# Adaptive Systeme

Strategien für anpassungsfähige Software

Alexander Kaserbacher

# Alexander Kaserbacher

Berater für Softwarearchitektur

ak@embarc.de

www.embarc.de

embarc

# Montag. Kickoff-Meeting

Sechs Entwickler, ein Product Owner und ein Whiteboard …

- Neues **B2B-Produkt**. Die ersten Kunden warten.
- Die **Requirements**? Klar genug, um anzufangen (sagt der Product Owner)
- **Aufbruchstimmung**. Alles scheint machbar.

Jetzt stellt sich die Frage:

**"Welche Architektur?"**

# Was das Team weiß

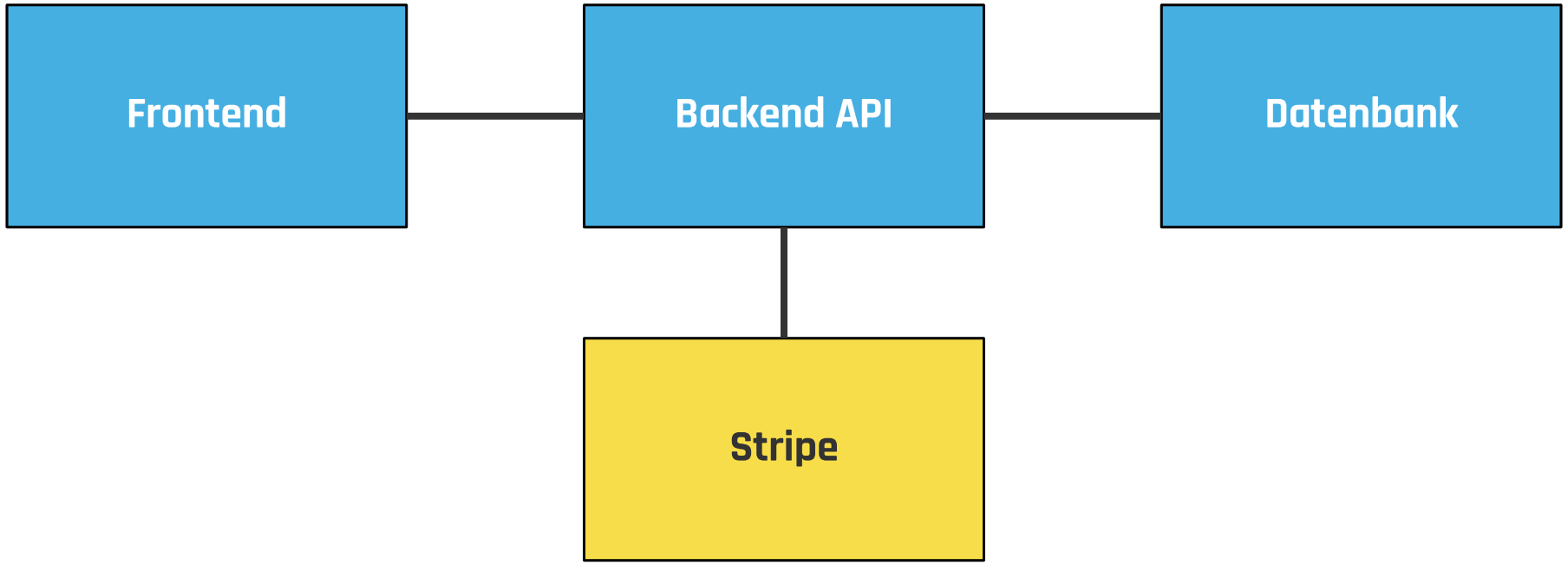
Ein SaaS-Tool für einzelne Fachleute. Ein User, ein Account, ein Abo.

**Das Team analysiert Requirements, entwirft eine Architektur, und baut…**

… schauen wir uns an, was passiert.

# Der Ausgangspunkt

Ein SaaS-Produkt für Einzelnutzer. Einfach und verständlich.



# Das Datenmodell

Die unsichtbare, aber tragende Annahme

| Users |
| --- |
| id |
| email |
| password_hash |
| name |
| subscription_tier |
| stripe_customer_id |

> **User = Account = Subscription**



# Das Team baut.

## Die Architektur geht in Produktion.

# Ein Jahr später…

Die Annahme „User = Account = Subscription“ hat sich **gefestigt**.
Alles baut darauf auf:

- **50 API-Endpunkte.** Alle auf User = Account.
- **Stripe-Integration**. Rechnet Users ab.
- Eine **Mobile-App**
- **Hunderte von Tests**. Alle auf dieser Annahme.
- Ein Team, das in diesem Modell **denkt**.



# Dann kommt der Anruf.

**Ein Kunde will sein Team einladen. Fünf Leute, ein Account.**

---

Aber User = Account = Subscription. Alles ist eins. Jeder Endpoint, jeder Test, die Billing-Logik, das mentale Modell des Teams...

# Kennt ihr das?

| | |
|---|---|
| **Denial** | "Das ist doch nur eine kleine Änderung." |
| **Anger** | "Wer hat das so designed?!" |
| **Bargaining** | "Vielleicht können wir einen Workaround bauen..." |
| **Depression** | "Das wird Monate dauern." |
| **Acceptance** | "Wir müssen alles neu bauen." |

Die **fünf Phasen der Architektur-Trauer**.

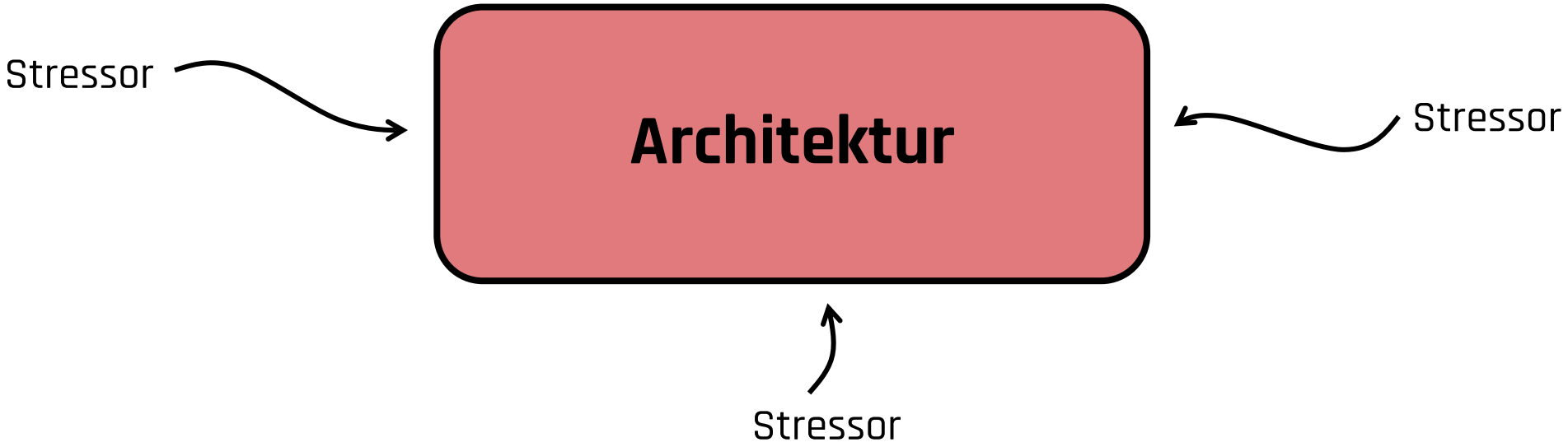


# Was wäre, wenn das nicht passieren müsste?

Reisen wir in der Zeit zurück. Nehmen wir an, das Team hätte Stress-Testing angewendet, bevor sie eine Zeile Code geschrieben haben.

# Die Architektur unter Druck setzen

Wir werfen Änderungen gegen die Architektur und schauen, was passiert. Jede Änderung nennen wir **Stressor**.




Die Ursachen/Herkunft der Stressoren sind nicht wichtig. Sie sind ein **Redeanlass** und **Diskussionsgrundlage für Verbesserungen.**

# Stressor A

## Ein Power-User will Zugang mit seiner Assistenz teilen.

| Option 1 | Option 2 |
| --- | --- |
| Login-Daten teilen | Eigenen Account erstellen |
| Sicherheitsrisiko, kein Audit-Trail | Kein Zugriff auf die Daten des Power-Users |

Die Annahme "User = Account" wird als **Einschränkung** sichtbar.

# Stressor A

## Der „Architectural Move“

Login- und Account-Daten trennen.

| Users |
| --- |
| id |
| email |
| password_hash |
| name |
| **account_id** |

→

| Accounts |
| --- |
| id |
| name |
| subscription_tier |
| stripe_customer_id |

> Alle Domänendaten gehören jetzt zu Accounts, nicht zu Users.

# Stressor B

## Ein kleines Unternehmen will fünf Teammitglieder mit unterschiedlichen Berechtigungen.

Nach Stressor A können mehrere User zu einem Account gehören. Fortschritt.

**Aber: Alle im Account können alles.**

| Admin | Editor | Viewer |
| --- | --- | --- |
| Abrechnung, Einladungen | Eigentliche Arbeit | Nur Lesen |

> Es gibt keine Möglichkeit, das auszudrücken.

# Stressor B

## Der „Architectural Move“

Die Beziehung zwischen User und Account trägt jetzt Information.

| Users |
| --- |
| id |
| email |
| password_hash |

→

| Memberships |
| --- |
| user_id |
| account_id |
| **role** |

←

| Accounts |
| --- |
| id |
| subscription_tier |
| stripe_customer_id |

**Mögliche Rollen:**

`admin` • `editor` • `viewer`

> Users und Accounts sind durch Memberships verbunden. Memberships tragen Bedeutung.

# Stressor C

## Ein Enterprise-Kunde braucht SSO und zentrale Admin-Steuerung.

### Problem 1: SSO

User authentifizieren sich über Okta, Azure AD, etc.

Aber password_hash ist in unserer Users-Tabelle.

### Problem 2: Admin-Steuerung

IT-Abteilung will User zentral verwalten: SCIM, Provisioning

Nicht über unsere UI.

Authentifizierung und Identität sind miteinander vermischt.

# Stressor C

## Der „Architectural Move“

Authentifizierung extrahieren.


> Authentifizierung ist jetzt austauschbar. Die Identität eines Users existiert unabhängig von der Login-Art.



# Das Team baut.

## Die Architektur geht in Produktion.

# Ein Jahr später…

Die Annahme „User = Account = Subscription“ hat sich **gefestigt**. Alles baut darauf auf:

- **50 API-Endpunkte.** Alle auf Membership und Accounts.
- **Stripe-Integration**. Rechnet Accounts ab.
- Eine **Mobile-App**
- **Hunderte von Tests**. Alle prüfen auf Membership und Permissions.
- Ein Team, das in diesem Modell **denkt**: Users haben Memberships zu Accounts.

# Dann kommt der Anruf.

## Ein Reseller will mehrere Kundenaccounts von einem Dashboard aus verwalten.

# Zwei Teams, eine Änderung

## Ohne Stress-Testing

- Schema-Migration (live)
- 50+ API-Endpoints umschreiben
- Stripe-Integration neu anbinden
- Hunderte Tests anpassen
- Team-Mentalmodell ändern
- Mobile App updaten

Wochen bis Monate

## Mit Stress-Testing

✓ Users existieren unabhängig

✓ Memberships verbinden zu Accounts

✓ Rollen sind bereits flexibel

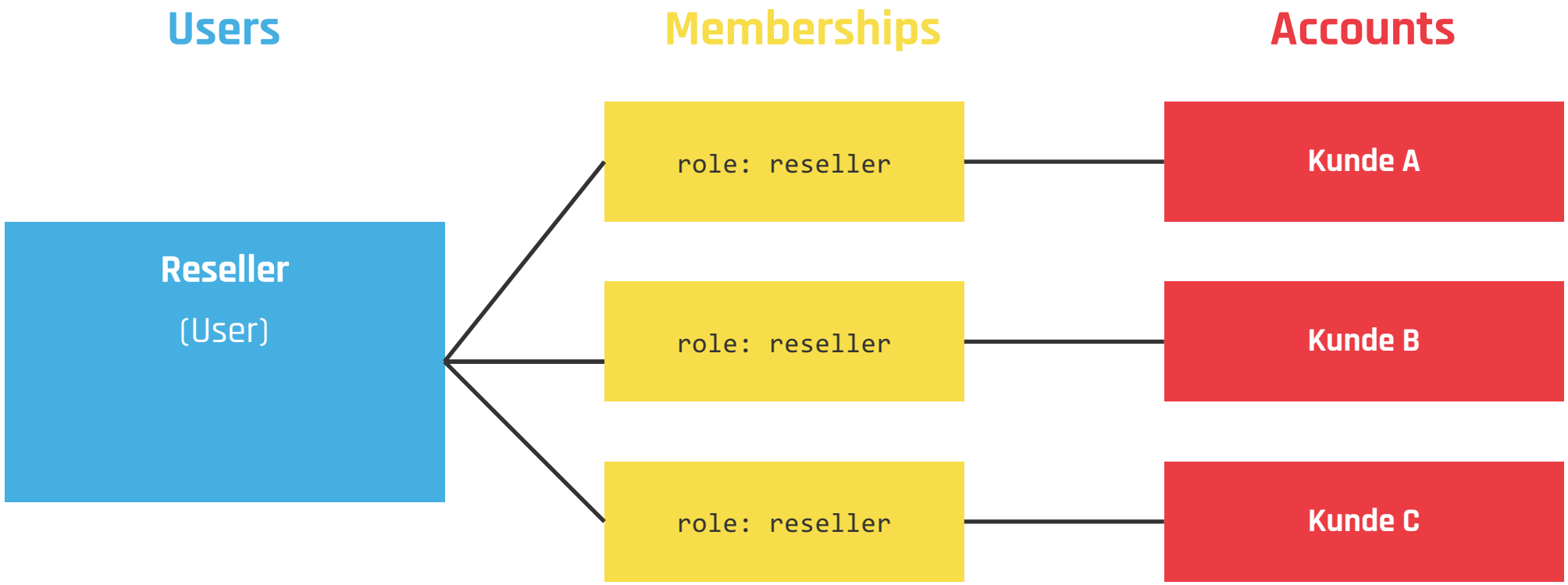
✓ Billing läuft über Accounts

Reseller = neuer Membership-Typ

Tage

> Beide Teams haben ein Jahr lang gebaut. Der Unterschied liegt in der **Struktur**, nicht im Aufwand.

# Die Struktur hält




**Keine neuen Tabellen. Kein Rekeying von Daten. Kein grundlegendes Redesign.**

## Normalerweise?

Denial → Anger → Bargaining → Depression → Acceptance

Wochen bis Monate Umbau. Verpasste Deadlines. Frustrierte Entwickler.

## Aber hier?

## Reseller = neuer Membership-Typ.

Das war's.

Keine der drei Stressoren erwähnte Reseller. Aber die Architektur hat es trotzdem überlebt.



# Das ist, was Stress-Testing produziert.

Keine Architektur, die für vorhergesagte Anforderungen designed ist.

Eine Architektur mit der Kapazität, Änderungen zu absorbieren, die du noch nicht vorhergesehen hast.

Weil die Schwäche, die ein Stressor aufdeckt, breiter ist als der Stressor selbst.



# Was ist ein Adaptives System?

> Ein System, dessen Architektur Änderungen **absorbieren** kann, die bei seiner Entstehung noch **nicht vorherzusehen** waren.

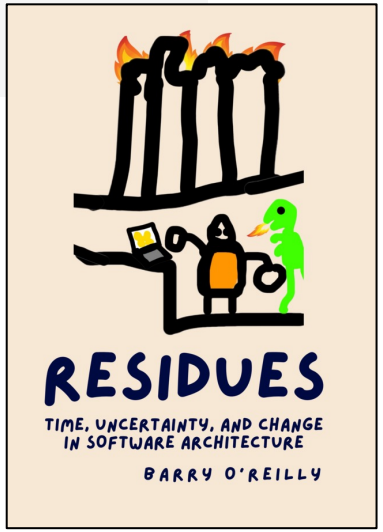
**Das habt ihr gerade gesehen.**

# Das theoretische Fundament

## Residuality Theory

Stressoren treiben ein System in **Attraktoren**: wiederkehrende Zustände, in die ein komplexes Umfeld immer wieder zurückfällt. Die Zahl möglicher Stressoren ist **um Größenordnungen höher** als die Zahl der Attraktoren. Wer einen Attraktor abdeckt, schützt gegen alle Stressoren, die dorthin führen, **auch unbekannte**.

Deshalb hat die Architektur den Reseller-Stressor überlebt, obwohl niemand ihn vorhergesehen hat: die vorherigen Änderungen deckten bereits den Attraktor ab, in den dieser Stressor das System treibt.




Für Vertiefung: Residues: Time, Change, and Uncertainty in Software Architecture, Barry O'Reily

# Die Methode

Schritt für Schritt und Herausforderungen



# Die Methode in einem Satz

**Wende zufällige Stressoren auf deine Architektur an, bis neue Stressoren sie nicht mehr brechen.**

Das klingt einfach. Ist es auch. Bis auf zwei Stellen.

# Die fünf Schritte

1. Stressoren sammeln
2. Stressor anwenden
3. Überlebt die Architektur?
4. Architektur anpassen
5. Wiederholen

# Die fünf Schritte

| | | |
|---|---|---|
| 1 | Stressoren sammeln | einfach |
| 2 | Stressor anwenden | einfach |
| **3** | **Überlebt die Architektur?** | **schwer** |
| **4** | **Architektur anpassen** | **schwer** |
| 5 | Wiederholen | einfach |

# Brainstorming

Stressoren können sein:

| | |
|---|---|
| **Business** | Neues Kundensegment, Preismodelländerung, Marktpivot |
| **Technisch** | 10x Last, Ausfall einer Komponente, Latenzspitzen |
| **Organisatorisch** | Team verlässt das Unternehmen, Reorg, Outsourcing |
| **Regulatorisch** | DSGVO, Branchenvorschriften, Audit-Anforderungen |

> Die Qualität und Eintrittswahrscheinlichkeit der Stressoren ist weniger wichtig als der Prozess.

# Warum ist das schwer?

## Es ist kein Ja oder Nein

"Überleben" hat Graustufen. Wie viel Aufwand ist akzeptabel? Wann ist eine Änderung "zu teuer"?

## Auswirkungen sind schwer zu verfolgen

Ein Stressor landet hier, aber wo breitet er sich aus? Was übersehen wir?

## Teams sind optimistisch

"Das schaffen wir schon" bis sie es nicht schaffen. Optimismus verdeckt strukturelle Schwächen.



# Die Architektur anpassen.

Die zweite schwierige Stelle.
(Dazu kommen wir gleich)

# Komplexität

Die Methode ist **einfach**. Das **Urteilsvermögen** nicht.

- **Überlebt** die Architektur? (Graustufen, Nachverfolgung, Optimismus)
- Welche Art von **Schwäche** liegt vor?
- Was ist der richtige **Architectural Move**?
- Welche **Trade-Offs** entstehen dabei?

# Herausforderungen

Praxis, Komplexität,
Menschen

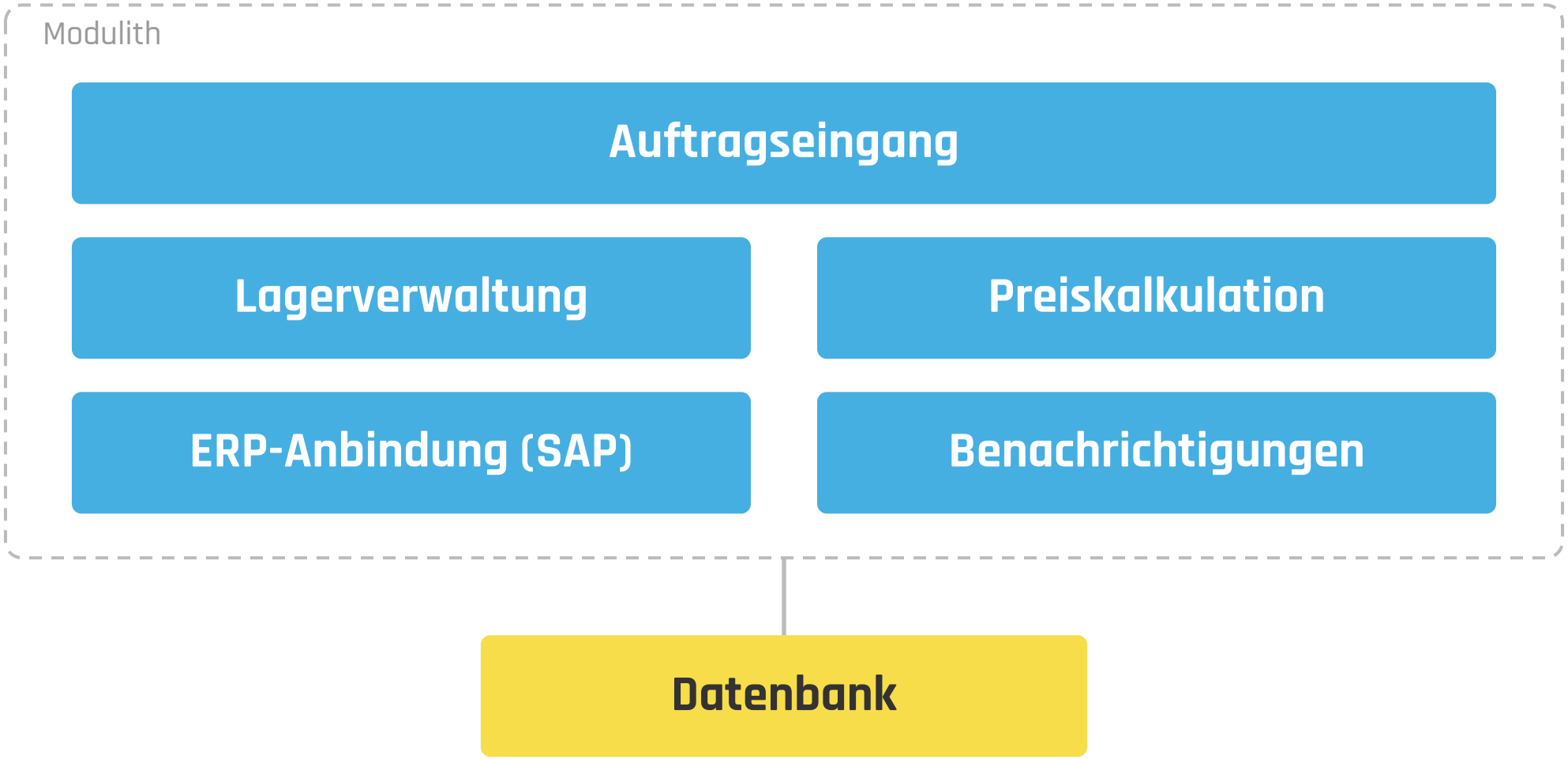
# Eine Beispiel-Session

**Auftragsverarbeitung bei einem Hersteller technischer Bauteile.**

- Auftragseingang: Bestellungen vom Vertrieb über ein Web-Portal
- Lagerverwaltung, Preiskalkulation, ERP-Anbindung (SAP), Benachrichtigungen
- Ein Modulith, eine Datenbank, ein Team (6 Entwickler)
- Die erste Architekturidee steht. Bevor das Team implementiert, stress-testen sie.

# Die Architektur

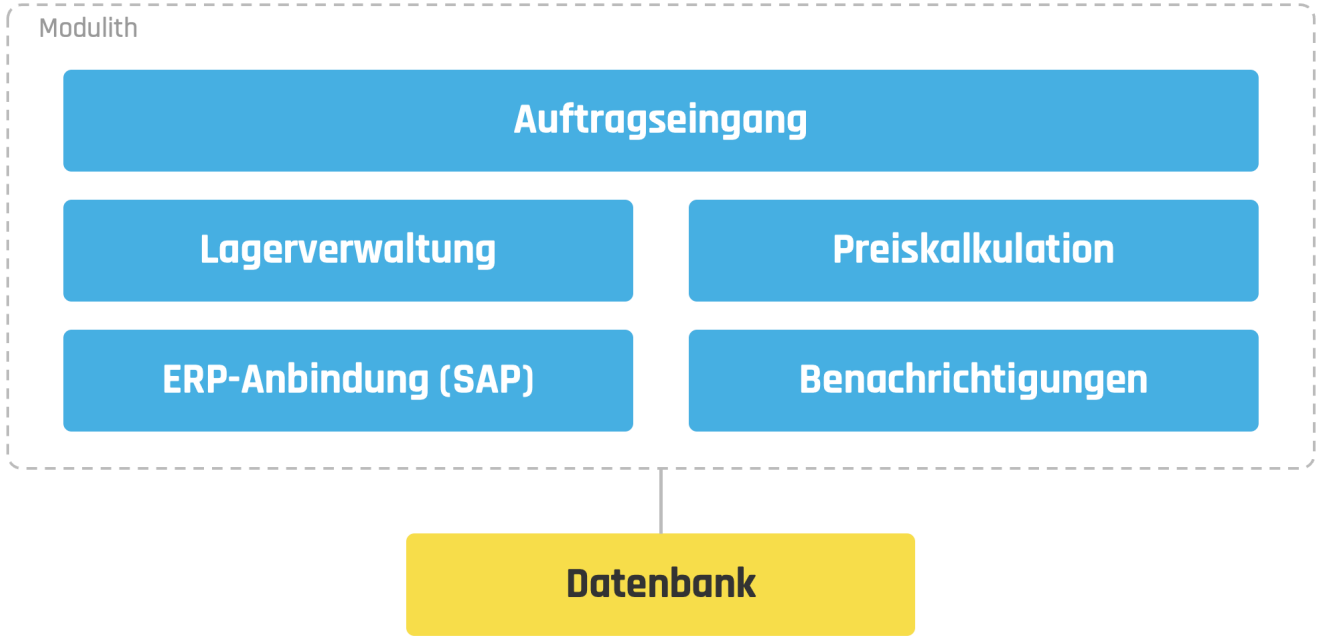
Ein Modulith, eine Datenbank, ein Team (6 Entwickler)



# Brainstorming

**Auftragsverarbeitung bei einem Hersteller technischer Bauteile.**

- Auftragseingang: Bestellungen vom Vertrieb über ein Web-Portal
- Lagerverwaltung, Preiskalkulation, ERP-Anbindung (SAP), Benachrichtigungen
- Ein Modulith, eine Datenbank, ein Team (6 Entwickler)
- Die erste Architekturidee steht. Bevor das Team implementiert, stress-testen sie.




**Welche Stressoren fallen euch ein?**

# Drei Stressoren

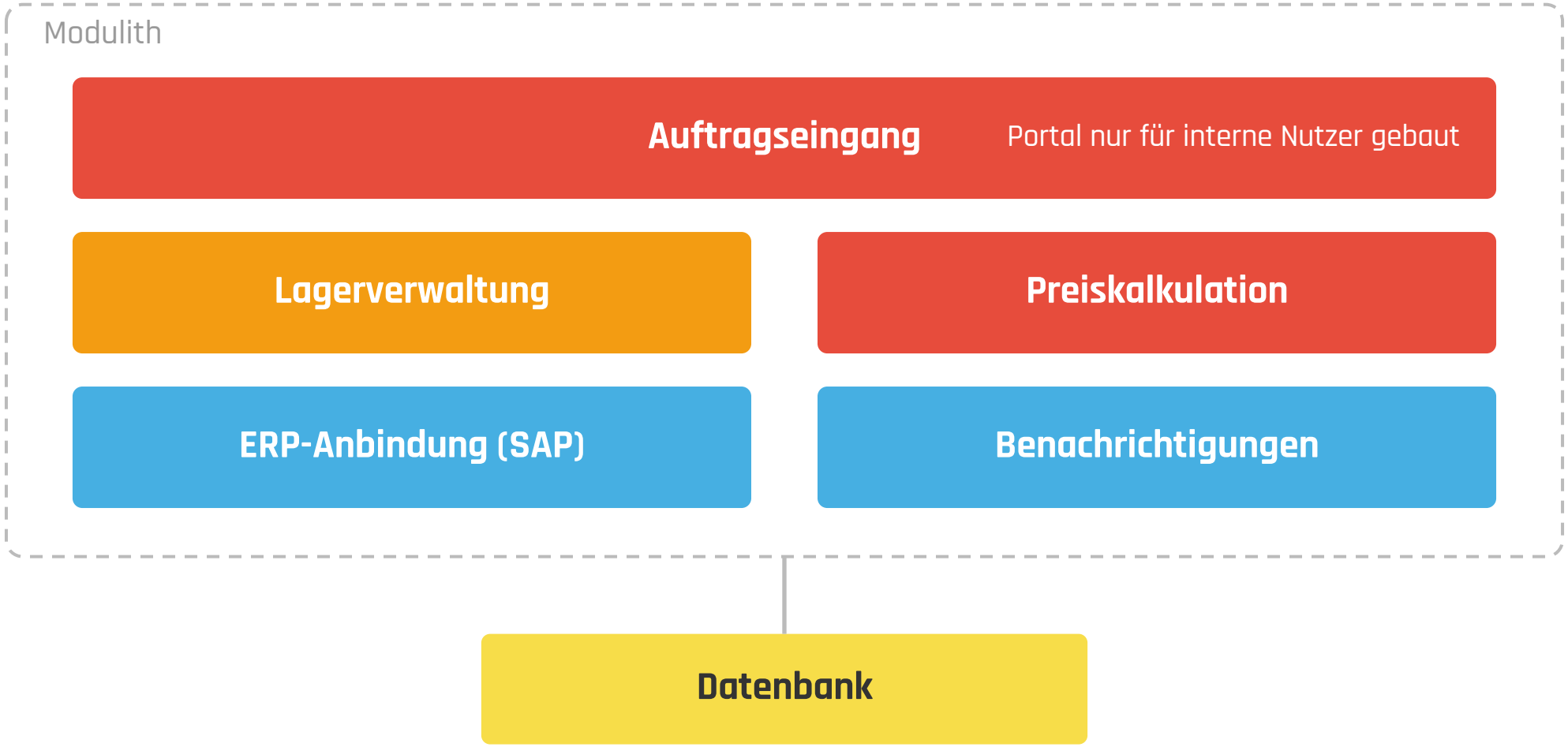
1. Der Vertrieb will einen Online-Shop für Endkunden
2. Ein Großkunde will per API bestellen
3. E-Rechnungspflicht

Zwei Business-Stressoren, ein regulatorischer. Los geht‘s…

Stressor 1

# Online-Shop für Endkunden

Der Vertrieb will B2C statt nur B2B




bricht

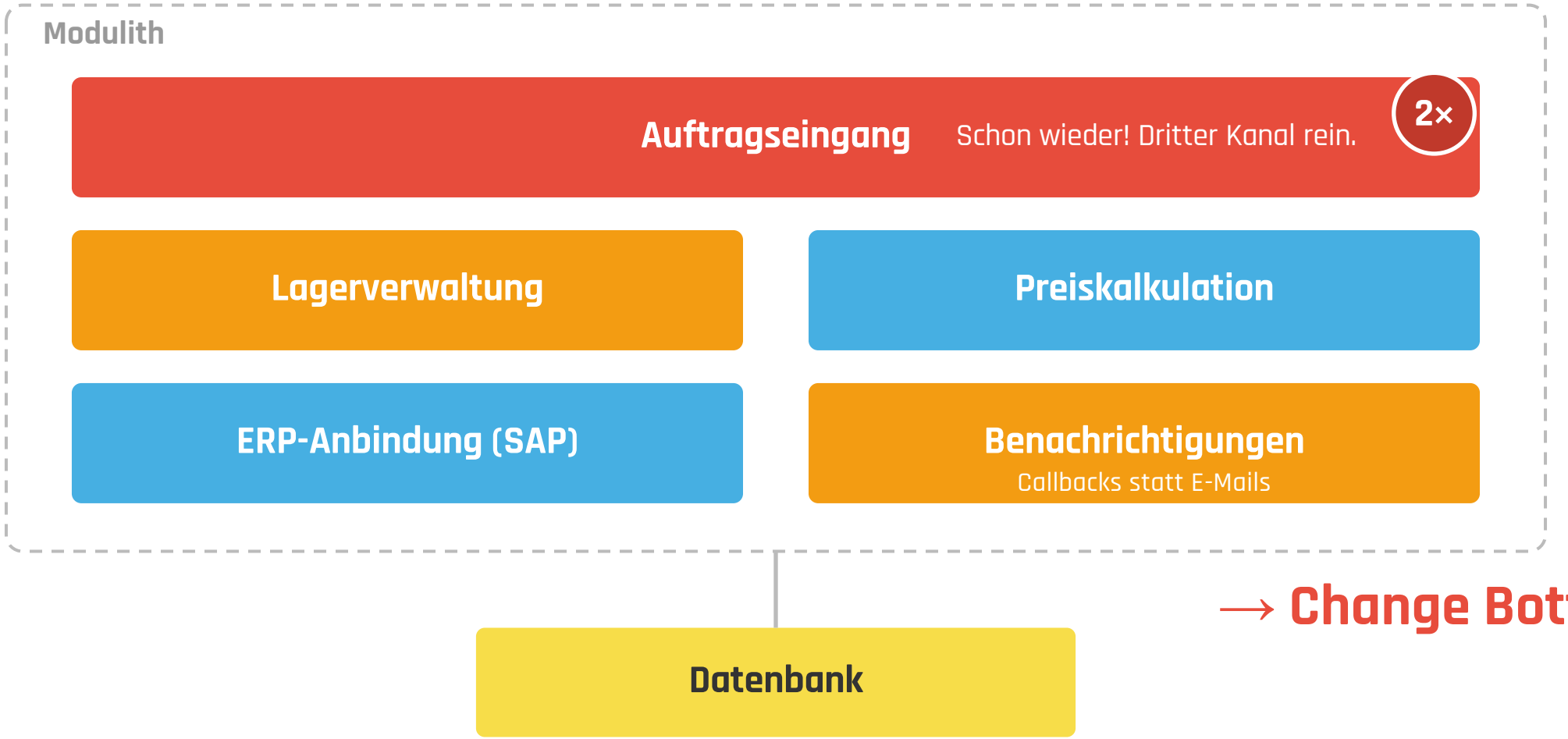
unter Druck

überlebt

Stressor 2

# Großkunde will per API bestellen

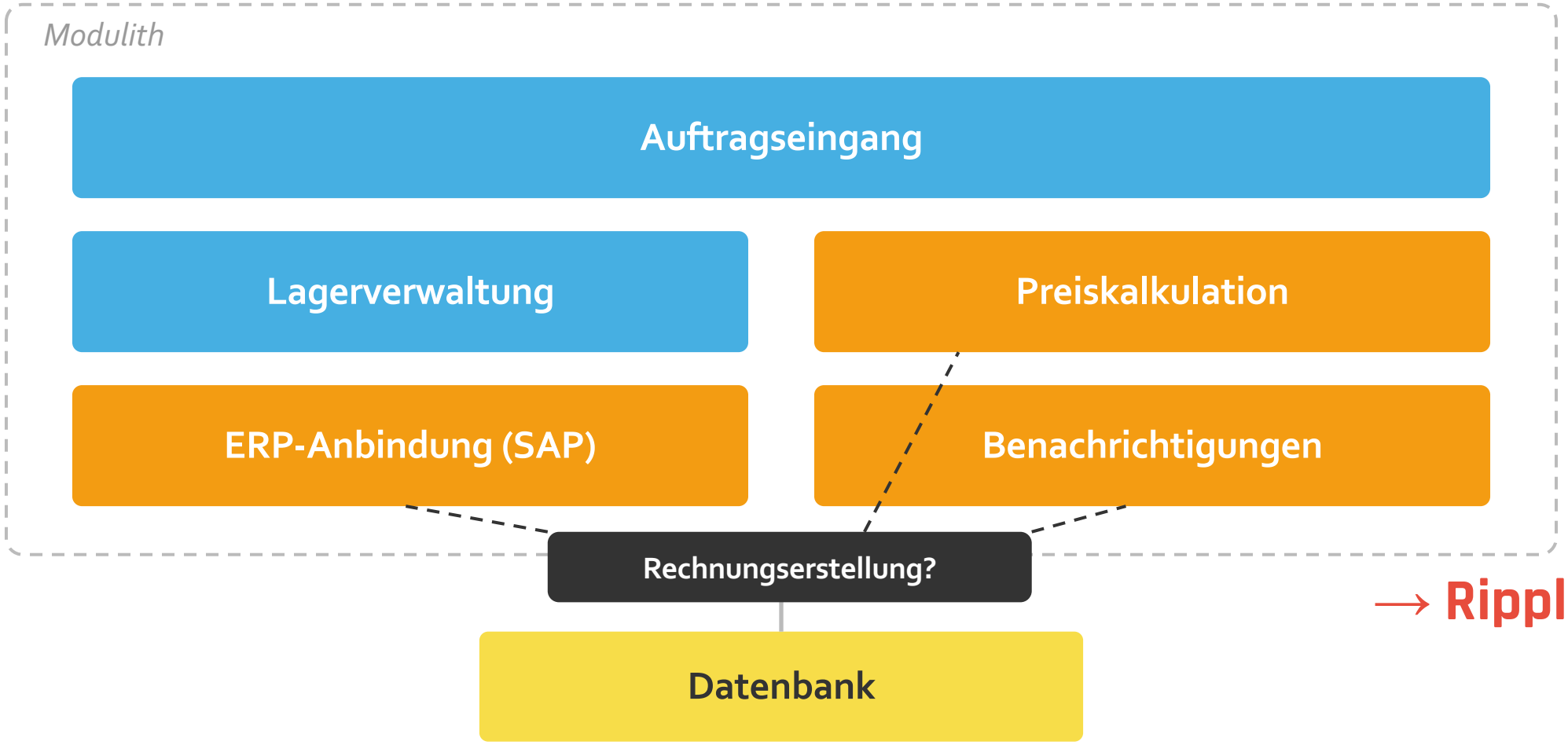
EDI-Integration, automatisierte Aufträge



Stressor 3

# E-Rechnungspflicht

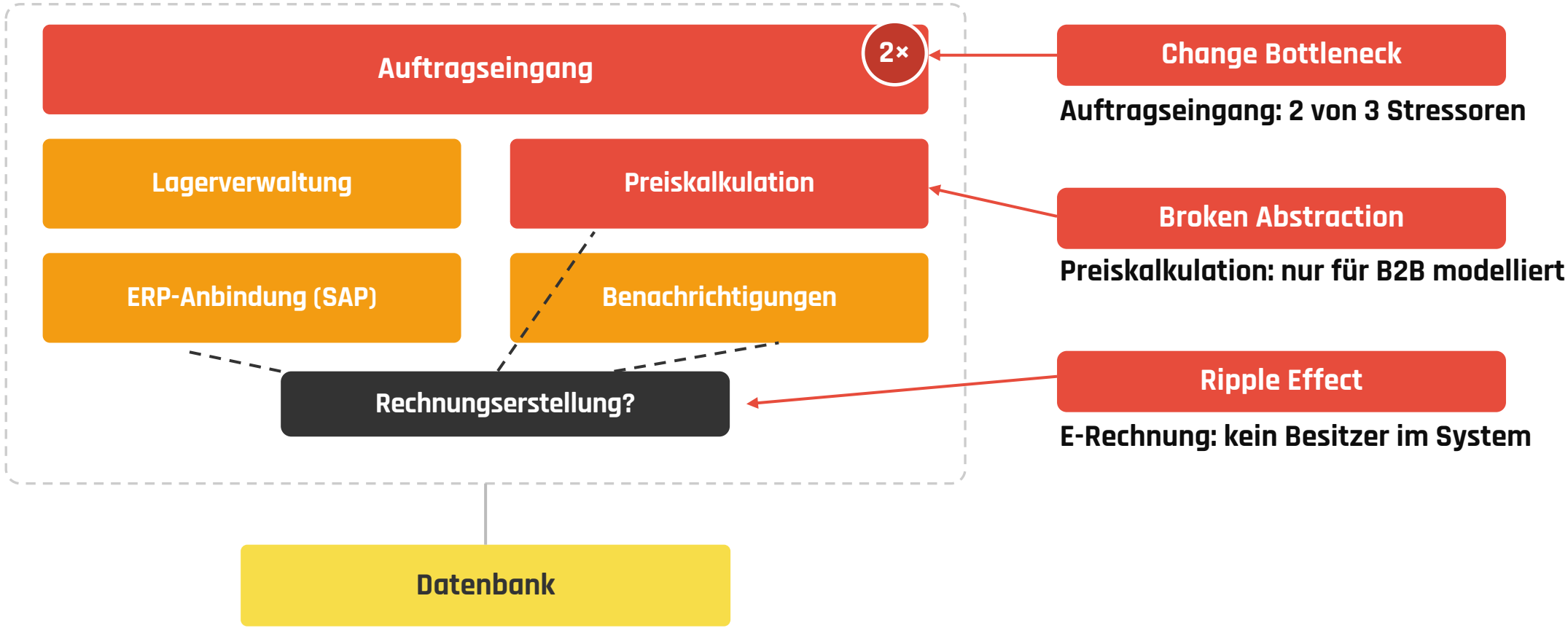
Xrechnung, ZUGFeRD, Peppol + Wer erstellt die Rechnung?




→ Ripple Effect

bricht  unter Druck  überlebt

# 3 Stressoren, 3 Schwächen



# Die Iteration

Aus jeder Schwäche leiten wir einen **Architectural Move** ab:

**Change Bottleneck** → **Auftragseingang aufteilen**
Kanal-Adapter von Auftragsverarbeitung trennen

**Broken Abstraction** → **Preismodell generalisieren**
Verschiedene Preisstrategien (B2B, B2C, Staffel) ermöglichen

**Ripple Effect** → **Rechnungserstellung zentralisieren**
Eigenes Modul mit klarer Verantwortung

# Bauen wir das alles?

Nicht alles, was Stress-Testing aufdeckt, muss sofort gebaut werden.

**Build**

**Die Architektur ist jetzt schon falsch.**
Das Modell bildet die aktuelle Realität nicht korrekt ab. Jetzt ändern.

**oder**

**Prepare**

**Die Naht sauber halten.**
Nichts Zusätzliches bauen, aber die Stelle definieren, an der später geschnitten wird.

# Angewendet auf unser Beispiel

Build

**Ripple Effect** →

**Rechnungsmodul bauen**

Regulierung kommt. Und das Konzept hat heute schon keinen Besitzer.

Build

**Broken Abstraction** →

**Preismodell generalisieren**

Die Abstraktion ist jetzt schon falsch.

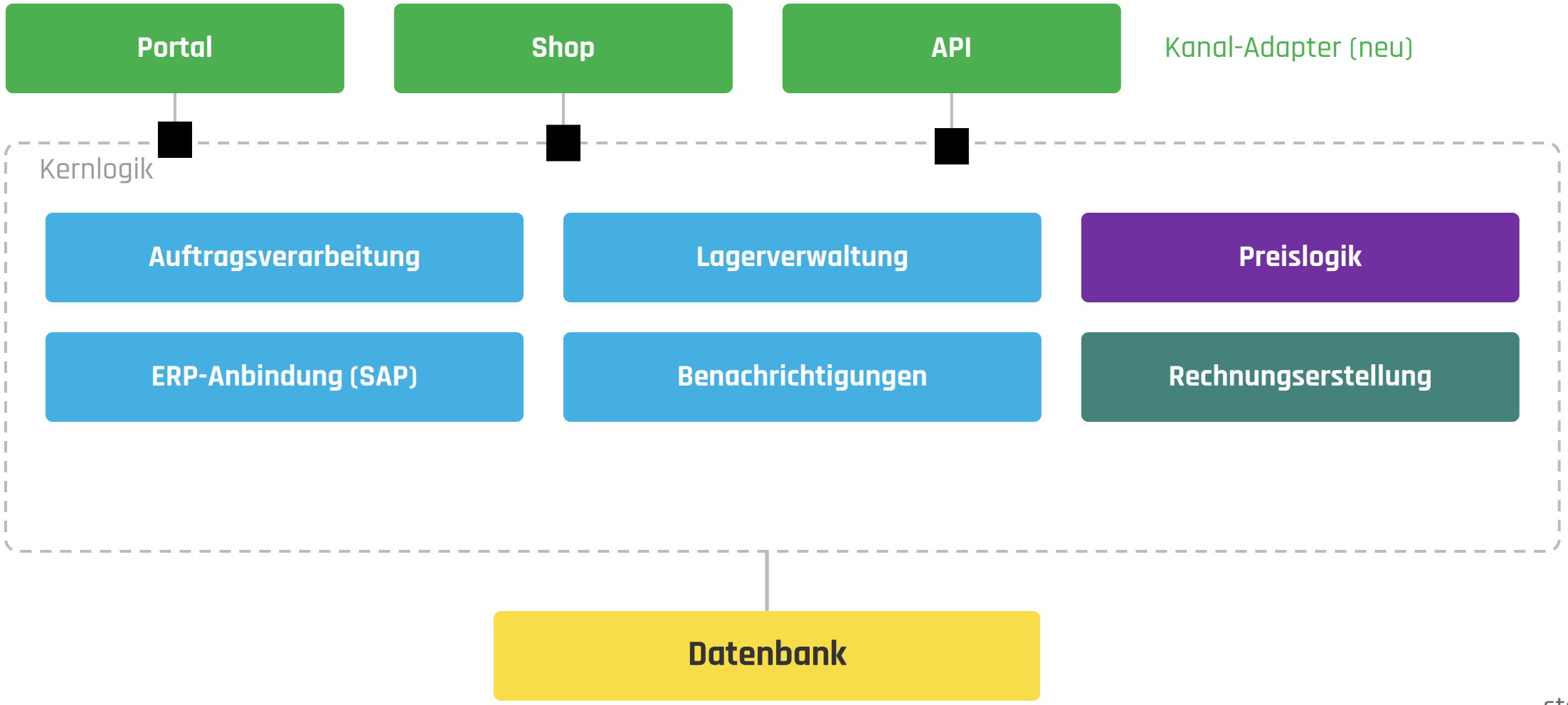
Prepare

**Change Bottleneck** →

**Saubere Naht zwischen Eingang und Verarbeitung**

Keine drei Kanäle bauen. Aber die Stelle definieren, an der später geschnitten wird.

# Die verbesserte Architektur



# Dieselben 3 Stressoren

| | Vorher | Nachher |
|---|---|---|
| **Online-Shop**<br>B2C statt nur B2B | Auftragseingang bricht, Preiskalkulation bricht | Neuer Kanal-Adapter, Preismodell trägt B2C |
| **API-Bestellung**<br>EDI-Integration | Auftragseingang bricht wieder, dritter Kanal | Neuer Kanal-Adapter, Kernlogik unberührt |
| **E-Rechnungspflicht**<br>XRechnung, ZUGFeRD | Kein Besitzer, überall Auswirkungen | Rechnungsmodul ist zuständig |

# Vier Arten von Schwächen

## Change Bottleneck

Viele Stressoren landen auf derselben Komponente

## Hidden Coupling

Ein Stressor bricht zwei Komponenten, die nicht verbunden schienen

## Broken Abstraction

Ein Konzept wurde zu spezifisch modelliert

## Ripple Effect

Ein Stressor breitet sich über viele Komponenten aus

# Hidden Coupling - Beispiel

**Stressor: "Enterprise-Kunden brauchen konfigurierbares Session-Timeout."**

Die Analyse zeigt:

### Auth-Service

Muss Session-Dauer konfigurierbar machen ✓

### Notification-Service

Bricht! Hält Websocket-Connections, die auf Session-Gültigkeit prüfen.

Die Schwäche: Beide Services teilen implizit ein Session-Konzept, ohne dass es eine explizite Abhängigkeit gibt. Der Notification-Service steht nicht in den Dependencies des Auth-Service, aber beide müssen sich ändern.

# Schwächen und Architectural Moves

| | | |
|---|---|---|
| **Change Bottleneck** | → | Verteilen (Bounded Contexts), Platform Teams |
| **Hidden Coupling** | → | Zusammenführen oder gemeinsames Konzept extrahieren |
| **Broken Abstraction** | → | Entkopplungspunkte schaffen, kollaboratives Modellieren |
| **Ripple Effect** | → | Capability zentralisieren, Value Stream Mapping |

Den richtigen Move zur richtigen Schwäche zu finden erfordert Erfahrung.

# Jeder Move hat Kosten

Es gibt keine Architekturentscheidungen ohne Trade-Offs.

| | |
|---|---|
| **Verteilen** | Mehr Koordination, Netzwerk-Komplexität, eventuelle Konsistenz |
| **Zusammenführen** | Größere Deployments, potenziell langsamere Teams |
| **Abstrahieren** | Indirektion, Lernkurve |
| **Zentralisieren** | Single Point of Failure, Bottleneck-Risiko |

> Die Kunst ist, den Trade-off zu wählen, mit dem du leben kannst.

# Workshop: 60-90 Minuten

## Wann passt es?

Erstes Ausprobieren der Methode, wenig Zeit, schnelle Einschätzung gefragt

## Ziel

Einen Teil eines Architekturentwurfs in einer Iteration verbessern.

## Ablauf

1. Einen Fokusbereich wählen (z.B. nur Multi-Tenancy)
2. 3-5 Stressoren sammeln und anwenden
3. Schwächen identifizieren und einordnen
4. Architektur anpassen (eine Iteration)

Output: Verbesserte Architektur für einen Fokusbereich, erste Build/Prepare-Entscheidungen

# Halber bis ganzer Tag

## Wann passt es?

Konkrete Architekturentscheidung steht an, Team-Alignment gewünscht

## Ziel

Angepasste Architektur mit dokumentierten Entscheidungen

## Ablauf

1. Architektur-Überblick erstellen
2. Stressoren sammeln und anwenden
3. Schwächen einordnen, Architektur iterativ anpassen
4. Build/Prepare/Skip-Entscheidungen treffen

Output: Angepasste Architektur, Build/Prepare-Entscheidungen, Decision Log



# Integration in den Alltag

Stress-Testing ist kein Event – es ist eine Denkweise.

## Wann im Alltag?

- Bei Architekturentscheidungen
- In Reviews
- Bei neuen Features und Epics
- Wenn großer Umbau ansteht (z.B. Migrationen, Modernisierungen)

# Stress-Testing in Brownfield-Systemen

### Für neue Features oder Umbauarbeiten

Du baust etwas Neues auf ein bestehendes System. Die Methode funktioniert wie bei Greenfield, aber mit mehr Constraints. Das bestehende System gibt dir Stressoren vor, die du nicht ignorieren kannst.

### Zur Priorisierung

Wo lohnt sich Refactoring am meisten? Wende Stressoren auf das bestehende System an und schau, wo es am meisten wehtut. Die Bereiche, die bei vielen Stressoren brechen, sind die Kandidaten.

# Die offenen Fragen

| Hier vereinfacht | Offene Fragen |
| --- | --- |
| 3 Stressoren | Wie viele braucht man, bis man aufhören kann? Wie lange dauert es? |
| Klare Schwächen | Wann ist eine Schwäche real – und wann Optimismus? |
| Eindeutige Moves | Welcher Move passt – und welche Kosten kauft man sich ein? |
| Unbegrenzte Zeit | Wie tief geht man unter Zeitdruck? |
| Greenfield | Was, wenn man das System nicht selbst gebaut hat? |

# Typische Facilitation-Herausforderungen

## Nur technische Stressoren

Teams denken an "10x Last" und "Service fällt aus", nicht an "Schlüsselkunde kündigt" oder "Regulierung ändert sich".

## Gruppendenken

Niemand will der Pessimist sein. "Das schaffen wir schon" verhindert ehrliche Bewertung.

## Die falschen Leute im Raum

Nur Entwickler? Dann fehlt Business-Kontext. Nur Management? Dann fehlt technische Tiefe.

## Verteidigung statt Analyse

Das Team hat das System gebaut. Kritik fühlt sich persönlich an.

# Stress-Testing.

Eine Methode, die du selbst anwenden kannst.

Stressoren sammeln, anwenden, iterieren. Das ist lernbar.

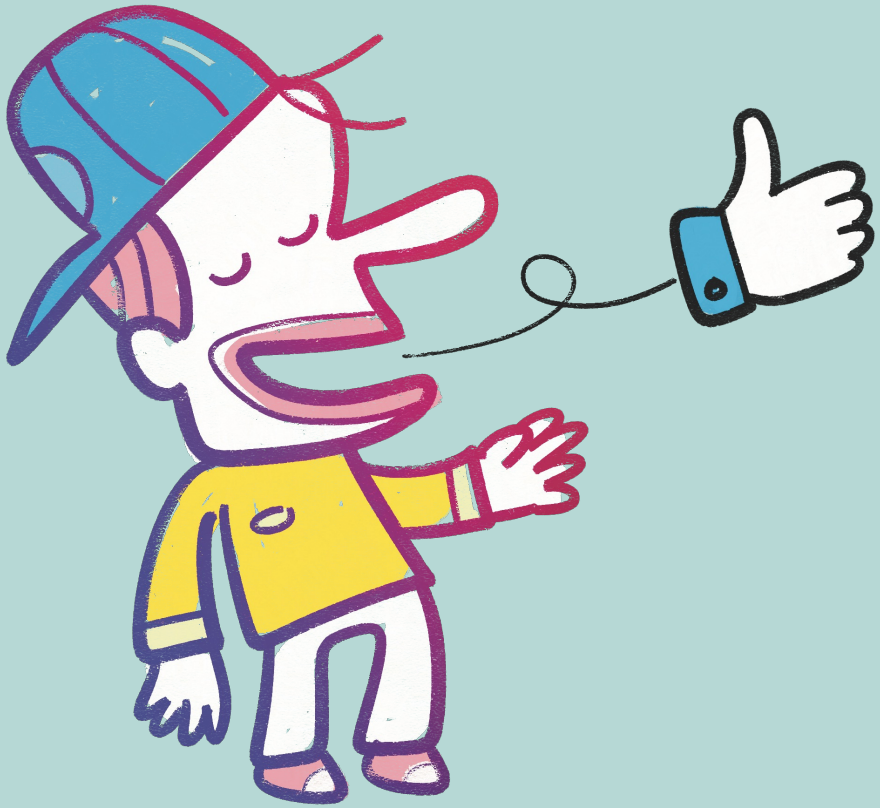
Aber:

Überlebt die Architektur? → erfordert Urteilsvermögen

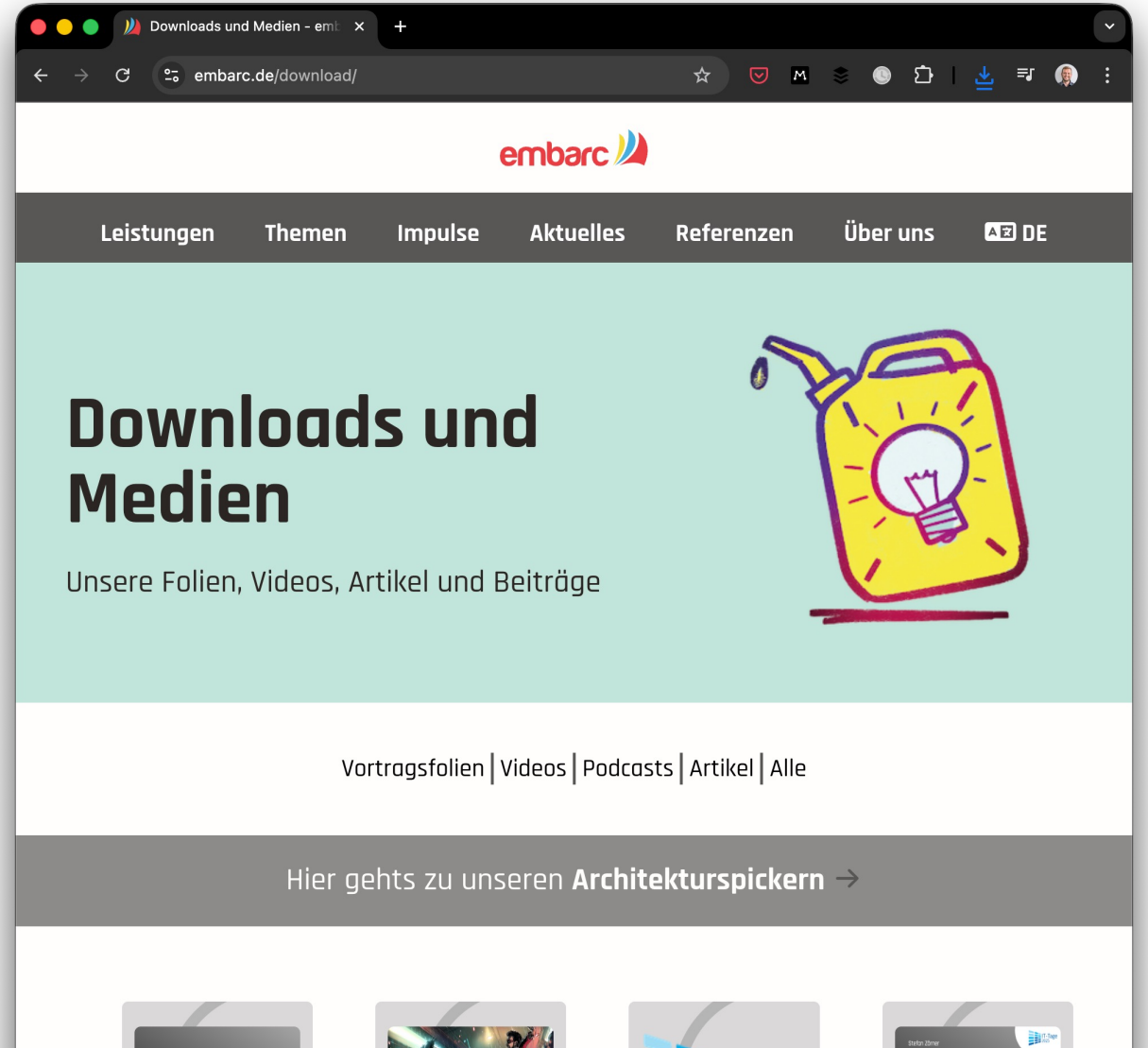
Welche Schwäche, welcher Move? → erfordert Erfahrung

Die menschliche Seite? → erfordert Facilitation-Skills

# Feedback & Fragen?



# Folien auf embarc.de




SCAN ME

# Findet mich auf LinkedIn...

SCAN ME

linkedin.com/in/alexksbr/