kurze Seite im Repository des Teams, die jedes Quartal zusammen mit der Architekturdokumentation überprüft wird.

Die Plattform verdient besondere Sorgfalt, weil hier gute Absichten am häufigsten schiefgehen. Eine Plattform, die als großes internes Projekt mit langer Roadmap und verpflichtender Einführung gebaut wird, kann leicht kognitive Last hinzufügen, statt sie zu verringern. Skelton und Pais empfehlen die Thinnest Viable Platform: die kleinste Menge an Diensten, Dokumentation und Standards, die ihre Nutzer tatsächlich schneller macht. Für eine kleine Organisation kann das eine gut gepflegte Seite sein, die auflistet, welche Dienste des Cloud-Anbieters die Teams nutzen sollen und wie; für eine große kann es eine Internal Developer Platform sein. Platform Engineering wird ausführlich in Kapitel 10 behandelt; entscheidend ist hier, dass eine Plattform ein Produkt ist, dessen Kunden andere Teams sind, und dass sie als solches gestaltet und gemessen werden sollte.

Bruchlinien: wo Software und Teams aufgeteilt werden

Wenn ein Teil des Systems für ein Team zu groß geworden ist, muss er aufgeteilt werden, und die Aufteilung muss sowohl für den Code als auch für die Teams funktionieren. Team Topologies nennt die natürlichen Nähte, entlang derer sich Software teilen lässt, Bruchlinien (Fracture Planes). Das Buch nennt mehrere, und die erste ist meist die beste: ein Bounded Context aus dem Domain-Driven Design, weil eine Grenze, die der Sprache und den Regeln des Geschäfts folgt, zusammengehörige Änderungen tendenziell beieinanderhält.

Die anderen sind nützlich, wenn sie mit einem echten Unterschied darin übereinstimmen, wie Teile des Systems behandelt werden müssen. Eine Aufteilung allein entlang einer Technologiegrenze (zum Beispiel ein Mobile-Team und ein Backend-Team) ist in der Regel die schwächste Wahl, weil die meisten Änderungen sie dann überqueren; das ist genau die geschichtete Teamstruktur, vor der das Gesetz von Conway warnt.

Bruchlinien mit Beispielen aus der Essenslieferung

Bruchlinie (Fracture Plane)	Aufteilen, wenn	Beispiel
Fachliche Domäne (Bounded Context)	Teile der Domäne haben ihre eigene Sprache, eigene Regeln und eigene Fachleute	Bestellung, Restaurant-Onboarding und Kurierverwaltung als getrennte Kontexte
Regulatorische Compliance	Ein Teil unterliegt Regeln, die den Rest nicht belasten sollen	Die Verarbeitung von Kartenzahlungen wird für PCI DSS in einem kleinen, auditierten Bereich gehalten
Änderungstakt	Manche Teile ändern sich täglich, andere jährlich	Aktionen und Kampagnen ändern sich wöchentlich; Rechnungsregeln ändern sich selten
Standort des Teams	Teams arbeiten in Zeitzonen mit wenig Überschneidung	Ein Team für den Kurierbetrieb in einem anderen Land verantwortet das Kurier-Onboarding
Risiko	Manche Teile vertragen Experimente, andere nicht	Empfehlungsexperimente getrennt von der Bestellannahme
Performance-Isolation	Ein Teil hat sehr andere Anforderungen an Last oder Latenz	Live-Standortaktualisierungen der Kuriere getrennt vom Durchsuchen der Speisekarten
Technologie	Ein Teil braucht einen wirklich anderen Technologie-Stack	Eine alte Tablet-Integration für Restaurants, die nur auf alter Hardware läuft
Personas	Verschiedene Nutzergruppen haben unterschiedliche Bedürfnisse und Journeys	Kunden, Restaurantpersonal und Kuriere haben jeweils ihre eigene App und ihr eigenes Team

Das Gesetz von Conway, das Inverse Conway Maneuver und seine Grenzen

Team Topologies ist im Grunde ein Werkzeugkasten für das Inverse Conway Maneuver, das Abschnitt 3.7 beschrieben hat: Legen Sie die gewünschte Architektur fest, und gestalten Sie dann Teams und Interaktionsmodi so, dass das Gesetz von Conway das System in diese Richtung treibt. Seine Ergänzungen sind praktischer Natur. Teamtypen sagen, welche Art von Team jeden Teil verantworten sollte; Interaktionsmodi sagen, welche Teamgrenzen dünn sein sollen (X-as-a-Service) und welche vorübergehend dick (Collaboration); die kognitive Last sagt, wie viel jedes Team verantworten kann; Bruchlinien sagen, wo geschnitten wird.

Die Grenzen aus Abschnitt 3.7 gelten weiterhin und verdienen es, in diesem Zusammenhang wiederholt zu werden. Eine Neuorganisation der Teams strukturiert eine eng gekoppelte Codebasis nicht um: Monatelang nach einer Neuorganisation kämpfen die neuen Teams gegen die alte Architektur, weshalb Teamänderungen und Architekturänderungen als ein gemeinsames Programm geplant werden sollten. Grenzen, die zu früh in Teams verfestigt werden, machen Fehler teuer; viele Organisationen halten unsichere Grenzen daher innerhalb eines Teams oder eines modularen Monolithen, bis sie Belege haben. Und häufige Neuorganisationen zerstören das Wissen langlebiger Teams, auf das der Ansatz angewiesen ist.

Fachleute sind sich auch uneinig, wie wörtlich der Ansatz anzuwenden ist. Befürworter schätzen das gemeinsame Vokabular, das vage Klagen (wir warten ständig auf das Datenteam) in gestaltbare Fragen verwandelt (welchen Interaktionsmodus wollen wir mit dem Datenteam, und was würde ihn zu X-as-a-Service machen?). Kritiker warnen, dass die vier Typen oft als Etiketten auf eine unveränderte Organisation geklebt werden, dass echte Teams manchmal Typen mischen (ein Plattformteam, das auch ein Complicated Subsystem verantwortet) und dass der Ansatz wenig über Finanzierung, Führungsstruktur oder Anreize sagt. Beide Punkte sind berechtigt: Die Typen sind eine Entwurfshilfe, kein Organigramm, und die Autoren selbst stellen sie als Ausgangspunkt für fortlaufende Anpassung dar.

WOHER DAS KOMMT

Woher soziotechnisches Denken kommt

Der Begriff soziotechnisches System stammt vom Tavistock Institute in London. 1951 veröffentlichten Eric Trist und Ken Bamforth eine Studie über den britischen Kohlebergbau, die zeigte, dass eine neue, stärker mechanisierte Abbaumethode die Produktivität gesenkt hatte, weil sie kleine, selbstorganisierte Arbeitsgruppen auseinanderriss. Ihre Schlussfolgerung, dass technische und soziale Systeme gemeinsam gestaltet werden müssen, ist dieselbe Einsicht, zu der Conway 1968 für Software gelangte und die Team Topologies heute anwendet.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Teams der Essenslieferplattform entlang von Strömen neu ordnen

Die Essenslieferplattform ist auf 38 Engineers in fünf Teams gewachsen, die nach technischen Schichten organisiert sind: Mobile-Apps, Web-Frontend, Backend-API, Datenbank und Betrieb. Die Durchlaufzeit von der Idee bis in die Produktion beträgt etwa sechs Wochen, Releases werden alle zwei Wochen über alle Teams hinweg koordiniert, und die letzten drei Incidents dauerten jeweils mehrere Stunden, weil niemand eine ganze Customer Journey verantwortete. Der Architekt und die Engineering-Leitung gestalten die Organisation in fünf Schritten neu.

1. Die Ströme kartieren	Event Storming und eine Durchsicht der letzten 50 Features zeigen drei Hauptströme von Änderungen: Kundenbestellung (Speisekarten, Warenkorb, Checkout, Verfolgung), Restaurantpartner (Onboarding, Speisekarten, Öffnungszeiten, Auszahlungen) und Lieferung (Kurier-Onboarding, Zuweisung, Live-Verfolgung). 44 der 50 Features fielen überwiegend in einen davon.
2. Die Bruchlinien der Software prüfen	Bestellung, Restaurants und Lieferung sind bereits getrennte Bounded Contexts mit eigenen Services und Datenbanken (Abschnitt 11.1), der Code kann also folgen. Die Kurierzuweisung nutzt einen Optimierungs-Solver, den nur drei Engineers verstehen: ein Kandidat für ein Complicated Subsystem. Kartenzahlungen bleiben bei Bestellung, aber in einem kleinen, separat auditierten Modul (regulatorische Bruchlinie).
3. Teams nach kognitiver Last bemessen	Drei Stream-aligned Teams mit je 7 Personen (jedes mit Mobile-, Web-, Backend-, Test- und Betriebskompetenz). Ein Complicated-Subsystem-Team mit 5 Personen für die Dispositionsoptimierung (die drei Fachleute plus zwei Engineers, die sich in die Domäne einarbeiten). Ein Plattformteam mit 9 Personen, das Pipelines, die Container-Runtime, Observability und den Event Broker verantwortet. Ein kleines Enabling Team mit 3 Personen für Zuverlässigkeitspraktiken. Insgesamt: 38.
4. Interaktionsmodi wählen	Stream-aligned Teams nutzen die Plattform vom ersten Tag an als X-as-a-Service, mit ihren bestehenden Pipelines. Lieferung und Disposition arbeiten ein Quartal lang zusammen (Collaboration), um die Zuweisungsschnittstelle neu zu gestalten, und wechseln dann zu X-as-a-Service. Das Enabling Team unterstützt Bestellung acht Wochen lang (Service Level Objectives, Runbooks für die Rufbereitschaft) und zieht dann zu Restaurants weiter.
5. Team-APIs veröffentlichen und messen	Jedes Team schreibt eine einseitige Team-API, jeder Service und jede Datenbanktabelle erhält genau ein verantwortliches Team, und die Leitung verfolgt jedes Quartal Durchlaufzeit, Deployment-Häufigkeit, teamübergreifende Tickets pro Feature und eine Umfrage zur Sicherheit in der Rufbereitschaft.

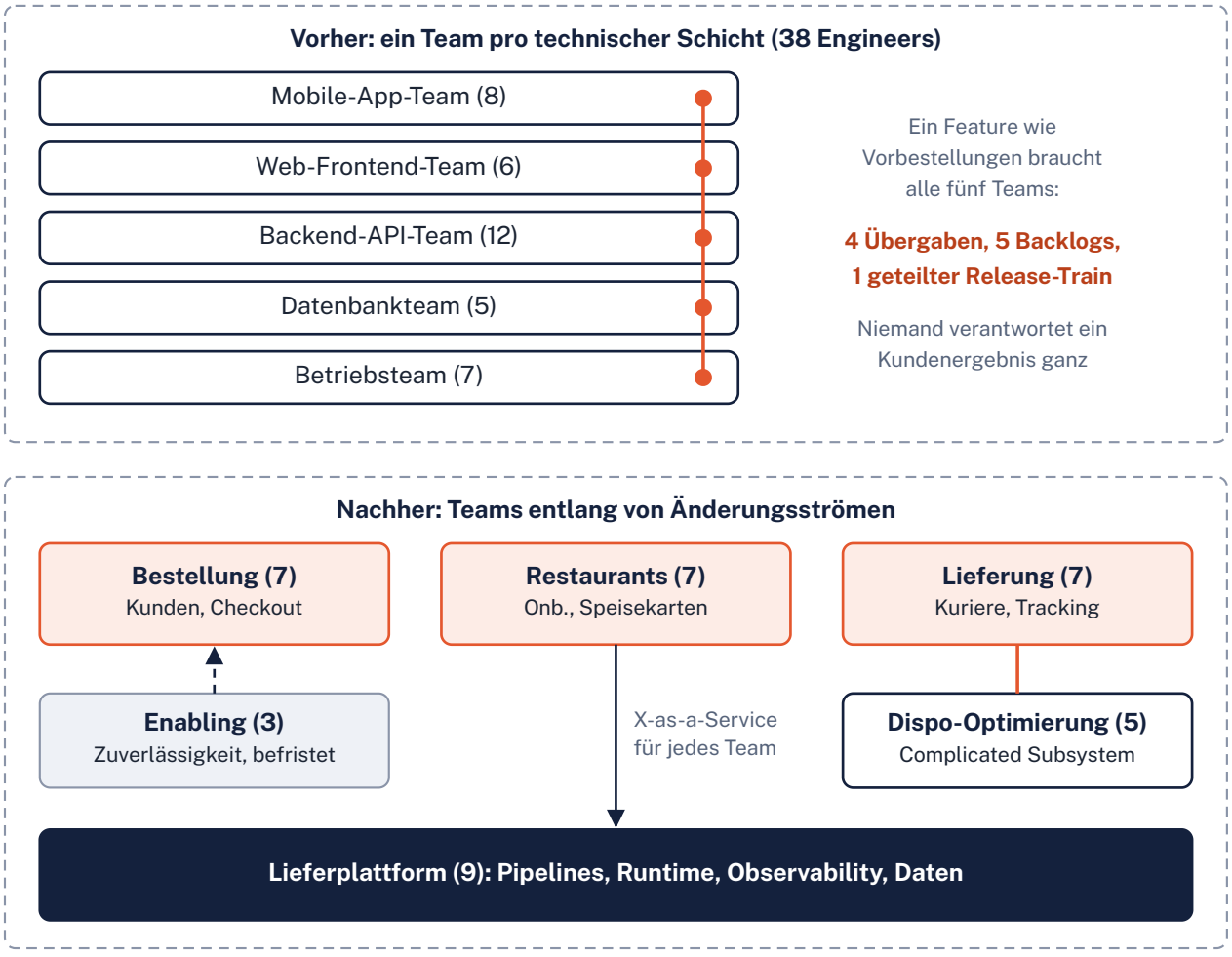
Angestrebtes Ergebnis	Nach zwei Quartalen sollten die meisten Features von einem Team ohne koordiniertes Release geliefert werden; anhaltende teamübergreifende Arbeit würde eine Grenze zeigen, die zu überdenken ist.

Die Neuordnung geht von Änderungsströmen und den bestehenden Grenzen der Software aus, nicht von den aktuellen Berichtswegen.

Fachleute werden nur dort gebündelt, wo die Domäne es verlangt (Disposition), und die Plattform existiert, um die Last der Stream-aligned Teams zu verringern, nicht um sie zu kontrollieren.

Interaktionsmodi haben ein Enddatum; von der Organisation wird also erwartet, dass sie sich weiterentwickelt, statt in einem neuen Organigramm festzuschreiben.

Vorher und nachher



Schichtteams zwingen jedes Feature durch jedes Team; Stream-aligned Teams, unterstützt von einer Plattform, einem Enabling Team und einem Spezialistenteam, können die meisten Features allein liefern.

ACHTUNG

Häufige Fehler

- Bestehende Teams mit Etiketten von Team Topologies umbenennen, während Verantwortung, Finanzierung und Zusammenarbeit gleich bleiben.
- Ein Plattformteam, das über Tickets und Genehmigungen arbeitet, sodass Stream-aligned Teams auf es warten: Das ist ein Engpass, keine Plattform.
- Ein Enabling Team, das am Ende dauerhaft die Arbeit anderer Teams erledigt, was Abhängigkeit statt Fähigkeit aufbaut.
- Collaboration ohne Enddatum, die stillschweigend die kognitive Last zweier Teams zusammenlegt.
- Teams entlang von Technologielinien aufteilen (Frontend, Backend, Datenbank) und einen schnellen Fluss erwarten.
- Erst Service-Grenzen ziehen und später Verantwortliche zuweisen, sodass verwaiste oder gemeinsam verantwortete Services übrig bleiben.

Häufige Fragen

Sind Komponententeams immer falsch?
Nicht immer. Ein Team, das eine gemeinsame Komponente verantwortet, kann die richtige Wahl sein, wenn die Komponente tiefes Spezialwissen erfordert (ein Complicated Subsystem) oder als echtes Self-Service-Produkt angeboten wird (Teil einer Plattform). Zum Problem wird es, wenn die meisten Features Änderungen an der Komponente brauchen und die Stream-aligned Teams dafür Schlange stehen müssen.

Wo sitzen Architekten in diesem Modell?
Meist nicht als eigenes Team, das als Torwächter fungiert. Architekten handeln oft wie ein Enabling Team, helfen Stream-aligned Teams, gute Entscheidungen zu treffen, halten systemweite Qualitäten sichtbar und wirken an der Gestaltung der Teamstruktur selbst mit. Abschnitt 11.4 erörtert, wie Governance ohne Gates funktionieren kann.

Wie oft sollten sich Teamgrenzen ändern?
Selten und bewusst. Von Interaktionsmodi wird erwartet, dass sie sich alle paar Monate ändern; Teamzugehörigkeit und Verantwortung sollten viel länger stabil bleiben, weil sich Domänen- und Systemwissen in langlebigen Teams ansammelt.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Nennen Sie drei Dinge, die eine Team-API neben ihren Code-Schnittstellen enthält.**

Zum Beispiel ihre Dokumentation und Onboarding-Leitfäden, ihre Roadmap und Prioritäten, ihre Arbeitsweisen, die Kanäle, über die das Team erreichbar ist, und wie man Arbeit bei ihm anfragt.

▼ **Warum sollen sich Interaktionsmodi im Lauf der Zeit ändern?**

Weil Collaboration zum Erkunden einer neuen Grenze passt und sich die Teams, sobald Grenze und Schnittstelle verstanden sind, auf X-as-a-Service einpendeln sollten; die Modi folgen der Reife der Grenze.

▼ **Was ist die Thinnest Viable Platform, und warum sollte man sie anstreben?**

Die kleinste Menge an Diensten, Dokumentation und Standards, die die nutzenden Teams tatsächlich schneller macht; eine größere Plattform läuft Gefahr, kognitive Last hinzuzufügen, statt sie zu verringern.

▼ **Warum ist eine Aufteilung in ein Frontend- und ein Backend-Team meist eine schwache Bruchlinie?**

Weil die meisten Features sie überqueren, sodass jede Änderung beide Teams, Übergaben und koordinierte Releases braucht; das ist die geschichtete Struktur, die das Gesetz von Conway in ein geschichtetes, gekoppeltes System verwandelt.

▼ **Probieren Sie es aus: Weisen Sie in einem Ticketing-Unternehmen einen Teamtyp und einen Interaktionsmodus zu: einem Team, das die Kaufstrecke verantwortet; einer kleinen Gruppe von Fachleuten für Algorithmen zur Platzzuteilung; einer Gruppe, die CI- und Deployment-Werkzeuge bereitstellt; zwei Fachleuten, die Teams beim Lernen des Testens helfen.**

Kaufablauf: Stream-aligned, der Standard. Spezialisten für die Platzvergabe: Complicated-Subsystem, bieten dem Ablauf-Team X-as-a-Service an, mit zeitlich begrenzter Collaboration bei neuen Schnittstellen. Tooling-Gruppe: Plattform, bietet X-as-a-Service an und hält es bei der Thinnest Viable Platform. Testfachleute: Enabling, nutzen den Facilitating-Modus und ziehen sich möglichst zurück, sobald die Fähigkeit übertragen ist. Prüfen Sie, ob die kognitive Last jedes Teams beherrschbar ist, und denken Sie daran, dass Teams umzubenennen allein gekoppelten Code nicht ändert.

QUELLEN

- Team Topologies: key concepts: https://teamtopologies.com/key-concepts

- Martin Fowler: Team Topologies (2023): https://martinfowler.com/bliki/TeamTopologies.html
- Melvin Conway: How Do Committees Invent? (1968): https://melconway.com/Home/Committees_Paper.html

Altsysteme modernisieren: Strangler Fig und schrittweise Migration

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.7 Technische Schulden
- 5.7 Domain-Driven Design
- 7.1 Datenhoheit

Begriffe von früher, die er verwendet: Technische Schulden, System of Record, API-Gateway

DIE IDEE IN KÜRZE

Stellen Sie sich eine Stadt vor, die eine neue Brücke braucht. Die alte abzureißen und dann einen Ersatz zu bauen, würde die Stadt jahrelang abschneiden. Stattdessen bauen Ingenieure die neue Brücke neben der alten, öffnen sie Spur für Spur, verlagern den Verkehr schrittweise und reißen die alte Brücke erst ab, wenn nichts sie mehr nutzt. Hat eine neue Spur ein Problem, fließt der Verkehr einfach für eine Weile wieder über die alte Brücke. Die Modernisierung alter Software gelingt am besten genauso: sie Stück für Stück ersetzen, während sie weiterläuft, damit Kunden nie den Dienst verlieren, auf den sie angewiesen sind.

WAS SIE WISSEN MÜSSEN

- Ein Altsystem liefert noch Wert, ist aber schwer, riskant oder teuer zu ändern; modernisieren Sie aus geschäftlichen Gründen wie blockierten Änderungen, verlorenem Support, knappen Fachkenntnissen oder Kosten, nicht aus Mode.
- Big-Bang-Neuentwicklungen scheitern, weil der Wert spät kommt, sich das Ziel verschiebt, verborgenes Verhalten verloren geht und Datenmigration und Umstellung zuletzt kommen; Spolskys Essay von 2000 und die Migration von TSB 2018 sind die klassischen Warnungen.
- Die sieben Rs (Retire, Retain, Rehost, Relocate, Repurchase, Replatform, Refactor oder Re-architect) geben jeder Anwendung in einem Portfolio eine bewusste Strategie; Neubau ist nur eine Option.
- Das Strangler-Fig-Muster (Fowler, 2004) leitet den Verkehr durch eine Fassade und verlagert Verhalten Stück für Stück in neue Komponenten, bis das Altsystem abgeschaltet werden kann; Branch by Abstraction leistet dasselbe innerhalb einer Codebasis.

- Ein Parallelbetrieb vergleicht alte und neue Ergebnisse in der Produktion, während das alte Ergebnis maßgeblich bleibt; Dark Launching belastet neue Pfade mit echter Last, ohne dass Nutzer sie sehen.
- Daten brauchen in jeder Phase ein einziges System of Record; Change Data Capture, Event Interception und Anti-Corruption Layers halten Systeme im Gleichschritt, und die Historie wird in geprobten, abgeglichenen Kohorten migriert.
- Übergangsarchitektur ist ein bewusster, vorübergehender Aufwand; eine Migration ist erst abgeschlossen, wenn das alte System und der Übergangscode entfernt sind.

Was ein System zum Altsystem macht

Ein Altsystem ist nicht einfach ein altes System. Es ist ein System, das noch etwas Wertvolles leistet, oft etwas, ohne das die Organisation nicht arbeiten könnte, das aber schwer, riskant oder teuer zu ändern geworden ist. Michael Feathers gab in *Working Effectively with Legacy Code* (2004) eine bewusst provokante Definition: Legacy-Code ist Code ohne Tests, weil ohne Tests jede Änderung ein Glücksspiel ist. Weitere häufige Ursachen sind knappe Fachkenntnisse (eine Sprache oder Plattform, die nur noch wenige beherrschen), nicht mehr unterstützte Technologie (ein Betriebssystem oder eine Datenbank nach dem Support-Ende des Herstellers), verlorenes Wissen (die Personen, die das Design verstanden, sind gegangen) und angesammelte technische Schulden in der Struktur selbst, die teure Art, die in Abschnitt 1.7 beschrieben ist.

Man sollte offen sagen, dass die meisten Altsysteme erfolgreiche Systeme sind. Sie haben überlebt, weil sie funktionierten und Geld verdienten. Dieser Erfolg ist auch der Grund, warum sie schwer zu ersetzen sind: Jahre realer Nutzung haben Tausende kleiner Entscheidungen, Fixes und Geschäftsregeln eingebettet, die nirgends aufgeschrieben sind, außer im Code und in den Gewohnheiten der Menschen, die ihn nutzen.

Modernisierung braucht daher einen Grund, der sich in geschäftlichen Begriffen ausdrücken lässt. Gute Gründe sind: Kosten oder Risiko von Änderungen blockieren wichtige Geschäftsziele; die Plattform verliert den Herstellersupport oder Sicherheitspatches; die Fachkenntnisse für ihre Wartung sind bald nicht mehr verfügbar; die Betriebskosten stehen in keinem Verhältnis zum Wert; oder das System kann neue Anforderungen an Skalierung, Verfügbarkeit oder Regulierung nicht erfüllen. Schwache Gründe sind Mode, Langeweile und der Glaube, neue Technologie sei automatisch besser. Abschnitt 11.8 zeigt, wie aus einem guten Grund ein Business Case wird.

Warum Neuentwicklungen scheitern

Die naheliegende Reaktion auf ein schmerzhaftes altes System ist, von vorn zu beginnen. Im April 2000 veröffentlichte Joel Spolsky den Essay „Things You Should Never Do, Part I“ und argumentierte, die Entscheidung, von Grund auf neu zu schreiben, gehöre zu den schlimmsten strategischen Fehlern, die ein Softwareunternehmen machen kann. Sein Beispiel war Netscape, dessen Browser fast drei Jahre lang ohne großes Release blieb, während der Code neu geschrieben wurde, und in dieser Zeit eroberten Wettbewerber den Markt. Sein Kernpunkt betraf Wissen: Alter Code wirkt unordentlich, auch weil er Jahre an Bugfixes für Situationen enthält, an die sich niemand erinnert, und wer ihn wegwirft, wirft auch diese Fixes weg.

Die Gründe, warum eine Big-Bang-Neuentwicklung scheitert, sind inzwischen gut verstanden. Der Wert kommt erst am Ende, sodass das Geschäft Monate oder Jahre wartet und die Geduld verliert. Das alte System ändert sich weiter, während das neue gebaut wird; das Ziel verschiebt sich also, und die Organisation friert entweder das alte System (und damit das Geschäft) ein oder bezahlt dafür, jede Änderung zweimal umzusetzen. Verborgenes Verhalten wird übersehen, weil niemand jede Regel auflisten kann, die das alte System durchsetzt. Schätzungen sind optimistisch, weil die sichtbaren Features nur einen Bruchteil der tatsächlichen Arbeit ausmachen. Die Datenmigration und die Umstellung werden bis zum Ende aufgeschoben, wenn sie am schwierigsten und riskantesten sind. Und das Team spaltet sich in eine angesehene Gruppe für das neue System und eine demoralisierte Gruppe für das alte. Fred Brooks’ Second-System-Effekt aus *The Mythical Man-Month* (1975) fügt ein weiteres Risiko hinzu: Wer ein bekanntes System ersetzt, neigt dazu, das neue mit jeder Idee zu überladen, die im alten keinen Platz hatte.

Große Migrationen, die in einem Schritt umstellen, können auch sehr öffentlich scheitern. Im April 2018 verlagerte die britische Bank TSB ihre Kundendaten und Dienste auf eine neue IT-Plattform; die Plattform hatte sofort ernste Probleme, alle Filialen und ein erheblicher Teil der 5,2 Millionen Kunden waren betroffen, und die Bank kehrte erst im Dezember 2018 zum Normalbetrieb zurück. Im Dezember 2022 verhängten die Financial Conduct Authority und die Prudential Regulation Authority gegen TSB Geldbußen von insgesamt £48,65 Millionen, weil die Bank die Migration nicht angemessen geplant, organisiert und gesteuert und die Risiken ihres ausgelagerten Dienstleisters nicht beherrscht hatte.

Nicht alle Fachleute halten Neuentwicklungen für immer falsch. Eine Neuentwicklung kann die richtige Wahl sein, wenn das System klein ist, sein Verhalten gut spezifiziert und durch Tests abgedeckt ist, die Plattform wirklich in einer Sackgasse steckt oder sich das Geschäft selbst so stark verändert hat, dass das alte Verhalten nicht mehr gewünscht ist. Selbst dann werden die sichersten Neuentwicklungen schrittweise geliefert, mit altem und neuem System nebeneinander, und genau darum geht es im Rest dieses Abschnitts.

Migrationsstrategien: die Rs

Wenn eine Organisation ein ganzes Anwendungsportfolio überprüft, etwa vor dem Auszug aus einem Rechenzentrum, braucht sie ein Vokabular, um zu entscheiden, was mit jeder Anwendung geschehen soll. Gartner beschrieb Anfang der 2010er-Jahre fünf solcher Optionen. 2016 veröffentlichte Stephen Orban von AWS sechs Strategien, und AWS ergänzte später eine siebte; daraus ergeben sich die sieben Rs, die heute in den Migrationsleitfäden von AWS verwendet werden. Andere Cloud-Anbieter und Beratungen nutzen ähnliche Listen mit leicht abweichenden Namen. Die Liste ist vor allem als Erinnerung nützlich, dass Neubau nur eine von mehreren Optionen ist, und meist die teuerste.

Die sieben Rs (nach AWS Prescriptive Guidance)

Strategie	Was sie bedeutet	Passt, wenn	Architekturänderung
Retire	Die Anwendung abschalten und ihre Daten archivieren	Niemand nutzt sie, oder ihre Funktion ist anderswo doppelt vorhanden	Keine; Abhängigkeiten von ihr entfernen
Retain	Sie vorerst belassen, wo sie ist, und später erneut prüfen	Kein dringender Grund für einen Umzug, oder andere Arbeit hat Vorrang	Vorerst keine
Rehost (Lift and Shift)	Sie unverändert auf neue Infrastruktur verlagern, etwa auf virtuelle Maschinen in der Cloud	Der Haupttreiber ist, ein Rechenzentrum schnell zu verlassen	Keine; Probleme ziehen mit um
Relocate	Ganze Plattformen verlagern, etwa eine virtualisierte Umgebung, ohne die Anwendungen zu ändern	Am Ziel gibt es eine kompatible Plattform	Keine
Repurchase	Sie durch ein Produkt ersetzen, oft Software as a Service	Die Fähigkeit ist generisch, und es gibt ein gutes Produkt	Integration und Datenmigration; Verlust von Anpassungen
Replatform	Sie mit einigen gezielten Optimierungen verlagern, etwa einer Managed Database	Günstige Verbesserungen sind ohne Neuentwurf möglich	Klein und lokal
Refactor oder Re-architect	Sie neu gestalten, oft schrittweise, um neue Qualitäts- oder Geschäftsanforderungen zu erfüllen	Die Fähigkeit ist zentral, ändert sich oft, und ihre Struktur blockiert Änderungen	Groß; nutzen Sie die schrittweisen Muster weiter unten

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: eine Strategie für eine Anwendung wählen

Stellen Sie diese Fragen für jede Anwendung im Portfolio, in dieser Reihenfolge. Die Antworten kommen aus Nutzungsdaten, vom fachlich Verantwortlichen und aus der Architektur, nicht aus technologischen Vorlieben.

Q1 **Braucht noch jemand, was diese Anwendung leistet?**

- **Ja** Weiter zur nächsten Frage.
- **Nein** Legen Sie sie still (Retire), nachdem Sie alle Daten archiviert haben, die Sie gesetzlich aufbewahren müssen.

Q2 **Gibt es ein Produkt, das den Bedarf deckt, für eine Fähigkeit, die das Unternehmen nicht vom Wettbewerb abhebt?**

- **Ja** Erwägen Sie Repurchase, und planen Sie Budget für Integration und Datenmigration ein.
- **Nein** Weiter zur nächsten Frage.

Q3 **Gibt es einen dringenden Grund, jetzt zu handeln, etwa Support-Ende, Auszug aus einem Rechenzentrum, Kosten oder blockierte Änderungen?**

- **Ja** Weiter zur nächsten Frage.
- **Nein** Behalten Sie sie (Retain), und legen Sie einen Termin fest, um die Entscheidung erneut zu prüfen.

Q4 **Liegt das Hauptproblem darin, wo sie läuft, statt darin, wie sie gebaut ist?**

- **Ja** Wählen Sie Rehost oder Relocate, oder Replatform, wo ein Managed Service günstige Verbesserungen bringt.
- **Nein** Weiter zur nächsten Frage.

Q5 **Ist sie eine zentrale Fähigkeit, deren Struktur häufige, wertvolle Änderungen blockiert?**

- **Ja** Wählen Sie Refactor oder Re-architect, schrittweise, mit dem Strangler-Fig-Muster und Branch by Abstraction.
- **Nein** Wählen Sie Replatform und dämmen Sie sie ein: Tests ergänzen, die schlimmsten Hotspots beheben und einen großen Neuentwurf vermeiden.

Das Strangler-Fig-Muster

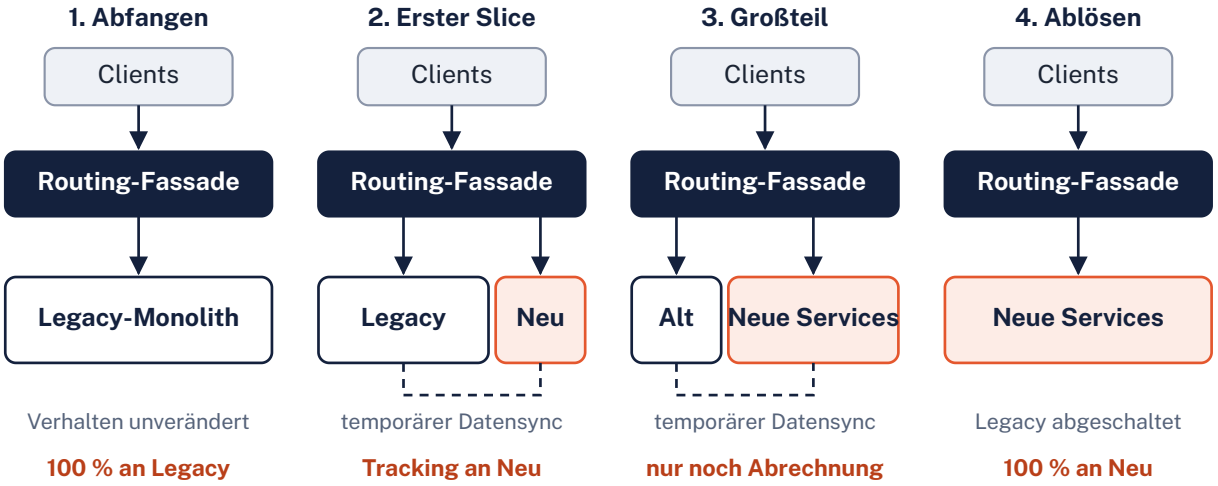
Martin Fowler benannte das am weitesten verbreitete schrittweise Muster nach den Würgefeigen (Strangler Figs), die er 2001 in den Regenwäldern von Queensland sah: Kletterpflanzen, die in den Ästen eines Wirtsbaums keimen, bis zum Boden hinunterwachsen und den Wirt allmählich umschließen, bis er abstirbt und die Feige allein steht. Er beschrieb es erstmals 2004 als Strangler Application, benannte es später in Strangler Fig Application um und überarbeitete den Artikel 2024 grundlegend. Die Softwareversion, meist einfach Strangler Fig genannt, hält das alte System am Laufen, während neue Komponenten um es herum wachsen und sein Verhalten Stück für Stück übernehmen, bis das alte System nichts mehr tut und abgeschaltet werden kann.

Der Mechanismus hat drei Teile. Erstens wird vor das Altsystem eine Abfangschicht gesetzt: ein Reverse Proxy, ein API-Gateway oder eine Routing-Fassade, durch die alle Anfragen laufen. Anfangs schickt sie alles an das alte System und ändert nichts. Zweitens wird ein Slice von Verhalten entlang einer Naht gewählt (in Feathers' Begriffen eine Stelle, an der sich Verhalten umleiten lässt, ohne viel vom alten Code zu ändern), als neue Komponente neu gebaut und durch Ändern einer Routing-Regel aktiviert, meist zunächst für einen kleinen Teil des Verkehrs. Drittens wiederholt sich der Vorgang, bis das Altsystem nichts mehr verarbeitet; dann wird es stillgelegt. Die ganze Zeit über macht die Routing-Schicht jeden Schritt umkehrbar: Verhält sich die neue Komponente falsch, wird die Regel zurückgeschaltet.

Der Code, der Alt und Neu nebeneinander existieren lässt (die Routing-Fassade, Adapter, Jobs zur Datensynchronisation), ist Übergangsarchitektur: Software, die gebaut wird, um weggeworfen zu werden. Teams sträuben sich oft dagegen, sie zu schreiben, weil es sich nach Verschwendung anfühlt. Fowlers Argument und die Erfahrung dahinter lauten, dass dieser Aufwand ein geringeres Risiko und viel früheren Nutzen erkauft, was fast immer ein guter Tausch ist. Die eigentliche Gefahr ist das Gegenteil: eine Migration, die auf halbem Weg steckenbleibt und beide Systeme samt dem gesamten Übergangscode auf unbestimmte Zeit in der Produktion zurücklässt. Jeder Slice sollte daher etwas aus dem alten System entfernen, und der Fortschritt sollte daran gemessen werden, was abgeschaltet wurde, nicht nur daran, was gebaut wurde.

Ian Cartwright, Rob Horn und James Lewis von Thoughtworks haben eine Familie verwandter Techniken als *Patterns of Legacy Displacement* auf martinfowler.com dokumentiert (ab 2021). Sie beschreiben die Arbeit als vier Tätigkeiten, die keine feste Reihenfolge haben: die gewünschten Ergebnisse verstehen, das Problem in Teile zerlegen, diese Teile schrittweise liefern und die Organisation so verändern, dass das neue System nicht so verfällt wie das alte. Sie hinterfragen auch den Reflex, jedes Feature nachzubauen: Ein Großteil des Verhaltens eines Altsystems wird vielleicht nicht mehr gebraucht, und das herauszufinden ist günstiger, als es neu zu bauen.

Eine Strangler-Fig-Migration in vier Phasen



Eine Routing-Fassade lässt den Verkehr Stück für Stück vom Altsystem zu neuen Services wandern, mit vorübergehender Datensynchronisation, bis das Altsystem abgeschaltet werden kann.

Branch by Abstraction

Das Strangler-Fig-Muster wirkt am Rand eines Systems, wo sich Anfragen routen lassen. Manche Ersetzungen müssen tief in einer einzigen Codebasis stattfinden: der Austausch eines Persistenz-Frameworks, einer Preis-Engine oder einer selbst gebauten Messaging-Schicht, die Hunderte von Aufrufstellen nutzen. Branch by Abstraction, ein Name, den Paul Hammant prägte, als er für Trunk-Based Development warb, und den Martin Fowler 2014 beschrieb, leistet das ohne langlebigen Branch in der Versionsverwaltung. Das System bleibt in jedem Schritt releasefähig, sodass sich die Änderung über Wochen oder Monate verteilen lässt, während auf derselben Hauptlinie andere Arbeit weitergeht.

Branch by Abstraction, Schritt für Schritt

1. **Eine Abstraktion einführen**
   Legen Sie eine Schnittstelle vor die zu ersetzende Komponente, mit einer Implementierung, die einfach an den bestehenden Code delegiert. Am Verhalten ändert sich nichts.

2. **Aufrufer auf die Abstraktion umstellen**
   Ändern Sie die Aufrufstellen nach und nach so, dass sie die Schnittstelle statt direkt die alte Komponente nutzen. Jede Änderung ist klein und wird normal released.

3. **Die neue Implementierung bauen**
   Schreiben Sie den Ersatz hinter derselben Schnittstelle und testen Sie ihn mit denselben Tests und idealerweise in einem Parallelbetrieb gegen das Produktionsverhalten.

4 **Schrittweise umschalten**
Nutzen Sie ein Feature-Flag oder eine Konfiguration, um einige Aufrufer oder einige Kunden an die neue Implementierung zu leiten, und erhöhen Sie den Anteil, während das Vertrauen wächst.

5 **Die alte Implementierung entfernen**
Sobald nichts mehr den alten Code nutzt, löschen Sie ihn, und entfernen Sie auch die Abstraktion, wenn sie ihren Platz nicht mehr rechtfertigt.

Gleichwertigkeit nachweisen: Parallelbetrieb und Dark Launching

Die schwierigste Frage jeder Migration ist, ob sich die neue Implementierung wirklich wie die alte verhält. Ein Parallelbetrieb beantwortet sie mit Belegen aus der Produktion: Beide Implementierungen erhalten dieselben Eingaben, die alte bleibt maßgeblich, und ihre Antwort wird an den Nutzer zurückgegeben, während die beiden Ergebnisse verglichen und alle Abweichungen festgehalten werden. GitHub veröffentlichte 2016 genau dafür seine Bibliothek Scientist, die einen Control- und einen Candidate-Pfad nebeneinander ausführt und Abweichungen meldet; ähnliche Werkzeuge gibt es für viele Sprachen, und die Technik lässt sich leicht von Hand schreiben, wie die Skizze unten zeigt.

Dark Launching, das Fowler als Aufruf neuen Backend-Verhaltens in der Produktion beschreibt, ohne dass Nutzer seine Auswirkungen sehen, ist eng verwandt. Es dient vor allem dazu, Performance und Stabilität des neuen Pfads unter echter Last zu testen, bevor ein Nutzer davon abhängt, während sich ein Parallelbetrieb auf die Korrektheit konzentriert. Beide unterscheiden sich von einem Canary Release, bei dem ein kleiner Teil der Nutzer das neue Verhalten tatsächlich erhält.

Beide Techniken haben Tücken. Der Candidate darf keine nach außen sichtbaren Nebenwirkungen haben: Er darf keine zweite E-Mail senden, keine Karte doppelt belasten und nicht in die gemeinsame Datenbank schreiben; Parallelbetrieb ist daher bei Lesezugriffen und Berechnungen am einfachsten, und Schreibzugriffe brauchen eine Sandbox oder einen separaten Speicher. Vergleiche müssen legitime Unterschiede wie Zeitstempel, Reihenfolge und Gleitkommaformatierung ignorieren, sonst geht der Abweichungsbericht im Rauschen unter. Und beide Pfade auszuführen kostet Kapazität, weshalb der Anteil des stichprobenartig geprüften Verkehrs oft begrenzt wird.

```
def quote_delivery_fee(order):
    old = legacy_pricing.quote(order)          # still authoritative
    try:
        new = new_pricing.quote(order)         # must have no side effects
        if new.amount_pence ≠ old.amount_pence:
            metrics.increment("pricing.mismatch")
            log.info("pricing mismatch", order_id=order.id,
                     old=old.amount_pence, new=new.amount_pence)
        else:
            metrics.increment("pricing.match")
    except Exception:
        metrics.increment("pricing.candidate_error")   # never hurts the user
    return old
```

Ein minimaler Parallelbetrieb: Das Legacy-Ergebnis wird weiterhin zurückgegeben, während die Antwort des Candidate verglichen und Abweichungen gezählt werden.

Daten: der schwierigste Teil

Anfragen zu routen ist relativ einfach; Daten zu verlagern und konsistent zu halten, während zwei Systeme gemeinsam für sie zuständig sind, ist der Punkt, an dem die meisten Migrationen kämpfen. Die erste Regel ist die aus Kapitel 7 zur Datenhoheit: Für jede Art von Daten muss das Team in jeder Phase der Migration sagen können, welches System das System of Record ist, welche Kopien existieren und wie veraltet jede sein darf. Das Diagramm unten zeigt den typischen Wechsel: Zuerst besitzt das Altsystem die Daten, und die neue Komponente liest eine Kopie; später besitzt die neue Komponente sie, und das Altsystem erhält eine Kopie für die Module, die sie noch brauchen.

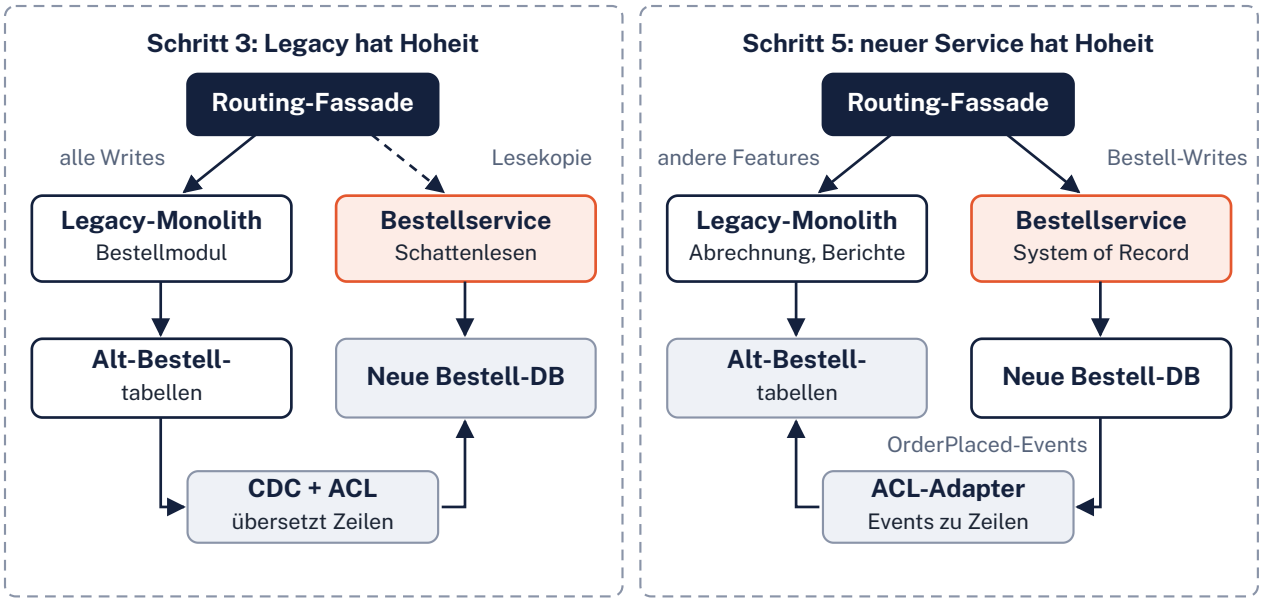
Change Data Capture ist der übliche Weg, Daten aus einer Legacy-Datenbank zu kopieren, ohne den Legacy-Code zu ändern: Ein Werkzeug wie Debezium liest das Transaktionslog der Datenbank und veröffentlicht jede Änderung als Event, derselbe Mechanismus, den Abschnitt 6.7 zum Weiterleiten einer Outbox verwendet hat. Event Interception, eines der Muster zur Ablösung von Altsystemen, setzt am anderen Ende an: Es fängt zustandsändernde Anfragen oder Nachrichten auf dem Weg in das Altsystem ab und leitet Kopien (oder später die Originale) an die neue Komponente weiter. In beiden Fällen gilt: Lassen Sie niemals beide Systeme unabhängig voneinander Schreibzugriffe auf dieselben Daten mit Zwei-Wege-Synchronisation annehmen, es sei denn, Sie haben keine Alternative; Konflikte sind dann nahezu garantiert.

Legacy-Daten passen selten zum neuen Modell. Statuscodes sind Ganzzahlen mit undokumentierter Bedeutung, eine Tabelle dient drei Zwecken, und Datumsangaben sind in Ortszeit gespeichert. Ein Anti-Corruption Layer, das Muster aus dem Domain-Driven Design,

das ein Modell vor dem eines Nachbarn schützt, übersetzt zwischen beiden, damit die Eigenheiten des Altsystems nicht in den neuen Code durchsickern. Er gehört zur Übergangsarchitektur und sollte mit dem Altsystem verschwinden.

Die abschließende Datenmigration, also die Übernahme der Historie und die Umstellung, verdient einen eigenen Plan. Profilieren Sie zuerst die Legacy-Daten, um Duplikate, verwaiste Datensätze und ungültige Werte zu finden. Schreiben Sie die Abbildung explizit auf und testen Sie sie an einer vollständigen Kopie in Produktionsgröße. Gleichen Sie nach jedem Lauf mit Zeilenzahlen, Summen der Geldspalten und Prüfsummen ab, nicht mit Stichproben. Proben Sie die Umstellung, messen Sie ihre Dauer und schreiben Sie das Rollback auf. Migrieren Sie nach Möglichkeit in Kohorten (eine Region, ein Kundensegment, eine Produktlinie) statt alles auf einmal, damit ein Problem nur einen Bruchteil der Kunden trifft und Sie vor der nächsten Kohorte etwas daraus lernen.

Das System of Record wechselt während der Migration



Anfangs kopiert Change Data Capture Legacy-Zeilen in den neuen Speicher; sobald der neue Service die Bestellungen besitzt, werden seine Events für noch nicht migrierte Module zurück in Legacy-Zeilen übersetzt.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: warum nicht zwei Systeme zugleich die Bestellungen besitzen dürfen

Die Idee, die hier die meiste Sorgfalt verlangt, ist, für jede Art von Daten ein einziges System of Record zu behalten, während die Hoheit vom alten auf das neue System übergeht. Kapitel 7 hat die Regel vermittelt; neu ist hier, dass der Eigentümer wechselt, während beide Systeme noch Nutzer haben.

- Angenommen, auf halbem Weg nehmen sowohl das Legacy-Bestellmodul als auch der neue Bestellservice Änderungen an Bestellungen an, und ein Job kopiert Änderungen alle paar Sekunden in beide Richtungen.
- Ein Kunde ändert in der App die Lieferadresse einer Bestellung; die Routing-Fassade schickt das an den neuen Bestellservice. Sekunden später storniert eine Support-Mitarbeiterin dieselbe Bestellung in den Support-Tools, die noch das Legacy-Modul nutzen.
- Kopiert der Job ganze Bestellungen, gewinnt die Kopie, die zuletzt ankommt: Die Stornierung wird vielleicht rückgängig gemacht, oder die Adressänderung geht verloren. Da sich Bestellungen den ganzen Tag ändern, sind solche Kollisionen nahezu garantiert.
- Die Abhilfe: In jeder Phase nimmt nur das System of Record Änderungen an Bestellungen an, und das andere hält eine schreibgeschützte Kopie. Anfangs ist das Legacy-Modul der Eigentümer, und Change Data Capture überträgt jede seiner Änderungen an den neuen Service.
- Später wird die Hoheit übergeben: Der neue Service nimmt Änderungen an, und seine Events werden für noch nicht migrierte Module zurück in Legacy-Zeilen übersetzt. Die Übergabe ist ein geplanter Schritt mit bereitstehendem Rollback, nie eine Phase, in der beide zuständig sind; das ausgearbeitete Beispiel unten vollzieht sie Stadt für Stadt, sodass jede Bestellung immer genau einen Eigentümer hat.
- **Die Lehre:** Benennen Sie für jede Art von Daten in jeder Phase das eine System, dem sie gehören, und in welche Richtung die Kopien fließen. Beide Systeme Änderungen an denselben Daten annehmen zu lassen, ist das letzte Mittel.

Lehren aus Kill It with Fire

Marianne Bellottis *Kill It with Fire: Manage Aging Computer Systems (and Future Proof Modern Ones)* (No Starch Press, 2021) schöpft aus ihrer Erfahrung mit der Modernisierung von Behörden- und Unternehmenssystemen, darunter Mainframe-Landschaften. Mehrere seiner Themen lohnt es, in jede Modernisierung mitzunehmen.

Modernisierung sollte durch den Wert, den sie liefert, und das Risiko, das sie beseitigt, begründet werden, nicht durch das Alter der Technologie; alte Technologie, die gut verstanden ist und funktioniert, kann der risikoärmste Teil der Landschaft sein.
Technologieentscheidungen verlaufen in Zyklen; ein Team, das ein System nur ersetzt, weil sein Design aus der Mode ist, baut daher wahrscheinlich das Altsystem der nächsten Generation. Die schrittweise Verbesserung des bestehenden Systems ist meist sicherer als ein Ersatz, und eine vollständige Neuentwicklung braucht eine ungewöhnlich starke Begründung.
Schließlich ist Modernisierung ebenso ein organisatorisches wie ein technisches Problem: Sie braucht sichtbare frühe Erfolge, um Schwung und Moral zu erhalten, klare Kriterien dafür, wann sie abgeschlossen ist, und Aufmerksamkeit für die Anreize, die das alte System verfallen ließen, sonst folgt ihm das neue.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: das Bestellmodul in sechs Schritten aus dem Monolithen herauslösen

Dieses Beispiel blickt darauf zurück, wie die Essenslieferplattform zu dem separaten Bestellservice kam, der in Abschnitt 11.1 bewertet wurde. Früher enthielt der ursprüngliche Monolith der Plattform ein Bestellmodul, das das Aufgeben, Ändern, Stornieren und Verfolgen von Bestellungen abwickelte. Änderungen an Bestellungen waren zum Hauptengpass geworden: Sie erforderten ein Deployment des gesamten Monolithen, und die Spitzen am Freitagabend verlangsamten jedes andere Feature. Vierzehn andere Module, darunter Abrechnung, Berichtswesen und die Support-Tools, lasen die Bestelltabellen direkt. Das Team plante eine schrittweise Migration zu einem separaten Bestellservice.

1. Ergebnisse festlegen und die Naht finden	Ergebnisse: Bestelländerungen täglich und unabhängig deploybar; das Dreifache der heutigen Freitagsspitze, ohne andere Features zu beeinträchtigen; keine verlorenen oder doppelten Bestellungen. Die Analyse der Abfragelogs bestätigt, dass nur das Bestellmodul in die Bestelltabellen schreibt und dass der gesamte Bestellverkehr über die Endpoints /api/orders hereinkommt. Das Team zeichnet mehrere Wochen Produktionsanfragen auf und macht daraus Charakterisierungstests, die das aktuelle Verhalten samt seinen Eigenheiten festhalten.
2. Die Routing-Fassade einfügen	Vor den Monolithen wird ein API-Gateway gesetzt, das anfangs alles an ihn weiterleitet. Für sich allein released, ändert es kein Verhalten; das Team prüft die zusätzliche Latenz (wenige Millisekunden beim 95. Perzentil) und übt das Umschalten von Routen. Rollback: das Gateway aus dem Pfad nehmen.
3. Den neuen Service im Schatten bauen	Der neue Bestellservice hat einen eigenen Speicher. Das Altsystem bleibt das System of Record: Change Data Capture kopiert Bestellzeilen hinüber, über einen Anti-Corruption Layer, der ganzzahlige Statuscodes und Datumsangaben in Ortszeit auf das neue Modell abbildet. Lese-Endpoints laufen als Parallelbetrieb gegen die Legacy-Antworten; die Abweichungsrate beginnt bei etwa 2 % (Rundung von Gebühren, Zeitzonen) und wird unter 0,01 % gedrückt, bevor es weitergeht.
4. Lesezugriffe verlagern	Lesezugriffe der Bestellverfolgung werden zu 5 %, dann zu 25 %, dann für den gesamten Verkehr an den neuen Service geleitet, wobei in jeder Phase Fehlerraten und Latenz beobachtet werden. Rollback: eine Routing-Regel.

5. Schreibzugriffe verlagern, Stadt für Stadt	Der neue Service wird zum System of Record für Bestellungen in einer Pilotstadt, dann innerhalb von sechs Wochen für jede Stadt. Er veröffentlicht OrderPlaced- und OrderCancelled-Events über eine Transactional Outbox; ein Übergangsadapter übersetzt sie in Zeilen im Legacy-Format, damit Abrechnung und Berichtswesen unverändert weiterarbeiten. Ein täglicher Abgleich vergleicht Anzahlen und Geldsummen zwischen den beiden Speichern. Diese Module wechseln dann in den folgenden Quartalen zu den Events oder zur Bestell-API.
6. Historie migrieren und ablösen	Drei Jahre historischer Bestellungen werden in gedrosselten Batches kopiert und über Anzahlen und Prüfsummen abgeglichen. Als die Audit-Logs der Datenbank 30 Tage lang keine Lesezugriffe auf die Legacy-Bestelltabellen zeigen, entfernt das Team den Adapter, die Change-Data-Capture-Pipeline und die alten Routing-Regeln und löscht dann den Bestellcode und die Tabellen aus dem Monolithen. Erst an diesem Punkt ist die Migration abgeschlossen.

Jeder Schritt liefert etwas Nutzbares oder beseitigt Risiko, und jeder Schritt außer dem letzten hat ein günstiges Rollback.

Das System of Record für Bestellungen ist in jeder Phase eindeutig, und Daten fließen jeweils nur in eine Richtung.

Die Übergangsarchitektur wird geplant, gemessen und bewusst entfernt; Erfolg wird daran gemessen, was abgeschaltet wurde.

ACHTUNG

Häufige Fehler

- Mit dem schwierigsten, zentralsten Teil des Systems beginnen statt mit einem Slice, der Wert liefert und dem Team beibringt, wie man migriert.
- Volle Feature-Parität anstreben, ohne zu fragen, welche alten Features noch genutzt werden.
- Zwei-Wege-Synchronisation, bei der beide Systeme Schreibzugriffe auf dieselben Daten annehmen.
- Migrationen, die auf halbem Weg steckenbleiben und zwei Systeme samt dem gesamten Übergangscode jahrelang laufen lassen.
- Die Datenmigration als letzte Aufgabe behandeln, statt sie von Anfang an zu entwerfen, zu proben und abzugleichen.
- Alle Änderungen am Altsystem für die gesamte Dauer einfrieren, was die Migration für das Geschäft zu einer langen Wartezeit macht.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum verliert eine Big-Bang-Neuentwicklung tendenziell Verhalten, auf das Nutzer angewiesen sind?

Weil Jahre an Bugfixes und Geschäftsregeln nur im alten Code und in den Gewohnheiten der Nutzer existieren; niemand kann sie alle auflisten, sodass eine Neuentwicklung nach Spezifikation einige davon übersieht.

▼ Was trägt die Routing-Fassade zu einer Strangler-Fig-Migration bei?

Sie lässt den Verkehr Stück für Stück verlagern und, wenn etwas schiefgeht, wieder zurückverlagern, ohne das Altsystem für jeden Schritt zu ändern.

▼ Was ist Übergangsarchitektur, und warum sollte man ihre Entfernung verfolgen?

Vorübergehender Code wie Routing-Regeln, Adapter und Synchronisationsjobs, der Alt und Neu nebeneinander bestehen lässt; wer seine Entfernung verfolgt, schützt sich vor einer Migration, die steckenbleibt, während beide Systeme auf unbestimmte Zeit laufen.

▼ Wann könnte Repurchase eine bessere Wahl sein als Refactoring?

Wenn die Fähigkeit generisch statt differenzierend ist und es ein gutes Produkt gibt, sodass ein Neubau Aufwand dort verbrauchen würde, wo das Unternehmen nicht im Wettbewerb steht.

▼ Probieren Sie es aus: Ein 15 Jahre altes Kassensystem verkauft noch Tickets, aber Änderungen sind langsam und riskant. Der Vorstand bevorzugt eine zweijährige Neuentwicklung mit einem einzigen Cutover. Was würden Sie empfehlen, und wie würden Sie während der Migration mit den Daten umgehen?

Vermeiden Sie den Big Bang: Der Wert kommt spät, Anforderungen verschieben sich, und verborgenes Verhalten geht verloren. Verwenden Sie eine Strangler Fig, indem Sie Verkehr durch eine Fassade leiten und ein Stück nach dem anderen verlagern, beginnend mit etwas Wertvollem, aber Risikoarmem wie dem Durchsuchen der Plätze. Nutzen Sie Parallelbetrieb, um Ergebnisse zu vergleichen, während das alte System maßgeblich bleibt. Jeder Datenbestand braucht in jeder Phase ein System of Record, synchron gehalten durch Change Data Capture. Entfernen Sie zum Schluss das alte System.

QUELLEN

- Martin Fowler: Strangler Fig Application: https://martinfowler.com/bliki/StranglerFigApplication.html

- Joel Spolsky: Things You Should Never Do, Part I (2000): https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/
- Cartwright, Horn and Lewis: Patterns of Legacy Displacement: https://martinfowler.com/articles/patterns-legacy-displacement/
- AWS Prescriptive Guidance: migration strategies (the 7 Rs): https://docs.aws.amazon.com/prescriptive-guidance/latest/large-migration-guide/migration-strategies.html
- Martin Fowler: Branch By Abstraction (2014): https://martinfowler.com/bliki/BranchByAbstraction.html

Führung, Einfluss und Entscheidungsfindung

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.4 Die Rolle des Architekten
- 1.6 Entscheidungen und Abwägungen
- 4.8 Kommunizieren

Begriffe von früher, die er verwendet: Architecture Decision Record (ADR), Stakeholder, Macht-Interesse-Matrix, Architektur-Kata

DIE IDEE IN KÜRZE

Stellen Sie sich eine Bergführerin vor, die eine Gruppe erfahrener Kletterer leitet. Sie kann niemandem befehlen, auf den Berg zu steigen. Die Leute folgen ihr, weil sie die Route kennt, die Risiken ehrlich erklärt, zuhört, wenn jemand eine Gefahr bemerkt, zugibt, wenn sie unsicher ist, und schon früher richtig lag. Ist sich die Gruppe über die Route uneinig, sorgt sie dafür, dass alle gehört werden, vereinbart, wie die Wahl getroffen wird, und hilft dann allen, sich darauf festzulegen. Die meisten Architekten sind in derselben Lage: verantwortlich für die Route, aber ohne Befehlsgewalt, also führen sie durch Vertrauen, klare Begründungen und gute gemeinsame Entscheidungen.

WAS SIE WISSEN MÜSSEN

- Architekten tragen meist Verantwortung ohne formale Autorität, daher beruht ihr Einfluss auf Fachwissen, Information und Beziehungen statt auf Positionsmacht (French und Raven, 1959).
- Vertrauen wächst mit Glaubwürdigkeit, Verlässlichkeit und Nähe und schrumpft mit Selbstbezogenheit (Maister, Green und Galford, 2000); psychologische Sicherheit erlaubt es Teams, Probleme anzusprechen, solange sie noch günstig zu beheben sind.
- Klären Sie Entscheidungsrechte und Entscheidungsverfahren, bevor Sie über die Sache debattieren; der Konsent nimmt einen Vorschlag an, sofern kein begründeter Einwand bleibt, während Konsens die Unterstützung aller verlangt.
- Sachgerechtes Verhandeln trennt Menschen von Problemen, konzentriert sich auf Interessen statt auf Positionen und vereinbart objektive Kriterien; wählen Sie einen Konfliktstil bewusst und eskalieren Sie gemeinsam.

- Hohpes Architect Elevator verbindet das Penthouse (Strategie) mit dem Maschinenraum (technische Realität): Risiken und Optionen gehen nach oben, Strategie und Begründungen nach unten.
- Wer andere durch Mentoring, Coaching, Sponsoring und Delegation fördert und Staff-plus-Karrierepfade unterstützt (Larson, 2021; Reilly, 2022), verhindert, dass der Architekt zum Engpass wird.
- Gruppen bringen eigene Entscheidungsfallen mit, etwa Unterordnung unter Hierarchie, Eskalation des Commitments, Bikeshedding und Entscheidung durch Zermürbung, und gegen jede gibt es eine praktische Gegenmaßnahme.

Führen ohne Autorität

Die meisten Architekten führen nicht die Menschen, die bauen, was sie entwerfen. Teams berichten an Engineering-Manager, Prioritäten liegen beim Produktmanagement, Budgets bei der Bereichsleitung. Dennoch wird vom Architekten erwartet, dafür zu sorgen, dass bedeutende Entscheidungen gut getroffen und über viele Teams hinweg eingehalten werden; das ist die Kernverantwortung, die in Abschnitt 1.4 beschrieben wurde. Die Lücke zwischen Verantwortung und formaler Autorität ist die prägende Bedingung dieser Arbeit, und sie zu schließen ist eine Frage des Einflusses.

John French und Bertram Raven unterschieden in ihrer klassischen Studie über soziale Macht (1959) Machtgrundlagen, die mit einer Position verbunden sind (legitime Autorität, die Fähigkeit zu belohnen und die Fähigkeit zu bestrafen), von solchen, die zu einer Person gehören (Expertenmacht und Identifikationsmacht, also Respekt und Identifikation, die andere empfinden); Raven ergänzte später die Informationsmacht, die Fähigkeit, Fakten einzubringen, die Meinungen ändern. Architekten haben selten viel Positionsmacht, und wenn doch, ist ihr Einsatz teuer: Eine kraft Rang auferlegte Entscheidung wird befolgt, aber nicht getragen, und wer anderer Meinung war, hört auf, Ihnen von Problemen zu berichten. Dauerhafter Einfluss in der Architektur beruht fast vollständig auf Fachwissen, Information und Beziehungen.

Das ist kein Soft Skill, der an die technische Arbeit angeflanscht wird. Ein solides Design, das niemand übernimmt, hat dieselbe Wirkung wie gar kein Design, und ein mittelmäßiges Design, das Teams verstehen und mittragen, schlägt oft ein brillantes, das sie umgehen. Der Rest dieses Abschnitts behandelt Einfluss als eine Reihe erlernbarer Praktiken: Vertrauen aufbauen, Entscheidungen moderieren, mit Konflikten umgehen, nach oben kommunizieren, andere fördern und die Fallen vermeiden, in die Gruppen tappen.

Vertrauen und Glaubwürdigkeit aufbauen

David Maister, Charles Green und Robert Galford bieten in *The Trusted Advisor* (2000) ein nützliches Modell. Vertrauen wächst mit Glaubwürdigkeit (Sie wissen, wovon Sie sprechen), Verlässlichkeit (Sie tun, was Sie sagen) und Nähe (Menschen fühlen sich sicher, Schwieriges bei Ihnen anzusprechen), und es schrumpft mit Selbstbezogenheit (dem Eindruck, dass Sie Ihren eigenen Interessen dienen, etwa recht zu behalten, klug zu wirken oder Ihr Design zu schützen). Die Erkenntnis des Modells für Architekten: Die Selbstbezogenheit steht unter dem Bruchstrich, sie teilt alles andere. Einem Architekten, der brillant und verlässlich ist, aber erkennbar jede Diskussion gewinnen muss, wird nicht vertraut.

In der Praxis entsteht Glaubwürdigkeit auf kleinen, sichtbaren Wegen. Liegen Sie bei kleinen Dingen richtig, bevor Sie bei großen um Vertrauen bitten. Bleiben Sie nah genug am Code, um seine wirklichen Kosten zu verstehen: Schreiben Sie den Spike, reviewen Sie den Pull Request, nehmen Sie an einem Incident-Review teil. Sagen Sie, was Sie nicht wissen und wie Sie es herausfinden würden. Halten Sie jede Zusage ein, besonders die langweiligen. Würdigen Sie die Leistung anderer öffentlich und übernehmen Sie die Verantwortung für Ihre eigenen Fehler, ebenfalls öffentlich. Erklären Sie die Begründung hinter jedem Prinzip, damit andere es auch in Situationen anwenden können, die Sie nicht vorhergesehen haben.

Vertrauen hängt auch vom Klima um Sie herum ab. Amy Edmondsons Forschung zur psychologischen Sicherheit, beginnend mit einer Studie über Arbeitsteams von 1999, zeigte, dass Teams, deren Mitglieder es für sicher halten, Fehler zuzugeben und andere herauszufordern, mehr lernen und bessere Leistungen erbringen. Architekten prägen dieses Klima stärker, als ihnen bewusst ist. Auf schlechte Nachrichten mit Neugier statt mit Schuldzuweisung zu reagieren, ausdrücklich um Einwände zu bitten (Was würde das scheitern lassen?) und denen zu danken, die sie vorbringen: Das macht es möglich, von einem fehlerhaften Design zu erfahren, solange es noch günstig zu beheben ist.

Entscheidungen moderieren

Viele Architekturdiskussionen sind in Wahrheit Diskussionen über das Verfahren: wer entscheiden darf, wer gefragt werden muss und wie Meinungsverschiedenheiten beigelegt werden. Werden die Entscheidungsrechte geklärt, bevor man über die Sache spricht, nimmt das viel Hitze heraus. Abschnitt 11.4 beschrieb den Advice Process, bei dem jeder eine Entscheidung treffen darf, sofern er den Rat der Betroffenen und der Fachleute einholt; andere Organisationen nutzen ein Architektur-Review-Gremium, eine benannte entscheidungsverantwortliche Person je Domäne oder Rollenmatrizen wie RACI. Der konkrete Mechanismus ist weniger wichtig, als dass er im Voraus bekannt ist.

Innerhalb der jeweils geltenden Entscheidungsrechte ist der Architekt oft der Moderator: die Person, die die Frage formuliert, dafür sorgt, dass die richtigen Leute und Belege da sind, und die Gruppe zu einer Entscheidung führt, die Bestand hat. Verschiedene Entscheidungsverfahren passen zu verschiedenen Situationen, und das Verfahren bewusst zu wählen gehört zur Aufgabe.

Zwei Verfahren werden oft verwechselt. Konsens sucht einen Vorschlag, den alle aktiv unterstützen; er schafft starkes Commitment, ist aber langsam und gibt jeder einzelnen Person faktisch ein Vetorecht. Die Konsententscheidung, entwickelt ab den 1970er-Jahren in Gerard Endenburgs Soziokratie in den Niederlanden, nimmt einen Vorschlag an, sofern niemand einen begründeten Einwand erhebt, dass er Schaden anrichten oder die Gruppe zurückwerfen würde. Die Frage ändert sich von „Sind alle einverstanden?“ zu „Kann jemand damit nicht leben, und warum?“. Ein Satz aus der soziokratischen Praxis fasst die Schwelle gut: gut genug für jetzt, sicher genug für einen Versuch. Der Konsent funktioniert besonders gut bei Zweiwegtür-Entscheidungen, die sich überprüfen lassen, sobald neue Belege vorliegen. Der Rough Consensus der IETF, beschrieben in Abschnitt 4.8, ist ein naher Verwandter.

Lässt sich keine Einigung erzielen, muss jemand entscheiden, und es sollte bereits klar sein, wer. Disagree and Commit, ebenfalls in Abschnitt 4.8 behandelt, verlangt, dass die Beteiligten ihren Standpunkt vor der Entscheidung voll vertreten und die Entscheidung danach voll mittragen. Das funktioniert nur, wenn der Widerspruch wirklich gehört wurde, der Dissens festgehalten ist (im Architecture Decision Record, mit seinen Gründen) und es ein vereinbartes Signal gibt, das die Frage wieder öffnen würde.

Ein Entscheidungsverfahren wählen

Verfahren	So funktioniert es	Einsetzen, wenn	Hauptrisiko
Allein entscheiden	Die verantwortliche Person entscheidet, eventuell nach kurzer Rückfrage	Zweiwegtüren, Notfälle, Entscheidungen mit engem Umfang	Übersehenes Wissen; geringes Commitment der anderen
Beraten, dann entscheiden (Advice Process)	Wer entscheidet, muss Rat von Betroffenen und Fachleuten einholen, entscheidet dann und hält die Gründe fest	Die meisten Architekturentscheidungen in autonomen Teams	Rat wird pro forma eingeholt und ignoriert
Konsent	Annehmen, sofern kein begründeter, schwerwiegender Einwand bleibt; Einwände verbessern den Vorschlag	Entscheidungen, die eine Gruppe betreffen und sich überprüfen lassen	Einwände als Veto aus bloßer Vorliebe missbraucht
Konsens	Weitermachen, bis alle den Vorschlag unterstützen	Kleine Gruppen, folgenreiche Werte oder Prinzipien, die tiefes Commitment brauchen	Langsam; Patt; Entscheidungen durch Nichtstun
Abstimmung	Mehrheits- oder gewichtete Abstimmung unter den Beteiligten	Wahl zwischen Optionen, die alle akzeptabel sind	Eine unterlegene Minderheit ohne Commitment; Beliebtheit vor Belegen
Delegieren	Die Entscheidung mit Vorgaben an ein Team oder eine Person näher an der Arbeit übergeben	Die Entscheidung ist lokal und die Vorgaben lassen sich benennen	Delegieren ohne Kontext oder Vorgaben

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: Konsent ist nicht Konsens

Der Punkt, an dem hier die meisten stolpern, ist der Unterschied zwischen Konsens, bei dem alle einen Vorschlag aktiv unterstützen müssen, und Konsent, bei dem er angenommen wird, sofern niemand einen begründeten Einwand erhebt. Verfolgen Sie eine kleine Entscheidung durch beide Verfahren.

- Nach der Risk-Storming-Sitzung in Abschnitt 11.2 müssen sich die sechs Teilnehmenden einigen, wann sie sich das Design erneut ansehen. Der Architekt schlägt alle drei Monate vor.
- Per Konsens: Ein Entwickler will monatliche Reviews, die Betriebsingenieurin nur vor größeren Änderungen. Die Diskussion geht weiter, bis alle sechs eine Antwort aktiv unterstützen, sodass jede einzelne Person sie aufhalten kann. Das kann sich hinziehen, bis es ins Stocken gerät oder bis die hartnäckigste Stimme gewinnt.
- Per Konsent: Die Moderation fragt: „Kann jemand mit alle drei Monate nicht leben, und warum?“ Der Entwickler bevorzugt weiterhin monatlich, kann aber mit vierteljährlich leben. Eine Vorliebe ist kein Einwand, also blockiert sie den Vorschlag nicht.
- Die Sicherheitsingenieurin erhebt einen Einwand, mit Begründung: Im nächsten Monat steht eine große Änderung am Zahlungspfad an, und eine Lücke von drei Monaten würde sie ungeprüft lassen, gerade wenn sie am neuesten und riskantesten ist. Das ist eine begründete Behauptung eines Schadens, also wird der Vorschlag verbessert: alle drei Monate, plus ein Review nach der Zahlungsänderung.
- Niemand hat einen Einwand gegen den verbesserten Vorschlag, also wird er angenommen. Er muss nicht perfekt sein: gut genug für jetzt, sicher genug für einen Versuch und später leicht zu überprüfen.
- **Die Lehre:** Der Konsent fragt „Kann jemand damit nicht leben, und warum?“ statt „Sind wir uns alle einig?“. Er passt zu Entscheidungen, die sich überprüfen lassen. Heben Sie den Konsens für die wenigen auf, die das tiefe Commitment aller brauchen, und vereinbaren Sie das Verfahren, bevor die Debatte beginnt.

Konflikte und Organisationspolitik

Organisationspolitik ist schlicht die Art, wie Gruppen mit unterschiedlichen Interessen, begrenzten Ressourcen und eigenen Karrieren zu Entscheidungen kommen. Sie ist keine Verfälschung der Architekturarbeit, sondern ihr Umfeld. Wer sie ignoriert, wird dadurch nicht prinzipientreuer, nur weniger wirksam. Die Werkzeuge aus früheren Kapiteln lassen sich direkt

anwenden: Die Stakeholder-Landkarte und das Macht-Interesse-Raster aus Abschnitt 1.5 zeigen, wer betroffen ist und wer blockieren kann, und die Dokumentation der Abwägungen aus Abschnitt 1.6 macht Positionen sichtbar.

Roger Fishers und William Urys *Getting to Yes* (1981) ist nach wie vor der nützlichste Leitfaden für sachgerechtes Verhandeln, und seine vier Ideen lassen sich direkt übertragen. Trennen Sie die Menschen vom Problem: Die Leiterin des Zahlungsteams blockiert nicht, sie sorgt sich um Kartendaten. Konzentrieren Sie sich auf Interessen, nicht auf Positionen: „Wir dürfen Kafka nicht verwenden“ ist eine Position, „Wir können keine zusätzliche Rufbereitschaftslast übernehmen“ ist ein Interesse, das mehrere Designs erfüllen könnten. Entwickeln Sie Optionen zum beiderseitigen Vorteil, bevor Sie entscheiden. Bestehen Sie auf objektiven Kriterien, die vereinbart werden, bevor Optionen verglichen werden. Auch ihre Idee der besten Alternative zu einer Verhandlungslösung (BATNA) ist nützlich: Zu wissen, was passiert, wenn keine Einigung zustande kommt, zeigt allen, wie viel die Einigung wert ist.

Das Konfliktstil-Modell von Kenneth Thomas und Ralph Kilmann (1974) beschreibt fünf Arten, auf Konflikte zu reagieren, je nachdem, wie viel Gewicht Sie Ihren eigenen Anliegen und denen der Gegenseite geben. Keine ist immer richtig; die Kunst besteht darin, bewusst zu wählen, statt in Gewohnheiten zurückzufallen.

Einige praktische Gewohnheiten helfen in jedem Fall. Überraschen Sie einen Stakeholder nie in einer öffentlichen Besprechung mit einem Vorschlag, der ihn betrifft; sprechen Sie vorher unter vier Augen mit ihm (japanische Organisationen nennen diese vorbereitende Abstimmung Nemawashi). Wenn eine Eskalation nötig ist, eskalieren Sie gemeinsam: Beide Parteien legen der Person, die entscheiden kann, die Meinungsverschiedenheit und die Optionen vor, statt getrennt Lobbyarbeit zu betreiben. Und halten Sie Meinungsverschiedenheiten über Designs getrennt von Urteilen über Menschen, in dem, was Sie sagen, und in dem, was Sie schreiben.

Fünf Konfliktstile (nach Thomas und Kilmann)

Modus	Was sie bedeutet	Nützlich, wenn	Übermäßig genutzt, führt es zu
Konkurrieren	Die eigene Position entschieden verfolgen	Fragen der Betriebssicherheit, IT-Sicherheit oder rechtliche Fragen, bei denen Sie hart bleiben müssen	Befolgung ohne Commitment; Menschen sprechen keine Probleme mehr an
Kooperieren	Gemeinsam daran arbeiten, die Interessen beider Seiten voll zu erfüllen	Wichtige Entscheidungen, bei denen die Anliegen beider Seiten berechtigt sind	Langsame Entscheidungen in Fragen, die das nicht nötig hatten
Kompromiss	Jede Seite gibt einen Teil dessen auf, was sie will	Die Zeit ist knapp und beide Seiten sind gleich mächtig	Designs, die niemanden zufriedenstellen und in sich unstimmig sind
Vermeiden	Das Thema aufschieben oder umgehen	Das Thema ist unbedeutend oder die Gemüter müssen sich abkühlen	Probleme, die wachsen, bis sie zu Krisen werden
Entgegenkommen	Der Position der anderen Seite nachgeben	Sie lagen falsch, oder das Thema ist der anderen Seite weit wichtiger	Verlorene Glaubwürdigkeit und aufgegebene Prinzipien

Mit der Geschäftsleitung kommunizieren

Abschnitt 4.8 behandelte, wie man eine Botschaft auf ihr Publikum zuschneidet und mit der Antwort beginnt. Führungskräfte verdienen etwas mehr Aufmerksamkeit, weil sie die Budgets und die Autorität innehaben, von denen Architekturarbeit abhängt, und weil Architekten sie am häufigsten verlieren, indem sie zu viel erklären und zu wenig fordern.

Stellen Sie jedes Thema in den Begriffen dar, die Führungskräfte bereits steuern: Umsatz, Kosten, Risiko, Time-to-Market, Kundenerlebnis, Regulierung und Personal. Bringen Sie Optionen statt eines einzigen Plans mit, jeweils mit ihren Kosten, ihrem Risiko und dem, was sie aufgeben, und sagen Sie, welche Sie empfehlen und warum. Quantifizieren Sie, wo Sie es ehrlich können, und sagen Sie, wie belastbar die Zahlen sind; eine Spanne mit genannter Grundlage ist glaubwürdiger als Scheingenauigkeit. Sagen Sie ausdrücklich, welche Entscheidung oder Unterstützung Sie von ihnen brauchen und bis wann. Bereiten Sie sich auf die Frage vor: Was würden Sie tun, wenn es Ihr Geld wäre? Und lassen Sie eine Führungskraft nie zuerst von jemand anderem von einem ernsten Risiko erfahren.

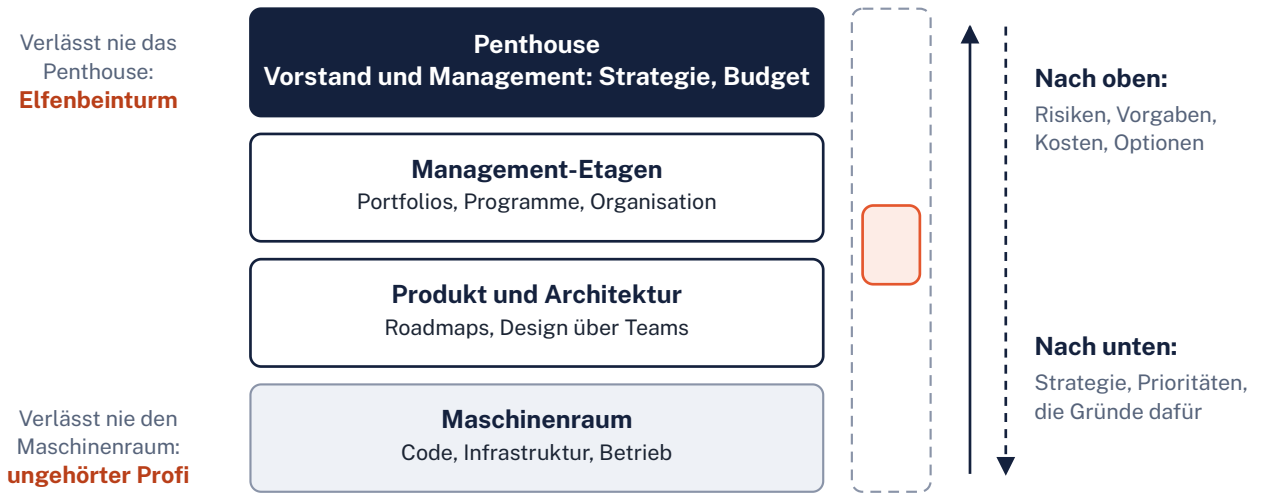
Führungskräfte interessiert oft weniger, wie ein System funktioniert, als ob seine Verantwortlichen es im Griff haben. Zu zeigen, dass Risiken bekannt sind, Verantwortliche haben, gemessen und verringert werden (zum Beispiel durch ein Architektur-Risikoregister oder die Fitness Functions aus Abschnitt 11.3), schafft mehr Vertrauen als jedes Diagramm. Abschnitt 11.8 zeigt, wie man Architekturarbeit in die Form bringt, mit der Führungskräfte Investitionen vergleichen: den Business Case.

Der Architect Elevator

Gregor Hohpes *The Software Architect Elevator* (O'Reilly, 2020) liefert das einprägsamste Bild dieser Arbeit. Große Organisationen sind wie hohe Gebäude. Im Penthouse legt die Geschäftsleitung Strategie und Budgets fest; im Maschinenraum bauen und betreiben Engineers die Systeme; dazwischen liegen viele Etagen des Managements, und Informationen, die sie durchlaufen, werden unterwegs gefiltert und verzerrt. Hohpes Architect Elevator fährt zwischen Penthouse und Maschinenraum hin und her und trägt Informationen in beide Richtungen: Strategie und ihre Gründe hinunter zu den Menschen, die Systeme bauen, und technische Realitäten (Randbedingungen, Risiken, Kosten und Optionen, von denen das Unternehmen nicht wusste, dass es sie hat) hinauf zu denen, die strategische Entscheidungen treffen.

Hohpes Punkt ist, dass der Wert des Architekten im Verbinden der Etagen liegt. Ein Architekt, der im Penthouse bleibt, wird zum Strategen im Elfenbeinturm, dessen Pläne ignorieren, was technisch möglich ist; einer, der im Maschinenraum bleibt, wird zum geschätzten Experten, dessen Warnungen nie die Menschen erreichen, die darauf reagieren könnten. Den Aufzug gut zu nutzen heißt, auf jeder Etage Sprache und Detailtiefe zu wechseln, ohne die Substanz zu ändern; genau das ist die Publikumskompetenz aus Abschnitt 4.8, angewandt auf eine ganze Organisation.

Mit dem Aufzug fahren



Der Wert des Architekten liegt im Verbinden der Etagen

Nach Gregor Hohpe: Der Architekt verbindet die Strategie im Penthouse mit der technischen Realität im Maschinenraum und übersetzt in beide Richtungen.

Andere Architekten fördern

Ein Architekt, der jede bedeutende Entscheidung selbst trifft, wird zum Engpass, und die Architekturkompetenz der Organisation geht mit ihm. Andere zu fördern ist daher Teil der Rolle, kein optionales Extra. Es hilft, drei Tätigkeiten zu unterscheiden. Mentoring gibt Ihre Erfahrung und Ihren Rat weiter. Coaching stellt Fragen, die jemandem helfen, zu einer eigenen Antwort zu kommen, und baut so schneller Urteilsvermögen auf. Sponsoring nutzt Ihre Glaubwürdigkeit, um jemandem Chancen zu verschaffen: ihn für die Leitung eines Designs empfehlen, seinen Namen auf den Vorschlag setzen, ihn zur Besprechung mit der Geschäftsleitung einladen.

Zu den praktischen Techniken gehören Pairing bei Designs und Reviews, weniger erfahrene Engineers den ADR zu einer Entscheidung schreiben zu lassen und ihn mit ihnen durchzugehen, Architektur-Katas (Abschnitt 11.2) als Übung durchzuführen, Entscheidungen innerhalb benannter Guardrails zu delegieren und in Designdiskussionen laut zu denken, damit andere sehen, wie Abwägungen gewichtet werden, und nicht nur das Ergebnis.

Viele Organisationen bieten heute einen technischen Karrierepfad parallel zum Management. Will Larsons *Staff Engineer: Leadership beyond the management track* (2021) machte den Begriff Staff-plus für Staff, Principal und noch erfahrenere Individual Contributors populär und beschrieb vier verbreitete Archetypen auf Grundlage von Interviews mit Menschen in diesen Rollen. Tanya Reillys *The Staff Engineer's Path* (O'Reilly, 2022) gliedert die Arbeit in Denken im großen Ganzen, die Umsetzung großer Projekte und die Weiterentwicklung der Engineers um

Sie herum. In ihrem Vortrag „Being Glue“ von 2019 lenkte Reilly zudem die Aufmerksamkeit auf die koordinierende Arbeit (Blockaden lösen, das fehlende Dokument schreiben, die Lücke zwischen Teams bemerken), die Projekte am Leben hält, bei Beförderungen aber oft nicht honoriert wird; Architekten sollten bemerken, wer sie leistet, und dafür sorgen, dass sie zählt.

Staff-plus-Archetypen (nach Will Larson)

Archetyp	Schwerpunkt	Typischer Beitrag zur Architektur
Tech Lead	Steuert Vorgehen und Umsetzung eines Teams, in Partnerschaft mit dessen Manager	Verantwortet die Architekturentscheidungen des Teams und hält sie im Einklang mit dem Gesamtsystem
Architect	Verantwortlich für Richtung, Qualität und Vorgehen in einem kritischen Bereich	Legt Prinzipien fest, prüft Designs und pflegt zentrale Qualitäten über mehrere Teams hinweg
Solver	Geht schwierigen, mehrdeutigen Problemen auf den Grund und findet einen Weg nach vorn, oft im Wechsel zwischen ihnen	Löst die festgefahrene Migration, die Performance-Krise oder die unklare Grenze
Right Hand	Erweitert die Aufmerksamkeit einer Führungskraft und leiht sich deren Reichweite und Autorität	Treibt organisationsweite technische Initiativen voran und verbindet Strategie mit Umsetzung

Entscheidungsfallen, noch einmal betrachtet

Abschnitt 1.6 nannte die individuellen Verzerrungen, die Architekturentscheidungen verfälschen: Ankereffekt, versunkene Kosten, Gruppendenken, lebenslaufgetriebene Entwicklung, Hype und Bestätigungsfehler. Wer Entscheidungen in Gruppen leitet, trifft auf zusätzliche eigene Fallen, die aus Status, Ermüdung und organisatorischer Eigendynamik entstehen statt aus individuellem Denken.

Fallen in Gruppen und Organisationen

Falle	Wie er aussieht	Gegenmaßnahme
Unterordnung unter die Meinung der bestbezahlten Person	Die ranghöchste Person spricht zuerst, und die Diskussion läuft auf ihre Sicht zu	Bitten Sie die ranghöchste Person, zuletzt zu sprechen; sammeln Sie schriftliche Einschätzungen vor der Besprechung
Eskalation des Commitments	Eine scheiternde Initiative erhält mehr Investitionen, weil schon so viel in sie gesteckt wurde (Barry Staw, 1976)	Legen Sie zu Beginn Abbruchkriterien und Überprüfungstermine fest; prüfen Sie mit Personen, die nicht beteiligt waren
Bikeshedding	Lange Debatte über triviale, leicht verständliche Details, während die schwierige Entscheidung nur Minuten bekommt (Parkinsons Gesetz der Trivialität, 1957)	Zeit nach Bedeutung begrenzen; Zweiwegtür-Details an das zuständige Team abgeben
Entscheidung durch Zermürbung	Die Option derjenigen Person mit dem längsten Atem gewinnt, sobald alle anderen müde sind	Entscheidungsverfahren, entscheidende Person und Frist im Voraus vereinbaren
Scheinkonsens	Schweigen wird als Zustimmung gewertet; Einwände tauchen nach der Entscheidung auf	Ausdrücklich nach Einwänden fragen, notfalls namentlich, und sie festhalten
Not invented here	Externe Lösungen werden abgelehnt, weil sie nicht intern entwickelt wurden	Selbst bauen und Zukaufen an denselben Kriterien messen, einschließlich der Betriebskosten des Selbstgebauten
Analyseparalyse	Es werden immer mehr Informationen verlangt; bis zum Last Responsible Moment ist nichts entschieden	Das Entscheidungsdatum benennen und festlegen, welche Belege genügen; umkehrbare Entscheidungen schnell treffen

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine umstrittene Entscheidung über einen Event-Bus steuern

Dieses Beispiel geht zurück zu der Entscheidung, mit der der Event Broker der Essenslieferplattform eingeführt wurde, bevor es die in Abschnitt 11.1 bewertete Architektur gab. Das Plattformteam schlägt vor, dass alle Services über einen unternehmensweiten Managed-Kafka-Cluster kommunizieren. Das Bestellteam ist begeistert, das Zahlungsteam widerspricht heftig, die Finanzabteilung sorgt sich um die Kosten, und der CTO will innerhalb eines Monats eine Entscheidung. Nach dem Advice Process des Unternehmens entscheidet der leitende Architekt des Plattformteams. Ein Principal Architect wird gebeten, zu moderieren.

1. Die Frage neu fassen	„Kafka für alles?“ ist eine Position. Das eigentliche Problem, sichtbar geworden bei zwei Ausfällen an Freitagen, ist, dass das Aufgeben von Bestellungen von synchronen Aufrufen an Benachrichtigungen und Disposition abhängt. Die Frage lautet nun: Wie sollen Services Geschäftsereignisse veröffentlichen, beginnend mit Bestellereignissen, sodass Konsumenten nicht mehr im Bestellpfad liegen? Umfang und Entscheidungsrechte werden oben in einem ADR-Entwurf festgehalten.
2. Zuerst unter vier Augen zuhören	Vor jeder Besprechung spricht die Moderation mit jeder Partei. Zahlungen: Kartendaten müssen aus neuen Systemen herausgehalten werden, und das Team kann nach einem schmerzhaften Incident mit einem selbst betriebenen Broker keine zusätzliche Rufbereitschaft übernehmen. Bestellungen: braucht Entkopplung für Lastspitzen. Finanzen: will gedeckelte Betriebskosten. Sicherheit: braucht Verschlüsselung und Zugriffskontrolle pro Topic. Das Plattformteam: will einen unterstützten Standard statt fünf. Jedes Anliegen wird als Interesse festgehalten, nicht als Position.
3. Kriterien vor Optionen vereinbaren	Im Architecture Advisory Forum einigt sich die Gruppe auf gewichtete Kriterien: Zustellgarantien und Reihenfolge pro Bestellung (5), betriebliche Last für Produktteams (5), Compliance-Umfang (4), Betriebskosten beim Dreifachen des heutigen Volumens (3), Kompetenzen (3) und Umkehrbarkeit (3).
4. Optionen entwickeln, auch die der Widersprechenden	A: Managed Kafka, verpflichtend für alle Services. B: Managed Kafka für Ereignisse der Bestelldomäne über eine Transactional Outbox, andere Teams steigen später freiwillig ein. C: die einfacheren Publish-Subscribe- und Queue-Dienste des Cloud-Anbieters. D: synchrone Aufrufe beibehalten und Wiederholungen und Circuit Breaker ergänzen. Das Zahlungsteam steuert Option D und die Kostenanalyse für C bei.
5. Belege sammeln	Ein zweiwöchiger Spike testet B und C unter produktionsnaher Last: End-to-End-Latenz, Verhalten bei einem Broker-Failover und geschätzte monatliche Kosten. Die Sicherheit bestätigt, dass Bestellereignisse so gestaltet werden können, dass sie keine Kartendaten enthalten, wodurch sie außerhalb des Geltungsbereichs für Zahlungskarten bleiben.

6. Konsentrunde	Vorschlag: B, wobei das Plattformteam den Cluster als X-as-a-Service betreibt. Das Zahlungsteam wendet ein, es könnte dennoch in den Broker-Betrieb hineingezogen werden; der Einwand wird ausgeräumt, indem Verantwortung und Supportzeiten des Plattformteams in dessen Team-API festgeschrieben werden. Das Anliegen der Finanzabteilung wird zu einem Kostenalarm und einem Überprüfungsschwellenwert. Eine Entwicklerin aus dem Bestellteam bevorzugt weiterhin C; sie kann mit B leben, und ihre Begründung wird festgehalten.
7. Entscheiden, festhalten und kommunizieren	Der Entscheider nimmt B an. Der ADR hält Optionen, Bewertungen, die Spike-Ergebnisse, den Dissens und die Auslöser für eine Überprüfung fest (mehr als drei Teams steigen ein, die Kosten überschreiten den vereinbarten Schwellenwert oder sechs Monate vergehen, je nachdem, was zuerst eintritt). CTO und CFO erhalten eine einseitige Zusammenfassung von Nutzen, Kosten und Risiko; die Teams bekommen eine Durchsprache; der Eintrag im Technologieradar wird auf Trial aktualisiert.
8. Dranbleiben	In der Pipeline jedes Produzenten läuft eine Prüfung der Schemakompatibilität, das Ausfallszenario wird erneut getestet, und beim Review nach sechs Monaten dienen die Belege dazu, zu entscheiden, ob die Entscheidung ausgeweitet, beibehalten oder rückgängig gemacht wird.

Die Moderation hat nie für eine Option argumentiert; die Arbeit floss in das Fassen der Frage, die Interessen, die Kriterien, die Belege und ein klares Entscheidungsverfahren.

Der Dissens hat das Ergebnis verändert (engerer Umfang, ausdrücklich geregelte Plattformverantwortung) und wurde festgehalten statt stillschweigend übergangen.

Die Entscheidung wurde durch begrenzten Umfang und vereinbarte Überprüfungsauslöser in der Praxis umkehrbar gemacht.

ACHTUNG

Anti-Patterns in der Architekturführung

- Diskussionen kraft Rang gewinnen, womit der Fluss schlechter Nachrichten versiegt.
- Im Stillen entscheiden und öffentlich verkünden, sodass die Beteiligung als Theater wahrgenommen wird.
- Jede Meinungsverschiedenheit als Widerstand behandeln, den man managen muss, statt als Information, die man nutzen kann.
- Der Geschäftsleitung Gewissheit versprechen, die die technische Arbeit nicht hergibt.
- Die interessanten Entscheidungen horten, sodass niemand sonst lernt, sie zu treffen.
- Bei Fragen der Betriebssicherheit, IT-Sicherheit oder Ethik neutral bleiben, wo der Architekt eine klare Linie halten sollte.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum sollten Sie vor einer Entscheidungsbesprechung unter vier Augen mit Stakeholdern sprechen?**

Um ihre Interessen vertraulich zu verstehen, sie nicht öffentlich zu überraschen und Einwände früh ans Licht zu bringen, solange sie den Vorschlag noch prägen können.

▼ **Wie unterscheiden sich Mentoring, Coaching und Sponsoring?**

Mentoring gibt Ihre Erfahrung und Ihren Rat weiter, Coaching stellt Fragen, die jemandem helfen, zu einer eigenen Antwort zu kommen, und Sponsoring nutzt Ihre Glaubwürdigkeit, um Chancen für die Person zu schaffen.

▼ **Was verliert ein Architekt, der den Maschinenraum nie verlässt?**

Die Fähigkeit, die Strategie zu beeinflussen: Sein Wissen über Risiken, Randbedingungen und Optionen erreicht nie die Menschen, die Investitions- und Organisationsentscheidungen treffen.

▼ **Warum sollte Dissens im Entscheidungsdokument festgehalten werden?**

Damit die Begründung erhalten bleibt, die Widersprechenden wissen, dass sie gehört wurden, und die Organisation, falls sich ihre Bedenken bewahrheiten, weiß, was sie überprüfen muss und warum.

▼ **Probieren Sie es aus: Das Plattformteam möchte einen gemeinsamen Kafka-Cluster; Payments lehnt das entschieden ab; Sie haben über keines der Teams Weisungsbefugnis, und der CTO will innerhalb eines Monats eine Entscheidung. Wie würden Sie damit umgehen, ohne ein Ergebnis zu erzwingen?**

Klären Sie zuerst, wer wie entscheidet, etwa per Konsent mit dem CTO als Stichentscheider. Trennen Sie dann Positionen von Interessen: Payments fürchtet vielleicht den Audit-Umfang oder Kontrollverlust, und das Plattformteam möchte weniger betreiben. Vereinbaren Sie objektive Kriterien wie Kosten, Latenz und Compliance, und prüfen Sie sie mit einem kleinen Spike. Zu den Optionen könnte ein eigener Cluster für Payments gehören. Halten Sie abweichende Meinungen im ADR fest. Wenn Sie nicht weiterkommen, eskalieren Sie gemeinsam mit einer geteilten Darstellung der Optionen, nicht als Rivalen.

QUELLEN

- Gregor Hohpe: The Architect Elevator: https://architectelevator.com/
- Will Larson: Staff archetypes: https://staffeng.com/guides/staff-archetypes/

- Andrew Harmel-Law: Scaling the Practice of Architecture, Conversationally: https://martinfowler.com/articles/scaling-architecture-conversationally.html
- RFC 7282: On Consensus and Humming in the IETF: https://www.rfc-editor.org/rfc/rfc7282

Ethik, Nachhaltigkeit und der Business Case

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.7 Technische Schulden
- 2.7 Sicherheit, Benutzbarkeit und mehr
- 8.7 Privacy by Design

Begriffe von früher, die er verwendet: Technische Schulden, Barrierefreiheit, Bedrohungsmodellierung, Altsystem (Legacy-System)

DIE IDEE IN KÜRZE

Bevor eine umsichtige Hausbesitzerin für ein neues Dach bezahlt, fragt sie, was es kostet, wie viel es an Reparaturen und Heizung spart, wie schnell es sich bezahlt macht und was passiert, wenn sie noch einen Winter wartet. Ein guter Handwerksbetrieb beantwortet diese Fragen ehrlich. Ein guter Handwerksbetrieb weigert sich auch, Leitungen zu verlegen, die einen Brand auslösen könnten, selbst wenn die Kundin es billiger will, und schlägt eine Dämmung vor, die die Energiekosten senkt. Architekten brauchen beide Gewohnheiten: die vorgeschlagene Arbeit finanziell ehrlich zu begründen und Verantwortung für den Schaden oder die Verschwendung zu übernehmen, die ihre Designs verursachen könnten.

WAS SIE WISSEN MÜSSEN

- Ein Business Case vergleicht Optionen, einschließlich des Nichtstuns, nach ihren Kosten, Nutzen und Risiken in Geld, Zeit und Wahrscheinlichkeit und empfiehlt eine davon; eine stufenweise Finanzierung verwandelt eine große Wette in kleinere Optionen.
- Der Kapitalwert diskontiert den Zahlungsstrom jedes Jahres: $NPV = \Sigma\ C_t \div (1 + r)^t$; ein positiver Kapitalwert bedeutet, dass die Investition den Diskontsatz übertrifft, den normalerweise die Finanzabteilung festlegt.
- Die Amortisationszeit ist die Zeit, bis der kumulierte Zahlungsstrom positiv wird; stellen Sie sie zusammen mit dem Kapitalwert und einer Sensitivitätsanalyse der wichtigsten Schätzungen dar.
- Technische Schulden werden zu einem Business Case, wenn die Zinsen als jährliche Kosten (Zeit, Verzögerung, Incidents, Risiko) und die Tilgungssumme als Investition ausgedrückt werden; die Behebungsaufwände von Analysetools messen die Tilgungssumme, nicht die Zinsen.

- Der ACM Code of Ethics (2018) und der BCS Code of Conduct verlangen von Architekten, das öffentliche Interesse zu berücksichtigen, Schaden zu vermeiden, Risiken gründlich zu bewerten und ehrlich darüber zu sprechen.
- Zu den vorhersehbaren Schäden gehören Verletzungen der Privatsphäre, Sicherheitsmängel, Verzerrungen (Bias), fehlende Barrierefreiheit, Gefahren für die Betriebssicherheit und manipulative Dark Patterns; mehrere davon brauchen Unterstützung durch die Architektur und sind inzwischen reguliert.
- Green-Software-Praxis umfasst CO2- und Energieeffizienz, CO2-Bewusstsein, Hardware-Effizienz, Messung und echte Reduktionen; die SCI-Spezifikation (ISO/IEC 21031:2024) misst Emissionen pro funktionaler Einheit.

Warum Architekten Business Cases brauchen

Architekturarbeit konkurriert um Geld und Personal mit allem anderen, was eine Organisation tun will. Eine Migration, eine Plattform, ein Resilienzprogramm oder der Abbau von Schulden bringt selten ein Feature hervor, das Kunden sehen, also lässt sie sich leicht zugunsten von Arbeit verschieben, die das tut. Die Architekten, die solche Arbeit finanziert bekommen, haben meist nicht die elegantesten Designs, sondern können den Fall in den Begriffen darlegen, mit denen Entscheider Investitionen vergleichen: was es kostet, was es einbringt, wie schnell, wie sicher und was passiert, wenn wir es nicht tun. Dieses strukturierte Argument ist ein Business Case.

Das ist die geschäftliche Seite von allem, was dieser Kurs über Abwägungen gesagt hat. Ein Business Case ist ein Entscheidungsdokument für eine Investition: Kontext, Optionen, Kriterien, Konsequenzen und eine Empfehlung, wobei die Konsequenzen so weit wie möglich in Geld, Zeit und Risiko ausgedrückt werden. Er ersetzt nicht das architektonische Urteil, und vieles, was Architektur schützt (Sicherheit, Lernen, Optionalität), lässt sich schwer mit einem Preis versehen. Doch wer ehrliche Zahlen nennt, wo es möglich ist, und klar sagt, wo nicht, verdient sich das Recht, auch beim Rest gehört zu werden.

Organisationen nutzen unterschiedliche Formate. Das Green Book der britischen Regierung, herausgegeben vom HM Treasury, gliedert Fälle im öffentlichen Sektor nach fünf Fragen: der strategische Fall (warum Veränderung nötig ist), der wirtschaftliche Fall (welche Option das beste Preis-Leistungs-Verhältnis bietet), der kaufmännische Fall (ob es sich beschaffen lässt), der finanzielle Fall (ob es bezahlbar ist) und der Management-Fall (ob es umsetzbar ist). Die meisten Unternehmen nutzen etwas Leichteres, doch es lohnt sich, dieselben Fragen für jede bedeutende Architekturinvestition zu beantworten.

Optionen, Kosten, Nutzen und Risiken

Jeder Business Case sollte mindestens drei Optionen vergleichen: Nichtstun (die Ausgangsbasis), das Minimum tun und die empfohlene Option, dazu alle ernsthaften Alternativen. Nichtstun ist selten kostenlos. Der Supportvertrag des Altsystems wird teurer, die Incident-Rate wächst mit dem Traffic, und Änderungen werden immer langsamer; diese Entwicklung ist die Ausgangsbasis, an der der Nutzen der anderen Optionen gemessen wird, und wer sie weglässt, lässt jede Investition schlechter aussehen, als sie ist.

Kosten und Nutzen sollten nach Kategorien aufgeführt werden, damit nichts vergessen und nichts doppelt gezählt wird. Risiken gehören ausdrücklich in den Business Case: sowohl die Risiken, die die Investition verringert, und die sich als vermiedene erwartete Verluste bewerten lassen (Wahrscheinlichkeit mal Auswirkung), als auch die Risiken der Investition selbst, etwa Überschreitungen oder Störungen während der Migration. Don Reinertsens Cost of Delay aus *The Principles of Product Development Flow* (2009) ist eine nützliche zusätzliche Perspektive: der Wert, der jede Woche verloren geht, in der eine Arbeit nicht geliefert ist. Teilt man die Cost of Delay durch die Dauer (Weighted Shortest Job First), erhält man eine gut begründbare Reihenfolge für konkurrierende Arbeiten.

Strukturieren Sie die Investition schließlich als Folge von Stufen mit Entscheidungspunkten statt als eine große Verpflichtung. Die schrittweise Migration in Abschnitt 11.6 ist ein gutes Beispiel: Jeder Schritt liefert etwas und erzeugt Belege, sodass Mittel Schritt für Schritt freigegeben und das Programm gestoppt werden kann, wenn der Nutzen ausbleibt. Solche Stage Gates verwandeln eine riskante Wette in eine Reihe kleinerer Optionen; das ist das Denken in Umkehrbarkeit aus Abschnitt 1.6, angewandt auf Geld.

Was in einen Business Case für Architektur gehört

Kategorie	Typische Posten	Wie man schätzt
Einmalige Kosten	Entwicklungszeit, Übergangsarchitektur, Datenmigration, Lizenzen, Schulungen, Parallelbetrieb, Stilllegung	Teamgröße mal Dauer mal Vollkosten-Tagessatz; Angebote von Anbietern
Laufende Kosten	Infrastruktur, Lizenzen und Abonnements, Support, Rufbereitschaft, neue Kompetenzen, die gehalten werden müssen	Aktuelle Rechnungen und Kapazitätsmodelle; Anbieterpreise beim prognostizierten Volumen
Effizienzgewinne	Weniger Aufwand pro Änderung, kürzere Durchlaufzeit, weniger manuelle Arbeit und weniger Übergaben	Änderungsvolumen mal eingesparte Zeit pro Änderung, aus Lieferdaten
Risikominderung	Weniger oder kürzere Incidents, vermiedene Sicherheitsvorfälle, vermiedene Bußgelder von Aufsichtsbehörden	Erwarteter Verlust: Häufigkeit mal Auswirkung, vorher und nachher
Umsatz und Optionen	Features, die möglich werden, Märkte, die sich öffnen, schnellere Experimente	Cost of Delay der ermöglichten Chancen; oft getrennt als Zusatzpotenzial ausgewiesen
Vermiedene Kosten	Upgrades zum Supportende, Hardware-Erneuerung, steigende Lizenzgebühren, die wegfallen	Verträge und Roadmaps der Anbieter

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: Warum späteres Geld weniger zählt

Die schwierigste Idee dieses Abschnitts ist das Diskontieren: warum Geld, das in einigen Jahren eingeht, weniger zählt als Geld heute, und wie der Kapitalwert das nutzt, um heute gezahlte Kosten gegen später eintreffenden Nutzen abzuwägen. Einfache Zahlen bereiten auf das ausgearbeitete Beispiel weiter unten vor.

- Die Migration des Bestellmoduls aus Abschnitt 11.6 kostet jetzt etwa £420.000; ihr Nutzen fällt über die nächsten vier Jahre an. Würde man die Pfund einfach addieren, behandelte man ein Pfund in Jahr 4 so, als sei es heute ein Pfund wert, und das ist es nicht.
- Geld, das man heute hat, lässt sich arbeiten lassen, und ein prognostizierter Nutzen tritt vielleicht nie ein. Deshalb legt die Finanzabteilung einen Diskontsatz fest, hier 10 %, der widerspiegelt, was Geld die Organisation kostet und wie riskant die Investition ist.
- Bei 10 % wachsen £100.000 heute in einem Jahr auf £110.000 und in zwei Jahren auf £121.000. Umgekehrt sind £121.000, die in zwei Jahren eingehen, heute £100.000 wert.
- Diskontieren ist diese Rechnung rückwärts: Teilen Sie einen Zahlungsstrom für jedes Jahr des Wartens einmal durch 1,1. Vier Divisionen machen aus einem Nutzen von £190.000 in Jahr 4 heute nur noch etwa £129.800.
- Der Kapitalwert addiert die diskontierten Zahlungsströme aller Jahre, einschließlich der Investition, die jetzt gezahlt und daher nicht diskontiert wird. Eine positive Summe bedeutet, dass die Investition mehr einbringt als den Diskontsatz. Da die Kosten zuerst und der Nutzen später anfallen, lässt das Diskontieren einen solchen Fall kleiner aussehen als eine einfache Summe.
- **Die Lehre:** Addieren Sie nie Pfund aus verschiedenen Jahren, als wären sie gleichwertig. Rechnen Sie den Zahlungsstrom jedes Jahres zum Satz der Finanzabteilung auf den heutigen Wert um und addieren Sie dann. Das Ergebnis ist nur so gut wie die Schätzungen; deshalb testet das ausgearbeitete Beispiel auch einen geringeren Nutzen.

Kapitalwert und Amortisationszeit

Geld jetzt ist mehr wert als Geld später: Es lässt sich investieren, und späteres Geld ist unsicherer. Um Zahlungsströme aus verschiedenen Jahren zu vergleichen, rechnen Finanzteams jeden in seinen heutigen Wert um, mit einem Diskontsatz, der die Kapitalkosten der Organisation und das Risiko der Investition widerspiegelt. Der Satz wird normalerweise von der Finanzabteilung festgelegt, und Architekten sollten ihn verwenden, statt einen eigenen zu wählen.

Der Kapitalwert einer Investition ist die Summe all ihrer Zahlungsströme, jeweils auf heute diskontiert: NPV = $\Sigma\ C_t \div (1 + r)^t$, summiert über die Jahre t = 0 bis n, wobei C_t der Netto-Zahlungsstrom in Jahr t ist (negativ für Kosten) und r der jährliche Diskontsatz. Der Zahlungsstrom in Jahr 0 ist die Anfangsinvestition und wird nicht diskontiert. Ein positiver Kapitalwert bedeutet, dass die Investition mehr einbringt als den Diskontsatz; bei der Wahl zwischen Optionen mit ähnlichem Risiko schafft diejenige mit dem höchsten Kapitalwert den meisten Wert.

Die Amortisationszeit ist einfacher: wie lange es dauert, bis der kumulierte Zahlungsstrom positiv wird. Führungskräfte mögen sie, weil sie leicht verständlich ist und das Risiko grob erfasst (je früher das Geld zurück ist, desto weniger kann schiefgehen), aber sie ignoriert alles nach dem Amortisationszeitpunkt. Die diskontierte Amortisationszeit führt dieselbe Rechnung mit diskontierten Zahlungsströmen durch. Stellen Sie Kapitalwert und Amortisationszeit dar und zeigen Sie, wie empfindlich beide auf die wichtigsten Schätzungen reagieren.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: der Business Case für die Ablösung des Bestellmoduls per Strangler Fig

Der Architekt der Essenslieferplattform erstellt den Business Case für die in Abschnitt 11.6 beschriebene Migration des Bestellmoduls. Die Finanzabteilung legt den Diskontsatz auf 10 % fest. Der Nutzen wird aus Liefer- und Incident-Daten geschätzt; Umsatz aus neuen Features bleibt außen vor und wird nur als Zusatzpotenzial erwähnt.

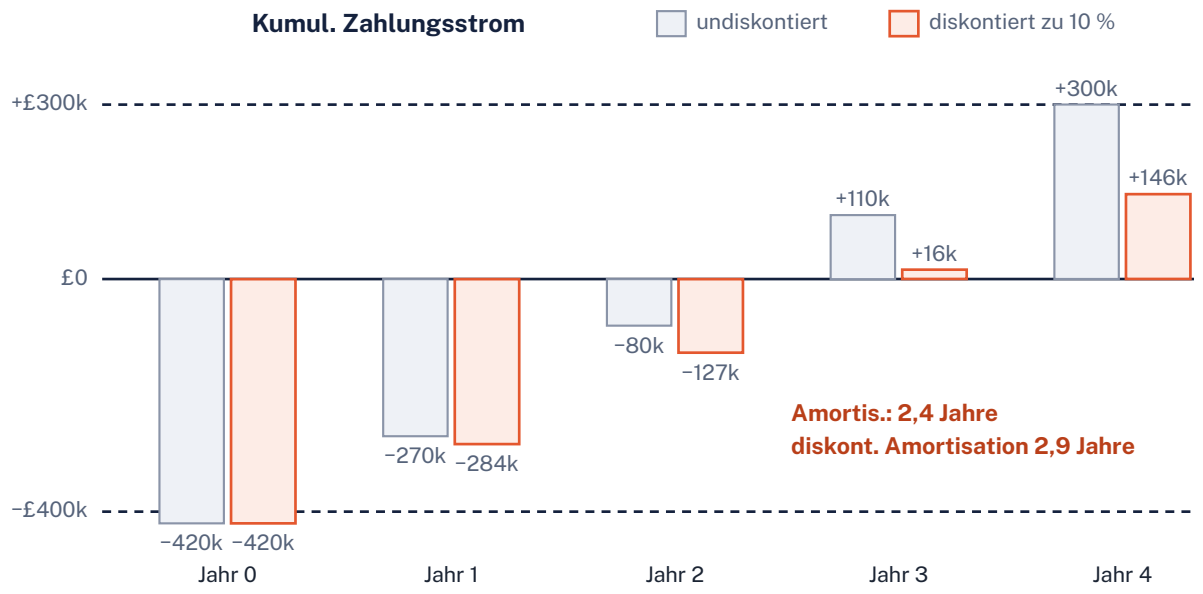
Investition (Jahr 0)	Sechs Engineers für etwa fünf Monate (etwa 630 Personentage zum selben Vollkosten-Tagessatz von £600, also £378.000), dazu Übergangsinfrastruktur und Parallelbetrieb (£42.000): £420.000.
Jährlicher Nutzen durch schnellere Änderungen (Jahre 2 bis 4)	Etwa 100 bestellbezogene Änderungen pro Jahr, die jeweils 2 Entwicklertage an Koordination, Warten auf Releases und Nacharbeit einsparen, zu einem Vollkosten-Tagessatz von £600: 100 × 2 × £600 = £120.000.
Jährlicher Nutzen durch weniger Incidents zu Spitzenzeiten	Sechs Incidents pro Jahr zur Freitagsspitze, die jeweils etwa £20.000 an Erstattungen, Gutscheinen und verlorenen Bestellungen kosten; die Migration soll zwei Drittel davon beseitigen: 4 × £20.000 = £80.000.
Zusätzliche Netto-Betriebskosten	Neuer Service und Broker abzüglich der im Monolithen frei werdenden Kapazität: −£10.000 pro Jahr.
Netto-Zahlungsströme	Jahr 0: −£420.000. Jahr 1 (der Nutzen steigt während der Migration an): +£150.000. Jahre 2, 3 und 4: jeweils +£190.000.
Diskontierte Zahlungsströme	$-420{,}000 + 150{,}000 \div 1.1 + 190{,}000 \div 1.1^2 + 190{,}000 \div 1.1^3 + 190{,}000 \div 1.1^4 =$ $-420{,}000 + 136{,}364 + 157{,}025 + 142{,}750 + 129{,}773$.
Kapitalwert	Kapitalwert ≈ +£145.900 über vier Jahre. Bei einem Diskontsatz von 15 % läge er immer noch bei etwa +£87.700.
Amortisation	Kumulierter undiskontierter Zahlungsstrom: −420.000, −270.000, −80.000, +110.000. Amortisationszeit = 2 + 80.000 ÷ 190.000 ≈ 2,4 Jahre. Mit diskontierten Zahlungsströmen wird die kumulierte Summe in Jahr 3 positiv: 2 + 126.612 ÷ 142.750 ≈ 2,9 Jahre.
Sensitivität	Fällt der Nutzen um 30 % geringer aus als geschätzt, sinkt der Kapitalwert auf etwa −£23.900. Der Business Case ist robust gegenüber dem Diskontsatz, aber empfindlich gegenüber den Nutzenschätzungen.
Empfehlung	Die Schritte 1 bis 4 der Migration genehmigen (£250.000), Durchlaufzeit von Änderungen und Incidents für Bestellungen messen und Mittel für die Schritte 5 und 6 nur freigeben, wenn die gemessenen Einsparungen mindestens 80 % der Schätzung bestätigen.

Die Rechnung ist einfach; der Wert liegt darin, die Annahmen offenzulegen, damit die Finanzabteilung sie hinterfragen und das Team sie messen kann.

Die Sensitivitätsanalyse zeigt, wo sich Aufwand lohnt: Die Einsparung an Entwicklerzeit früh zu bestätigen ist weit wichtiger, als den Diskontsatz zu verfeinern.

Die stufenweise Finanzierung verwandelt eine einzige große Wette in eine Folge von Optionen, die jeweils auf Grundlage von Belegen freigegeben werden.

Amortisation auf einen Blick



Die kumulierten Zahlungsströme werden in Jahr 3 positiv; das Diskontieren verzögert die Amortisation und verringert den Endwert auf einen Kapitalwert von etwa £146.000.

```
def npv(rate, cash_flows):
    """cash_flows[0] is year 0 (the investment, usually negative)."""
    return sum(c / (1 + rate) ** t for t, c in enumerate(cash_flows))

flows = [-420_000, 150_000, 190_000, 190_000, 190_000]
print(round(npv(0.10, flows)))        # 145911
print(round(npv(0.15, flows)))        # 87663

pessimistic = [flows[0]] + [c * 0.7 for c in flows[1:]]
print(round(npv(0.10, pessimistic)))  # -23862
```

Dieselbe Rechnung in wenigen Zeilen, praktisch, um verschiedene Annahmen zu testen.

Technische Schulden in Geld und Zeit

Abschnitt 1.7 erklärte technische Schulden als Tilgungssumme (die Kosten, ein Problem zu beheben) und Zinsen (die Mehrkosten, die bei jeder Änderung anfallen, solange es besteht), und schätzte die Zinsen für ein verworrenes Modul. Ein Business Case braucht dieselben Größen in einer Form, die die Finanzabteilung kennt. Die Zinsen werden zu jährlichen Kosten, die Tilgungssumme wird zur Investition, und die Rückzahlung der Schulden wird zu einer Investition, deren Ertrag die nicht mehr gezahlten Zinsen sind, plus etwaige Risikominderung und ermöglichte Chancen.

Mehrere Arten von Belegen lassen sich umrechnen. Entwicklerzeit: zusätzliche Tage pro Änderung, aus Vergleichen mit ähnlichen Änderungen an anderer Stelle oder aus Schätzungen der Entwicklerinnen und Entwickler, multipliziert mit dem Änderungsvolumen und einem Vollkosten-Tagessatz. Verzögerung: längere Durchlaufzeiten mal Cost of Delay der betroffenen Arbeit. Incidents: Anzahl und Dauer der auf die Schulden zurückgeführten Incidents, multipliziert mit ihren Kosten. Risiko: Wahrscheinlichkeit und Auswirkung der Ausfälle, die die Schulden wahrscheinlicher machen, etwa ein Sicherheitsvorfall über eine nicht mehr unterstützte Komponente. Chancen: Geschäftsinitiativen, die die Schulden ganz blockieren.

Seien Sie vorsichtig mit den Schuldenzahlen statischer Analysetools, die typischerweise einen geschätzten Behebungsaufwand angeben (wie lange es dauern würde, jedes gemeldete Problem zu beheben). Das ist eine Schätzung der Tilgungssumme, nicht der Zinsen, und sie sagt nichts darüber, ob der Code überhaupt je geändert wird. Schulden in selten angefasstem Code kosten wenig; die Hotspots aus Abschnitt 1.7 sind die Stellen, an denen tatsächlich Zinsen gezahlt werden und an denen die Rückzahlung den besten Ertrag bringt. Schulden als Portfolio kleiner Investitionen darzustellen, jede mit geschätztem Ertrag und Amortisationszeit, überzeugt meist weit mehr als eine einzige Schlagzeilenzahl für die Gesamtschulden.

Wardley Mapping

Business Cases vergleichen Optionen; eine **Wardley Map**, entwickelt von Simon Wardley, hilft Ihnen zu erkennen, welche Optionen es gibt. Die Karte ist an einem Nutzer und seinem Bedarf verankert. Die vertikale Achse ist die Wertschöpfungskette, oben das, was der Nutzer sehen kann, darunter das, wovon es abhängt. Die horizontale Achse ist die Evolution, von Genesis über Maßanfertigung und Produkt bis Commodity, und Komponenten wandern nach rechts, wenn der Wettbewerb sie reifen lässt. Für die Essenslieferplattform ist der Bedarf, eine Mahlzeit geliefert zu bekommen. Die Bestell-App steht weit oben; die Dispositionsoptimierung ist eine Maßanfertigung und differenziert; Kartenzahlungen sind ein Produkt; Rechenleistung ist Commodity. Etwas weit rechts Stehendes von Hand zu bauen ist ein Warnzeichen. Das GOV.UK Service Manual empfiehlt das Kartieren, um zu klären, was man bauen und was man kaufen sollte. Positionen sind Einschätzungen, der Wert liegt also im Gespräch.

Selbst bauen, kaufen oder etwas dazwischen

Der Rat des Government Digital Service ist ein solider Standard: Kaufen Sie gängige Dienste, und bauen Sie dort, wo die Fähigkeit zum Kern Ihrer Organisation gehört. Das entspricht den Core- und Generic-Subdomänen aus Abschnitt 5.7. Die Essenslieferplattform würde Zahlungen und Messaging kaufen, die Disposition aber selbst bauen. Zwischen diesen Polen liegen die Übernahme von Open Source, die Konfiguration eines Produkts und Partnerschaften. Der GDS warnt, dass starke Anpassungen teuer sind und Produkte schwerer aktualisierbar machen, bevorzugen Sie also Konfiguration. Er warnt außerdem, dass *man Risiko nicht auslagern kann*: Welchen Weg Sie auch wählen, Sie brauchen weiterhin die Menschen und das Wissen, um ihn zu überwachen. Vergleichen Sie Optionen über mehrere Jahre mit den oben genannten Methoden, und rechnen Sie Integration, Betrieb, Personalzeit, Preiserhöhungen und die Kosten des Ausstiegs ein (Abschnitt 10.8). Halten Sie den Lieferanten hinter Ihrer eigenen Schnittstelle. Überprüfen Sie die Entscheidung, während sich die Komponenten weiterentwickeln.

Anbieter bewerten und einen Proof of Concept durchführen

Legen Sie die Kriterien vor der ersten Vorführung fest, sonst bestimmt sie der beste Verkäufer. Trennen Sie K.-o.-Kriterien von bewerteten Kriterien, und gewichten Sie Letztere wie in Abschnitt 1.6. Zu den K.-o.-Kriterien gehören oft Datenstandort, Ausstiegsbedingungen und Sicherheitsnachweise. Verarbeitet ein Lieferant personenbezogene Daten, lässt Artikel 28 Absatz 1 der UK GDPR nur Auftragsverarbeiter zu, die hinreichende Garantien bieten. Prüfen Sie dann die riskanteste Behauptung mit einem **Proof of Concept**: eine Frage, ein fester Zeitrahmen und eine im Voraus vereinbarte Bestehensgrenze. Verwenden Sie Ihre eigenen Daten und Ihre Spitzenlast, und beziehen Sie Ausfälle ein: Was tut ein neuer Kartenanbieter für die Disposition, wenn seine API beim Abendessen am Freitag langsamer wird? Testen Sie nach Möglichkeit mindestens zwei Kandidaten. Behandeln Sie den Code als Wegwerfcode, und vereinbaren Sie vorab, welches Ergebnis die Bewertung beenden würde, damit nicht der bereits investierte Aufwand entscheidet.

Ethik: was die Berufskodizes verlangen

Architekturentscheidungen prägen, was Systeme mit den Menschen machen, die sie nutzen, und mit denen, die sie ohne deren Zustimmung betreffen; daher tragen Architekten neben technischer auch ethische Verantwortung. Zwei Berufskodizes sind besonders relevant. Der ACM Code of Ethics and Professional Conduct, 2018 von der Association for Computing Machinery aktualisiert, beginnt mit allgemeinen Grundsätzen: zur Gesellschaft und zum menschlichen Wohlergehen beitragen und dabei anerkennen, dass alle Menschen Stakeholder der Informatik sind; Schaden vermeiden; ehrlich und vertrauenswürdig sein; fair sein und aktiv gegen Diskriminierung vorgehen; die Arbeit respektieren, die neue Ideen erfordern; die

Privatsphäre respektieren; und Vertraulichkeit wahren. Zu den beruflichen Pflichten gehören umfassende und gründliche Bewertungen von Computersystemen und ihren Auswirkungen, einschließlich einer Analyse möglicher Risiken, sowie der Entwurf von Systemen, die robust und benutzbar sicher sind; die Führungsgrundsätze verlangen besondere Sorgfalt bei Systemen, die Teil der Infrastruktur der Gesellschaft werden.

Der BCS Code of Conduct von BCS, The Chartered Institute for IT im Vereinigten Königreich, ist um vier Pflichten herum aufgebaut: gegenüber dem öffentlichen Interesse (einschließlich öffentlicher Gesundheit, Privatsphäre, Sicherheit, Wohlergehen, Umwelt und gleichberechtigtem Zugang), gegenüber beruflicher Kompetenz und Integrität, gegenüber der maßgeblichen Stelle (Arbeitgeber oder Auftraggeber) und gegenüber dem Berufsstand. Der gemeinsame Software Engineering Code of Ethics von ACM und IEEE Computer Society (1999) ergänzt acht Grundsätze, beginnend mit dem öffentlichen Interesse, das Vorrang vor den Interessen von Auftraggeber und Arbeitgeber hat.

Für Architekten werden diese Kodizes zu konkreten Pflichten. Analysieren Sie die vorhersehbaren Auswirkungen eines Designs, nicht nur seine funktionale Korrektheit. Informieren Sie Entscheider ehrlich über Risiken, auch über solche, die sie lieber nicht hören würden. Weigern Sie sich, Systeme zu entwerfen, die täuschen oder schaden sollen, und eskalieren Sie, wenn Sie darum gebeten werden. Halten Sie Dissens aus ethischen Gründen im Entscheidungsdokument fest. Und wissen Sie, an wen Sie sich wenden können, wenn die interne Eskalation scheitert: Im Vereinigten Königreich zum Beispiel schützt der Public Interest Disclosure Act 1998 Beschäftigte, die bestimmte Arten von Fehlverhalten melden.

Vorhersehbare Schäden und Antworten der Architektur

Schaden	Beispiel Essenslieferung	Antworten der Architektur
Privatsphäre	Wohnadressen und Bestellhistorien von Kunden werden unbegrenzt aufbewahrt und breit mit der Analytik geteilt	Datenminimierung, Aufbewahrungsfristen, zweckgebundener Zugriff, Pseudonymisierung; siehe Kapitel 8 zu Privacy by Design
Sicherheit	Ausweisdokumente von Kurieren in einem Bucket, der aus dem Internet erreichbar ist	Bedrohungsmodellierung, Least Privilege, Verschlüsselung, Secrets-Management, Kontrollen der Software-Lieferkette
Bias und unfaire Ergebnisse	Ein Dispositionsalgorithmus, der Kurieren in bestimmten Gegenden oder mit älteren Fahrzeugen systematisch weniger Aufträge anbietet	Fairnesskriterien in den Anforderungen, Überwachung der Ergebnisse nach Gruppen, Erklärbarkeit, menschliche Prüfung angefochtener Entscheidungen
Barrierefreiheit	Ein Checkout, der sich mit einem Screenreader nicht abschließen lässt	WCAG 2.2 Stufe AA als Randbedingung, barrierefreie Komponenten im Designsystem, automatisierte und manuelle Tests
Betriebssicherheit (Safety)	Anreize, die Kuriere für Geschwindigkeit belohnen, oder Allergeninformationen, die uneinheitlich angezeigt werden	Anforderungen an die Betriebssicherheit und Gefährdungsanalyse, Begrenzung von Anreizen, eine einzige maßgebliche Quelle für Allergene
Manipulation	Versteckte Servicegebühren, die erst im letzten Schritt angezeigt werden; Abonnements, die leicht abzuschließen und schwer zu kündigen sind	Transparente Preise im API-Vertrag, symmetrische Wege für Anmeldung und Kündigung, Prüfung von persuasivem Design

Dark Patterns und andere vorhersehbare Schäden

Die Tabelle nennt die Schäden, die am häufigsten im Einflussbereich eines Architekten liegen. Mehrere werden an anderer Stelle ausführlich behandelt: Datenschutz und Sicherheit in Kapitel 8, Barrierefreiheit als Qualität in Abschnitt 2.7 und Bias in Systemen des maschinellen Lernens in Kapitel 12. Barrierefreiheit ist zunehmend eine rechtliche wie auch eine ethische Randbedingung; in der EU gilt der European Accessibility Act seit dem 28. Juni 2025 für viele Verbraucherdienste, darunter den elektronischen Geschäftsverkehr.

Manipulatives Design verdient besondere Erwähnung, weil es oft bewusst verlangt wird. Harry Brignull prägte 2010 den Begriff Dark Pattern für Oberflächen, die Menschen zu Entscheidungen verleiten sollen, die sie sonst nicht treffen würden; heute bevorzugt er den Begriff Deceptive Patterns. Bekannte Beispiele sind Drip Pricing (Gebühren, die erst am Ende

auftauchen), Confirmshaming (die Nein-Option soll beschämend wirken), falsche Dringlichkeit und Knappheit, vorangekreuzte Einwilligungskästchen und Abonnements, die leicht abzuschließen und schwer zu kündigen sind. Die Regulierung hat reagiert: Das Gesetz über digitale Dienste der EU (2022) verbietet Online-Plattformen, Oberflächen so zu gestalten, dass sie ihre Nutzer täuschen oder manipulieren, und die US Federal Trade Commission veröffentlichte 2022 einen Bericht ihrer Mitarbeiter über Dark Patterns. Viele dieser Muster brauchen Unterstützung durch die Architektur (eine Preis-API, die Gebühren bis zum Bezahlvorgang verbirgt, ein Kündigungsablauf, der über einen Kundenbindungsservice geleitet wird), daher sind Architekten in der Lage, sie zu bemerken und infrage zu stellen.

Der einfachste Weg, Schäden sichtbar zu machen, ist, gezielt nach ihnen zu suchen, und zwar in denselben Momenten, in denen Sie nach technischen Risiken suchen. Ergänzen Sie Bedrohungsmodellierung und Design-Reviews um eine Frage nach Schäden: Wer könnte durch dieses System zu Schaden kommen, wenn es wie vorgesehen funktioniert, wenn es ausfällt oder wenn es missbraucht wird? Praktiken wie das Consequence Scanning, entwickelt vom britischen Thinktank Doteveryone, geben Teams ein kurzes, strukturiertes Format für dieses Gespräch während der Planung.

Nachhaltigkeit: Green Software

Software verbrennt keinen Treibstoff, wohl aber der Strom und die Hardware, die sie nutzt, und Architekturentscheidungen legen einen Großteil dieses Fußabdrucks für Jahre fest. Die Green Software Foundation, 2021 unter dem Dach der Linux Foundation gegründet, beschreibt sechs Kernbereiche der Green-Software-Praxis: CO2-Effizienz (so wenig CO2 wie möglich ausstoßen), Energieeffizienz (so wenig Energie wie möglich verbrauchen), CO2-Bewusstsein (mehr tun, wenn der Strom sauberer ist, und weniger, wenn er schmutziger ist), Hardware-Effizienz (so wenig grauen Kohlenstoff wie möglich verursachen), Messung und Klimaverpflichtungen (genau verstehen, wie eine Reduktion erreicht wird).

Carbon-Aware Computing nutzt die Tatsache, dass die CO2-Intensität des Stroms je nach Stunde und Region mit dem Mix aus Wind, Sonne, Gas und anderen Quellen schwankt. Demand Shifting verlagert flexible Arbeit wie Batch-Jobs, Modelltraining, Backups und Berichtserstellung in Zeiten oder Regionen, in denen der Strom sauberer ist; Demand Shaping verringert, was die Software tut, wenn er schmutziger ist, etwa indem Medien in geringerer Auflösung ausgeliefert oder nicht dringende Neuberechnungen aufgeschoben werden. Netzdatendienste wie die Carbon Intensity API für Großbritannien und Open-Source-Tools wie das Carbon Aware SDK der Foundation liefern die Signale. Ein einfaches Beispiel: Ein nächtlicher Batch-Job mit einem Verbrauch von 400 kWh verursacht bei beispielhaft angenommenen 300 g pro kWh am frühen Abend etwa 120 kg CO_2e, in einer windigen Nacht mit 120 g pro kWh dagegen nur etwa 48 kg.

Grauer Kohlenstoff (Embodied Carbon), also die Emissionen aus Herstellung und Entsorgung von Hardware, ist der Grund, warum Hardware-Effizienz zählt: höhere Auslastung von weniger Maschinen, Skalierung auf null im Leerlauf und eine längere Lebensdauer von Geräten (auch der Smartphones der Kunden, indem man nicht für jedes Release neue Hardware verlangt) verringern ihn. Die Spezifikation Software Carbon Intensity der Foundation, veröffentlicht als ISO/IEC 21031:2024 und eingeführt in Abschnitt 2.7, kombiniert betriebliche und graue Emissionen pro funktionaler Einheit, etwa pro ausgelieferter Bestellung, und schließt Kompensationen aus, sodass nur echte Reduktionen den Wert verbessern.

Zu den Hebeln der Architektur gehören Right-Sizing und Autoscaling, das Entfernen ungenutzter Umgebungen, Aufbewahrungsfristen für Daten (gespeicherte Daten haben Energie- und Hardwarekosten), effiziente Datenformate und Caching, die Wahl von Regionen mit saubererem Strommix, wo Latenz und Datenresidenz es zulassen, und die Frage, ob wirklich jedes Feature einen Aufruf eines großen Modells braucht. Achten Sie auf den Rebound-Effekt, die moderne Form von William Stanley Jevons' Beobachtung zur Kohle von 1865: Effizienz, die etwas billiger macht, kann den Gesamtverbrauch erhöhen; messen Sie daher die Gesamtemissionen ebenso wie die Intensität.

Checkliste: ein Architektur-Review zu Ethik und Nachhaltigkeit

Verwenden Sie sie zusammen mit den technischen Review-Methoden der Abschnitte 11.1 und 11.2. Jeder Punkt sollte entweder eine festgehaltene Entscheidung oder eine Maßnahme mit verantwortlicher Person ergeben.

- [ ] **Benennen, wer betroffen ist**
  Listen Sie die vom System betroffenen Menschen auf, auch diejenigen, die es nie direkt nutzen (Kuriere, Nachbarn, Menschen, deren Daten verarbeitet werden), und ihre Interessen.

- [ ] **Fragen, wie es schaden könnte**
  Fragen Sie für den vorgesehenen Betrieb, für Ausfälle und für Missbrauch, wer zu Schaden kommen könnte und wie; halten Sie die Antworten zusammen mit dem Bedrohungsmodell fest.

- [ ] **Personenbezogene Daten minimieren**
  Prüfen Sie, dass jedes personenbezogene Datenelement einen Zweck, eine Aufbewahrungsfrist und eine Zugriffsregel hat und dass das System Löschungen umsetzen kann.

- [ ] **Fairness automatisierter Entscheidungen prüfen**
  Ermitteln Sie Entscheidungen, die Algorithmen treffen oder gewichten, legen Sie Fairnesskriterien fest und planen Sie, wie Ergebnisse nach Gruppen überwacht und angefochten werden können.

- [ ] **Barrierefreiheit bestätigen**
  Behandeln Sie WCAG 2.2 Stufe AA (oder den geltenden gesetzlichen Standard) als Randbedingung und bestätigen Sie, wie sie getestet wird.

- [ ] **Nach manipulativem Design suchen**
  Prüfen Sie die Abläufe für Preise, Einwilligung, Anmeldung und Kündigung auf versteckte Kosten, vorangekreuzte Kästchen, falsche Dringlichkeit und Asymmetrie.

- [ ] **Folgen für die Betriebssicherheit prüfen**
  Ermitteln Sie mögliche körperliche oder gesundheitliche Folgen, etwa durch Allergene, Anreize oder Fahrzeugnutzung, und die Kontrollen dafür.

- [ ] **Den CO2-Fußabdruck schätzen**
  Schätzen Sie Energie, CO2-Intensität des Stromnetzes und graue Emissionen pro funktionaler Einheit (SCI) und ermitteln Sie die größten Verursacher.

- [ ] **CO2-bewusste Optionen nutzen**
  Ermitteln Sie Arbeit, die sich zeitlich oder regional verlagern lässt, ungenutzte Kapazität, die sich entfernen lässt, und Daten, die früher gelöscht werden können.
- [ ] **Den Business Case ehrlich machen**
  Berücksichtigen Sie Betriebskosten, die Ausgangsbasis des Nichtstuns und eine Sensitivitätsanalyse; trennen Sie gemessenen Nutzen von erhofftem Zusatzpotenzial.
- [ ] **Dissens und Eskalation festhalten**
  Halten Sie ethische Einwände im Entscheidungsdokument fest und kennen Sie den Weg, um ungelöste Bedenken zu eskalieren.
- [ ] **Messung planen**
  Wählen Sie Indikatoren (Schadensmeldungen, Barrierefreiheitsmängel, SCI, Kosten pro Bestellung) und einen Termin, um sie an den obigen Erwartungen zu überprüfen.

ACHTUNG

Häufige Fehler

- Annehmen, dass die Option des Nichtstuns nichts kostet, wodurch jede Investition schlechter aussieht, als sie ist.
- Denselben Nutzen doppelt zählen, zum Beispiel als eingesparte Entwicklerzeit und noch einmal als schnellere Lieferung derselben Features.
- Scheingenauigkeit: ein auf das Pfund genauer Kapitalwert auf Basis von Schätzungen, die auf 30 % genau sind.
- Betriebskosten, Übergangskosten und Stilllegung weglassen.
- Ethik als Compliance-Häkchen am Ende behandeln statt als Frage, die während des Designs gestellt wird.
- CO2-Reduktionen über Kompensationen oder Änderungen der Bilanzierung beanspruchen statt über echte Einsparungen bei Energie und Hardware.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum muss ein Business Case eine Option des Nichtstuns enthalten?**

Weil Nichtstun eigene, steigende Kosten und Risiken hat, die die Ausgangsbasis bilden, an der der Nutzen jeder anderen Option gemessen wird.

▼ Warum sollte ein Architekt den von der Finanzabteilung festgelegten Diskontsatz verwenden, statt selbst einen zu wählen?

Weil er die Kapitalkosten und die Risikopolitik der Organisation widerspiegelt und derselbe Satz Architekturinvestitionen mit allen anderen Investitionen vergleichbar macht, die um das Geld konkurrieren.

▼ Was ist grauer Kohlenstoff, und nennen Sie einen Weg, ihn mit Mitteln der Architektur zu verringern.

Die Emissionen aus Herstellung, Transport und Entsorgung von Hardware; sie sinken durch höhere Auslastung von weniger Maschinen, Skalierung auf null im Leerlauf oder eine längere Lebensdauer von Geräten.

▼ Was ist der Unterschied zwischen Demand Shifting und Demand Shaping?

Demand Shifting verlagert flexible Arbeit in sauberere Zeiten oder Regionen; Demand Shaping verändert, wie viel Arbeit die Software leistet, je nachdem, wie sauber der Strom ist.

▼ Probieren Sie es aus: Eine Migration kostet jetzt £200.000 und spart drei Jahre lang £90.000 pro Jahr. Der Diskontsatz der Finanzabteilung beträgt 10 %. Berechnen Sie Kapitalwert (NPV) und Amortisation. Prüfen Sie dann Einsparungen von £75.000 pro Jahr. Was sagen Sie dem Vorstand?

Kapitalwert (NPV) bei £90.000: 81.818 + 74.380 + 67.618 - 200.000 = etwa +£23.800. Amortisation: etwa 2,2 Jahre. Bei £75.000 beträgt der Kapitalwert etwa -£13.500, der Business Case wird also negativ. Sagen Sie dem Vorstand, dass er die Hürde nimmt, aber knapp, und von der Einsparungsschätzung abhängt. Empfehlen Sie eine stufenweise Finanzierung: Finanzieren Sie ein erstes Stück, messen Sie die echten Einsparungen, und entscheiden Sie dann über den Rest. Zeigen Sie auch die Option, nichts zu tun.

QUELLEN

- ACM Code of Ethics and Professional Conduct: https://www.acm.org/code-of-ethics
- BCS Code of Conduct: https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/
- Green Software Foundation: Principles of green software: https://learn.greensoftware.foundation/introduction
- Software Carbon Intensity (SCI) specification: https://sci.greensoftware.foundation/
- Harry Brignull: Deceptive Patterns: https://www.deceptive.design/

- Finding a path (Simon Wardley, chapter 2): https://medium.com/wardleymaps/finding-a-path-cdb1249078c0
- On being lost (Simon Wardley, chapter 1): https://medium.com/wardleymaps/on-being-lost-2ef5f05eb1ec
- The Four Stages of Evolution (wardleymaps.com): https://www.wardleymaps.com/book-passages/wardley-on-evolution
- Choosing technology: an introduction (GOV.UK Service Manual): https://www.gov.uk/service-manual/technology/choosing-technology-an-introduction
- To build or to buy, that's the technology question (GDS, 2021): https://technology.blog.gov.uk/2021/02/03/to-build-or-to-buy-thats-the-technology-question
- UK GDPR Article 28: https://www.legislation.gov.uk/eur/2016/679/article/28

Weiterführende Literatur

- *Software Architecture in Practice, 4th edition*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. Das Standardlehrbuch; seine Kapitel zur Architekturbewertung beschreiben ATAM und ihre leichtgewichtige Variante aus der Feder der Urheber der Methode.
- *ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004)*, Rick Kazman, Mark Klein and Paul Clements (2000), Software Engineering Institute, Carnegie Mellon University. Der ursprüngliche technische Bericht, der Schritte, Rollen und Ergebnisse von ATAM festlegt, mit einem ausgearbeiteten Fall.
  https://www.sei.cmu.edu/library/atam-method-for-architecture-evaluation/
- *Building Evolutionary Architectures: Automated Software Governance, 2nd edition*, Neal Ford, Rebecca Parsons, Patrick Kua and Pramod Sadalage (2022), O'Reilly Media. Die Quelle der architektonischen Fitness Functions und die klarste Darstellung evolutionärer Architektur.
  https://oreilly.com/library/view/building-evolutionary-architectures/9781492097532
- *Performing a Project Premortem*, Gary Klein (2007), Harvard Business Review. Der kurze Artikel, der das Pre-Mortem eingeführt hat: zuerst das Scheitern vorstellen, dann es verhindern.
  https://hbr.org/2007/09/performing-a-project-premortem
- *Scaling the Practice of Architecture, Conversationally*, Andrew Harmel-Law (2021), martinfowler.com. Beschreibt den Advice Process, Beratungsforen, von Teams erarbeitete Prinzipien und ein internes Technologieradar.
  https://martinfowler.com/articles/scaling-architecture-conversationally.html
- *Design It! From Programmer to Software Architect*, Michael Keeling (2017), Pragmatic Bookshelf. Ein praktischer Leitfaden für Architektur als Teamaktivität, mit vielen Workshop-, Entwurfs- und Bewertungsaktivitäten für Teams.
- *Kill It with Fire: Manage Aging Computer Systems (and Future Proof Modern Ones)*, Marianne Bellotti (2021), No Starch Press. Ein offener, erfahrungsbasierter Leitfaden zur Modernisierung von Altsystemen ohne zum Scheitern verurteilte Neuentwicklung.

Glossar

ACM Code of Ethics

Der Code of Ethics and Professional Conduct der Association for Computing Machinery, 2018 überarbeitet; zu seinen Grundsätzen gehören ein Beitrag zur Gesellschaft, die Vermeidung von Schaden, Ehrlichkeit, Fairness, Privatsphäre und Vertraulichkeit.

Altsystem (Legacy-System)
Ein bestehendes System, das noch Nutzen stiftet, aber schwer, riskant oder teuer zu ändern geworden ist, oft wegen seines Alters, fehlender Tests, knapper Fachkenntnisse oder nicht mehr unterstützter Technologie.

Amortisationszeit
Wie lange es dauert, bis der kumulierte Nutzen die Kosten einer Investition zurückgezahlt hat; die dynamische (abgezinste) Amortisationszeit tut dasselbe mit abgezinsten Zahlungsströmen.

Anti-Corruption Layer
Eine Übersetzungsschicht aus dem Domain-Driven Design, die zwischen dem Modell eines anderen Systems und dem eigenen übersetzt, damit dessen Konzepte und Eigenheiten nicht in den eigenen Code durchsickern.

Architect Elevator
Das Bild von Gregor Hohpe für den Architekten oder die Architektin als jemand, der zwischen dem Penthouse einer Organisation (Strategie und Führungsebene) und ihrem Maschinenraum (Code und Infrastruktur) hin- und herfährt und beide verbindet.

Architecture Advisory Forum
Ein regelmäßiges, offenes Treffen, in dem vorgeschlagene Entscheidungen, Spikes und die Ratschläge dazu besprochen werden. Es bietet Rat und Transparenz, entscheidet aber selbst nicht.

Architecture Review Board
Ein ständiges Gremium erfahrener Architekten, das wesentliche Entwürfe prüft und genehmigt, meist an festgelegten Punkten eines Projekts. Es kann unabhängiges Urteil einbringen, wird aber oft zu einem langsamen Gate.

Architektur-Governance
Die Praktiken, mit denen eine Organisation sicherstellt, dass wesentliche technische Entscheidungen gut getroffen werden, mit ihren Zielen übereinstimmen und tatsächlich befolgt werden, einschließlich der Frage, wer wie und mit welchen Prüfungen entscheidet.

Architektur-Kata
Eine Übungsaufgabe, entwickelt von Ted Neward, bei der kleine Teams für eine kurze, bewusst unvollständige Aufgabenstellung eine Architektur entwerfen und diese dann vor anderen präsentieren und verteidigen.

Architekturansatz
Eine wesentliche Designentscheidung oder Kombination von Entscheidungen, etwa ein Stil, ein Muster oder eine Taktik, mit der die Architektur ein Qualitätsmerkmal erreicht: zum Beispiel ein Event Broker mit Outbox oder eine Warm-Standby-Region.

Architekturbewertung
Eine strukturierte Untersuchung einer Architektur vor oder nach ihrer Umsetzung, um herauszufinden, ob ihre Entscheidungen es dem System ermöglichen, seine Geschäftsziele und Qualitätsanforderungen zu erfüllen, und wo sie diese Ziele gefährden.

Architekturprinzip
Eine dauerhafte Regel, die Architekturentscheidungen leitet, im TOGAF-Stil formuliert mit einem Namen, einer Aussage, einer Begründung und ihren Auswirkungen.

ATAM (Architecture Tradeoff Analysis Method)
Die szenariobasierte Bewertungsmethode des SEI (Kazman, Klein und Clements, 2000). In neun Schritten über zwei Treffen setzt sie geschäftliche Treiber und priorisierte Szenarien in Beziehung zu Architekturansätzen und liefert Risiken, Nicht-Risiken, Sensitivitätspunkte, Trade-off-Punkte und Risikothemen.

Atomare Fitness Function
Eine Fitness Function, die ein Architekturmerkmal in einem einzigen Kontext prüft, etwa ein Test, der Importe zwischen zwei Paketen verbietet.

Big-Bang-Neuentwicklung
Die Ablösung eines Systems, indem ein neues von Grund auf gebaut und auf einen Schlag umgestellt wird; gilt weithin als riskant, weil der Nutzen erst am Ende eintritt und verborgenes Verhalten leicht verloren geht.

Branch by Abstraction
Eine Technik für große Änderungen innerhalb einer Codebasis: Man stellt eine Abstraktion vor den zu ersetzenden Teil, stellt die Aufrufer auf sie um, baut die neue Implementierung dahinter, schaltet um und entfernt die alte, wobei das System durchgehend releasefähig bleibt.

Bruchlinie (Fracture Plane)
Eine natürliche Nahtstelle, an der sich ein Softwaresystem so aufteilen lässt, dass jeder Teil zu einem Team passt, zum Beispiel ein Bounded Context, eine regulatorische Grenze, ein anderer Änderungsrhythmus oder ein anderes Risikoprofil.

Business Case
Eine strukturierte Begründung für eine Investition, die das Problem, die Optionen (einschließlich Nichtstun), ihre Kosten, Nutzen und Risiken sowie eine Empfehlung darlegt, die Entscheidungsträger mit anderen Verwendungen des Geldes vergleichen können.

Carbon-Aware Computing
Software so zu betreiben, dass sie mehr leistet, wenn der Strom sauberer ist, und weniger, wenn er schmutziger ist, indem Arbeit zeitlich oder örtlich verschoben oder die Nachfrage gesteuert wird.

CBAM (Cost Benefit Analysis Method)
Eine Methode des SEI, die auf der ATAM aufbaut: Sie schätzt den Nutzen jeder Architekturstrategie anhand des Utility-Gewinns für priorisierte Szenarien und teilt ihn durch ihre Kosten, um das Kosten-Nutzen-Verhältnis zu vergleichen.

Chaos-Experiment
Ein kontrollierter Test, der einen echten Ausfall herbeiführt, etwa das Beenden einer Instanz oder das Verlangsamen einer Abhängigkeit, um eine formulierte Hypothese darüber zu prüfen, wie das System seinen Normalzustand (Steady State) hält.

Charakterisierungstest
Ein Test, der festhält, was bestehender Code tatsächlich tut, und nicht, was er tun sollte, damit Änderungen am aktuellen Verhalten überprüft werden können. Der Begriff stammt von Michael Feathers.

Complicated-Subsystem-Team
Ein Team, das einen Teil des Systems verantwortet, der seltenes Spezialwissen erfordert, etwa einen Optimierungsalgorithmus oder einen Video-Codec, damit Stream-aligned Teams ihn nutzen können, ohne ihn beherrschen zu müssen.

Cost of Delay
Der Wert, der für jede Zeiteinheit verloren geht, in der eine Arbeit nicht geliefert wird; Don Reinertsen machte ihn als Mittel zur wirtschaftlichen Priorisierung von Arbeit bekannt.

Dark Launching
Das Aufrufen neuen Backend-Verhaltens in Produktion mit echtem Traffic, wobei seine Ergebnisse vor den Benutzern verborgen bleiben, um Performance und Korrektheit vor einem sichtbaren Release zu testen.

Dark Pattern
Eine Benutzeroberfläche, die Menschen zu Entscheidungen verleiten oder drängen soll, die sie sonst nicht treffen würden, etwa durch versteckte Kosten oder schwer kündbare Abonnements; Harry Brignull prägte den Begriff 2010 und nennt sie heute Deceptive Patterns.

Design-Review (RFC-Review)
Eine schriftliche Prüfung eines vorgeschlagenen Designdokuments oder Request for Comments, bei der Betroffene asynchron kommentieren, bevor das Design genehmigt und seine Entscheidungen festgehalten werden.

Die sieben Rs (Seven Rs)
Die Migrationsstrategien, die AWS für den Umzug von Anwendungen in die Cloud nennt: Retire, Retain, Rehost, Relocate, Repurchase, Replatform sowie Refactor oder Re-architect. Sie gehen auf die früheren fünf Rs von Gartner zurück.

Disagree and Commit
Eine Norm, nach der Beteiligte ihren Widerspruch vor einer Entscheidung offen äußern und sich nach der Entscheidung voll für deren Umsetzung einsetzen.

Diskontsatz
Der jährliche Zinssatz, mit dem künftige Geldbeträge in ihren heutigen Wert umgerechnet werden; er spiegelt Kapitalkosten und Risiko wider und wird meist von der Finanzabteilung festgelegt.

Enabling Team
Ein Team aus Spezialisten, das anderen Teams für begrenzte Zeit hilft, eine fehlende Fähigkeit wie Testautomatisierung oder Observability aufzubauen, und dann weiterzieht, statt die Arbeit für sie zu erledigen.

Entscheidungsrechte
Eine ausdrückliche Festlegung, wer welche Entscheidungen treffen darf, wer konsultiert und wer informiert werden muss; wer sie vor einer Debatte klärt, verhindert die meisten Streitigkeiten über das Verfahren.

Event Interception
Ein Muster zur Ablösung von Altsystemen, das zustandsändernde Anfragen oder Ereignisse auf dem Weg in ein Altsystem abfängt und einige davon zusätzlich oder stattdessen an eine neue Komponente weiterleitet.

Evolutionäre Architektur
Eine Architektur, die gelenkte, schrittweise Veränderung über mehrere Dimensionen hinweg unterstützt, wobei Fitness Functions die Lenkung übernehmen (Ford, Parsons, Kua und Sadalage).

Gate (Governance)
Ein Kontrollpunkt, an dem die Arbeit anhält, bis jemand sie freigibt, etwa die Zustimmung eines Review Boards vor Build oder Release.

Geschäftliche Treiber
Die Geschäftsziele, Randbedingungen und der Kontext, die erklären, warum ein System existiert und was Erfolg bedeutet, etwa Wachstumsziele, Regulierung oder Verträge. In der ATAM werden sie zuerst vorgestellt, damit sich jedes Ergebnis auf sie zurückführen lässt.

Grauer Kohlenstoff (Embodied Carbon)
Die Treibhausgase, die bei Herstellung, Transport und Entsorgung von Hardware entstehen, im Gegensatz zu den Emissionen aus dem Strom, den sie im Betrieb verbraucht.

Holistische Fitness Function
Eine Fitness Function, die mehrere Merkmale gemeinsam in einem geteilten Kontext prüft, etwa ein Lasttest, der die Latenz misst, während Sicherheit und Logging eingeschaltet sind.

Interaktionsmodus
In Team Topologies eine von drei bewussten Arten der Zusammenarbeit zweier Teams: Collaboration, X-as-a-Service oder Facilitating. Die Modi sollen sich ändern, während Grenzen entdeckt werden und sich festigen.

Kapitalwert (NPV)
Die Summe der künftigen Zahlungsströme einer Investition, jeweils auf den heutigen Wert abgezinst, abzüglich der Anfangskosten; ein positiver Kapitalwert bedeutet, dass die Investition mehr als den Diskontsatz erwirtschaftet.

Konsens
Eine Entscheidungsregel, bei der die Gruppe so lange weitermacht, bis alle dem Vorschlag aktiv zustimmen oder ihn zumindest unterstützen; inklusiv, aber langsam und in großen Gruppen anfällig für Blockaden.

Konsententscheidung
Eine Entscheidungsregel, bei der ein Vorschlag angenommen wird, sofern niemand einen begründeten Einwand erhebt, dass er Schaden anrichten oder die Gruppe zurückwerfen würde; sie fragt, ob die Beteiligten mit einem Vorschlag leben können, nicht, ob sie ihn bevorzugen.

Kontinuierliche Fitness Function
Eine Fitness Function, die ständig gegen das Live-System läuft, etwa SLO-Monitoring oder synthetische Transaktionen, statt durch einen Build oder ein Ereignis ausgelöst zu werden.

Latenzbudget
Der Anteil eines End-to-End-Ziels für die Antwortzeit, der jedem Schritt oder Service auf dem Weg einer Anfrage zugeteilt wird, sodass jedes Team weiß, welche Grenze es einhalten muss.

Nicht-Risiko (Non-Risk)
Eine Entscheidung, die für ein bestimmtes Szenario als tragfähig beurteilt und zusammen mit den Annahmen festgehalten wird, auf denen sie beruht, damit sie erneut geprüft werden kann, wenn sich diese Annahmen ändern.

Parallelbetrieb
Der parallele Betrieb der alten und der neuen Implementierung mit denselben Eingaben und der Vergleich ihrer Ergebnisse, während die alte maßgeblich bleibt, um vor der Umstellung nachzuweisen, dass die neue gleichwertig ist.

Paved Road (Golden Path)
Ein unterstützter, gut dokumentierter Standardweg, Software zu bauen und zu betreiben, etwa eine Service-Vorlage mit bereits eingerichtetem Logging, Sicherheit und Deployment, den Teams verlassen dürfen, wenn sie gute Gründe haben.

Plattformteam
Ein Team, das interne Dienste, Werkzeuge und Infrastruktur als Self-Service-Produkt bereitstellt, damit Stream-aligned Teams liefern können, ohne tiefes Fachwissen über die zugrunde liegende Technologie zu benötigen.

Pre-Mortem
Gary Kleins Technik (2007), bei der sich ein Team vorstellt, sein Projekt sei bereits gescheitert, und jede Person aufschreibt, warum; so kommen Risiken zum Vorschein, die sonst ungern angesprochen werden.

Psychologische Sicherheit
Die im Team geteilte Überzeugung, dass es sicher ist, zwischenmenschliche Risiken einzugehen, etwa Fehler zuzugeben oder einer erfahrenen Kollegin oder einem Vorgesetzten zu widersprechen; von Amy Edmondson seit 1999 erforscht.

Risiko (in der Architekturbewertung)
Eine Architekturentscheidung oder eine fehlende Entscheidung, die verhindern kann, dass das System eine wichtige Qualitätsanforderung oder ein wichtiges Geschäftsziel erfüllt.

Risikoregister
Eine gepflegte Liste erkannter Risiken mit Wahrscheinlichkeit, Auswirkung, Verantwortlichem, Gegenmaßnahme und Status, damit Risiken bis zu ihrem Abschluss verfolgt und nicht nach einem Meeting vergessen werden.

Risikothema
Eine Gruppe zusammenhängender Risiken mit gemeinsamer Ursache, verknüpft mit den geschäftlichen Treibern, die sie gefährdet. Risikothemen sind für das Management meist das nützlichste Ergebnis einer Bewertung.

Risk-Storming
Simon Browns kollaborative Technik, bei der Beteiligte zunächst schweigend Risiken mit Haftnotizen auf Architekturdiagrammen markieren, bewertet nach Wahrscheinlichkeit und Auswirkung, und sich dann annähern und Prioritäten und Gegenmaßnahmen vereinbaren.

Sensitivitätspunkt
Eine Eigenschaft einer oder mehrerer Komponenten oder Verbindungen, die entscheidend dafür ist, eine bestimmte Qualitätsantwort zu erreichen, etwa die Anzahl synchroner Hops auf dem Bestellpfad.

Software Carbon Intensity (SCI)
Die Spezifikation der Green Software Foundation, veröffentlicht als ISO/IEC 21031:2024, die Software als ((E × I) + M) pro R bewertet: Energie mal CO_2-Intensität des Stromnetzes plus graue Emissionen, pro funktionaler Einheit.

Soziotechnische Architektur
Der gemeinsame Entwurf von Softwarestruktur und Teamstruktur, weil jede die andere prägt. Die Idee geht auf die Forschung zu soziotechnischen Systemen am Tavistock Institute zurück, beginnend mit Eric Trists und Ken Bamforths Studie zum Kohlebergbau von 1951.

Staff-plus Engineer
Ein erfahrener Individual Contributor (Staff, Principal oder Distinguished Engineer), der teamübergreifend technische Führung ausübt, ohne Personalverantwortung zu tragen; ein von Will Larson populär gemachter Begriff.

Strangler Fig
Ein schrittweiser Modernisierungsansatz, von Martin Fowler benannt, bei dem neue Komponenten hinter einer Abfangschicht nach und nach Verhalten eines Altsystems übernehmen, bis das alte System abgeschaltet werden kann.

Stream-aligned Team
Ein langlebiges, funktionsübergreifendes Team, das auf einen Änderungsstrom ausgerichtet ist, etwa ein Produkt, eine User Journey oder eine fachliche Fähigkeit, und seinen Teil durchgängig bauen, betreiben und ändern kann. Die meisten Teams einer Organisation sollten von diesem Typ sein.

TARA
Der Tiny Architectural Review Approach, Eoin Woods' leichtgewichtige Methode zur Bewertung eines bestehenden Systems, die Expertenurteil mit automatisierter Analyse seines Codes und seiner Laufzeitdaten verbindet.

Team Topologies
Ein Ansatz zur Organisation von Softwareteams für einen schnellen Fluss, dargelegt von Matthew Skelton und Manuel Pais im gleichnamigen Buch (2019; zweite Auflage 2025): vier Teamtypen, drei Interaktionsmodi und die kognitive Last des Teams als zentrale Randbedingung.

Team-API
Alles, was ein Team anderen Teams anbietet und von ihnen erwartet: seine Code-Schnittstellen, Dokumentation, Versionierungsregeln, Arbeitsweisen, Kommunikationskanäle und wie man Arbeit bei ihm anfragt.

Technologieradar
Eine regelmäßig aktualisierte Landkarte von Technologien und Techniken, in Quadranten gruppiert und auf Ringe wie Adopt, Trial, Assess und Hold verteilt (in neueren Ausgaben von Thoughtworks in Caution umbenannt), bekannt gemacht durch Thoughtworks.

Übergangsarchitektur
Temporäre Software wie Routing-Schichten, Adapter und Synchronisierungsjobs, die nur gebaut wird, damit alte und neue Systeme während einer Migration nebeneinander bestehen können, und nach deren Abschluss entfernt wird.

X-as-a-Service
Ein Interaktionsmodus, bei dem ein Team etwas nutzt, das ein anderes Team bereitstellt, über eine klare, dokumentierte Schnittstelle und mit wenig alltäglicher Zusammenarbeit.