

# **Enterprise Architektur-Muster**

**Modul “Software Engineering” (Prof. Dr. Andreas Both, Wi-Se 2024/2025)  
an der HTWK Leipzig**

## Motivation: Anwendungsfall E-Commerce I

- Ziel: Backend für internationales E-Commerce-System
- MVP: Bestellungen, Bezahlung und Versand
- Zukünftig viele Nutzer und hoher Traffic erwartet
- Geringes Kapital für Infrastruktur
- Rechtliche Regularien teilweise unklar, weil international
- Hohe Sicherheitsanforderungen
- Agiles Team von acht fähigen Entwicklern
- Geldgeber wollen erste Auslieferung in zwei Wochen

## Motivation: Anwendungsfall E-Commerce II



Abbildung 1: Sequenzdiagramm zum Aufgeben einer Bestellung

# Klassische Enterprise-Architektur

## Service-oriented Architecture

- Bisher: Eng gekoppelte Komponenten, die nicht isoliert voneinander verwendet werden können
- Jetzt: Architekturmuster, die den Fokus auf lose gekoppelte Dienste liegt
- Service-oriented Architecture (SOA) ermöglicht die Wiederwendung von Funktionalitäten

## Service-oriented Architecture: Komponenten

- Service Provider: Bietet einen spezifischen Dienst an
- Broker: Dienst, das die Kommunikation zwischen Service Consumer und Service Provider gewährleistet
- Service Consumer: Nutzt einen bereitgestellten Dienst, dieses kann einen End-User oder ein anderer Dienst sein
- Service Registry: Sammlung von Metadaten zu Services und deren Provider

## Service-oriented Architecture: Struktur

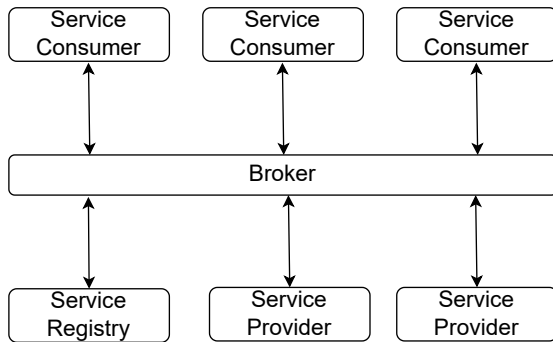


Abbildung 2: Aufbau der Service-oriented Architecture

## Service-oriented Architecture: Beispiel E-Commerce I

- `OrderService`: Dienst für Verwaltung von Bestellungen
- `PaymentService`: Dienst verantwortlich für die Abwicklung von Zahlungen
- `ShipmentService`: Dienst zuständig für den Versandprozess



## Service-oriented Architecture: Beispiel E-Commerce II

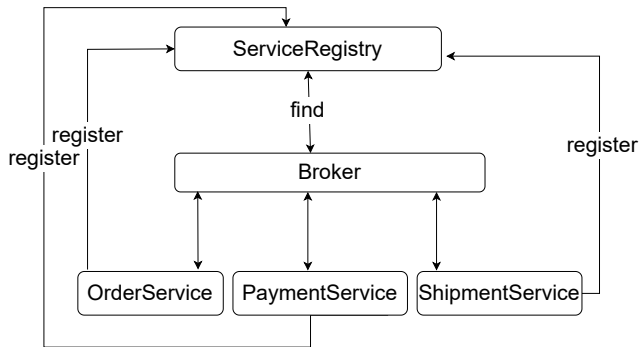


Abbildung 3: E-Commerce-Beispiel mit Service-oriented Architecture

## Service-oriented Architecture: Agilität

- Lose Kopplung ermöglicht die unabhängige Entwicklung von Services, wodurch Teams autonom arbeiten und die parallele Entwicklung gefördert wird
- Dank loser Kopplung und autonomer Teams können Kundenanforderungen schnell umgesetzt werden
- Zeit- und Kosteneinsparungen durch die Wiederverwendung von Services
- Nachteil: Langfristig könnten sich Abhängigkeiten zwischen den Services entwickeln

# Moderne Enterprise-Architektur

## Microkernel Architecture

- Aufteilung der Anwendung in zwei Komponenten [4]
- Kern:
  - Minimale Funktionalität
  - Bietet Schnittstelle für Erweiterungen/Plugins
  - Enthält Plugin-Registry
- Module:
  - Erweitern Kern um Funktionalitäten
  - Kommunikation über definierte Schnittstellen
  - Lose gekoppelt, unabhängig und isoliert voneinander
  - Verbindung über verschiedene Wege möglich (REST, Messaging, Objekt Instanziierung, ...)

## Microkernel Architecture: Struktur

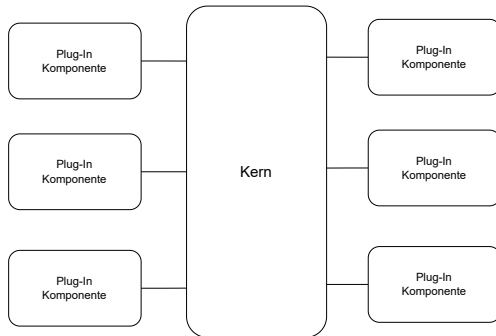


Abbildung 4: Aufbau einer Microkernel Architecture

## Microkernel Architecture: Beispiel E-Commerce I

- Kern: Großteil der Funktionalität
- Module: Auslagerung von Business-Logik zum Bezahlen und Versenden
- Einfache Erweiterbarkeit durch neue Dienstleister
- Komplexer Kern, hohe Kopplung der übrigen Funktionalitäten
- Als einziges Architekturmuster eher nicht geeignet, jedoch in Kombination mit anderen sinnvoll

## Microkernel Architecture: Beispiel E-Commerce II

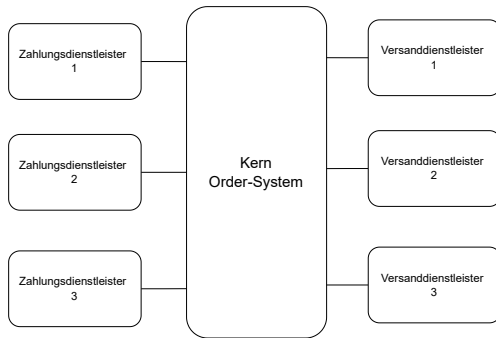


Abbildung 5: E-Commerce-Beispiel mit Microkernel Architecture

## Microkernel Architecture: Agilität

- Lose Kopplung der Module  $\Rightarrow$  schnelle Reaktionsfähigkeit auf Änderungen
- Module können einfach ausgetauscht werden  $\Rightarrow$  geringe Downtime
- Einfach testbar, da Module unabhängig und isoliert voneinander
- Kurze Iterationen und Auslieferungszeiten bei Erweiterung der Anwendung
- Aber: Hoher initialer Aufwand durch teuren Kern, eher hohe Time-to-Market
- Aber: Hoher Aufwand, wenn Kern später Anpassungen benötigt
- Fazit: In ausgewählten Anwendungsfällen sinnvoll, aber nicht universell in agilen Umgebungen einsetzbar



## Event-Driven Architecture

- Bisher: Expliziter Aufruf von Funktionalitäten
- Jetzt: Impliziter Aufruf durch Reaktion auf Ereignisse [2]
- System reagiert asynchron auf Zustandsänderung (Ereignis in System)
- Alte Idee: David Garlan und Mary Shaw, 1994, *An Introduction to Software Architecture*

## Event-Driven Architecture: Komponenten

- Event: Kapselt Information einer Zustandsänderung eines Systems [3]
- Produzent: Erzeugt Event
- Publisher: Publiziert erzeugtes Event
- Konsument: Reagiert auf Event
- Mediator: Vermittler zwischen Produzenten und Konsumenten
- Event-Broker: Infrastruktur für Gesamtheit der Vermittler

## Event-Driven Architecture: Struktur

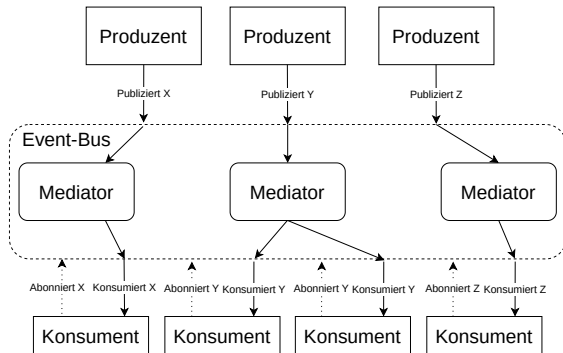


Abbildung 6: Vertrag zwischen Produzenten und Konsumenten am Event-Bus

## Event-Driven Architecture: Beispiel E-Commerce I

- OrderCreated: Genau dann, wenn Bestellung aufgegeben wird
- PaymentProcessed: Genau dann, wenn Bezahlvorgang abgeschlossen wird
- ShipmentInitiated: Genau dann, wenn Bestellung versandt wird
- Event-Kette: OrderCreated → PaymentProcessed → ShipmentInitiated
- Implementierung in Diensten: OrderService, PaymentService, ShipmentService

## Event-Driven Architecture: Beispiel E-Commerce II

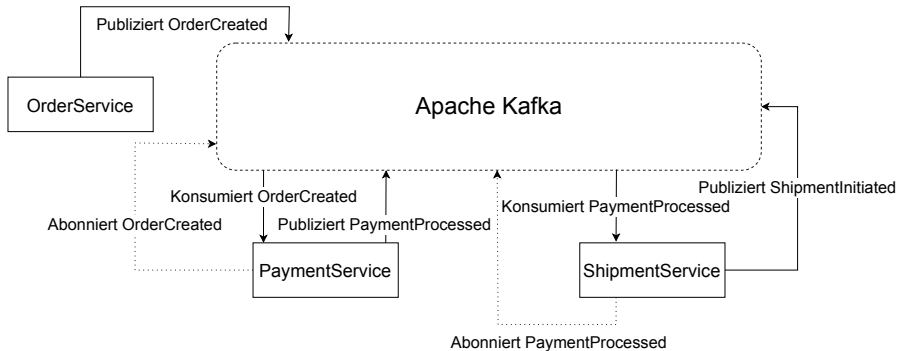


Abbildung 7: E-Commerce-Beispiel mit Event-Driven Architecture

## Event-Driven Architecture: Agilität

- Event ist Vertrag zwischen Produzent und Konsument am Event-Broker  
⇒ Hohe Kohäsion ⇒ Lose Kopplung
- Feature: Menge von Events, deren Produzenten und Konsumenten  
⇒ Klare Abgrenzung ⇒ einfach definierbare Iterationen
- Events sind sehr realitätsnah - domain-driven
- Sehr hohe Flexibilität & maximale Skalierung durch lose Kopplung
- Schnelle Auslieferung, kurze Intervalle
- Exzellente Kombination mit Microservices & Cloud-Integration
- Aber: Erhöhte Komplexität ⇒ Hohe Anforderungen an Entwickler

## Cloud-Native Architecture

- Bisher: Vorab-Allokation von Ressourcen
- Jetzt: Allokation genau dann, wenn notwendig
- Cloud-Native: Explizit für die Cloud entwickelte Applikationen [1]
- Annahme: Infrastruktur ist in ständigem Wandel
- Folgerung: Infrastruktur auslagern - an Cloud-Vendor
  - Globale Nutzung durch Geo-Redundanz: Starke Verteilung und hohe Verfügbarkeit
  - Auto-Scaling: Dynamische Skalierung basierend auf Nachfrage
  - Pay-as-you-go, Scale-to-zero: Nur verwendete Ressource wird bezahlt
  - Zero Downtime

## Cloud-Native Architecture: Technologie

- Containerisierung: Jede Komponente eines Systems ist Container
- Dynamische Orchestrierung: Aktives Container-Management zur Ressourcenoptimierung
- Microservice-Architektur ist Fundament - ergänzt durch Cloud-Dienste
- Fully Managed Cloud-Services: Business-Logik statt Infrastruktur



## Cloud-Native Architecture: Beispiel E-Commerce

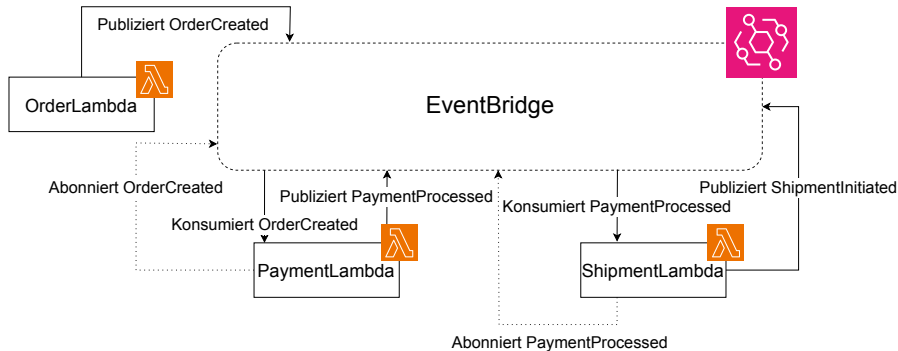


Abbildung 8: E-Commerce-Beispiel mit Cloud-Native Architecture in AWS

## Cloud-Native Architecture: Agilität

- Alle agilen Vorteile von Microservice- und Event-Driven Architecture
- Fokus auf Business-Logik & einfaches Deployment  $\Rightarrow$  Kurze Iterationen
- Maximale Flexibilität für Nachfrage durch Auto-Scaling
- Finanzielle Agilität: Pay-as-you-go
- Aber: Kostenrisiken durch Auto-Scaling und Pay-as-you-go
- Achtung: Vendor-Lock-In durch proprietäre Fully Managed Cloud-Services

# Zusammenfassung

— Bar

## Literatur I

- [1] Dennis Gannon, Roger Barga und Neel Sundaresan. "Cloud-Native Applications". In: *IEEE Cloud Computing* 4.5 (2017), S. 16–21. DOI: 10.1109/MCC.2017.4250939.
- [2] David Garlan und Mary Shaw. *An Introduction to Software Architecture*. Techn. Ber. CMU/SEI-94-TR-021. Accessed: 2025-Jan-2. Jan. 1994. URL: <https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/>.
- [3] Ramakrishna Manchana. "Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries". In: *International Journal of Science and Research (IJSR)* 10 (Jan. 2021), S. 1706–1716. DOI: 10.21275/SR24820051042.
- [4] Mark Richards. *Software Architecture Patterns*. O'Reilly Media, Inc., 2015,