Architectures of Distributed Applications

A first overview.

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

Slides: https://delors.github.io/ds-architectures/folien.en.rst.html
https://delors.github.io/ds-architectures/folien.en.rst.html.pdf

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

Selected slides are based on slides by Maarten van Steen (*Distributed Systems*)

All errors are my own.

1. Basic architectures

Architectural Styles

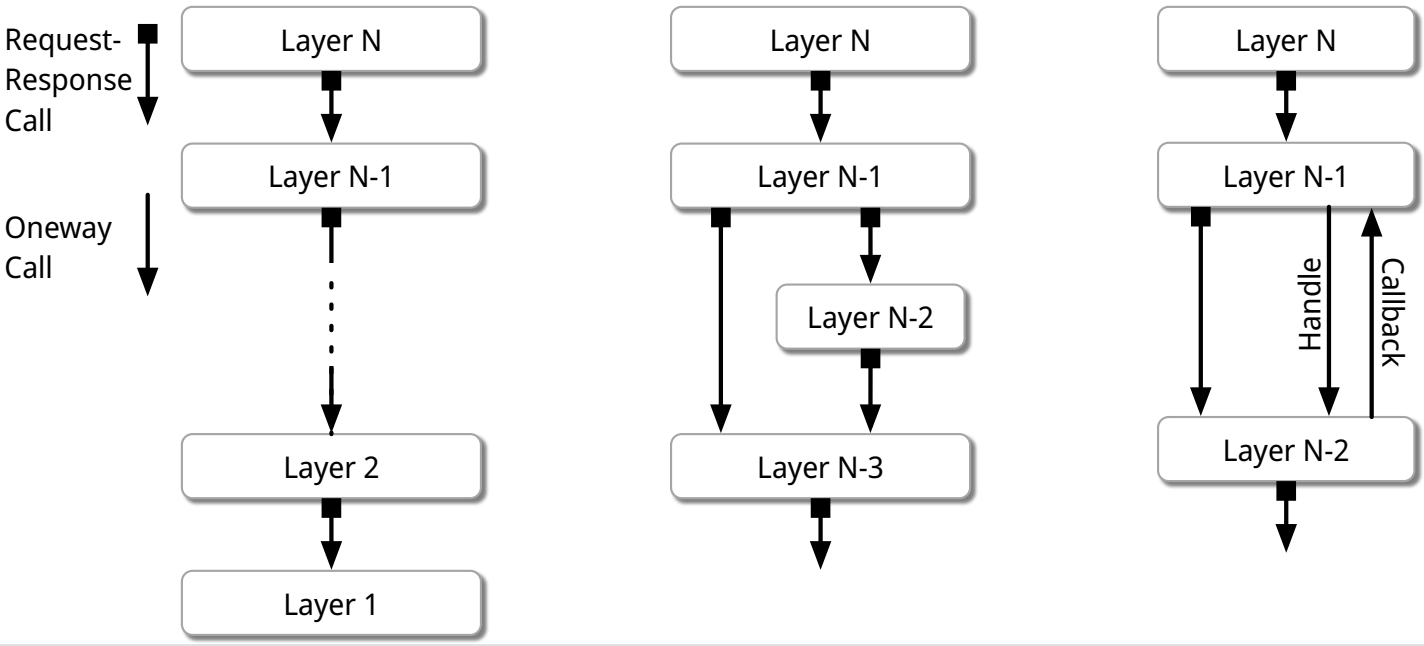
An architectural style is formulated in the form of

- (interchangeable) components with clearly defined interfaces
- the way in which the components are connected to each other
- the data exchanged between the components
- the way in which these components and connections are configured together to form a system System.

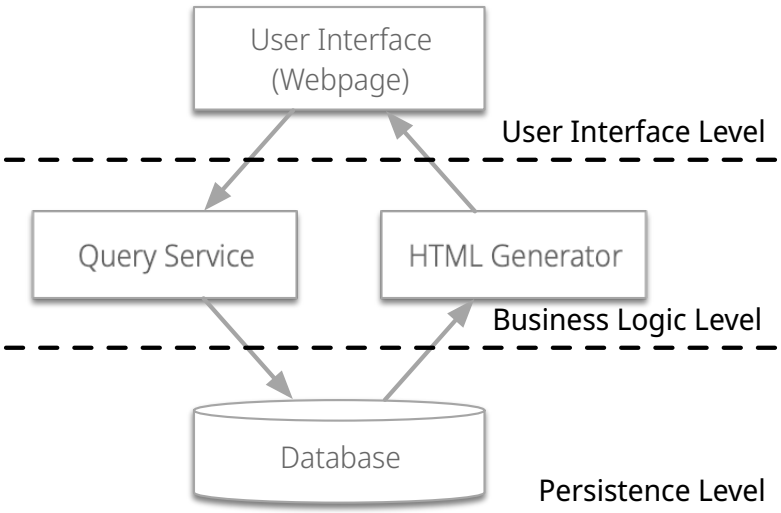
Connector

A mechanism that mediates communication, coordination or co-operation between components.
Example: Facilities for (remote) procedure calls (RPC), message transmission or streaming.

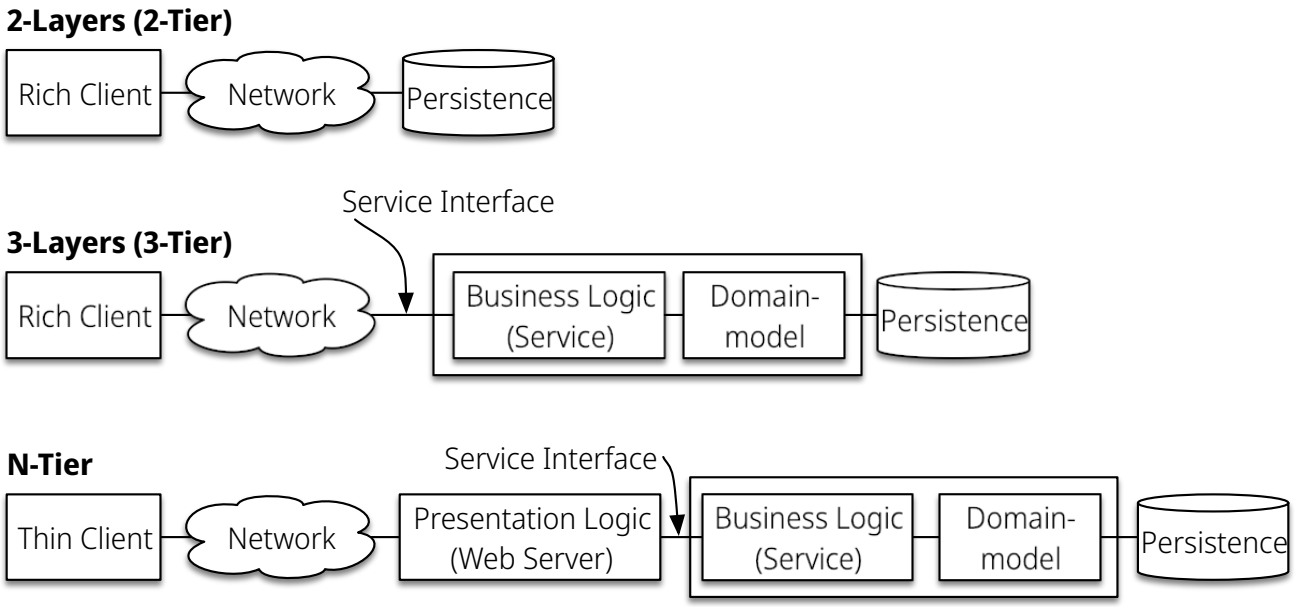
Layered Architectures



Example of a 3-tier Architecture



Traditional Architectures



Traditional 3-tier Architecture

This architecture can be found in many distributed information systems with traditional database technology and associated applications.

- The presentation layer represents the interface to users or external applications.
- The processing layer implements the business logic.
- The persistence/data layer is responsible for data storage.

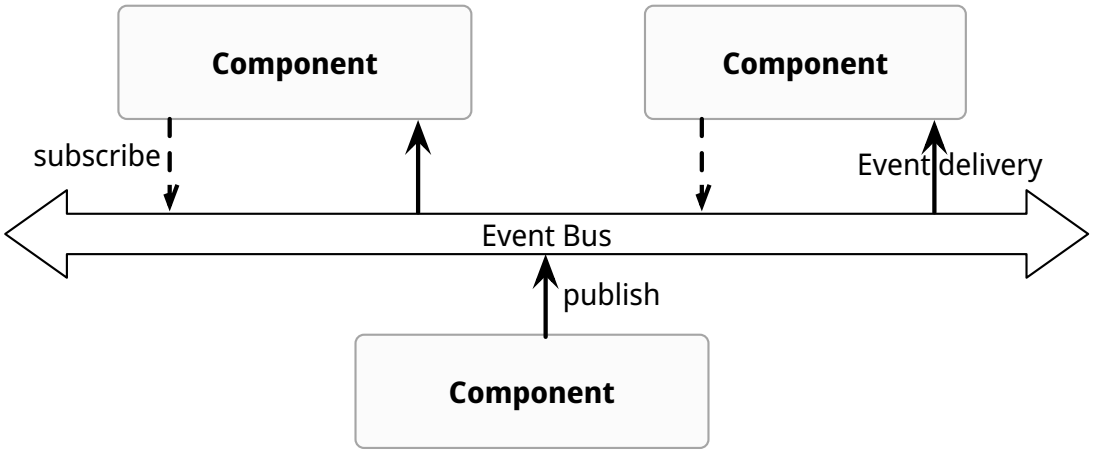
Publish and Subscribe Architectures

Dependencies between the components are realised using the *Publish and Subscribe* paradigm with the aim of loose coupling.

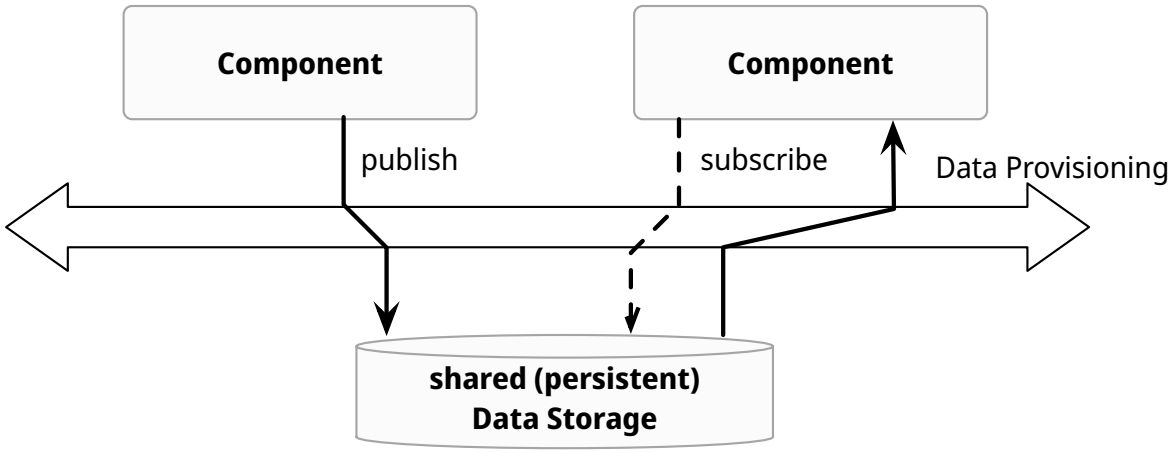
Taxonomy of coordination approaches with regard to communication and coordination:

	Coupled in time	Decoupled in time
Referentially coupled	Direct Coordination	Mailbox Coordination
Referentially decoupled	Event-based Coordination	Shared Data Space

Event-based Coordination



Shared Data Space



Event-based coordination in combination with *shared data space* is often used to realise publish and subscribe architectures.

Direct coordination

A process interacts directly (⇒ temporal coupling) with exactly one other well-defined process (⇒ referential coupling).

Mailbox coordination

The processes communicating with each other do not interact directly with each other, but via a unique mailbox (⇒ referential coupling). This means that the processes do not have to be available at the same time.

Event-based coordination

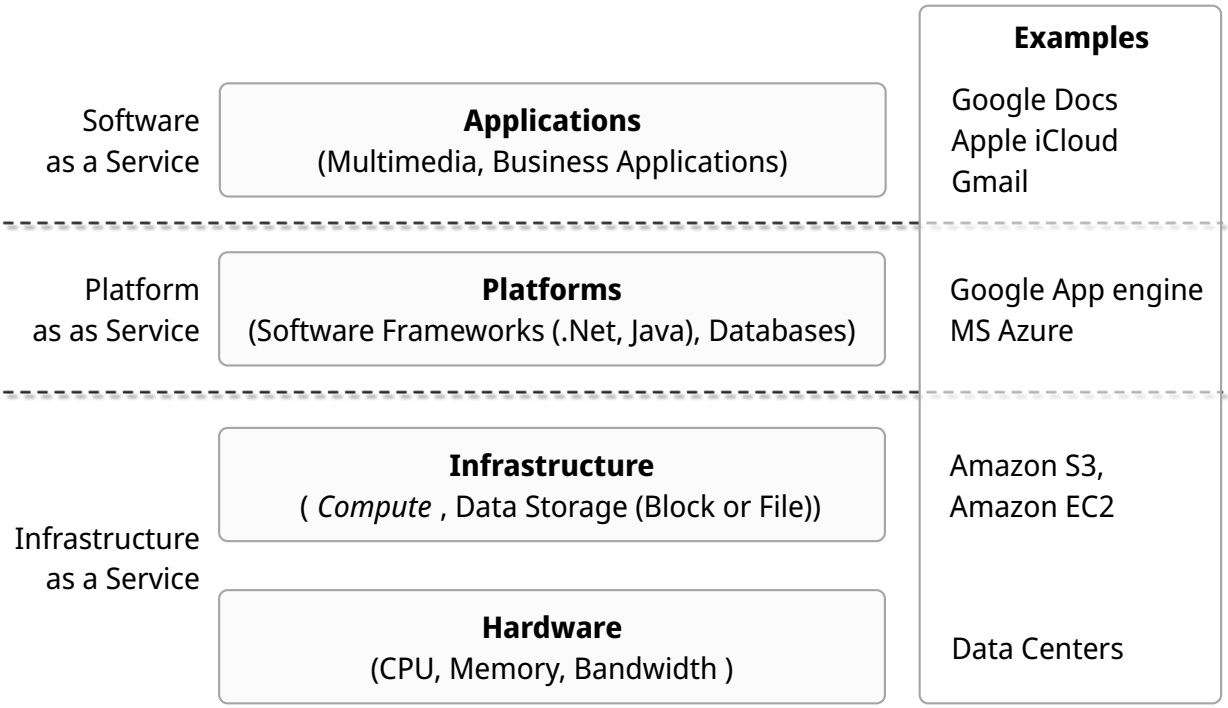
A process triggers events to which *any* other process reacts directly. A process that is not available

at the time the event occurs does not see the event.

Shared data storage

Processes communicate via tuples that are stored in a shared data space. A process that is not available at the time of writing can read the tuple later. Processes define patterns with regard to the tuples they want to read.

Structure of cloud computing applications



A distinction can be made between four layers:

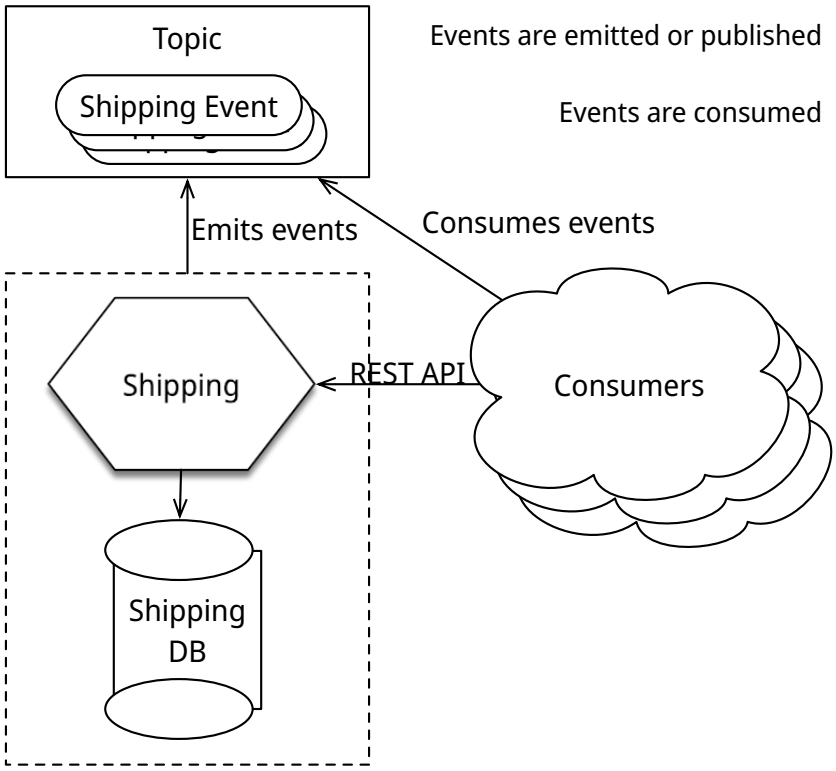
- Hardware: processors, routers, power supply and cooling systems.
  Normally completely transparent for customers.
- Infrastructure: Use of virtualization techniques for the purpose of allocating and managing virtual storage and virtual servers.
- Platforms: Provides higher level abstractions for storage and the like.
  Example: The Amazon S3 storage system provides an API for (locally created) files that can be organized and stored in so-called buckets.
- Application: Actual applications, such as office suites (word processing programmes, spreadsheet programmes, presentation applications).
  Comparable to the suite of applications that are delivered with operating systems.

2. Microservices [Newman2021]

Microservices

A simple microservice that offers a REST interface and emits events.

Where are the challenges?



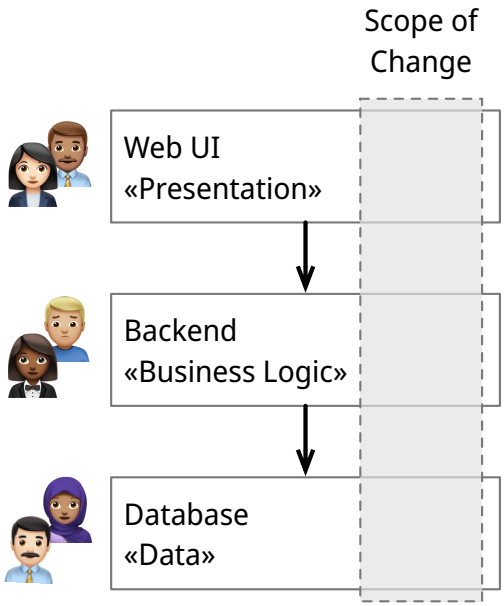
A major challenge is the design of the interfaces. To achieve true independence, the interfaces must be very well defined. If the interfaces are not clearly defined or inadequate, this can lead to a lot of work and coordination between the teams, which is actually undesirable!

Key Concepts of Microservices

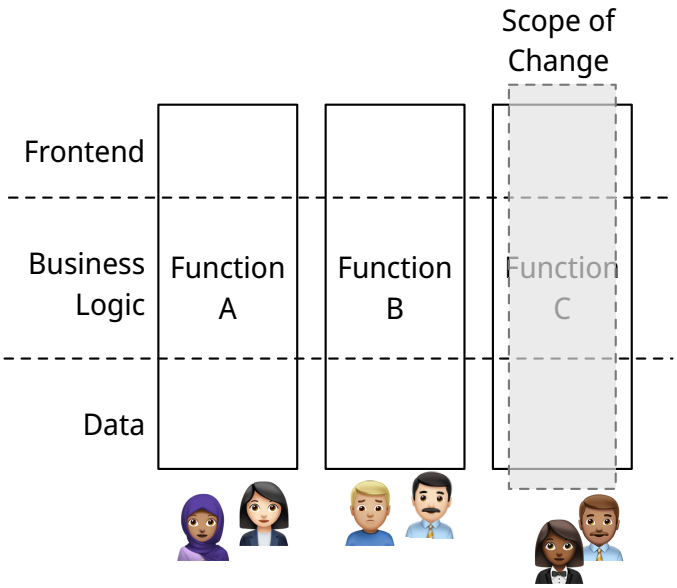
- can be deployed independently/are independently deployable
  (... and are developed independently.)
- model a business domain
  (Often along a bounded context or an aggregate determined using DDDs.)
- manage their own state
  (I.e. they have no shared database.)
- are small
  (Small enough to be developed by (max.) one team.)
- flexible in terms of scalability, robustness and the used technologies
- allow the architecture to be aligned with the organization (see Conway's Law)

Microservices and Conway's Law

Traditional Layered Architectures

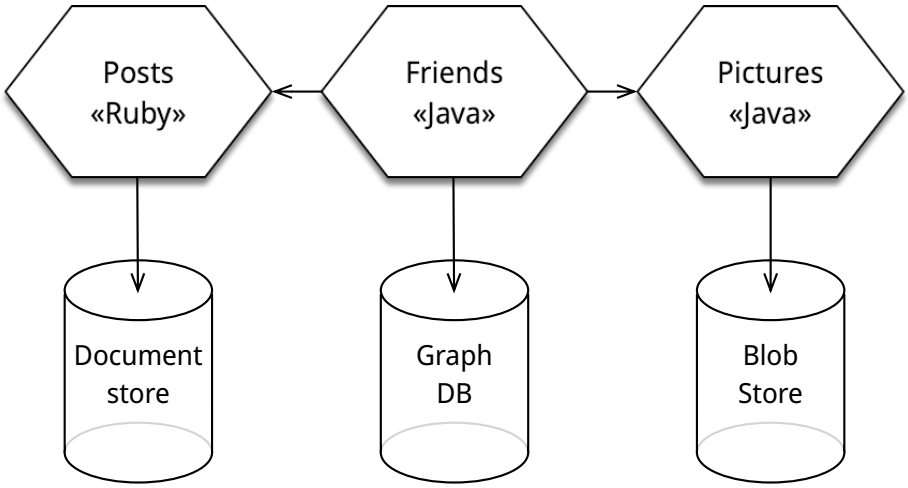


Microservices Architectures



Microservices and Usage of Technologies

Microservices are flexible with regard to the use of technology and enable the use of “the most suitable” technology.



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

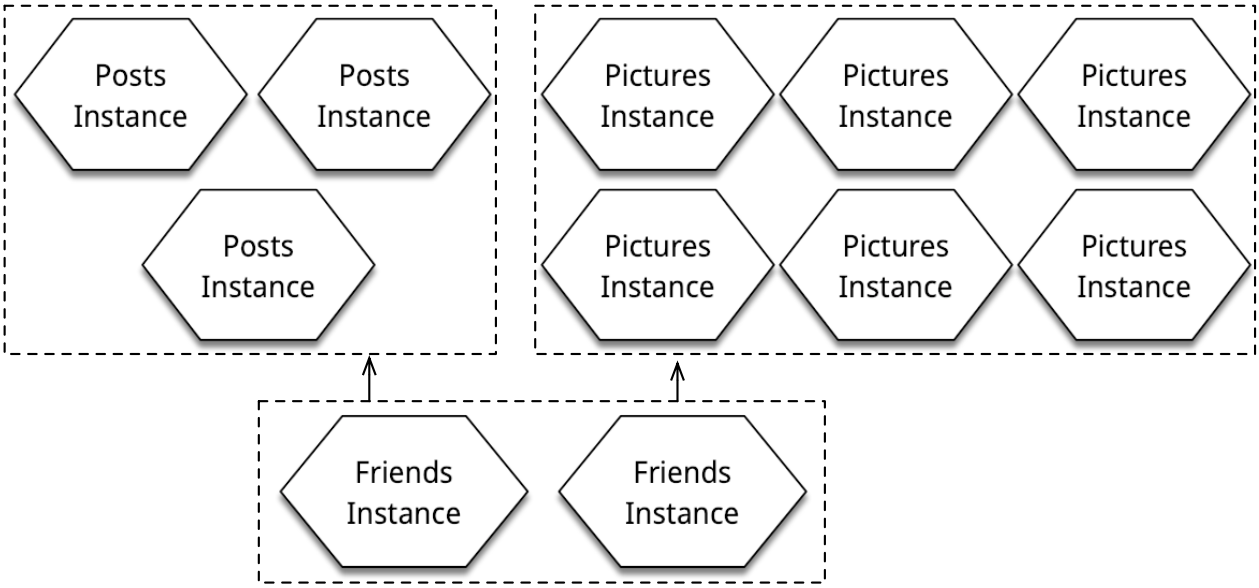
Quelle: TIOBE Programming Community Index - April 2012

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%

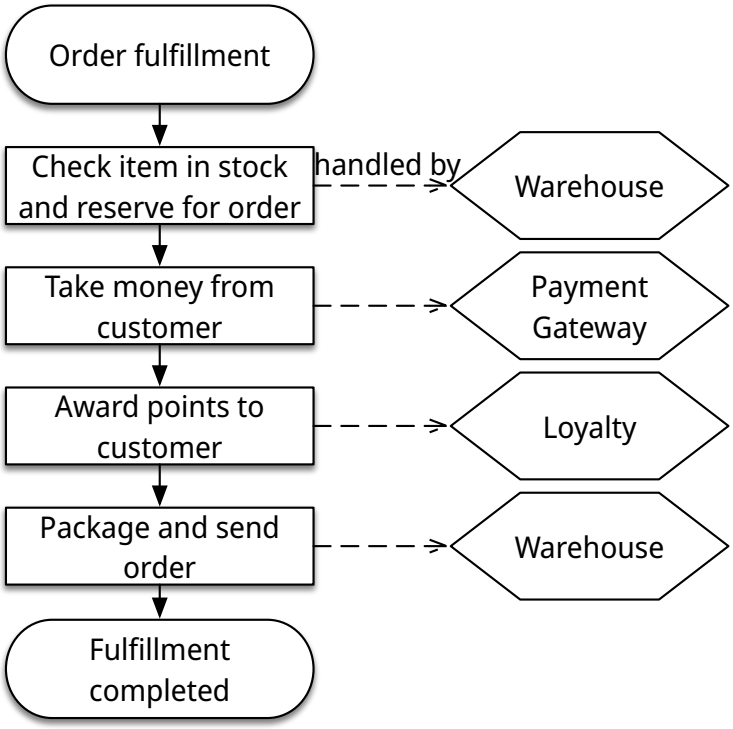
Quelle: TIOBE Programming Community Index - Feb. 2024

Microservices and Scalability

Well designed microservices can also be scaled very well.

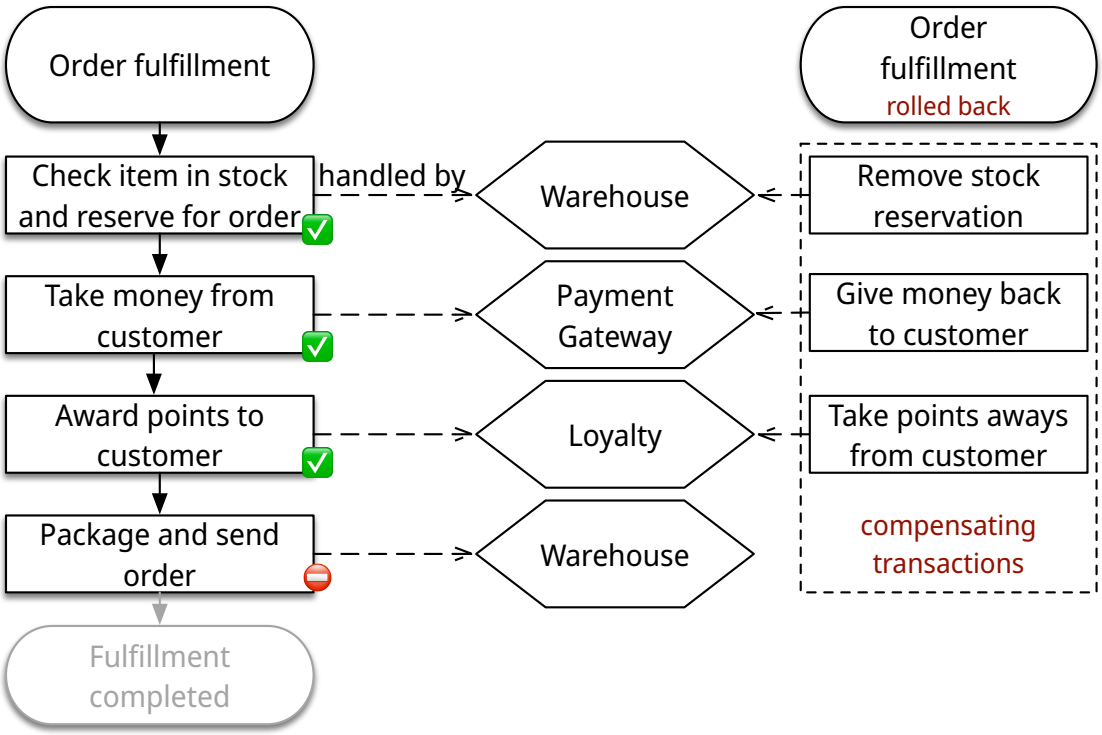


Implementation of a long-lived transactions?



The implementation of transactions is one of the biggest challenges in the development of microservices.

Using SAGAs for long-lived transactions

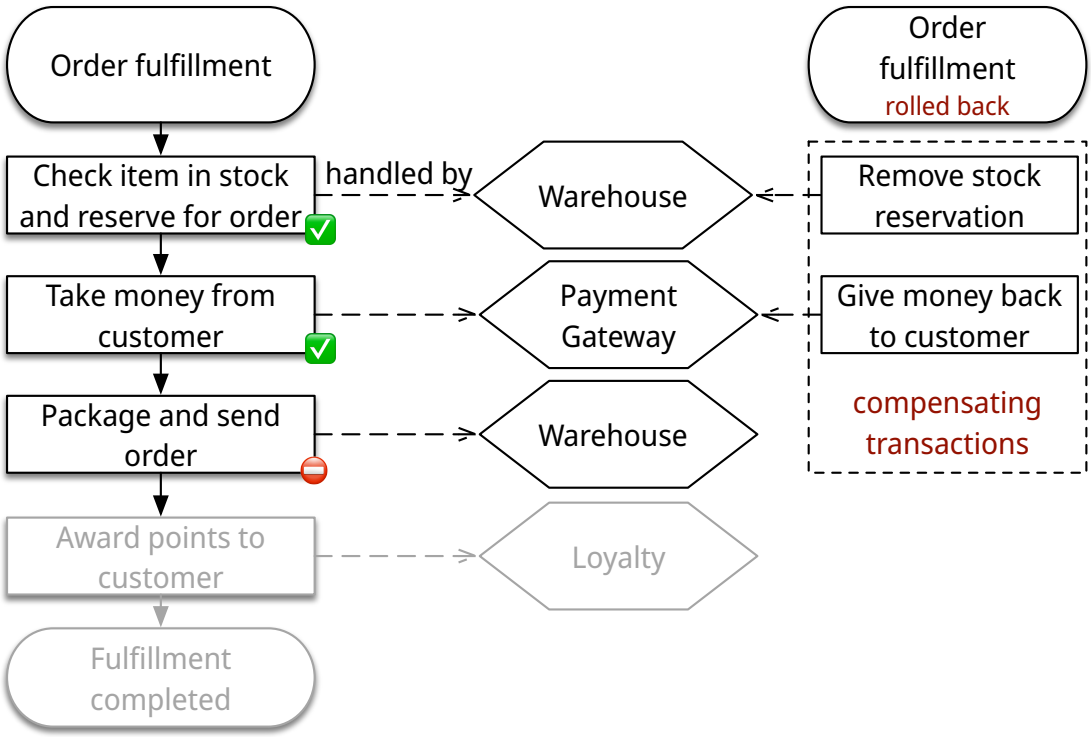


A *saga* is a sequence of actions that are executed to implement a long-lived transaction.

Sagas cannot guarantee atomicity. However, each system can guarantee atomicity (e.g. by using traditional database transactions).

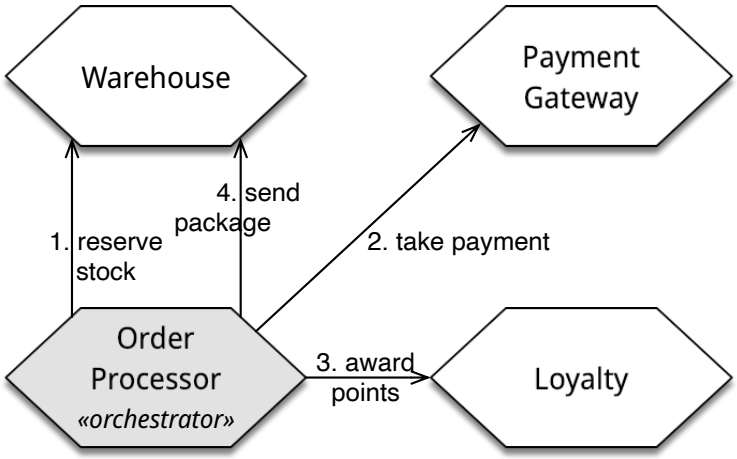
If the transaction needs to be aborted, a traditional *rollback* cannot be performed. The saga must then carry out the corresponding compensating transactions, which undo all previously successful actions.

Minimize the probability of possible *rollbacks*



The processing sequence of the actions can be optimized to minimize the probability of *rollbacks*.
In this case, the probability of a *rollback* occurring during the "package and send order" step is significantly higher than for the “award customer bonus” step.

Long-lived transactions with orchestrated sagas



The orchestrated saga is one way of implementing long-lived transactions.

✓Conceptually simple

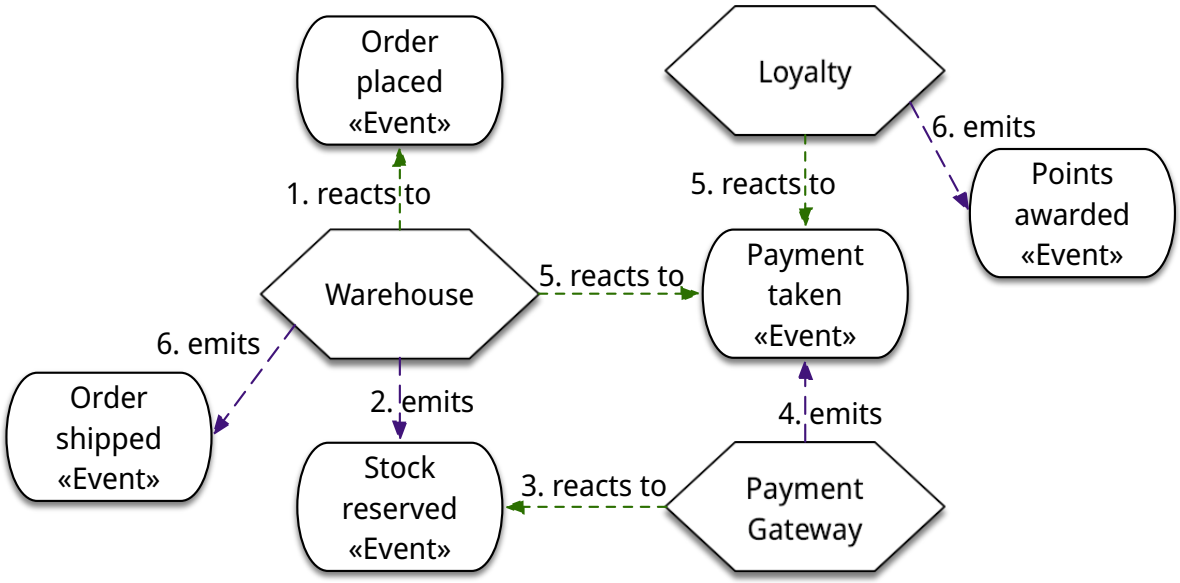
! High degree of *domain coupling*

As this is essentially domain-driven coupling, this coupling is often acceptable. The coupling does not generate any technical debt.

! High degree of *request-response* interactions

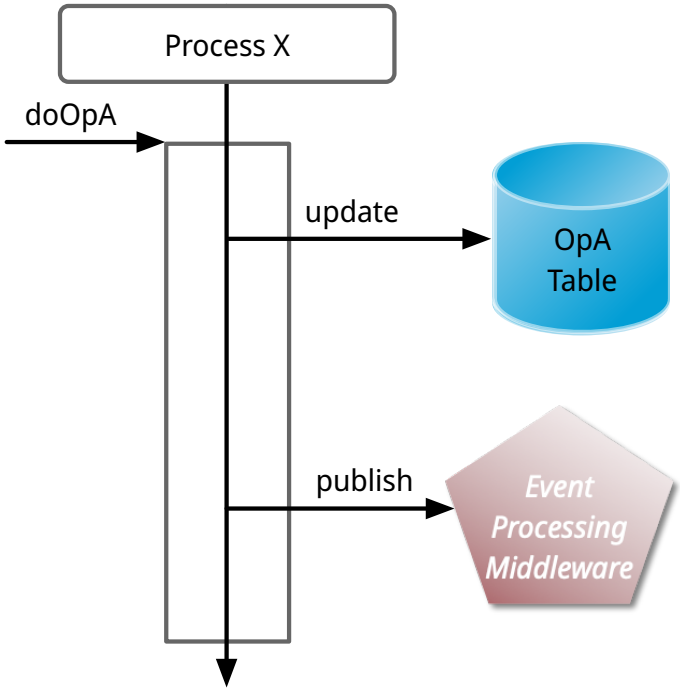
! Risk that functionality that would be better accommodated in the individual services (or possibly new services) is moved to the ordering service.

Long-lived transactions with choreographed sagas



A major problem with choreographed sagas is keeping track of the current status. This problem can be alleviated by using a “correlation ID”.

Dual-write Problem

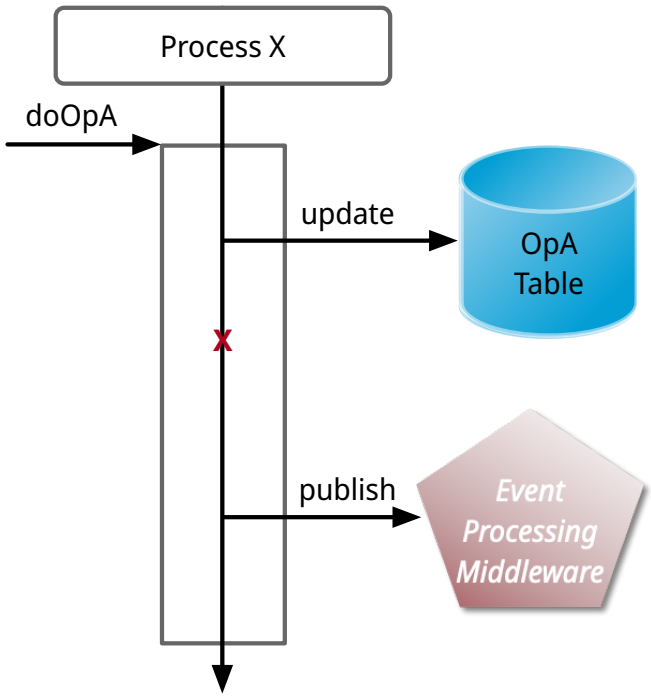


Where could there be a problem?

Warning

Writing to two different systems (here: database and event-processing middleware) always requires a transactional context.

If this cannot be established, inconsistencies can occur (*dual-write problem*).

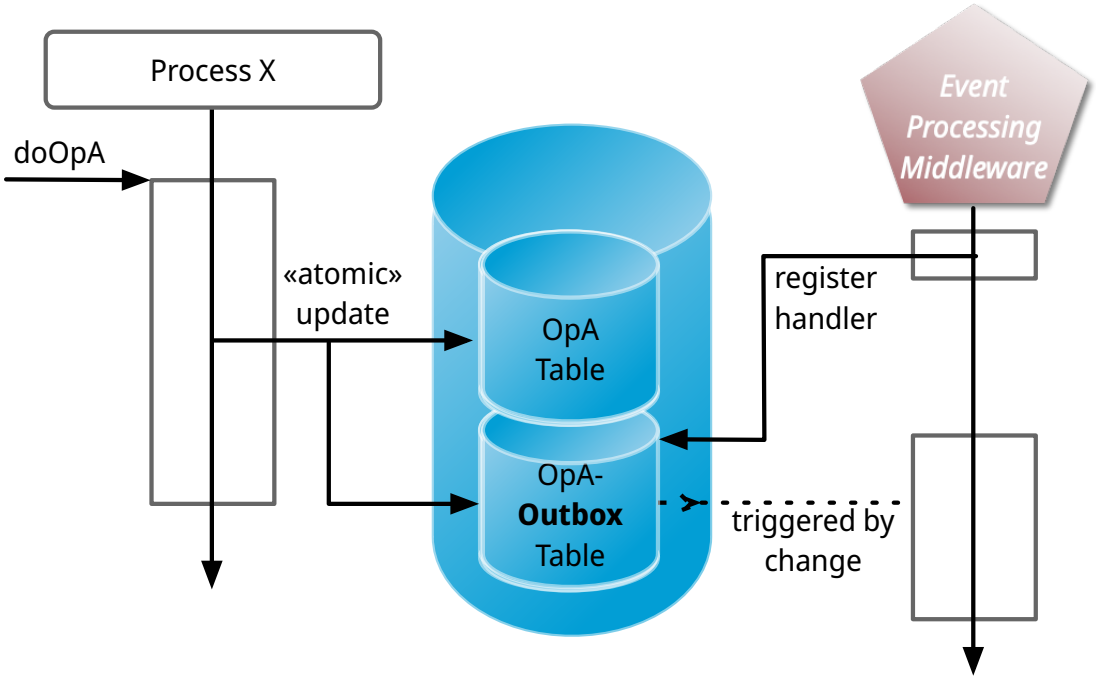


Solution Ideas

- ! 2PC is not an option in the context of microservices (too slow, too complex)
- ! Changing the order of actions (1st *publish* then 2nd *update*) still leads to inconsistencies
- ! notifying the event processing middleware (synchronously) - i. e. as part of the database update - is also not an option:
  - ! What happens if the middleware cannot be reached?
  - ! What happens if the event cannot be processed?

Strict consistency cannot be achieved.

Dual-write Problem - Outbox Pattern



(a) Solution: Outbox Pattern

- The actions are (additionally) saved in an outbox table and then processed **asynchronously**.
- This enables *eventual consistency* to be achieved.

The choice of software architecture is always a consideration of many trade-offs!

Other aspects that can/must be considered:

- Cloud (and possibly serverless)
- Mechanical Sympathy
- Testing and deployment of microservices (keyword: *Canary Releases*)
- Monitoring and logging
- Service meshes
- ...

Literature

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