Kapitel 26. Architektonische Überschneidungen

Bisher haben wir in diesem Buch gezeigt, wie man die kritischen Merkmale, die eine Architektur unterstützen muss, identifiziert, wie man den am besten geeigneten Architekturstil für diese Merkmale und für das Geschäftsproblem auswählt, wie man effektive Architekturentscheidungen trifft und wie man Entwicklungsteams bei der Implementierung der Architektur leitet und anleitet. Damit eine Architektur funktioniert, muss sie jedoch auch mit anderen Aspekten des technischen und geschäftlichen Umfelds abgestimmt werden. Wir nennen diese Abstimmungen die *Schnittstellen der Architektur*.

In diesem Kapitel besprechen wir mehrere wichtige Schnittpunkte, die bei der Erstellung oder Validierung einer Softwarearchitektur auftreten:

Umsetzung
Ist die Implementierung mit den architektonischen Belangen in Bezug auf betriebliche Merkmale, architektonische Einschränkungen und die interne Struktur der Architektur abgestimmt?

Infrastruktur

Stimmen die Infrastruktur und die Art und Weise, wie die Architektur eingesetzt wird, mit den betrieblichen Belangen der Architektur überein, z. B. Skalierbarkeit, Reaktionsfähigkeit, Fehlertoleranz und Verfügbarkeit?

Daten-Topologien
Eine weithin ignorierte Abstimmung ist die zwischen der Architektur und den Datentopologien und Datentypen. Die Datentopologie (monolithisch, Domänendatenbanken und Datenbank pro Dienst) muss mit dem Architekturstil übereinstimmen, damit das System funktioniert.

Technische Praktiken
Passt die Art und Weise, wie das Entwicklungsteam Software erstellt, pflegt und testet, zur entsprechenden Architektur? Passt die Deployment-Pipeline zum Stil der Architektur?

Team-Topologien
Die Art und Weise, wie die Teams organisiert sind, kann sich erheblich auf die Architektur auswirken - und umgekehrt. Wenn die Teamstruktur nicht richtig auf die Architektur abgestimmt ist, haben die Entwicklungsteams in der Regel Schwierigkeiten und finden selbst die einfachsten Änderungen schwierig.

Systemintegration
Mit welchen anderen Systemen oder Diensten muss die Architektur kommunizieren? Wenn du dieser Schnittstelle keine Aufmerksamkeit schenkst, kann das verheerende Folgen für die Wartung, die

Zuverlässigkeit und die betrieblichen Eigenschaften wie Skalierbarkeit, Reaktionsfähigkeit und Verfügbarkeit haben.

Das Unternehmen
Ist die Architektur mit den Rahmenwerken, Praktiken, Leitprinzipien und Standards in der Organisation und im Unternehmen abgestimmt?

Das Geschäftsumfeld
Ist die Architektur richtig auf das Geschäftsumfeld und die Problemdomäne abgestimmt? Allzu oft ignorieren Architekten diesen wichtigen Schnittpunkt, und infolgedessen schlägt die Architektur fehl, um die Ziele oder Bedürfnisse des Unternehmens zu erfüllen.

Generative KI
Wie wirkt sich der zunehmende Einsatz von großen Sprachmodellen (LLMs) auf die Architektur aus? Diese Schnittstelle wird schnell zu einer wichtigen, da immer mehr Unternehmen generative KI in ihren Systemen einsetzen.

In den folgenden Abschnitten werden diese Kreuzungen genauer beschrieben.

Architektur und Implementierung

Das erste Gesetz der Softwarearchitektur ist auch die häufigste Antwort von Softwarearchitekten auf eine Frage: "Es kommt darauf an." Die zweithäufigste Antwort ist vielleicht: "Das ist ein

Implementierungsdetail". Wenn eine Softwarearchitektur ihre Ziele nicht erreicht, ist oft diese zweite Antwort schuld.

Damit eine Architektur richtig funktioniert, muss ihre *Implementierung* - also ihr Quellcode - auf drei Dinge abgestimmt sein: die betrieblichen Belange der Architektur (wie Fehlertoleranz, Reaktionsfähigkeit, Skalierbarkeit usw.), die interne Struktur und die Einschränkungen. Dieser Abschnitt befasst sich mit jedem der drei Punkte.

Betriebliche Belange

Betriebliche Belange sind die architektonischen Merkmale, auf die wir uns in Teil I des Buches konzentriert haben und die die Grundlage jeder Softwarearchitektur bilden und die die Architektur unterstützen muss, um das jeweilige Geschäftsproblem zu lösen. Die architektonischen Merkmale sind die Grundlage für Architekturentscheidungen (wie in Kapitel 21 beschrieben).

Was bedeutet es also, wenn die Architektur und die Implementierung in Bezug auf die betrieblichen Belange des Systems nicht übereinstimmen? Angenommen, du bist Architekt und arbeitest an einem neuen Auftragseingabesystem, das mehrere tausend bis zu einer halben Million Kunden gleichzeitig bedienen soll. Du entscheidest dich für eine Microservices-Architektur auf der Grundlage der in Kapitel 18 unter "Stilmerkmale" aufgeführten Sternebewertungen, die für die hohen Anforderungen an Skalierbarkeit und Elastizität des Systems geeignet ist. Während der *Implementierung* stellt das Entwicklungsteam jedoch fest, dass der Dienst `Order Placement` nicht direkt auf die

Bestandsdatenbank zugreifen kann, weil die Kontexte der Dienste so eng begrenzt sind (siehe "Begrenzter Kontext" in Kapitel 18). Stattdessen muss er synchron den Dienst `Inventory` aufrufen, um den aktuellen Bestand für jeden Artikel abzufragen, den ein Kunde kaufen möchte. Dieser synchrone Aufruf sorgt nicht nur für eine enge Kopplung der beiden Dienste, sondern verlangsamt auch die Reaktionsfähigkeit des Systems erheblich.

Das Entwicklungsteam beschließt, einen *replizierten In-Memory-Cache* zwischen diesen Diensten zu verwenden, wie in Abbildung 26-1 dargestellt: Die Daten befinden sich im *internen Speicher* jeder Dienstinstanz und werden im Hintergrund durch Caching immer synchron gehalten. Zu den Caching-Produkten für diesen Zweck gehören Apache Ignite und Hazelcast. In diesem Fall würde der Dienst `Inventory` über einen beschreibbaren In-Memory-Cache verfügen, der die Artikel-IDs und die aktuellen Bestandszahlen enthält; jede Instanz des Dienstes `Order Placement` würde über eine replizierte Nur-Lese-Version dieses Caches in ihrem internen Speicher verfügen. Die Verwendung eines replizierten In-Memory-Caches entkoppelt die Dienste und verbessert die Reaktionsfähigkeit *erheblich*.

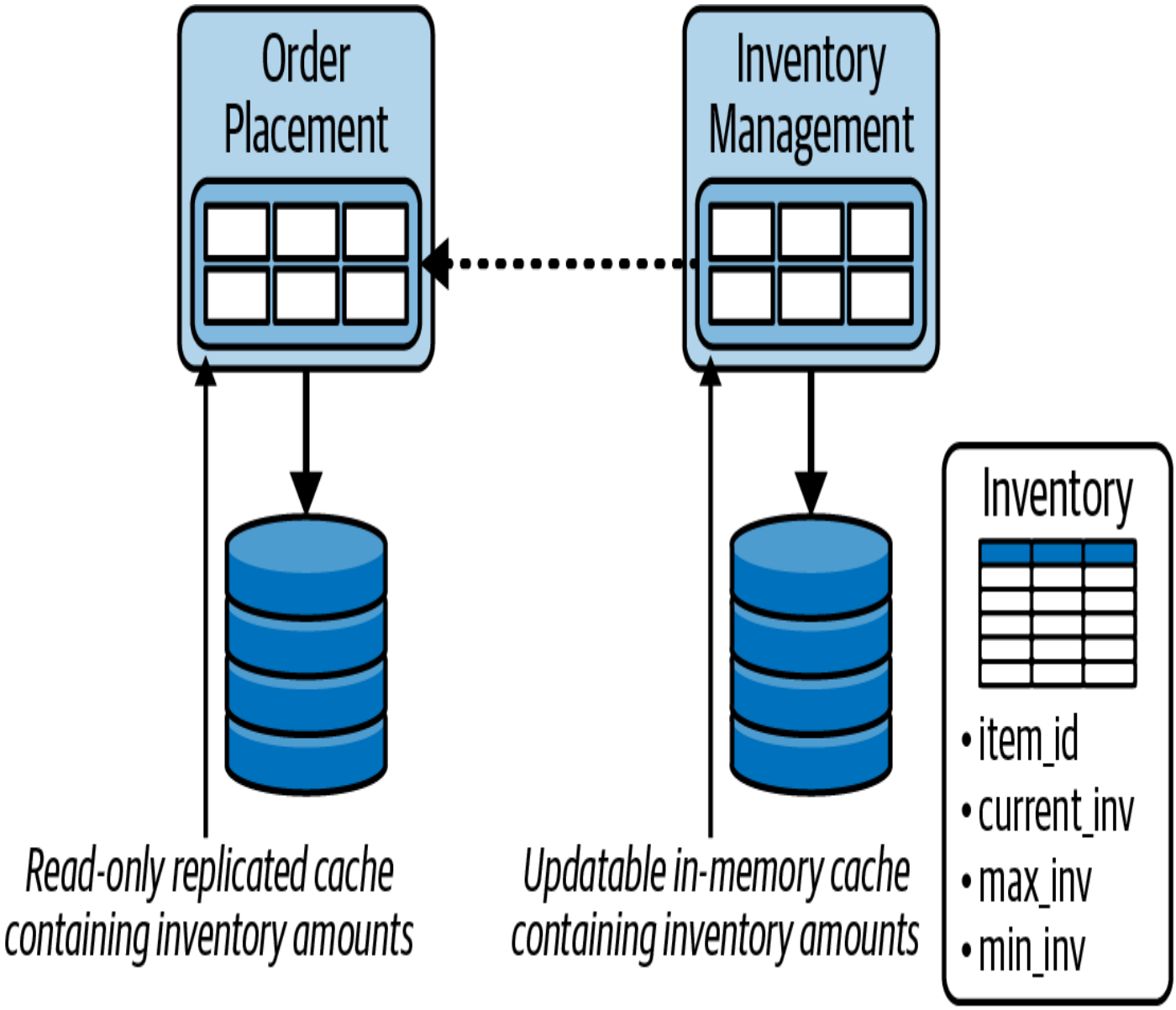


Abbildung 26-1. Das Entwicklungsteam entschied sich für die Verwendung von In-Memory Replicated Caching zwischen den Diensten, was bei der Skalierung der Dienste zu Out-of-Memory-Bedingungen führte

Nach der Produktionsfreigabe, wenn die Anzahl der gleichzeitigen Nutzer steigt, werden mehr Instanzen jedes Dienstes benötigt, um die Last zu bewältigen. Wenn die Last 80.000 gleichzeitige Kunden erreicht, stürzt das System ab, weil der Speicherbedarf des internen Caches zu hoch ist, was dazu führt, dass in allen virtuellen Maschinen der Speicher nicht mehr ausreicht.

In diesem Szenario sind die Architektur und ihre Umsetzung nicht aufeinander abgestimmt. Während sich die Architektur auf die Unterstützung eines hohen Maßes an *Skalierbarkeit* und *Elastizität* konzentriert, liegt der Schwerpunkt bei der Umsetzung auf *Reaktionsfähigkeit* und *Entkopplung der Dienste*. Beide Teams haben gute Entscheidungen getroffen, aber sie verfolgen unterschiedliche Ziele.

Strukturelle Integrität

Du hast in Kapitel 8 gelernt, dass logische Komponenten die Bausteine jedes Systems sind und seine *logische Architektur* bilden. Sie werden normalerweise durch die Verzeichnisstrukturen im Quellcode-Repository (oder Namensräume, je nach Programmiersprache) dargestellt. Da die logische Architektur beschreibt, wie das System funktioniert und welche Teile des Systems mit anderen Teilen interagieren, ist es wichtig, dass die Struktur des Quellcodes mit der logischen Architektur übereinstimmt.

Ohne angemessene Anleitung, Wissen und Kontrolle ist es für Entwickler leicht, die logische Architektur des Systems zu ignorieren und nach Belieben Verzeichnisstrukturen und Namensräume zu erstellen, ohne sich um deren Auswirkungen auf die Integrität des Systems zu kümmern. Diese Fehlanpassung führt zu Architekturen, die schwer zu warten, zu testen und zu implementieren sind. Infolgedessen werden sie weniger zuverlässig und lassen sich nur schwer weiterentwickeln oder an neue Funktionen anpassen, wie die in Abbildung 26-2 dargestellte logische Architektur.

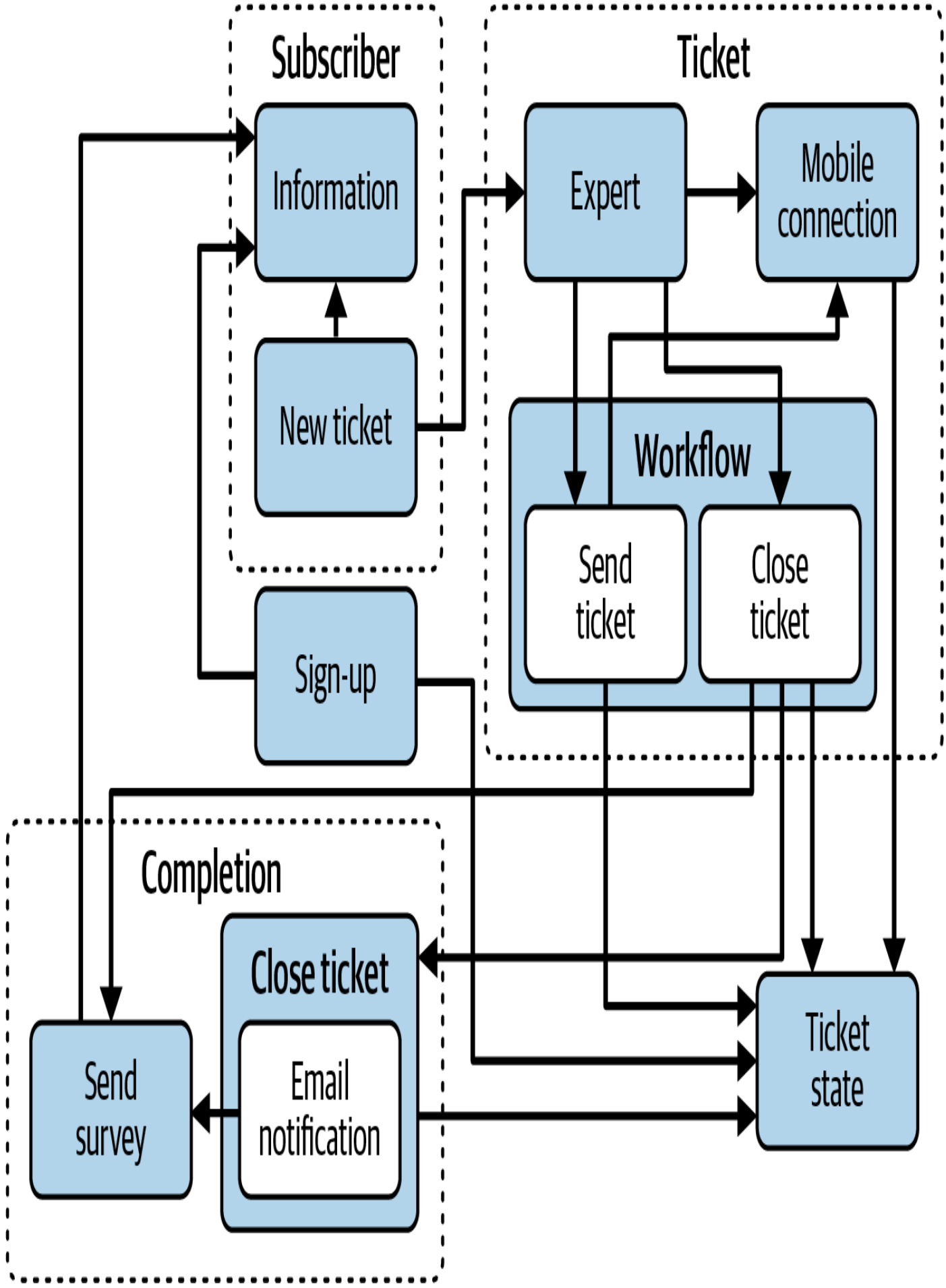

Subscriber
Information
New ticket
Sign-up
Ticket
Expert
Mobile connection
Workflow
Send ticket
Close ticket
Completion
Close ticket
Email notification
Send survey
Ticket state

Abbildung 26-2. Ein Beispiel für eine interne logische Architektur, der es an Governance und Abstimmung fehlt

Um sicherzustellen, dass die Struktur des Quellcodes mit der logischen Architektur übereinstimmt, empfehlen wir den Einsatz automatisierter Governance-Tools, wie ArchUnit für die Java-Plattform, ArchUnitNet und NetArchTest für die .NET-Plattform, PyTestArch für Python oder TSArch für TypeScript und JavaScript. In Verbindung mit einer guten Kommunikation und Zusammenarbeit zwischen dem Architekten und dem Entwicklungsteam sorgen diese automatisierten Werkzeuge für Implementierungen, die genau auf die Architektur abgestimmt sind, wie in Abbildung 26-3 dargestellt.

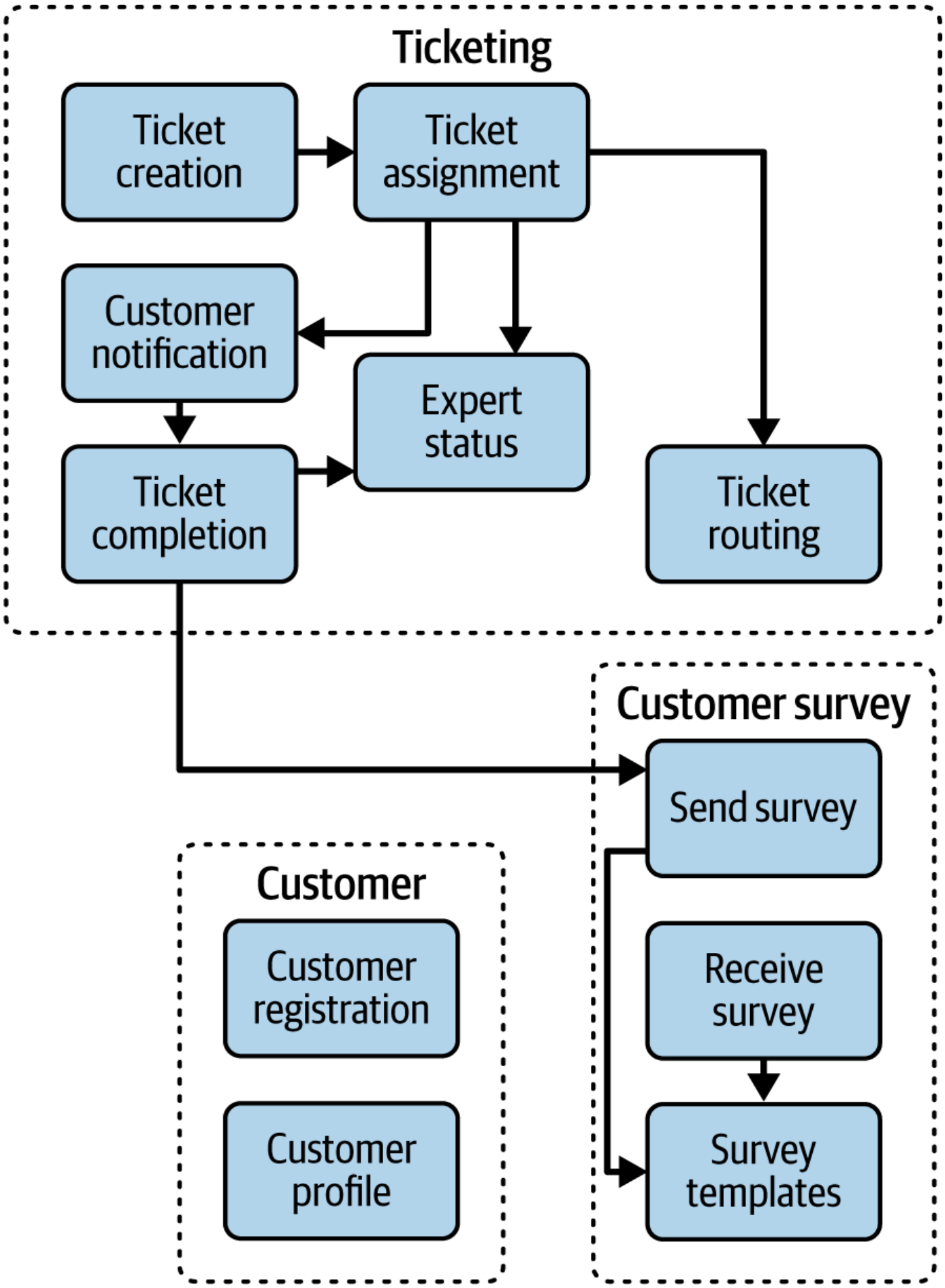

Ticketing
Ticket creation
Ticket assignment
Customer notification
Expert status
Ticket completion
Ticket routing
Customer survey
Send survey
Customer
Customer registration
Receive survey
Customer profile
Survey templates

Abbildung 26-3. Ein Beispiel für eine interne logische Architektur, die auf einer angemessenen Governance und Ausrichtung basiert

Vergleiche Abbildung 26-2 mit Abbildung 26-3. Beachte, dass die Architektur in Abbildung 26-3 viel wartbarer, testbarer, einsatzfähiger, zuverlässiger, anpassungsfähiger und erweiterbarer ist als die falsch ausgerichtete Architektur in Abbildung 26-2. Dieser Vergleich verdeutlicht, wie wichtig diese Art der Ausrichtung der Implementierung ist.

Architektonische Zwänge

Ein *Constraint* ist eine Regel oder ein Prinzip, das eine Art von Einschränkung innerhalb der Architektur beschreibt (z. B. die Beschränkung der Kommunikation auf REST oder die Verwendung eines bestimmten Datenbanktyps), die zum Erreichen der Ziele erforderlich ist. Wenn sich die Implementierung des Systems nicht an die Beschränkungen hält, schlägt die Architektur fehl. Ein Teil der Aufgabe eines Softwarearchitekten ist es daher, die *Einschränkungen* einer Architektur zu identifizieren und zu kommunizieren.

Um zu verdeutlichen, was wir mit dieser speziellen Überschneidung meinen, stellen wir uns ein Unternehmen vor, das ein neues System mit einem sehr begrenzten Budget und einem engen Zeitplan einführen will. Dieses Unternehmen erwartet viele strukturelle Änderungen an der Datenbank und muss diese Änderungen so schnell wie möglich durchführen. In diesem Fall wäre eine traditionelle Schichtenarchitektur (siehe Kapitel 10) aufgrund ihrer Einfachheit, Kosteneffizienz und

technischen Partitionierung eine hervorragende Lösung. Da bei diesem Stil die Schichten getrennt sind, können Datenbankänderungen nur auf einer Schicht vorgenommen werden, was sie einfacher und schneller macht.

Damit die Schichtenarchitektur bei diesem speziellen Geschäftsproblem funktioniert, muss der Architekt die folgenden Einschränkungen festlegen:

- Die gesamte Datenbanklogik muss sich in der Persistenzschicht befinden.
- Die Präsentationsschicht kann nicht direkt auf die Persistenzschicht zugreifen, sondern muss alle Schichten durchlaufen, selbst bei einfachen Abfragen.

Diese Einschränkungen sind notwendig, um zu verhindern, dass sich die Datenbanklogik über die gesamte Architektur ausbreitet, und damit sich Änderungen an der physischen Datenbankstruktur (wie das Löschen einer Tabelle oder das Ändern eines Spaltennamens) nicht auf Code außerhalb der Persistenzschicht auswirken.

Nehmen wir an, die UI-Entwickler entscheiden, dass es schneller ist, die Datenbank direkt aufzurufen und implementieren die Architektur stattdessen auf diese Weise. Außerdem stellen die Backend-Entwickler fest, dass es viel einfacher ist, den Code zu warten und zu testen, wenn die Geschäftslogik und die Datenbanklogik zusammen sind, also ignorieren sie die Einschränkungen und koppeln diese Belange in der Geschäftsschicht der Architektur. Diese Implementierung ist nicht auf die

Einschränkungen der Architektur abgestimmt, was bedeutet, dass sich Datenbankänderungen auf den gesamten Code in jeder Schicht auswirken, zu lange dauern und das System die Geschäftsziele nicht erfüllt.

Architektonische Werkzeuge sind auch nützlich, um architektonische Beschränkungen zu regeln.

Architektur und Infrastruktur

Der Aufgabenbereich der Softwarearchitektur hat sich in den letzten Jahrzehnten erweitert und umfasst immer mehr Verantwortung und Perspektiven. Mitte der 2000er Jahre war die typische Beziehung zwischen Architektur und Betrieb vertraglich und formal geregelt, mit viel Bürokratie. Die meisten Unternehmen lagerten den Betrieb an Dritte aus, um die Komplexität des eigenen Betriebs mit SLAs für Betriebszeit, Skalierung, Reaktionsfähigkeit und andere wichtige Merkmale zu vermeiden. Heute jedoch nutzen Architekturen wie Microservices Merkmale, die früher ausschließlich betrieblich waren. So war beispielsweise die elastische Skalierung früher in raumbezogene Architekturen eingebaut (siehe Kapitel 16), doch heute wird sie bei Microservices durch eine engere Zusammenarbeit zwischen Architekten und DevOps weniger schmerzhaft gehandhabt.

GESCHICHTE: WIE PETS.COM UNS EINE ELASTISCHE WAAGE SCHENKTE

Die Menschen gehen oft davon aus, dass unsere heutigen technischen Möglichkeiten (wie elastische Skalierung) eines Tages von einem cleveren Entwickler erfunden wurden. In Wirklichkeit werden die besten Ideen aber oft aus harten Lektionen geboren. Pets.com ist ein frühes Beispiel. Diese E-Commerce-Website entstand 1998 in der Hoffnung, das Amazon.com für Haustierbedarf zu werden. Die brillante Marketingabteilung kreierte ein überzeugendes Maskottchen: eine Sockenpuppe mit einem Mikrofon, die respektlose Dinge sagte. Das Maskottchen wurde zum Superstar und trat bei Paraden und nationalen Sportereignissen in der Öffentlichkeit auf.

Leider hat die Geschäftsführung von Pets.com offenbar das ganze Geld für das Maskottchen und nicht für die Infrastruktur ausgegeben. Als die Bestellungen eintrudelten, waren sie nicht darauf vorbereitet. Die Website war langsam, Transaktionen gingen verloren, Lieferungen verzögerten sich ... es war so ziemlich das Worst-Case-Szenario. Kurz nach einem katastrophalen Weihnachtsgeschäft schloss Pets.com und verkaufte sein einziges wertvolles Gut, das Maskottchen.

Was Pets.com brauchte, war *elastische Skalierbarkeit*: die Möglichkeit, mehr Instanzen von Ressourcen hochzufahren, wenn sie gebraucht wurden. Cloud-Provider bieten diese Funktion heute als Standard an, aber die ersten E-Commerce-Unternehmen mussten ihre eigene Infrastruktur verwalten, und viele wurden Opfer eines bis dahin unbekannten Phänomens: Zu viel Erfolg kann ein Unternehmen umbringen. Der Fall von Pets.com und andere ähnliche

Horrorgeschichten veranlassten Architekten dazu, bei der Entwicklung von Softwarearchitekturen stärker auf solche Überschneidungen zu achten.

Die Überschneidung von Architektur und Infrastruktur ist wichtig, weil sie die operativen Architektureigenschaften ermöglicht. Nur weil eine *Architektur* eine hohe Skalierbarkeit unterstützen kann, heißt das noch lange nicht, dass sie das auch tut - wenn die dazugehörige *Infrastruktur* das nicht unterstützt, wird sie es auch nicht tun (wie das Beispiel von Pets.com zeigt). Bei unseren Kunden haben wir nur allzu oft erlebt, dass Architekten und Entwickler für Fehler in der Architektur verantwortlich gemacht wurden, die in Wirklichkeit auf eine falsche Abstimmung zwischen Architektur und Infrastruktur zurückzuführen waren.

In den meisten Fällen liegt diese Fehlanpassung an der mangelnden Kommunikation und Zusammenarbeit zwischen dem Architekten und den Verantwortlichen für Infrastruktur und Betrieb. Den Architekten fehlt oft das Bewusstsein für den Einfluss der Infrastruktur auf Eigenschaften wie Skalierbarkeit, Reaktionsfähigkeit, Fehlertoleranz, Leistung, Verfügbarkeit, Elastizität usw. Aus dieser Diskrepanz ist der Bereich DevOps entstanden.

Viele Jahre lang betrachteten viele Unternehmen den Betrieb als von der Softwareentwicklung getrennt und lagerten ihn oft aus Kostengründen an ein Drittunternehmen aus. In den 1990er und 2000er Jahren wurden viele Architekturen in der Annahme entworfen, dass der Betrieb

ausgelagert wird und somit außerhalb der Kontrolle der Architekten liegt. (Ein gutes Beispiel dafür findest du in Kapitel 16.) Mitte der 2000er Jahre begannen die Unternehmen jedoch, mit neuen Formen der Architektur zu experimentieren, die viele betriebliche Belange vereinen. Ältere Architekturen, wie z. B. die orchestrierte SOA, erforderten aufwendige Tools und Frameworks, um Funktionen wie Skalierbarkeit und Elastizität zu unterstützen, was die Implementierung erheblich erschwerte. Also bauten Architekten Architekturen, die Skalierung, Leistung, Elastizität und eine Vielzahl anderer Funktionen *intern* handhaben konnten. Der Nebeneffekt war, dass diese Architekturen sehr viel komplexer wurden.

Die Schöpfer der Microservices-Architektur haben erkannt, dass betriebliche Belange besser vom Betrieb erledigt werden können. Durch die Zusammenarbeit zwischen Architektur und Betrieb erkannten die Architekten, dass sie ihre Entwürfe vereinfachen und sich darauf verlassen konnten, dass die Betriebsmitarbeiter sich um die Dinge kümmern, die sie am besten beherrschen. Sie taten sich mit dem Betrieb zusammen, um Microservices zu entwickeln und den Grundstein für die spätere DevOps-Bewegung zu legen. DevOps hat zwar geholfen, aber die Überschneidung von Architektur und Infrastruktur ist für die meisten Unternehmen immer noch problematisch.

Auch wenn die Infrastruktur weniger problematisch ist, können Cloud-Umgebungen dennoch falsch auf eine Architektur abgestimmt werden. So kann beispielsweise die Bereitstellung von Diensten über Regionen oder sogar Verfügbarkeitszonen hinweg die Leistungs- und

Datenintegritätsvorteile von replizierten In-Memory-Caches und verteilten Caches verringern oder sogar zunichte machen. Ebenso kann die gemeinsame Nutzung von Diensten, Containern oder sogar Kubernetes Pods auf derselben virtuellen Maschine zwar die Leistung deutlich erhöhen, aber auch die Skalierbarkeit, Fehlertoleranz, Verfügbarkeit und Elastizität beeinträchtigen.

Die Abstimmung von Architektur und Infrastruktur erfordert eine enge Kommunikation und Zusammenarbeit zwischen Architekten und Mitgliedern des Infrastrukturteams oder sogar die Einführung von DevOps-Praktiken, damit alle Beteiligten die entscheidenden betrieblichen Belange verstehen. Nur dann können Architekten die betrieblichen Vorteile der von ihnen gewählten Architektur wirklich erkennen - die Vorteile, die uns zu diesen wunderbaren Fünf-Sterne-Bewertungen geführt haben.

Architektur und Datentopologien

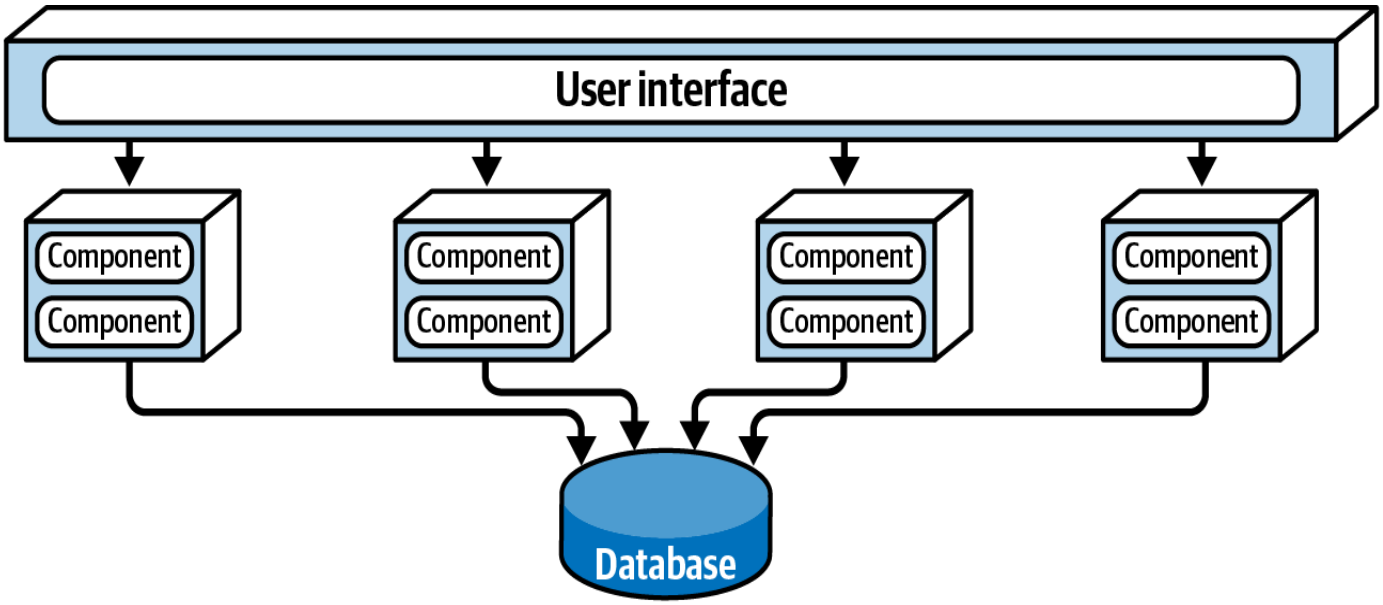
Die Überschneidung von Architektur und Datentopologien wird oft übersehen. Die Wahl des falschen Datenbanktyps oder der falschen Topologie kann einer Architektur schaden und ihre besten architektonischen Eigenschaften zunichte machen. Monolithische Datenbanken zum Beispiel bieten zwar eine gute Datenkonsistenz und Transaktionsunterstützung, können aber die Skalierbarkeit und Fehlertoleranz beeinträchtigen. Ebenso können verteilte Datenbanktopologien, die zwar gut für Skalierbarkeit und

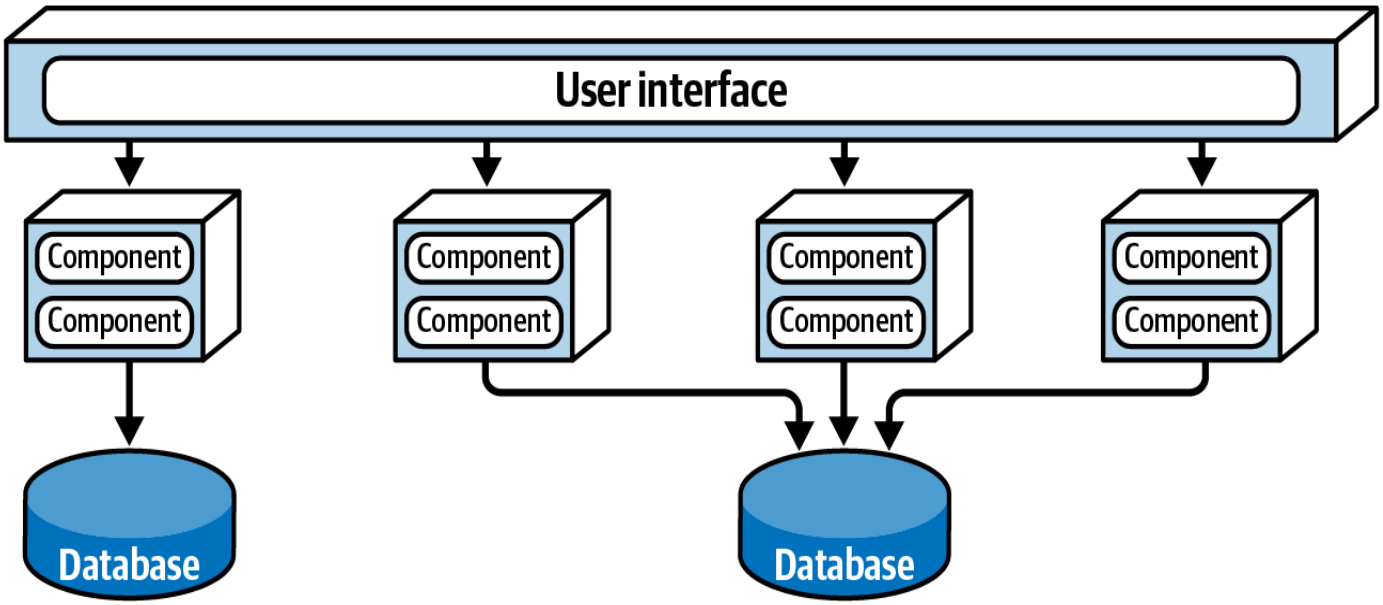
Änderungskontrolle sind, die Datenintegrität, Datenkonsistenz und Leistung eines Systems beeinträchtigen.

In den folgenden Abschnitten wird die Überschneidung von Architektur und Datentopologien beschrieben.

Datenbank-Topologie

Die *Topologie* einer Datenbank, die wir in Kapitel 15 besprochen haben, bezieht sich darauf, wie die physischen Datenbanken innerhalb der Architektur konfiguriert sind. Es gibt drei grundlegende Topologien: eine monolithische Datenbank, verteilte domänenbasierte Datenbanken oder die verteilte Datenbank-per-Service-Topologie. Abbildung 26-4 veranschaulicht diese grundlegenden Datenbanktopologien.


User interface
Component
Component
Component
Component
Component
Component
Component
Component
Database


User interface
Component
Component
Component
Component
Component
Component
Component
Component
Database
Database

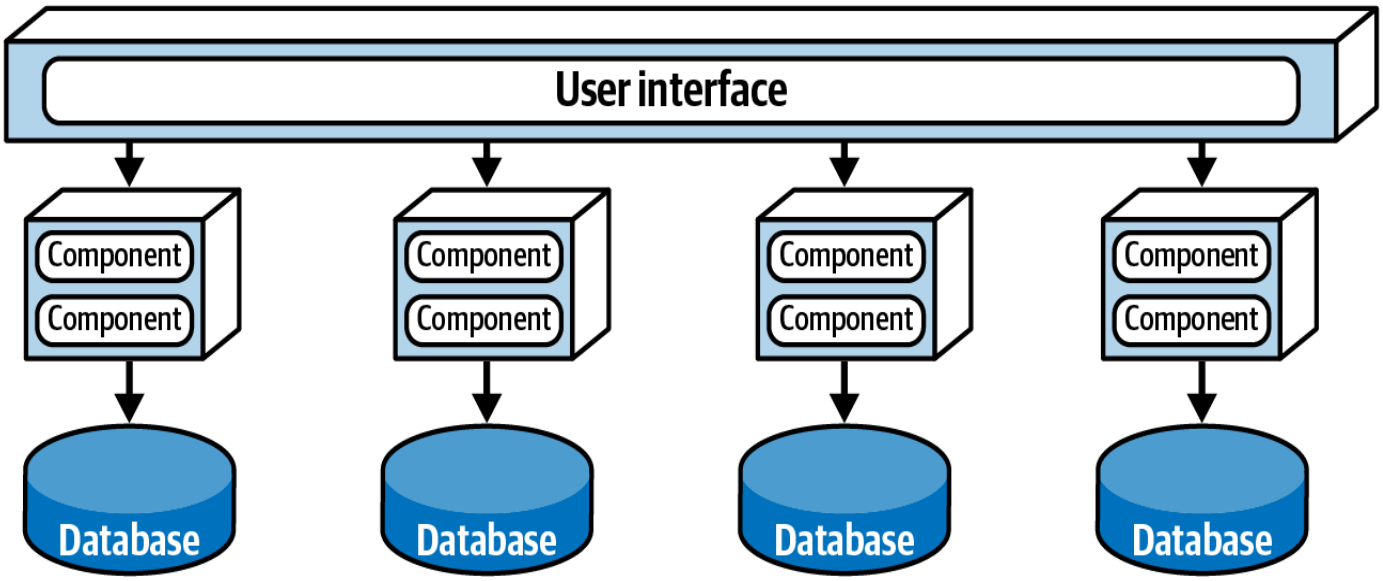

User interface
Component
Component
Component
Component
Component
Component
Component
Component
Database
Database
Database
Database

Abbildung 26-4. Gängige Arten von Datenbanktopologien

Die Datenbanktopologie muss auf die Architektur abgestimmt sein, damit sie richtig funktioniert. Microservices-Architekturen beispielsweise verwenden in der Regel ein Datenbank-pro-Service-Muster, um einen strikt begrenzten Kontext zu erhalten. Ohne diese Ausrichtung wäre es für die Architekten extrem schwierig, Änderungen zu kontrollieren, und die Betriebseigenschaften des Systems wie Fehlertoleranz, Skalierbarkeit, Elastizität, Wartbarkeit, Testbarkeit und Einsatzfähigkeit würden darunter leiden. Allerdings sind einige Stile, wie z. B. die servicebasierte Architektur (siehe Kapitel 14), in Bezug auf die physische Datenbanktopologie flexibler.

Architektonische Merkmale

In Teil II dieses Buches haben wir gezeigt, dass jeder Architekturstil seine Superkräfte (mit 4 bis 5 Sternen bewertet) und seine Schwächen (1 bis 2 Sterne) hat. Das gilt auch für Datenbanktypen - und es ist wichtig, die architektonischen Superkräfte eines Systems mit den entsprechenden Superkräften des Datenbanktyps in Einklang zu bringen. In unserem Buch *Softwarearchitektur: The Hard Parts* haben deine Autoren die Eigenschaften von sechs verschiedenen Datenbanktypen bewertet: relational, Key-Value, Document, Columnar, Graph und NoSQL. Du erinnerst dich vielleicht daran, dass Skalierbarkeit und Elastizität Superkräfte für Microservices, ereignisgesteuerte und raumbezogene Architekturen sind. Das gilt auch für Key-Value- und spaltenbasierte

Datenbanken, was bedeutet, dass diese Datenbanktypen eine gute Wahl sind, um dieseArchitektureigenschaften zu verstärken.

Datenstruktur

Die Struktur der Daten, die gespeichert und auf die zugegriffen wird, ist ebenfalls ein Faktor für diese Überschneidung. Wenn die *Datenstruktur* relational ist, d.h. auf einer Hierarchie von voneinander abhängigen Beziehungen beruht, ist eine relationale Datenbank gut geeignet. Die Speicherung von Schlüssel-Wert-Paaren in einer relationalen Datenbank ist jedoch eine Fehlausrichtung, die zu Ineffizienzen sowohl in der Datenbank als auch in der Architektur führen kann. Nicht alle Daten haben die gleiche Struktur, also behalte das im Auge. Einige Daten können relational sein, andere dokumentenbasiert (vor allem bei der Speicherung von JSON-basierten Ereignis- oder Anfrageladungen) und wieder andere Daten können schlüsselwertgesteuert sein. Angesichts der potenziellen Vielfalt der Datenstrukturen innerhalb einer bestimmten Architektur empfehlen wir, wann immer möglich, polyglotte Datenbanken zu nutzen.

Lese-/Schreibpriorität

Wenn das Geschäftsproblem ein hohes Lese- oder Schreibvolumen erfordert, ist das eine wichtige Information, um die Datenbanktopologie auf die Architektur abzustimmen. Wenn die Architektur z. B. ein hohes Schreibvolumen gegenüber seltenen Lesevorgängen erfordert, wäre eine spaltenbasierte Datenbank eine gute Lösung. Wenn das Gegenteil

der Fall ist und ein hohes Lesevolumen Priorität hat, wäre eine Schlüssel-Wert-Datenbank, eine Dokumentendatenbank oder eine Graphdatenbank besser geeignet. Wenn das System Lese- und Schreibvorgänge etwa gleich stark gewichtet, sind relationale und NewSQL-Datenbanken eine gute Wahl. Wenn du diesen Faktor nicht berücksichtigst, kann das zu schlecht funktionierenden Systemen führen.

Architektur- und Ingenieurpraktiken

Im späten 20. Jahrhundert wurden Dutzende von Softwareentwicklungsmethoden populär, darunter Wasserfall und viele Varianten von Agile (wie Scrum, Extreme Programming, Lean und Crystal). Damals glaubten die meisten Architekten, dass all dies keinen Einfluss auf die Softwarearchitektur hat, da sie die Entwicklung als einen völlig separaten Prozess betrachteten. In den letzten Jahren haben die Fortschritte in der Technik jedoch dazu geführt, dass auch die Softwarearchitektur von den Prozessen betroffen ist. Es ist sinnvoll, die *Softwareentwicklungsprozesse* von den *technischen Verfahren* zu trennen. Mit *Prozessen* meinen wir, wie Teams gebildet und verwaltet werden, wie Besprechungen ablaufen und Arbeitsabläufe organisiert werden - kurz gesagt, die Mechanismen, wie Menschen sich organisieren und zusammenarbeiten. *Entwicklungspraktiken* hingegen beziehen sich auf die prozessunabhängigen Techniken und Werkzeuge, die Teams zur Entwicklung und Veröffentlichung von Software einsetzen. Extreme Programming (XP), Continuous Integration (CI), Continuous Delivery

(CD) und Test Driven Development (TDD) sind zum Beispiel bewährte technische *Verfahren*, die sich nicht auf einen bestimmten Prozess stützen. Der Begriff *Softwareentwicklung* umfasst also sowohl die Softwareentwicklung als auch diese Praktiken.

Die Konzentration auf technische Praktiken ist wichtig. Der Softwareentwicklung fehlen viele der Merkmale ausgereifterer Ingenieurdisziplinen. Bauingenieure können zum Beispiel bauliche Veränderungen viel genauer vorhersagen als Softwareentwickler ähnliche Aspekte der Softwarestruktur. Das bedeutet, dass die Achillesferse der Softwareentwicklung die Schätzung ist: wie viel Zeit, wie viele Ressourcen, wie viel Geld? Ein Grund für diese Schwierigkeit ist, dass die traditionellen Methoden der Schätzung dem explorativen Charakter der Softwareentwicklung und den Unbekannten, die bei der Entwicklung von Software typischerweise auftreten, nicht gerecht werden.

Der Prozess ist zwar größtenteils von der Architektur getrennt, aber iterative Prozesse passen besser zu ihrer Natur. Der Versuch, ein modernes System wie Microservices mit einem antiquierten Prozess wie Wasserfall aufzubauen, wird zu großen Reibungen führen. Ein Aspekt der Architektur, bei dem agile Methoden wirklich glänzen, ist die Migration von einem Architekturstil zu einem anderen. Agile Methoden unterstützen solche Veränderungen besser als planungslastige Prozesse, weil sie enge Feedbackschleifen haben und Techniken wie das Strangler-Muster und Feature-Toggles fördern.

Architekten fungieren oft auch als technische Leiter von Projekten, was bedeutet, dass sie bestimmen, welche technischen Verfahren das Team anwendet. Genauso wie die Problemdomäne vor der Wahl einer Architektur sorgfältig geprüft werden muss, müssen Architekten auch sicherstellen, dass ihr Architekturstil und ihre technischen Praktiken ineinandergreifen. Die Architekturphilosophie der Microservices geht beispielsweise davon aus, dass die Teams Dinge wie die Bereitstellung, das Testen und das Deployment von Maschinen automatisieren werden. Der Versuch, eine Microservices-Architektur mit einer veralteten Betriebsgruppe, manuellen Prozessen und wenig Tests aufzubauen, würde wahrscheinlich zum Scheitern führen. So wie sich verschiedene Problembereiche für bestimmte Architekturen eignen, so sind es auch verschiedene technische Verfahren.

Die Evolution des Denkens geht weiter, von Extreme Programming bis hin zu Continuous Delivery und darüber hinaus, da Fortschritte in den technischen Praktiken neue Architekturfähigkeiten ermöglichen. Neals Buch *Building Evolutionary Architectures* zeigt neue Denkansätze für die Überschneidung von technischen Praktiken und Architektur auf, die die Automatisierung der Architektursteuerung verbessern können. Das Buch bietet eine wichtige neue Nomenklatur und Denkweise für architektonische Merkmale und behandelt Techniken zum Aufbau von Architekturen, die sich im Laufe der Zeit anmutig verändern.

In der Welt der Softwareentwicklung bleibt nichts statisch. Architekten mögen ein System entwerfen, um bestimmte Kriterien zu erfüllen, aber um sicherzustellen, dass ihre Entwürfe sowohl die Implementierung als

auch den unvermeidlichen Wandel überleben, brauchen wir eine *evolutionäre Architektur*.

Building Evolutionary Architectures führt in das Konzept ein, *architektonische Fitnessfunktionen* zu nutzen, um architektonische Merkmale zu schützen (und zu steuern), wenn sich diese im Laufe der Zeit verändern. Aus Kapitel 6 wissen wir, dass architektonische Fitnessfunktionen eine Möglichkeit sind, eine objektive Bewertung der Integrität bestimmter Architektureigenschaften zu erhalten. Diese Bewertung kann eine Vielzahl von Mechanismen umfassen, z. B. Metriken, Einheitstests, Monitore und Chaos Engineering.

Um zu sehen, wie Fitnessfunktionen dabei helfen können, diesen Schnittpunkt auszurichten, betrachte das Unternehmensproblem einer kurzen Markteinführungszeit. Die Markteinführungszeit ist gleichbedeutend mit *Agilität - der*Fähigkeit des Systems, schnell auf Veränderungen zu reagieren. Agilität ist eine zusammengesetzte Architektureigenschaft, die sich aus Wartbarkeit, Testbarkeit und Einsatzfähigkeit zusammensetzt (siehe Kapitel 6). Alle drei Architektureigenschaften werden durch technische Praktiken und Verfahren beeinflusst und können daher durch Fitnessfunktionen gemessen und verfolgt werden. Zum Beispiel unterstützen sowohl Microservices als auch servicebasierte Architekturen ein hohes Maß an Agilität. Wenn jedoch die technischen Verfahren, die diese Architekturmerkmale umgeben, nicht mit der Architektur übereinstimmen, wird das System die Agilitätsziele und -anforderungen nicht erfüllen. Mit Hilfe von Fitnessfunktionen kann eine Fehlanpassung

erkannt und der Architekt aufgefordert werden, Maßnahmen zu ergreifen, um die Engineering-Praktiken an die Architektur anzupassen (oder umgekehrt).

Architektur und Team-Topologien

Wie wir in Teil II des Buches besprochen haben, kann sich die Teamtopologie direkt auf die Softwarearchitektur auswirken und umgekehrt. Diese Abstimmung ist so wichtig, dass wir für jeden Architekturstil, den wir in diesem Buch vorstellen, einen Abschnitt über Teamtopologien eingefügt haben.

Eine der grundlegendsten Arten, wie Team-Topologien mit der Architektur übereinstimmen können, ist die Art der Partitionierung. Wie Architekturen können auch Teams in Bereiche oder technische Bereiche unterteilt sein. Fachlich gegliederte Teams sind nach Fachgebieten organisiert und in der Regel funktionsübergreifend, mit Spezialisierung innerhalb des Teams. Ein fachlich gegliedertes Team könnte sich z. B. auf den kundenorientierten Teil eines Systems konzentrieren und wäre dann für die End-to-End-Verarbeitung der kundenbezogenen Funktionen von der Benutzeroberfläche bis zur Datenbank zuständig. Technisch unterteilte Teams konzentrieren sich dagegen jeweils auf eine bestimmte technische Funktion der Architektur und sind in der Regel nach technischen Kategorien organisiert: UI-Teams, Backend-Processing-Teams, Shared-Services-Teams und Datenbank-Teams würden zum Beispiel sehr gut zum Stil der Schichtenarchitektur passen. Alternativ können diese Teams technisch in Geschäftsfunktionsteams und

Datensynchronisationsteams aufgeteilt werden, was gut zum raumbezogenen Architekturstil passen würde.

Um den Erfolg des Systems zu gewährleisten, ist es wichtig zu verstehen, wie die Teams organisiert sind. Wenn die Teamtopologie der Organisation nicht mit der Architektur übereinstimmt, werden die Teams Schwierigkeiten haben, die Architektur zu implementieren und zu pflegen, und es ist unwahrscheinlich, dass sie ihre Geschäftsziele erreichen.

Architektur und Systemintegration

Systeme leben selten in Isolation. Die meisten benötigen zusätzliche Verarbeitungen und Daten von anderen Systemen - und das bringt uns zum Schnittpunkt von Architektur und Systemintegration. Wenn ein System mit einem anderen System kommunizieren muss, um zusätzliche Verarbeitungen durchzuführen oder Daten abzurufen, steht der Architekt vor einer Reihe von Herausforderungen und Auswirkungen. Ist das aufgerufene System zum Beispiel verfügbar? Ist es so skalierbar und leistungsfähig, dass es den Anforderungen des aufrufenden Systems entspricht?

Wenn Architekten der Systemintegration nicht genügend Aufmerksamkeit schenken, führt die statische und dynamische Kopplung der Systeme oft zu Architekturen, die nicht skalierbar, nicht reaktionsschnell und nicht flexibel genug sind. Bei der Integration mit anderen Systemen ist zu überlegen, welche Kommunikationsprotokolle

verwendet werden sollen, welche Arten von Verträgen zwischen den Systemen bestehen sollen, ob die architektonischen Merkmale der Systeme kompatibel sind und ob die Integration das architektonische Quantum jedes Systems bewahrt.

Architektur und das Unternehmen

Jedes Unternehmen hat eine Reihe von Standards und Leitprinzipien. Mit *Unternehmen* meinen wir die Gesamtheit aller Systeme und Produkte innerhalb eines Unternehmens (oder einer Abteilung oder eines Bereichs innerhalb des Unternehmens). Viele Unternehmen schreiben zum Beispiel bestimmte Sicherheitsstandards, Praktiken oder Verfahren für architektonische Lösungen vor, unabhängig von der Art des Systems. Unternehmensstandards können auch Plattformen, Technologien, Dokumentationsstandards und Diagrammstandards umfassen, um nur einige zu nennen. Achte auf die Standards und Praktiken auf Unternehmensebene und stelle sicher, dass die Architektur mit ihnen übereinstimmt.

Wir haben schon viele Situationen erlebt, in denen der Architekt Praktiken, Standards und Verfahren auf Unternehmensebene ignoriert hat. Das Ergebnis war in der Regel, dass die architektonische Lösung, so effektiv sie auch sein mochte, als fehlgeschlagene "Einzellösung" betrachtet und verworfen wurde. Wir können gar nicht oft genug betonen, wie wichtig es ist, die Architektur mit den Praktiken des Unternehmens abzustimmen, um ihren Erfolg zu gewährleisten.

Architektur und das wirtschaftliche Umfeld

Das Unternehmensumfeld hat einen erheblichen und *direkten* Einfluss auf die Architektur seiner Systeme (und umgekehrt), und das Unternehmensumfeld ändert sich ständig. Muss das Unternehmen strenge Kostensenkungsmaßnahmen durchführen, um sich über Wasser zu halten, oder expandiert es aggressiv? Muss sich das Unternehmen jedes Quartal neu positionieren, um seine Nische in einem stark schwankenden und wettbewerbsintensiven Markt zu finden, oder befindet es sich in einer stabilen Position? Ein effektiver Softwarearchitekt versteht die Position und die Richtung des Unternehmens und passt die Architekturen der kritischen Systeme an das Geschäftsumfeld an.

Wir nennen diese Ausrichtung *Domänen-zu-Architektur-Isomorphismus*. Unternehmen, die extreme Kostensenkungsmaßnahmen durchführen, würden beispielsweise nicht gut zu Microservices oder raumbezogenen Architekturen passen, da diese sehr kostspielig in der Erstellung und Wartung sind. Umgekehrt wären Unternehmen, die durch Fusionen und Übernahmen aggressiv expandieren, mit monolithischen Architekturen, die sich nicht weiterentwickeln und anpassen können, nicht gut bedient.

Ein typisches Problem, mit dem sich Architekten an dieser Schnittstelle konfrontiert sehen, ist die Veränderung von Unternehmen,

insbesondere die *unbekannte Veränderung*. Der ehemalige US-Verteidigungsminister Donald Rumsfeld hat einmal gesagt, dass:

> *Wir wissen, dass es Dinge gibt, von denen wir wissen, dass wir sie wissen. Wir wissen auch, dass es bekannte Unbekannte gibt; das heißt, wir wissen, dass es Dinge gibt, die wir nicht wissen. Aber es gibt auch unbekannte Unbekannte - Dinge, von denen wir nicht wissen, dass wir sie nicht wissen.*

Viele Produkte und Systeme beginnen mit einer Liste *bekannter Unbekannter*: Dinge, die die Entwickler über den Bereich und die Technologie lernen müssen, von denen sie wissen, dass sie sich ändern werden. Dieselben Systeme werden aber auch Opfer von *unbekannten Unbekannten*: Dinge, von denen niemand wusste, dass sie auftauchen würden, die aber unerwartet auftauchen. "Unbekannte Unbekannte" sind die Erzfeinde von Softwaresystemen. Das ist der Grund, warum alle "Big Design Up Front" Software-Bemühungen leiden: Architekten können nicht für unbekannte Unbekannte entwerfen. Um Mark zu zitieren:

> *Alle Architekturen werden aufgrund unbekannter Unbekannter iterativ. Agile erkennt das einfach und macht es früher.*

Die Planung für Veränderungen in der Softwarearchitektur ist *schwierig*. Evolutionäre Architekturpraktiken helfen dabei, die sich ständig verändernde Geschäftslandschaft zu bewältigen, ebenso wie die iterative Architektur. Architekturmerkmale wie *Übertragbarkeit*, *Skalierbarkeit*,

Entwicklungsfähigkeit und *Anpassungsfähigkeit* tragen ebenfalls dazu bei, eine Softwarearchitektur flexibler und anpassungsfähiger zu machen.

Barry O'Reilly, ein erfahrener Softwarearchitekt, der sich auf Komplexitätstheorie und Softwaredesign spezialisiert hat, entwickelte eine neue Denkweise über den ständigen Wandel in Unternehmen, die sogenannte *Residualitätstheorie*. In seinem Buch *Residues: Time, Change, and Uncertainty in Software Architecture* (Leanpub, 2024) beschreibt O'Reilly Techniken, um geschäftliche Veränderungen als *Stressoren* und die entsprechenden architektonischen Veränderungen als *Residuen* zu behandeln. Seine Theorie besagt, dass, wenn der Architekt auf Veränderungen reagiert, indem er immer mehr Rückstände in die Architektur einbaut, diese Rückstände schließlich *unbekannte* Veränderungen adressieren, die der Architekt unmöglich vorhersagen kann, wodurch eine Architektur entsteht, die einen kritischen Zustand im Sinne der Komplexitätstheorie erreicht hat. Das ist eine interessante Theorie, die wir genau beobachten werden.

Architektur und generative KI

Während wir Anfang 2025 die zweite Auflage dieses Buches schreiben, haben *generative künstliche Intelligenz* (Gen AI) und große Sprachmodelle (LLMs) die Welt der Softwareentwicklung und des Softwaredesigns infiltriert. Viele Unternehmen bauen sie in ihre Systeme ein, um Aufgaben zu erledigen, die früher nur von Menschen manuell ausgeführt wurden. Es überrascht nicht, dass Gen AI auch in die Softwarearchitektur einfließt: Architekten integrieren LLMs in ihre

Softwarearchitekturen, und einige setzen sogar Gen AI-Tools ein, um sie beim Durchdenken schwieriger Probleme zu unterstützen.

Generative KI in die Architektur einbinden

Ein Ansatz, den wir für die Einbindung von Gen AI in eine Architektur empfehlen, ist die Nutzung von *Abstraktion* und *Modularität*. Es ist wichtig, dass eine LLM schnell durch eine andere ersetzt werden kann und dass Leitplanken (rails) und Evaluierungsergebnisse (evals) von verschiedenen LLMs berücksichtigt werden.

Nehmen wir zum Beispiel an, ein Jobsuchunternehmen möchte Gen AI nutzen, um Lebensläufe zu anonymisieren, um Vorurteile abzubauen und sich auf die Fähigkeiten der Arbeitssuchenden zu konzentrieren, anstatt auf ihre demografischen Daten oder andere Faktoren. Diese Aufgabe wird normalerweise von Menschen erledigt, könnte aber auch leicht von einem LLM übernommen werden. Aber sind die Ergebnisse des LLM genau? Entfernt er zu viele Informationen aus dem Lebenslauf? Behält es zu viele demografische Informationen an Ort und Stelle? Für diese Art von System ist es unerlässlich, Stichproben und Kennzahlen zu sammeln und LLM-Maschinen zu vergleichen. Tools wie Langfuse helfen dabei, diese Art der Beobachtbarkeit innerhalb der Architektur zu schaffen.

Generative KI als Assistent für Architekten

Mit der richtigen Eingabeaufforderung kann ein LLM (wie Copilot) Quellcode erzeugen, der Entwicklern viel Zeit und Mühe erspart. Sie

eignen sich hervorragend für die Lösung sehr spezifischer, deterministischer Probleme, wie z. B. "Schreibe Quellcode in der Programmiersprache C#, der eine eindeutige vierstellige PIN-Nummer erzeugt, die keine sich wiederholenden Ziffern hat." Aber kann die LLM-Technologie Softwarearchitekten bei typischen Aufgaben unterstützen? Hier sind einige allgemeine Beispiele für architekturbezogene Eingabeaufforderungen:

- Risikobewertung: "Gibt es in dieser Architektur Risikobereiche?"
- Risikominderung: "Wie sollte ich dieses Risiko angehen?"
- Antipatterns: "Gibt es in dieser Architektur gängige Gegenmuster?"
- Entscheidungen: "Soll ich Orchestrierung oder Choreografie für diesen Arbeitsablauf verwenden?"

Zum Zeitpunkt der Erstellung dieser zweiten Ausgabe (Anfang 2025) hatten wir noch keinen großen Erfolg mit diesem Unterfangen. Wenn man einen LLM fragt, ob Microservices oder eine raumbezogene Architektur für eine bestimmte Situation am besten geeignet sind, erhält man selten (wenn überhaupt) die richtige Antwort. Warum? Weil, wie wir in diesem Buch gezeigt haben, *alles in der Softwarearchitektur ein Kompromiss ist*. LLMs sind großartig, wenn es darum geht, *Wissen* zu verstehen, aber bis heute fehlt ihnen die nötige *Weisheit*, um angemessene Entscheidungen zu treffen. Diese Weisheit umfasst so viel Kontext, dass es für den Architekten viel schneller ist, ein Geschäftsproblem selbst zu lösen, als einem LLM alles über das Problem und seine erweiterte Umgebung und den Kontext beizubringen. Dass dies eine gewaltige Aufgabe ist, zeigt die Tatsache, dass wir noch acht

weitere Schnittpunkte aufgeführt haben, über die man sich Gedanken machen sollte.

Dennoch haben wir einige vielversprechende Tools gesehen. Thoughtworks Haiven zum Beispiel kann ein Architekturdiagramm interpretieren und eine Softwarearchitektur vollständig beschreiben. Das erspart die Arbeit, ein Diagramm in ein maschinenlesbares Format wie XML zu exportieren und es zur Eingabeaufforderung für das LLM zu verwenden. Sobald die Informationen importiert sind, können die Nutzer/innen Haiven einfache Fragen zur Architektur stellen, z. B. ob es Engpässe oder Probleme erkennen kann. Außerdem wird ein LLM eingesetzt, um ein PlantUML-Diagramm oder eine pseudosprachliche Beschreibung einer Architektur in ausführbaren ArchUnit-Code zu übersetzen, um die Struktur eines Systems zu steuern. In diesem Bereich gibt es viele Aktivitäten, so dass in den kommenden Jahren mit raschen Veränderungen in der Art und Weise zu rechnen ist, wie Gen AI Architekten unterstützen kann.

Zusammenfassung

Softwarearchitektur ist eine ganzheitliche Aktivität, die viele Facetten einer Organisation umfasst. Effektive Softwarearchitekten wissen, dass die Erstellung und Pflege einer Architektur viel mehr ist als nur die Auswahl eines bestimmten Architekturstils und die Umsetzung. Es geht auch darum, sicherzustellen, dass die Architektur mit anderen Aspekten der Umgebung abgestimmt ist, und die in Teil III beschriebenen

Fähigkeiten zur Kommunikation und Zusammenarbeit einzusetzen, um diese Abstimmung zu erreichen.