3 Analyse der bestehenden Architektur

Die vorliegende Analyse untersucht die bestehende Backend-Architektur der Plattform für Simulationsanwendungen im Hinblick auf funktionale Anforderungen wie verschiedene Benutzeranforderungen und Funktionalitäten, die die Anwendung bieten muss, sowie nicht-funktionale Anforderungen, wie Wartbarkeit, Skalierbarkeit, Sicherheit, Performance und Stabilität.

3.1 Bewertung funktionaler Anforderungen

Benutzeranforderungen

Aus funktionaler Perspektive erfüllt die Architektur die Kernaufgaben des Systems sehr gut. Die Plattform ermöglicht die gleichzeitige Ausführung mehrerer unabhängiger Simulationen, die jeweils in einem eigenen Service-Container laufen. Jeder Service implementiert eine standardisierte Schnittstelle mit einer run_method zur Ausführung der Simulation und einer verify_method zur Validierung der Eingabeparameter vor der Ausführung. Dies gewährleistet konsistente Funktionalität über alle Simulationstypen hinweg. Für die Eingabe der Parameter steht ein Hilfefeld zur Unterstützung zur Verfügung. Die Job-Verwaltung ist durch die Datenbank implementiert, bei der jeder Auftrag mit seinen Input- und Output-Parametern sowie einem Status-Tracking versehen wird. Das System nutzt eine State-Machine zur Kontrolle des Simulationsablaufs mit definierten Zuständen wie IDLE, STARTING, RUNNING und STOPPING, was eine strukturierte Verwaltung des Job-Lebenszyklus ermöglicht. Nach Abschluss einer Simulation stehen verschiedene Möglichkeiten zum Abruf von Ergebnissen bereit, einschließlich des Zugriffs auf einzelne HTML-Dateien sowie des Downloads kompletter Ergebnisordner als ZIP-Archive. Die Trennung zwischen Auftragsannahme und Ausführung durch das Queue-System ermöglicht es, dass Benutzer nicht auf die Fertigstellung ressourcenintensiver Berechnungen warten müssen. Die WebSocket-Integration bietet Echtzeit-Rückmeldungen, ob der Server erreichbar ist. Das Frontend präsentiert die Anwendungen in einer benutzerfreundlichen Oberfläche und ermöglicht

Benutzern, Simulationsparameter einzugeben, Berechnungen zu starten und Ergebnisse zu visualisieren.

Allerdings gibt es erhebliche funktionale Lücken. Das System verfügt über kein echtes User-Management, Authentifizierungs- oder Autorisierungsmechanismen, wodurch alle Benutzer denselben Zugriff auf alle Funktionen haben. Rollenverwaltung zur Unterscheidung zwischen Administratoren, Ingenieuren und nur-lesenden Benutzern ist nicht implementiert. Hinzu kommt die fehlende Möglichkeit, Daten nach Benutzern zu isolieren oder Zugriffe basierend auf Berechtigungen zu kontrollieren. Das Audit-Logging ist nur minimal vorhanden, was für industrielle Anwendungen mit regulatorischen Anforderungen kritisch ist (vgl. [1]). Projektmanagement-Features wie das Gruppieren, Organisieren oder Archivieren von Simulationen pro Benutzer oder Projekt sind nicht vorhanden. Genauso ist kein Abbrechen laufender Simulationen möglich.

3.2 Bewertung nicht-funktionaler Anforderungen

Wartbarkeit

Die Wartbarkeit des Codes ist moderat. Die Dokumentation ist umfangreich mit einer Backend- und Frontend-Dokumentation vorhanden, allerdings nicht überall inhaltlich korrekt. Der Code folgt teilweise Best Practices durch die Nutzung von SQLModel für type-safe ORM-Operationen (vgl. [2]). Positiv zu bewerten ist die klare Schichtenarchitektur mit definierter Trennung zwischen Frontend, Backend und Datenhaltung. Die Existenz separater Entwicklungs- und Produktivumgebungen ermöglicht die Weiterentwicklung ohne Beeinträchtigung des laufenden Betriebs.

Kritisch ist jedoch die monolithische Struktur des Backends zu bewerten. Das Backend bündelt zahlreiche Verantwortlichkeiten in einer gemeinsamen Codebasis, darunter API-Gateway-Funktionalität, Geschäftslogik, Datenbankzugriff, Dateiverwaltung und die Orchestrierung der Simulationsausführung. Diese Zusammenführung erschwert die getrennte Weiterentwicklung einzelner Funktionsbereiche und erhöht die Komplexität bei der Fehlersuche. (vgl. [3])

Zudem führt das Modell eines dedizierten Backends pro SimApp zu Code-Duplikation und erhöhtem Wartungsaufwand, da bei mehreren Simulationsanwendungen gemeinsame Funktionalitäten mehrfach implementiert werden. Dies zeigt sich unter anderem daran, dass vier verschiedene *main.py*-Dateien existieren, deren Inhalt nahezu identisch ist. Die Fehlerbehandlung ist mangelhaft, mit mehreren Fällen von try-except: *pass*, das Fehler ignoriert und das Debugging erschwert. Unit-Tests und Integrationstests sind nicht vorhanden, was das Risiko von Regressionen erhöht (vgl. [4]). Das System hat über 75 externe Abhängigkeiten in der requirements.txt, darunter die gesamte ANSYS-Suite, was die Komplexität erhöht und die Deployment-Zeit verlängert.

Skalierbarkeit

Die Skalierbarkeit der Architektur stellt eine der wesentlichen Herausforderungen dar. Das Konzept eines dedizierten Backends pro SimApp impliziert, dass bei wachsender Anzahl von Simulationsanwendungen proportional mehr Backend-Instanzen betrieben werden müssen. Dies führt zu erhöhtem Ressourcenbedarf und Verwaltungsaufwand. Horizontale Skalierung einzelner Backends bei steigender Last ist aufgrund der monolithischen Struktur nur eingeschränkt möglich, da das gesamte Backend repliziert werden müsste, auch wenn nur einzelne Komponenten unter Last stehen (vgl. [5]).

Die Job-Queue funktioniert nur innerhalb einer einzelnen Instanz und ist so nicht zwischen mehreren Maschinen verteilbar. Dies bedeutet, dass eine horizontale Skalierung der Simulationskapazität nicht möglich ist. Zudem gibt es kein Caching-System, sodass wiederholte Abfragen die Datenbank unnötig belasten. Die gemeinsame Nutzung von PostgreSQL und Blob Storage durch alle Backend-Instanzen könnte zu Bottlenecks führen.

Sicherheit

Bezüglich der Sicherheit befindet sich das System in einem nicht produktionsreifen Zustand. Die Datenbank-Anmeldedaten sind direkt im Code hartcodiert, was ein Sicherheitsrisiko darstellt. Es sind keine Authentifizierungs- und

Autorisierungsmechanismen implementiert, um sicherzustellen, dass nur berechtigte Anwender Simulationen starten oder Ergebnisse abrufen können. Jeder, der die IP-Adresse und den Port kennt, kann API-Aufrufe durchführen. Es gibt keine HTTPS/TLS-Verschlüsselung für die Datenübertragung, was bedeutet, dass alle Kommunikation unverschlüsselt über das Netzwerk läuft. Die CORS-Einstellung ist aktiviert, ohne dass eine spezifische Whitelist definiert ist, was das System anfällig für Cross-Site-Request-Forgery-Angriffe macht. Es gibt keine Input-Validierung auf SQL-Injection, da Benutzer-Eingaben direkt in Datenbankabfragen eingebunden werden können. (vgl. [6])

Die Speicherung von Nutzerdaten und Simulationsergebnissen in PostgreSQL und Blob Storage erfordert angemessene Zugriffskontrollmechanismen und gegebenenfalls Verschlüsselung sensibler Daten. Ein Sicherheitsrisiko könnte in der fehlenden Isolation zwischen verschiedenen SimApps bestehen. Wenn mehrere Simulationsanwendungen dieselbe Datenbankinstanz nutzen, muss sichergestellt sein, dass strikte Datenkapselung gewährleistet ist und keine unautorisierten Zugriffe auf Daten anderer Anwendungen möglich sind. (vgl. [6])

Performance

Die Performance der Architektur wird durch mehrere Faktoren beeinflusst. Die asynchrone Verarbeitung von Simulationsaufträgen über das Queue-System ermöglicht eine Entkopplung zwischen API-Response-Zeiten und Simulationsdauer, was die wahrgenommene Performance aus Nutzersicht verbessert (vgl. [7]). Die WebSocket-basierte Statusaktualisierung minimiert unnötige Polling-Requests und reduziert dadurch Netzwerklast und Serverlast. FastAPI bietet moderne asynchrone Möglichkeiten durch das ASGI-Protokoll (vgl. [8]). Allerdings können die Simulationsservices aufgrund von Lizenzkosten nur einen Job nach dem anderen verarbeiten.

Die Verwendung von Blob Storage für Dateien und Medien ist performancetechnisch sinnvoll, da diese Last von der relationalen Datenbank ferngehalten wird. Allerdings ist die Koordination zwischen Datenbankeinträgen und Blob Storage-Objekten

performancekritisch: Bei jedem Dateizugriff muss zunächst der Dateipfad aus der Datenbank abgerufen werden, bevor die Datei aus dem Storage geladen werden kann. Caching-Strategien könnten hier Latenzzeiten reduzieren.

Kritische Performance-Aspekte ergeben sich aus der Datenbankarchitektur. Alle Backends greifen auf eine gemeinsame PostgreSQL-Instanz zu, was bei steigender Anzahl paralleler Anfragen zu Contention und erhöhten Antwortzeiten führen kann. Die Notwendigkeit, für jeden Datenbankzugriff eine Verbindung herzustellen, könnte durch Connection-Pooling optimiert werden. (vgl. [9])

Stabilität

Die Stabilität und Fehlertoleranz ist ein weiterer Schwachpunkt. Das System verfügt über keine Hochverfügbarkeit. Wenn ein Service abstürzt, gehen alle laufenden Jobs verloren, da sie nur im Speicher gehalten werden und nicht persistent sind. Es gibt keinen automatischen Restart von fehlgeschlagenen Services. Die Fehlerbehandlung ist unzureichend: Wenn eine Simulation fehlschlägt, gibt es keine Retry-Logik und keine Dead-Letter-Queue für gescheiterte Jobs. Der einzige aktuelle Mechanismus ist die State-Machine für grundlegende Zustandsübergänge, aber sie behandelt Fehlerfälle nicht angemessen. Die Datenbank ist ein Single Point of Failure ohne Replikation. Wenn die PostgreSQL-Instanz ausfällt, ist das gesamte System nicht funktionsfähig. Graceful Shutdown ist teilweise implementiert, aber unvollständig. Health-Checks sind basic und bieten keine detaillierten Metriken über die Systemgesundheit. Es gibt kein zentrales Monitoring oder Alerting System, daher können Probleme erst bemerkt werden, wenn Benutzer sie melden. (vgl. [10])

Die Konsistenz zwischen PostgreSQL-Datenbank und Blob Storage ist ein weiterer stabilitätskritischer Aspekt. Ohne entsprechende Transaktionsmechanismen oder Kompensationslogik können Inkonsistenzen entstehen, wenn beispielsweise ein Datenbankeintrag erfolgreich angelegt wird, aber das zugehörige File-Upload fehlschlägt.

Die Architektur sollte Idempotenz-Mechanismen vorsehen, um bei Retries keine Duplikate zu erzeugen. (vgl. [11])

Zusammengefasst ist das SimApps-System in seiner aktuellen Form gut für Prototyping, Entwicklung und interne Ingenieursanwendungen in vertrauenswürdigen Umgebungen geeignet. Die funktionalen Anforderungen für die Simulationsausführung sind erfüllt. Jedoch ist das System für Produktionsbetrieb mit mehreren Benutzern, regulatorischen Anforderungen oder Hochverfügbarkeitsbedarf nicht geeignet. Für einen produktiven Einsatz wären umfangreiche Änderungen in Sicherheit, Skalierbarkeit, Fehlertoleranz und Überwachung erforderlich.

4 Konzeption einer optimierten Backend-Architektur

Im Folgenden Kapitel wird die Konzeption einer optimierten Backend-Architektur vorgestellt mit Hinblick auf die zuvor identifizierten negativen Aspekte.

4.1 Grundprinzip

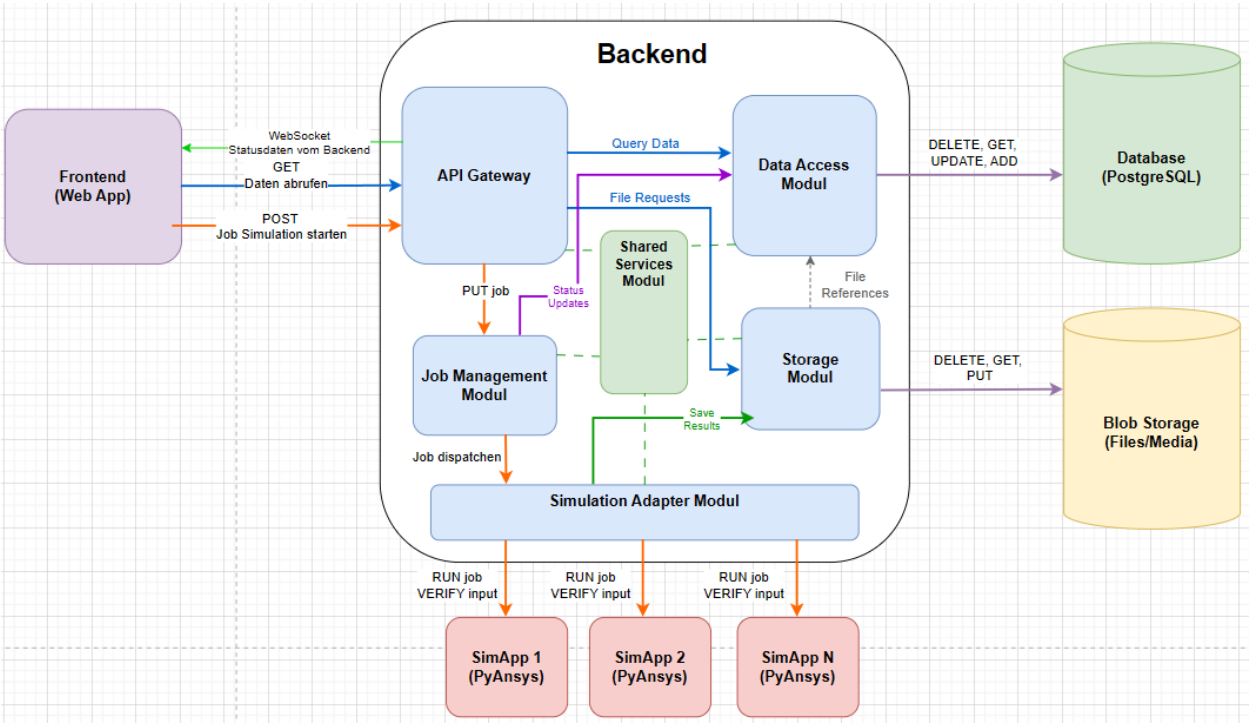

Abbildung 2: Optimierte Architektur

Dieses Konzept behält die monolithische Deployment-Unit bei, strukturiert jedoch das Backend intern in klar abgegrenzte Module mit definierten Schnittstellen. Jedes Modul kapselt eine spezifische Verantwortlichkeit und kommuniziert mit anderen Modulen ausschließlich über definierte APIs. Dies ermöglicht die Vorteile einfachen Deployments bei gleichzeitiger Verbesserung von Wartbarkeit und Testbarkeit.

4.2 Module

Im Folgenden wird der Inhalt der einzelnen Module grob beschrieben beziehungsweise der Aufbau der Architektur grob erklärt.

API-Gateway-Modul: Zentrale Eingangsstelle für alle HTTP- und WebSocket-Requests. Verantwortlich für Request-Routing, Validierung, Authentifizierung und Response-Formatierung. Delegiert Business-Logik an entsprechende Fachmodule.

Job-Management-Modul: Implementiert die Statemachine und Queue-Verwaltung. Kapselt die Geschäftslogik des Simulationslebenszyklus. Kommuniziert mit dem Simulation-Adapter-Modul zur Steuerung der SimApp.

Simulation-Adapter-Modul: Abstraktionsschicht zwischen Backend und SimApp. Implementiert ein Adapter-Pattern, das die Anbindung verschiedener Simulationswerkzeuge ermöglicht. PyAnsys-spezifische Logik ist nur in diesem Modul konzentriert.

Data-Access-Modul: Zentrale Schnittstelle zu PostgreSQL. Implementiert Repository-Pattern für Datenbankzugriffe. Kapselt SQL-Queries und ORM-Logik. Stellt anderen Modulen domänenorientierte APIs bereit.

Storage-Modul: Verwaltung von Dateien im Blob Storage. Koordiniert Uploads und Downloads. Implementiert Konsistenzmechanismen zwischen Datenbank-Referenzen und tatsächlichen Files.

Shared-Services-Modul: Querschnittsfunktionalitäten wie Logging, Monitoring, Konfigurationsmanagement und Fehlerbehandlung. Wird von allen anderen Modulen genutzt.

Shared-Backend-Ansatz

Ein zentraler Optimierungsaspekt dieses Konzepts besteht in der Einführung eines Shared-Backend-Ansatzes. Anstatt für jede SimApp ein dediziertes Backend zu

betreiben, wird ein gemeinsames Backend implementiert, das mehrere Simulationsanwendungen bedienen kann. Über Konfigurationsparameter oder Multitenancy-Mechanismen wird zur Laufzeit bestimmt, welche SimApp für einen konkreten Request zuständig ist. Dies reduziert Code-Duplikation drastisch und vereinfacht Wartung und Weiterentwicklung.

Die SimApp-spezifische Logik wird über das Simulation-Adapter-Modul abstrahiert. Für jede SimApp existiert eine konkrete Adapter-Implementierung, die das gemeinsame Adapter-Interface implementiert. Das Job-Management-Modul arbeitet ausschließlich gegen das Interface und ist damit unabhängig von spezifischen SimApp-Implementierungen. Neue Simulationsanwendungen können durch Hinzufügen einer neuen Adapter-Implementierung integriert werden, ohne das restliche Backend anzupassen.