Der Microservice Trade-Off

Weighting the benefits and downsides of slicing your software

Ein Artikel von Arnold Franke

Seit 12 Jahren ist Arnold Franke für synyx als Software Ingenieur, Berater und Architekt in Projekten verschiedener Branchen unterwegs. Seine Motivation: Mit nachhaltigen Lösungen geringer Komplexität echten Mehrwert für Menschen zu schaffen.

Microservices sind eine tolle Sache, das weiß ja inzwischen jeder. Seit dem Hype darum hat sich diese Architekturform in der Softwareindustrie großflächig etabliert. Leider wird die Entscheidung Microservices zu verwenden aber häufig aus den falschen Gründen getroffen, was auf lange Sicht zu großen Problemen führt.

Bei dieser Entscheidung fehlt oft ein Bewusstsein für sowohl das tatsächliche Potential als auch die negativen Implikationen vieler kleiner, verteilter Services. Kriterien für die Bewertung dieses Trade-Offs sind schwer greifbar, verbreitete Missverständnisse erschweren Objektivität.

Wenn man sich die richtigen Fragen stellt, ist es dennoch möglich die Abwägung für oder gegen Softwareschnitte gut informiert zu treffen und damit schrittweise eine zweckdienliche, verteilte Systemarchitektur zu designen. Auf diese Reise begeben wir uns auf den folgenden Seiten.

Der Artikel reflektiert den Einzug der Microservices in die Enterprise-Architektur und räumt mit verbreiteten Missverständnissen darüber auf. Er stellt Potential und negative Implikationen von Softwareschnitten gegenüber, um ein Bewusstsein für den Trade-Off „Softwareschnitt oder nicht“ zu schaffen. Er gibt Architekten und Entwicklungsteams Methoden an die Hand, um diesen Trade-Off im Einzelfall informierter entscheiden zu können. Für strategische Entscheider liefert er einen Vorschlag für die evolutionäre Entwicklung einer Systemlandschaft als Alternative für die riskante Entscheidung, sich von vorne herein auf Microservices festzulegen.

Microservices – der Hype

Wer kennt das nicht in der Entwickler Community? Auf den Konferenzen, in Zeitschriften, Blogs, Streams und Social Media wird mal wieder eine Sau durchs Dorf getrieben. Es gibt ein neues, hippes Thema, das jeder unbedingt ausprobieren will und zu dem jeder seinen Senf dazu geben muss. Währenddessen fragt sich die breite Masse der Entwickler und Architekten: „Ist das jetzt nur ein flüchtiger Hype oder ein Zug, auf den man tatsächlich aufspringen sollte?“ Ein Beispiel für so einen Hype, von dem tatsächlich einiges hängen geblieben ist, ist das Thema „Microservices“, das in den Jahren 2015 und folgenden in aller Munde war und die Konferenzen dominierte.

Microservices – Potential und Verbreitung

Der Kern dieser Architekturform ist, dass man große Systeme nicht mehr als einen Monolithen implementiert und deployt sondern in viele, unabhängige Deploymenteinheiten („Services“) zerteilt, die unabhängig voneinander geliefert werden. Diese kommunizieren dann entkoppelt über das Netzwerk miteinander (Abb. 1). Die dadurch gewonnene Unabhängigkeit entfesselt neue Potentiale in Entwicklung und Betrieb des Systems:

- Unabhängiges Skalieren: Jeder Teil kann gezielt horizontal oder vertikal skaliert werden
- Es ist leichter, die einzelnen Teile in verschiedenen Teams zu entwickeln

- Man ist nicht an eine Technologie gebunden sondern kann für jeden Teil den passenden Stack wählen
- Das System ist resilienter. Wenn ein Teil ausfällt, dann kann der Rest des Systems weiter funktionieren
- Die einzelnen Teile des Systems sind leichter austauschbar

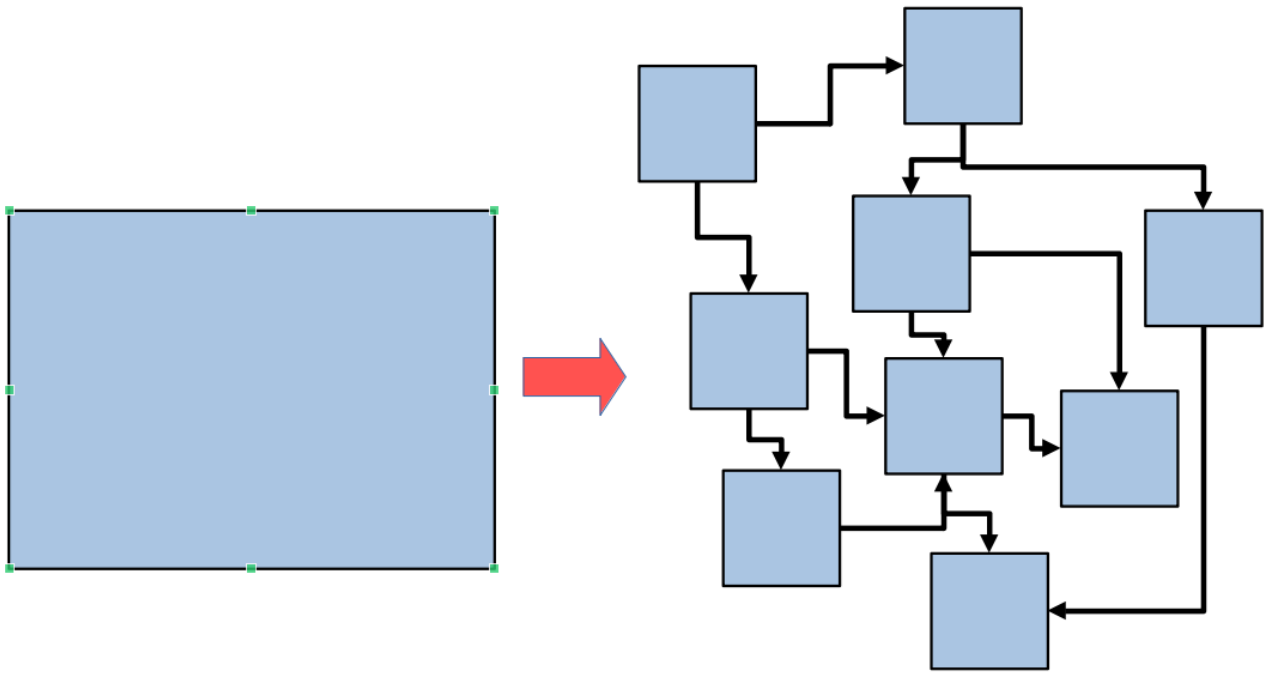

Abb. 1: Microservices statt Monolith: Große Systeme werden in viele Deploymenteinheiten zerteilt, die leichtgewichtig über das Netzwerk miteinander kommunizieren.

Weitere positive Eigenschaften wie saubere Modularisierung, Wartbarkeit & Erweiterbarkeit, durchsetzen von Architekturregeln, einfache Containerisierung und CI/CD wurden ebenfalls den Microservices zugeschrieben, sind aber keinesfalls dieser Architekturform vorbehalten. Diese Liste der potentiellen Vorteile wurde zur Zeit des Hypes rauf und runter gebetet, so dass der Eindruck entstand, dass es sich bei Microservices um eine echte Silver Bullet handelt.

In der Tat fanden Microservices in den folgenden Jahren eine breite Akzeptanz in der Software-Branche und haben sich als eine grundlegende Architekturform für große Systeme etabliert – in vielen Unternehmen mit Erfolg. Bereits wenige Jahre nach dem Hype konnten Umfragen der Branchengrößen eine großflächige Verbreitung belegen. In einer Nginx [1] Umfrage in 2020 gaben 60% der Unternehmen an, Microservices zu verwenden – die Leser des O'Reilly Verlags [2] meldeten im gleichen Jahr sogar eine

Verbreitung von 77%. Seither scheint das Niveau gleichbleibend hoch, worauf z.B. eine Gartner-Umfrage [3] von 2023 hinweist. Spannend wird es im O'Reilly Technology Trends Report von 2025, der auf einmal einen Rückgang beim Interesse an Microservices attestiert.

„Microservices Adoption":

2019: 40% (Nginx) [1]
2020: 60% (Nginx) [1]
2020: 77% (O'Reilly) [2]
2020: 84% (Kong) [4]
2021: 73% (Research Nester) [5]
2023: 74% (Gartner) [3]

Missverständnisse und Fehlannahmen

Es scheint, dass jetzt – 10 Jahre nach Beginn des Hypes – nicht nur der Hype selbst vorbei ist sondern auch dass bei manchen Entwicklungsteams die rosarote Microservice-Brille abgesetzt wird. Eine Ursache dafür ist, dass die Entscheidung mit Microservices zu arbeiten oft aus den falschen Gründen getroffen wird, was schwerwiegende Auswirkungen auf die langfristige Entwicklung der Projekte haben kann. Häufige Grundlage für solche falschen Entscheidungen sind eine Reihe von Missverständnissen und Fehlannahmen über Microservices, die sich aus der Zeit des Hypes noch hartnäckig in den Köpfen halten:

„Services müssen klein sein. Je kleiner desto besser."
Für viele klingt das logisch. Je mehr und je kleinere Services man hat, desto mehr profitiert man doch von den genannten Vorteilen? Die Fehlannahmen bei dieser Aussage sind, dass ein Schnitt in der Software „kostenlos" ist und dass man bei jedem Schnitt automatisch vom oben genannten Potential profitiert. Das ist leider nicht der Fall, weshalb das „micro" hier zum Selbstzweck verkommt.

„Viele kleine Services sind einfacher zu warten/betreiben als ein großer Service."
Es ist nicht zu bestreiten, dass eine kleine Softwareeinheit leichter handhabbar ist als eine große. Dass aber durch die Abhängigkeiten und Kommunikation vieler kleiner Einheiten die Komplexität steigt, wird hier außer Acht gelassen und führt oft zum gegenteiligen Effekt.

„Wenn wir gute ‚DevOps Automatismen' haben, dann gibt es keinen Overhead durch mehr Microservices."
Ohne Zweifel steckt in dieser Aussage ein Stück Wahrheit. Ein großer Teil des build/test/release/deploy Overhead lässt sich durch einen hohen Automatisierungsgrad einsparen. Doch großflächige Automatisierung bekommt man nicht geschenkt sondern muss sie sich erst mal hart erarbeiten. Und selbst die beste Automatisierung ist weder

fehler- noch wartungsfrei, so dass jedes weitere zu automatisierende Stück Software nach wie vor auch weiteren - wenn auch geringeren - Overhead bedeutet.

„Microservices sind der beste Weg, seine Software zu modularisieren.“
Modularisierung ist so alt wie das Handwerk der Softwareentwicklung. Es gibt unzählige Wege, Software zu modularisieren, wobei jeder seine eigenen Vor- und Nachteile hat. Keiner dieser Wege kann für sich beanspruchen, generell „der Beste“ zu sein.

„Microservices lösen alle unsere Performance Probleme.“
Diese optimistische Hoffnung bleibt leider all zu oft unerfüllt. Zwar ist gezielte Skalierung ein probater Weg, um Bottlenecks zu weiten – allerdings ist auch diese Skalierung nicht umsonst. Zudem ist die Ursache eines Performanceproblems häufig gar nicht durch Skalierung lösbar. Man kommt hier um Ursachenforschung und gezielte individuelle Maßnahmen nicht herum.

„Wenn wir fünf Teams haben, dann bauen wir am besten fünf Services.“ (Abb.2)
Schlussfolgerungen wie diese sind eine der vielen Ausprägungen von Conway’s Law [6].Wenn Softwarearchitektur auf der Grundlage von bestehenden Organisationsstrukturen entsteht, dann ist das Ergebnis selten optimal. Der bessere Weg ist, zuerst eine Architekturvision zu entwerfen und daraufhin zu überlegen, mit welcher Teamstruktur diese am Besten umsetzbar ist.

Für Architekten und Entscheider ist hier der Zeitpunkt zu hinterfragen: „Machen wir Microservices, weil wir mit jedem einzelnen Schnitt ein konkretes Problem adressieren und eine Verbesserung erreichen? Oder weil wir aus einem der vermeintlich guten Gründe von einem der Missverständnisse eine pauschale Entscheidung getroffen haben?“

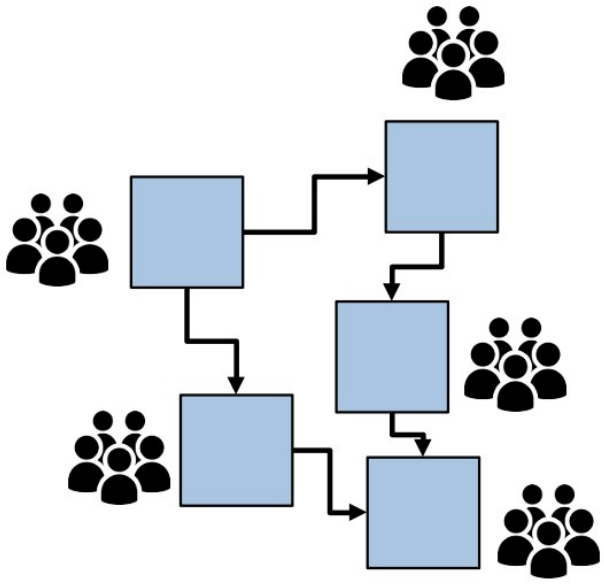

Abb. 2: Conway's Law: Ein Service für jedes bestehende Team. Bestehende Organisationsstruktur ist ein schlechter Treiber für Softwarearchitektur.

Die Kehrseite der Medaille

Zusätzlich zu den Missverständnissen ist der Glaube weit verbreitet, dass es keinen Nachteil hat, ein Stück Software in mehrere Teile zu zerschneiden. Leider ist es so, dass man nichts geschenkt bekommt, auch keine Softwareschnitte. Es ist sehr wichtig für alle Beteiligten – von Entwicklungsteam bis zu strategischen Entscheidern, sich der Implikationen eines Softwareschnitts und des damit einher gehenden Trade-Offs bewusst zu sein. Nur so hat man eine Chance, die richtige Entscheidung zu treffen.

Beginnen wir mit dem Netzwerk. Wenn man zwei Module, die miteinander kommunizieren, in zwei Deploymenteinheiten teilt, dann müssen diese auf einmal über das Netzwerk kommunizieren anstatt mit einem Methodenaufruf innerhalb der Anwendung. Das bedeutet man muss sich mit einem Protokoll der ISO/OSI Anwendungsschicht auseinandersetzen und es beherrschen. HTTP, MQTT, AMQP oder etwas anderes? Jedes davon hat seine eigenen Eigenheiten und Fallstricke. Zusätzlich muss auch neuer Code entstehen - Client und Server oder Publisher und Subscriber. Weiterer Overhead entsteht durch Serialisierung/Deserialisierung und der Abstraktion der Kommunikationsschicht. Auch das Design der Schnittstelle ist plötzlich schwieriger denn anstatt einer einfachen Methodensignatur muss man jetzt über API-Designparadigmen wie z.B. RESTful, GraphQL, Eventing Patterns nachdenken. Zu guter Letzt schlagen noch die „Fallacies of distributed computing“ [7] zu. Kommunikation über das Netzwerk ist unzuverlässig. Was passiert bei Timeouts? Braucht man Patterns wie Retries oder Circuit Breaker? Wie werden Fehler kommuniziert? Welche Delivery-Garantien wie Reihenfolge oder „at-least-once“ sollte es geben?
Alles in Allem macht die Kommunikation über das Netzwerk ein System deutlich komplizierter, aufwändiger, fehleranfälliger und schwerer zu verstehen. Entgegen der Intentionen vieler Microservice-Adopter wird alles dabei auch erst mal langsamer!

Weitere Herausforderungen warten im Build, Test & Deploy Umfeld. Mehr Services bedeuten hier eine Menge Duplikation. Alle Pipelines müssen dupliziert werden. Jede Deploymenteinheit braucht ein eigenes Testsystem plus Infrastruktur. Auch Anwendungsinfrastruktur wie Datenbanken müssen ggfs. mehrmals bereitgestellt werden. Komplexität der Betriebsumgebung wie z.B. Load Balancing, Instanzen, Ressourcenzuweisung müssen für jede Komponente bereitgestellt werden. Am Ende purzeln statt einem einzelnen Artefakt mehrere Artefakte heraus, deren Abhängigkeiten und Kompatibilität untereinander jedem Beteiligten bewusst sein müssen und wieder alles komplizierter machen.

Im Code der Services entsteht einiges an Duplikation durch geteilte Konfiguration, Duplikation, Glue Code und Abhängigkeiten. Cross Cutting Concerns der Systemlandschaft müssen in allen Teilen konsistent gehalten werden. Die Replikation und

Synchronisation von Daten von einer Stelle an die Andere macht eine ganz neue Komplexitätsdimension auf.

In der Masse kann man einige dieser Aspekte mit mehr Infrastruktur lösen. Infrastructure as Code, Kubernetes, Service Mesh, fully-fledged API Gateways & Co lindern einige der Probleme, fügen aber beträchtliche Komplexität zum Stack hinzu.

All das steigert die Cognitive Load des Teams, das die Software verantwortet, beträchtlich. Dieser Effekt ist nicht zu vernachlässigen und kann sich vernichtend auf Fokus und Performance des Teams niederschlagen. [8]

Die genannten Schattenseiten sollten nicht nur technische Entscheider sondern auch geschäftliche Entscheider zum Nachdenken bringen. Kostenfaktoren wie Entwicklungskosten, Betriebskosten und Wartungskosten werden davon ziemlich direkt in Mitleidenschaft gezogen. Selbst Business-Aspekte, die von Microservices profitieren sollen wie Resilienz und Time-to-Market verbessern sich dadurch nicht automatisch.

It’s a Trade-Off!

Wenn man sich diese ganzen Implikationen bewusst vor Augen führt, dann stellt sich die Frage, ob es der Softwareschnitt immer noch wert ist. Oder n Softwareschnitte für n Microservices. Die Antwort ist wie so oft „es kommt drauf an“. Es handelt sich hier um einen Trade-Off und zwar um einen der schwierigeren Sorte. Denn selten ist es möglich vorab an messbaren Kriterien festzumachen, wann sich ein Schnitt in der Software lohnt. Dennoch gibt es Methoden, sich einer objektiv sinnvollen Entscheidung zu nähern. Es handelt sich dabei weniger um feste Regeln als eher um Fragen, die man sich stellen kann, um Indizien in die eine oder andere Richtung zu sammeln.

Domain Driven Design

Das Ausrichten der Softwarearchitektur an der fachlichen Domäne gilt heute als einer der wichtigsten Erfolgsfaktoren für große und kleine Projekte. Das Domain Driven Design (DDD) [9] gibt zahlreiche Hinweise, an welchen Stellen es sich lohnen kann, einen Softwareschnitt einzuführen.

Die ersten Beispiele dafür findet man im Workshop-Format „Event Storming“ [10]. In einem Event Storming sammelt man die Domain Events, die sich in der abzubildenden Domäne ereignen, bringt diese in eine grobe zeitliche Struktur und reichert weitere Informationen an. Das Ergebnis kann z.B. so aussehen wie in Abb. 3.

Abb. 3: Event Storming: Bewährtes Workshop-Format zur Ergründung der eigenen Business-Domäne

Die ersten Hinweise ergeben sich aus den Clustern von Events (Abb. 4), die offensichtlich eng gekoppelt sind und einen so genannten „Bounded Context“ ergeben. Bounded Contexts sind hervorragende Kandidaten für Module in der späteren Software und je nach Stärke der Kapselung auch für eigene Deploymenteinheiten. Man darf bloß nicht die Fehlannahme treffen, dass jeder Bounded Context genau ein Service sein muss. Oft ist es sinnvoll, mehrere Bounded Contexts innerhalb einer Deploymenteinheit zu modularisieren. In manchen Fällen kann es auch Gründe geben, einen Bounded Context auf mehrere Deploymenteinheiten aufzuteilen.

Abb. 4: DDD Bounded Contexts – gekapselte Teile der Domäne sind gute Kandidaten für Software Module

Ein weiteres Indiz für potentielle Schnitte sind so genannte Pivotal Events (Abb. 5). Sie markieren ein Schlüsselereignis, das den Übergang von einem großen Prozess in einen anderen großen Prozess markiert - wie der Klick auf „Bestellung abschicken“ in einem Webshop, der den Übergang von „einkaufen“ des Käufers zu „liefern“ des Händlers bedeutet. Pivotal Events sind erstklassige Kandidaten für Softwareschnitte, da man solche Prozesse explizit voneinander entkoppeln möchte.

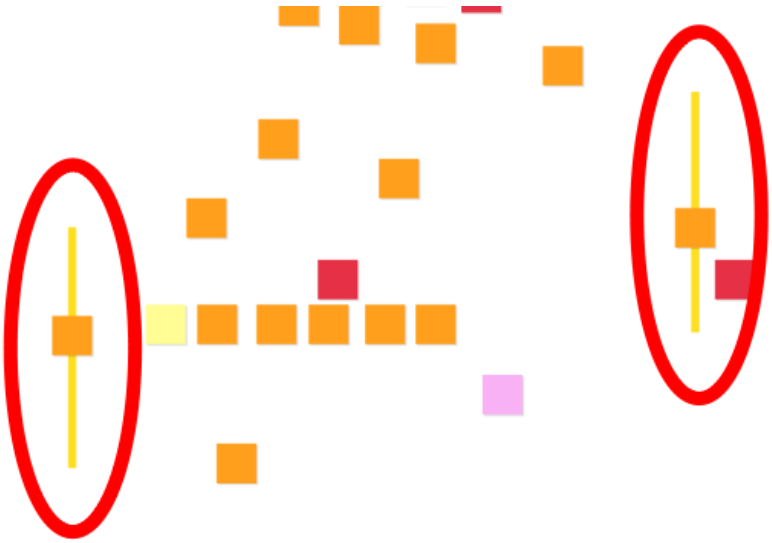

Abb. 5: Pivotal Events: Übergänge zwischen Prozessteilen sind natürliche Sollbruchstellen für Softwaresysteme

Es gibt hier auch Hinweise darauf, wo man die Software lieber nicht schneiden sollte. So genannte „Swim Lanes“ (Abb. 6) sind feste Prozessteile aus mehreren Events, die zusammengehörig implementiert werden sollten.

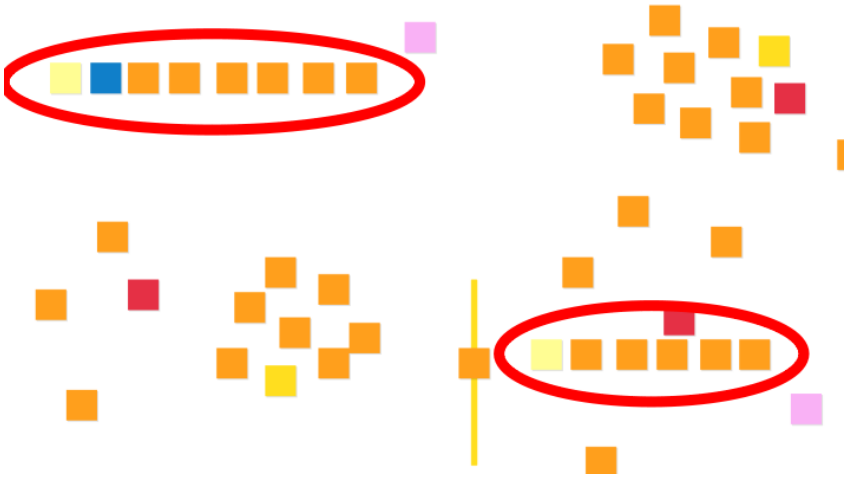

Abb. 6: Swim Lanes: Zusammenhängende Prozessteile sollten nicht getrennt implementiert werden

Im „strategic“ Teil des DDD wird versucht zu ermitteln, welche Teile der fachlichen Domäne strategisch wertvoller sind als andere. Man unterscheidet zwischen „Core Domain“, „Supporting Domain“ und „Generic Domain“ – je nach Komplexität und potentiellem Wettbewerbsvorteil der entsprechenden Subdomänen (Abb. 7). Je nach strategischer Relevanz behandelt man die Umsetzung der einzelnen Subdomänen sehr unterschiedlich. Die Umsetzung einer Core Domain beansprucht viel mehr Energie, Wissen und Manpower und hat deutlich höhere Anforderungen, während man die Lösung einer Generic Domain minimalistisch hält oder vielleicht sogar von der Stange kauft. Um den Anforderungen dieses Unterschieds gerecht zu werden, kann es sich lohnen, die unterschiedlichen

Domänen nach strategischer Bedeutung auf unterschiedliche Deploymenteinheiten zu verteilen.

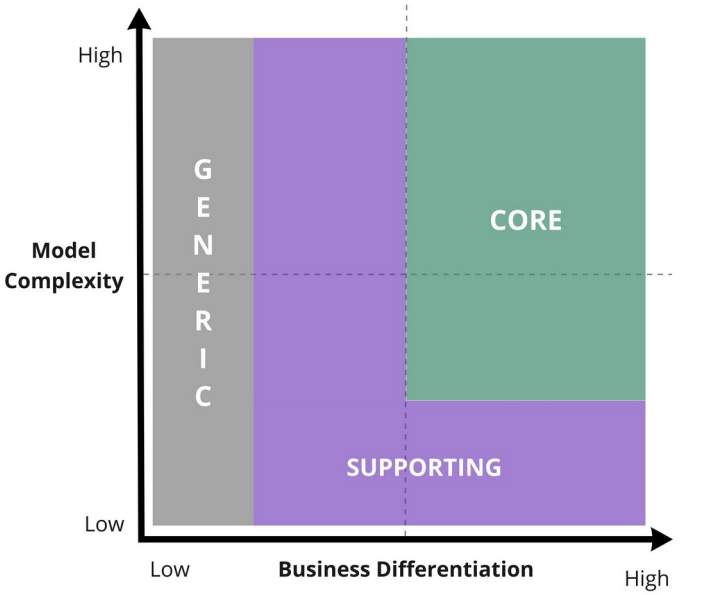

Abb. 7: Core Domain Chart: Die strategische Einstufung der Subdomänen wirkt sich auf die Softwarearchitektur aus.

Fracture Planes

Auch abseits der fachlichen Domäne gibt es eine lange Liste an Aspekten, für die es sich ggfs. lohnen kann, sie in einer Deploymenteinheit zu isolieren. Einige davon stammen aus dem Buch Team Topologies [11], das für sie den Begriff „Fracture Planes“ geprägt hat. Also eine Art „Sollbruchstelle“ in der Software, an der sich auf natürliche Weise Schnitte ergeben können. Die Bewertung dieser Aspekte gibt Hinweise darauf, welchen Vorteil man sich durch einen Schnitt an dieser Stelle erarbeitet. Da es sehr viele davon gibt, werden hier nur die relevantesten aufgezählt.

„Fracture Planes“

- Performance Isolation
- Criticality
- Regulatory Compliance
- User Personas
- Technology (existing or new)
- Replacability

- Longevity
- Change Cadence
- Support Frequency
- Security
- … u.v.m.

Fracture Plane „Performance Isolation“: Dieser Aspekt zahlt direkt auf ein der Kernversprechen der Microservice Architektur ein, die Performance Optimierung. Wenn es einen Teil des Systems gibt, der im Vergleich zum Rest besonders schnell sein muss oder besonders viel Last verarbeiten muss, dann kann es vorteilhaft sein, diesen in einer eigenen Deploymenteinheit zu isolieren. Dadurch kann man gezielt optimierende Maßnahmen durchführen und ggfs. horizontal oder auch vertikal skalieren, ohne dass der Rest des Systems davon betroffen ist.

Fracture Plane „Criticality“: Häufig gibt es in Softwaresystemen Teile, die so kritisch für die Domäne sind, dass sie auf gar keinen Fall ausfallen dürfen. In diesem Fall kann man das Resilienzversprechen der Microservice Architektur einlösen und die kritischen Teile in einem eigenen Stück Software entkoppeln. Das ermöglicht es, diese Teile durch Redundanzen und Resilienzmechanismen so zu stärken, dass sie ausfallsicherer werden als der Rest des Systems.

Fracture Plane „Regulatory Compliance“: Manchmal unterliegen Teile der Domäne gesetzlichen oder organisatorischen Regularien, deren Sicherstellung erheblichen Aufwand bedeutet. Durch die Konzentration dieser Teile in einem eigenen Service kann man die Compliance Mechanismen auf diesen Teil beschränken und spart sich den Aufwand in den anderen Teilen des Systems.

Fracture Plane „User Personas“: Ein großes System hat oft unterschiedliche Gruppen an Nutzern mit unterschiedlichen Anforderungen. Klassische Beispiele sind User&Admins, Autoren&Leser, Eltern&Kinder. Es kann von Vorteil sein, spezifischen Nutzergruppen eigene Deploymenteinheiten zur Verfügung zu stellen, um diese speziell auf die Anforderungen der Nutzergruppe zuschneiden zu können.

Fracture Plane „Technology“: Oft ist existierende alte Technologie ein unangenehmer Constraint in der Architektur eines Systems. Wenn man die Möglichkeit hat, die Berührungspunkte mit einer unschönen, alten API in einem Stück Software zu isolieren, dann kann man damit verhindern, dass deren Probleme in das neue System rüber schwappen. In anderen Fällen setzt man bewusst unterschiedliche Technologien für die Umsetzung unterschiedlicher Services ein, wenn die Anforderungen so speziell sind, dass sie durch einen spezialisierten Stack besser erfüllt werden.

Fracture Plane „Replacability“: Wenn es die Anforderung gibt, dass ein Teil der Software leicht austauschbar sein muss, kann ein Schnitt in der Software mit standardisierter API das ermöglichen. Auch Kurzlebigkeit kann eine Rolle spielen. Wenn von vorne herein klar ist, dass ein Aspekt der Software in absehbarer Zukunft wieder weggeworfen wird, dann wird das durch eine abgetrennte Deploymenteinheit vereinfacht.

Den Trade-Off bewerten

Wenn man sich das Gelesene vor Augen führt, dann kann man die Einzelentscheidung „Schnitt oder nicht“ als Architekt oder umsetzendes Entwicklungsteam jetzt viel qualifizierter treffen. Die Missverständnisse über Bord werfen und sich darauf konzentrieren, was tatsächlich relevant ist. Man kann sich bewusst machen, welche Nachteile ein potentieller Schnitt unweigerlich impliziert. Dem gegenüber wird gesammelt, welche potentiellen Vorteile eines Schnitts durch einzelne Fracture Planes oder die geschickte Ausrichtung an der fachlichen Domäne ausgeschöpft werden. Nur wenn diese Vorteile den Aufwand und die Komplexität eines Schnitts aufwiegen sollte man sich dafür entscheiden.

Evolution einer Systemlandschaft

Bisher wurde hauptsächlich der Trade-Off eines einzelnen Softwareschnitts beleuchtet. In einem großen System steht man allerdings nicht nur einmal vor dieser Entscheidung. Wie wendet man das Gelernte jetzt an, um eine Architektur für das Gesamtsystem zu entwickeln? Es ist jetzt klar, dass es große Probleme mit sich bringen kann, von vorne herein alles in Microservices zu zerteilen. Ebenso verzichtet man auf eine Menge Potential, wenn man sich langfristig auf einen Monolithen festlegt. Oft fühlen sich strategische Entscheider im Management verpflichtet, eine solche Entscheidung frühzeitig und pauschal für ihre Teamlandschaft zu treffen, ohne den notwendigen Kontext zu haben und ohne die Implikationen zu kennen.

Zitate aus Strategic Monoliths and Microservices [12]

„Choosing Microservices first is dangerous. Choosing Monoliths for the long term is also dangerous“

„Modules first, deployment last“

Ein häufig bewährter Mittelweg in dieser Situation ist, sich auf eine schrittweise Evolution der Systemarchitektur einzulassen. Mit Fokus darauf, Mehrwert zu liefern anstatt sich an der Architektur zu verkünsteln. Mit den Prinzipien der Agile Architecture eine Entscheidung nach der anderen zu treffen.

Es empfiehlt sich, mit einer Deploymenteinheit zu beginnen und darin erst mal Features zu implementieren, die direkt einen Mehrwert liefern. Um von vorne herein in Modulen zu denken, bietet sich die modulithische Architektur an. Sie erreicht mit innerhalb des Codes gekapselten Modulen bereits die Vorteile von fachlicher Kapselung, flexibler Erweiterbarkeit und leichter Testbarkeit. Auch ein Modulith kann containerisiert werden und von CI/CD profitieren – für die Einhaltung von Architekturregeln können Tools wie Archunit [13] oder Spring Modulith [14] sorgen. Mehr Details, Tooling und Codebeispiele sind in der Artikelserie „Architekturpatterns in Modulithen“ [15] zu finden.

Während der Weiterentwicklung wird man immer wieder an Stellen kommen, die Potential für die Abtrennung einer Deploymenteinheit haben. Mit den oben genannten Methoden wird der Trade-Off dafür bewertet und danach die Entscheidung für oder gegen den Schnitt getroffen. Im Zweifelsfall verschiebt man diese schwierig rückgängig zu machende Entscheidung auf einen späteren Zeitpunkt, zu dem man mehr über das System weiß. In einem sauberen Modulithen sollte es auch später problemlos möglich sein, ein Modul in eine neue Deploymenteinheit auszulagern. So entwickelt sich auf natürliche Weise nach und nach eine Landschaft aus Komponenten mit bewusst isolierten Aspekten und Schnitten, die klare Zwecke haben. Die Menge des Overheads durch viele Komponenten und Netzwerkkommunikation bleibt so minimal wie möglich aber so groß wie nötig.

Bei diesem fortwährenden Prozess ist es unabdingbar, jede dieser Entscheidungen gründlich zu dokumentieren. Die Abwägung jedes Trade-Offs und der Zweck jedes Schnitts (oder nicht-Schnitts) sollte jederzeit nachvollziehbar sein. In der agilen Architektur ist es gang und gäbe, vergangene Entscheidungen bei Bedarf neu zu bewerten, wobei eine solche Dokumentation (beispielsweise in ADRs [16]) unschätzbaren Wert hat.

Sind Microservices jetzt toll oder nicht?

Am Ende dieses Artikels sollte nun klar sein, dass diese Frage zu komplex ist, um sie pauschal zu beantworten. Das Potential von Microservices auszuschöpfen ist nicht einfach und blind hinein zu tauchen birgt viele Probleme. Mit Hilfe von DDD und Fracture Planes sowie einem agilen Architekturansatz gelingt es dennoch, von den richtigen Schnitten an den richtigen Stellen zu profitieren, ohne die Komplexität explodieren zu lassen.

Der aufmerksame Leser wird bemerkt haben, dass der Artikel kein Plädoyer für oder gegen Microservices ist. Eher ein Plädoyer gegen blinde Entscheidungen und für bewusstes und zweckorientiertes Architekturdesign. Für Nachdenken über tatsächliche Probleme und Erarbeitung gezielter Lösungen. Für Agilität und nachhaltiges Arbeiten im Sinne von langlebiger Software. So erreicht man das, worauf es wirklich ankommt: Software, die von Anfang bis Ende kontinuierlich Mehrwert liefert und langfristig flexibel und erfolgreich bleibt.

Literaturverzeichnis

[1]: Nginx, The State of Modern App Delivery, 2020, https://www.f5.com/pdf/infographic/NGINX-survey-infographic_2020.pdf
[2]: Mike Loukides and Steve Swoyer, Microservice Adoption in 2020, 2020, https://www.oreilly.com/radar/microservices-adoption-in-2020/
[3]: Gartner Peer Community, Microservices Architecture: Have Engineering Organizations Found Success?, 2023, https://www.gartner.com/peer-community/oneminuteinsights/omi-microservices-architecture-have-engineering-organizations-found-success-u6b
[4]: Kong Inc., Kong 2020 Digital Innovation Benchmark, , https://kong-mwe-web-assets.s3-accelerate.amazonaws.com/wp-content/uploads/2019/12/Digital-Innovation-Benchmark-2020-Report.pdf
[5]: Preeti Wani, Microservices Orchestration Market, 2025, https://www.researchnester.com/reports/microservices-orchestration-market/6991
[6]: Melvin Conway, Conway's Law, 1967, https://www.melconway.com/Home/Conways_Law.html
[7]: Peter Deutsch, The Eight Fallacies of Distributed Computing, 1994, https://nighthacks.com/jag/res/Fallacies.html
[8]: Arnold Franke, Team Cognitive Load, Objektspektrum 01/2022, https://media.synyx.de/publications/Arnold_Franke_Team_Cognitive_Load.pdf
[9]: DDD Crew, Welcome to DDD, 2022, https://github.com/ddd-crew/welcome-to-ddd
[10]: Alberto Brandolini, Event Storming, 2021
[11]: Matthew Skelton, Manuel Pais, Team Topologies, 2019
[12]: Vaughn Vernon, Tomasz Jaskula, Strategic Monoliths and Microservices, 2021
[13]: TNG Technology Consulting, ArchUnit, 2025, https://www.archunit.org/
[14]: Oliver Drotbohm, Spring Modulith, 2025, https://spring.io/projects/spring-modulith
[15]: Arnold Franke, Architekturpatterns in Modulithen, Java Magazin 12/20+01/21+02/21, https://media.synyx.de/publications/Architekturpatterns-in-Modulithen-1.pdf, https://media.synyx.de/publications/Architekturpatterns-in-Modulithen-2.pdf, https://media.synyx.de/publications/Architekturpatterns-in-Modulithen-3.pdf
[16]: Oliver Fischer, Gut dokumentiert: Architecture Decision Records, 2020, https://www.heise.de/hintergrund/Gut-dokumentiert-Architecture-Decision-Records-4664988.html