Architektur-Bewertung: EPIC / GPA

Datum: 2026-03-16 **Grundlage:** Analyse nach T300 FE Refactoring Analyse (DHBW / GEMÜ) **Projektstatus:** Aktiv in Entwicklung

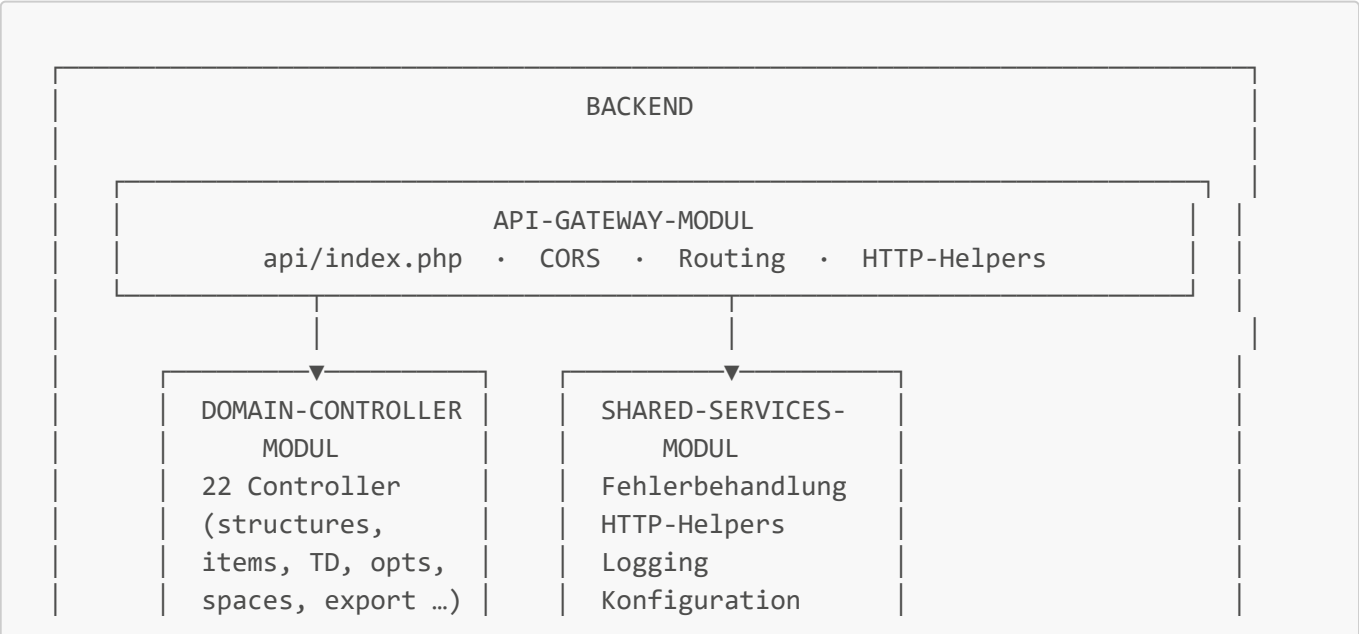
1. Projektübersicht

EPIC/GPA ist eine PHP/MySQL-Webanwendung zur Verwaltung von Produktstrukturen, technischen Daten, Komponenten und deren Beziehungen. Sie läuft auf XAMPP (Windows), ohne Framework, Build-Tools oder Package-Manager — reines PHP-Backend mit Vanilla-JavaScript + Bootstrap 5 im Frontend.

Kennzahl	Wert
PHP-Backend (gesamt)	~11.400 Zeilen
API-Controller	22 Dateien, ~9.400 Zeilen
Business-Logic-Services	10 Dateien, ~6.100 Zeilen
JavaScript-Frontend	~27.900 Zeilen, 19 Dateien
API-Endpunkte	50+ (REST)
Datenbankschema	36,5 MB (vollständig mit Daten)
Externe Microservices	1 (Java/Choco Constraint Solver)

2. Grundprinzip der optimierten Architektur (Kap. 4.1 — adaptiert für EPIC)

Das folgende Diagramm überträgt das Modular-Monolith-Prinzip aus dem Referenzdokument auf die EPIC-Domäne: statt SimApps und Job-Management stehen hier das Domain-Service-Modell und der Choco-Adapter im Vordergrund.



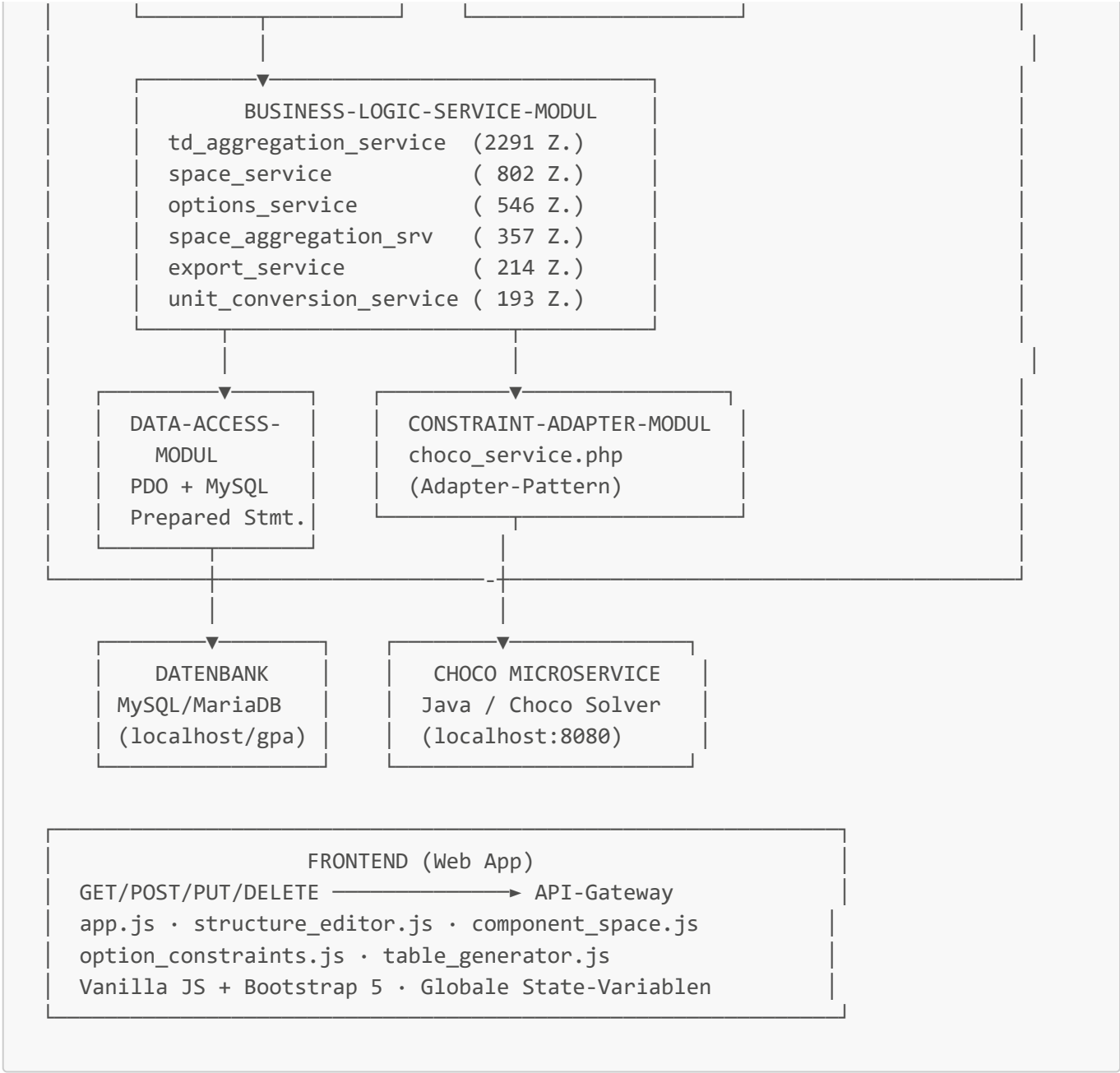


Abbildung 1: Optimierte Architektur für EPIC (analog Kap. 4.1)

---

3. Bewertung funktionaler Anforderungen

Domänenfunktionalität — **SEHR GUT** ☑

Die Kernfunktionen sind vollständig und konsistent implementiert:

- **Produktstruktur-Verwaltung:** Hierarchische Baumstruktur mit vollständigem CRUD (`structures`, `structure_items`)
- **Technische Datenverwaltung:** Direkte, vererbte und aggregierte TD-Werte mit vollständiger Tabellengruppen- und Composite-Logik
- **Raum-/Variantenmodell:** Vollständige Kombinations- und Abdeckungsvalidierung inkl. Lückendetektierung
- **Constraint-System:** Vier Constraint-Typen (required/forbidden/exclusive/dependent) mit Java-Choco-Integration

- **Export:** ST4 und generischer CSV-Export
- **Einheitensystem:** Lineares Einheitenkonvertierungssystem (Faktor + Offset)

Funktionale Lücken — **KRITISCH** ❌

Analog zur PDF-Analyse fehlen folgende Bereiche vollständig:

Fehlende Funktion	Auswirkung
Kein User-Management	Alle Benutzer sehen und bearbeiten alle Daten
Keine Authentifizierung	Jeder mit Netzwerkzugriff kann alle API-Endpoints aufrufen
Keine Rollenverwaltung	Kein Unterschied zwischen Admin, Redakteur, Leser
Kein Audit-Logging	Keine Nachvollziehbarkeit von Änderungen
Keine Versionierung	Kein History-Tracking für TD-Änderungen (Plan: `toDo.md`)
Kein Projekt-/Mandantenkonzept	Keine Datentrennung zwischen Projekten oder Teams

> Implementierungsplan für Versionierung + Freigabe-Workflow: siehe `docs/toDo.md`

---

4. Bewertung nicht-funktionaler Anforderungen

4.1 Wartbarkeit — **MODERAT** ⚠

Positiv:

- Klare Schichtenarchitektur: Frontend → Controller → Service → DB
- Konsistente Controller-Konvention (`handle{Resource}()`)
- PDO mit Named Placeholders durchgehend
- Umfangreiche Dokumentation vorhanden (`docs/`)
- Separater Java-Microservice für den komplexen Constraint-Solver ist ein sinnvolles Adapter-Muster

Kritisch:

- **Monolithische Services:** `td_aggregation_service.php` mit 2.291 Zeilen bündelt zu viele Verantwortlichkeiten — direkte, vererbte und aggregierte TD-Logik in einer Datei
- **Code-Duplikation im Frontend:** 27.900 Zeilen Vanilla-JS mit globalen State-Variablen; `app.js` (5.490 Z.) und `structure_editor.js` (7.674 Z.) sind sehr groß und schwer isoliert testbar
- **Keine automatisierten Tests:** Nur manuelle Test-Skripte in `Tests/`. Regressionsgefahr bei jeder Änderung an den Service-Schichten
- **Kein Dependency-Management:** Kein Composer für PHP, kein npm für JS — externe Bibliotheken werden manuell eingepflegt
- **Hybrid-Zustand Legacy/Neu:** Alte Tabellen (`products`, `components`, `sub_components`) und neue `items`-Tabelle koexistieren im Schema (`database/gpa.sql`)

Empfehlung: `td_aggregation_service.php` aufteilen in mindestens: `td_direct_service`, `td_inheritance_service`, `td_composite_service`.

4.2 Skalierbarkeit — **EINGESCHRÄNKT** ⚠

Aspekt	Bewertung
Vertikale Skalierung	Möglich (XAMPP-Limits)
Horizontale Skalierung	Nicht vorgesehen — kein Session-Sharing, kein Caching
Datenbankzugriff	Kein Connection-Pooling, neue PDO-Verbindung pro Request
Caching	Kein Caching-System; wiederholte komplexe Aggregationsabfragen belasten die DB
Microservice-Skalierung	Choco-Service nur als Single Instance; kein Load-Balancing

Kontext: Für eine interne XAMPP-Anwendung mit begrenzter Nutzerzahl ist die Skalierbarkeit akzeptabel. Bei wachsender Nutzerzahl oder größeren Datensätzen sind DB-Abfragen in `td_aggregation_service.php` der primäre Flaschenhals.

4.3 Sicherheit — **NICHT PRODUKTIONSREIF** ❌

Dies ist die kritischste Dimension, identisch zur Einschätzung im PDF:

Kritische Schwachstellen

Problem	Datei	Details	Risiko
Keine Authentifizierung	Alle API-Endpunkte	Kein Login, keine Rollenprüfung — jeder kann Daten ändern/löschen	**KRITISCH**
Keine Autorisierung / RBAC	Alle API-Endpunkte	Kein Unterschied zwischen Rollen	**KRITISCH**
CORS offen für alle Origins	`api/index.php`	`Access-Control-Allow-Origin: *` erlaubt Zugriff von jeder Domain	**HOCH**
Kein CSRF-Schutz	Alle API-Endpunkte	Keine Token-Validierung bei POST/PUT/DELETE	**HOCH**
Fehlende Security-Headers	`api/index.php`	`X-Frame-Options`, `Content-Security-Policy`, `X-Content-Type-Options` fehlen	**HOCH**
Fehlermeldungen leaken DB-Details	`config/database.php`	`$e->getMessage()` wird ungefiltert an den Client zurückgegeben	**HOCH**
Datei-Upload unsicher	`api/controllers/items.php`	MIME-Typ-Prüfung allein reicht nicht; keine Doppelendungs-Prüfung (z.B. `file.php.jpg`)	**HOCH**

Problem	Datei	Details	Risiko
Hardcoded Credentials	`config/database.php`	`root` ohne Passwort im Quellcode	**MITTEL** (dev-only)
HTTPS/TLS	Nicht konfiguriert	Alle Kommunikation läuft unverschlüsselt	**MITTEL** (lokal)
Input-Validierung	API-Controller	Keine Längenbegrenzung, kein Whitelist-Check für `item_type` o.Ä.	**MITTEL**

Positiv hervorgehoben

Aspekt	Status
SQL-Injection	PDO + Prepared Statements durchgehend ☑
Datei-Upload-Ausführungsschutz	PHP-Ausführung via `.htaccess` in `uploads/` gesperrt ☑

> **Hinweis:** Der Upload-Schutz in `uploads/.htaccess` schützt vor Code-Ausführung, aber die Doppelendungs-Prüfung im Controller (`items.php`) fehlt noch.

4.4 Performance — **AUSREICHEND** ⚠

Positiv:

- Synchrones Request-Response-Modell reicht für das Nutzungsszenario
- Datei-Uploads werden außerhalb der Datenbank in `uploads/` gespeichert

Kritisch:

- **N+1 Query Problem** in `api/controllers/items.php` (~Zeile 466): Query in Schleife für jede Space-Definition
- **N+1 Query Problem** in `api/controllers/components.php` (~Zeile 57): Query in Schleife für Option-Values
- `td_aggregation_service.php` führt für jeden Aggregationsaufruf mehrere verschachtelte Datenbankabfragen durch — kein Caching
- **Fehlende Datenbankindizes:** FK-Spalten in `component_space_definitions` und `component_space_option_values` ohne Index — langsame JOINs bei wachsendem Datenbestand
- Kein Query-Profiling oder Slow-Query-Log aktiv
- Frontend lädt ~28.000 Zeilen JS ohne Bundling/Minification — initiale Ladezeit suboptimal
- Kein HTTP-Caching für statische Assets (kein `Cache-Control`-Header)

4.5 Stabilität — **AUSREICHEND** ⚠

Positiv:

- PDO `ERRMODE_EXCEPTION` verhindert stilles Fehlschlagen von DB-Operationen
- Konsistentes `fetchAll() ?: []`-Muster verhindert Undefined-Variable-Fehler

- `serverError()`-Helper mit Logging vorhanden

Kritisch:

- **Kein Monitoring:** Probleme werden erst bemerkt, wenn Nutzer sie melden
- **Single Point of Failure:** MySQL-Instanz ohne Replikation; Ausfall = Totalausfall
- **Inkonsistente Transaktionsbehandlung:** Mehrere Controller fangen generische `Exception` statt `PDOException`; fehlende `inTransaction()`-Checks
- **Fehlende Foreign Keys:** `component_space_definitions` und `component_space_option_values` haben keine FK-Constraints → Orphaned Records möglich. Bewusst dokumentiert im Schema: `-- fk_csd_item omitted`
- **Choco-Microservice:** Kein automatischer Restart bei Absturz; Status nur über manuelle Batch-Skripte
- **Keine Health-Checks:** Kein Endpunkt zur Systemstatusabfrage

---

4.6 Frontend-Qualität — **VERBESSERUNGSBEDARF** ⚠

Hoch

Problem	Dateien	Details
Globale Variablen-Verschmutzung	`app.js`, `component_space.js`, `structure_editor.js`	30+ globale `let`-Variablen, keine ES-Module/Closures
Window-Objekt Pollution	`consolidated_index.js`, `interface_management.js`	Funktionen direkt auf `window` zugewiesen

Mittel

Problem	Dateien	Details
XSS in Fehlermeldungen	`component_space.js`, `consolidated_index.js`	`error.message` wird unescaped in `innerHTML` eingefügt
Memory Leaks	`structure_view_functions.js`, `structure_editor.js`	Event-Listener werden bei Modal-Schließung nicht entfernt
Race Condition	`structure_editor.js`	`setTimeout` nach `loadStructureTree()` — Race zwischen Laden und Auto-Selection
Inkonsistente API-Pfade	`component_space.js`, `technical_data_entities.js`	Manche nutzen `API_BASE`, andere hardcoded `'api/index.php/...'`
Toter Code	`app.js`	Große auskommentierte Code-Blöcke

---

5. Gesamtbewertung

Dimension	Bewertung	Note
Domänenfunktionalität	Sehr gut	☑ Kernfunktionen vollständig

Dimension	Bewertung	Note
User-Management / Auth	Nicht vorhanden	❌ Kritische Lücke
Wartbarkeit	Moderat	⚠ Monolithische Services, fehlende Tests
Skalierbarkeit	Eingeschränkt	⚠ Für internes Tool ausreichend
Sicherheit	Nicht produktionsreif	❌ Nur für Vertrauensnetzwerk geeignet
Performance	Ausreichend	⚠ N+1 Queries, kein Caching, kein Bundling
Stabilität	Ausreichend	⚠ Kein Monitoring, fehlende FK-Constraints
Frontend-Qualität	Verbesserungsbedarf	⚠ Globale Variablen, XSS-Risiken

Zusammenfassung: Analog zur Einschätzung aus dem Referenzdokument ist EPIC in seiner aktuellen Form **gut für interne Entwicklung und Prototyping in einer vertrauenswürdigen Umgebung (Intranet)** geeignet. Die fachliche Tiefe der Domänenlogik (TD-Aggregation, Space-Modell, Constraint-System) ist bemerkenswert. Für einen produktiven Einsatz mit mehreren Nutzern, regulatorischen Anforderungen oder öffentlichem Netzwerkzugang wären umfangreiche Änderungen vor allem in **Sicherheit, Authentifizierung und Monitoring** erforderlich.

---

6. Priorisierte Handlungsempfehlungen

Sofort (falls öffentlich erreichbar)

- ☐ **Authentifizierung einführen** — mindestens HTTP Basic Auth (Apache) oder Session-basiertes Login in PHP
- ☐ **CORS einschränken** — `Access-Control-Allow-Origin` auf konkrete Herkunft begrenzen (`api/index.php`)
- ☐ **CSRF-Tokens** für alle POST/PUT/DELETE-Requests einführen
- ☐ **Security-Headers** hinzufügen: `X-Frame-Options`, `Content-Security-Policy`, `X-Content-Type-Options` (`api/index.php`)

Kurzfristig

- ☐ **Foreign Keys ergänzen** — `component_space_definitions` und `component_space_option_values` (`database/gpa.sql`)
- ☐ **Datenbankindizes ergänzen** — FK-Spalten in `component_space_*`-Tabellen
- ☐ **N+1 Queries beheben** — durch JOINs ersetzen (`items.php` ~466, `components.php` ~57)
- ☐ **Fehlermeldungen sanitizen** — `$e->getMessage()` nicht an Client zurückgeben (`config/database.php`)
- ☐ **DB-Credentials auslagern** — `.env`-Datei, aus Versionskontrolle ausschließen
- ☐ **Upload-Schutz verbessern** — Doppelendungs-Prüfung in `items.php` ergänzen

Mittelfristig

- ☐ **XSS beheben** — `error.message` vor `innerHTML`-Einbindung escapen (`component_space.js`, `consolidated_index.js`)

- ☐ **Inkonsistente API-Pfade vereinheitlichen** — `API_BASE` durchgehend verwenden (`component_space.js`, `technical_data_entities.js`)
- ☐ **Globale JS-Variablen reduzieren** — ES-Module einführen, `window`-Pollution beseitigen
- ☐ **`td_aggregation_service.php` aufteilen** — in `td_direct_service`, `td_inheritance_service`, `td_composite_service`
- ☐ **Automatisierte Tests einführen** — zumindest für TD-Aggregation und Space-Berechnung

Langfristig

- ☐ **Versionierung + Freigabe-Workflow implementieren** — vollständiger Plan in `docs/toDo.md`
- ☐ **Modul-Trennung schärfen** — Adapter-Pattern für Choco-Service formal definieren (Interface), analog Kap. 4.1
- ☐ **Monitoring einführen** — Logging in Datei + Health-Check-Endpoint (`/api/health`)
- ☐ **Caching für Aggregationsabfragen** — serverseitiger Cache (z.B. APCu) für teure TD-Aggregationen
- ☐ **Frontend modularisieren** — Funktionsgruppen in ES-Module aufteilen; `structure_editor.js` und `app.js` aufbrechen

---

Erstellt auf Basis: T300_FE_Refactoring_Analyse.pdf (DHBW / GEMÜ, 2026) Ergänzt durch: `docs/toDo.md` (Projektanalyse — Identifizierte Probleme)