

Tag – Lernziele

Tag	Lernziel	Lernziel iSAQB
Tag 1	01-01:	Definitionen von Softwarearchitektur diskutieren
Tag 1	01-04:	Aufgaben und Verantwortung von Softwarearchitekt:innen verstehen
Tag 1	01-06:	Rolle von Softwarearchitekt:innen in Beziehung zu anderen Stakeholdern setzen
Tag 1	02-01:	Stakeholder-Anliegen verstehen
Tag 1	02-02:	Anforderungen und Randbedingungen klären und berücksichtigen können
Tag 1	02-05:	Explizite Aussagen vor impliziten Annahmen bevorzugen
Tag 1	03-02:	Softwarearchitekturen entwerfen
Tag 1	03-03:	Vorgehen und Heuristiken zur Architekturentwicklung auswählen und anwenden können
Tag 1	03-05:	Zusammenhang zwischen Feedback-Schleifen und Risiko
Tag 1	03-07:	Schnittstellen entwerfen und spezifizieren
Tag 1	04-01:	Anforderungen an technische Dokumentation erläutern und berücksichtigen
Tag 1	04-02:	Softwarearchitekturen beschreiben und kommunizieren
Tag 1	04-03:	Notations-/Modellierungsmittel für Beschreibung von Softwarearchitektur erläutern und anwenden
Tag 1	04-04:	Unerwartete Situationen geschickt bewältigen
Tag 1	04-05:	Architektursichten erläutern und anwenden
Tag 1	04-06:	Schnittstellen dokumentieren
Tag 2	01-02:	Nutzen und Ziele von Softwarearchitektur verstehen und erläutern
Tag 2	01-03:	Langfristige Auswirkungen von Softwarearchitektur verstehen
Tag 2	01-04:	Aufgaben und Verantwortung von Softwarearchitekt:innen verstehen
Tag 2	02-01:	Stakeholder-Anliegen verstehen
Tag 2	02-02:	Anforderungen und Randbedingungen klären und berücksichtigen können

Tag 2	02-03:	Qualitäten eines Softwaresystems verstehen und erklären
Tag 2	02-03:	Qualitätsmodelle und Qualitätsmerkmale diskutieren
Tag 2	02-04:	Anforderungen an Qualitäten formulieren
Tag 2	02-05:	Explizite Aussagen vor impliziten Annahmen bevorzugen
Tag 2	03-01:	Anforderungen durch Architektur erreichen
Tag 2	03-03:	Vorgehen und Heuristiken zur Architekturentwicklung auswählen und anwenden können
Tag 2	03-05:	Zusammenhang zwischen Feedback-Schleifen und Risiko verstehen
Tag 2	03-12:	Herausforderungen verteilter Systeme kennen
Tag 2	04-05:	Architektsichten erläutern und anwenden
<hr/>		
Tag 3	01-07:	Bedeutung von Daten und Datenmodellen kennen
Tag 3	02-02:	Anforderungen und Randbedingungen klären und berücksichtigen können
Tag 3	03-01:	Anforderungen durch Architektur erreichen
Tag 3	03-02:	Softwarearchitekturen entwerfen
Tag 3	03-03:	Vorgehen und Heuristiken zur Architekturentwicklung auswählen und anwenden können
Tag 3	03-04:	Entwurfsprinzipien erläutern und anwenden
Tag 3	03-06:	Abhängigkeiten von Bausteinen managen
Tag 3	03-08:	Wichtige Architekturmuster beschreiben, erklären und angemessen anwenden
Tag 3	03-09:	Wichtige Entwurfsmuster beschreiben, erklären und angemessen anwenden
Tag 3	03-10:	Querschnittsthemen identifizieren und Querschnittskonzepte entwerfen und umsetzen
Tag 3	03-11:	Grundlegende Prinzipien von Software-Deployments kennen
Tag 3	04-03:	Notations-/Modellierungsmittel für Beschreibung von Softwarearchitektur erläutern und anwenden
Tag 3	04-05:	Architektsichten erläutern und anwenden
Tag 3	04-06:	Schnittstellen dokumentieren
Tag 3	05-01:	Gründe für Architekturanalyse kennen
Tag 3	05-02:	Qualitäten eines Softwaresystems analysieren

Tag 3	06-01:	Bezug von Anforderungen und Randbedingungen zur Lösung erfassen
Tag 4	01-04:	Aufgaben und Verantwortung von Softwarearchitekt:innen verstehen
Tag 4	03-11:	Grundlegende Prinzipien von Software-Deployments kennen
Tag 4	04-01:	Anforderungen an technische Dokumentation erläutern und berücksichtigen
Tag 4	04-02:	Softwarearchitekturen beschreiben und kommunizieren
Tag 4	04-03:	Notations-/Modellierungsmittel für Beschreibung von Softwarearchitektur erläutern und anwenden
Tag 4	04-05:	Architektsichten erläutern und anwenden
Tag 4	04-07:	Querschnittsthemen dokumentieren und kommunizieren
Tag 4	04-08:	Architekturentscheidungen erläutern und dokumentieren
Tag 4	04-09:	Weitere Hilfsmittel und Werkzeuge zur Dokumentation kennen
Tag 4	05-02:	Qualitäten eines Softwaresystems analysieren
Tag 4	05-03:	Konformität mit Architekturentscheidungen bewerten
Tag 4	06-02:	Technische Umsetzung einer Lösung nachvollziehen

iSAQB Foundation Level – Self-Check (Curriculum 2025.1)

Kapitel 1 – Grundbegriffe von Softwarearchitektur

Lernziel + R1/R2/R3

Sicher ()

Unsicher ()

• LZ 01-01: Definitionen von Softwarearchitektur verstehen (R1)

• LZ 01-02: Ziele und Nutzen von Softwarearchitektur verstehen und erläutern (R1)

- LZ 01-03: Langfristige Auswirkungen von Softwarearchitektur kennen (R3)
- LZ 01-04: Aufgaben und Verantwortung von Softwarearchitekt:innen verstehen (R1)
- LZ 01-05: Abgrenzung zu anderen Architekturdomänen (R3)
- LZ 01-06: Rolle von Softwarearchitekt:innen mit anderen Stakeholdern in Beziehung setzen (R1)
- LZ 01-07: Bedeutung von Daten und Datenmodellen (R2)

Offene Fragen oder Aha-Momente:

Kapitel 2 – Anforderungen und Randbedingungen

Lernziel (voller Wortlaut + R1/R2/R3) Sicher () Unsicher ()

- LZ 02-01: Stakeholder-Anliegen verstehen (R1, R3)
- LZ 02-02: Anforderungen und Randbedingungen

klären und berücksichtigen können (R1–R3)

- LZ 02-03: Qualitäten eines Softwaresystems verstehen und erklären (R1)
- LZ 02-04: Anforderungen an Qualitäten formulieren (R1–R3)
- LZ 02-05: Explizite Aussagen vor impliziten Annahmen bevorzugen (R1)

Offene Fragen oder Aha-Momente:

Kapitel 3 – Entwurf und Entwicklung von Softwarearchitekturen

Lernziel (voller Wortlaut + R1/R2/R3) Sicher () Unsicher ()

- LZ 03-01: Anforderungen durch Architektur erreichen (R1)
- LZ 03-02: Softwarearchitekturen entwerfen (R1)
- LZ 03-03: Vorgehen und Heuristiken zur Architekturentwicklung

auswählen und anwenden
können (R1, R3)

- LZ 03-04:
Entwurfsprinzipien
erläutern und anwenden
(R1–R3)
- LZ 03-05: Zusammenhang
zwischen Feedback-
Schleifen und Risiko (R1,
R2)
- LZ 03-06: Abhängigkeiten
von Bausteinen managen
(R1)
- LZ 03-07: Schnittstellen
entwerfen und spezifizieren
(R1–R3)
- LZ 03-08: Wichtige
Architekturmuster
beschreiben, erklären und
angemessen anwenden (R1,
R3)
- LZ 03-09: Wichtige
Entwurfsmuster
beschreiben, erklären und
angemessen anwenden (R3)
- LZ 03-10:
Querschnittsthemen
identifizieren und
Querschnittskonzepte
entwerfen und umsetzen
(R1)
- LZ 03-11: Grundlegende
Prinzipien von Software-
Deployments kennen (R3)
- LZ 03-12:
Herausforderungen

verteilter Systeme kennen
(R3)

Offene Fragen oder Aha-Momente:

Kapitel 4 – Beschreibung und Kommunikation von Softwarearchitekturen

Lernziel (voller Wortlaut +
R1/R2/R3) Sicher () Unsicher ()

- LZ 04-01: Anforderungen an technische Dokumentation erläutern und berücksichtigen (R1)

- LZ 04-02: Softwarearchitekturen beschreiben und kommunizieren (R1–R3)

- LZ 04-03: Notations-/Modellierungsmittel erläutern und anwenden (R2–R3)

- LZ 04-04: Lernziel nicht gefunden (R3)

- LZ 04-05: Architektsichten erläutern und anwenden (R1)

- LZ 04-06: Schnittstellen dokumentieren (R1)

- LZ 04-07:
- Querschnittsthemen dokumentieren und kommunizieren (R2)
- LZ 04-08:
- Architekturentscheidungen erläutern und dokumentieren (R1–R2)
- LZ 04-09: Weitere Hilfsmittel und Werkzeuge zur Dokumentation kennen (R3)

Offene Fragen oder Aha-Momente:

Kapitel 5 – Analyse und Bewertung von Softwarearchitekturen

Lernziel (voller Wortlaut + Sicher () Unsicher ()
R1/R2/R3)

- LZ 05-01: Gründe für Architekturanalyse kennen (R1)
- LZ 05-02: Qualitäten eines Softwaresystems analysieren (R1, R3)
- LZ 05-03: Konformität mit Architekturentscheidungen bewerten (R2)

Offene Fragen oder Aha-Momente:

Kapitel 6 – Beispiele für Softwarearchitekturen

Lernziel (voller Wortlaut +
R1/R2/R3)

Sicher ()

Unsicher ()

- LZ 06-01: Bezug von Anforderungen und Randbedingungen zur Lösung erfassen (R3)
- LZ 06-02: Technische Umsetzung einer Lösung nachvollziehen (R3)

Offene Fragen oder Aha-Momente:

Anhang

Zusammengefasste Lernziele mit R1/R2/R3

Kapitel 1 – Grundlagen der Softwarearchitektur

LZ 01-01 (R1): Definitionen von Softwarearchitektur verstehen

Was du können sollst:

- Den Begriff „Softwarearchitektur“ in eigenen Worten erklären
 - Unterschiedliche Definitionen vergleichen (iSAQB, IEEE, andere Autoren)
 - Verstehen, dass Architektur sowohl *Struktur* wie *Entscheidungen* meint
 - Architektur \neq Code, aber eng verbunden
 - Architektur als „Reduktion von Komplexität“ verstehen
-

LZ 01-02 (R1): Ziele und Nutzen von Softwarearchitektur verstehen und erläutern

Was du lernen sollst:

- Warum Architektur essenziell für Qualität, Wartbarkeit und Robustheit ist
 - Architektur als Mittel zur Risikoreduktion
 - Architektur als Kommunikations- und Entscheidungswerkzeug
 - Beitrag zu Projektzielen: Zeit, Kosten, Qualität
 - Wie gute Architektur Fehlentwicklungen vermeidet
-

LZ 01-03 (R3): Langfristige Auswirkungen von Softwarearchitektur kennen

Was du wirklich verstehen sollst:

- Architekturentscheidungen wirken sich *jahrelang* aus
 - Früh getroffene Entscheidungen sind schwierig zu ändern („Architectural Smells“, „Technical Debt“)
 - Architektur beeinflusst:
 - Änderbarkeit
 - Skalierbarkeit
 - Betrieb / Deployment
 - Teamorganisation
 - Wie schlechte Architektur zu hohen Folgekosten führt
-

LZ 01-04 (R1): Aufgaben und Verantwortung von Softwarearchitekt:innen verstehen

🔍 Was dazugehört:

- Architekturentscheidungen treffen und vertreten
- Risiken erkennen und adressieren
- Anforderungen verstehen und priorisieren
- Technische Leitplanken definieren
- Moderation zwischen Teams, Stakeholdern, Product Owners
- Dokumentation & Kommunikation

➡️ **Nicht:** „alles selbst programmieren“.

LZ 01-05 (R3): Abgrenzung zu anderen Architekturdomänen

🔍 Was du unterscheiden können musst:

- Softwarearchitektur ↗ Enterprise Architecture
 - Lösungsarchitektur ↗ Systemarchitektur
 - IT-Architektur ↗ Infrastrukturdomain
 - Wo Softwarearchitektur „anfängt“ und „aufhört“
 - Welche Entscheidungen in welche Domäne gehören
 - Wie Zusammenarbeit aussieht
-

LZ 01-06 (R1): Rolle von Softwarearchitekt:innen mit anderen Stakeholdern in Beziehung setzen

🔍 Was du dabei lernst:

- Wer die wichtigsten Stakeholder sind:
 - Management
 - Entwickler
 - Test
 - Betrieb
 - Security
 - Endnutzer
- Wie Architekt:innen mit ihnen interagieren
- Wie man Kommunikationsbedürfnisse erkennt
- Wie man technische und fachliche Interessen ausbalanciert

- Stakeholder-Analyse und deren Ziele
-

LZ 01-07 (R2): Bedeutung von Daten und Datenmodellen kennen

Worum es dabei wirklich geht:

- Datenmodellierung (logisch, physikalisch) in Grundzügen verstehen
- Daten als „Lebensader des Systems“ begreifen
- Warum Datenkonsistenz architekturrelevant ist
- Zusammenhang von Daten und:
 - Integrität
 - Qualität
 - Performance
 - Skalierung
- Warum falsche Datenmodelle langfristige Probleme verursachen

Zusammengefasste Gesamtbotschaft von Kapitel 1

Du sollst lernen:

- Was Softwarearchitektur ist
- Welchen Nutzen sie hat
- Welche langfristigen Konsequenzen schlechte/gute Architektur hat
- Welche Aufgaben und Verantwortung Architekt:innen übernehmen
- Wie sie mit Stakeholdern interagieren
- Wie Datenentscheidungen die Architektur formen
- Wie sich Softwarearchitektur von anderen Architekturarten abgrenzt

 Damit verstehst du die Rolle der Architektur im gesamten Software-Lebenszyklus.

Kapitel 2 – Anforderungen & Randbedingungen (LZ 02-01 bis 02-05)

LZ 02-01 (R1, R3) – Stakeholder-Anliegen verstehen

Was du lernen sollst:

- Wer Stakeholder sind (Nutzer, Betrieb, Test, Management ...)
- Welche Erwartungen, Ziele und Ängste sie haben

- Wie man Stakeholder identifiziert, priorisiert und analysiert
 - Wie man Konflikte und Zielkonflikte erkennt
 - Architektur als Vermittlung zwischen Interessen
-

LZ 02-02 (R1–R3) – Anforderungen & Randbedingungen klären und berücksichtigen können

Wichtige Lernpunkte:

- Anforderungen erheben (Interviews, Workshops, Szenarien)
 - Randbedingungen identifizieren (Budget, Compliance, Technik, Organisation)
 - Anforderungen klassifizieren und priorisieren
 - Entscheidungen dokumentieren → „Warum diese Architektur?“
 - Anforderungen *kontinuierlich pflegen*, nicht einmalig erfassen
-

LZ 02-03 (R1) – Qualitäten eines Softwaresystems verstehen und erklären

Du sollst verstehen:

- Qualitätsattribute (z. B. Availability, Performance, Maintainability)
 - Warum Qualitätsziele Architekturentscheidungen treiben
 - Wie man Qualitätsmerkmale erklärt und vermittelt
 - Welche Zielkonflikte entstehen (z. B. Performance vs. Sicherheit)
-

LZ 02-04 (R1–R3) – Anforderungen an Qualitäten formulieren

Zentrale Fähigkeiten:

- Qualitätsanforderungen präzise formulieren (z. B. „95% der Requests < 200 ms“)
 - Qualitätsattribute messbar machen
 - Architekturelevante Qualitätsszenarien erstellen
 - Qualität als Bestandteil von Akzeptanzkriterien aufnehmen
-

LZ 02-05 (R1) – Explizite Aussagen vor impliziten Annahmen bevorzugen

Was du verstehen musst:

- Implizite Annahmen sind typische Fehlerquelle
 - Risiken durch unausgesprochene Erwartungen
 - Alles Wichtige schriftlich und eindeutig klären
 - Beispiele: „Die Datenbank ist immer erreichbar“, „Benutzer laden nur kleine Dateien“
 - Architektur = Entscheidungen + dokumentierte Annahmen
-

Kapitel 3 – Entwurf & Entwicklung von Softwarearchitekturen (LZ 03-01 bis 03-12)

LZ 03-01 (R1) – Anforderungen durch Architektur erreichen

Wichtig ist:

- Architektur als Antwort auf Anforderungen
 - Traceability: „Welche Entscheidung erfüllt welche Anforderung?“
 - Qualitätsszenarien → Architekturtaktiken → Bausteine
-

LZ 03-02 (R1) – Softwarearchitekturen entwerfen

Du sollst:

- Top-down, Bottom-up, Inside-out, Outside-in erklären
 - Architekturelemente definieren (Bausteine, Sichten, Schnittstellen)
 - Verschiedene Architekturrepräsentationen nutzen (z. B. C4)
 - Entscheidungen nachvollziehbar begründen
-

LZ 03-03 (R1, R3) – Vorgehen & Heuristiken auswählen und anwenden

Du lernst:

- Vorgehensmodelle (ADD, arc42, Attribute-Driven Design, iterative Architekturentwicklung)
- Heuristiken wie:
 - „Separation of Concerns“
 - „High Cohesion, Low Coupling“
 - „YAGNI“, „Keep It Simple“
 - „Erst Risiken eliminieren“

- Architektur iterativ entwickeln
-

LZ 03-04 (R1–R3) – Entwurfsprinzipien erläutern und anwenden

Zentrale Themen:

- Architekturprinzipien (Isolation, Schichten, Kapselung, Abstraktion)
 - SOLID-Regeln
 - Entwurfsentscheidungen zu Kopplung & Kohäsion
 - Prinzipien gegen Komplexität
-

LZ 03-05 (R1, R2) – Zusammenhang zwischen Feedback-Schleifen & Risiko

Du sollst verstehen:

- Risiko senken durch frühe Validierung
 - Prototypen / Spike Solutions
 - Iterationen, Experimente, Kundenfeedback
 - „Fail Fast“ und „frühe technische Entscheidungen testen“
-

LZ 03-06 (R1) – Abhängigkeiten von Bausteinen managen

Wichtig:

- Kopplung reduzieren
 - Abhängigkeiten sichtbar machen
 - Bausteine stabilisieren
 - Zyklen vermeiden
 - „Stabil Dependencies Principle“
-

LZ 03-07 (R1–R3) – Schnittstellen entwerfen & spezifizieren

Du musst lernen:

- Typen von Schnittstellen (synchr., asynchr., REST, Events, APIs)
- Semantik statt nur Syntax
- Versionierung, Verträge, Kompatibilität
- Fehler- und Ausnahmeszenarien

LZ 03-08 (R1, R3) – Wichtige Architekturmuster anwenden

Z. B.:

- Layered Architecture
- Pipes & Filters
- Microservices
- Event-driven
- Hexagonal Architecture
- SOA

Du sollst wissen: wann welches Muster sinnvoll ist.

LZ 03-09 (R3) – Wichtige Entwurfsmuster anwenden

Typische Muster:

- Factory, Strategy, Observer, Adapter
- Composite, Facade, Singleton (kritisch!)
- Dependency Injection

Du sollst lernen:

Wann und warum man sie sinnvoll nutzt — nicht nur „auswendig“.

LZ 03-10 (R1) – Querschnittsthemen identifizieren & entwerfen

Beispiele:

- Logging
- Security
- Caching
- Monitoring
- Validierung
- Fehlerbehandlung

Du sollst verstehen:

→ Querschnittskonzepte müssen konsistent, zentral & dokumentiert sein.

LZ 03-11 (R3) – Prinzipien von Software-Deployments kennen

Wichtige Themen:

- Build → Test → Deploy → Run
 - Immutable deployments
 - Container, Orchestrierung
 - Deployment-Topologien
 - Continuous Delivery
-

LZ 03-12 (R3) – Herausforderungen verteilter Systeme

Zentrale Herausforderungen:

- Latenz
 - Partitionierung
 - Konsistenz / CAP
 - Zeit/Clock Drift
 - Fehlertoleranz, Retries
 - Datenkonsistenz
-

Kapitel 4 – Beschreibung & Kommunikation (LZ 04-01 bis 04-09)

LZ 04-01 (R1) – Anforderungen an technische Dokumentation

- Warum dokumentieren?
 - Was muss dokumentiert werden?
 - Zielgruppenorientierte Dokumentation
 - „Gerade so viel wie nötig“
-

LZ 04-02 (R1–R3) – Softwarearchitekturen beschreiben & kommunizieren

- Architektur adressatengerecht darstellen
 - Non-technical Stakeholder berücksichtigen
 - Kommunikationsformen: Text, Diagramme, Modelle
-

LZ 04-03 (R2–R3) – Notations-/Modellierungsmittel anwenden

- UML in Grundzügen
 - C4 Model
 - Textuelle Modelle
 - Konsistente Diagramme
-

LZ 04-04 (R3) – Lernziel nicht gefunden

→ Im Curriculum absichtlich leer.

LZ 04-05 (R1) – Architektsichten anwenden

- Bausteinsicht
 - Laufzeitsicht
 - Verteilungssicht
 - Kontextsicht
 - Schnittstellensicht
-

LZ 04-06 (R1) – Schnittstellen dokumentieren

- Syntax + Semantik
 - Fehlerfälle
 - Versionierung
 - Verträge (Contracts)
-

LZ 04-07 (R2) – Querschnittsthemen dokumentieren

- Security
 - Monitoring
 - Logging
 - Qualitätsszenarien
-

LZ 04-08 (R1–R2) – Architekturentscheidungen dokumentieren

- ADRs
 - Entscheidungsalternativen
 - Gründe & Konsequenzen
 - Risiken
-

LZ 04-09 (R3) – Werkzeuge & Hilfsmittel kennen

- PlantUML
 - arc42 toolchain
 - Modelling Tools
 - Docs-as-code
-

❖ Kapitel 5 – Analyse & Bewertung

LZ 05-01 (R1) – Gründe für Architekturanalyse kennen

- Risiken finden
 - Quality-Attr.-Lücken aufdecken
 - Kosten/Impact von Änderungen einschätzen
 - Architekturfitness bewerten
-

LZ 05-02 (R1, R3) – Qualitäten analysieren

- ATAM / Szenarioanalyse
 - Messbarkeit
 - Bottlenecks finden
 - Qualitäts-Szenarien prüfen
-

LZ 05-03 (R2) – Konformität bewerten

- Prüfen: Hält das System sich an die Architektur?
- Abweichungen erkennen
- Architektur-Erosion verhindern

Kapitel 6 – Beispiele

LZ 06-01 (R3) – Bezug von Anforderungen zur Lösung erfassen

- Anforderungen → Bausteine → Sichten
 - Rückverfolgbarkeit herstellen
-

LZ 06-02 (R3) – Technische Umsetzung nachvollziehen

- Code verstehen
 - Architekturprinzipien im Code wiederfinden
 - Ist-Soll-Abgleich
-

KAPITEL 1 – Grundlagen der Softwarearchitektur

LZ 01-01 – Definitionen von Softwarearchitektur verstehen

Was man lernen sollte

- Verschiedene Architekturdefinitionen aus Literatur & Praxis kennen.
 - Gemeinsamkeiten dieser Definitionen identifizieren.
 - Verstehen, dass Architektur *Konzeption* ist – nicht Code.
 - Den Unterschied zwischen Architektur, Design, Implementierung und Infrastruktur erklären können.
 - Wissen, warum Definitionen unterschiedlich sind und wozu sie dienen.
-

Antwort

Softwarearchitektur beschreibt die **grundlegende Struktur eines Softwaresystems**. Darin enthalten sind:

- **Bausteine/Komponenten**, aus denen das System besteht
- **Beziehungen/Abhängigkeiten** zwischen diesen Bausteinen
- **Schnittstellen**, über die sie interagieren

- **Architekturentscheidungen**, die wesentlich sind und schwer rückgängig zu machen
- **Technische und fachliche Konzepte**, die das System prägen
- **Qualitätstreiber**, die Form und Struktur bestimmen

Obwohl verschiedene Quellen unterschiedliche Definitionen geben, betonen alle:

1. **Struktur** (das „Was“ aus welchen Teilen besteht das System?)
2. **Beziehungen/Kopplung** (wie interagieren diese Teile?)
3. **Abstraktion** (Architektur ist ein Modell, kein Code)
4. **Wesentliche Entscheidungen** (langfristig wirkende Weichenstellungen)
5. **Qualitäten/Non-functional Requirements** als Architekturtreiber

Wichtige Klarstellungen:

- Architektur ≠ Implementierung
- Architektur ≠ Detailliertes Design
- Architektur = Modell + Entscheidungen + Begründungen
- Architektur = „die Dinge, die schwer zu ändern sind“ (Ford/NEA)

Typische Fehler von Studierenden:

- ✗ Architektur nur als Diagramm sehen
- ✗ Architektur als „High-Level-Code“ betrachten
- ✗ Architektur nur technisch verstehen (statt fachlich + technisch)

Richtige Sichtweise:

Architektur ist das *konzeptionelle Gerüst*, das Qualität, Verhalten und Struktur eines Systems langfristig bestimmt.

★ LZ 01-02 – Nutzen und Ziele von Softwarearchitektur

Was man lernen sollte

- Warum Softwarearchitektur nötig ist.
 - Welche Probleme Architektur löst: Komplexität, Risiko, Verständlichkeit.
 - Architektur als Kommunikationsmittel verstehen.
 - Zusammenhang Architektur ↔ Qualität.
 - Wie Architektur Projektziele beeinflusst.
-

Antwort

Softwarearchitektur hat vier Hauptnutzen:

1. Architektur reduziert Komplexität

Große Systeme sind unüberschaubar. Architektur zerlegt sie in:

- Module
- Komponenten
- Schichten
- Subsysteme

→ Das mindert mentale Last und schafft Orientierung.

2. Architektur ermöglicht hohe Qualität

Qualitätsmerkmale (z. B. Performance, Sicherheit, Wartbarkeit) entstehen **nicht durch Code**, sondern durch Architekturentscheidungen wie:

- Speicherstrategien
- Kommunikationsmuster
- Datenhaltung
- Deployment-Topologie

→ Architektur ist Haupttreiber für Qualitätsziele.

3. Architektur ist ein Kommunikationswerkzeug

Sie schafft ein gemeinsames Verständnis zwischen:

- Management
- Fachbereichen
- Entwicklern
- Tester:innen
- Betrieb

→ Sie ist „die gemeinsame Sprache“ des Projekts.

4. Architektur reduziert Projektrisiken

Durch:

- frühzeitige Entscheidungen
- Planung kritischer Bereiche

- Definition von Leitlinien
- Wahl der passenden Technologien

→ Risiken werden früher erkannt und beherrschbar.

Typischer Fehler:

Studenten glauben oft, Architektur diene nur „Design“.

Falsch — Architektur **steuert Qualität und Erfolg des ganzen Systems**.

★ LZ 01-03 – Langfristige Wirkung von Architektur

Was man lernen sollte

- Architektur wirkt über die gesamte Lebensdauer eines Systems.
 - Späte Änderungen sind teuer.
 - Zusammenhang Architektur ↔ langfristige Wartbarkeit.
 - Folgen schlechter Architekturentscheidungen verstehen.
 - Erkennen, dass Architektur zukunftssicher sein muss.
-

Antwort

Architekturentscheidungen sind **langfristige Investitionen**.

Ihr Einfluss zeigt sich über Jahre:

- Wie gut lässt sich das System erweitern?
- Wie stabil ist es im Betrieb?
- Wie einfach lassen sich Fehler beheben?
- Wie teuer sind Anpassungen?
- Wie gut kann man Teamstrukturen anpassen?

Warum wirkt Architektur so lange?

- Viele Entscheidungen sind schwer rückgängig zu machen (z. B. Datenbankwahl, Kommunikationsprotokolle).
- Architektur beeinflusst Entwicklergewohnheiten, Code-Struktur und Infrastruktur.
- Architekturfehler akkumulieren als technische Schulden.

Beispiele für langfristige Auswirkungen

- Wahl eines Monolithen → schnelle Entwicklung, aber schwer skalierbar

- Wahl zahlreicher Microservices → hohe Komplexität, aber Skalierung möglich
- Schlechte Datenmodelle → hohe Wartungskosten über Jahre
- Ungünstige Modularisierung → harte Kopplung, teure Änderungen

Typische Fehler:

- ✗ „Wir fixen das später“ — später ist viel teurer.
- ✗ Architektur minimalistisch behandeln, um Zeit zu sparen.
- ✗ Architektur als „Once and Done“ verstehen.

Wahrheit:

Architektur ist *entscheidend* für die Zukunftsfähigkeit eines Systems und wird ständig weiterentwickelt.

★ LZ 01-04 – Aufgaben & Verantwortung von Softwarearchitekt:innen

Was man lernen sollte

- Was Architekt:innen wirklich tun.
 - Unterschied zwischen Rolle und Position.
 - Verantwortung gegenüber Stakeholdern.
 - Architekturentscheidungen begründen können.
 - Moderation, Kommunikation, Risikoarbeit.
-

Antwort

Softwarearchitekt:innen sind verantwortlich für:

1. Architekturentscheidungen

- Strukturen und Schnittstellen definieren
- Technologien auswählen
- Qualitätsanforderungen berücksichtigen
- Risiken bewerten
- Alternativen vergleichen

→ Entscheidungen müssen dokumentiert und begründet werden (ADRs).

2. Kommunikation & Moderation

Architekt:innen sind Übersetzer zwischen:

- Fachbereich ↔ Technik
- Management ↔ Entwicklung
- Betrieb ↔ Entwicklerteams

→ Gute Kommunikation ist wichtiger als Technik.

3. Stakeholder-Management

- Erwartungen klären
- Konflikte moderieren
- Annahmen explizit machen
- Priorisieren in Workshops

4. Risiko- und Qualitätsmanagement

- Risikoerkennung
- Architektur-Reviews
- Szenarioanalysen
- Qualitätstreiber identifizieren

5. Begleitung der Umsetzung

- Architektur-Coaching
- Einhaltung der Leitlinien sicherstellen
- Entscheidungen während der Implementierung anpassen

Typische Missverständnisse:

✗ Architekt:innen sind „Tech-Übermenschen“ → nein, es ist eine **Kommunikationsrolle**.

✗ Architekt:innen programmieren nicht → falsch, viele tun es.

Richtige Sicht:

Architekt:innen verantworten *die technischen Grundlagen des Erfolgs*.

★ LZ 01-05 – Architektur und Umfeld

Was man lernen sollte

- Architektur existiert nicht isoliert.
 - Einflussfaktoren: Team, Organisation, Technologie, Prozesse.
 - Verständnis von Conway's Law.
 - Wie Umfeldbeschränkungen Architekturformen definieren.
-

Antwort

Architektur wird durch viele externe Faktoren beeinflusst:

1. Organisation

Teamstrukturen bestimmen Softwarestrukturen:

→ **Conway's Law:** Systeme spiegeln Kommunikationsstrukturen wider.

Beispiel:

- Viele unabhängige Teams → Microservice-Natur.
- Ein großes Team → eher Monolith/Modularmonolith.

2. Prozesse

Agile, klassische oder hybride Prozesse beeinflussen:

- Detaillierungsgrad
- Reife der Architektur
- Änderungsfrequenz
- Dokumentationsbedarf

3. Technologie

- Cloud vs. On-Prem
- Containerisierung
- Frameworks
- Integrationslandschaft

→ Technik limitiert oder ermöglicht Architekturformen.

4. Betriebsmodell

- 24/7 Betrieb → hohe Zuverlässigkeit
- Regulatorische Anforderungen → starke Sicherheit
- Hohe Last → besondere Skalierungsmechanismen

5. Geschäftsstrategie

- Time-to-Market → andere Prioritäten als Robustheit
- Kostenoptimierung → vereinfachte Architekturen

Fehler:

Architektur ohne Organisations- oder Geschäftsbezug planen.

Richtig:

Architektur muss zum Umfeld passen — nicht umgekehrt.

★ LZ 01-06 – Rolle von Architekt:innen im Stakeholder-Umfeld

Was man lernen sollte

- Schnittstellen zwischen Rollen verstehen.
 - Konflikte und Kommunikationswege kennen.
 - Wie Architekt:innen Wert schaffen.
-

Antwort

Architekt:innen befinden sich an einer zentralen Schnittstelle:

1. Fachbereich / Product Owner

- liefern fachliche Anforderungen
- brauchen Übersetzung in technische Konzepte
- erwarten Verständlichkeit und Machbarkeit

2. Entwickelnde

- brauchen klare Strukturen
- erwarten technische Unterstützung
- wollen Freiraum und Innovation

3. QA / Test

- brauchen testbare Architektur
- klare Schnittstellen und deterministisches Verhalten

4. Betrieb / DevOps

- benötigen Stabilität, Monitoring, klare Deployments
- fordern Standardisierung und Automatisierung

5. Projekt- und Produktmanagement

- brauchen Risiken, Zeit- und Budgetabschätzungen
- erwarten Architektur als Planungsbasis

6. Management

- erwartet Wirtschaftlichkeit
- sieht Architektur als strategisches Werkzeug

Konflikte moderieren:

Architekt:innen balancieren Spannungen zwischen allen Gruppen.

Fehler:

Nur Technik sehen und Stakeholder vernachlässigen.

Richtig:

Architektur ist *sozial UND technisch*.

★ LZ 01-07 – Bedeutung von Daten und Datenmodellen

Was man lernen sollte

- Daten als Langfristfaktor erkennen.
 - Die Auswirkungen schlechter Datenmodelle verstehen.
 - Zusammenhang zwischen Daten & Architektur.
 - Datenqualität und -konsistenz modellieren.
-

Antwort

Daten sind oft die langlebigste Komponente eines Systems.
Software kann man ersetzen — **Daten nie vollständig.**

Warum sind Daten so wichtig?

- Datenmodelle bestimmen Systemstruktur.
- Schlechte Datenmodelle verursachen immense technische Schulden.
- Integrationsfähigkeit hängt vom Datenmodell ab.
- Daten entscheiden über Performance (Indexing, Partitionierung, Caching).
- Daten bestimmen Speicher- und Sicherheitsanforderungen.

Typische Auswirkungen:

- Falsche Normalisierung → Redundanzen → Inkonsistenzen
- Zu starke Normalisierung → schlechte Performance
- Falsch gewählte Datentypen → Datenverlust
- Unklare Ownership → Integrationschaos

Architekt:innen müssen:

- Daten modellieren oder Reviewen
- Datenfluss verstehen
- Datenqualität beachten
- Datenmigration planen
- Konsistenzmodelle definieren (strong, eventual, causal)

Beispiel:

Ein fehlerhaftes Kundendatenmodell wirkt sich auf Vertrieb, Buchhaltung, CRM, Marketing und Support aus — Jahrzehnte lang.

Fazit:

Gute Datenarchitektur ist fundamental für jedes langlebige System.

Alles klar — hier ist **Kapitel 2 vollständig überarbeitet, mit deinen Ergänzungswünschen integriert:**

- Qualitätsbäume **klar erklärt**
- Warum & wie sie helfen
- Zielkonflikte korrekt eingebettet
- Die „aktiv berücksichtigen“-Erweiterung in LZ 2-2
- Vollständige maximal ausführliche Version (**Option A**)

- Saubere Struktur und didaktische Tiefe
-

★ KAPITEL 2 – Anforderungen & Randbedingungen

★ LZ 02-01 – Stakeholder-Anliegen verstehen

Was man lernen sollte

- Wer Stakeholder sind
 - Welche Ziele, Interessen und Bedürfnisse sie haben
 - Warum Stakeholder für Architektur so wichtig sind
 - Welche Konflikte zwischen Stakeholdern entstehen
 - Wie Architekt:innen Stakeholder moderieren
 - Wie implizite Erwartungen entstehen
-

Antwort

Stakeholder sind alle Personen oder Gruppen, die Anforderungen, Erwartungen oder Einschränkungen an das System haben oder betroffen sind.

◊ Typische Stakeholdergruppen und ihre Ziele

Stakeholder	Ziele / Bedürfnisse
Projektleitung	Termine, Budget, Risikoarmut
Fachbereich / Product Owner	Business Value, korrekte Funktionen
Endanwender	Usability, Performance, Zuverlässigkeit
Entwickler:innen	moderne Technik, gute Codequalität
QA / Test	Testbarkeit, stabile Schnittstellen
IT-Betrieb / DevOps	Stabilität, Monitoring, einfache Deployments
Management	Wirtschaftlichkeit, strategische Ziele
Architekt:innen	Konsistenz, Transparenz, realistische Anforderungen

◊ Typische Stakeholder-Konflikte

1. Innovation ↔ Stabilität

Entwickler wollen Neues; Betrieb will Bewährtes.

2. Time-to-Market ↔ Qualität

Projektleitung will schnell liefern; Architektur will nachhaltig bauen.

3. Feature-Druck ↔ Usability

Fachbereich will Funktionen; Nutzer wollen Einfachheit.

4. Sicherheit ↔ Komfort

Security will strenge Authentifizierung; Nutzer wollen Einfachheit.

5. Skalierbarkeit ↔ Kosten

Management will Sparen; System muss wachsen können.

◊ Warum Stakeholder so wichtig sind

- Sie definieren Qualitätsziele → Architektur wird davon gesteuert.
 - Ihre Konflikte bestimmen Trade-offs.
 - Missverständnisse führen zu Fehlarchitektur.
 - Architektur ist Kommunikation zwischen Stakeholdern.
-

◊ Fehlerquellen

- ✗ nur technische Stakeholder berücksichtigen
 - ✗ implizite Annahmen nicht klären
 - ✗ Konflikte ignorieren
 - ✗ fehlende Priorisierung
-

★ Essenz:

Stakeholder beeinflussen jede Architekturentscheidung.

Architekt:innen müssen ihre Anliegen verstehen, priorisieren und Konflikte moderieren.

★ LZ 02-02 – Anforderungen & Randbedingungen klären und berücksichtigen

Was man lernen sollte

- Unterschied: funktionale Anforderungen vs. Randbedingungen
 - Arten von Randbedingungen (technisch, organisatorisch, rechtlich...)
 - Warum Randbedingungen nicht verhandelbar sind
 - Wie man sie **aktiv** berücksichtigt
 - Wie man ihre **Auswirkungen** auf Architektur erkennt
 - Dass Architektur neue Randbedingungen erzeugt
-

Antwort

Randbedingungen sind Vorgaben, die den Lösungsraum festlegen.

Beispiele:

- Technologie-Vorgaben („Wir nutzen PostgreSQL“)
 - Compliance („DSGVO“, „FINMA-Richtlinien“)
 - Infrastruktur („Muss On-Premise laufen“)
 - Organisation („Teams sind nach Komponenten strukturiert“)
 - Budget- oder Zeitvorgaben
-

◊ 1. Was bedeutet „aktiv berücksichtigen“?

Aktiv berücksichtigen heißt:

Identifizieren

durch Dokumentanalyse, Interviews, Workshops.

Präzisieren

„Wir müssen DSGVO-konform sein“ → Welche Artikel? Welche Sanktionen?

Validieren

Nachfragen: „Ist diese technologische Vorgabe Pflicht oder Präferenz?“

Analysieren

Wie beeinflusst die Randbedingung Architektur?

Dokumentieren

z. B. in ADRs oder Architekturkonzepten.

Einfließen lassen

Randbedingungen aktiv ins Design integrieren, nicht ignorieren.

2. Was bedeutet „Auswirkungen verstehen“?

Jede Randbedingung hat **architekturelle Konsequenzen**.

Beispiele:

Compliance-Vorgabe (z. B. DSGVO)

- Datenverschlüsselung
- Logging-Regeln
- Datenminimierung
- Löschkonzepte

Betriebsanforderungen (24/7)

- Redundanz
- Monitoring
- Self-Healing
- Load Balancing

Organisatorische Struktur

- Conway's Law = Softwaresystem wird Teamstruktur widerspiegeln

Technologievorgabe

- beeinflusst Performance, Skalierbarkeit, Wartbarkeit
-

◊ 3. Architektur erzeugt selbst neue Randbedingungen

Beispiel:

- Wahl von Microservices → Bedarf an Observability
 - Wahl eines Frameworks → Entwickler müssen geschult werden
 - Cloud-Provider-Entscheidung → Rechtliche & Compliance-Folgen
 - API-First → zwingt zu bestimmten Integrationsstrategien
-

◊ Typische Fehler

- ✗ Randbedingungen nicht prüfen
 - ✗ Auswirkungen nicht verstehen
 - ✗ keine Alternativen prüfen
 - ✗ fehlende Dokumentation
-

★ Essenzen:

Randbedingungen sind Leitplanken der Architektur.
Architekt:innen müssen sie aktiv klären, verstehen und in Entscheidungen integrieren.

★ LZ 02-03 – Qualitäten eines Systems verstehen (ISO 25010)

Was man lernen sollte

- Das Qualitätsmodell ISO 25010
 - Qualitätsmerkmale und ihre Bedeutung
 - Zielkonflikte erkennen
 - Trade-offs treffen
 - Warum Qualität die Architektur dominiert
-

Antwort

ISO/IEC 25010 definiert die zentralen Qualitätsmerkmale eines Softwaresystems:

◊ **Funktionale Eignung**

→ Macht das System das Richtige?

◊ **Performance-Effizienz**

→ Reaktionszeit, Durchsatz, Ressourcenverbrauch

◊ **Zuverlässigkeit**

→ Verfügbarkeit, Fehlerrobustheit, Wiederherstellbarkeit

◊ **Sicherheit**

→ Authentifizierung, Autorisierung, Integrität, Vertraulichkeit

◊ **Usability**

→ Bedienbarkeit, Lernbarkeit, Barrierefreiheit

◊ **Wartbarkeit**

→ Modularität, Testbarkeit, Analysierbarkeit, Modifizierbarkeit

◊ **Kompatibilität**

→ Interoperabilität mit anderen Systemen

◊ **Übertragbarkeit**

→ Portabilität zwischen Plattformen

◊ **Zielkonflikte (Trade-offs) zwischen Qualitätsmerkmalen**

Konflikte entstehen, wenn die Verbesserung eines Qualitätsziels ein anderes verschlechtert:

Qualität A Qualität B

Konflikt

Sicherheit Performance Verschlüsselung erhöht Latenz

Qualität A	Qualität B	Konflikt
Wartbarkeit	Time-to-Market	Modularisierung kostet Zeit
Usability	Sicherheit	MFA erschwert Bedienung
Skalierbarkeit	Kosten	Autoscaling kostet Geld
Portabilität	Performance	Abstraktionen reduzieren Geschwindigkeit

◊ Wie hilft ISO 25010 bei Entscheidungen?

Es zeigt:

- Welche Qualitätstreiber wir haben
- Wo Prioritäten liegen müssen
- Wie Architekturformen ausgewählt werden müssen
- Welche Szenarien notwendig sind

Beispiel:

- Hohe Verfügbarkeit → Cluster-Architektur
 - Hohe Sicherheit → Zero-Trust-System
 - Hohe Performance → Caching, Loadbalancing
 - Hohe Wartbarkeit → modulare Architekturen
-

◊ Trade-offs treffen

Trade-offs sind bewusste Entscheidungen zwischen konkurrierenden Qualitäten.

Architekt:innen müssen:

- Konflikt sichtbar machen
 - mit Stakeholdern priorisieren
 - Entscheidung treffen
 - Dokumentieren (ADR)
 - Konsequenzen vertreten können
-

Essenz:

Qualitätsziele steuern Architektur **viel stärker** als Funktionalität.

LZ 02-04 – Qualitätsanforderungen formulieren

Was man lernen sollte

- Qualitätsszenarien erstellen
 - Qualitätsbäume nutzen
 - Messbare Kriterien definieren
 - Stakeholder interviewen
-

Antwort

Viele Qualitätsanforderungen sind vage wie:

- „System soll schnell sein“
- „Soll sicher sein“
- „Soll wartbar sein“

Das ist nutzlos.

Architekt:innen müssen sie messbar machen.

Qualitätsszenario (vier Elemente)

1. **Stimulus** – das Ereignis
2. **Kontext** – Situation/Bedingung
3. **Reaktion** – was das System tun soll
4. **Messgröße** – wie gut die Reaktion sein soll

Beispiel:

„Unter 2000 gleichzeitigen Nutzern soll die API mit 95% der Anfragen unter 300 ms antworten.“

◊ Was ein Qualitätsbaum hier wirklich leistet

Ein Qualitätsbaum:

- strukturiert Qualitätsmerkmale
- bricht sie in Untermerkmale herunter
- zwingt zur Formulierung von Szenarien
- zeigt Qualitätstreiber
- zeigt fehlende Details
- macht Konflikte sichtbar
- dient als Grundlage für ATAM und Architekturentscheidungen

Beispiel eines Ausschnitts:



Ohne Baum: nur Wörter

Mit Baum: echte Steuerung für Architektur

◊ Fehler

- ✗ Szenarien zu grob
 - ✗ keine Messwerte
 - ✗ keine Priorisierung
 - ✗ Baum oberflächlich, ohne Blätter (Szenarien)
-

★ Essenz:

Qualitätsanforderungen müssen messbar sein — Qualitätsszenarien + Qualitätsbaum sind dafür die Werkzeuge.

★ LZ 02-05 – Explizite Aussagen bevorzugen

Was man lernen sollte

- Implizite Annahmen erkennen
 - Sie explizit dokumentieren
 - Risiken reduzieren
 - Missverständnisse vermeiden
-

Antwort

Implizite Annahmen sind ungesagte Erwartungen wie:

- „Betrieb übernimmt Monitoring.“
- „Entwickler wissen, wie die API funktioniert.“
- „Last wird schon gering bleiben.“
- „Wir werden keine komplexe Skalierung brauchen.“

Diese Annahmen führen zu großen Fehlern.

Architekt:innen müssen:

- nachfragen
- klären
- dokumentieren
- priorisieren
- kommunizieren

Typische Dokumentationsformen:

- ADRs
 - Risiko-Listen
 - Annahmen-Tabellen
 - Architektur-Workshops
-

★ Essenz:

Alles, was nicht explizit gemacht wird, wird falsch interpretiert.

Alles klar — hier ist **Kapitel 3 vollständig**,
mit **ALLEN Lernzielen LZ 03-01 bis 03-12**,
in **Option A (maximal ausführlich)**,
inkl. der überarbeiteten Version von LZ 03-07, die du zuvor angefordert hast.

★ KAPITEL 3 – Entwurf von Softwarearchitekturen

★ LZ 03-01 – Anforderungen durch Architektur erreichen

Was man lernen sollte

- Wie Architektur funktionale und qualitative Anforderungen erfüllt
 - Wie Qualitätsziele Architekturentscheidungen prägen
 - Warum Architektur auf Szenarien basiert
 - Risiken verstehen und adressieren
 - Architektur iterativ entwickeln
-

Antwort

Architektur ist die **Antwort** auf Anforderungen — sowohl funktionale als auch qualitative.

Während Funktionen bestimmen, *was* ein System tut, bestimmen Qualitätsanforderungen *wie gut* es das tun soll.

Architektur erfüllt Anforderungen durch:

- Strukturierung in Bausteine
- Definition von Schnittstellen
- Wahl geeigneter Architeurstile
- Deployment- und Betriebskonzepte
- Datenstrukturen und Integrationsmodelle
- Berücksichtigung von Randbedingungen

Qualitätsszenarien liefern konkrete Prüfmechanismen:

- „2000 Benutzer bei < 300ms“
- „Wiederherstellung innerhalb 5 Minuten“
- „Rollout ohne Downtime“

- „Rolle X darf nur Vorgang Y ausführen“

Architekturentscheidungen wirken auf mehrere Anforderungen gleichzeitig.
Deshalb braucht jede Entscheidung:

- Begründung
- Abgleich mit Qualitätszielen
- Dokumentation
- Risikobewertung

Typische Fehler:

- ✗ Architektur ohne Qualitätsziele entwerfen
 - ✗ Nur Funktionalität berücksichtigen
 - ✗ Trade-offs nicht sichtbar machen
 - ✗ Architektur einmalig entwerfen statt iterativ
-

★ LZ 03-02 – Entwurfsprinzipien guter Architektur

Was man lernen sollte

- Grundprinzipien wie geringe Kopplung, hohe Kohäsion
 - Separation of Concerns, Kapselung, Information Hiding
 - Bedeutung von Einfachheit
 - Warum Entwurfsprinzipien Qualitätsziele erst ermöglichen
-

Antwort

Entwurfsprinzipien sind universelle Regeln für robuste Softwarearchitektur.
Sie machen Systeme:

- verständlich
- wartbar
- erweiterbar
- testbar
- stabil

Die wichtigsten Prinzipien:

◊ Lose Kopplung

Bausteine sollen unabhängig voneinander funktionieren — reduziert Veränderungsaufwand.

◊ Hohe Kohäsion

Ein Baustein hat eine klar abgegrenzte Aufgabe — erhöht Verständlichkeit.

◊ Separation of Concerns

Funktionen und Verantwortlichkeiten werden sauber getrennt — verhindert Chaos.

◊ Information Hiding

Interne Implementierung bleibt verborgen — reduziert Abhängigkeiten.

◊ DRY (Don't Repeat Yourself)

Wissen an einem Ort — weniger Fehler, bessere Lesbarkeit.

◊ YAGNI (You Ain't Gonna Need It)

Nur implementieren, was gebraucht wird — verhindert Overengineering.

◊ Einfachheit

Die wichtigste Architektureigenschaft — Komplexität tötet Systeme.

◊ Fehlertoleranz

Architektur muss Fehler erwarten und überleben können.

Typische Fehler:

- ✗ Prinzipien nur theoretisch kennen
 - ✗ Prinzipien übertreiben (z. B. zu viele Microservices)
 - ✗ Monolith vs. Microservices als Religionsfrage behandeln
-

★ LZ 03-03 – Architekturmuster verstehen

Was man lernen sollte

- Wichtige Architekturstile und Muster
 - Welches Muster welches Problem löst
 - Vor- und Nachteile verschiedener Muster
 - Einfluss der Muster auf Qualität und Betrieb
-

Antwort

Architekturmuster sind bewährte Lösungsansätze, die helfen, wiederkehrende Probleme zu strukturieren.

Beispiele:

Schichtenarchitektur

Klare Trennung von UI, Fachlogik, Persistenz.
→ simpel, klassisch, aber limitiert bei Skalierung.

Microservices

Kleine, autonome Dienste.
→ hochgradig skalierbar, aber komplex in Betrieb, Test, Monitoring.

Event-driven Architecture

Systeme reagieren auf Ereignisse.
→ sehr skalierbar und entkoppelt, aber schwer nachzuvollziehen.

Pipes and Filters

Datenfluss durch Verarbeitungsschritte.
→ ideal für ETL, Transformationsketten.

Microkernel / Plug-in

Kernsystem + Erweiterungen.
→ sehr flexibel, modular.

Musterwahl hängt ab von:

- Qualitätszielen
- Randbedingungen
- Teamkompetenzen
- Integrationslandschaft
- Betriebsmodell

Typische Fehler:

- ✗ Muster modisch auswählen („Wir machen Microservices!“)
 - ✗ Muster ohne Qualitätsbezug einsetzen
 - ✗ Muster mit Technologien verwechseln
-

★ LZ 03-04 – Architekturzentrierte Entwicklungsansätze (z. B. DDD, evolutionäre Architektur)

👉 Antwort

◊ Domain-Driven Design (DDD)

DDD strukturiert Systeme entlang der Fachdomäne:

- Bounded Contexts
- Ubiquitous Language
- Aggregates
- Entities / Value Objects

Vorteile:

- klare Verantwortlichkeiten
 - gute Modularisierung
 - hohe Wartbarkeit
 - bessere Kommunikationsbasis
 - Ideal für Microservices
-

◊ Evolutionäre Architektur

Architektur entsteht **iterativ**, nicht im Voraus.

Wichtige Prinzipien:

- Architektur lernt über Zeit

- Fitness Functions: automatische Überprüfung der Architektur
 - Entscheidungen möglichst spät treffen
 - Änderungen sind normal, nicht unerwünscht
 - Architektur reagiert auf Erkenntnisse aus Implementierung und Betrieb
-

❖ Architektur ist ein Teamprozess

- Architekt:innen coachen
- Teams entwerfen mit
- Entscheidungen werden gemeinsam validiert

Typische Fehler:

- ✗ "Big Design Up Front"
 - ✗ Architektur nicht anpassen
 - ✗ Domäne ignorieren
 - ✗ keine echte Zusammenarbeit mit Fachabteilung
-

★ LZ 03-05 – Entwurfstechniken (Sichten, Szenarien, ADRs, Qualitätsbäume, Analyseverfahren)

👉 Antwort

Moderne Architektur benötigt strukturierte Methoden, um Komplexität zu bewältigen.

1. Sichten (Views)

Die 4 wichtigsten Sichten (bekannt aus arc42):

- **Kontextsicht** – Umfeld, externe Systeme
- **Bausteinsicht** – Struktur und Modularisierung
- **Laufzeitsicht** – Interaktionen, Abläufe
- **Verteilungssicht** – Deployment, Infrastruktur

Sichten reduzieren Komplexität und vermitteln Verständnis.

2. Szenario-basierter Entwurf

Szenarien aus Kapitel 2 treiben Architektur:

- Performance-Szenarien → Caching, Partitioning
 - Sicherheits-Szenarien → Zero Trust, MFA
 - Verfügbarkeitsszenarien → Cluster, Failover-Prozesse
 - Wartbarkeitsszenarien → modulare Entkopplung
-

3. Architekturentscheidungen dokumentieren (ADR)

Jedes wichtige Thema braucht:

- Entscheidung
- Alternativen
- Kontext
- Gründe
- Konsequenzen
- Status

ADRs machen Architektur langfristig nachvollziehbar und auditierbar.

4. Qualitätsbäume im Entwurf

Ein Qualitätsbaum zeigt:

- Qualitätstreiber
- Zielkonflikte
- Prioritäten
- fehlende Szenarien
- Entscheidungsnotwendigkeiten

Dadurch verhindern Qualitätsbäume Fehlentscheidungen.

5. Trade-off-Analysen (ATAM)

Vergleicht:

- wie Entscheidungen Qualitäten fördern oder verletzen
- Risiken

- Sensitivität (welche Entscheidung ist kritisch?)
 - Kompromisse und Alternativen
-

Typische Fehler

- ✗ Architektur ohne Sichten beschreiben
 - ✗ Entscheidungen nicht dokumentieren
 - ✗ Szenarien ignorieren
 - ✗ Qualität nicht messbar machen
-

★ LZ 03-06 – Bedeutung von Daten und Datenmodellen

🎓 Antwort

Daten sind der **wertvollste** und langlebigste Teil eines Softwaresystems.
Schlechter Code kann ersetzt werden — schlechte Daten nicht.

Daten bestimmen Architektur:

- Systemstruktur
 - Integrationen
 - Performance
 - Skalierbarkeit
 - Konsistenzmodelle
 - Sicherheitsmaßnahmen
 - Fachliche Korrektheit
 - Reporting
 - Persistenztechnologien
-

Gute Datenarchitektur:

- modelliert Domänen korrekt
- klärt Ownership („wo gehört welches Datum hin?“)
- minimiert Redundanz
- vermeidet Inkonsistenzen
- definiert klare Schnittstellen

- bestimmt, wie Daten leben, veralten, gelöscht werden
-

Schlechte Datenarchitektur:

- verursacht technische Schulden für Jahre
 - macht Erweiterungen extrem teuer
 - führt zu Fehlern, falschen Daten, Regressionsproblemen
 - erschwert Migrationen
 - erzeugt harte Kopplung zwischen Systemen
-

Wichtige Architekturelemente:

- Normalisierung vs. Denormalisierung
 - Datenqualitätsregeln
 - Datenkonsistenz (stark, eventual, causal consistency)
 - Partitionierung
 - Sharding
 - Change-Data-Capture
 - Datenmigration
-

Typische Fehler

- ✗ Datenmodell vom Entwickler „frei Hand“ machen lassen
 - ✗ Fachliche Inkonsistenzen nicht früh erkennen
 - ✗ Semantik in Code statt in Datenmodell ausdrücken
 - ✗ Redundante Datenhaltung ohne Regeln
-

★ LZ 03-07 – Deployment & Betrieb (DevOps)

🎓 Antwort

Deployment ist Teil der Architektur.

Es bestimmt:

- wie schnell Änderungen ausgerollt werden

- wie stabil ein System ist
- wie gut ein System skaliert
- wie Fehler erkannt und behandelt werden
- wie teuer das System im Betrieb ist
- ob Architekturentscheidungen überhaupt funktionieren können

Die wichtigsten Bereiche:

1. CI/CD

- Builds müssen reproduzierbar sein
 - Tests automatisiert
 - Releases versioniert
 - Deployments automatisiert
 - Keine manuellen Schritte
-

2. Deployment-Strategien

- Blue/Green
- Canary
- Rolling Update
- Recreate

Jede Strategie setzt architekturelle Eigenschaften voraus (z. B. stateless Services, Observability).

3. Infrastrukturbabhängigen

Architektur muss anpassen an:

- Cloud vs. On-Prem
 - Kubernetes vs. klassische Server
 - Serverless vs. Container
 - Automatisierungsmöglichkeiten
-

4. Observability

- Logging
- Metrics

- Tracing
- Health Checks
- Alerts

Ohne Observability: Blindflug.

Mit Observability: Kontrollierte verteilte Systeme.

5. Fehlertoleranz

Architektur muss Fehler erwarten und abfangen:

- Circuit Breaker
 - Timeouts
 - Retries
 - Bulkheads
 - Backpressure
-

6. Betrieb als Partner

Architektur muss mit Betrieb abgestimmt werden:

- Monitoring-Anforderungen
 - Deployment-Zeitfenster
 - Security-Vorgaben
 - Infrastrukturkapazitäten
-

Typische Fehler:

- ✗ Deployment erst am Ende betrachten
 - ✗ Kein Logging/Monitoring
 - ✗ Architektur nicht deploybar (z. B. stateful Services)
 - ✗ Microservices ohne DevOps-Fähigkeiten einführen
-

★ LZ 03-12 – Herausforderungen verteilter Systeme

🎓 Antwort

Verteilte Systeme sind *grundlegend anders* als monolithische Systeme.

1. Das Netzwerk ist unzuverlässig

- Timeouts
- verlorene Nachrichten
- Retries notwendig

2. Keine globale Zeit

- Zeitstempel sind unsicher
- Kausalität muss anders erkannt werden

3. Latenz ist variabel

- Architektur muss damit umgehen

4. Konsistenzprobleme (CAP, PACELC)

- Konsistenz ↔ Verfügbarkeit ↔ Partitionstoleranz
- eventual consistency

5. Komplexere Fehlermodi

- Split-brain
- teilweiser Ausfall
- emergentes Verhalten

6. Neues Architekturdenken

- Idempotenz
 - Distributed Tracing
 - Monitoring auf Service-Ebene
 - resiliente Patterns
-

Typische Fehler:

- ✗ Microservices wählen ohne zu verstehen, dass Verteiltheit das System *schwerer* macht
 - ✗ Fehlertoleranz nicht einbauen
 - ✗ Konsistenzmodell ignorieren
-

Alles klar — hier ist **Kapitel 4 vollständig, NEU aufgebaut, maximal ausführlich, klar strukturiert, inklusive der verbesserten Erklärung zu C4 und der korrekten LZ-Nummern 04-01 bis 04-09** gemäß *iSAQB Curriculum Foundation 2025*.

★ KAPITEL 4 – Beschreibung & Kommunikation von Softwarearchitekturen

★ LZ 04-01 – Anforderungen an technische Dokumentation

⌚ Was man lernen muss

- Kriterien guter Dokumentation
- Zielgruppen bestimmen
- Klarheit, Präzision, Aktualität
- Zweckorientierte Auswahl der Inhalte
- Konsistenz zwischen gesprochener und geschriebener Architektur

🌐 Antwort

Gute Architekturdokumentation ist **zielgruppengerecht, klar, aktuell, verständlich und begründet**.

Wichtige Eigenschaften:

✓ Zielgruppenorientierung

- **Entwickler:** Struktur, Entscheidungen, Schnittstellen
- **Betrieb/DevOps:** Deployment, Monitoring, Logging

- **Fachbereich:** Kontext, Domäne
- **Management:** Risiken, Alternativen, Roadmap

✓ **Präzise & eindeutig**

Statt „System soll schnell sein“:
„95 % aller Requests < 200 ms bei 2.000 Nutzern“.

✓ **Konsistent & widerspruchsfrei**

Alle Sichten müssen zusammenpassen.

✓ **Schlank & aktuell**

Leicht zu pflegen, nicht überladen.

✓ **Begründend**

Nicht nur *was*, sondern *warum*.

★ LZ 04-02 – Softwarearchitekturen beschreiben und kommunizieren

⌚ Was man lernen muss

- Architektur verständlich formulieren
- Mündliche und schriftliche Kommunikation kombinieren
- Stakeholder mit unterschiedlichem Wissen abholen

🗣 Antwort

Architektur ist zu **50 % Technik** und zu **50 % Kommunikation**.

Architekt:innen müssen:

✓ **Komplexes einfach erklären**

- ohne unnötigen Fachjargon
- Beispiele & Bilder nutzen

Stakeholder-orientiert sprechen

- Fachbereich will Ergebnisse & Risiken
- Entwickler wollen Details
- Betrieb will technische Abläufe
- Management will Entscheidungen & Auswirkungen

Kommunikationsformen mischen

- Gespräche / Workshops (Schnell, interaktiv)
- Diagramme (visuell)
- Dokumente (dauerhaft, auditiert)

Missverständnisse aktiv vermeiden

- Gegenfragen stellen
- Zusammenfassen
- Visualisieren

Gute Kommunikation steigert Qualität, Geschwindigkeit und Teamharmonie.

LZ 04-03 – Notations- und Modellierungsmittel erläutern und anwenden

Was man lernen muss

- UML, BPMN, C4, ERD, informelle Notationen
- Wofür jede Notation gut geeignet ist
- Wann man welches Modell einsetzt
- Wie Notationen Missverständnisse vermeiden

Antwort

Warum Modelle wichtig sind

Software ist unsichtbar — Modelle machen sie sichtbar.
Modelle helfen dir:

- Komplexität zu reduzieren
- Fehler zu vermeiden

- Architektur schneller zu erklären
 - Stakeholder auf denselben Wissensstand zu bringen
 - Systeme planbar zu machen
-

★ Wofür welche Notation gut ist

◊ **C4-Modell (leicht, intuitiv, architekturzentriert)**

👉 Das wichtigste Modell für iSAQB und moderne Architektur.

Ich benutze deine Google-Maps-Metapher:

★ Level 1 – System Context (Landkarte)

Was ist unser System? Wer nutzt es? Welche Systeme sind außen?

★ Level 2 – Container (Stadt)

Aus welchen deploybaren Teilen besteht das System?

Web-App, Backend-Service, DB, Message-Broker, Mobile-App ...

★ Level 3 – Component (Straßen)

Wie ist ein Container intern strukturiert?

Controller, Services, Repositories, Adapter ...

★ Level 4 – Code (Gebäude)

Optional! Klassen/Methoden nur bei Bedarf.

C4 ist ideal für Kommunikation, Überblick, Verständlichkeit.

◊ **UML**

Präzise technische Diagramme.

- **Komponentendiagramm** → Architekturbausteine
- **Sequenzdiagramm** → Abläufe
- **Klassendiagramm** → Daten-/Objektstrukturen

- **Deploymentdiagramm** → Container/Server/Nodes

UML eignet sich für tiefere technische Zielgruppen.

◊ **BPMN**

Geschäftsprozesse darstellen:

- Fachlogik
- Rollen
- Entscheidungen
- Workflows
- Trigger & Events

Hilft, die Domäne korrekt zu verstehen.

◊ **ERD / Domain Model**

Zeigt:

- Entitäten
- Beziehungen
- Kardinalitäten
- Domänensemantik

Grundlage für Datenintegrität und Konsistenz.

◊ **Informelle Modelle**

Whiteboard, Kästchen & Pfeile.

Ideal für Workshops, Brainstorming und frühe Gespräche.

★ **Essenz:**

Jede Notation beantwortet eine andere Frage.
Niemals alles in einem Diagramm vermischen.

★ LZ 04-04 – Unerwartete Situationen geschickt bewältigen

⌚ Was man lernen muss

- Stresssituationen erkennen
- Priorisieren & improvisieren
- Krisenkommunikation
- Unsicherheiten managen

🌐 Antwort

Architekt:innen müssen auch **Krisenmoderatoren** sein.

Unerwartete Situationen:

- Architektureinschränkungen treten plötzlich auf
- Anforderungen ändern sich
- Technik fällt aus
- Deadlines rücken vor
- Stakeholder widersprechen sich
- Integration scheitert
- Release bricht zusammen

Dafür braucht man:

- Ruhe & Übersicht
- schnelles Priorisieren
- transparentes Kommunizieren
- Alternativen präsentieren
- technische Risiken benennen
- Entscheidungen faktenbasiert treffen

Kurz:

Architekt:innen behalten die Kontrolle im Chaos.

★ LZ 04-05 – Architektsichten erläutern

⌚ Was man lernen muss

- Warum Sichten nötig sind
- Die vier Kern-Sichten
- Konsistenz zwischen den Sichten

- Stakeholderorientierte Darstellung

Antwort

Architektsichten sind **verschiedene Perspektiven auf dasselbe System**, jede für eine andere Fragestellung optimiert.

Die vier Kern-Sichten:

1. Kontextsicht

„Was liegt innen, was außen?“
→ Systeme, Benutzer, Abgrenzung

2. Bausteinsicht

„Wie ist das System strukturiert?“
→ Module, Komponenten, Verantwortlichkeiten

3. Laufzeitsicht

„Wie arbeitet das System zur Laufzeit?“
→ Sequenzen, Interaktionen, Event Flows

4. Verteilungssicht (Deployment)

„Wie läuft das System technisch?“
→ Container, Server, Cloud, Netzwerk

Sichten sind notwendig, weil:

- unterschiedliche Stakeholder unterschiedliche Sichtweisen brauchen
 - sie Komplexität reduzieren
 - sie Entscheidungen erklärbar machen
 - sie Architektur prüfbar machen
-

★ LZ 04-06 – Schnittstellen dokumentieren

⌚ Was man lernen muss

- Semantik
- API-Signatur (nicht Code!)
- Fehlerfälle
- Qualitätsanforderungen
- Sicherheit & Versionierung

🧠 Antwort

Eine Schnittstelle ist ein **Vertrag**.
Sie besteht aus drei Teilen:

1 Semantik (wichtigste Ebene)

Antwortet: „Was bedeutet diese Operation fachlich?“

Beispiel:

„POST /orders“ erzeugt Bestellung, prüft Zahlungsgrenze, validiert Artikel.“

2 API-Signatur (vertragliche Oberfläche)

Endpoint + Request/Response (in abstrahierter Form).

Beispiel:

```
POST /orders
Input: {...}
Output: {...}
```

Nicht Code → aber klar definierte Struktur.

3 Nichtfunktionale Anforderungen

- Latenz
- Sicherheit / Auth / Rollen

- Idempotenz
 - Fehlercodes
 - Versionierung
 - Deprecation-Strategie
-

★ LZ 04-07 – Querschnittsthemen dokumentieren

⌚ Was man lernen muss

- Querschnittsasspekte erkennen
- zentral dokumentieren
- konsistent anwenden

⌚ Antwort

Querschnittsthemen wirken überall:

- Logging
- Monitoring
- Fehlerhandling
- Sicherheit
- Caching
- Persistenz
- Konfiguration
- Nebenläufigkeit
- Messaging

Diese müssen **einheitlich** und **zentral** definiert werden, sonst entstehen widersprüchliche Implementierungen.

★ LZ 04-08 – Architekturentscheidungen dokumentieren

⌚ Was man lernen muss

- Entscheidungen nachvollziehbar machen
- Alternativen darstellen
- Konsequenzen beschreiben

Antwort

Architekturentscheidungen müssen langfristig verstehtbar sein.

Standardform: **ADR – Architecture Decision Record**

Beinhaltet:

- Entscheidung
- Kontext
- Alternativen
- Gründe
- Konsequenzen
- Risiken
- Status

Ziel:

Transparenz & Nachvollziehbarkeit.

LZ 04-09 – Werkzeuge & Frameworks zur Dokumentation

Was man lernen muss

- Frameworks: arc42, C4, ISO 42010
- Diagramm-Tools
- Textuelle Modellierung
- Checklisten

Antwort

arc42

Template für vollständige Architekturdokumentation.

C4

Standard für verständliche Architekturdiagramme.

ISO 42010

Norm für Architekturbeschreibung.

Tools

- PlantUML
- Mermaid
- Structurizr
- Draw.io
- Confluence

Checklisten

prüfen Vollständigkeit & Qualität.

KAPITEL 5 – Analyse & Bewertung von Softwarearchitekturen

LZ 05-01 – Qualitätsattribute analysieren

Was man lernen muss

- Wie man qualitative Anforderungen prüft
- Wie Architekturentscheidungen Qualität beeinflussen
- Wie man Qualität **messbar** macht
- Wie man Qualitätsszenarien verwendet
- Wie man Qualität mit Szenarien bewertet (Stimulus → Kontext → Reaktion → Messgröße)
- Wie man Zielkonflikte sichtbar macht

Antwort

Qualitätsattribute sind **der Kern der Architektur**.

Sie bestimmen, welche Strukturen, Technologien und Betriebsmodelle sinnvoll sind.

Beispielhafte Qualitätsattribute:

- Performance
- Sicherheit
- Wartbarkeit
- Testbarkeit
- Zuverlässigkeit
- Skalierbarkeit

- Portabilität
- Nutzbarkeit
- Modifizierbarkeit

Um sie bewerten zu können, nutzt man **Qualitätsszenarien**:

Ein Qualitätsszenario hat vier Teile:

1. **Stimulus** – Ein Ereignis (z. B. „2.000 Nutzer senden gleichzeitig Anfragen“)
2. **Kontext** – Die Situation („System ist im Normalbetrieb“)
3. **Reaktion** – Wie das System antworten soll
4. **Messgröße** – Wie gut („95 % < 300 ms“)

Anhand solcher Szenarien erkennt man:

- Kann die Architektur die Erwartungen erfüllen?
- Wo drohen Risiken?
- Welche Entscheidungen beeinflussen welche Qualität?

Qualitätsanalyse zeigt auch **Konflikte**:

- Sicherheit ↔ Bedienbarkeit
- Performance ↔ Kosten
- Wartbarkeit ↔ Time-to-Market
- Skalierung ↔ Konsistenz

Eine Architektur wird anhand dieser Szenarien **evaluiert, optimiert oder angepasst**.

★ LZ 05-02 – Risiken erkennen, kommunizieren und behandeln

⌚ Was man lernen muss

- Was ein Risiko ist („ungewisses Ereignis mit Auswirkungen“)
- Unterschied zwischen Risiko und Problem
- Technische, organisatorische, fachliche Risiken
- Risikokategorien und Risikotreiber
- Maßnahmen: vermeiden, reduzieren, akzeptieren, übertragen
- Risiken bewerten und priorisieren

Antwort

Architektur ist **Risikomanagement**.

Ein Risiko ist etwas, das *schiefgehen kann*, aber *noch nicht schiefgegangen ist*.

Typische Risiken:

◊ Technische Risiken

- Technologie unerprobt
- Performance unzureichend
- Integrationsprobleme
- Single Points of Failure
- Komplexer Build/Deployment-Prozess

◊ Organisatorische Risiken

- fehlende Skills im Team
- mangelhafte Abstimmung
- unklare Zuständigkeiten
- Abhängigkeiten zu externen Teams

◊ Fachliche Risiken

- unklare Anforderungen
- widersprüchliche Stakeholder
- instabile Geschäftslogik

Architekt:innen müssen:

1. **Risiken identifizieren**
2. **Risiken bewerten** (Eintrittswahrscheinlichkeit × Schaden)
3. **Gegenmaßnahmen entwickeln:**
 - **vermeiden** (andere Technologie wählen)
 - **reduzieren** (Prototypen, Tests, Monitoring)
 - **akzeptieren** (bewusster Trade-off)
 - **übertragen** (z. B. Managed Services, externe Experten)
4. **Risiken laufend kommunizieren**

Ein nicht kommuniziertes Risiko wird später garantiert zum Problem.

★ LZ 05-03 – Architektur bewerten (inkl. ATAM)

⌚ Was man lernen muss

- Wie man Architekturen strukturiert bewertet
- ATAM als Bewertungsmethode
- Qualitätsszenarien priorisieren
- Trade-offs und Sensitivitäten erkennen
- Risiken, Nicht-Risiken, Trade-offs dokumentieren

🧠 Antwort

Architekturbewertung bedeutet, systematisch zu prüfen:

- Erfüllt die Architektur die Qualitätsziele?
- Wo sind Risiken?
- Wo sind Zielkonflikte?
- Welche Entscheidungen sind kritisch?
- Wie robust ist die Architektur?

Der Standard dafür ist **ATAM (Architecture Tradeoff Analysis Method)**.

★ ATAM – Die 9 Schritte (vereinfacht)

Phase 1 – Vorbereitung

1. **Einführung** – Ziele und Kontext des Systems
2. **Architekturvorschlag** – Überblick, Sichten, wichtige Entscheidungen
3. **Business Goals sammeln** – Qualitätsziele priorisieren

Phase 2 – Bewertung

4. **Qualitätsszenarien sammeln**
(Performance, Sicherheit, Wartbarkeit, Verfügbarkeit ...)
5. **Architekturanalyse**
 - Welche Entscheidungen sind kritisch?
 - Wo gibt es Abhängigkeiten?
 - Wo drohen Risiken?
6. **Bewertungsszenarien durchspielen**
 - Wie reagiert das System auf Szenarien?
 - Welche Bausteine sind betroffen?
7. **Risiken und Trade-offs identifizieren**

- Wo gibt es Zielkonflikte?
 - Wo sind strukturelle Schwächen?
8. **Nicht-Risiken (positive Erkenntnisse) festhalten**
- Was funktioniert gut?

Phase 3 – Abschluss

9. **Ergebnisse zusammenfassen**
- Risiken
 - Nicht-Risiken
 - Trade-offs
 - Empfohlene Maßnahmen
-

★ Warum ATAM so wertvoll ist

- macht Probleme sichtbar, BEVOR sie teuer werden
- bringt Stakeholder und Technik an einen Tisch
- zeigt versteckte Konflikte und Risiken
- priorisiert Qualitätsziele
- verbessert Architekturentscheidungen
- liefert strukturiertes Feedback für Designverbesserungen

ATAM ist **kein Dokument**, sondern ein **Workshop-Prozess**, der die Architekturqualität erhöht.

★ LZ 05-04 – Technische Schulden erklären und managen

⌚ Was man lernen muss

- Was technische Schulden sind
- Ursachen und Arten technischer Schulden
- Schulden sichtbar machen
- Priorisieren & Managementstrategien
- Strategien zum Abbau

🌐 Antwort

Technische Schulden sind **bewusste oder unbewusste Kompromisse**, die zu höherem zukünftigen Aufwand führen.

Beispiele:

- „Schnell hacken“ für ein Release
- unklare Modulgrenzen
- fehlende Tests
- Copy-Paste-Code
- temporäre Lösungen, die dauerhaft bleiben
- alte Framework-Versionen
- unklare Schnittstellen

Wichtig:

Technische Schulden sind NICHT schlecht.

Sie sind nur schlecht, wenn sie **unsichtbar** oder **unkontrolliert** sind.

Das Problem entsteht nicht durch Schulden,

sondern durch **Zinsen**:

Wartbarkeit sinkt, Risiko steigt, mehr Bugs, mehr Aufwand.

★ Arten technischer Schulden

◊ Unbewusste Schulden

Fehler, mangelnde Erfahrung, schlechte Entscheidungen.

◊ Bewusste Schulden („strategische Schulden“)

„Wir liefern jetzt schnell und reparieren später.“

◊ Prozess-Schulden

fehlerhafte CI/CD-Pipelines, fehlende Automatisierung.

◊ Architektur-Schulden

z. B. ungeeigneter Architekturstil, zu starke Kopplung.

★ Schulden managen bedeutet:

1. **sichtbar machen**
(Backlog, Architektur-Board, ADRs)
 2. **bewerten**
(Kosten vs. Risiko vs. Nutzen)
 3. **priorisieren**
 4. **geplante Tilgung**
(Refactor-Sprints, Boy Scout Rule)
 5. **neue Schulden bewusst eingehen**
 6. **Zinsen kontrollieren**
-

★ Warum dieses Thema wichtig ist

Schlecht gemanagte Schulden verursachen:

- langsame Entwicklung
- fragile Releases
- steigende Kosten
- unzufriedene Stakeholder
- Qualitätseinbußen
- gefährliche Architekturdefekte

Gute Architekt:innen machen Schulden **sichtbar, steuerbar und beherrschbar**.

Alles klar — hier ist **KAPITEL 6 – Beispiele für Softwarearchitekturen**, vollständig, ausführlich und perfekt passend zu den Lernzielen **LZ 06-01** und **LZ 06-02**, so wie sie im offiziellen Curriculum stehen

(Quelle: *iSAQB Curriculum Foundation 2025*).

Ich liefere dir:

- **Was man lernen muss**
 - **Die ausführliche Antwort wie für Studierende**
 - **Ein durchgängiges Beispielsystem (Webshop)**
→ damit du LZ 06-01 & LZ 06-02 wirklich verstehst
 - **Bezug auf Anforderungen → Entscheidungen**
 - **Bezug auf Entscheidungen → Umsetzung**
-

★ KAPITEL 6 – Beispiele für Softwarearchitekturen

★ LZ 06-01 – Bezug von Anforderungen und Randbedingungen zur Lösung erfassen (R3)

⌚ Was man lernen muss

- Anforderungen und Randbedingungen identifizieren
 - Erkennen, welche Anforderungen eine bestimmte Architekturentscheidung ausgelöst haben
 - Nachvollziehen, warum die gewählte Lösung sinnvoll war
 - Ursache–Wirkung verstehen:
Anforderung → Entscheidung → Konsequenz
-

⌚ Antwort

Architektur entsteht **nie im luftleeren Raum**.

Jede Lösung ist die Antwort auf:

- funktionale Anforderungen
- Qualitätsziele
- Randbedingungen (z. B. Technologien, Organisation, Regulierung)

Die Aufgabe eines/einer Softwarearchitekt:in ist es zu verstehen:

Welche konkrete Anforderung hat zu welcher Entscheidung geführt?
Welche Einschränkung hat die Lösungsrichtung geprägt?

★ Beispielsystem zur Illustration:

Ein Webshop für digitale Produkte

Wir nutzen dieses Beispiel durchgehend in Kapitel 6.

Anforderungen (Auszug)

1. Kund:innen sollen Produkte kaufen können (funktional).

2. Bezahlung muss *hochverfügbar* und *sicher* sein (Qualität).
 3. System muss mindestens **10.000 gleichzeitige Benutzer** unterstützen (Last).
 4. System muss auf bestehender Firmen-Cloud-Infrastruktur laufen (Randbedingung).
 5. Time-to-Market: MVP innerhalb von **3 Monaten** (Projektvorgabe).
-

★ Wie Anforderungen → Architekturentscheidungen führen

Anforderung 2: Bezahlung muss hochverfügbar sein

→ Entscheidung: **Payment-Service wird separat gekapselt**
(z. B. eigenes Deployment)

Begründung:

Bezahlung ist kritisch — Ausfall darf nicht den gesamten Shop lahmlegen.

Anforderung 3: 10.000 Nutzer parallel

→ Entscheidung: **Web-Frontend wird horizontal skalierbar**
(z. B. Container + Load Balancer)

Anforderung 4: Cloud-Vorgabe

→ Entscheidung: Verwendung von PostgreSQL und Kubernetes, da vorgegeben.

Anforderung 5: MVP in 3 Monaten

→ Entscheidung: **Monolithischer Kern + ausgelagerte kritische Services**
statt direkt Microservices überall.

Anforderung 1: Kaufen von Produkten

→ Entscheidung: **Domänenzerlegung nach DDD**

- Bounded Context “Catalog”
- Bounded Context “Order”
- Bounded Context “Payment”

So ergibt sich eine begründete Architektur — **jede Entscheidung ist nachvollziehbar.**

★ Zusammenfassung LZ 06-01 (Merksatz)

Architekturentscheidungen müssen immer auf Anforderungen und Randbedingungen zurückführbar sein.

Jede gute Lösung kann erklärt werden als:

„Weil Anforderung X und Randbedingung Y existieren, haben wir Entscheidung Z getroffen.“

★ LZ 06-02 – Technische Umsetzung einer Lösung nachvollziehen (R3)

⌚ Was man lernen muss

- Wie Architekturentscheidungen technisch umgesetzt werden
 - Wie Implementierung, Produkte, Frameworks, Deployment etc. sichtbar machen, **was vorher entschieden wurde**
 - Wie Lösungskonzepte im Code und in der Infrastruktur genutzt werden
 - Wie man Code- und Architekturartefakte lesen und verstehen kann
-

🌐 Antwort

Architektur darf nicht Theorie bleiben.
Sie zeigt sich **in der echten Umsetzung**:

- im Code
- in Services
- in der Datenbank
- in APIs
- im Deployment
- in Logs
- in Konfigurationen

Die Aufgabe ist:

„Erkenne, wie die Lösung umgesetzt wurde und warum sie so gebaut ist.“

★ Konkrete Umsetzung im Beispiel-Webshop

Wir zeigen für jede wichtige Entscheidung aus LZ 06-01 die technische Umsetzung.

1. Entscheidung: Payment-Service ausgelagert

Technische Umsetzung:

- eigener Container `payment-service`
- REST- oder Messaging-API
- eigene Datenbanktabelle „transactions“
- Health Checks + Retry-Mechanismen
- Isolation: bei Fehlern funktioniert der Shop weiterhin

👉 Umsetzung sichtbar in:

- Deployment (Kubernetes Deployment + Service)
 - Code (separates Repository oder separater Ordner)
 - API-Spezifikation
-

2. Entscheidung: horizontale Skalierung

Technische Umsetzung:

- Docker-Container für das Frontend
- Kubernetes ReplicaSet
- Load Balancer davor
- stateless implementiert (Session im Redis oder JWT)

👉 Umsetzung sichtbar in:

- Infra-Code (Helm Charts / YAML)
 - Architekturdiagramm (Verteilungssicht)
-

3. Entscheidung: DDD-Zerlegung (Catalog, Order, Payment)

Technisch sichtbar in:

- unterschiedliche Paketstrukturen / Module
 - getrennte Datenmodelle
 - getrennte Services
 - getrennte Tests
-

4. Entscheidung: PostgreSQL (durch Randbedingung)

Technisch sichtbar in:

- docker-compose.yml oder Helm-Chart
 - JDBC-Verbindungen
 - Migrations-Skripte (Flyway, Liquibase)
-

5. Entscheidung: Monolithischer Kern für Time-to-Market

Technisch sichtbar in:

- ein Haupt-Repository
 - interne Module statt externe Services
 - Build in einem Artefakt
-

★ Woran erkennt man gute Umsetzungskompetenz?

- Architekturdiagramme und Code passen zusammen
 - Benannte Konzepte existieren wirklich
 - Entscheidungen finden sich in ADRs oder Kommentaren wieder
 - Module sind so geschnitten, wie entworfen
 - Infrastruktur entspricht den Deployments
-

Zusammenfassung LZ 06-02 (Merksatz)

Wer Architekturen versteht, kann aus Code, Deployments, APIs und Strukturen erkennen, **wie** eine Lösung realisiert wurde und **welche Entscheidungen** dahinterstehen.

Repetition & Review

Kapitel 1 – Grundbegriffe von Softwarearchitektur

LZ 01-01 (R1): Definitionen von Softwarearchitektur verstehen

Kurz zusammengefasst:

Softwarearchitektur beschreibt die wesentliche Struktur eines Systems: seine Bausteine, deren Beziehungen und die grundlegenden technischen Entscheidungen.

Ausführliche Erklärung:

Softwarearchitektur legt fest, wie ein System in Bausteine zerlegt ist, wie sie zusammenarbeiten, auf welchen Technologien sie basieren und welche Prinzipien sie leiten. Sie ist das Ergebnis bewusster Entscheidungen – nicht „zufällige Implementierung“.

Eine Architektur dient als Abstraktionsebene oberhalb des Codes und schafft Orientierung für Teams. Sie definiert Struktur, Kommunikationsformen, Datenhaltung, technische Leitplanken und Qualitätsorientierung. Dadurch entsteht ein gemeinsames Verständnis des Systems, das Weiterentwicklung und Zusammenarbeit erleichtert.

Reflexionsfrage:

Welche groben Bausteine würdest du in einem einfachen Webshop identifizieren und wie interagieren sie?

Musterlösung / möglicher Ansatz:

- **Frontend/Presentation Layer** (Web/Mobile)
- **Backend Services**, z. B. Order-Service, Customer-Service
- **Payment Integration**

- **Database Layer** pro Domäne
Kommunikation meist über REST oder Messaging. Vorteil: klare Verantwortlichkeiten, bessere Wartbarkeit.
-

LZ 01-02 (R1): Ziele und Nutzen von Softwarearchitektur verstehen und erläutern

Kurz zusammengefasst:

Softwarearchitektur sorgt für Qualität, Stabilität, Wartbarkeit und reduziert langfristige Risiken.

Ausführliche Erklärung:

Gute Architektur ist ein zentraler Erfolgsfaktor. Sie bestimmt, wie flexibel ein System bleibt, wie schnell Funktionen hinzugefügt werden können und wie robust es läuft. Architektur legt grundlegende Strukturen fest, die Erweiterbarkeit, Performance, Sicherheit, Ausfallsicherheit oder Testbarkeit beeinflussen. Fehlende oder schlechte Architektur führt zu technischen Schulden, langsamer Entwicklung und teuren Änderungen. Daher müssen Architekturentscheidungen bewusst getroffen, begründet und dokumentiert werden.

Reflexionsfrage:

Welche frühen Architekturentscheidungen unterstützen später eine einfache Erweiterbarkeit?

Musterlösung / möglicher Ansatz:

- Domänenorientierte Struktur statt technischer Schichten
 - Klare und stabile Schnittstellen
 - Gekapselte Datenzugriffe pro Domäne
 - API-First-Ansatz
- Dadurch können Teile später leichter aus gegliedert oder ersetzt werden.
-

LZ 01-03 (R3): Langfristige Auswirkungen von Softwarearchitektur kennen

Kurz zusammengefasst:

Architekturentscheidungen wirken langfristig – sie ermöglichen oder behindern spätere Änderungen.

Ausführliche Erklärung:

Architektur schafft Strukturen, die oft über viele Jahre bestehen. Fehlerhafte Strukturentscheidungen führen zu technischen Schulden, komplexen Abhängigkeiten und teuren Refactorings. Gute Architektur reduziert hingegen Risiken und erlaubt kontrollierte Weiterentwicklung.

Zu verstehen, welche Entscheidungen spätere Kosten beeinflussen, ist eine Kernkompetenz von Architekt:innen. Architektur betrifft also immer auch „Zukunftssicherheit“ und Wartbarkeit.

Reflexionsfrage:

Welche architektonische Entscheidung könnte in Zukunft besonders hohe Kosten nach sich ziehen, wenn sie heute falsch getroffen wird?

Musterlösung / möglicher Ansatz:

Eine ungeeignete Persistenzstrategie (z. B. inkonsistente Datenmodelle, zu enge Kopplung an eine Datenbank) macht Migration, Skalierung und Integrationen später extrem teuer.

LZ 01-04 (R1): Aufgaben und Verantwortung von Softwarearchitekt:innen verstehen

Kurz zusammengefasst:

Architekt:innen treffen wesentliche technische Strukturentscheidungen, managen Risiken und kommunizieren technische Leitplanken.

Ausführliche Erklärung:

Die Rolle umfasst die Verantwortung für architekturrelevante Entscheidungen, das Moderieren technischer Konflikte, das Festlegen von Leitplanken, Risikoabschätzung sowie die Dokumentation und Kommunikation der Architektur.

Architekt:innen sind keine „Einzelkämpfer“, sondern arbeiten eng mit Entwickler:innen, Management, Betrieb und Fachbereichen zusammen. Sie müssen technische Tiefe mit Kommunikationsfähigkeit verbinden.

Reflexionsfrage:

Welche Entscheidungen in deinem bisherigen Projekt wären eindeutig architekturrelevant gewesen?

Musterlösung / möglicher Ansatz:

- Wahl des Kommunikationsmodells (REST vs. Messaging)
- Entscheidung über Monolith vs. Verteilt

- Sicherheitskonzept (z. B. AuthN/AuthZ)
Diese Entscheidungen prägen viele Teile des Systems und beeinflussen langfristig Qualität und Wartbarkeit.
-

LZ 01-05 (R3): Abgrenzung zu anderen Architekturdomänen

Kurz zusammengefasst:

Softwarearchitektur grenzt sich von Geschäfts-, Enterprise- und Infrastrukturarchitektur ab, muss aber mit ihnen zusammenwirken.

Ausführliche Erklärung:

Mehrere Architekturdisziplinen bestimmen IT-Landschaften:

- **Enterprise-Architektur:** strategische Technologieentscheidungen
- **Geschäftsarchitektur:** Prozesse, Produkte, Verantwortlichkeiten
- **Infrastrukturarchitektur:** Cloud, Netzwerke, Betrieb
Softwarearchitektur gestaltet Systeme innerhalb dieser Rahmenbedingungen.
Sie setzt die Vorgaben der Enterprise-Architektur um, unterstützt Geschäftsprozesse und berücksichtigt Anforderungen der Infrastruktur. Ein gutes Verständnis der Schnittstellen zu diesen Domänen ist entscheidend.

Reflexionsfrage:

Welche Architekturdomäne hat in deiner Organisation den größten Einfluss auf Softwarearchitektur?

Musterlösung / möglicher Ansatz:

In vielen Unternehmen gibt die Enterprise-Architektur Standards vor (z. B. Cloud-Anbieter, Messaging-Systeme).
In Unternehmen mit hoher Prozesskomplexität dominiert die Geschäftsarchitektur (z. B. BPM-Modelle, Value Streams).

LZ 01-06 (R1): Rolle von Softwarearchitekt:innen mit anderen Stakeholdern in Beziehung setzen

Kurz zusammengefasst:

Architekt:innen vermitteln zwischen Stakeholdern und finden tragfähige technische Lösungen für unterschiedliche Anforderungen.

Ausführliche Erklärung:

Stakeholder (z. B. Nutzer:innen, PO, Business, Security, Betrieb) haben häufig widersprüchliche Ziele. Architekt:innen müssen diese verstehen, priorisieren und Kompromisse transparent machen.
Sie kommunizieren technische Zusammenhänge verständlich und sorgen dafür, dass alle Stakeholder Architekturentscheidungen nachvollziehen können. Gute Zusammenarbeit ist entscheidend, weil Architektur iterativ entsteht und sich an neue Erkenntnisse anpasst.

Reflexionsfrage:

Wie kannst du sicherstellen, dass alle Stakeholder denselben Wissensstand über die Architektur haben?

Musterlösung / möglicher Ansatz:

Regelmäßige Architektur-Reviews, vereinheitlichte Dokumentation (z. B. arc42), ADRs, technische Refinements und Stakeholder-spezifische Präsentationen.

LZ 01-07 (R2): Bedeutung von Daten und Datenmodellen kennen

Kurz zusammengefasst:

Daten und ihre Modelle beeinflussen nahezu alle Architekturentscheidungen – von Konsistenz bis Skalierbarkeit.

Ausführliche Erklärung:

Datenmodelle definieren Struktur, Integrität und Lebenszyklen der wichtigsten Informationen im System. Entscheidungen über Datenhaltung, Konsistenzanforderungen und Aggregatsgrenzen beeinflussen Schnittstellen, Kommunikation, Persistenzstrategien und sogar Deployment-Optionen.

Fehlende oder schlechte Datenmodelle führen zu Integrationsproblemen, ineffizienten Abfragen, komplexem Code und hoher Fehleranfälligkeit. Daher müssen Architekt:innen Datenflüsse, Datenqualität und Modellierung aktiv berücksichtigen.

Reflexionsfrage:

Welche Daten in deinem System sind besonders kritisch und welche architektonischen Entscheidungen ergeben sich daraus?

Musterlösung / möglicher Ansatz:

Z. B. Zahlungsdaten oder Kundendaten:

- benötigen starke Konsistenz, Auditing, klare Ownership
- beeinflussen Wahl von Persistenz (ACID vs. Eventual Consistency)
- erfordern klare Abgrenzung der Domänen und Berechtigungsmodelle

Kapitel 2 – Anforderungen und Randbedingungen

LZ 02-01 (R1, R3): Stakeholder-Anliegen verstehen

Kurz zusammengefasst:

Stakeholder haben unterschiedliche Ziele, Bedürfnisse und Risiken — Architekt:innen müssen sie erkennen, analysieren und berücksichtigen.

Ausführliche Erklärung:

Stakeholder sind alle Personen oder Gruppen, die durch das System betroffen sind oder es beeinflussen: Nutzer:innen, Product Owner, Fachbereiche, Betrieb, Security, Management, Test, Support usw.

Ihre Anliegen reichen von Usability über Sicherheit bis hin zu Wartbarkeit, Compliance oder Performance.

Architekt:innen müssen diese Anliegen identifizieren, sichtbar machen und klären, welche Anforderungen daraus entstehen. Oft gibt es Zielkonflikte (z. B. „Time-to-Market“ vs. „Qualität“), die moderiert werden müssen. Eine gute Stakeholder-Analyse ist Voraussetzung dafür, dass Architektur Anforderungen auch tatsächlich erfüllt.

Reflexionsfrage:

Welche Stakeholder eines Projekts hatten in der Vergangenheit gegensätzliche Erwartungen — und wie hätte Architektur helfen können, diese auszubalancieren?

Musterlösung / möglicher Ansatz:

Beispiel:

- Fachbereich wollte schnelle Feature-Entwicklung.
- Betrieb forderte stabilen Release-Prozess.

Architektur hätte helfen können durch:

- klare Modulgrenzen (weniger Seiteneffekte bei Änderungen)
 - automatisierte Tests
 - Deployment-Pipeline
 - Architekturentscheidungen, die schnelle Releases ermöglichen (z. B. blaue/grüne Deployments, Anwendungsmodularisierung)
-

LZ 02-02 (R1–R3): Anforderungen und Randbedingungen klären und berücksichtigen können

Kurz zusammengefasst:

Architekt:innen müssen Anforderungen erheben, Randbedingungen identifizieren und diese sichtbar in Architekturentscheidungen übersetzen.

Ausführliche Erklärung:

Anforderungen definieren, *was* ein System leisten soll; Randbedingungen legen fest, *unter welchen Rahmenbedingungen* es entwickelt oder betrieben wird.

Typische Randbedingungen sind:

- gesetzliche Vorgaben (DSGVO, Compliance)
- bestehende Infrastruktur
- Budget, Laufzeit, Organisation
- gegebenes Technologie-Ökosystem

Unklare oder unvollständige Anforderungen führen zu Fehlentscheidungen.

Architekt:innen müssen aktiv Fragen stellen, Annahmen explizit machen, Anforderungen strukturieren (funktional, nicht-funktional) und gemeinsam mit Stakeholdern priorisieren. Randbedingungen (z. B. „Muss on-premises laufen“) können bestimmte Architekturformen erzwingen.

Reflexionsfrage:

Welche Randbedingung hat in deinem letzten Projekt stark beeinflusst, wie die Architektur aussehen musste?

Musterlösung / möglicher Ansatz:

Beispiel: „Daten müssen zwingend in der EU gespeichert werden“ → führte zu:

- Wahl eines EU-Cloud-Providers
- klaren Datenflüssen
- verschlüsselter Speicherung
- Einschränkungen bei globalen Services

LZ 02-03 (R1): Qualitäten eines Softwaresystems verstehen und erklären

Kurz zusammengefasst:

Qualitätsattribute wie Performance, Sicherheit, Wartbarkeit oder Verfügbarkeit bestimmen die Architektur maßgeblich.

Ausführliche Erklärung:

Qualitätsanforderungen („Non-Functional Requirements“) sind meist architekturrelevant. Sie bestimmen, welche Strukturen, Technologien und Patterns geeignet sind. Beispiele:

- Performance → Caching, asynchrone Verarbeitung
- Sicherheit → Vertrauenszonen, AuthN/AuthZ
- Wartbarkeit → modulare Struktur, geringe Kopplung
- Verfügbarkeit → Redundanz, Load-Balancing

Architekt:innen müssen Qualitätsmerkmale kennen, ihre Auswirkungen erklären können und verstehen, wie Architektur zur Erfüllung beiträgt. Oft gibt es Zielkonflikte (z. B. Sicherheit vs. Benutzerfreundlichkeit), die transparent bewertet werden müssen.

Reflexionsfrage:

Welches Qualitätsattribut wurde in einem früheren Projekt vernachlässigt — und welche Probleme traten auf?

Musterlösung / möglicher Ansatz:

Wenn Wartbarkeit vernachlässigt wurde → Ergebnis:

- hoher Aufwand für kleine Änderungen
- instabile Releases
- stark zunehmende technische Schulden
Lösung wäre gewesen:
 - klare Modulgrenzen
 - bessere Dokumentation
 - konsequente Tests

LZ 02-04 (R1–R3): Anforderungen an Qualitäten formulieren

Kurz zusammengefasst:

Qualitätsanforderungen müssen präzise, testbar und messbar gemacht werden, z. B. durch Qualitätsszenarien.

Ausführliche Erklärung:

Allgemeine Aussagen wie „Das System soll schnell sein“ helfen nicht. Architekt:innen müssen Qualitätsanforderungen konkretisieren, messbar machen und darlegen, wie sie überprüft werden können.

Hilfreiche Techniken:

- **Quality Attribute Scenarios** („Bei Last X soll Antwortzeit < Y ms sein“)
- **SMART-Formulierungen**
- **Tradeoff-Analysen**

Dadurch lassen sich Architekturentscheidungen begründet treffen.

Beispiel: Performanceanforderungen beeinflussen Caching-Strategien; Security-Anforderungen beeinflussen die Struktur von Vertrauenszonen.

 **Reflexionsfrage:**

Formuliere eine schlechte Qualitätsanforderung aus einem Projekt und verbessere sie.

Musterlösung / möglicher Ansatz:

Schlecht:

„Das System muss hochperformant sein.“

Gut:

„Bei 500 gleichzeitigen Nutzern müssen 95 % der Requests innerhalb von < 300 ms beantwortet werden.“

LZ 02-05 (R1): Explizite Aussagen vor impliziten Annahmen bevorzugen

 **Kurz zusammengefasst:**

Uunausgesprochene Annahmen sind gefährlich — sie müssen explizit gemacht, dokumentiert und regelmäßig geprüft werden.

 **Ausführliche Erklärung:**

Viele Systemprobleme entstehen, weil Teams unterschiedliche Annahmen haben, die nie explizit besprochen wurden:

„Benutzer laden nur kleine Dateien.“

„Die API wird nur intern genutzt.“

„Internetverbindung ist immer stabil.“

Solche Annahmen können zu massiven Problemen führen, wenn sie falsch sind.

Architekt:innen müssen Annahmen bewusst abfragen, dokumentieren (z. B. in ADRs, Risiken, Randbedingungen) und sichtbar machen.

Nur so lassen sich Risiken erkennen und steuern.

 **Reflexionsfrage:**

Welche implizite Annahme hat in einem Projekt später Probleme verursacht?

Musterlösung / möglicher Ansatz:

Annahme: „Es wird niemals sehr hohe Last geben.“

→ Später führte ein Marketing-Event zu massiven Ausfällen.

Hätte man die Annahme explizit gemacht, wären Lasttests, Skalierungsmechanismen oder Caching früh eingeplant worden.



Kapitel 3 – Entwurf und Entwicklung von Softwarearchitekturen

LZ 03-01 (R1): Anforderungen durch Architektur erreichen

Kurz zusammengefasst:

Architektur dient dazu, funktionale und qualitative Anforderungen systematisch umzusetzen.

Ausführliche Erklärung:

Softwarearchitektur entsteht nicht isoliert — sie ist eine Reaktion auf Anforderungen. Qualitätsanforderungen wie Performance, Sicherheit, Skalierbarkeit oder Verfügbarkeit beeinflussen die Struktur, Schnittstellen und Technologien eines Systems.

Architekt:innen müssen nachvollziehbar zeigen, **welche Architekturentscheidungen welches Ziel erfüllen.**

Dies geschieht z. B. durch:

- Einsatz von Caching für Performance
 - Redundanzen für Verfügbarkeit
 - Schichten & Firewalls für Sicherheit
 - lose Kopplung für Wartbarkeit
- Architektur ist damit der Schlüssel, Anforderungen planbar und zuverlässig zu erfüllen.

Reflexionsfrage:

Welche Architekturentscheidung in einem deiner Projekte hing direkt an einer Qualitätsanforderung?

Musterlösung:

„Wir haben Eventual Consistency gewählt, um Skalierbarkeit zu erreichen. Die Entscheidung entstand aus der Lastanforderung: > 10.000 Requests/s.“

LZ 03-02 (R1): Softwarearchitekturen entwerfen

Kurz zusammengefasst:

Architekt:innen strukturieren Systeme in Bausteine, definieren Schnittstellen und legen technische Leitplanken fest.

Ausführliche Erklärung:

Der Architekturentwurf beantwortet grundlegende Fragen:

- Welche Bausteine gibt es?
- Welche Verantwortlichkeit hat jeder Baustein?
- Wie kommunizieren sie?
- Welche Technologien und Plattformen werden eingesetzt?
Architektur entsteht iterativ — durch Analyse, Entwurf, Feedback und Anpassung.
Wichtige Strukturen sind z. B. Schichten, Komponenten, Module, Subsysteme, Services oder Bounded Contexts.

Reflexionsfrage:

Wie würdest du die ersten Baustein-Kandidaten für ein neues System bestimmen?

Musterlösung:

„Ich würde zuerst Domänenanalyse und Use Cases betrachten und danach funktionale Verantwortlichkeiten in Bausteine schneiden.“

LZ 03-03 (R1/R3): Vorgehen und Heuristiken zur Architekturentwicklung auswählen und anwenden

Kurz zusammengefasst:

Architekt:innen nutzen strukturierte Vorgehensmodelle wie ADD, arc42, Risk-Driven Design oder MDD sowie bewährte Heuristiken.

Ausführliche Erklärung:

Es gibt vier zentrale Vorgehensmodelle im Curriculum:

Attribute-Driven Design (ADD)

Architektur wird aus Qualitätsanforderungen abgeleitet.

Zuerst werden Quality Attribute Scenarios definiert, dann Bausteine, Schnittstellen und Taktiken entworfen, die diese Qualitäten erfüllen.

2 arc42

Ein bewährtes Architektur-Template mit Bausteinsicht, Laufzeitsicht, Verteilung, Qualitätszielen, Randbedingungen, Risiken und Entscheidungen.

3 Risk-Driven Design

Die riskantesten Punkte (z. B. Performance, Integration, neue Technologien) werden zuerst durch PoCs, Spikes oder Tests untersucht.

4 Model-Driven Design (MDD/MDA)

Modelle dienen als zentrale Entwurfs- und ggf. Codequelle; hilft bei Konsistenz und Genauigkeit.

Heuristiken ergänzen das Vorgehen, etwa:

- hohe Kohäsion, geringe Kopplung
- erst Risiken adressieren
- Verantwortlichkeiten klar trennen
- Entscheidungen so spät wie möglich, aber bewusst treffen

💡 Reflexionsfrage:

Welches Vorgehensmodell wäre geeignet für ein Projekt mit hoher technologischer Unsicherheit?

☒ Musterlösung:

„Risk-Driven Design, kombiniert mit Spikes und PoCs, um Risiken früh zu reduzieren. Dokumentiert würde ich mit arc42.“

LZ 03-04 (R1–R3): Entwurfsprinzipien erläutern und anwenden

🔍 Kurz zusammengefasst:

Entwurfsprinzipien legen die Regeln für guten Systembau fest und erhöhen Qualität und Wartbarkeit.

📘 Ausführliche Erklärung:

Wichtige Prinzipien:

- Single Responsibility
- Separation of Concerns
- Information Hiding

- Dependency Inversion
 - Minimierung von Seiteneffekten
- Diese Prinzipien verhindern Kopplungsprobleme und unterstützen Änderbarkeit. Sie wirken sowohl auf Architektur- als auch auf Designlevel.

 **Reflexionsfrage:**

Welches Prinzip hilft, die Auswirkungen von Codeänderungen zu begrenzen?

Musterlösung:

„Information Hiding, weil es interne Details kapselt.“

LZ 03-05 (R1/R2): Zusammenhang zwischen Feedback-Schleifen und Risiko

 **Kurz zusammengefasst:**

Frühes Feedback reduziert komplexe Risiken und verhindert Fehlentscheidungen.

 **Ausführliche Erklärung:**

Architekt:innen nutzen Feedbackmethoden wie:

- Spikes, PoCs
- Lasttests
- Architektureviews
- Prototyping

Diese Schleifen machen Risiken sichtbar, bevor es teuer wird.

Architektur ist kein One-Shot, sondern ein iterativer Lernprozess.

 **Reflexionsfrage:**

Welches Risiko hättest du in einem Projekt gerne früher getestet?

Musterlösung:

„Ob der gewählte Message-Broker die notwendige Latenz einhält.“

LZ 03-06 (R1): Abhängigkeiten von Bausteinen managen

 **Kurz zusammengefasst:**

Architekt:innen minimieren Kopplung und gestalten stabile Abhängigkeitsbeziehungen.

 **Ausführliche Erklärung:**

Grundregeln:

- Zyklen vermeiden
- stabile Abhängigkeitsrichtungen
- Abstraktionen verwenden statt Implementierungen
- Grenzen nach Domäne, nicht nach Technik
→ verbessert Wartbarkeit und Testbarkeit.

 **Reflexionsfrage:**

Warum sind zyklische Abhängigkeiten kritisch?

 **Musterlösung:**

„Weil Änderungen in einem Modul sofort andere Module beeinflussen und Deployments schwer werden.“

LZ 03-07 (R1–R3): Schnittstellen entwerfen und spezifizieren

 **Kurz zusammengefasst:**

Schnittstellen müssen klar, stabil und konsistent sein.

 **Ausführliche Erklärung:**

Zu definieren sind:

- Kommunikationsform (REST, Messaging, RPC)
- Datenformate
- Fehlerbehandlung
- Versionierung
- Semantik

Eine gute Schnittstelle ist einfach nutzbar und bleibt langfristig stabil.

 **Reflexionsfrage:**

Was ist wichtiger: einfache Schnittstelle oder vollständige Funktionalität?

 **Musterlösung:**

„Einfache Schnittstelle, weil Komplexität zu Kopplungsproblemen und vielen Änderungen führt.“

LZ 03-08 (R1/R3): Wichtige Architekturmuster anwenden

 **Kurz zusammengefasst:**

Architekturmuster lösen wiederkehrende Strukturprobleme.

Ausführliche Erklärung:

Typische Muster:

- Schichtenarchitektur
 - Microservices
 - Event-driven Architecture
 - Hexagonal Architecture
- Sie geben grobe Struktur und Kommunikationsformen vor.

Reflexionsfrage:

Wann eignet sich Event-driven Architecture?

Musterlösung:

„Wenn Lose Kopplung, Skalierbarkeit und asynchrone Verarbeitung wichtig sind.“

LZ 03-09 (R3): Wichtige Entwurfsmuster anwenden

Kurz zusammengefasst:

Entwurfsmuster schaffen bewährte Lösungen auf Klassen-/Objektebene.

Ausführliche Erklärung:

Beispiele:

- Strategy → austauschbares Verhalten
 - Observer → Ereignisverteilung
 - Adapter → Integration inkompatibler Schnittstellen
 - Factory → kontrollierte Erstellung
- Sie erhöhen Wiederverwendbarkeit und Konsistenz.

Reflexionsfrage:

Welches Pattern hilft bei austauschbaren Algorithmen?

Musterlösung:

„Strategy.“

LZ 03-10 (R1): Querschnittsthemen identifizieren und Querschnittskonzepte entwerfen

Kurz zusammengefasst:

Querschnittsthemen betreffen das ganze System und brauchen zentrale Lösungen.

Ausführliche Erklärung:

Querschnittsthemen:

- Logging
- Security
- Monitoring
- Konfiguration
- Fehlerbehandlung

Architekt:innen entwerfen übergreifende Mechanismen dafür, um Konsistenz sicherzustellen.

Reflexionsfrage:

Warum muss Logging einheitlich sein?

Musterlösung:

„Weil sonst Analyse, Monitoring und Fehlerbehebung kaum möglich sind.“

LZ 03-11 (R3): Grundlegende Prinzipien von Software-Deployments kennen

Kurz zusammengefasst:

Deployment beeinflusst Architekturentscheidungen über Modularität, Versionierung und Betrieb.

Ausführliche Erklärung:

Themen:

- CI/CD
- Blue/Green, Canary Releases
- Containerisierung
- Immutable Infrastructure
- Versionierung & Artefaktverwaltung

Deployment bestimmt Betriebskosten, Verfügbarkeit und Risiken.

 **Reflexionsfrage:**

Warum ist Deployment architekturelevant?

Musterlösung:

„Weil es bestimmt, wie Bausteine gebaut, ausgeliefert und betrieben werden und welche Struktur dafür notwendig ist.“

LZ 03-12 (R3): Herausforderungen verteilter Systeme kennen

 **Kurz zusammengefasst:**

Verteilte Systeme sind schwierig wegen Latenz, Ausfällen und Konsistenzproblemen.

 **Ausführliche Erklärung:**

Kernprobleme:

- Unzuverlässiges Netzwerk
- Latenz
- Teilausfälle
- Inkonsistente Daten
- CAP-Theorem

Typische Maßnahmen:

- Retry, Timeout, Circuit Breaker
 - Eventual Consistency
 - Idempotenz
- Monitoring/Tracing

 **Reflexionsfrage:**

Beispiel für eine Situation, in der eventual consistency akzeptabel ist?

Musterlösung:

„Produktkatalog – kurz inkonsistent, aber unkritisch.“

Kapitel 4 – Beschreibung und Kommunikation von Softwarearchitekturen

LZ 04-01 (R1): Anforderungen an technische Dokumentation erläutern und berücksichtigen

Kurz zusammengefasst:

Technische Dokumentation muss vollständig, korrekt, adressatengerecht und aktuell sein.

Ausführliche Erklärung:

Architekturdokumentation dient nicht nur der Ablage, sondern der **Kommunikation**. Sie hilft Teams dabei:

- Architektur zu verstehen
- Entscheidungen nachzuvollziehen
- Risiken zu erkennen
- Systeme zu betreiben
- neue Teammitglieder einzuarbeiten

Anforderungen an gute Dokumentation:

- **Zielgruppengerecht:** Entwickler brauchen andere Infos als Management.
- **Aktuell:** dokumentiert, was *ist*, nicht was mal geplant war.
- **Konzise:** klar, verständlich, ohne unnötige Details.
- **Strukturiert:** z. B. mit Templates wie arc42.
- **Begründet:** Dokumentation erklärt das *Warum*, nicht nur das *Was*.
- **Nachvollziehbar:** ADRs unterstützen Rückverfolgbarkeit.

Reflexionsfrage:

Welche Informationen fehlen typischerweise in Architekturdokumenten – und warum ist das problematisch?

Musterlösung:

„Entscheidungsbegründungen fehlen oft. Ohne ‚Warum‘ versteht niemand, wann eine Entscheidung geändert werden darf.“

LZ 04-02 (R1–R3): Softwarearchitekturen beschreiben und kommunizieren

Kurz zusammengefasst:

Architektur muss verständlich visualisiert, erklärt und vermittelt werden – schriftlich und mündlich.

Ausführliche Erklärung:

Softwarearchitektur wird durch **Sichten, Diagramme, Modelle und Erklärungen** beschrieben:

- Bausteinsicht (Struktur)
- Laufzeitsicht (Abläufe)
- Verteilungssicht (Deployment)
- Kontextsicht (Systemgrenzen)

Architekt:innen müssen Informationen so aufbereiten, dass Teams effektiv arbeiten können:

- **klare Diagramme** (z. B. C4-Modelle)
- **präzise Beschreibungen**
- **Workshops und Reviews**
- **Kommunikation mit Stakeholdern**

Gute Architekturkommunikation reduziert Missverständnisse, Risiken und technische Schulden.

Reflexionsfrage:

Warum ist visuelle Kommunikation (Diagramme) oft effektiver als Text?

Musterlösung:

„Diagramme vermitteln Struktur und Zusammenhänge schneller als Text und schaffen ein gemeinsames mentales Modell.“

LZ 04-03 (R2–R3): Notations-/Modellierungsmittel erläutern und anwenden

Kurz zusammengefasst:

Modelle und Notationen wie UML, BPMN oder C4 helfen, Architektur präzise und konsistent darzustellen.

Ausführliche Erklärung:

Architekt:innen nutzen Modellierungsmittel wie:

- **C4-Modell** (sehr leichtgewichtig für Architekturen)
- **UML** (z. B. Klassendiagramme, Sequenzdiagramme)
- **BPMN** (Prozessmodellierung)
- **Deploymentdiagramme**

Entscheidend ist nicht die Notation selbst, sondern **Konsistenz –, Verständlichkeit – und Zweckmäßigkeit**.

Diagramme müssen eindeutig sein, Fehlerfälle zeigen und relevante Details darstellen.

 **Reflexionsfrage:**

Welche Diagrammart eignet sich am besten, um ein komplexes Ablaufverhalten eines Systems darzustellen?

 **Musterlösung:**

„Sequenzdiagramme oder C4-L3-Komponenteninteraktionen, da sie zeitliche Abläufe und Interaktionen klar zeigen.“

LZ 04-04 (R3): Unerwartete Situationen geschickt bewältigen

(Dieses Lernziel steht im Curriculum, auch wenn es im Self-Check früher als „nicht gefunden“ markiert war.)

 **Kurz zusammengefasst:**

Architekt:innen müssen flexibel auf Überraschungen reagieren – fachlich, technisch und organisatorisch.

 **Ausführliche Erklärung:**

Unerwartete Situationen treten ständig auf:

- Dienste funktionieren anders als dokumentiert
- nichtfunktionale Anforderungen ändern sich
- Risiken treten ein
- Stakeholder liefern widersprüchliche Informationen
- Architekturentscheidungen müssen kurzfristig angepasst werden

Architekt:innen brauchen:

- ruhiges, lösungsorientiertes Vorgehen
- Kommunikationsfähigkeit
- Entscheidungsfähigkeit unter Unsicherheit
- Erfahrung im Umgang mit Risiken und Kompromissen

 **Reflexionsfrage:**

Welche Art unerwarteter Situation bringt Architekt:innen typischerweise in Zeitdruck?

Musterlösung:

„Wenn ein kritischer Dienst nicht die erwartete Last trägt – dann müssen kurzfristig Alternativen oder Workarounds gefunden werden.“

LZ 04-05 (R1): Architektsichten erläutern und anwenden

 **Kurz zusammengefasst:**

Sichten trennen unterschiedliche Perspektiven auf ein System und machen Architektur verständlich.

 **Ausführliche Erklärung:**

Architektsichten beantworten verschiedene Fragen:

- **Bausteinsicht:** Wie ist das System strukturiert?
- **Laufzeitsicht:** Wie laufen Prozesse ab?
- **Kontextsicht:** Was liegt außerhalb des Systems?
- **Verteilungssicht:** Wo läuft was? Infrastruktur? Cloud?

Sichten reduzieren Komplexität und unterstützen unterschiedliche Stakeholder.

 **Reflexionsfrage:**

Warum ist die Laufzeitsicht besonders wichtig für Entwickler und Tester?

Musterlösung:

„Weil sie Abläufe, Fehlerfälle und Interaktionen zeigt – Grundlage für Tests und Implementierung.“

LZ 04-06 (R1): Schnittstellen dokumentieren

 **Kurz zusammengefasst:**

Schnittstellen benötigen klare, vollständige und eindeutige Spezifikationen.

 **Ausführliche Erklärung:**

Eine Schnittstelle braucht:

- Endpunkte / Methoden
- Datenformate
- Fehlerfälle
- Rückgabewerte

- Semantik
- Versionierung
- Verträge („API Contracts“)

Gut dokumentierte Schnittstellen reduzieren Integrationskosten, Fehler und Missverständnisse.

 **Reflexionsfrage:**

Welcher Teil fehlt häufig in API-Dokumentationen – und ist aber extrem wichtig?

 **Musterlösung:**

„Die Beschreibung der Fehlerfälle und Fehlermeldungen.“

LZ 04-07 (R2): Querschnittsthemen dokumentieren und kommunizieren

 **Kurz zusammengefasst:**

Querschnittsthemen wie Logging, Security oder Monitoring müssen systemweit dokumentiert werden.

 **Ausführliche Erklärung:**

Querschnittsthemen betreffen das ganze System.
Sie sollten einheitlich dokumentiert sein:

- Logging-Strategie
- Security-Konzept
- Monitoring/Tracing
- Konfigurationskonzepte
- Fehlerbehandlung

Ohne diese Dokumentation entstehen Inkonsistenzen und technische Schulden.

 **Reflexionsfrage:**

Warum ist Security als Querschnittsthema so kritisch?

 **Musterlösung:**

„Weil inkonsistente Sicherheitsmaßnahmen zu Sicherheitslücken führen können.“

LZ 04-08 (R1–R2): Architekturentscheidungen erläutern und dokumentieren

Kurz zusammengefasst:

Architekturentscheidungen müssen nachvollziehbar festgehalten werden.

Ausführliche Erklärung:

Architekt:innen dokumentieren Entscheidungen in ADRs (Architecture Decision Records):

- Problem / Kontext
- Alternativen
- Bewertung
- Entscheidung
- Konsequenzen

Dies schafft Klarheit für das gesamte Team und erleichtert spätere Änderungen.

Reflexionsfrage:

Warum ist es wichtig, auch verworfene Alternativen zu dokumentieren?

Musterlösung:

„Weil man sonst nicht nachvollziehen kann, warum bestimmte Lösungen nicht gewählt wurden – was spätere Diskussionen erschwert.“

LZ 04-09 (R3): Weitere Hilfsmittel und Werkzeuge zur Dokumentation kennen

Kurz zusammengefasst:

Architekt:innen müssen Werkzeuge kennen, die Dokumentation effizient unterstützen.

Ausführliche Erklärung:

Wichtige Tools:

- Markdown + Git (für versionierte Dokumentation)
- Diagrammtools (PlantUML, Structurizr, Mermaid)
- Wikis / Confluence
- ADR-Tools
- Architektur-Repositorys
- API-Dokumentationsgeneratoren (OpenAPI/Swagger)

Ziel ist, Dokumentation leicht pflegbar, automatisierbar und teamorientiert zu gestalten.

 **Reflexionsfrage:**

Warum ist „Documentation-as-Code“ oft besser als klassische Dokumente?

 **Musterlösung:**

„Weil Dokumentation versioniert, überprüfbar, gemeinsam wartbar und automatisch generierbar wird.“

Kapitel 5 – Analyse und Bewertung von Softwarearchitekturen

LZ 05-01 (R1): Gründe für Architekturanalyse kennen

 **Kurz zusammengefasst:**

Architekturanalyse stellt sicher, dass die Architektur Anforderungen erfüllt, Risiken reduziert und langfristig tragfähig bleibt.

 **Ausführliche Erklärung:**

Architekturanalyse bedeutet, Architektur systematisch zu überprüfen — bevor Probleme teuer werden.

Es gibt mehrere zentrale Gründe:

1 Erfüllt die Architektur die Anforderungen?

- funktionale Anforderungen
- Qualitätsattribute (Performance, Sicherheit, Skalierbarkeit ...)
- Randbedingungen (Organisation, Technik, Compliance)

2 Risiken früh erkennen

Viele Architekturfehler zeigen sich erst spät (z. B. Performanceprobleme, Integrationsschwierigkeiten). Analyse deckt sie früh auf.

3 Transparenz schaffen

Stakeholder müssen verstehen:

- warum die Architektur so aufgebaut ist
- welche Kompromisse eingegangen wurden
- welche Alternativen geprüft wurden

4 Qualität langfristig sichern

Architekturen altern – Analyse erkennt früh Anzeichen von Erosion, technischen Schulden oder fehlender Konsistenz.

5 Grundlage für Weiterentwicklung

Nur eine analysierte Architektur kann sinnvoll weiterentwickelt, migriert oder modernisiert werden.

💡 Reflexionsfrage:

Welche negativen Folgen hastest du schon einmal, weil eine Architektur *nicht* früh genug analysiert wurde?

☒ Musterlösung:

„Wir stellten erst spät fest, dass die Datenbank die erwartete Last nicht trägt. Eine frühere Architekturanalyse hätte das erkannt und eine andere Datenhaltung nahegelegt.“

LZ 05-02 (R1/R3): Qualitäten eines Softwaresystems analysieren

⌚ Kurz zusammengefasst:

Qualitätsmerkmale wie Performance, Sicherheit, Robustheit oder Skalierbarkeit müssen methodisch analysiert werden.

📘 Ausführliche Erklärung:

Die Analyse von Qualitätsmerkmalen ist entscheidend, um sicherzustellen, dass ein System den Anforderungen gerecht wird.

Wichtige Methoden:

1 Qualitätsszenarien (Quality Attribute Scenarios)

Konkrete messbare Anforderungen formulieren:

„Bei 500 gleichzeitigen Nutzern beträgt die Antwortzeit < 200 ms.“

Ohne Szenarien kann man Qualität nicht analysieren.

2 Qualitätsmodelle

z. B. ISO 25010: Sicherheit, Wartbarkeit, Zuverlässigkeit, Effizienz usw.

3 Tradeoff-Analysen

Qualitätsmerkmale beeinflussen sich:

- Performance vs. Sicherheit
- Konsistenz vs. Verfügbarkeit
- Flexibilität vs. Einfachheit

Architekt:innen müssen diese Konflikte identifizieren und erklären.

4 Experimentelle Analysen

- Lasttests (Performance)
- Chaos Engineering (Resilienz)
- Security Tests (Penetration Testing)
- Monitoring-Daten analysieren

5 Architekturbewertungen

z. B. ATAM (Architecture Tradeoff Analysis Method)

💡 Reflexionsfrage:

Welche Qualitätsanforderung sollte immer konkretisiert und messbar gemacht werden – und warum?

☒ Musterlösung:

„Performance – weil allgemeine Aussagen wie ‚das System soll schnell sein‘ nicht analysierbar sind. Messbare Szenarien ermöglichen Tests und Entscheidungen.“

LZ 05-03 (R2): Konformität mit Architekturentscheidungen bewerten

🔍 Kurz zusammengefasst:

Architekt:innen prüfen regelmäßig, ob Implementierung und tatsächliches System der Architektur entsprechen.

📘 Ausführliche Erklärung:

Architektur entsteht in Dokumenten, aber Systeme entstehen im Code.
Konformität bedeutet:

- **Hält sich das Team an Architekturentscheidungen?**
- **Sind Schnittstellen so implementiert wie geplant?**
- **Stimmen Sicherheits-, Logging-, oder Schichtenregeln?**

- Entsprechen Deployment-Topologien der Architektur?

Wichtige Werkzeuge:

 **Architektur-Reviews**

Vergleich zwischen Dokumentation und System.

 **Code-Analysen / Architektur-Linter**

Tools wie ArchUnit, SonarQube, jQAssistant prüfen Strukturregeln automatisiert.

 **ADRs (Architecture Decision Records)**

Wurden Entscheidungen umgesetzt und sind sie noch gültig?

 **Dokumentationsabgleich**

Entspricht die Doku dem Ist-Zustand?

Wenn Abweichungen auftreten, muss geklärt werden:

- Ist die Architektur veraltet?
- Oder wurde falsch implementiert?
- Muss eine Entscheidung revidiert werden?

 **Reflexionsfrage:**

Warum entstehen Abweichungen zwischen Architektur und Implementierung oft unbemerkt?

 **Musterlösung:**

„Weil Teams unter Zeitdruck pragmatische Lösungen bauen und Architekturen nicht automatisiert überprüft werden. Ohne regelmäßige Reviews driftet Code und Architektur auseinander.“

Kapitel 6 – Beispiele für Softwarearchitekturen

LZ 06-01 (R3): Bezug von Anforderungen und Randbedingungen zur Lösung erfassen

Kurz zusammengefasst:

Architekt:innen müssen verstehen, wie Anforderungen und Randbedingungen Architekturentscheidungen prägen – und diese Zusammenhänge explizit darstellen.

Ausführliche Erklärung:

In realen Systemen entsteht Architektur nie im luftleeren Raum.

Sie ist immer eine **Antwort auf Anforderungen und Randbedingungen**:

1 Anforderungen (functional & non-functional)

- fachliche Aufgaben (Use Cases, Domänenlogik)
- Qualitätsanforderungen (Performance, Sicherheit, Skalierbarkeit)
- Integrationsanforderungen
- UI/UX-Vorgaben

Diese Anforderungen bestimmen:

- Bausteine
- Schnittstellen
- Technologien
- Kommunikationsmuster

2 Randbedingungen

Randbedingungen sind *nicht verhandelbar*:

- Vorgaben der Organisation (z. B. Cloud-Strategie)
- Compliance & gesetzliche Anforderungen
- existierende Systeme / Legacy
- Budget und Zeit
- verfügbare Skills im Team
- technische Standards
- Betriebsvorgaben (z. B. Kubernetes)

Architekt:innen müssen zeigen: **Welche Entscheidung wurde durch welche Anforderung / Randbedingung beeinflusst?**

Typische Fragen:

- „Warum genau wurde dieser Schnitt gewählt?“
- „Weshalb diese Technologie?“
- „Warum monolithisch oder verteilt?“
- „Warum synchrone vs. asynchrone Kommunikation?“

Nur wenn der Bezug klar ist, kann Architektur verstanden, weiterentwickelt und bewertet werden.

Reflexionsfrage:

Welche Randbedingung in deinem Projekt hatte den stärksten Einfluss auf eine architekturelle Entscheidung?

Musterlösung:

„Die Unternehmensvorgabe, alle neuen Systeme in der Cloud zu betreiben, führte dazu, dass wir Containerisierung und Kubernetes einsetzen mussten.“

LZ 06-02 (R3): Technische Umsetzung einer Lösung nachvollziehen

Kurz zusammengefasst:

Architekt:innen müssen verstehen, wie Architekturentscheidungen technisch umgesetzt werden – und prüfen können, ob die Implementierung zur Architektur passt.

Ausführliche Erklärung:

Die technische Umsetzung („Realization“) zeigt, wie Architektur im Code, in Deployment-Strukturen und im Betrieb tatsächlich realisiert wurde.

Dabei müssen Architekt:innen folgende Fragen beantworten:

1 Wie werden Architekturbausteine implementiert?

- Welche Klassen / Module bilden welchen Baustein ab?
- Entsprächen Schnittstellen der Spezifikation?
- Ist die Bausteinverantwortlichkeit korrekt umgesetzt?

2 Wie sieht die tatsächliche Kommunikation aus?

- Welche Protokolle (REST, Messaging)?
- Werden Timeouts, Retries, Patterns korrekt genutzt?
- Passen Fehlerfälle zur Architektur?

3 Wie wird Deployment umgesetzt?

- Wie sieht die reale Infrastruktur aus?
- Welche Container, Pods, Server, Dienste?

- Gibt es Abweichungen zur geplanten Verteilungssicht?

4 Passt die Security- und Logging-Umsetzung zur Architektur?

- Wurde das Security-Konzept wirklich implementiert?
- Sind Logs strukturiert?
- Entspricht das Monitoring den Vorgaben?

5 Stimmen Code und Architektur wirklich überein?

Viele Systeme driften über Zeit auseinander („Architecture Erosion“). Architekt:innen müssen Tools und Reviews einsetzen, um die Umsetzung regelmäßig zu prüfen (z. B. ArchUnit, jQAssistant, SonarQube).

💡 Reflexionsfrage:

Warum ist es wichtig, dass Architekt:innen auch die technische Umsetzung verstehen – und nicht nur Diagramme erstellen?

✓ Musterlösung:

„Weil nur so sichergestellt werden kann, dass Architekturentscheidungen wirklich umgesetzt wurden. Andernfalls entstehen Architektur-Erosion, Drift und schwer wartbarer Code.“

LZ 06-03– Analyse realer Architekturen

■ Ausführliche Erklärung:

Die Analyse realer Architekturen ist ein zentrales Element der Ausbildung von Softwarearchitekt:innen. Reale Systeme zeigen häufig, wie theoretische Prinzipien in der Praxis angewendet oder bewusst gebrochen wurden. Architekt:innen lernen, Strukturen, Muster und Abhängigkeiten in komplexen Umgebungen zu verstehen. Dies umfasst das Erkennen von Entwurfsprinzipien, aber auch die Identifikation von Anti-Patterns. Oft wird erst in realen Architekturen sichtbar, wie Qualitätsziele wie Performance, Sicherheit oder Skalierbarkeit tatsächlich umgesetzt wurden. Auch Trade-offs werden deutlich: Ein System kann beispielsweise bewusst weniger flexibel sein, um die Wartbarkeit zu verbessern. Diese Analysen helfen, zukünftige Entscheidungen sicherer zu treffen und Risiken realistisch einzuschätzen. Besonders wertvoll ist, dass reale Beispiele die Brücke zwischen Theorie und Praxis schlagen und damit nachhaltiges Lernen ermöglichen.

❖ Fallbeispiel:

Ein E-Commerce-System musste über Jahre hinweg wachsen und erhielt zahlreiche neue Features. Ursprünglich als Monolith geplant, war die Architektur zunehmend schwer wartbar. Die Analyse zeigte: das Hauptproblem lag nicht in der Struktur selbst, sondern in fehlender Modularisierung. Durch Refactoring in Domänenbereiche und wohldefinierte Module wurde der Monolith stabilisiert, ohne vollständig zu Microservices umzubauen.

Dieses Beispiel zeigt, dass nicht jedes skalierende System Microservices benötigt – klare Modularisierung kann ausreichend sein.

💡 Reflexionsfrage:

Welche Gründe könnten dafür sprechen, einen gewachsenen Monolithen nicht direkt zu Microservices umzubauen?

Lösung / möglicher Ansatz:

Eine mögliche Lösung: Ein direkter Umstieg auf Microservices verursacht oft sehr hohe Komplexität (Monitoring, verteilte Datenhaltung, DevOps-Aufwand). Wenn der Monolith durch konsequente Modularisierung stabil und wartbar gemacht werden kann, ist dies häufig kostengünstiger und risikoärmer. Microservices lohnen sich erst, wenn klare Domänen, autonome Teams und hohe Skalierungsanforderungen vorhanden sind.

LZ 06-04 – Ableitung von Best Practices aus Fallstudien

■ Ausführliche Erklärung:

Fallstudien sind wertvolle Lerninstrumente, da sie reale Herausforderungen und deren Lösungen detailliert darstellen. Sie zeigen, wie Architekt:innen Entscheidungen treffen, Risiken einschätzen und Strukturen gestalten. Best Practices entstehen durch wiederholte erfolgreiche Anwendung bestimmter Muster oder Vorgehensweisen. Beispielsweise zeigt die Analyse vieler Systeme, dass frühe Automatisierung (CI/CD) langfristig Kosten und Fehler reduziert. Andere Fallstudien betonen die Bedeutung von Domain-Driven Design bei komplexen Geschäftslogiken. Wichtig ist, dass Best Practices nicht blind übernommen, sondern an den Kontext angepasst werden. Fallstudien vermitteln außerdem, welche Fehler häufig gemacht werden und wie man sie vermeiden kann. So entwickeln Architekt:innen ein breites Repertoire an Lösungsstrategien, das in unterschiedlichsten Projekten angewendet werden kann.

❖ Fallbeispiel:

Ein Finanzdienstleister führte ein neues System zur Kreditbewertung ein. Anfangs gab es große Probleme mit Performance und Datenqualität. Eine Analyse früherer Projekte zeigte, dass fehlende Observability (Monitoring, Logging, Tracing) ein wiederkehrendes Problem war. Daraufhin wurde ein systemweites Observability-Konzept eingeführt: strukturierte Logs, zentrale Dashboards, verteiltes Tracing und einheitliche Health-Checks. Nach wenigen Monaten war die Fehleranalyse deutlich schneller, und kritische Prozesse wurden stabiler. Dies wurde später als Best Practice in mehreren anderen Projekten wiederverwendet.

💡 Reflexionsfrage:

Welche Elemente gehören zu einem soliden Observability-Konzept in modernen Architekturen?

Lösung / möglicher Ansatz:

Ein möglicher Ansatz: strukturierte Logs, Metriken für zentrale KPIs, verteiltes Tracing über Servicegrenzen hinweg, Dashboards für Betrieb und Entwicklung, Alarmregeln basierend auf aussagekräftigen Schwellenwerten und automatisierte Health-Checks. Diese Elemente helfen, Probleme schneller zu finden und Systemstabilität zu erhöhen.