

Probeklausur

Architektur von Anwendungssystemen

Universität: Technische Universität Berlin
Kurs/Modul: Architektur von Anwendungssystemen
Bearbeitungszeit: 120 Minuten
Erstellungsdatum: September 19, 2025



Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

<https://study.AllWeCanLearn.com>

Architektur von Anwendungssystemen

Aufgabe 1.

(a) Beschreiben Sie eine verteilte Anwendung im Kontext einer E-Commerce-Plattform. Nennen Sie typische Schichten (Präsentation, Domäne, Integrationslayer, Persistenz) und zentrale Komponenten innerhalb der Schichten. Erläutern Sie, welche Middleware-Komponenten sinnvoll eingesetzt werden (API-Gateway, Nachrichten-Broker, Konfigurations-Server) und welche Aufgaben sie übernehmen.

(b) Welche Architekturstile eignen sich für verteilte Anwendungen in diesem Kontext? Skizzieren Sie die Unterschiede zwischen Monolith, mehrschichtiger Architektur (N-Tier) und Microservice-Architektur sowie Event-getriebener Architektur. Diskutieren Sie, unter welchen Rahmenbedingungen welcher Stil bevorzugt wird.

(c) Erläutern Sie, wie asynchrone Kommunikation in einer verteilten Architektur zu Anpassungen bei Konsistenz, Fehlertoleranz und Transaktionsmanagement führt. Nennen Sie zwei Muster der verteilten Architektur und beschreiben Sie deren Zweck (Beispiele: Event Sourcing, CQRS).

Aufgabe 2.

- (a) Vergleichen Sie Messaging-Broker-Systeme mit Streaming-Plattformen. Nehmen Sie RabbitMQ bzw. einen typischen Messaging-Broker einerseits und Kafka als Streaming-Plattform andererseits als Beispiele. Diskutieren Sie Unterschiede in Latenz, Liefergarantien, Ordering, Skalierbarkeit und Fehlertoleranz sowie typische Einsatzszenarien.
- (b) Entwickeln Sie einen Kriterienkatalog zur Auswahl von Web-Technologien für Service-Interfaces (REST, gRPC, GraphQL). Berücksichtigen Sie Aspekte wie Performance, Typensicherheit, Interoperabilität, Tooling, Sicherheit, Observability und Wartbarkeit.
- (c) Sicherheit und Governance in verteilten Architekturen. Beschreiben Sie Grundprinzipien der Zugriffskontrolle (RBAC/RBAC-ähnlich), Authentifizierung, Autorisierung, Audit-Logging und zentrale Governance-Aspekte.

Aufgabe 3.

- (a) Beschreiben Sie eine event-getriebene Architektur für einen Benachrichtigungsdienst. Definieren Sie Ereignistypen (z. B. UserRegistered, OrderShipped), das Topic bzw. Thema, das Schema der Ereignisse sowie die Consumer-Gruppen. Erklären Sie, wie Idempotenz bei der Verarbeitung von Ereignissen sichergestellt wird.
- (b) Beschreiben Sie die Rolle von API-Gateway, Service-Mesh und Service-Discovery in einer Microservice-Architektur und erläutern Sie, wie diese Komponenten zusammenarbeiten, um Zuverlässigkeit, Observability und Sicherheit zu unterstützen.
- (c) Welche verteilten Muster unterstützen Robustheit und Skalierbarkeit? Diskutieren Sie Konzepte wie Retries, Circuit Breaker, Fallbacks und das Verhältnis von Konsistenz zu Verfügbarkeit im Kontext der Architektur.

Aufgabe 4.

- (a) Eine Unternehmensanwendung soll skalierbar und hochverfügbar betrieben werden. Skizzieren Sie eine Referenz-Architektur mit Frontend, Backend-Services, Datenhaltung, Messaging-Komponenten und Observability. Beschreiben Sie, wie Dienste, Datenbanken und Messaging zusammenwirken und welche Muster für Skalierung und Ausfallsicherheit eingesetzt werden.
- (b) Beurteilen Sie die Architektur hinsichtlich Skalierbarkeit, Verfügbarkeit und Sicherheit. Welche Metriken und Beobachtungsinstrumente würden Sie verwenden? Diskutieren Sie potenzielle Engpässe und Strategien zu deren Beseitigung.
- (c) Migrationspfad: Skizzieren Sie einen schrittweisen Plan von einer monolithischen Anwendung hin zu einer service-orientierten oder mikroservice-basierten Architektur. Beschreiben Sie Phasen, Risikominimierung, Strangler-Pattern und Governance-Aspekte.

Lösungen

Aufgabe 1.

Lösung 1.(a).

- Schichten einer verteilten E-Commerce-Anwendung:
 - *Präsentation*: Web- bzw. Mobile-Frontend, meist zustandslos, Rendering-Logik meist auf der Client-Seite oder im Frontend-Service.
 - *Domäne (Business-Logik)*: Domänenmodelle und Services, die zentrale Geschäftsregeln implementieren; oft als eigenständige Dienste oder Module abgegrenzt (Bounded Contexts).
 - *Integrationslayer*: Schnittstellenwege zwischen Domäne und Persistenz; API-Gateway, Orchestrierung, Event-basierte Kommunikation, Saga- oder Orchestrations-Mechanismen.
 - *Persistenz*: Datenhaltung je nach Kontext (relationale DBs, NoSQL, Event-Store, Cache). In verteilten Architekturen oft polyglot persistiert, d.h. pro Dienst eigene DB.
- Zentrale Middleware-Komponenten und deren Aufgaben:
 - *API-Gateway*: Zentraler Einstiegspunkt für Client-Anfragen, Routing, Authentifizierung, Token-Überprüfung, Lastverteilung, Zugriffskontrollen, ggf. API-Quota- und Caching-Funktionen.
 - *Nachrichten-Broker* (z. B. *RabbitMQ*, *ActiveMQ*): Entkopplung von Produzenten und Konsumenten, asynchrone Verarbeitung, Failover-Unterstützung, belastbare Messaging-Patterns (Queue, Topic/Publish-Subscribe).
 - *Konfigurations-Server* (z. B. *Spring Cloud Config*, *Consul*): Zentralisierung von Konfigurationen, Umgebungsunabhängigkeit, dynamische Konfigurations-Aktualisierung (eventuell ohne Neustart).
- Nutzen einer solchen Architektur: Lose Kopplung, bessere Skalierbarkeit, unabhängige Deployments, Fehlertoleranz durch asynchrone Pfade und zentrale Governance durch das Integrationslayer-Konzept.

Lösung 1.(b).

- *Monolith*: Eine einzige Codebasis, ein Deployment; Vorteile: einfache Konsistenz, geringerer Deploy-Aufwand, einfache Entwicklung bei kleinem Team. Nachteile: begrenzte Skalierbarkeit, riskantes Release-Management, hohe Kopplung.
- *Mehrschichtige Architektur (N-Tier)*: Sinnvolle Trennung von UI, Geschäftslogik und Persistenz; bessere Strukturierung und Wartbarkeit, aber noch eine zentrale Deploy-Einheit; eignet sich gut bei mittleren Teams und stabilen Anforderungen.
- *Microservice-Architektur*: Viele lose gekoppelte Dienste, jeder mit eigener Persistenz, eigenem Deployment, oft eigene Technologie-Stacks; Vorteile: unabhängige Skalierung, Fehlertrennung, schnelle Evolution einzelner Domänen; Nachteil: erhöhte Komplexität, Verteilte-System-Herausforderungen (Transaktionen, Konsistenz, Deployment-Universum).
- *Event-getriebene Architektur*: Systeme kommunizieren asynchron über Events; Vorteile: Entkopplung, resiliente Systeme, Skalierbarkeit, Observability; geeignete Nutzung bei hoher asynchroner Interaktion und losen Kopplungen.

- *Rahmenbedingungen für die Stilwahl:*
 - Teamgröße, Organisationsstruktur und Release-Strategie.
 - Kommunikations- und Transaktionsanforderungen (starke Konsistenz vs. eventual consistency).
 - Anforderungen an Skalierung, Verfügbarkeit, Fehlertoleranz.
 - Bereits vorhandene Infrastruktur (Containerisierung, Orchestrierung wie Kubernetes).
 - Regulatory/Compliance-Anforderungen (Datenhoheit, Auditbarkeit).

Lösung 1.(c).

- *Auswirkungen asynchroner Kommunikation:*
 - **Konsistenz:** Von starker Transaktionskonsistenz hin zu eventual consistency; Änderungsfluss asynchron, damit Antwortzeiten niedrig bleiben.
 - **Fehlertoleranz:** Entkopplung erhöht Robustheit; Ausfälle eines Dienstes beeinflussen andere nicht unmittelbar.
 - **Transaktionsmanagement:** Verteilte Transaktionen werden vermieden; stattdessen eventual konsistente Muster (Sagas, sagabasierte Orchestrierung oder Choreografie).
- *Zwei Muster der verteilten Architektur (Beispiele):*
 - **Event Sourcing:** Alle Zustandsänderungen werden als unveränderliche Ereignisse gespeichert; der aktuelle Zustand wird durch Rehydrierung aus Ereignissen aufgebaut; Vorteile: vollständige Auditierbarkeit, zeitliche Rekonstruktion, Undo/Replay-Möglichkeiten.
 - **CQRS (Command Query Responsibility Segregation):** Trennung von Schreib- und Lesewege; Commands verändern den Zustand, Queries lesen ihn; oft kombiniert mit Event Sourcing; Vorteile: optimierte Leseperformance, skalierbare Lese- und Schreibpfade.

Aufgabe 2.

Lösung 2.(a).

- *Messaging-Broker-Systeme (z. B. RabbitMQ):*
 - Nachrichtenbasierte, oft punkt-zu-punkt oder Publish-Subscribe Modelle.
 - Geringere Latenz für kurze, asynchrone Tasks; starke Guarantee-Optionen (at-least-once, exactly-once (je nach System)).
 - Gute Unterstützung für RPC-ähnliche Muster über Correlation IDs; starke Priorisierung und TTLs möglich.
 - Skalierbar durch Consumer-Worker-Vermehrung; Fokus auf zuverlässige Zustellung und Ordnung innerhalb Queues/Partitionen.
- *Streaming-Plattformen (z. B. Apache Kafka):*
 - Append-only Log-Repositories mit Partitionierung; ideal für Event-Streaming, Audit-Trails, Event Sourcing.
 - Höhere Durchsatzraten, robuste Skalierung über Partitionen; starke Lese-/Schreib-Semantik über Consumer Groups.
 - Garantien typischerweise *at-least-once*, teilweise *exactly-once* (mit Konfiguration und Idempotenz-Handling).
 - Reihenfolge innerhalb einer Partition ist gewährleistet; global geordnetes Ordering erfordert sorgfältige Partitionierung.
- *Unterschiede in Kennzahlen und Einsatzszenarien:*
 - Latenz: Broker-Systeme tendenziell niedrigere End-to-End-Latenz für kleine Aufgaben; Kafka bietet hohen Durchsatz, oft etwas höhere durchschnittliche Latenz.
 - Liefergarantien: Broker-Systeme bieten robuste QoS, Kafka liefert starke Persistenz/Audit-Trails.
 - Ordering: Broker-Systeme bieten je nach Modell Ordering innerhalb einer Queue; Kafka gewährleistet Ordering pro Partition.
 - Skalierung: Beide skalierbar, Kafka durch Partitionen; Broker können horizontal skaliert werden, oft kompliziertere Konfiguration.
 - Typische Einsatzszenarien: RPC/Task-Queues (RabbitMQ) vs. Event-Streaming, Event-Driven Architectures, Data Pipelines (Kafka).

Lösung 2.(b).

- *REST:*
 - Klar definierte Ressourcen-URIs, stateless, gut unterstützt durch HTTP-Standards.
 - Typensicherheit oft durch OpenAPI; breite Interoperabilität.
 - Gute Tool-Unterstützung, einfache Debugging- und Caching-Möglichkeiten.
- *gRPC:*

- Binäres Protokoll (Protocol Buffers), geringe Latenz, starke Typisierung, effizienter Payload.
- Gute Inter-Service-Kommunikation, eignet sich für Mikroservices untereinander.
- Starke Tooling-Unterstützung für Code-Generierung in vielen Sprachen; TLS-Unterstützung integriert.
- *GraphQL*:
 - Client-getriebene Abfragen über eine einzige Endpoint, schont Bandbreite durch selektive Felder.
 - Vorteilhaft bei umfangreichen, flexiblen Frontends; Typensicherheit über Schemas.
 - Komplexere Caching- und Schema-Management-Herausforderungen; Instrumentierung für Observability notwendig.
- *Zentrale Kriterien (zusammengefasst)*:
 - Performance und Latenz
 - Typensicherheit und Schnittstellen-Stabilität
 - Interoperabilität und Ökosysteme (Tooling, Spawn-Umgebungen)
 - Security-Modelle (Authentication/Authorization, Token-Handling)
 - Observability (Tracing, Metriken, Logs)
 - Wartbarkeit, Schema-/Vertragsverwaltung, Evolution von Schnittstellen
 - Wartungskosten und operativer Overhead

Lösung 2.(c).

- *Zugriffskontrolle*: Prinzipien von RBAC bzw. RBAC-ähnlich (z. B. ABAC als Erweiterung); Rollen modellieren Berechtigungen pro Ressource.
- *Authentifizierung/Autorisierung*: Zentrale Identitäts- und Zugriffsverwaltung (z. B. OIDC, OAuth 2.0); Token-basierte Authentifizierung; kurze Zugriffstoken mit Scopes.
- *Audit-Logging*: Unveränderliche Logs über Zugriffe, Änderungen, Erfolgs-/Fehlversuche; Nachvollziehbarkeit für Compliance.
- *Zentrale Governance-Aspekte*: Richtlinienbasiertes Sicherheitsmanagement, zentrale Policy-Engine, regelmäßige Audits, Konfigurations- und Change-Management, Geheimnisverwaltung (Secrets), Infrastruktur als Code.
- *Praktische Hinweise*:
 - Implementierung von rollenbasierter Zugriffskontrolle auf Ressourcenebene, feingranulare Berechtigungen dort, wo es sinnvoll ist.
 - Verwenden von Short-Lived Tokens, regelmäßige Rotation von Secrets.
 - Zentralisierung von Audit-Logs und ausreichende Speicherkapazität für Aufbewahrungsfristen.

Aufgabe 3.

Lösung 3.(a).

- *Event-getriebene Architektur für Benachrichtigungsdienst:*
 - **Ereignistypen:** z.B. `UserRegistered`, `OrderPlaced`, `OrderShipped`, `PaymentFailed`, `PromotionActivated`.
 - **Topic/Thema:** `notifications.events` bzw. separate Topics pro Ereignistyp z.B. `events.user`, `events.order`, je nach Granularität.
 - **Schema der Ereignisse:** definiertes Payload-Schema (z.B. Avro/JSON Schema) mit Feldern wie `eventId`, `timestamp`, `userId`, `payload` (mit relevanten Feldern).
 - **Consumer-Gruppen:** z.B. `email-notifier`, `sms-notifier`, `push-notifier`; jede Gruppe behandelt die relevanten Events und sorgt für parallele Verarbeitung.
 - *Idempotenz in der Verarbeitung:*
 - * Identitäts- bzw. Dedupelierungsschicht, z.B. Speicherung von `eventId` in einer Deduplication-Store; bei erneutem Empfang wird das Event ignoriert.
 - * Idempotente Handler-Logik: wiederholte gleiche Aktion führt zu keinem zusätzlichen Effekt (z.B. Versand einer einzigen Benachrichtigung pro Event).
 - * Nutzung von deduplizierenden Produzenten-IDs und idempotenten Schreiboperationen in Zielsystemen.

Lösung 3.(b).

- *Rolle von API-Gateway, Service-Mesh und Service-Discovery:*
 - **API-Gateway:** Ingress-Zugangspunkt, externes Routing, Authentifizierung, Autorisierung, einfache Load-Balancing-Funktionalität, API-Versionierung, Caching.
 - **Service-Mesh:** Interne Service-Kommunikation mit mTLS, Routing, Retries, Timeouts, Traffic-Shaping, Observability (Tracing, Metrics, Logging) auf Service-Ebene.
 - **Service-Discovery:** Dynamische Erkennung von Diensten (DNS-basiert oder Registry wie Consul); ermöglicht automatische Service-Orchestrierung und Lastverteilung.
- *Zusammenarbeit zur Zuverlässigkeit, Observability und Sicherheit:*
 - API-Gateway schützt Eingangspunkte, dient als zentraler Sicherheits- und Policy-Hub.
 - Service-Mesh sorgt für sichere, beobachtbare Inter-Service-Kommunikation und reduziert SSL-/TLS-Management auf Dev-/Prod-Ebene.
 - Service-Discovery ermöglicht dynamische, robuste Service-Lookups, unterstützt Skalierung und Canary/Blue-Green-Deployments.

Lösung 3.(c).

- *Verteilte Muster zur Robustheit und Skalierbarkeit:*
 - **Retries mit exponentiellem Backoff:** Wiederholungsversuche bei vorübergehenden Fehlern; schützt vor flachen Fehlerzuständen, muss aber mit idempotenten Operationen kombiniert werden.

- **Circuit Breaker:** Verhindert, dass fehlerhafte Abhängigkeiten das gesamte System lähmen; öffnet den Stromfluss, bis der Teil wieder stabil ist.
- **Fallbacks:** Alternative Pfade oder Standardantworten, wenn ein Dienst nicht erreichbar ist.
- *Konsumverhalten in Bezug auf Konsistenz vs. Verfügbarkeit (CAP):*
 - * In verteilten Systemen kann man unter Ausfall von Konsistenz (zyklische Updates) oft Verfügbarkeit priorisieren; eventual consistency wird angestrebt.
 - * Trade-offs müssen bei Designentscheidungen vorab festgelegt werden (z. B. Lesepfade strikt konsistent oder eventual konsistent).

Aufgabe 4.

Lösung 4.(a).

- *Referenz-Architektur für Skalierbarkeit und Hochverfügbarkeit:*
 - **Frontend/Edge:** Content Delivery Network (CDN) für statische Ressourcen; Stateful bzw. stateless Web-/Mobile-Clients; Session-Management im Frontend-Service.
 - **Backend-Services:** Stateless API-Gateway an der Spitze; eine Vielzahl von Microservices mit eigenem Persistenz-Layer (polyglot Persistenz je Dienst).
 - **Datenhaltung:**
 - * primäre relationale Datenbanken pro Dienst bzw. per Domain;
 - * NoSQL-/Dokumenten-/In-Memory-Caches (z. B. Redis) dort, wo es sinnvoll ist.
 - * Event-Store oder Thema-basiertes Messaging (z. B. Kafka) für asynchrone Integrationen.
 - **Messaging-Komponenten:** Event-Bus/Message-Broker (z. B. Kafka/RabbitMQ) zur Entkopplung von Produzenten und Consumern; ermöglicht Skalierung und robuste Fehlerbehandlung.
 - **Observability:** verteilte Tracing (z. B. OpenTelemetry), Metriken (Prometheus/Grafana), Log-Aggregation (ELK/EFK); zentrale Dashboards für MTTR/MTBF.
 - **Zusammenwirken:** Frontend-Anfragen gehen an API-Gateway; Backend-Services kommunizieren asynchron über Messaging; Datenhaltung folgt eigenständigen Scoped-Buckets pro Service; Observability sammelt Metriken aus allen Schichten.
 - *Muster für Skalierung und Ausfallsicherheit:* horizontale Skalierung von Stateless-Services, Replikation, Read-Templates/Caching, circuit-breakers, Disaster-Recovery-Pläne, Multi-Region-Deployments.

Lösung 4.(b).

- *Architekturbewertung:* Kriterien zu Skalierbarkeit, Verfügbarkeit und Sicherheit.
 - **Metriken:**
 - * Latenz (p50, p95, p99), Durchsatz (Requests/sek), Fehlerrate (
 - * Verfügbarkeitskennzahlen (SLA, SLI, SLO), MTTR, Change-Failure-Rate.
 - **Beobachtbarkeit:** verteiltes Tracing, Logs, Metriken; Monitorings-Dashboards für Engpässe.
 - **Potenzielle Engpässe:**
 - * Zentralisierte DBs oder Schemata, die nicht skalieren.
 - * Synchroner Tight-Coupling zwischen Diensten.
 - * Grenzen der Messaging-Systeme (Durchsatz, Lag, Partitionen).
 - **Strategien zur Behebung:**
 - * Skalierung/Sharding von Datenbanken; verbesserte Indizes, NoSQL-Optionen je nach Bedarf.
 - * Entkopplung durch asynchrone Kommunikation; Einführung von CQRS/Event-Sourcing, Caching-Strategien.

- * Optimierung der Service-Interaktion (asynchrone Pfade, resilienter Code, Circuit Breaker).

Lösung 4.(c).

- *Schrittweiser Migrationspfad (Strangler-Pattern):*
 - **Phase 1: Bestandsaufnahme und Grenzziehung** – Identifikation von Domänen/Bounded Contexts; Definition klarer Schnittstellen.
 - **Phase 2: Plattform-/Infrastruktur-Aufbau** – Containerisierung, Orchestrierung (z. B. Kubernetes), zentrale Services (API-Gateway, Registry, Messaging) als Plattform.
 - **Phase 3: Evolution der Funktionen** – Neue Funktionen als Microservices implementieren; schrittweise Schnittstellen des Monolithen durch Strangler-Endpoints ersetzen.
 - **Phase 4: Migration der Geschäftsprozesse** – Portieren von Funktionen schrittweise; Monolith bleibt bis zur vollständigen Ersetzung aktiv, danach Abkündigung.
 - **Phase 5: Governance und Betrieb** – Vertrags- und Schnittstellentests, forward- und backward- Kompatibilität, Migrations-Backups, Canary- und Blue-Green-Deployments.
- *Risikominderung und Governance:*
 - **Phasen- und Feature-Flags:** schrittweise Aktivierung neuer Dienste, schattenhafte Tests und schrittweise Lieferung.
 - **Contract Testing:** sicherstellen, dass neue Microservices und Altsysteme die gleichen Verträge erfüllen.
 - **Dokumentation und Compliance:** zentrale Richtlinien, Auditierbarkeit, Geheimnis- und Secrets-Management, Sicherheits-Review.