Architekturen verteilter Anwendungen

Ein erster Überblick.

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 1.0

Ausgewählte Folien basieren auf Folien von Maarten van Steen (*Distributed Systems*)

Alle Fehler sind meine eigenen.

Folien: https://delors.github.io/ds-architekturen/folien.de.rst.html

https://delors.github.io/ds-architekturen/folien.de.rst.html.pdf

Fehler melden:

https://github.com/Delors/delors.github.io/issues

1. Grundlegende Architekturen

Architekturstile (🇺🇸 *Architectural Styles*)

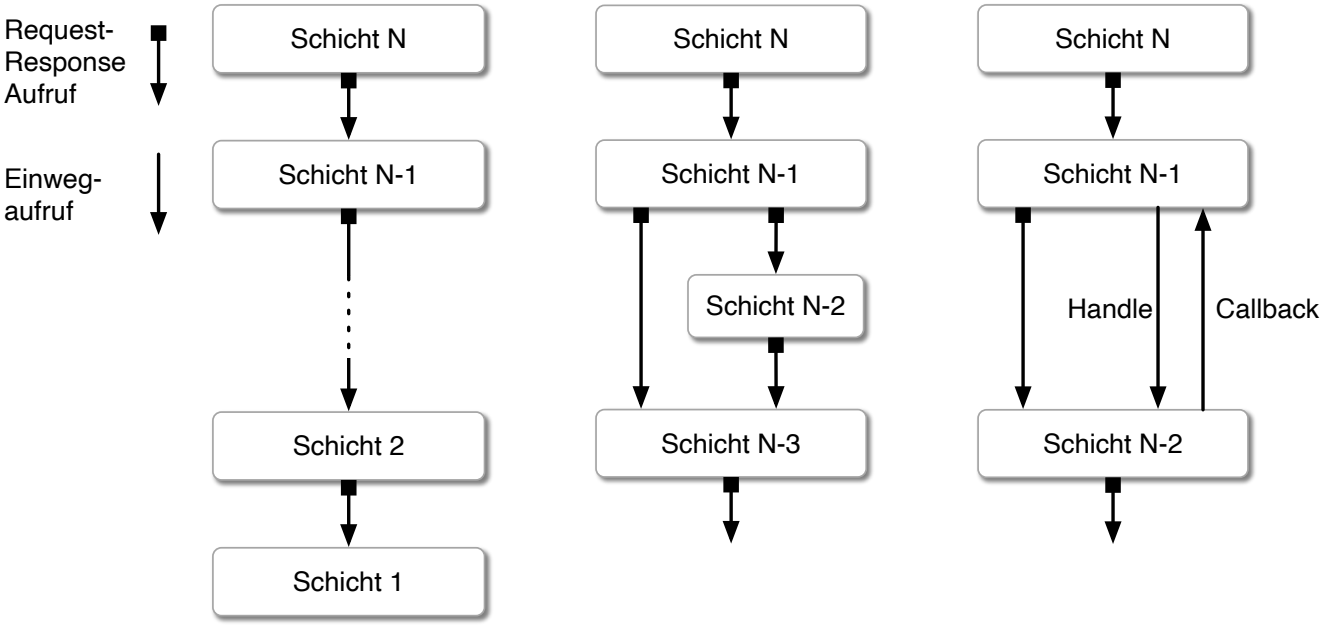
Ein Architekturstil wird formuliert in Form von

- (austauschbare) Komponenten mit klar definierten Schnittstellen
- der Art und Weise, wie die Komponenten miteinander verbunden sind
- die zwischen den Komponenten ausgetauschten Daten
- der Art und Weise, wie diese Komponenten und Verbindungen gemeinsam zu einem System konfiguriert werden System.

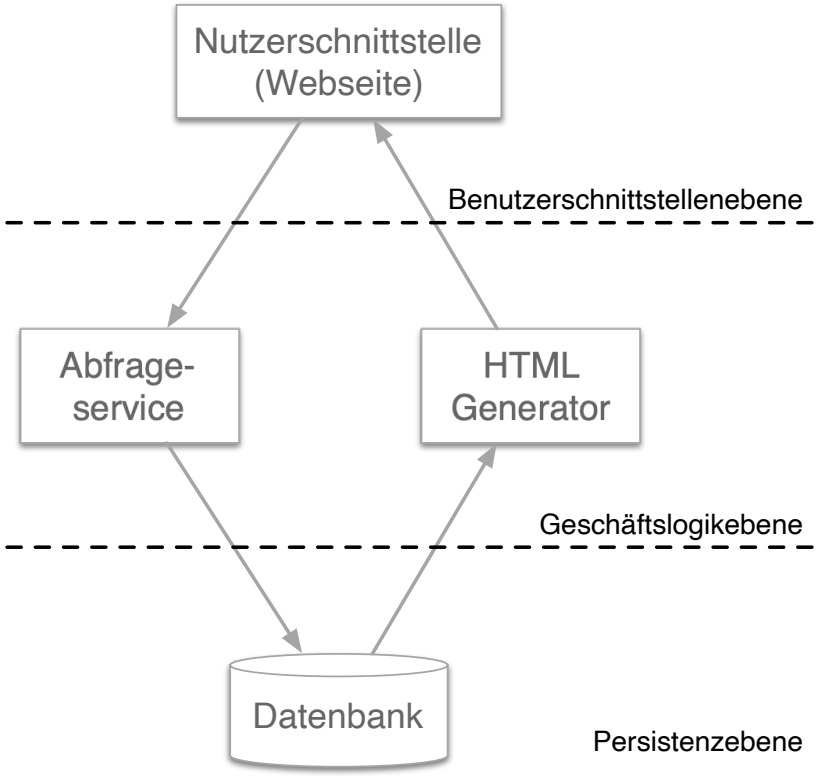
Konnektor

Ein Mechanismus, der die Kommunikation, Koordination oder Kooperation zwischen Komponenten vermittelt. Beispiel: Einrichtungen für (entfernte) Prozeduraufrufe (RPC), Nachrichtenübermittlung oder Streaming.

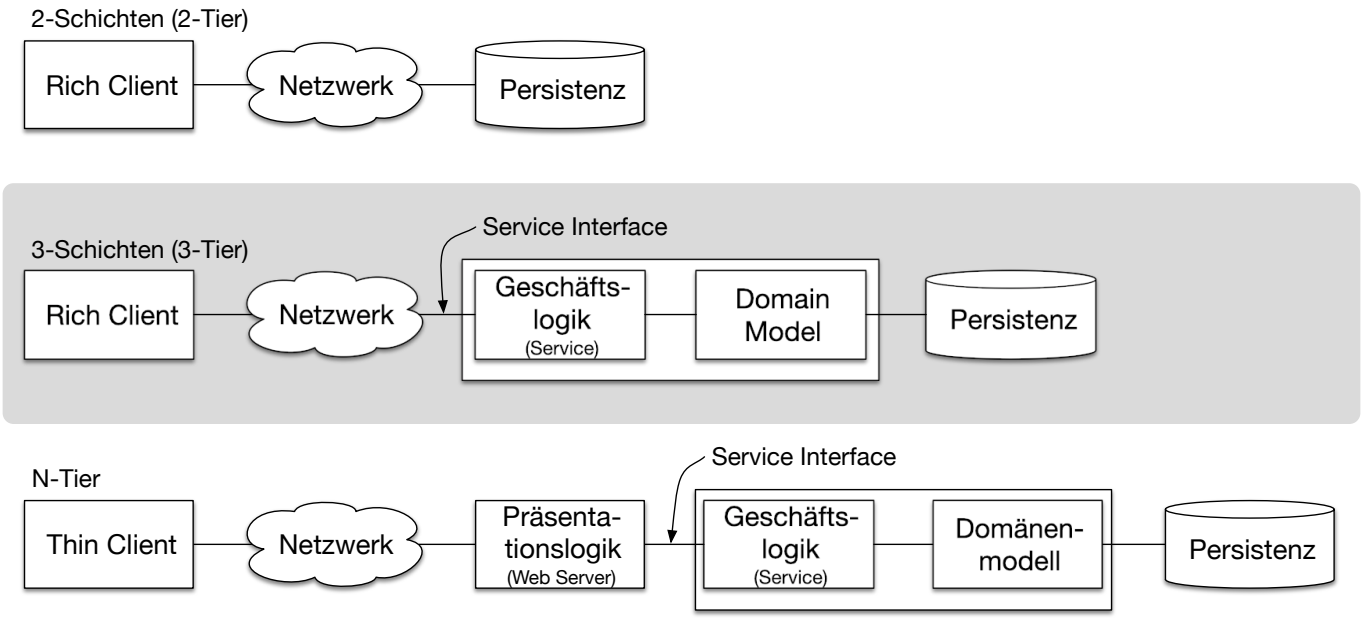
Schichtenarchitekturen



Beispiel einer 3-Schichtenarchitektur



Klassische Architekturen



Traditionelle Dreischichtenarchitektur

Diese Architektur findet sich in vielen verteilten Informationssystemen mit traditioneller Datenbanktechnologie und zugehörigen Anwendungen.

- Die Präsentationsschicht stellt die Schnittstelle zu Benutzern oder externen Anwendungen dar.
- Die Verarbeitungsschicht implementiert die Geschäftslogik.
- Die Persistenz-/Datenschicht ist für die Datenhaltung verantwortlich.

Publish and Subscribe Architekturen

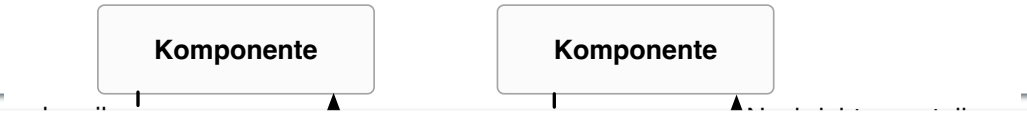
Abhängigkeiten zwischen den Komponenten werden durch das *Publish and Subscribe* Paradigma realisiert mit dem Ziel der loosen Kopplung.

Taxonomie der Koordinierungsansätze in Hinblick auf Kommunikation und Koordination:

	Zeitlich gekoppelt	Zeitlich entkoppelt
Referentiell gekoppelt	Direkt Koordination	Mailboxkoordination
Referentiell entkoppelt	ereignisbasierte Koordination (*Event-based Coordination*)	gemeinsam genutzter Datenspeicher (*Shared Data Space*)

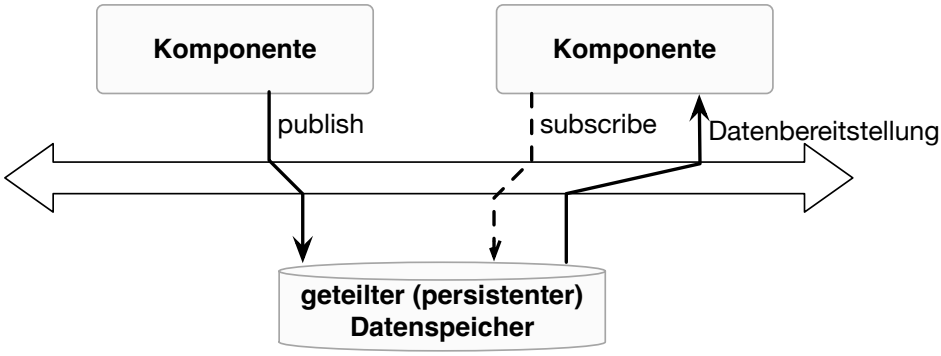
1

Ereignisbasierte Koordination



2

Shared Data Space



3

Häufig wird die *ereignisbasierte Koordination* in Kombination mit *Shared Data Space* zur Realisierung von *Publish and Subscribe* Architekturen.

7

Direkte Koordination

Ein Prozess interagiert unmittelbar (⇒ zeitliche Kopplung) mit genau einem anderen wohl-definierten Prozess (⇒ referentielle Kopplung).

Mailboxkoordination

Die miteinander kommunizierenden Prozesse interagieren nicht direkt miteinander, sondern über eine eindeutige Mailbox (⇒ referentielle Kopplung). Dies ermöglicht es, dass die Prozesse nicht zeitgleich verfügbar sein müssen.

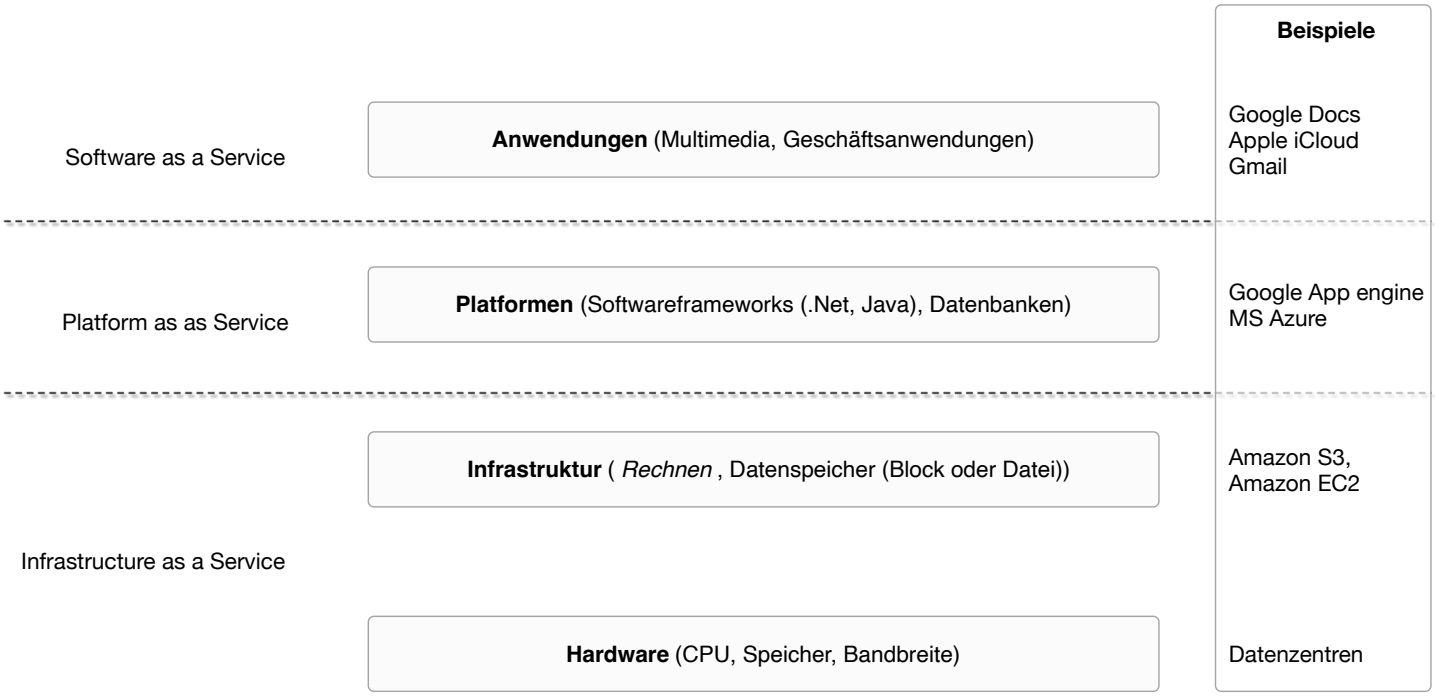
Ereignisbasierte Koordination

Ein Prozess löst Ereignisse aus, auf die *irgendein* anderer Prozesse direkt reagiert. Ein Prozess, der zum Zeitpunkt des Auftretens des Ereignisses nicht verfügbar ist, sieht das Ereignis nicht.

Gemeinsam genutzter Datenspeicher

Prozesse kommunizieren über Tuples, die in einem gemeinsam genutzten Datenspeicher hinterlegt werden. Ein Prozess, der zum Zeitpunkt des Schreibens nicht verfügbar ist, kann das Tuple später lesen. Prozesse definieren Muster in Hinblick auf die Tuples, die sie lesen wollen.

Aufbau von Cloud Computing Anwendungen



Es können vier Schichten unterschieden werden:

- Hardware: Prozessoren, Router, Stromversorgungs- und Kühlsysteme.

  Für Kunden normalerweise vollkommen transparent.

- Infrastruktur: Einsatz von Virtualisierungstechniken zum Zwecke der Zuweisung und Verwaltung virtueller Speichere und virtueller Server.
- Plattformen: Bietet Abstraktionen auf höherer Ebene für Speicher und dergleichen.

  Beispiel: Das Amazon S3-Speichersystem bietet eine API für (lokal erstellte) Dateien, die in sogenannten Buckets organisiert und gespeichert werden können.

- Anwendung: Tatsächliche Anwendungen, wie z. B. Office-Suiten (Textverarbeitungsprogramme, Tabellenkalkulationsprogramme, Präsentationsanwendungen).

  Vergleichbar mit der Suite von Anwendungen, die mit Betriebssystemen ausgeliefert werden.

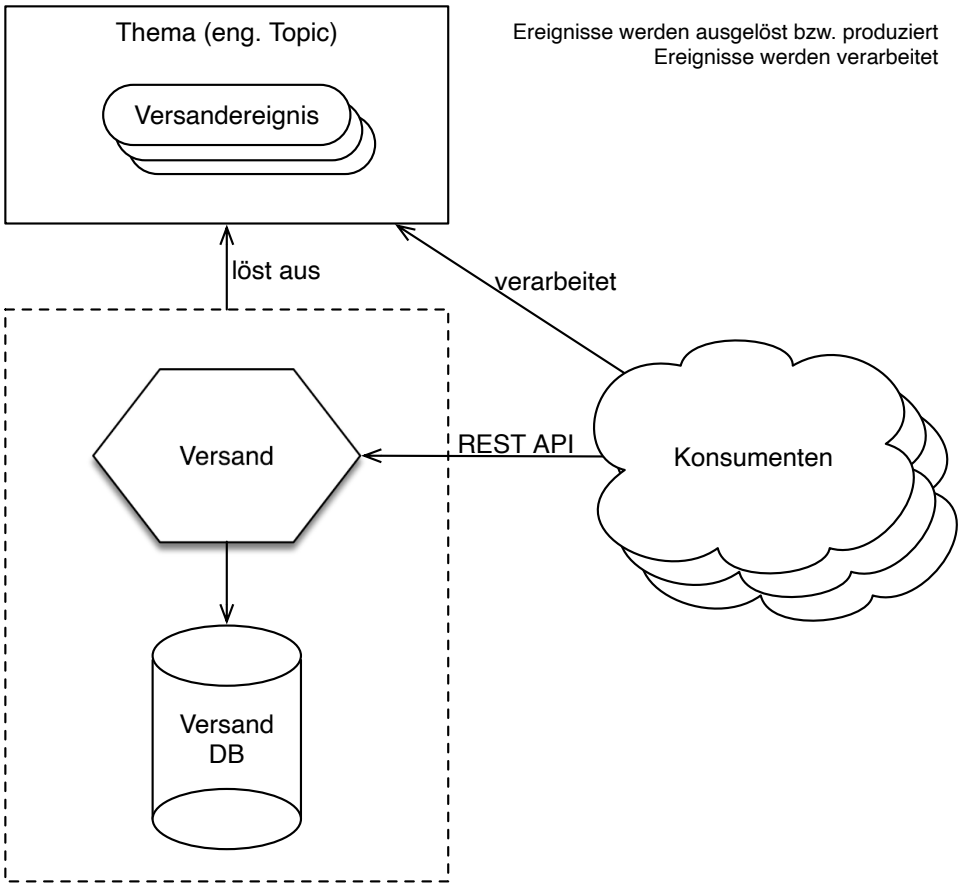
2. Microservices

[Newman2021]

Microservices

Ein einfacher Microservice, der eine REST Schnittstelle anbietet und Ereignisse auslöst.

Wo liegen hier die Herausforderungen?



Eine große Herausforderung ist das Design der Schnittstellen. Um wirkliche Unabhängigkeit zu erreichen, müssen die Schnittstellen sehr gut definiert sein. Sind die Schnittstellen nicht klar definiert oder unzureichend, dann kann das zu viel Arbeit und Koordination zwischen den Teams führen, die eigentlich unerwünscht ist!

Schlüsselkonzepte von Microservices

- können unabhängig bereitgestellt werden (🇺🇸 *independently deployable*)

  und werden unabhängig entwickelt

- modellieren eine Geschäftsdomäne

  Häufig entlang einer Kontextgrenze (eng. Bounded Context) oder eines Aggregats aus DDD

- verwalten Ihren eigenen Zustand

  d. h. keine geteilten Datenbanken

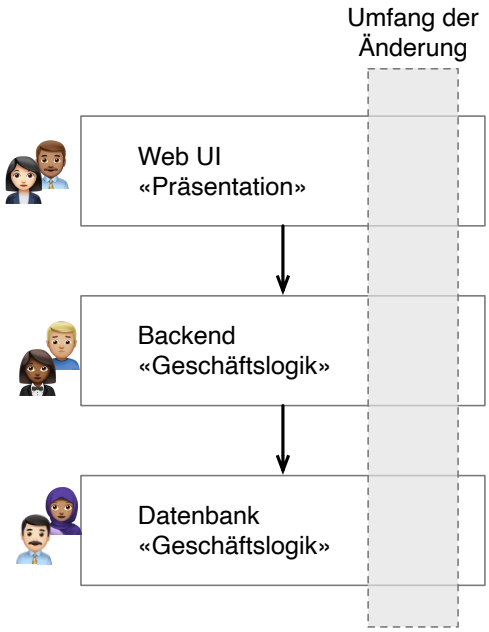
- sind klein

  Klein genug, um durch (max.) ein Team entwickelt werden zu können

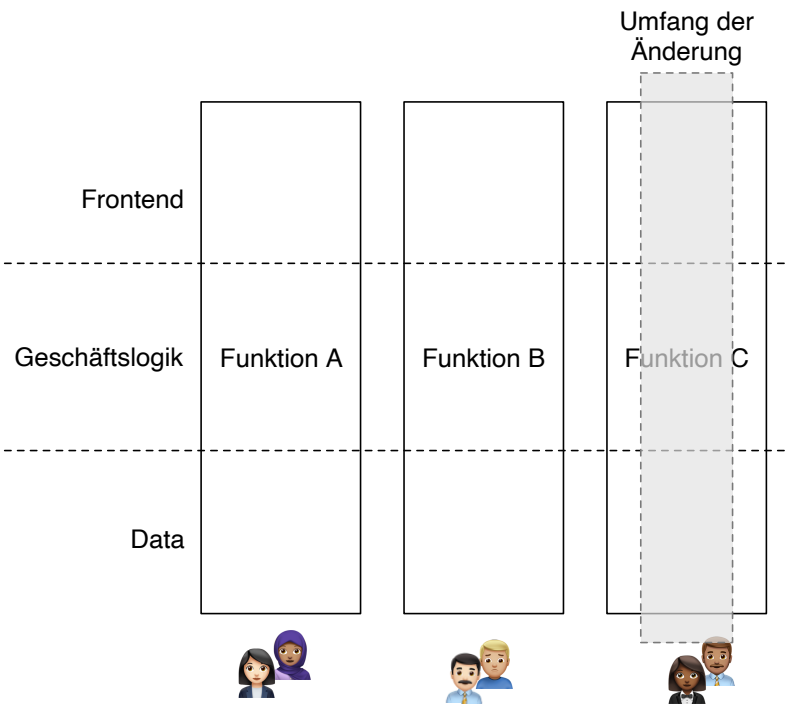
- flexibel bzgl. Skalierbarkeit, Robustheit, eingesetzter Technik
- erlauben das Ausrichten der Architektur an der Organisation (vgl. Conway's Law)

Microservices und Conway's Law

Traditionelle Schichtenarchitektur

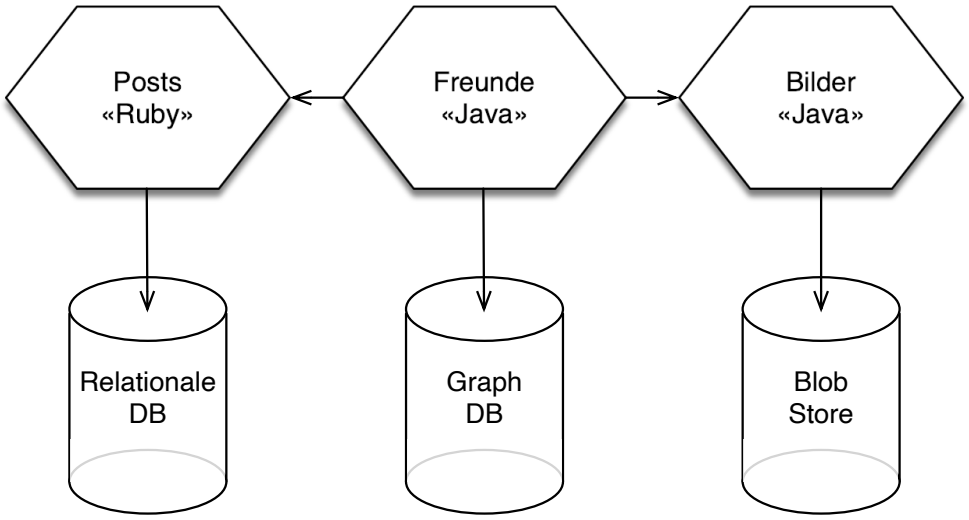


Microservices Architektur



Microservices und Technologieeinsatz

Microservices sind flexibel bzgl. des Technologieeinsatzes und ermöglichen den Einsatz „der geeignetsten“ Technologie.



Aktuelle Standardtechnologien

Position Apr 2012	Position Apr 2011	Delta in Position	Programming Language	Ratings Apr 2012	Delta Apr 2011	Status
1	2	↑	C	17.555%	+1.39%	A
2	1	↓	Java	17.026%	-2.02%	A
3	3	=	C++	8.896%	-0.33%	A
4	8	↑↑↑↑	Objective-C	8.236%	+3.85%	A
5	4	↓	C#	7.348%	+0.16%	A
6	5	↓	PHP	5.288%	-1.30%	A
7	7	=	(Visual) Basic	4.962%	+0.28%	A
8	6	↓↓	Python	3.665%	-1.27%	A
9	10	↑	JavaScript	2.879%	+1.37%	A
10	9	↓	Perl	2.387%	+0.40%	A
11	11	=	Ruby	1.510%	+0.03%	A
12	24	↑↑↑↑↑↑↑↑↑↑↑↑	PL/SQL	1.373%	+0.92%	A
13	13	=	Delphi/Object Pascal	1.370%	+0.34%	A
14	35	↑↑↑↑↑↑↑↑↑↑↑↑	Visual Basic .NET	0.978%	+0.64%	A
15	15	=	Lisp	0.951%	+0.02%	A
16	17	↑	Pascal	0.812%	+0.10%	A
17	16	↓	Ada	0.783%	+0.01%	A–
18	18	=	Transact-SQL	0.760%	+0.18%	A
19	22	↑↑↑	Logo	0.652%	+0.12%	B
20	52	↑↑↑↑↑↑↑↑↑↑↑↑	NXT-G	0.578%	+0.35%	B

Quelle: TIOBE Programming Community Index - April 2012

1

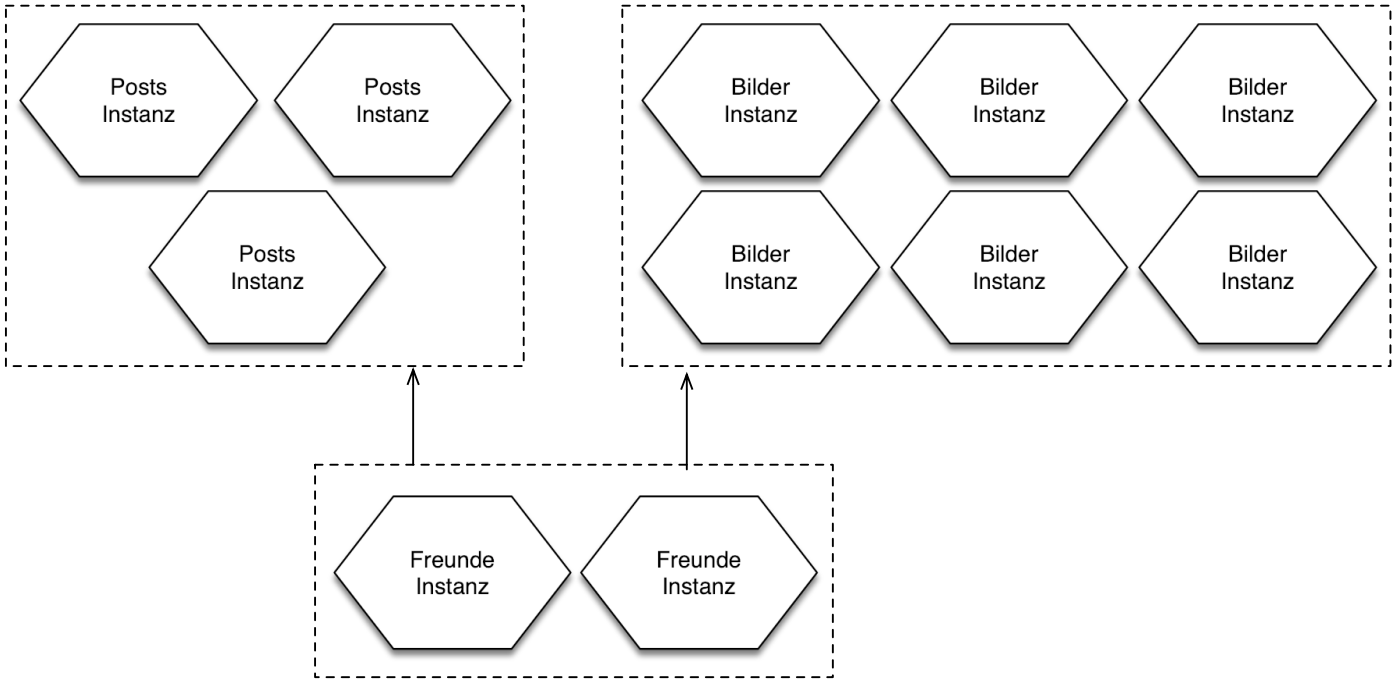
Feb 2024	Feb 2023	Change	Programming Language	Ratings	Change
1	1		Python	15.16%	-0.32%
2	2		C	10.97%	-4.41%
3	3		C++	10.53%	-3.40%
4	4		Java	8.88%	-4.33%
5	5		C#	7.53%	+1.15%
6	7	⌃	JavaScript	3.17%	+0.64%
7	8	⌃	SQL	1.82%	-0.30%
8	11	⌃	Go	1.73%	+0.61%
9	6	⌄	Visual Basic	1.52%	-2.62%
10	10		PHP	1.51%	+0.21%
11	24	⌃⌃	Fortran	1.40%	+0.82%
12	14	⌃	Delphi/Object Pascal	1.40%	+0.45%
13	13		MATLAB	1.26%	+0.27%
14	9	⌄⌄	Assembly language	1.19%	-0.19%
15	18	⌃	Scratch	1.18%	+0.42%
16	15	⌄	Swift	1.16%	+0.23%
17	33	⌃⌃	Kotlin	1.07%	+0.76%
18	20	⌃	Rust	1.05%	+0.35%
19	30	⌃⌃	COBOL	1.01%	+0.60%
20	16	⌄⌄	Ruby	0.99%	+0.17%

Quelle: TIOBE Programming Community Index - Feb. 2024

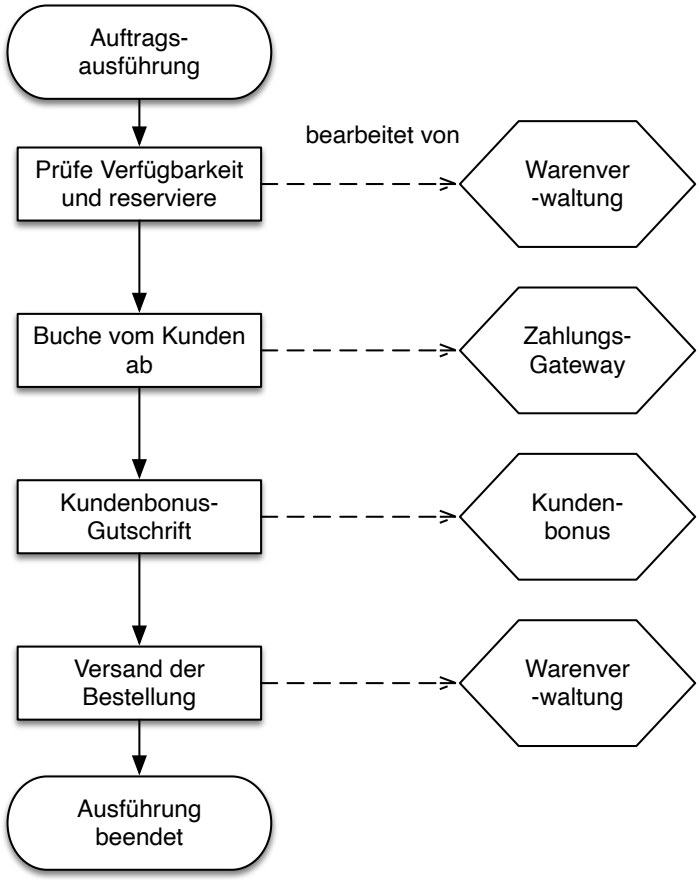
2

Microservices und Skalierbarkeit

Sauber entworfene Microservices können sehr gut skaliert werden.

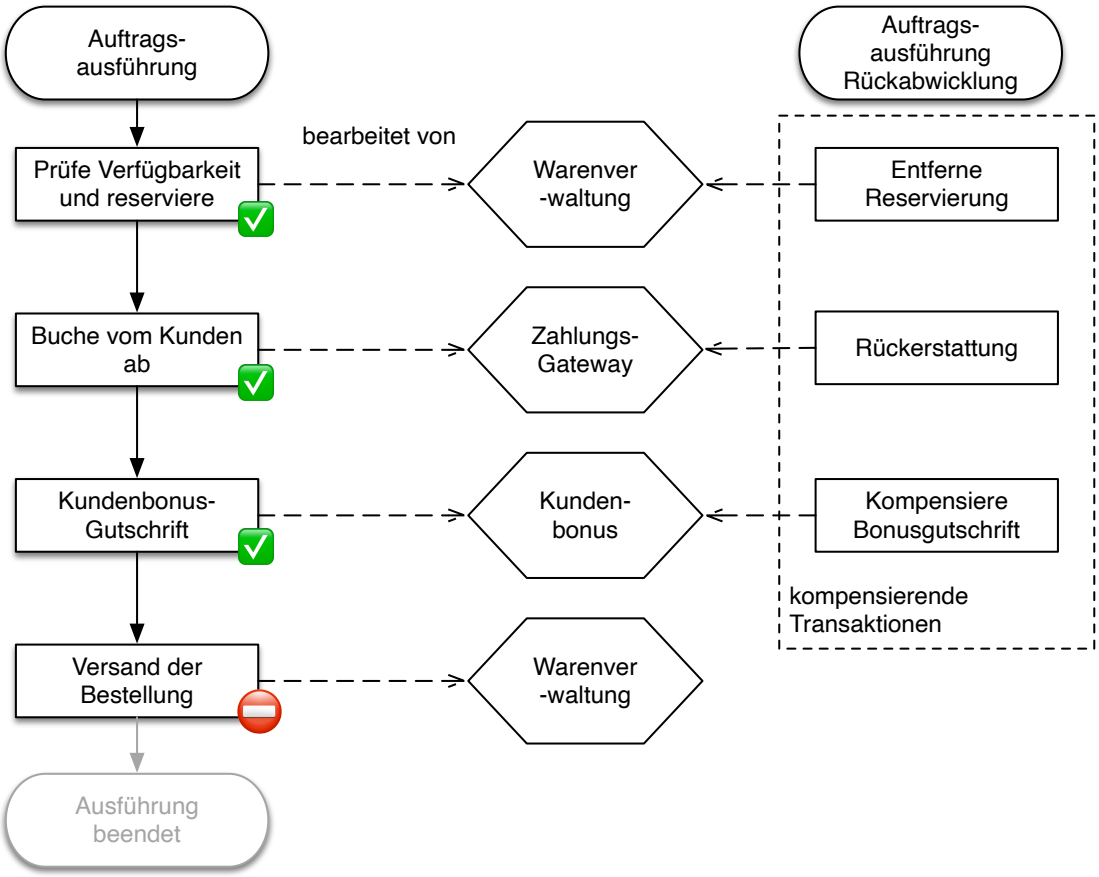


Implementierung einer langlebigen Transaktion?



Die Implementierung von Transaktionen ist eine der größten Herausforderungen bei der Entwicklung von Microservices.

Aufteilung einer langlebigen Transaktion mit Hilfe von Sagas

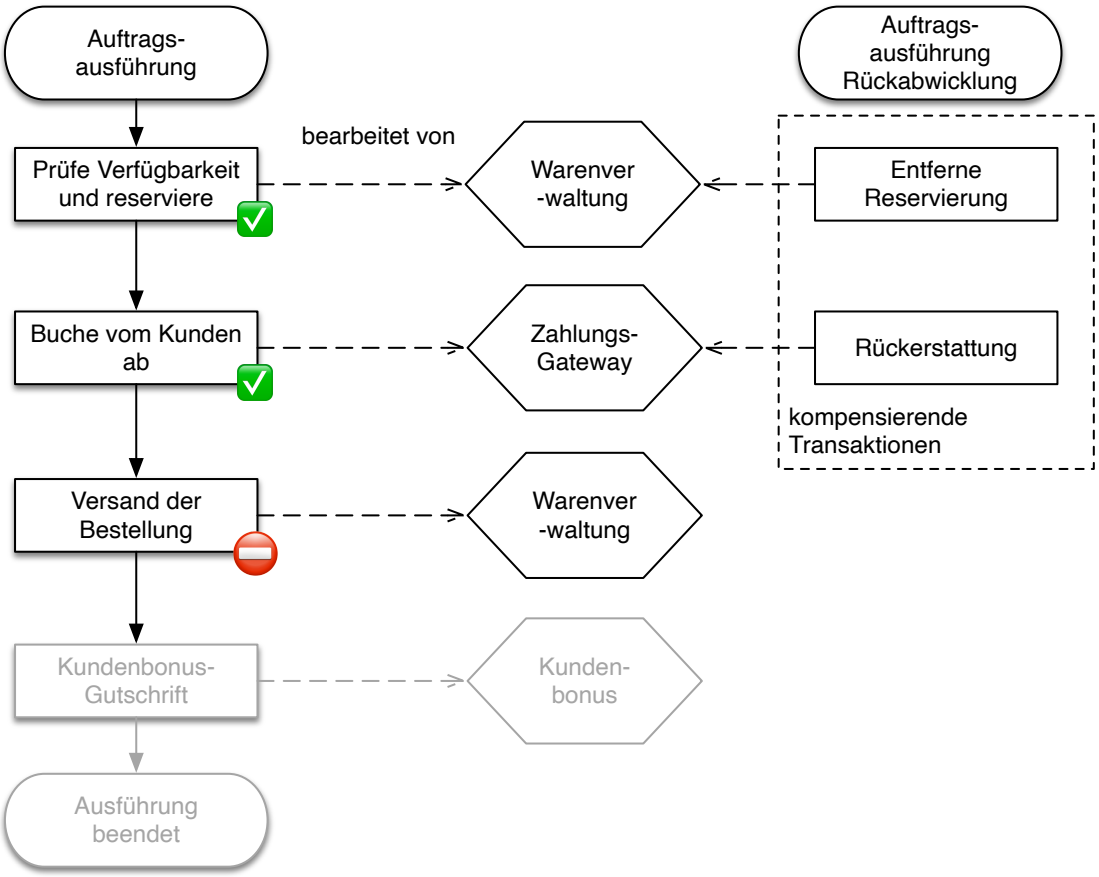


Eine *Saga* ist eine Sequenz von Aktionen, die ausgeführt werden, um eine langlebige Transaktion zu implementieren.

Sagas können keine Atomizität garantieren. Jedes System für sich kann jedoch ggf. Atomizität garantieren (z. B. durch die Verwendung traditioneller Datenbanktransaktionen).

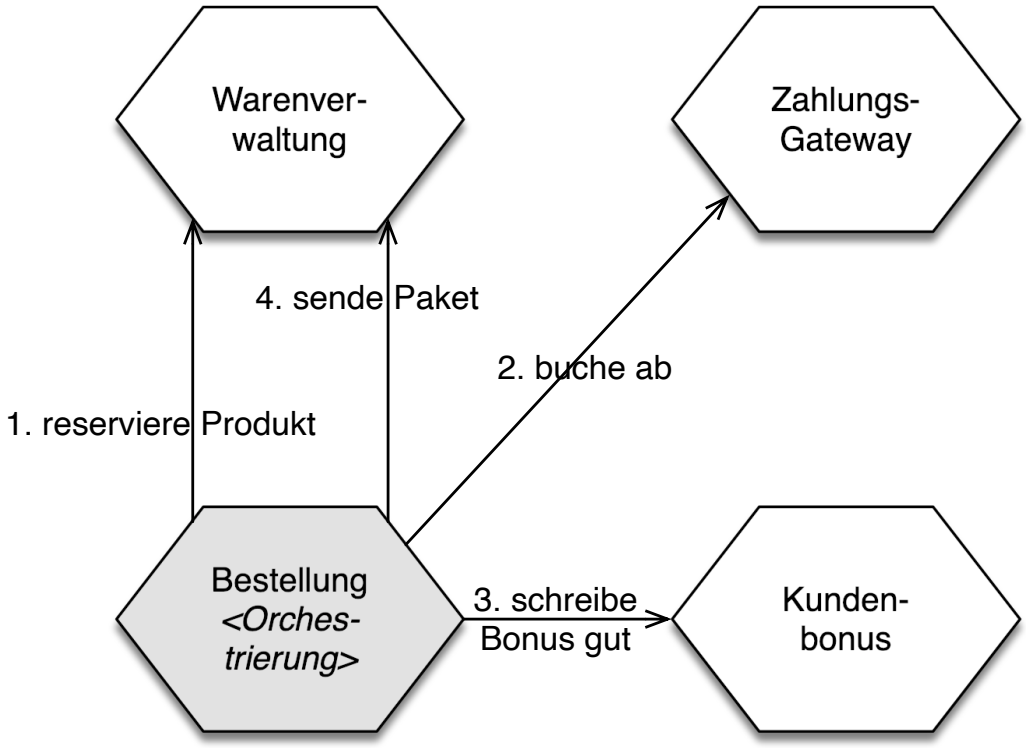
Sollte ein Abbruch der Transaktion notwendig sein, dann kann kein traditioneller *Rollback* erfolgen. Die Saga muss dann entsprechende kompensierende Transaktionen durchführen, die alle bisher erfolgreich durchgeführten Aktionen rückgängig machen.

Optimierung der Abarbeitungsreihenfolge zwecks Minimierung von mgl. *Rollbacks*



Die Abarbeitungsreihenfolge der Aktionen kann so optimiert werden, dass die Wahrscheinlichkeit von *Rollbacks* minimiert wird. In diesem Falle ist die Wahrscheinlichkeit, dass es zu einem *Rollback* während des Schritts „Versand der Bestellung“ kommt, wesentlich höher als beim Schritt „Kundenbonus gutschreiben“.

Langlebige Transaktionen mit orchestrierten Sagas



Die orchestrierte Saga ist eine Möglichkeit, um langlebige Transaktionen zu implementieren.

✓Mental einfach

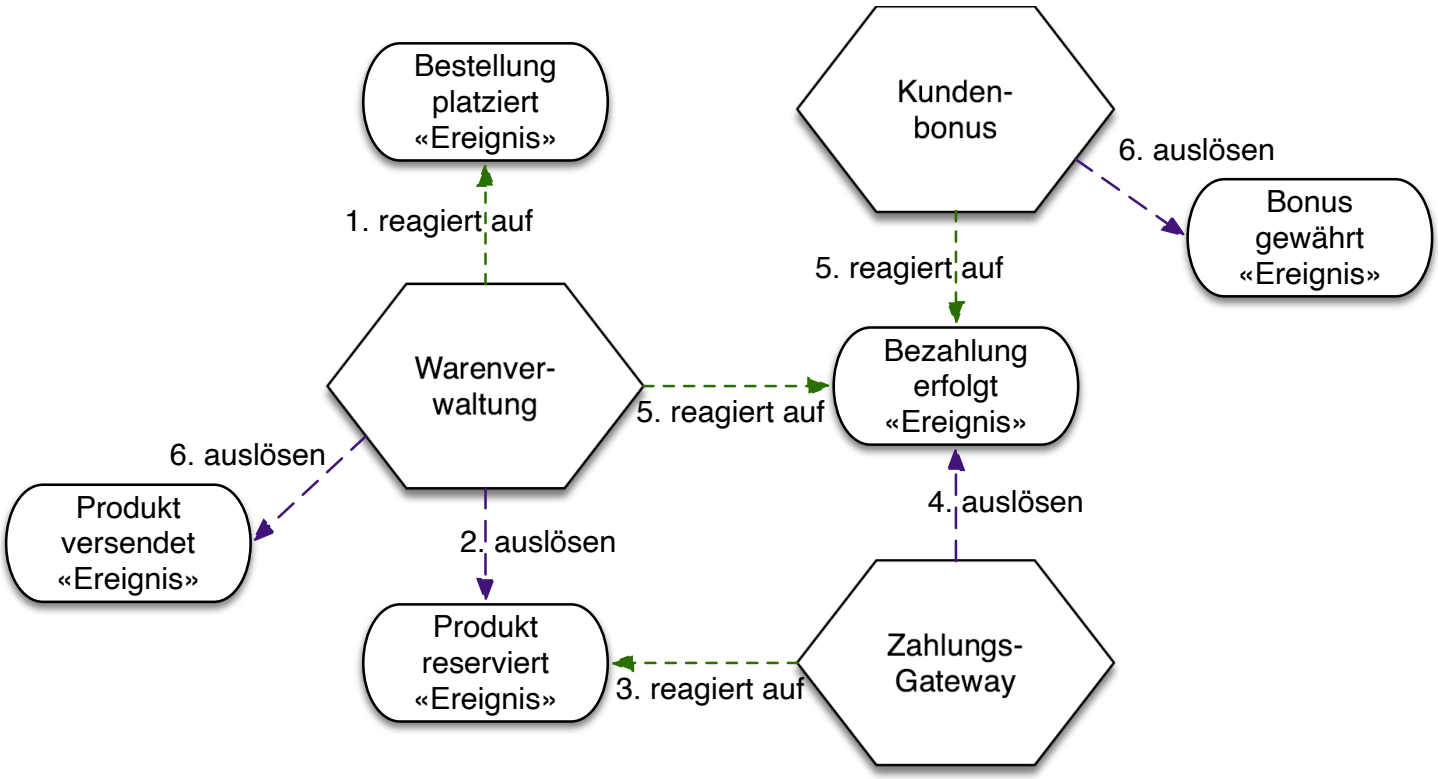
❗ Hoher Grad an *Domain Coupling*

Da es sich im Wesentlichen um fachlich getriebene Kopplung handelt, ist diese Kopplung häufig akzeptabel. Die Kopplung erzeugt keine technischen Schulden (*technical debt*).

❗ Hoher Grad an *Request-Response* Interaktionen

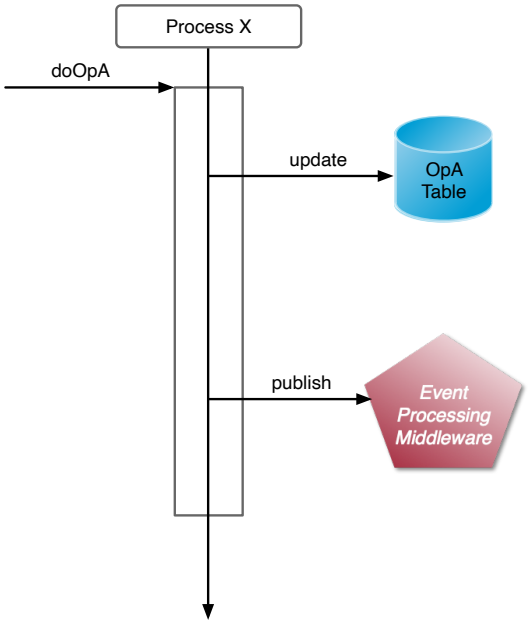
❗ Gefahr, dass Funktionalität, die besser in den einzelnen Services (oder ggf. neuen Services) unterzubringen wäre, in den Bestellung Service wandert.

Langlebige Transaktionen mit choreografierten Sagas



Ein großes Problem bei choreografierten Sagas ist es den Überblick über den aktuellen Stand zu behalten. Durch die Verwendung einer "Korrelations-ID" kann diese Problem gemindert werden.

Dual-write Problem



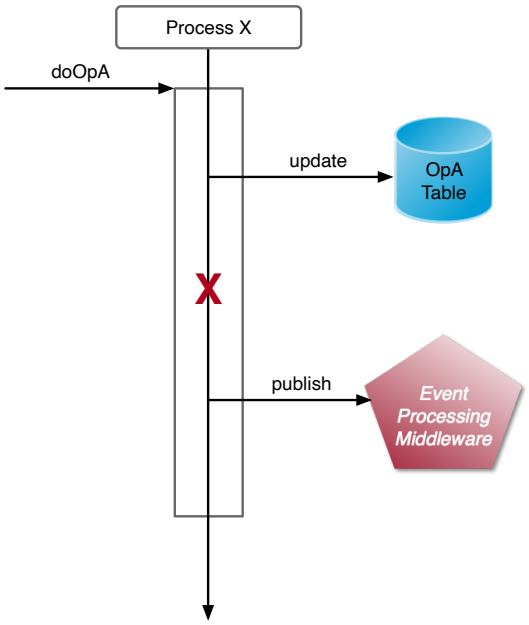
An welcher Stelle könnte es zu einem Problem kommen?

Warnung

Das „Schreiben“ auf zwei unterschiedliche Systeme (hier: Datenbank und Event-processing Middleware) erfordert immer einen transaktionalen Kontext.

Kann dieser nicht hergestellt werden, dann kann es zu Inkonsistenzen kommen (🇺🇸 *Dual-write Problem*).

1



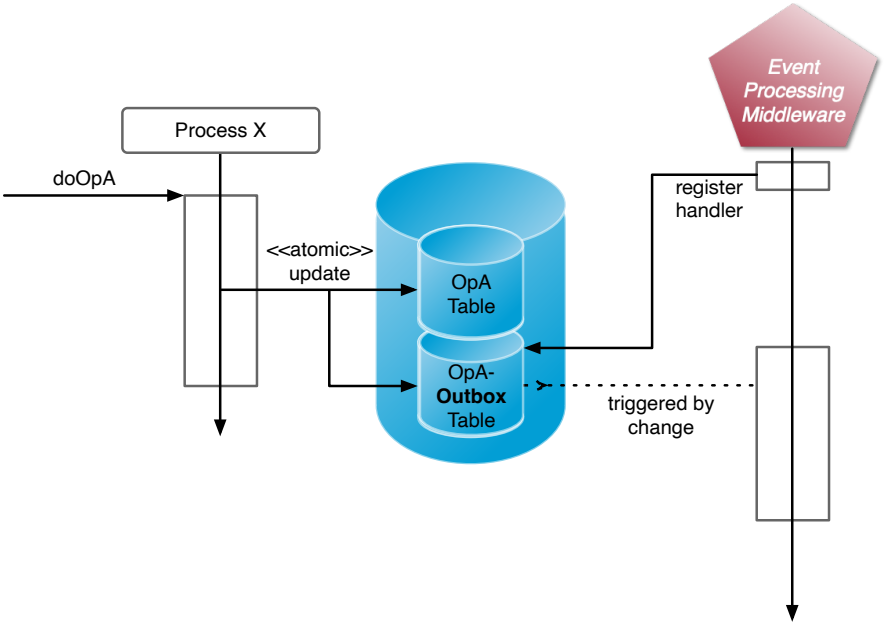
Lösungsideen

- ! 2PC ist im Kontext von Microservices keine Option (zu langsam, zu komplex)
- ! Änderung der Reihenfolge der Aktionen (1. *publish* dann 2. *update*) führt noch immer zu Inkonsistenzen
- ! die Event Processing Middleware (synchron) zu notifizieren - d. h. als Teil des Datenbankupdates - ist auch keine Option:
  - ! Was passiert, wenn die Middleware nicht erreichbar ist?
  - ! Was passiert, wenn das Event nicht verarbeitet werden kann?

Strikte Konsistenz ist nicht erreichbar.

2

Dual-write Problem - Outbox Pattern



(eine) Lösung: Outbox Pattern

- Die Aktionen werden (zusätzlich) in einer Outbox-Tabelle gespeichert und dann **asynchron** verarbeitet.
- Damit kann *Eventual Consistency* erreicht werden.

Die Wahl der Softwarearchitektur ist immer eine Abwägung von vielen Tradeoffs!

Weitere Aspekte, die berücksichtigt werden können/müssen:

- Cloud (und ggf. Serverless)
- Mechanical Sympathy
- Testen und Deployment von Mircoservices (Stichwort: *Canary Releases*)
- Monitoring und Logging
- Service Meshes
- ...

Literatur

[Newman2021] Sam Newman, **Building Microservices: Designing Fine-Grained Systems**, O'Reilly, 2021.