Team Topologies

Organisation von Business- und IT-Teams für einen schnellen Arbeitsfluss

6 Entscheiden Sie sich für teamorientierte Grenzen

> Wenn Code nicht funktioniert [...] liegt das Problem darin, wie Teams organisiert sind und wie Menschen interagieren.
>
> **– Eric Evans,** *Domain-Driven Design*

Ein reibungsloser Arbeitsfluss ist schwer zu erreichen, wenn jedes Team von einem komplizierten Netz an Interaktionen mit vielen anderen Teams abhängig ist. Damit Änderungen an Softwaresystemen schnell vonstattengehen können, müssen wir Übergaben abschaffen und die meisten Teams auf die wichtigsten Änderungsströme innerhalb der Organisation ausrichten. Viele Unternehmen haben jedoch große Probleme mit den Zuständigkeitsgrenzen, die den Teams zugewiesen werden. In der Regel wird wenig über die Tragfähigkeit der Grenzen für Teams nachgedacht, was zu mangelnder Eigenverantwortung, mangelndem Engagement und einem sehr langsamen Arbeitstempo führt.

In diesem Kapitel definieren und erkunden wir Wege, um geeignete Grenzen innerhalb und zwischen Softwaresystemen zu finden, die es den Teams ermöglichen, ihren Teil des Systems effektiv und nachhaltig zu verantworten und weiterzuentwickeln, und zwar auf eine Weise, die den Arbeitsfluss fördert. Diese Techniken lassen sich sowohl auf monolithische Software als auch auf Software anwenden, die bereits lockerer gekoppelt ist. Entscheidend ist, dass diese Grenzen »teamgerecht« sind: Wir passen die Software- und Systemgrenzen an die Fähigkeiten eines einzelnen Teams an, wodurch die Eigenverantwortung und die kontinuierliche Weiterentwicklung der Software sofort viel leichter möglich sein werden.

Indem wir die Zuständigkeitsgrenzen zwischen den Teams sorgfältig ausloten und validieren – und Techniken wie Domain-driven Design und Fracture Planes einsetzen – richten wir die Softwarearchitektur an der Domäne der Problemstellung aus, verbessern den Änderungsfluss und geben der Organisation die Möglichkeit, das sozio-technische System schneller und effektiver weiterzuentwickeln.

Ein teamorientierter Ansatz für Zuständigkeiten und Grenzen von Software

Viele Probleme bei der Bereitstellung von Software entstehen durch versehentlich unklare Abgrenzungen zwischen verschiedenen Teams und deren Verantwortlichkeiten. Dies wird oft durch eine Softwarearchitektur mit hoher Kopplung zwischen ihren einzelnen Bestandteilen überschattet (auch wenn die Architektur auf dem Papier eigentlich hochgradig modular und erweiterbar sein sollte), wie das Conway'sche Gesetz besagt. Ein solches System wird gemeinhin als »Monolith« bezeichnet.

Die in *Accelerate* veröffentlichte Forschung zeigt, dass eng gekoppelte Architekturen sich negativ auf die Fähigkeit autonomer Teams mit klaren Verantwortlichkeiten auswirken. Die Autorin und die Co-Autoren erwähnen auch architektonische Ansätze, die helfen, solche Architekturen zu entkoppeln: »Zu den architektonischen Ansätzen, die diese Strategie [der Unterstützung der vollen Eigenverantwortung der Teams von der Konzeption bis zur Bereitstellung] ermöglichen, gehört die Verwendung von Bounded Contexts und APIs als Möglichkeit, große Domänen in kleinere, lose gekoppelte Einheiten zu zerlegen.«[1]

Aber wenn wir von einem monolithischen Softwaresystem zu lose gekoppelten Services übergehen wollen, müssen wir auch berücksichtigen, wie sich die neue Architektur auf die beteiligten Teams auswirkt. Wir müssen ihre kognitiven Fähigkeiten, ihren Arbeitsort und ihr Interesse an den neuen Services in Betracht ziehen.

Wenn wir den Blickwinkel des Teams nicht berücksichtigen, laufen wir Gefahr, den Monolithen an den falschen Stellen aufzuteilen oder sogar ein komplexes System aus miteinander verwobenen Services zu schaffen. Dies wird als »verteilter Monolith« bezeichnet und führt dazu, dass Teams keine Autonomie über ihre Services haben, da fast alle Änderungen eine Aktualisierung der anderen Services erfordern. Wie das Beispiel der Service-Teams von Amazon (Kapitel 4) zeigt, müssen wir die Teaminteraktionen durchdenken und gezielt steuern, um die gewünschte Unabhängigkeit der Services zu erreichen.

Versteckte Monolithen und Kopplung

Es gibt viele Arten von monolithischer Software, von denen einige auf den ersten Blick schwer zu erkennen sind. Viele Unternehmen haben sich beispielsweise die Zeit und Mühe gemacht, eine monolithische Anwendung in kleinere Services aufzuteilen, nur um dann im weiteren Verlauf der Deployment-Pipeline einen monolithischen Release zu produzieren und damit eine Gelegenheit zu verpassen, schneller und sicherer zu arbeiten. Wir müssen uns darüber im Klaren sein, mit welchen Arten von Monolithen wir arbeiten, bevor wir Änderungen vornehmen.

1 Forsgren et al., »Accelerate«, 63.

Anwendungsmonolith

Ein *Anwendungsmonolith* ist eine einzelne, große Anwendung mit vielen Abhängigkeiten und Verantwortlichkeiten, die möglicherweise viele Services und/oder verschiedene User Journeys bereitstellt. Solche Anwendungen werden in der Regel als Einheit ausgeliefert, was Benutzerinnen (die Anwendung ist während der Auslieferung nicht verfügbar) und Betreibern (unerwartete Probleme, weil die Produktionsumgebung ein bewegliches Ziel ist; selbst wenn wir den Monolithen in einer Umgebung getestet haben, die der Produktionsumgebung ähnelt, hat er sich seitdem sicherlich verändert) oft Kopfzerbrechen bereitet.

Datenbankmonolith

Ein *Datenbankmonolith* besteht aus mehreren Anwendungen oder Diensten, die alle an dasselbe Datenbankschema gekoppelt sind, was es schwierig macht, sie separat zu ändern, zu testen und einzusetzen. Dieser Monolith resultiert oft daraus, dass das Unternehmen die Datenbank und nicht die Services als den zentralen Geschäftsmotor ansieht. Häufig wurden ein oder mehrere Datenbankadministrations-Teams (DBA-Teams) eingesetzt, um nicht nur die Datenbank zu pflegen, sondern auch Änderungen an der Datenbank zu koordinieren – eine Aufgabe, für die sie oft unterbesetzt sind –, und sie werden zu einem großen Engpass bei der Umsetzung.

Monolithische Builds (alles neu bauen)

Ein *monolithischer Build* verwendet einen gigantischen Continuous-Integration-(CI-)Build, um eine neue Version einer Komponente zu erzeugen. Monolithische Anwendungen führen zu monolithischen Builds, aber auch bei kleineren Services ist es möglich, dass die Build-Skripte darauf abzielen, die gesamte Codebasis zu bauen, anstatt standardmäßige Mechanismen für das Dependency-Management zwischen Komponenten (wie etwa Pakete oder Container) zu verwenden.

Monolithische (gekoppelte) Releases

Ein *monolithisches Release* ist eine Reihe kleinerer Komponenten, die zu einem »Release« gebündelt werden. Wenn Komponenten oder Services unabhängig voneinander in der CI erstellt werden können, aber nur in einer gemeinsamen statischen Umgebung ohne Service Mocks getestet werden können, bringt man schließlich alle aktuellen Versionen der Komponenten in dieselbe Umgebung. Sie setzen dann alle Komponenten als eine Einheit ein, weil sie dann sicher sein können, dass das, was sie getestet haben, auch in der Produktion läuft. Manchmal ist dieser Ansatz auch das Ergebnis eines separaten QA-Teams, das für das Testen der verschiedenen Komponenten zuständig ist (das Zusammenfassen mehrerer Änderungen an einem Service ist aus der Sicht eines QA-Teams mit begrenzter Kapazität sinnvoll).

Monolithisches Modell (eine Sicht auf die Welt)

Ein monolithisches Modell ist eine Software, die versucht, eine einzige Domänensprache und -darstellung (Format) in vielen verschiedenen Kontexten durchzusetzen. Es mag zwar sinnvoll sein, diese Art von Konsistenz in kleinen Organisationen zu bevorzugen (aber auch nur dann, wenn die Teams ausdrücklich zustimmen, dass dies eine gute Idee ist), aber sobald eine Organisation mehr als eine Handvoll Teams und/oder Domänen umfasst, kann dieser Ansatz ungewollt zu Einschränkungen in der Architektur und der Implementierung führen.

Monolithisches Denken (Standardisierung)

Monolithisches Denken ist ein »One Size Fits All«-Denken (Einheitsdenken) für Teams, das zu unnötigen Einschränkungen bei Technologien und Implementierungsansätzen zwischen Teams führt. Alles zu standardisieren, um Abweichungen zu minimieren, vereinfacht zwar die Kontrolle durch das Management von Entwicklungsteams, ist aber mit einem hohen Preis verbunden. Gute Entwicklerinnen sind in der Lage und bereit, neue Techniken und Technologien zu erlernen. Wenn man den Teams die Wahlfreiheit nimmt, indem man ihnen einen einzigen Technologie-Stack und/oder ein einziges Tool vorschreibt, schadet das ihrer Fähigkeit, das richtige Tool für die jeweilige Aufgabe zu verwenden, und verringert (oder tötet manchmal) ihre Motivation. In *Accelerate* erwähnen die Autorin und ihre Co-Autoren, dass ihre Forschungsergebnisse darauf hindeuten, dass der Zwang zur Standardisierung die Lern- und Experimentierfreudigkeit von Teams verringert und zu einer schlechteren Lösungsauswahl führt.[2]

Monolithischer Arbeitsplatz (Großraumbüro)

Bei einem *monolithischen Arbeitsplatz* handelt es sich um ein einheitliches Bürolayout-Muster für alle Teams und Einzelpersonen am selben geografischen Standort – typischerweise isolierte Einzelarbeitsplätze (Kabinen) oder ein Großraumbüro ohne explizite Barrieren zwischen den Schreibtischen der Mitarbeiter.

Die Vorstellung, dass Büros ein standardisiertes Layout haben sollten, ist weit verbreitet. Das mag zwar die Arbeit des ausführenden Bauunternehmens vereinfachen, kann aber immer wieder negative Auswirkungen auf Einzelpersonen und Teams haben. Darüber hinaus wurde der weit verbreitete Glaube, dass Großraumbüros die Zusammenarbeit fördern, durch eine Praxisstudie widerlegt, in der festgestellt wurde, dass in zwei Unternehmen, die Großraumbüros einführten, »der Umfang der persönlichen Interaktion deutlich abnahm (ca. 70 %) [...] bei gleichzeitiger Zunahme der elektronischen Interaktion«.[3] Unserer Erfahrung nach geschieht dies, wenn das Missverständnis besteht, dass es um die Zusammenführung

2 Forsgren et al., »Accelerate«, 66.

3 Bernstein and Turban, »The Impact of the ›Open‹ Workspace on Human Collaboration«.

von Zielvorstellungen und nicht nur um die Zusammenführung von Körpern geht. (Siehe Kapitel 2 für weitere Ideen zur teamorientierten Gestaltung von Büroräumen und Kapitel 7 für verschiedene Formen der Teaminteraktion.)

Softwaregrenzen oder »Bruchflächen«

Obwohl jede Art von Monolith bestimmte Nachteile mit sich bringt, gibt es auch Gefahren, die bei der Aufteilung von Software zwischen Teams zu beachten sind. Die Aufteilung von Software kann die Konsistenz zwischen verschiedenen Teilen der Software beeinträchtigen und zu einer unbeabsichtigten Duplizierung von Daten über mehrere Teilsysteme hinweg führen. Die User Experience (UX) über mehrere Teile der Software hinweg kann sich verschlechtern, wenn wir nicht darauf achten, eine kohärente UX zu erreichen, und es kann zusätzliche Komplexität entstehen, wenn wir Software in ein verteiltes System zerlegen.

Zunächst müssen wir verstehen, was eine Bruchfläche ist. Eine *Bruchfläche (Fracture Plane)* ist eine natürliche Verbindungsnaht im Softwaresystem, die eine einfache Aufteilung des Systems in zwei oder mehr Teile ermöglicht. Diese Aufteilung von Software ist besonders nützlich bei monolithischer Software. Das Wort Monolith selbst kommt aus dem Griechischen und bedeutet »einzelner Stein«. Traditionelle Steinmetze schlagen Steine in bestimmten Winkeln, um sie in saubere Segmente zu spalten, wobei sie die natürlichen Bruchflächen nutzen. Wir können nach ähnlichen Bruchflächen in Software suchen, um die natürlichen Spaltpunkte zu finden, die zu Softwaregrenzen führen.

In der Regel ist es am besten, wenn man versucht, die Grenzen der Software an die verschiedenen Bereiche der geschäftlichen Domäne anzupassen. Ein Monolith ist schon aus technischer Sicht problematisch genug (insbesondere, weil er die Wertschöpfung im Laufe der Zeit verlangsamt, da das Erstellen, Testen und Beheben von Problemen immer mehr Zeit in Anspruch nimmt). Wenn dieser Monolith auch noch mehrere geschäftliche Domänen antreibt, ist das ein Garant für ein Desaster, das die Priorisierung, den Arbeitsfluss und die User Experience beeinträchtigt.

Es gibt jedoch eine Vielzahl anderer möglicher Bruchflächen für Software, und zwar nicht nur die Bereiche der Geschäftsdomänen. Wir können und sollten einen Monolithen aufbrechen, indem wir verschiedene Arten von Bruchflächen kombinieren.

Bruchfläche: an Geschäftsdomäne angelehnter Bounded Context

Die meisten unserer Bruchflächen (Grenzen der Software-Verantwortlichkeit) sollten auf Bounded Contexts entlang von Geschäftsdomänen ausgerichtet sein. Ein Bounded Context ist eine Einheit zur Partitionierung eines größeren Domänen- (oder System-)Modells in kleinere Teile, von denen jeder einen intern konsistenten

Geschäftsdomänenbereich darstellt (der Begriff wurde in dem Buch *Domain-Driven Design* von Eric Evans[4] eingeführt).

Martin Fowler erklärt, dass ein Bounded Context ein in sich konsistentes Modell des Bereichs einer Domäne haben muss:

> Bei DDD [Domain-driven Design] geht es um den Entwurf von Software auf der Grundlage von Modellen der zugrunde liegenden Domäne. Ein Modell dient als Ubiquitous Language (allgegenwärtige Sprache), die die Kommunikation zwischen Softwareentwicklern und Domänen-Expertinnen erleichtert. Es dient auch als konzeptionelle Grundlage für den Entwurf der Software selbst – wie sie in Objekte oder Funktionen zerlegt wird. Um effektiv zu sein, muss ein Modell einheitlich sein, d. h., es muss in sich konsistent sein, damit es keine Widersprüche gibt.[5]

In dem Buch *Designing Autonomous Teams and Services* geben die DDD-Experten Nick Tune und Scott Millett ein Beispiel für einen Online-Musik-Streaming-Service mit drei Subdomänen, die sich gut an die Geschäftsbereiche anpassen lassen: Media Discovery (Suche nach neuer Musik), Media Delivery (Streaming an Hörer) und Licensing (Rechteverwaltung, Tantiemenzahlungen usw.).[6]

Die Identifizierung von Bounded Contexts erfordert ein gewisses Maß an fachlichem und technischem Know-how, sodass anfangs Fehler nicht ungewöhnlich sein dürften. Das sollte Sie aber nicht davon abhalten, sich zu verbessern und anzupassen, wenn Sie Ihren Kontext besser verstehen, auch wenn das eine Art von wiederkehrenden »Kosten« für die Neugestaltung von Services bedeutet. In unserem Design gibt es oft ein gewisses Maß an semantischer Kopplung, wobei, um es mit den Worten von Michael Nygard zu sagen: »Ein Konzept kann als atomar erscheinen, weil wir nur ein einziges Wort haben, um es zu beschreiben. Wenn man genau hinschaut, findet man Bruchstellen, an denen das Konzept aufgebrochen werden kann.«[7] Mit anderen Worten: Bei der Dekomposition von Systemen nach Bounded Contexts ist eine stückweise Evolution zu erwarten.

Weitere Vorteile der Anwendung von DDD sind die Konzentration auf die Kernkomplexität und die Geschäftsmöglichkeiten innerhalb eines Bounded Context für eine bestimmte Domäne, die Erforschung von Modellen durch die Zusammenarbeit von Fachleuten (weil es jetzt kleinere Domänen gibt, über die man nachdenken kann), die Entwicklung von Software, die diese Modelle explizit ausdrückt, und die Tatsache, dass sowohl Business Owner als auch Technologieexpertinnen eine Ubiquitous Language innerhalb eines Bounded Context sprechen.

Zusammenfassend lässt sich sagen, dass die Bruchfläche der Domäne die Technologie mit der Fachlichkeit in Einklang bringt und Unstimmigkeiten in der Terminologie und »Lost in Translation«-Probleme reduziert, wodurch der Änderungsfluss verbessert und Nacharbeit reduziert wird.

4 Evans, »Domain-Driven Design«.

5 Fowler, »Bliki: BoundedContext«.

6 Tune and Millett, »Designing Autonomous Teams and Services«, 38.

7 Nygard, »The Perils of Semantic Coupling«.

Bruchfläche: regulatorische Compliance

In stark regulierten Branchen wie dem Finanz- oder Gesundheitswesen setzen gesetzliche Vorschriften der Software häufig klare Rahmenbedingungen. Sie verlangen von Unternehmen oftmals spezielle Mechanismen für die Auditierung, die Dokumentation, das Testen und die Auslieferung von Software, die in den Geltungsbereich dieser Vorschriften fällt, sei es für Kreditkartenzahlungen, Transaktions-Reporting usw.

Einerseits ist es eine gute Idee, die Abweichungen zwischen diesen Prozessen in den verschiedenen Systemen zu minimieren. So sollten beispielsweise je nach Art des Systems und der Änderungen unterschiedliche Freigabe- und Auslieferungsprozesse bestehen. Wenn jedoch sichergestellt wird, dass solche Prozesse, einschließlich manueller Genehmigungen oder Maßnahmen, immer in der Delivery-Pipeline abgebildet werden und geeignete Zugriffskontrollen auf die Pipeline vorhanden sind, können Änderungen über alle Systeme hinweg nachvollzogen werden, während gleichzeitig die meisten Prüfungsanforderungen erfüllt bleiben.

Andererseits sollte die Einhaltung strenger Anforderungen nicht auf Bereiche des Systems übertragen werden, die nicht so kritisch sind. Die Abspaltung von Teilsystemen oder -abläufen innerhalb eines Monolithen, die in den Geltungsbereich von Regulierungen fallen, ist eine logische Bruchfläche.

So legt beispielsweise der Payment Card Industry Data Security Standard (PCI DSS) eine Reihe von Regeln für die Abfrage und Speicherung von Kreditkartendaten fest. Die Einhaltung des PCI DSS sollte einem speziellen Subsystem für die Verwaltung von Kartendaten obliegen, aber diese Anforderungen sollten nicht für einen ganzen Monolithen gelten, der zufällig Zahlungsfunktionen enthält. Die Aufteilung entlang der Bruchfläche für regulatorische Compliance vereinfacht die Auditierung und Konformität und verringert den Einzugsbereich der regulatorischen Aufsicht.

Schließlich spielt hier auch ein Aspekt der Teamzusammensetzung und -interaktion eine Rolle, insbesondere in größeren Organisationen. Wenn ein einziges, größeres Team für den Monolithen verantwortlich ist, ist es typisch, dass Mitarbeiter aus den Compliance- und/oder Rechtsabteilungen nur gelegentlich an Planungs- und Priorisierungsrunden teilnehmen, bei denen der Arbeitsumfang keine Vollzeitmitgliedschaft im Team für diese Interessenvertreterinnen rechtfertigt. Wenn das Teilsystem ausgegliedert wird, ist es plötzlich sinnvoller, ein kleineres, aber auf die Einhaltung der Vorschriften fokussiertes Team zu haben, dem auch Geschäftsverantwortliche aus dem Bereich Compliance und/oder Legal angehören.

Bruchfläche: Änderungskadenz

Eine weitere natürliche Bruchfläche ist dort, wo sich verschiedene Teile des Systems mit unterschiedlichen Frequenzen verändern müssen. Bei einem Monolithen bewegt sich jedes Teil mit der Geschwindigkeit des langsamsten Teils. Wenn neue

Reporting-Funktionen nur vierteljährlich benötigt und veröffentlicht werden, dann wird es wahrscheinlich sehr schwierig, wenn nicht sogar unmöglich, andere Arten von Funktionen häufiger auszuliefern, da die Codebase sich noch im Umbruch befindet und nicht produktionsreif ist. Änderungen werden in einen Topf geworfen, und die Geschwindigkeit der Auslieferung wird erheblich beeinträchtigt.

Die Abspaltung der Teile des Systems, die sich in der Regel unterschiedlich schnell verändern, ermöglicht eine beschleunigte Änderungsgeschwindigkeit. Die Geschäftsanforderungen bestimmen nun die Geschwindigkeit der Veränderungen, anstatt dass der Monolith eine feste Geschwindigkeit für alle vorgibt.

Bruchfläche: Teamstandort

Teams, die geografisch verteilt sind und in verschiedenen Zeitzonen arbeiten, sind natürlich nicht an einem gemeinsamen Standort. Aber auch Teams, deren Mitglieder im selben Bürogebäude auf verschiedenen Etagen oder in verschiedenen Räumlichkeiten arbeiten, können als geografisch getrennt betrachtet werden.

Innerhalb verteilter Teams ist die Kommunikation begrenzt, da sie explizit einen physischen oder virtuellen Raum und eine Zeit vereinbaren müssen, um standortübergreifend zu kommunizieren. Die verbleibende (ungeplante) teaminterne Kommunikation (die bis zu 80 % betragen kann) findet innerhalb der physischen Grenzen der einzelnen Teilbereiche des Teams statt.

Die Arbeit über verschiedene Zeitzonen hinweg verstärkt diese Kommunikationsverzögerungen und führt zu Engpässen, wenn manuelle Genehmigungen oder Code-Reviews von Personen in verschiedenen Zeitzonen benötigt werden, deren Arbeitszeiten sich kaum überschneiden. Heidi Helfand weist in ihrem Buch *Dynamic Reteaming* auf die Probleme mit unterschiedlichen Zeitzonen hin:

> Wenn Sie auf Remote-Mitarbeiter angewiesen sind, müssen Sie zusätzliche Arbeit leisten, um die Zusammenarbeit innerhalb des Teams und zwischen den Teams zu fördern, damit eine Gemeinschaft entsteht. Sie sollten versuchen, dieselbe Zeitzone und nicht unterschiedliche Zeitzonen zu haben. Andernfalls werden sich die Mitarbeiterinnen nicht miteinander treffen wollen, weil dies in ihre Freizeit zu Hause eingreift.[8]

Wir sind der Meinung, dass ein Team nur dann effizient kommunizieren kann, wenn es zwischen einer vollständigen Co-Location (alle Teammitglieder teilen sich denselben physischen Raum) und einem echten teamorientierten Ansatz (ausdrückliche Beschränkung der Kommunikation auf vereinbarte Kanäle wie Messaging und Collaboration-Apps, auf die alle Teammitglieder Zugriff haben und die sie regelmäßig nutzen) wählen kann. Wenn keine dieser Optionen realisierbar ist (vollständige Co-Location oder Remote-first), dann ist es besser, den Monolithen in separate Subsysteme für Teams an verschiedenen Standorten aufzuteilen. Auf

8 Helfand, »Dynamic Reteaming«, 203.

diese Weise kann ein Unternehmen das Conway'sche Gesetz nutzen und die Systemarchitektur an die Kommunikationsbeschränkungen im realen Leben anpassen.

Bruchfläche: Risiko

Innerhalb eines großen Monolithen können verschiedene Risikoprofile nebeneinander bestehen. Mehr Risiko einzugehen, bedeutet, eine höhere Ausfallwahrscheinlichkeit des Systems oder der Geschäftsergebnisse in Kauf zu nehmen, um die Änderungen schneller in die Hände der Kunden zu bekommen. Nebenbei bemerkt: Echte Continuous-Delivery-Funktionen mit einer lose gekoppelten Systemarchitektur (kein Monolith) verringern tatsächlich das Risiko, kleine Änderungen sehr häufig auszuliefern.

Es gibt verschiedene Arten von Risiken (die in der Regel mit der Bereitschaft des Unternehmens zu Veränderungen zusammenhängen), die auf Bruchflächen hindeuten können. Die Einhaltung gesetzlicher Vorschriften ist eine spezielle Art von Risiko, die wir bereits angesprochen haben. Andere Beispiele sind Marketing-getriebene Änderungen mit einem höheren Risikoprofil (mit Schwerpunkt auf der Kundenakquise) gegenüber Änderungen mit einem geringeren Risikoprofil bei umsatzsteigernden Funktionalitäten für Geschäftsabschlüsse (mit Schwerpunkt auf der Kundenbindung).

Auch die Anzahl der Nutzer kann das akzeptable Risiko bestimmen. Ein mehrstufiges SaaS-Produkt könnte beispielsweise Millionen von Nutzerinnen in der kostenlosen Stufe und nur ein paar hundert Kunden in den kostenpflichtigen Stufen haben. Änderungen an beliebten Funktionen in der kostenlosen Version könnten ein höheres Risikoprofil aufweisen, da jeder größere Fehler den Verlust von Millionen potenzieller zahlender Kunden bedeuten könnte. Änderungen an kostenpflichtigen Funktionen sind dagegen weniger risikoreich, wenn die Schnelligkeit und Individualität des Supports für diese wenigen Hundert Kunden gelegentliche Ausfälle wettmachen. Aus ähnlichen Gründen können interne Systeme in einem Unternehmen in der Regel ein höheres Risikoprofil aufweisen (was allerdings nicht bedeutet, dass sie nicht wie ein reguläres Produkt behandelt werden sollten, selbst wenn sie nur für den internen Gebrauch bestimmt sind).

Die Abtrennung von Teilsystemen mit deutlich unterschiedlichen Risikoprofilen ermöglicht die Anpassung der Technologie an die geschäftlichen Anforderungen oder die gesetzlichen Vorschriften. Außerdem kann jedes Teilsystem im Laufe der Zeit sein eigenes Risikoprofil entwickeln, indem es Praktiken wie Continuous Delivery anwendet, die eine höhere Änderungsgeschwindigkeit ermöglichen, ohne ein höheres Risiko einzugehen.

Bruchfläche: Performance-Isolierung

Bei bestimmten Systemtypen kann es von Vorteil sein, zwischen verschiedenen Performance-Stufen zu unterscheiden. Natürlich sollte die Performance eines jeden

Systems immer ein Thema sein, und sie sollte analysiert, getestet und nach Möglichkeit optimiert werden.

Teile von Anwendungen, die in großem Umfang Lastspitzen ausgesetzt sind (wie die jährliche Steuererklärung am letzten Tag), erfordern jedoch ein Maß an Skalierung und Failover-Optimierung, das für den Rest des Systems nicht erforderlich ist.

Die Aufteilung eines solchen Subsystems auf der Grundlage bestimmter Performance-Anforderungen trägt dazu bei, dass es unabhängig skalieren kann, was die Leistungsfähigkeit erhöht und die Kosten senkt. Eine Anwendung für Steuererklärungen könnte dann z. B. aus einem Subsystem für die Steuereinreichung und -validierung bestehen, das besonders leistungsstark ist und Millionen von Einreichungen in einem kurzen Zeitraum verarbeiten kann. Andere Teilsysteme wie die Steuersimulation, die Verarbeitung und die Auszahlung können mit einer weniger anspruchsvollen Systemleistung auskommen.

Bruchfläche: Technologie

Technologie ist oft (historisch gesehen) die einzige Art von Grenze, die bei der Aufteilung von Teams verwendet wird. Denken Sie daran, wie üblich es ist, getrennte Teams für Frontend, Backend, Datenhaltung usw. zu haben.

Diese gängigen Arten von technologiebasierten Aufteilungen führen jedoch in der Regel zu mehr Einschränkungen und stören den Arbeitsfluss mehr, als dass sie ihn verbessern. Das liegt daran, dass die getrennten Teams weniger autonom sind, da die Produktabhängigkeiten bestehen bleiben, während jedes Team weniger Einblick in die Arbeit als Ganzes hat, und die Kommunikationswege zwischen den Teams langsamer sind als innerhalb eines einzelnen Teams.

Es gibt Situationen, in denen die Ausgliederung eines Teilsystems auf der Grundlage der Technologie effektiv sein kann, insbesondere bei Systemen, die ältere oder weniger automatisierbare Technologie beinhalten. Der Fluss kann erheblich langsamer sein, wenn Änderungen an einer solchen älteren Technologie erforderlich sind, weil entweder mehr manuelle Tests durchgeführt werden müssen oder Schwierigkeiten bei der Implementierung von Änderungen aufgrund schlechter Dokumentation und fehlender offener, unterstützender Benutzer-Communitys zu erwarten sind (eine Selbstverständlichkeit für moderne Technologie-Stacks). Schließlich verhält sich das Ökosystem der Tools (IDEs, Build-Tools, Test-Tools usw.), das diese Technologie umgibt, in der Regel ganz anders als moderne Ökosysteme, was die kognitive Belastung der Teammitglieder erhöht, die zwischen diesen sehr unterschiedlichen Technologien wechseln müssen. Die Aufteilung der Zuständigkeiten im Team entlang von Technologien kann in diesen Fällen dazu beitragen, dass die Teams die Software effektiv beherrschen und weiterentwickeln können.

Bei der Entscheidung, ob eine Aufteilung entlang technologischer Bruchflächen erfolgen soll, sollten Sie zunächst untersuchen, ob alternative Ansätze dazu beitragen

könnten, das Tempo des Wandels bei älteren Technologien zu erhöhen, da auf diese Weise Einschränkungen beseitigt würden und das Unternehmen davon profitieren könnte (während die Aufteilung eines Monolithen entlang sinnvollerer Bruchflächen wie geschäftsorientierter Bounded Contexts möglich wäre). In seinem Buch *DevOps for the Modern Enterprise* erklärt Mirco Hering beispielsweise, wie man gute Codierungs- und Versionskontrollpraktiken im Umgang mit proprietären COTS-Produkten anwendet.[9]

Bruchfläche: User Personas

In dem Maße, wie Systeme wachsen und ihren Funktionsumfang erweitern, wächst auch ihr (interner oder externer) Kundenstamm und diversifiziert sich. Einige Benutzergruppen werden sich auf eine bestimmte Teilmenge von Funktionen stützen, um ihre Arbeit zu erledigen, während andere Gruppen eine andere Teilmenge benötigen. Bei Produkten mit Preisstaffelung ist die Untermenge von vornherein eingebaut (Kunden, die mehr bezahlen, haben Zugang zu mehr Funktionen als Kunden, die weniger oder gar nicht bezahlen). Bei anderen Systemen haben Admin-Benutzer Zugriff auf mehr Optionen und Steuerelemente als normale Benutzer. Oder ganz einfach: Erfahrene Benutzerinnen nutzen bestimmte Funktionen (wie Tastaturkürzel) häufiger als unerfahrene Benutzer. In solchen Situationen ist es also sinnvoll, Subsysteme für bestimmte User Personas abzuspalten.

Der Aufwand, der zur Beseitigung von Abhängigkeiten oder Kopplungen zwischen Funktionen erforderlich ist, wird durch eine stärkere Konzentration auf die Bedürfnisse der Kunden und die Erfahrungen mit dem System kompensiert, was zu einer höheren Kundenzufriedenheit führen und das Ergebnis des Unternehmens verbessern sollte. Tatsächlich kann eine solche Struktur auch die Geschwindigkeit und Qualität des Kunden-Supports verbessern – es wird einfacher, Probleme einem bestimmten Subsystem und Team zuzuordnen. Teams, die für Subsysteme verantwortlich sind, die auf die Personas des Unternehmens ausgerichtet sind, möchten vielleicht sicherstellen, dass sie jederzeit in der Lage sind, (Enterprise-)Support-Angelegenheiten so reibungslos wie möglich zu bearbeiten.

Natürliche Bruchflächen für Ihre spezifische Organisation oder Technologien

Manchmal lassen sich auch andere natürliche oder teamorientierte Bruchflächen für die Verortung von Aufgaben identifizieren. Der Lackmustest für die Anwendbarkeit einer Bruchfläche lautet: Unterstützt die resultierende Architektur autonomere Teams (weniger abhängige Teams) mit geringerer kognitiver Belastung (weniger unterschiedliche Verantwortlichkeiten)?

9 Hering, »DevOps for the Modern Enterprise«, 45.

Natürlich erfordert das Erzielen solcher Ergebnisse oft einige anfängliche Experimente und eine spätere Feinjustierung. Es ist unwahrscheinlich, dass man ein bestimmtes Endergebnis garantieren kann, ohne es vorher auszuprobieren. Eine einfache Heuristik, die Ihnen bei der Beurteilung Ihrer System- und Teamgrenzen helfen kann, ist die Frage: Können wir als Team dieses Teilsystem effektiv als Service nutzen oder bereitstellen? Wenn die Antwort ja lautet, dann ist das Teilsystem ein guter Kandidat für eine Ausgliederung und die Zuweisung an ein Team, das es verantworten und weiterentwickeln kann.

FALLSTUDIE: Das Finden von guten Softwaregrenzen bei Poppulo

Stephanie Sheehan, VP of Operations bei Poppulo

Damien Daly, Director of Engineering bei Poppulo

Poppulo ermöglicht es Unternehmen, die Wirkung ihrer Kommunikation über mehrere digitale Kanäle hinweg zu planen, auszurichten, zu veröffentlichen und zu messen – alles an einer einzigen Stelle. Zwischen 2012 und 2016 haben wir unsere Größe verdreifacht, Niederlassungen in den USA eröffnet und ein umfangreiches Kundenportfolio mit den größten Marken der Welt aufgebaut, darunter Nestlé, Experian, LinkedIn, Honda und Rolls-Royce. 2019 wird die Poppulo-Plattform von mehr als 15 Millionen Mitarbeitern in mehr als hundert Ländern genutzt. Um an diesen Punkt zu gelangen, mussten wir innerhalb von drei Jahren von einem einzigen Entwicklungsteam auf acht Produktteams, ein SRE-Team und ein Infrastruktur-Team expandieren.

Damals, im Jahr 2015, erwarteten wir ein erhebliches Wachstum unseres Kundenstamms und der Zahl unserer Engineering-Mitarbeiterinnen. Deshalb wollten wir sicherstellen, dass wir unseren Monolithen so aufteilen, dass die neuen Teams weitgehend unabhängig und eigenständig arbeiten konnten. Als wir mehr Entwickler einstellten, waren die Architektur und die Praktiken, die für ein einzelnes Team funktionierten, nicht mehr skalierbar. Wir stellten DevOps und Continuous-Delivery-Praktiken in den Mittelpunkt unserer Design-Entscheidungen und begannen mit dem Übergang von unserem bestehenden (erfolgreichen) monolithischen System zu einer Microservices-ähnlichen Architektur.

Wir begannen damit, uns stärker auf »das Team« als das Instrument zur Bewältigung unserer Arbeit zu konzentrieren. Zuvor hatten wir manchmal Engpässe bei Einzelpersonen, aber durch einen teamorientierten Ansatz und Praktiken wie Pairing (und später Arbeiten im Mob) konnten wir einen besseren Arbeitsfluss beobachten, da sich die Teammitglieder gegenseitig bei der Erledigung von Aufgaben halfen. Dann begannen wir, unseren Code zu instrumentalisieren und Telemetrie hinzuzufügen, damit wir sehen konnten, wie der Code in der Produktion tatsächlich funktioniert. Zusammen mit den Ende-zu-Ende-Deployment-Pipelines ermöglichten die verbesserte Protokollierung und die Metriken den Teams, den Code besser zu verstehen und Verantwortung zu übernehmen.

Die Produkte von Poppulo helfen Unternehmen bei der elektronischen Kommunikation mit einer großen Anzahl von Menschen. Unsere geschäftlichen Domänen konzentrieren sich daher auf Konzepte wie Menschen, Inhalte, Ereignisse, E-Mail, Mobilgeräte und Analytik. Wir wussten aus der Literatur und aus Konferenzvorträgen, wie wichtig es ist, den Delivery-Teams durch eine saubere Trennung der Domänen eine angemessene Autonomie zu geben. Wir haben daher einige Zeit damit verbracht, zu prüfen, wie unabhängig die einzelnen Domänen wirklich sind, und Szenarien auf Whiteboards durchgespielt, bevor wir die Software entlang dieser Domänengrenzen unterteilt haben. Wir wollten sicherstellen, dass sich das Conway'sche Gesetz nicht zu nachteilig auf uns auswirkt. Daher war eine effektive Trennung der Domänen für uns von entscheidender Bedeutung.

Wir legen großen Wert auf Kollaboration und Autonomie bei unserer Arbeit. Deshalb haben wir uns in »Matrix-Produktteams« organisiert: cross-funktionale Teams, die zusammensitzen und vollständig für einen Bereich des Produkts verantwortlich sind. Unsere Produktteams bestehen in der Regel aus vier Entwicklern, einer Produktionsleiterin, einem QA und einer UX/UI-Designerin. Unsere Teams sprechen direkt mit Kunden und Stakeholdern: Sie begleiten im Hintergrund Support-Anrufe, sie entwerfen, erstellen und messen die Auswirkungen ihrer Lösungen und sind für die Qualität der von ihnen gelieferten Lösungen verantwortlich.

Wir verwenden einige Techniken aus DDD, insbesondere Event Storming, um die Domänen in unserem Geschäftskontext zu verstehen und zu modellieren. Auf einer eher technischen Ebene verwenden wir Pact für Contract Testing Services und die teamübergreifende Kommunikation. Pact hat uns wirklich geholfen, einen klaren, definierten Ansatz für das Testen von Services zu verfolgen und die Erwartungen aller Teams an das Testen und die Interaktion mit anderen Teams festzulegen.

Die meisten unserer Delivery-Teams sind auf die Bounded Contexts der Geschäftsdomänen zugeschnitten, z. B. E-Mail, Kalender, Personen, Umfragen usw. Wir haben auch einige Teile des Systems, die sich nach gesetzlichen Vorgaben richten (insbesondere ISO 27001 für das Informationssicherheits-Management) und nach dem Bedarf an bereichsübergreifendem Reporting zur Nutzung von Funktionen. Diese Bereiche werden entweder von einem kleinen Spezialistenteam oder durch die Zusammenarbeit mehrerer Teams verwaltet.

Wir haben auch ein Team, das für eine einheitliche User Experience (UX) in allen Teilen der Software sorgt. Das UX-Team fungiert als interner Berater für alle Entwicklungsteams, damit diese schnell gute UX-Praktiken übernehmen können. Wir betreiben eine SRE-Abteilung, um das hohe Volumen an Anfragen zu bewältigen und die Betriebsfähigkeit zu verbessern.

Indem wir uns die Zeit genommen haben, unsere geschäftlichen Domänen zu verstehen und unsere monolithische Software so aufzuteilen, dass sie zu den Domänen passt, konnten wir unsere Entwicklungsabteilung seit 2015 von sechzehn auf siebzig Mitarbeiter vergrößern. Die Investition in Telemetrie und einen guten operativen Fokus hat den Teams geholfen, die Software zu verstehen, die sie entwickeln. Durch die Einführung von cross-funktionalen Produktteams zusammen mit dem, was wir »ausgerichtete Autonomie« nennen, haben wir eine gute Eigenverantwortung für Software-Services innerhalb der Teams erreicht, was uns wiederum einen schnellen Veränderungsfluss bei gleichzeitiger Minimierung der Ausfallzeiten ermöglicht.

Beispiel aus der realen Welt: Fertigungsindustrie

Als wir über Technologie als Bruchfläche sprachen, betonten wir, dass dies nur sparsam und vor allem bei älterer Technologie angewendet werden sollte, die sich deutlich anders verhält und anfühlt als moderne Software-Stacks. Unvermeidlich werden Sie Ausnahmen finden. Die Schwierigkeit besteht darin, zu verstehen, wann eine Ausnahme gültig ist und wann eine einfache Möglichkeit, schnell voranzukommen, letztlich die Effektivität einschränkt.

Lassen Sie uns zur Veranschaulichung ein Szenario betrachten, das wir bei einem ziemlich großen Kunden aus der Fertigungsindustrie vorgefunden haben. Dieses große Industrieunternehmen stellt physische Geräte für Verbraucherinnen her. Alle Geräte sind mit IoT-Funktionen ausgestattet, einschließlich der Steuerung über eine mobile App und Software-Updates aus der Ferne über die Cloud. Die Geräte werden sowohl über die Cloud (über geplante Aktivitäten) als auch durch interaktive Benutzersteuerung (über die mobile App) gesteuert. Alle Aktivitätsprotokolle und Produktdaten werden an die Cloud gesendet, wo sie verarbeitet, gefiltert und gespeichert werden.

Für ein Stream-aligned Team wäre es eine große Herausforderung, diese gesamte Ende-zu-Ende-User-Experience – Mobile-App, Cloud-Verarbeitung und Embedded-Software für das Gerät – angesichts des Umfangs und der kognitiven Belastung, die zu Beginn des Buchs hervorgehoben wurden, zu bewältigen. Für Ende-zu-Ende-Änderungen an drei sehr unterschiedlichen Technologie-Stacks (Embedded, Cloud und Mobile) ist eine Mischung von Fähigkeiten erforderlich, die nur schwer zu finden ist, und die damit verbundene kognitive Belastung und das Wechseln des Kontexts wären untragbar. Im besten Fall wären die Änderungen in technischer und architektonischer Hinsicht suboptimal, im schlimmsten Fall wären sie fehleranfällig, würden zu stetig wachsenden technischen Schulden führen und möglicherweise insgesamt eine schlechte User Experience für die Kunden bieten.

Wenn Sie die technischen Einschränkungen des Systems akzeptieren, könnten die Teams stattdessen entlang der natürlichen Technologiegrenzen organisiert werden (ein Embedded-Team, ein Cloud-Team und möglicherweise ein Mobile-Team). Die Kluft zwischen diesen Technologien (in Bezug auf die Fachkenntnisse und die Geschwindigkeit des Einsatzes) erzwingt ein unterschiedliches Veränderungstempo für jede dieser Technologien, was der Hauptgrund für separate Teams ist.

In diesem Fall gibt es zwei Hauptoptionen (siehe Abbildung 6-1): (1) Behandeln Sie die Cloud-Software als Plattform und die mobile und eingebettete IoT-Software als Clients/Konsumenten der Plattform. Das funktioniert gut, wenn sich die konsumierenden Anwendungen mindestens genauso schnell oder einfach ändern wie die Cloud-Plattform. (2) Behandeln Sie die eingebetteten IoT-Geräte als eine Plattform und machen Sie die Cloud und die mobilen Apps zu Clients/Konsumenten dieser

Plattform. Beide Modelle können funktionieren, aber in beiden Fällen muss das Team, das als Plattform fungiert, plattformähnliche Ansätze verfolgen.

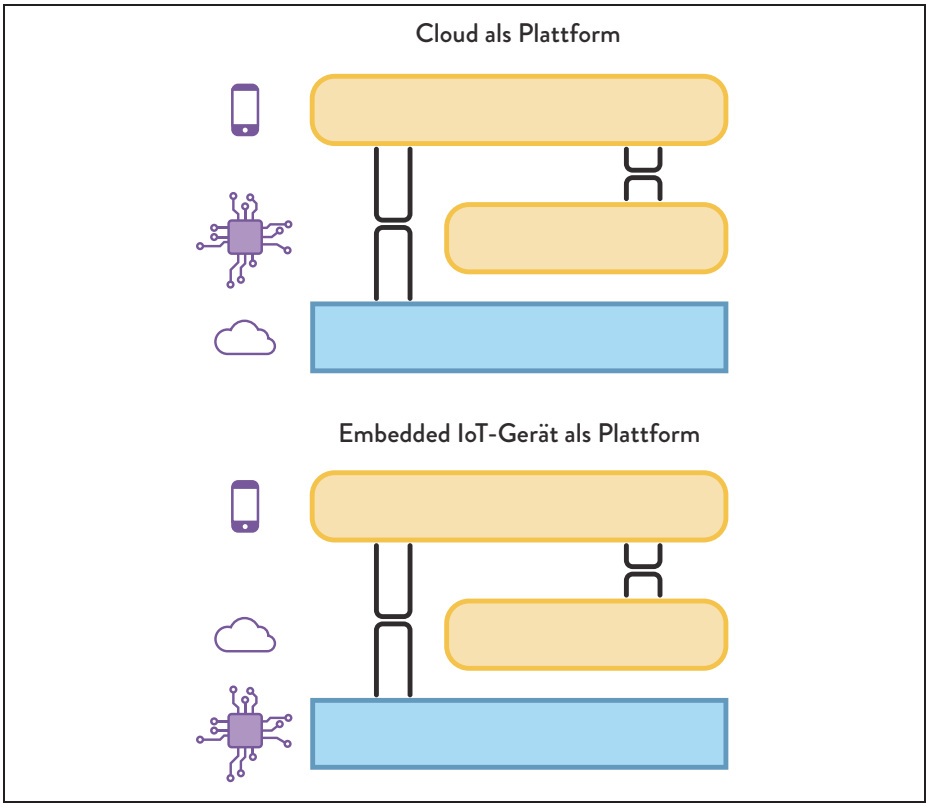

Abbildung 6-1: ***Mobile, Cloud und IoT: Technologie-Bruchflächen-Szenario***
Bei drei sehr unterschiedlichen Technologien (Mobile, Cloud und IoT) muss ein Unternehmen einen Ansatz für die Bruchflächen wählen, der in Abhängigkeit von folgenden Faktoren sinnvoll ist: der kognitiven Belastung und der Kadenz der Veränderungen in jedem Bereich.

Dieser Ansatz erfordert eine regelmäßige Abstimmung zwischen den Teams für Features, die zwei oder mehr Technologiebereiche betreffen. Möglicherweise müssen Änderungen an den APIs der Cloud-Plattform geplant werden, bevor eine neue Softwareversion für die Embedded-Geräte ausgerollt werden kann. Diese Abstimmung sollte jedoch auch dazu dienen, gemeinsame Arbeitsweisen zwischen den Teams zu etablieren (z. B. semantische Versionierung, Logging-Ansätze, API-first-Entwicklung).

Im Laufe der Zeit könnten diese gemeinsamen Praktiken und Kenntnisse eine künftige Umstrukturierung der Teamgrenzen ermöglichen, da sich das Tempo der Änderungen über die verschiedenen Technologien hinweg immer mehr angleichen wird.

Zusammenfassung: Wählen Sie Softwaregrenzen, die der kognitiven Belastung des jeweiligen Teams entsprechen

Bei der Optimierung für einen reibungslosen Arbeitsfluss sollten Stream-aligned Teams für eine einzige Domäne zuständig sein. Das ist eine Herausforderung, wenn Domänen in monolithischen Systemen stecken, die viele verschiedene Aufgabenbereiche beinhalten und größtenteils durch technologische Entscheidungen getrieben werden, die Funktionalitäten für mehrere Geschäftsbereiche bieten.

Wir müssen nach geeigneten Verfahren suchen, um das System aufzubrechen (Bruchflächen), die es den entstehenden Teilen ermöglichen, sich so unabhängig wie möglich zu entwickeln. Folglich werden die Teams, die diesen Teilbereichen zugewiesen sind, mehr Autonomie und Eigenverantwortung für sie haben.

Es ist ein guter Ansatz, die Grenzen von Subsystemen an (meist unabhängigen) Geschäftsbereichen auszurichten, und die Methodik des Domain-driven Designs unterstützt diesen Ansatz sehr gut. Aber wir müssen auch auf andere Bruchflächen achten, wie z. B. die Kadenz von Änderungen, Risiken, die Einhaltung regulatorischer Vorgaben usw. Oft ist eine Kombination von Bruchflächen erforderlich.

Nicht zuletzt müssen wir uns der verschiedenen Arten von Monolithen bewusst sein, die den Arbeitsfluss behindern und unnötige Abhängigkeiten zwischen den Teams verursachen. Während wir uns eine Systemarchitektur typischerweise als einen Monolithen vorstellen, gibt es auch andere, subtilere Arten der Kopplung, selbst wenn die Systemarchitektur bereits modular ist (z. B. gemeinsame Datenbanken, gekoppelte Builds und/oder Releases und mehr). Amy Phillips drückt es so aus: »Wenn Sie Microservices haben, aber warten und eine Kombination von ihnen Ende-zu-Ende testen, bevor Sie sie bereitstellen, dann haben Sie einen verteilten Monolithen«.[10]

Bei der Betrachtung von Subsystemgrenzen sollte das Hauptziel darin bestehen, Software-Bruchflächen zu finden, die sich an den Bounded Contexts der Geschäftsdomäne orientieren, da die meisten dieser Bounded Contexts den für die Organisation natürlichen Veränderungsströmen entsprechen. Das wiederum bedeutet, dass die Grenzen der Domäne auf Stream-aligned Teams ausgerichtet werden können, um den Fokus auf den Arbeitsfluss innerhalb der Organisation zu richten.

Bruchflächen können um spezifische Herausforderungen herum gewählt werden (z. B. Technologie, Regulierung, Performance, geografische Lage der Mitarbeiter, User Personas), um Übergaben zwischen Teams zu vermeiden und den Arbeitsfluss zu fördern.

In jedem Fall ist es wichtig, die Softwaresegmente in Teams aufzuteilen, sodass die Teams ihre Software effektiv verantworten und nachhaltig weiterentwickeln können.

10 Phillips, »Testing Observability«.