

Microservices-Migration-Assessment

Kostenlose Checkliste zur realistischen Aufwands-Einschätzung

Von Elyndra Valen, Senior Entwicklerin & Code-Archäologin
Java Fleet Systems Consulting, Essen-Rüttenscheid



Ein Wort von eurer Code-Archäologin

Hi, Entwickler-Gilde!

Als jemand, die schon **drei große Legacy-Modernisierungen** begleitet hat – von 10-Jahre-alten Jakarta-EE-Monolithen zu modernen Spring Boot Microservices – weiß ich: **Microservices-Migration ist wie Archäologie, nur in umgekehrter Richtung.**

Statt antike Ruinen freizulegen, **zerlegen wir gewachsene Code-Kathedralen** in handhabbare, moderne Service-Tempel. Und genau wie in der echten Archäologie kann man dabei **unglaublich viel kaputt machen**, wenn man nicht systematisch vorgeht!



Warum ich dieses Assessment entwickelt habe

Nach meiner letzten Migration (18 Monate, 3x länger als geplant, fast das halbe Team verbrannt) dachte ich mir: **"Es muss einen besseren Weg geben, das Chaos vorherzusehen."**

Dieses Assessment basiert auf den **Schmerzen und Learnings** aus 50+ Enterprise-Migrationen – inklusive meiner eigenen Fehler. Es soll euch helfen, **realistische Erwartungen** zu setzen und **teure Überraschungen** zu vermeiden.



Besonders für Junior Devs (Nova, schau hin! 🙄)

Falls du denkst, Microservices sind "nur ein bisschen Refactoring" – **think again!** Migration bedeutet:

- **Distributed Systems Complexity** (Netzwerk kann ausfallen!)
- **Data Consistency Challenges** (ACID wird zu BASE)
- **Operational Overhead** (ein Service = einfach, 20 Services = exponentiell komplexer)
- **Team-Coordination** (Conway's Law schlägt gnadenlos zu)

Das ist nicht nur "Code umziehen" – das ist **Architektur, Kultur und Operations** komplett neu denken.



Was dieses Assessment kann (und was nicht)



Das Assessment hilft bei:

- Realistische Zeitschätzung (keine "2 Wochen, dann sind wir fertig"-Fantasien)
- Risk-Identifikation (welche Fallen lauern?)
- Team-Readiness (sind wir überhaupt bereit dafür?)

- Go/No-Go-Entscheidung (manchmal ist "nicht migrieren" die richtige Antwort)

✗ Das Assessment ersetzt NICHT:

- Detaillierte Architektur-Planung
- Technology-Evaluation
- Proof-of-Concept-Entwicklung
- Change-Management-Strategie



Wie ihr das Assessment nutzt

Zeitaufwand: 45-60 Minuten (nicht hetzen!)

Empfohlene Teilnehmer: Tech Lead, Senior Developer, DevOps Engineer, Product Owner

Step-by-Step:

1. **Ehrlich bewerten** (Selbstbetrug hilft niemandem)
2. **Punkte zusammenzählen** (Mathematik lügt nicht)
3. **Risiken diskutieren** (Red Flags ernst nehmen)
4. **Entscheidung treffen** (manchmal ist "Nein" die richtige Antwort)

Denkt daran: Microservices sind kein Selbstzweck. **Ein gut strukturierter Monolith ist besser als schlecht gemachte Microservices.**



Legacy-Code-Weisheit: Was ich gelernt habe

Nach Jahren der Code-Archäologie und drei größeren Migrationen kann ich euch eines sagen:

Jeder Monolith erzählt eine Geschichte.

Die wichtigsten Learnings:

1. Conway's Law ist Realität

Eure Service-Architektur wird eure Team-Struktur widerspiegeln – ob ihr wollt oder nicht. Plant entsprechend!

2. "Strangler Fig" over "Big Bang"

Große Umstellungen auf einen Schlag enden meist im Chaos. Schrittweise Migration ist langsamer, aber sicherer.

3. Data ist der wahre Boss

Die schwersten Probleme sind nie "nur Code" – sie sind **Daten-Migration, Transaktionen und Consistency**. Plant dafür die meiste Zeit ein.

4. Team-Burnout ist real

Microservices-Migration ist mental anstrengend. **Plant Pausen ein und rotiert Leute raus** – sonst brennt euch das Team weg.

5. "Working software over comprehensive documentation"

Aber bei Migrationen braucht ihr **mehr Dokumentation als sonst**. Distributed Systems sind komplex – schreibt auf, wie sie funktionieren.

Jetzt aber genug der Vorrede – lasst uns euren Monolithen unter die Lupe nehmen! 🔍



Teil 1: Domain-Analyse-Framework

1.1 Bounded Contexts identifizieren

Bewertet euer System auf einer Skala von 1-5 (1 = schlecht, 5 = excellent):

☐ **Fachliche Abgrenzung erkennbar** (___)

Gibt es klar abgrenzbare Geschäftsbereiche? (User Management, Payment, Inventory, etc.)

☐ **Datenmodell-Trennung möglich** (___)

Können Datenbankschemas sauber getrennt werden ohne viele Cross-References?

☐ **Team-Zuständigkeiten klar** (___)

Arbeiten verschiedene Teams an verschiedenen fachlichen Bereichen?

☐ **API-Grenzen definierbar** (___)

Welche Schnittstellen würden zwischen Services entstehen?

Bewertung Domain-Analyse: ___/20 Punkte

1.2 Monolith-Komplexität



Codebase-Metriken:

☐ **Lines of Code (LOC)** (___)

1 = >500k LOC, 2 = 200-500k, 3 = 100-200k, 4 = 50-100k, 5 = <50k LOC

☐ **Module/Package-Anzahl** (___)

1 = >100 Module, 2 = 50-100, 3 = 20-50, 4 = 10-20, 5 = <10 Module

☐ **Cyclomatic Complexity** (___)

1 = Sehr hoch (SonarQube rot), 2 = Hoch, 3 = Mittel, 4 = Niedrig, 5 = Sehr niedrig



Deployment & Operations:

☐ **Deployment-Frequenz** (___)

1 = Monatlich, 2 = 2-wöchentlich, 3 = Wöchentlich, 4 = Täglich, 5 = Mehrmals täglich

☐ **Build-Zeit** (___)

1 = >30 Min, 2 = 15-30 Min, 3 = 10-15 Min, 4 = 5-10 Min, 5 = <5 Min

☐ **Startup-Zeit** (___)

1 = >5 Min, 2 = 3-5 Min, 3 = 1-3 Min, 4 = 30s-1Min, 5 = <30s

Team & Development:

☐ Team-Größe (aktive Entwickler) (___)

1 = >20 Devs, 2 = 15-20, 3 = 10-15, 4 = 5-10, 5 = <5 Devs

☐ Feature-Entwicklungszeit (durchschnittlich) (___)

1 = >3 Monate, 2 = 2-3 Monate, 3 = 1-2 Monate, 4 = 2-4 Wochen, 5 = <2 Wochen

☐ Merge-Conflicts pro Woche (___)

1 = >10 Conflicts, 2 = 5-10, 3 = 2-5, 4 = 1-2, 5 = <1 Conflict

Bewertung Monolith-Komplexität: ___/45 Punkte



Teil 2: Technical Debt Assessment

2.1 Code-Qualität

☐ Test-Coverage (___)

1 = <30%, 2 = 30-50%, 3 = 50-70%, 4 = 70-85%, 5 = >85%

☐ Dependency-Management (___)

1 = Viele veraltete Dependencies, 2 = Einige veraltete, 3 = Größtenteils aktuell, 4 = Aktuell, 5 = Cutting-edge

☐ Code-Struktur (___)

1 = Spaghetti-Code, 2 = Schwer verständlich, 3 = OK struktur, 4 = Gut strukturiert, 5 = Clean Architecture

☐ Dokumentation (___)

1 = Nicht vorhanden, 2 = Veraltet, 3 = Teilweise, 4 = Gut, 5 = Excellent

Bewertung Technical Debt: ___/20 Punkte

2.2 Refactoring-Readiness

☐ Backward Compatibility (___)

Können APIs versioniert werden ohne Breaking Changes?

☐ Database-Migration-Fähigkeit (___)

Ist das Schema flexibel genug für schrittweise Trennung?

☐ Configuration-Management (___)

Externalized Config, Environment-specific Settings?

☐ Logging & Monitoring (___)

Structured Logging, APM-Tools, Health Checks vorhanden?

Bewertung Refactoring-Readiness: ___/20 Punkte

Teil 3: Team-Readiness-Check

3.1 Technische Skills

☐ **Container-Experience** (___)

1 = Keine, 2 = Grundlagen, 3 = Docker in Dev, 4 = Prod-Experience, 5 = K8s-Expert

☐ **DevOps-Kultur** (___)

1 = Silos, 2 = Erste Schritte, 3 = CI/CD basic, 4 = DevOps etabliert, 5 = Platform Engineering

☐ **Distributed Systems Knowledge** (___)

1 = Keine, 2 = Theorie, 3 = Erste Erfahrung, 4 = Solide Praxis, 5 = Expert-Level

☐ **Monitoring/Observability** (___)

1 = Logs only, 2 = Basic Metrics, 3 = APM-Tools, 4 = Tracing, 5 = Full Observability

Bewertung Team-Skills: ___/20 Punkte

3.2 Organisatorische Readiness

☐ **Conway's Law Alignment** (___)

Spiegelt die gewünschte Service-Architektur eure Team-Struktur wider?

☐ **Ownership-Modell** (___)

Können Teams End-to-End-Verantwortung für Services übernehmen?

☐ **Communication Patterns** (___)

Async Communication, Event-driven Thinking etabliert?

☐ **Failure-Tolerance** (___)

Kultur des "Fast Fail", Blameless Post-Mortems?

Bewertung Orga-Readiness: ___/20 Punkte

Teil 4: Infrastruktur-Requirements

4.1 Platform-Readiness

☐ **Container-Orchestration** (___)

1 = Nicht vorhanden, 2 = Docker-Compose, 3 = Single-Node K8s, 4 = Managed K8s, 5 = Multi-Cluster

☐ **Service Discovery** (___)

1 = Hardcoded IPs, 2 = Load Balancer, 3 = DNS-based, 4 = Service Mesh basic, 5 = Full Service Mesh

☐ **Configuration Management** (___)

1 = Files, 2 = Environment Vars, 3 = Config Server, 4 = GitOps, 5 = Policy-as-Code

☐ **Secrets Management** (___)

1 = Plaintext, 2 = Environment, 3 = Encrypted Files, 4 = Vault/K8s Secrets, 5 = Zero-Trust

Bewertung Platform: ___/20 Punkte

4.2 Data-Strategy

☐ **Database-per-Service möglich** (___)

Können fachliche Bereiche eigene Datenbanken bekommen?

☐ **Event-Streaming-Plattform** (___)

Kafka, Pulsar oder ähnlich für Async Communication?

☐ **Data-Consistency-Strategy** (___)

Eventual Consistency, Saga Pattern, CQRS verstanden?

☐ **Backup/Disaster Recovery** (___)

Für verteilte Systeme ausgelegt?

Bewertung Data-Strategy: ___/20 Punkte



Teil 5: Aufwands-Schätzer

5.1 Migration-Complexity-Score

Gesamtpunktzahl: ___/140 Punkte

Bewertung:

- **120-140 Punkte:** **Go for it!** (6-12 Monate)
- **100-119 Punkte:** **Vorbereitung nötig** (12-18 Monate)
- **80-99 Punkte:** **Hohes Risiko** (18-24 Monate)
- **<80 Punkte:** **Nicht empfohlen** (>24 Monate oder Fail)

5.2 Realistische Zeitschätzung

Basis-Formel: Monate = (140 - Score) / 8 + Baseline

☐ **Baseline je nach Monolith-Größe:**

- Klein (<50k LOC): 6 Monate
- Mittel (50-200k LOC): 12 Monate
- Groß (>200k LOC): 18 Monate

☐ **Multiplikatoren:**

- Team < 5 Devs: x1.5
- Keine Container-Experience: x1.3
- Legacy-Database: x1.4
- Regulatory Requirements: x1.2

Geschätzte Migrationsdauer: ___ Monate

5.3 Effort-Breakdown

☐ **Infrastructure Setup** (20-30% der Zeit)

☐ **Service Extraction** (40-50% der Zeit)

- ☐ **Data Migration** (15-25% der Zeit)
 - ☐ **Testing & Stabilization** (15-20% der Zeit)
-

Teil 6: Risk-Assessment-Matrix

6.1 High-Risk-Indicators

Markiert alle zutreffenden Risiken:

- ☐ **Performance-kritische Anwendung** (sub-second SLAs)
- ☐ **Tight gekoppelte Database-Schema**
- ☐ **Regulatory/Compliance-heavy** (Banking, Healthcare)
- ☐ **Team < 3 Senior Developers**
- ☐ **Keine Rollback-Strategie**
- ☐ **Customer-facing ohne Downtime-Toleranz**
- ☐ **Complex Business-Transaktionen über Module hinweg**
- ☐ **Legacy-Dependencies ohne Support**

Risk-Score: ____/8

6.2 Mitigation-Strategies

Bei Risk-Score > 4:

- **Strangler Fig Pattern** statt Big Bang Migration
- **Feature Toggles** für schrittweise Umstellung
- **Parallel Run** für kritische Services
- **Extended Testing-Phase** (Canary, Blue-Green)

6.3 Go/No-Go Decision Framework

☐ **GO wenn:**

- Score > 100 UND Risk-Score < 4
- Klare Business-Value-Begründung vorhanden
- Team committed für >12 Monate
- Budget für 1.5x der geschätzten Zeit vorhanden

☐ **NO-GO wenn:**

- Score < 80 ODER Risk-Score > 6
 - Keine klaren Service-Boundaries erkennbar
 - Team-Turnover > 30% erwartet
 - Mission-critical ohne Rollback-Option
-



Teil 7: Business-Case-Template

7.1 Kosten-Nutzen-Analyse

Migration-Kosten:

- Development: ____ Personenmonate x €____ = €____
- Infrastructure: €____ (Container Platform, Monitoring, etc.)
- Training: €____ (Team-Weiterbildung)
- **Total Migration Cost:** €____

Laufende Kosten-Änderung:

- Infrastructure: +/- €____ pro Monat
- Operations: +/- €____ pro Monat (mehr DevOps-Aufwand)
- Development Velocity: +/- €____ pro Monat (nach Stabilisierung)

7.2 Erwartete Benefits

- ☐ **Faster Time-to-Market:** ____ % weniger Feature-Entwicklungszeit
- ☐ **Independent Deployments:** ____ x häufigere Releases
- ☐ **Team Autonomy:** ____ % weniger Cross-Team-Dependencies
- ☐ **Scalability:** ____ % bessere Resource-Utilization
- ☐ **Technology Diversity:** ____ neue Tech-Stacks möglich

7.3 ROI-Berechnung

Break-Even-Point: Nach ____ Monaten

3-Jahres-ROI: ____%



Teil 8: Nächste Schritte

8.1 Immediate Actions (nächste 4 Wochen)

- ☐ **Service-Boundaries definieren** - Domain-Storming-Workshop
- ☐ **Proof-of-Concept** - Ein kleiner Service extrahieren
- ☐ **Team-Training planen** - Container, Monitoring, Distributed Systems
- ☐ **Infrastructure-Evaluation** - K8s-Platform, Service Mesh evaluieren

8.2 Short-term (3 Monate)

- ☐ **Pilot-Service in Production** - Non-kritischer Service als Test
- ☐ **Monitoring-Stack etablieren** - Centralized Logging, Metrics, Tracing
- ☐ **CI/CD für Services** - Pipeline-Template für neue Services
- ☐ **Data-Migration-Strategy** - Event Sourcing, Database-Splitting POC

8.3 Medium-term (6-12 Monate)

- ☐ **Core-Service-Migration** - Business-kritische Services migrieren
- ☐ **Cross-Service-Communication** - Events, APIs, Consistency-Patterns

- ☐ **Performance-Optimization** - Service-to-Service-Latency optimieren
 - ☐ **Team-Scaling** - Ownership-Modell für Services etablieren
-

Kontakt & Support

Fragen zu diesem Assessment?

 **Direkt an Elyndra:** elyndra.valen@java-developer.online

Interesse an persönlicher Beratung?

Schreibt Elyndra eine E-Mail mit euren Assessment-Ergebnissen - sie meldet sich für ein kostenloses 30-Min-Gespräch!

Weiterführende Ressourcen

Empfohlene Bücher:

- **"Microservices Patterns"** - Chris Richardson
- **"Building Microservices"** - Sam Newman
- **"Monolith to Microservices"** - Sam Newman

Useful Tools:

- **Domain Modeling:** Event Storming, Domain Storytelling
- **Architecture:** C4 Model, Arc42 Template
- **Migration:** Strangler Fig, Branch by Abstraction

Java Fleet Blog-Serie (Community-driven):

 **Bereits verfügbar:**

- **"Microservices-Migration Assessment"** - Diese Checkliste zum Download

 **Geplant (bei Community-Interesse):**

- **"Monolith zu Microservices"** - Schritt-für-Schritt-Anleitung
- **"Service-Boundaries finden"** - Domain-Driven Design in der Praxis
- **"Microservices Testing-Strategy"** - Contract Testing, Service Virtualization

 **Community-Feedback gewünscht!**

 **Schreibt uns:** blog@javafleet.de

 **Twitter:** @JavaFleetSystems

 **Kommentiert unter diesem Post:** Welche Themen interessieren euch am meisten?

Je nach Interesse vertiefen wir:

- Domain-Analyse-Workshops (Event Storming)
- Strangler Fig Pattern in der Praxis
- Database-Decomposition-Strategien

- Team-Organization für Microservices
- Monitoring & Observability-Setup

© 2025 Java Fleet Systems Consulting - Kostenlose Nutzung für interne Assessments erlaubt

Dieses Assessment basiert auf 50+ Microservices-Migrationen in Enterprise-Umgebungen.