Sonderdruck aus OBJEKTspektrum 05/2018

Skalierbare datenflussbasierte Architektur

Alles im Fluss

Skalierbare Software sollte nicht nur elastisch sein, sondern auch mit deren Funktionsumfang und ihrer Entwicklung skalieren. Dies kann bereits im Entwurf der Architektur berücksichtigt werden – aber wie? Damit Entwickler sich der benötigten Funktionen bewusst werden, können diese auf bereits bekannte Konzepte zurückgreifen, wie Datenflussdiagramme. Dieser Beitrag zeigt anhand eines Fallbeispiels, wie diese genutzt werden können, um eine skalierbare Architektur zu entwerfen.

Skalierung umfasst weitaus mehr als nur die Elastizität eines Systems. Ein System ist dann elastisch, wenn es angemessen auf sich verändernde Lasten reagieren kann, das heißt, weiter stabil läuft. Doch was ist mit anderen Aspekten der Skalierbarkeit?

Skalierbarkeit in Softwaresystemen kann auch dazu dienen, einen höheren Grad an Flexibilität und Dynamik zu erreichen, um verschiedene Funktionen überhaupt erst zu ermöglichen. Dieser Aspekt ist insbesondere in der heutigen Zeit von Bedeutung, in der agile Entwicklungsmethoden immer verbreiteter sind. Durch den Einsatz kurzer Entwicklungszyklen und Continuous Deployment können Wünsche von Kunden schnell in den laufenden Betrieb integriert werden. Voraussetzung hierfür ist jedoch, dass dies auch vom Softwaresystem unterstützt wird. Ein System, welches nicht mit den sich entwickelnden Anforderungen skaliert, wird hier Grenzen aufzeigen, die nicht überwunden werden können.

Ein weiterer Faktor, der Einfluss auf die Effizienz der Entwicklung von Software nimmt, ist die Struktur der Entwicklerteams. Dieser Aspekt ist nur scheinbar nebensächlich, bietet aber dennoch großes Potenzial für Verbesserungen. Insbesondere Teams, die an verteilten Standorten tätig sind, bemerken einen signifikanten Overhead, der durch notwendige Absprachen zwischen den Standorten entsteht. Auch hier kann in der Entwurfsphase darauf Rücksicht genommen werden, indem das zu entwickelnde System mit der Teamstruktur skaliert. Durch geeignete Systemschnitte und eine geeignete Zuweisung an entsprechende Entwicklerteams kann der Kommunikations-Overhead reduziert werden.

Fallbeispiel – generische Forschungsdateninfrastruktur

In der Entwurfsphase unseres Projektes, welches wir hier als Fallbeispiel betrachten, fanden wir uns mit den oben genannten Problemstellungen konfrontiert. Bei *GeRDI* (Generic Research Data Infrastructure) handelt es sich um ein Projekt, welches eine generische Infrastruktur für das Forschungsdatenmanagement bereitstellen soll [Gru17].

Ziel ist es, Forschern eine integrative Infrastruktur bereitzustellen, die Forschungsdaten und Verarbeitungsmöglichkeiten multidisziplinär vereint. Somit sollen Quellen von Forschungsdaten direkt mit Anbietern von Analysediensten verknüpft werden können, um den Forschern die Arbeit in einer integrierten Umgebung zu ermöglichen. Diese soll generisch sein, das heißt, es soll nicht ein spezielles Fachgebiet allein abgedeckt werden, sondern es soll diversen Fachgebieten gleichermaßen ermöglichen, diese Infrastruktur gemeinsam zu nutzen. Daraus ergeben sich jedoch Herausforderungen, wie beispielsweise unbekannte Anforderungen, die zur Entwurfsphase noch nicht erfasst werden können. Des Weiteren wird das Projekt deutschlandweit an fünf Standorten entwickelt, was zusätzliche Herausforderungen birgt.

Die im Folgenden vorgestellten Ansätze wurden in diesem Anwendungsfall umgesetzt [Tav18] und werden auch hier im Einzelnen diskutiert.

Skalierbarer Funktionsumfang durch Datenflussorientierung

In einer perfekten Welt sind alle Anforderungen an ein System zu Beginn der Entwicklung bekannt. Dieser Ansatz spiegelt sich im klassischen Wasserfallmodell wider. Jedoch ist in der Realität etwas Anderes zu beobachten: Viele Anforderungen werden erst im Laufe der Entwicklung klar oder werden initial missverstanden.

Dies führt dazu, dass verstärkt agile Entwicklungsmethoden statt des Wasserfallmodells genutzt werden, wo kurze Entwicklungszyklen es erlauben, schnell auf Änderungen eingehen zu können.
Um dies auf Architekturebene zu ergänzen, bietet es sich auch bei agiler Entwicklung an, zu Beginn einen Architekturstil zu wählen, welcher „Leitplanken“ für die weitere Entwicklung definiert und mit den geforderten Funktionen skaliert [Has18]. Durch einen geeigneten Entwurf können neue oder geänderte Funktionen in das System integriert werden, ohne Einfluss auf andere Komponenten zu haben. Dies reduziert den Aufwand und somit auch die Time-to-Market für solche Änderungen.
In GeRDI stellte sich eine datenflussbasierte Architektur als geeignete „Leitplanke“ heraus. In der Entwurfsphase, und auch jetzt in der Entwicklung, konnten wir noch nicht alle Anforderungen an das System erfassen. Zwar soll GeRDI eine homogene Anwendergruppe bedienen, doch hat selbst diese homogene Gruppe verschiedene Anforderungen. Forscher aus unterschiedlichen Disziplinen führen unterschiedliche Analysen auf den gewählten Datensätzen aus. Darüber hinaus können weitere Anforderungen in der Zukunft hinzukommen.
Daher stellte sich für uns die Frage, wie wir mit diesen Herausforderungen umgehen sollen. Ein naheliegender Ansatz war es, die Datenflüsse in unserem System zu betrachten, da GeRDI inhärent datenflussbasiert ist. Bestärkend hierzu kommen Beobachtungen Dritter, wie jene des UK Data Archive, die bereits die Arbeit mit Forschungsdaten als eine Art Datenfluss identifizieren. **Abbildung 1** veranschaulicht die verschiedenen Phasen, die Forschungsdaten durchlaufen.
Im Hinblick auf den Architekturentwurf haben wir zunächst technische Funktionen identifiziert. Diese beschreiben auf einer abstrakten Ebene, welche Schritte Forschungsdaten in unserer geplanten Infrastruktur durchlaufen müssen. Das Ergebnis ist in **Abbildung 2** als Funktionskette dargestellt. Aus technischer Sicht durchlaufen Forschungsdaten die gezeigten Funktionen von links nach rechts, möglicherweise auch in mehreren Iterationen.
Um den Entwurf eines Monolithen zu vermeiden, wird im folgenden Schritt die Kette in distinkte Funktionen geschnitten, um diese in ein Datenflussdiagramm (**siehe Abbildung 3**) zu übertragen. Ein Datenflussdiagramm erleichtert es uns, die Kommunikationswege der verschiedenen Datentypen zu identifizieren und diese in den Entwurf der Schnittstellen einfließen zu lassen.
Wir sehen in unserem Beispiel, dass sowohl Daten als auch Metadaten übertragen werden. So kann in der Implementierung der Funktion *Search* auf Schnittstellen verzichtet werden, die Forschungsdaten annehmen, und stattdessen können wir uns auf Schnittstellen fokussieren, die deren Metadaten akzeptieren. Dies vereinfacht und optimiert die Implementierung der Schnittstellen, da an dieser Stelle klar wird, welche Daten außer Acht gelassen werden können. Mit dem Datenflussdiagramm sind nun mehrere Punkte klar:

- Welche Logiken benötigt werden,
- welche Persistenzmöglichkeiten benötigt werden und
- wie die Schnittstellen definiert werden.

Dies kann für jede Funktion einzeln gekapselt werden. Es bleibt also nur noch die Frage, wie dies in einem Architekturentwurf übertragen werden kann.
Betrachten wir die Anforderungen, so bietet sich das Muster der Self-Contained Systems (SCS, **siehe Kasten 1**) an. SCS kapseln einzelne Funktionen der Kerndo-

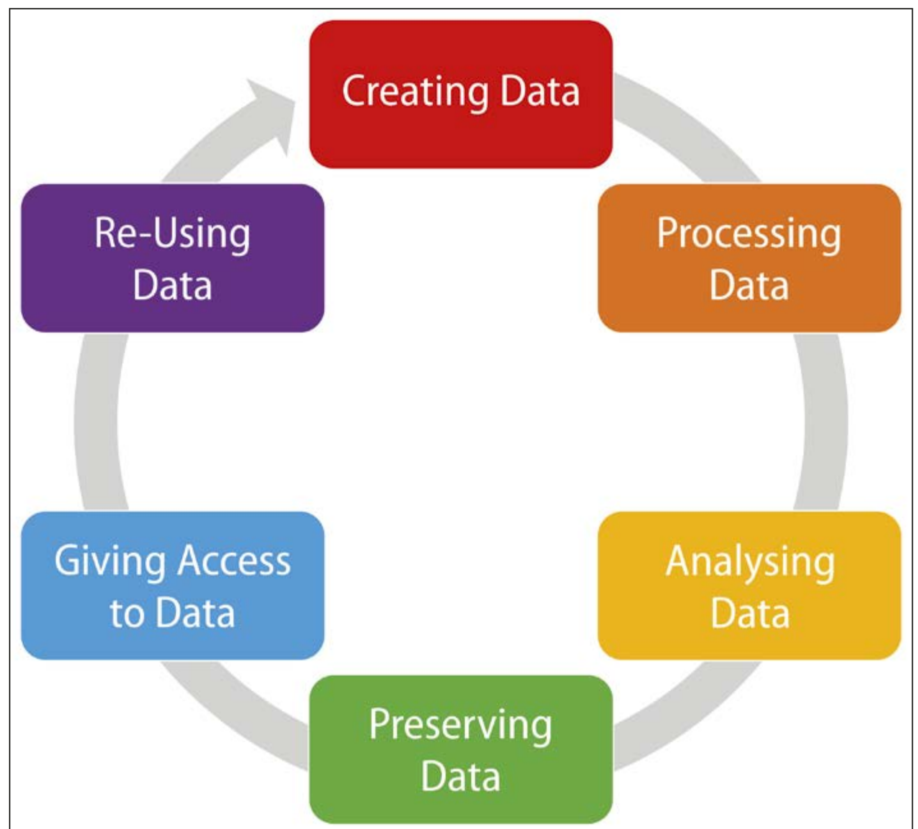

Abb. 1: Der Forschungsdatenzyklus des UK Data Archives

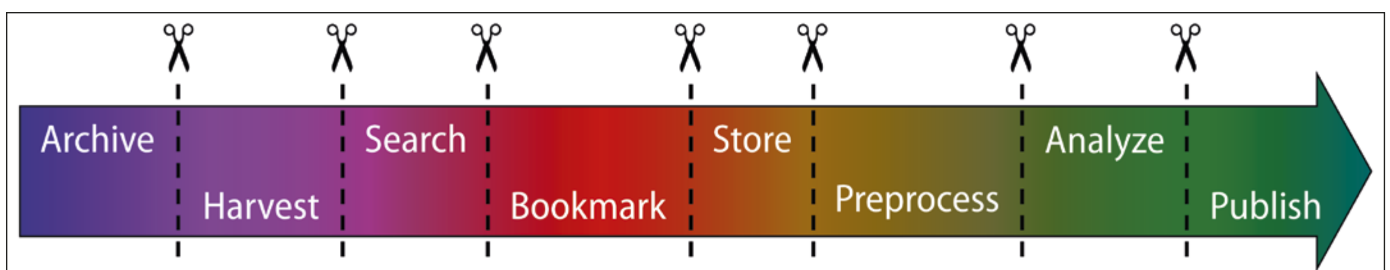

Abb. 2: Funktionskette – Forschungsdaten durchlaufen eine Kette mehrerer Funktionen

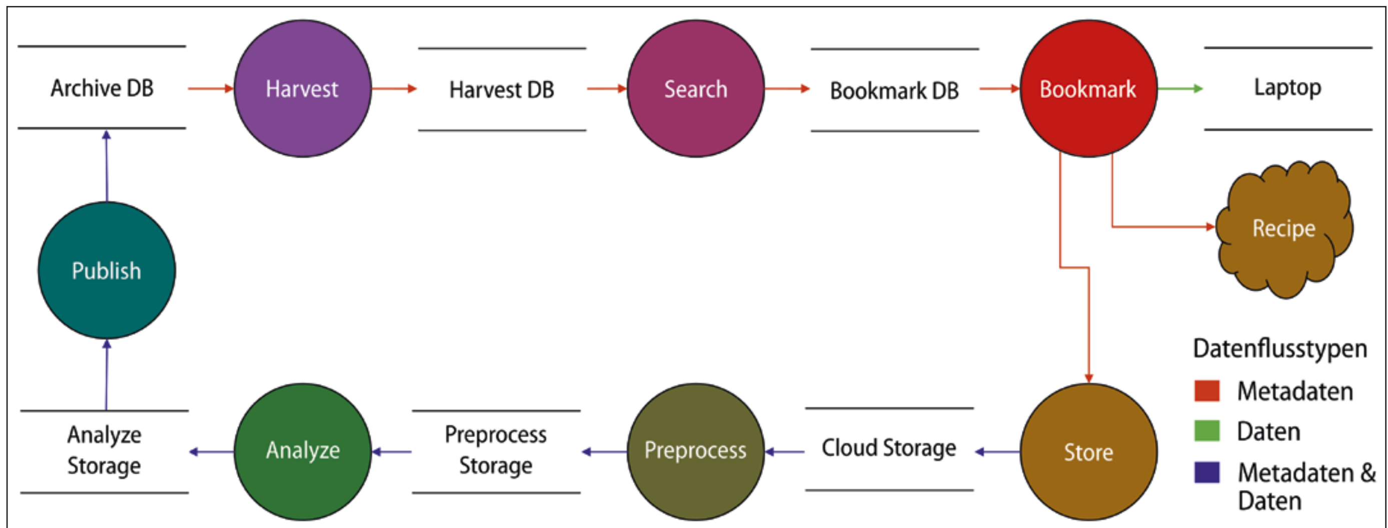

Abb. 3: Das Arbeiten mit Forschungsdaten kann als Datenfluss formalisiert werden

mäne als eigene Applikation und nutzen auch eigene Persistenzlösungen für jede Applikation. Des Weiteren kommunizieren SCS nur über aufrufbare Schnittstellen nach außen.

In **Abbildung 4** sehen wir, wie dies in GeRDI umgesetzt wird. Die einzelnen Funktionen des Datenflussdiagramms sind jeweils als eigenes abstraktes SCS in der Architektur repräsentiert. Um die Anforderungen der Nutzer zu implementieren, muss ein passendes SCS implementiert und an entsprechender Stelle „angeschlossen" werden. Dies ist möglich, da die Schnittstellen durch das abstrakte SCS vorgegeben werden. Da SCS, mit ihrem vertikalen Ansatz, sowohl die Geschäftslogik als auch die entsprechende Benutzeroberfläche und Persistenzlösung mit sich bringen, können verschiedene SCS als ein ganzes System integriert werden.

Ein solcher Ansatz für eine Architektur erlaubt es, Funktionen dynamisch zu integrieren, wie es auch in Softwareproduktlinien erlaubt wird. Arbeitsabläufe können repliziert und anschließend den eigenen Wünschen angepasst werden. So kann ein Forscher A die SCS eines Forschers B übernehmen, aber die verwendete **Analyze**-Komponente durch eine andere ersetzen und somit das System seinen Anforderungen anpassen.

Wir sehen also, dass der große Vorteil dieses Ansatzes in der Fähigkeit liegt, mit der Menge der Funktionen zu skalieren. Nachteilig hingegen ist die Verwendung bisher unbekannter Konstrukte, da das Konzept der SCS noch keinen breiten Bekanntheitsgrad genießt. So ist durchaus zu Beginn der Implementierung zu beobachten, dass vor allem der Paradigmenwechsel von einem Schichtenmodell zu einem Vertikalenmodell eine größere Hürde darstellen kann. Hier muss vor allem auf eine robuste Integration sowohl des Front- als auch des Backends geachtet werden. Für ähnliche Projekte, mit anfänglich unbekannten Anforderungen, kann sich dieser Aufwand dennoch auf längere Sicht lohnen.

Self-Contained System

Self-Contained System [SCS] ist ein Muster, welches genau abgegrenzte Funktionen eines Systems als eigenständige Applikation umsetzt. Der dahinterliegende Gedanke ist, einen Monolithen vertikal zu zerschneiden. Somit kapseln SCS das Benutzerinterface, die Geschäftslogik und die Persistenz einer Funktion, um diese vom restlichen System zu isolieren. Kommunikation findet zwischen verschiedenen SCS dabei bevorzugt asynchron über ReST-Schnittstellen statt. Die einzelnen Teile eines SCS können als Microservices implementiert werden und erlauben so eine Modularisierung der Interna eines SCS.

Vorteile dieses Musters sind direkt durch die Isolation der Funktionen bedingt. SCS bieten ein großes Potenzial für Elastizität, eine erhöhte Fehlertoleranz aufgrund der asynchronen Kommunikation und können unabhängig voneinander entwickelt und betrieben werden.

Kasten 1

Skalierung der Softwareentwicklung

Jedes Meeting, an dem Entwickler teilnehmen, bedeutet unproduktive Zeit, zumindest hinsichtlich der Softwareproduktion. Lange Kommunikationswege erschweren zusätzlich produktiven Output, weshalb beides vermieden werden sollte. Bei größeren Projekten kommt erschwerend hinzu, dass diese eventuell an verteilten Standorten durchgeführt werden. Der kurze Gang zum Büro der Kollegen verwandelt sich dann zu einer langen E-Mail-Kette. Um dies zu vermeiden, gilt es hier – aber auch in kleineren Projekten –, das zu entwickelnde System diesen Gegebenheiten anzupassen. Es sind natürlich Meetings erforderlich zur Absprache der Schnittstellen, diese sollten jedoch auf ein notwendiges Minimum reduziert werden. Eine geeignete Möglichkeit ist hierbei das System entlang der Teamstruktur zu organisieren, wie es bereits Conway's Gesetz bewirbt [Her99]. Das System wird in fachlich getrennte Komponenten geschnitten, welche unabhängig voneinander entwickelt werden. Da jede Komponente einem Team zugewiesen werden kann, skaliert die Systementwicklung somit auch mit den Entwicklerteams. Soll eine neue Komponente bereitgestellt werden, so kann dies einem neu gegründeten Team übergeben werden. Dessen Aufgabe besteht zu Beginn darin, Schnittstellen zu definieren. Diese Aufgabe benötigt die meiste Koordinierung mit anderen Teams. Sobald robuste Schnittstellen gefunden sind, kann sich das Team der Implementierung der Komponente zuwenden. Rücksprachen mit anderen Teams sind hier seltener erforderlich. Da einzelne Teams tendenziell aus einer geringen An-

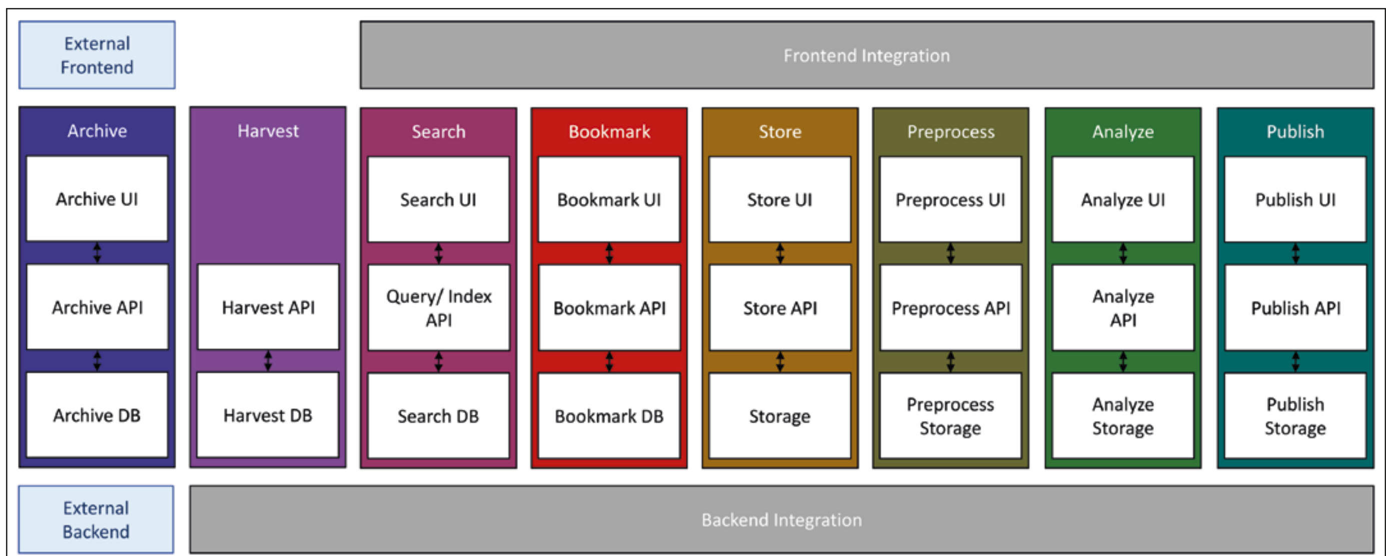

Abb. 4: GeRDIs Architektur – die einzelnen Funktionen des Datenflussdiagramms werden mit SCS umgesetzt

zahl von Entwicklern bestehen, sind die Kommunikationswege kurz, was in einem geringen Overhead resultiert.

Der obige Absatz scheint auf den ersten Blick leicht umsetzbar. Doch genauer betrachtet ergeben sich auch hier Herausforderungen. Eine davon betrifft die Tatsache, dass bestimmte Komponenten nicht leicht getrennt werden können. Middleware-Code oder bestimmte Funktionen werden an mehreren Stellen des Systems verwendet und sind somit nicht allein einer Komponente zuzuschreiben.

Solche Cross-Cutting Concerns treten oft im Bereich der Backend-Integration auf. So betrifft beispielsweise eine „Authentication and Authorization Infrastructure“ (AAI) das ganze System und ist inhärent keiner einzelnen Komponente zuweisbar. Zwei einfache Lösungsansätze wären hier, diese Systemteile entweder von einem einzigen Team entwickeln und warten zu lassen oder von allen Teams gemeinsam. Beide Ansätze bringen jedoch Probleme mit sich.

Wird solch ein Cross-Cutting Concern von einem einzigen Team entwickelt, so werden tendenziell eher die eigenen Anforderungen abgedeckt. Benötigen andere Teams Änderungen oder neue Funktionen für diese Software, so entsteht wieder Kommunikationsbedarf, dessen Minimierung das eigentliche Ziel ist.

Werden hingegen alle betroffenen Teams involviert, so sind Gespräche und Diskussionen bereits strukturell bedingt unvermeidbar und es widerspricht unserem Ziel der Trennung der Teams.

Otto.de und andere Internet-Scale Systeme haben für dieses Problem eine einfache Lösung gefunden: Open-Source-Software (OSS) [Otto]. OSS beweist, dass es möglich ist, Software effizient mit verteilten Teams zu entwickeln, was genau das gewünschte Ziel für Cross-Cutting Softwarekomponenten ist. Die verwendeten Methoden erlauben es, dynamisch Entwickler einzubinden, wodurch Teams in Eigeninitiative ihren Beitrag leisten können.

OSS hat darüber hinaus auch weitere Vorteile. Die höhere Sichtbarkeit von OSS zwingt Entwickler dazu, sauberer zu arbeiten. Dies bedeutet auch, dass insbesondere ein hoher Wert auf eine hohe Qualität gelegt wird. Dies soll eine größere Akzeptanz gewährleisten und auch das Potenzial für Fehler reduzieren. Da der Programmcode weltweit sichtbar ist, wird eher darauf geachtet, dass dieser generisch ist und nicht nur die eigenen Rahmenbedingungen erfüllt. Des Weiteren ist es möglich, dass externe Entwickler einen Beitrag leisten, sollte die Software ein bereits bestehendes Problem angehen.

All dies sind Punkte, die beispielsweise Netflix verstärkt auf die Entwicklung von OSS fokussieren lässt [OSS]. Hier zeigte es sich, dass die Vorteile von OSS auch auf lange Sicht überwiegen können. Unser Anwendungsfall erstellt ausschließlich OSS, daher liegt hier der Fokus auf der Nutzung und Bereitstellung von OSS für Cross-Cutting Concerns, die nicht Teil der Kerndomäne des Systems sind.

Ein Architektur-Framework für skalierbare und flexible Systeme

Sowohl die Implementierung einer vertikalen Architektur als auch die Zuweisung von Applikationen an dedizierte Teams benötigt ein geeignetes technisches Framework, um zu funktionieren. Es ist schwer, eine allgemeingültige Lösung hierfür zu formulieren. Unsere gesammelten Erfahrungen im GeRDI-Projekt zeigen jedoch, was sich als Best Practice erwiesen hat.

Wie bereits erwähnt, setzt die Architektur auf dem Pattern der SCS auf. Da jede Instanz eines solchen Systems für sich abgeschlossen ist, kann eine individuelle Implementierungstechnik für diese gewählt werden. Dies bedeutet insbesondere, dass verschiedene Programmiersprachen oder Lösungen für die Persistenz genutzt werden können (polyglotte Stacks).

Dies kam in GeRDI als naheliegende Lösung infrage, da sich die bestehende Expertise an den verschiedenen Standorten als sehr unterschiedlich erwies. Die freie Wahl des Technologiestacks für jedes SCS ermöglicht einen schnelleren Release einer ersten Version einzelner Komponenten, da Einarbeitungszeit in eine vornherein ausgewählte Technologie entfällt. Dies trägt somit insbesondere zur Skalierbarkeit in Bezug auf die Entwicklungsteams bei.

Ein weiterer Vorteil ist die Wahl passender Technologien für jeden Anwendungsfall. So kann beispielsweise für jedes SCS die passende Datenbank für den entsprechenden Anwendungsfall ausgewählt werden. In unserem Fallbeispiel werden verschiedene Java-Technologiestacks oder auch PHP-Frameworks genutzt. Auch erste Beispiele mit der Sprache Go wurden schon implementiert. Im Falle der Persistenzlösungen setzen wir beispielsweise auf MongoDB, Elasticsearch und ein direktes Speichern von Dokumenten auf dem Hauptspeicher.

Die Verwendung verschiedener Technologiestacks hat auch Nachteile, wie den Bedarf verschiedener Systemumgebungen. Für Java ist dies eine geeignete Java Vir-

Literatur & Links

[Gru17] R. Grunzke u. a., Challenges in Creating a Sustainable Generic Research Data Infrastructure, in: Softwaretechnik-Trends 37 (2), siehe: **http://eprints.uni-kiel.de/38756/**

[Has17] W. Hasselbring, G. Steinacker, Microservice Architectures for Scalability, Agility and Reliability in E-Commerce, in: IEEE Int. Conf. on Software Architecture Workshops 2017 (ICSAW 2017), siehe: **http://eprints.uni-kiel.de/37489/**

[Has18] W. Hasselbring, Software Architecture: Past, Present, Future, in: The Essence of Software Engineering, Springer, 2018, siehe: **http://dx.doi.org/10.1007/978-3-319-73897-0_10**

[Her99] J. D. Herbsleb, R. E. Grinter, Splitting the Organization and Integrating the Code: Conway's Law Revisited, in: Proc. of the 21st Int. Conf. on Software Engineering (ICSE '99), 1999, siehe: **https://dl.acm.org/citation.cfm?doid=302405.302455**

[OSS] R. Meshenber, Open Source at Netflix, Medium, 2012, siehe: **https://medium.com/netflix-techblog/open-source-at-netflix-c2c4e036e144**

[Otto] G. Steinacker, On Monoliths and Microservices, Otto.de, 2015, siehe: **https://dev.otto.de/2015/09/30/on-monoliths-and-microservices/**

[SCS] Self-Contained Systems, siehe: **http://scs-architecture.org/**

[Tav18] N. Tavares de Sousa, W. Hasselbring, T. Weber, D. Kranzlmüller, Designing a Generic Research Data Infrastructure Architecture with Continuous Software Engineering, in: Combined Proc. of the Workshops of the German Software Engineering Conference 2018 (SE 2018), 2018, siehe: **http://ceur-ws.org/Vol-2066/cse2018paper03.pdf**

tual Machine, für andere Programmiersprachen eventuell das Vorhandensein bestimmter Bibliotheken. Dies ist für den Betrieb eines solchen Softwaresystems eine Herausforderung.

Diesen Nachteil können wir jedoch durch eine geeignete Handhabung minimieren. Dies ist durch eine geeignete Abstraktion des Betriebs möglich. Um diese umzusetzen, setzen wir auf Docker und liefern alle Komponenten als Images aus, welche den gesamten Umgebungskontext mit sich bringen. Die Verantwortung für die Bereitstellung solcher Images liegt bei den Entwicklern der entsprechenden Komponenten.

Wir haben die Erfahrung gemacht, dass polyglotte Technologiestacks ein geeignetes Framework für skalierbare Systeme darstellen, welche auch in der Praxis durch die Verwendung von Containertechnologien beherrschbar ist. Für den Betrieb einer ganzen Infrastruktur setzen wir auf Kubernetes, um die Container zu orchestrieren und auch Ressourcen gezielt zuzuweisen. Dies in Kombination mit Continuous Deployment erlaubt es uns, agiler neue Funktionen zu veröffentlichen, um diese unseren Anwendern schnell zukommen zu lassen.

Es werden ausgiebige Tests benötigt, um sicherzustellen, dass möglichst fehlerfreie Software in den laufenden Betrieb übergeht. Auch erlaubt das schnelle Deployen neuer Versionen ein schnelleres Beheben von Fehlern. In ähnlichen Architekturen wurde bereits gezeigt, dass dieses Verfahren zu einer Reduzierung von Fehlern in Produktivsystemen führen kann [Has17].

Für die asynchrone Kommunikation zwischen den einzelnen SCS nutzen wir in unserem Fallbeispiel Kafka als Message-Broker. Statt direkter Aufrufe eines SCS werden hier Nachrichten gesendet, welche von denjenigen SCS gelesen werden, die diese Daten benötigen. Dies erhöht in erster Linie die Toleranz gegen teilweise Systemausfälle, da die Nachrichten für eine definierte Zeit vorgehalten werden und somit nach einem Neustart oder einer Behebung des Fehlers neu ausgelesen werden können.

Für uns bringt Kafka einen weiteren Vorteil: Die einzelnen Systemteile müssen nicht mehr dem ganzen System bekannt sein. Da alle SCS lediglich auf Kafka zugreifen müssen, entfällt die Notwen-

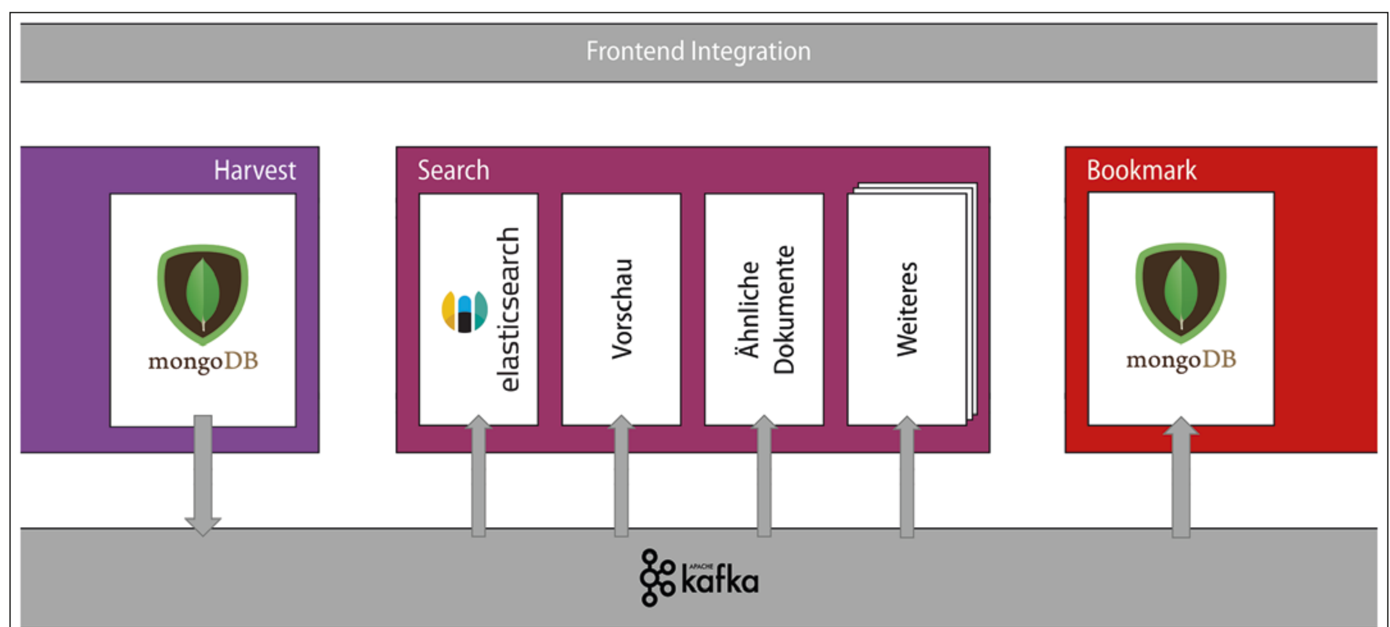

Abb. 5: Asynchrone Kommunikation erleichtert die Funktionserweiterung zu späteren Zeitpunkten

digkeit einer Service-Registry/Discovery. Dies stellt sich insbesondere als großer Vorteil heraus, sobald neue Funktionen in das System integriert werden müssen. Bei einem klassischen Ansatz mit synchroner Kommunikation oder einer Service-Registry müssen alle Endpunkte dem restlichen System bekannt sein, da diese sonst nicht angesprochen werden können. Mit einem Message-Broker entfällt dies und es können neue Systemteile ohne Weiteres integriert werden.
In unserem Fallbeispiel nutzen wir dies bei der Suchmaschine (**siehe Abbildung 5**). Der geforderte Funktionsumfang der Suchmaschine ist noch nicht vollständig geklärt. Somit können weitere Anforderungen, wie eine Vorschau der hinterlegten Daten, benötigt werden. Da alle für die Suchmaschine gesammelten Daten über Kafka an die angeschlossenen Systemteile verteilt werden, können neue SCS an Kafka angeschlossen werden und erhalten sofort Zugriff auf die Daten ohne jede weitere Konfiguration. Dies kann mit einer beliebigen Anzahl weiterer Consumer geschehen, auch systemweit.

Fazit

Software sollte bezüglich diverser Aspekte skalieren. Wir haben eine datenflussbasierte Architektur vorgestellt, die unter anderem auch mit den Anforderungen und der Softwareentwicklung skaliert. Für diverse Anwendungsfälle wird dieser Ansatz sicher viele Vorteile mit sich bringen, dennoch muss dies von Fall zu Fall neu betrachtet werden.
Ist der Ansatz anwendbar, zeigen sich mehrere Vorteile. So kann der Funktionsumfang der Architektur erweitert werden, ohne dass tiefgehende Eingriffe in das System notwendig werden. Des Weiteren können auch die Verantwortlichkeiten der einzelnen Systemkomponenten so an Entwicklerteams verteilt werden, dass die Entwicklung effizienter gestaltet werden kann.
Für unser Fallbeispiel erweist sich dieser Ansatz als vorteilhaft, da die verteilte Entwicklung vereinfacht wird und auch nicht alle Anforderungen klar sein müssen. Ersteres hat sich bereits im Alltag beweisen können. Unser Fallbeispiel wird an fünf Standorten entwickelt mit jeweils unterschiedlichem Fachwissen. Letzteres erlaubt es, ein flexibles System zu entwerfen. Wir erwarten von diesem System eine gesteigerte Resilienz und Flexibilität, die nicht auf Kosten der Performanz oder Wartbarkeit geht. ||

Die Autoren

Nelson Tavares de Sousa
(tavaresdesousa@email.uni-kiel.de)
ist wissenschaftlicher Mitarbeiter an der Christian-Albrechts-Universität zu Kiel. Im DFG-geförderten Projekt GeRDI (Generic Research Data Infrastructure) ist er als Chefarchitekt verantwortlich für die Koordination der Gesamtarchitektur.

Prof. Dr. Wilhelm Hasselbring
(hasselbring@email.uni-kiel.de)
lehrt und forscht zu Software-Engineering an der Universität Kiel. Er ist Sprecher der Kieler Projekte im „Kompetenzverbund Software Systems Engineering" KoSSE und fachlicher Leiter der User Group „Softwarearchitektur und Softwareentwicklung" der Softwareforen Leipzig.