2 Enterprise Engineering Standards: Architektur und Implementierung des Projekts 'MargenKalkulator'

Executive Summary

Enterprise Engineering Standards: Architektur und Implementierung des Projekts 'MargenKalkulator'

Executive Summary

Das Projekt 'MargenKalkulator' repräsentiert eine kritische Klasse von Unternehmensanwendungen, bei denen die Berechnungslogik – die Ermittlung von Margen, Preisen und Risikokoeffizienten – nicht nur eine funktionale Anforderung darstellt, sondern das primäre geistige Eigentum (Intellectual Property, IP) und den zentralen Wettbewerbsvorteil der Organisation bildet. Im Gegensatz zu standardisierten CRUD-Anwendungen (Create, Read, Update, Delete), die lediglich Daten verwalten, verarbeitet ein Margenkalkulator hochsensible finanzielle Eingangsparameter, um kommerziell tragfähige Ausgaben zu produzieren. Die Integrität dieser Berechnungen ist geschäftskritisch; ein Fehler oder ein Leck in diesem System könnte unmittelbare finanzielle Verluste oder den Verlust der Marktposition bedeuten.
Dieser Bericht definiert die umfassenden Enterprise-Engineering-Standards für die Implementierung des 'MargenKalkulator' unter Verwendung eines modernen Technologie-Stacks, bestehend aus **React** (Frontend), **Supabase** (Backend-as-a-Service) und **Lovable** (AI-gestützte UI-Entwicklung). Die Wahl von Supabase bietet ein robustes, auf PostgreSQL basierendes Fundament, verschiebt jedoch, im Vergleich zu traditionellen Application-Server-Architekturen, einen erheblichen Teil der Sicherheitsverantwortung von der Applikationsschicht direkt in die Datenbankschicht, spezifisch durch den Einsatz von Row Level Security (RLS). Darüber hinaus führt der Einsatz von Lovable zur Beschleunigung der Frontend-Entwicklung spezifische Herausforderungen in Bezug auf die Code-Herkunft (Provenance), die Wartbarkeit und das Risiko ein, dass Geschäftslogik unbeabsichtigt in das clientseitige Bundle gelangt und somit exponiert wird.
Die nachfolgend detaillierten Standards sind für die Compliance und Sicherheit des Projekts nicht verhandelbar und basieren auf einer tiefgehenden Analyse der aktuellen technologischen Fähigkeiten und Einschränkungen der gewählten Plattformen.[1]
Zentrale Säulen dieser Architektur sind:

1. **Zero-Trust-Datenbankarchitektur:** Jeder Datenzugriff muss durch strikte RLS-Richtlinien legitimiert werden. Es gibt kein implizites Vertrauen, selbst nicht für authentifizierte Benutzer.
2. **Server-Side Execution Boundary (Black-Box-Schutz):** Die Margin-Berechnungslogik darf unter keinen Umständen auf dem Client ausgeführt werden. Sie muss exklusiv in Supabase Edge Functions (Deno) oder nativen PostgreSQL-Funktionen residieren.
3. **Memory-Safe Document Generation:** Die Generierung von PDF-Dokumenten im Serverless-Kontext erfordert spezifische Streaming-Protokolle, um Speicherüberläufe (OOM) zu verhindern, die in isolierten V8-Umgebungen wie Deno häufig auftreten.
4. **Immutable Build Pipelines:** Die Integration von KI-generiertem Code erfordert rigide CI/CD-Prozesse, die mittels kryptografischer Prüfsummen sicherstellen, dass sicherheitskritische Module nicht manipuliert wurden.

1. Architektonische Prinzipien und Systemintegration

Die Architektur für den 'MargenKalkulator' wird als **Headless Calculation Engine** mit einem **Thin Client Interface** definiert. Diese Unterscheidung ist fundamental für das Sicherheitsmodell. Das React-Frontend, welches primär durch das Tool Lovable generiert wird, dient ausschließlich als Schicht zur Datenerfassung und Visualisierung. Es besitzt keinerlei Wissen über die zugrundeliegenden Regeln der

Margenberechnung. Diese strikte Trennung dient dem Schutz des geistigen Eigentums und der Verhinderung von Reverse-Engineering durch Konkurrenten oder böswillige Akteure.

1.1 Definition der "Black Box"-Grenze

In der traditionellen Webentwicklung neigen Entwickler aus Gründen der User Experience (UX) und Latenzvermeidung dazu, Validierungslogik und einfache Berechnungen im Frontend zu platzieren. Für den 'MargenKalkulator' ist diese Praxis für jegliche Logik, die Rückschlüsse auf die Margenstruktur zulässt, strengstens untersagt. Das Prinzip der "Black Box" verlangt, dass der Client Inputs liefert und Outputs empfängt, ohne Einblick in den Transformationsprozess zu erhalten.
Der architektonische Standard mandatiert die folgenden Verantwortlichkeiten der Komponenten:

- **React (Lovable Generated):** Dieses Modul ist ausschließlich verantwortlich für das Rendering von Formularen, die Verwaltung des Benutzereingabestatus und die Anzeige des finalisierten PDFs oder Berechnungsergebnisses. Es darf *keine* Koeffizienten, Formeln, Margenschwellenwerte oder bedingte Logik enthalten, die auf Geschäftsgeheimnissen basiert.[2] Die Gefahr bei Tools wie Lovable besteht darin, dass sie durch Prompt-Engineering dazu verleitet werden könnten, Geschäftslogik direkt in den React-Code zu schreiben. Dies muss durch Code-Reviews und Linter-Regeln unterbunden werden.
- **Supabase Client (SDK):** Wird für die Authentifizierung und das Abrufen von *öffentlichen* Benutzerprofildaten verwendet. Es ist dem Client explizit untersagt, direkt Tabellen abzufragen, die zur margin_rules-Domäne gehören. Selbst wenn RLS den Zugriff verhindern würde, stellt der Versuch einer direkten Abfrage ein Architektur-Anti-Pattern dar.
- **Supabase Edge Functions (Deno):** Der Kern der "Black Box". Diese Funktionen empfangen Eingaben, rufen private Regeln aus der Datenbank ab (unter Verwendung eines Service-Role-Keys, um RLS für interne Berechnungen zu umgehen), berechnen das Ergebnis und geben nur den finalen Margenwert oder das generierte Dokument zurück. Die Verwendung von TypeScript in einer Deno-Umgebung bietet hierbei Typsicherheit, ohne den Code an den Browser auszuliefern.[3]
- **PostgreSQL:** Der authoritative Zustandsspeicher. Er erzwingt die Datenisolation via RLS und speichert hochwertige Konfigurationsdaten in Tabellen, die für die öffentliche API unzugänglich sind. Die Datenbank fungiert hier nicht nur als Speicher, sondern als aktive Sicherheitsbarriere.[1]

1.2 Integration von Lovable und "Core vs. Shell" Architektur

Lovable beschleunigt die Frontend-Entwicklung signifikant, führt jedoch das Risiko ein, manuelle architektonische Einschränkungen zu überschreiben. Da Lovable auf der Basis von Prompts und existierendem Code arbeitet, kann es vorkommen, dass bei einer Aktualisierung der UI manuell eingefügte Sicherheitslogik überschrieben wird, wenn diese nicht korrekt isoliert ist.[6]
Um dieses Risiko zu mitigieren, muss das Projekt eine **"Core vs. Shell"**-Verzeichnisstruktur adoptieren:

- **The Shell (Lovable Managed):** Die Verzeichnisse /src/components, /src/pages und UI-Layout-Dateien werden primär durch Lovable's KI verwaltet. Entwickler sollten diese Dateien als semi-ephemer (flüchtig) betrachten. Änderungen hier sollten primär durch Prompts an Lovable erfolgen, um die Synchronisation zwischen Design und Code nicht zu brechen.
- **The Core (Engineer Managed):** Ein geschütztes Verzeichnis, z.B. /src/lib/core, enthält die API-Konnektoren, Typ-Definitionen und State-Management-Logik, die die Kommunikation mit der "Black Box" steuern. Dieses Verzeichnis muss durch GitHub Actions und .gitignore bzw. .lovableignore (sofern unterstützt) oder durch strikte Code-Ownership-Regeln geschützt werden, um zu verhindern, dass KI-generierte Commits die Sicherheitslogik verändern.[8]

Die Synchronisation zwischen Lovable und GitHub ist ein mächtiges Werkzeug, birgt aber die Gefahr, dass ungetesteter Code in den main-Branch gelangt. Daher ist die Konfiguration von Branch-Protection-Regeln in GitHub essenziell. Direkte Pushes auf main durch den Lovable-Bot sollten unterbunden werden; stattdessen sollte Lovable Pull Requests erstellen, die dann von Senior Engineers geprüft werden.

2. Row Level Security (RLS) und Multi-Tenancy Strategien

Für den 'MargenKalkulator' ist Datenisolation nicht bloß ein Feature, sondern eine fundamentale Sicherheitsanforderung. In einem Szenario, in dem mehrere Mandanten (z.B. verschiedene Vertriebsregionen oder unabhängige Klientenorganisationen) dieselbe Plattform nutzen, wäre der Zugriff eines Mandanten auf die Margenkonfigurationen eines anderen ein katastrophales Geschäftsversagen.

2.1 Das Multi-Tenancy Pattern: Diskriminator-Spalte vs. Schema-Separation

Während Supabase theoretisch die Erstellung separater Schemata für Mandanten unterstützt, ist das **"Discriminator Column"**-Pattern der Standard für dieses Projekt. Die Gründe hierfür liegen in der Kompatibilität mit dem Supabase Auth-Ökosystem, der einfacheren Wartung von Datenbankmigrationen und der besseren Skalierbarkeit bei einer hohen Anzahl von Mandanten.[9]

Standard: Jede Tabelle in der Datenbank, die mandantenspezifische Daten enthält, muss ausnahmslos eine tenant_id (oder organization_id) Spalte enthalten.

Implementierungsregel:

RLS-Richtlinien dürfen sich niemals auf vom Client bereitgestellte IDs verlassen. Ein Client könnte in einem HTTP-Request tenant_id=123 senden, obwohl er zu Tenant 456 gehört. Die Sicherheit muss aus dem authentifizierten Sitzungskontext (JWT) abgeleitet werden.

Die folgende SQL-Richtlinie demonstriert den Standard für die Isolierung:

SQL

```
-- STANDARD RLS POLICY FOR TENANT ISOLATION
CREATE POLICY "Tenant Isolation Policy"
ON public.calculations
FOR ALL
TO authenticated
USING (
  tenant_id = (select auth.jwt() -> 'app_metadata' ->> 'tenant_id')::uuid
)
WITH CHECK (
  tenant_id = (select auth.jwt() -> 'app_metadata' ->> 'tenant_id')::uuid
);
```

Diese Richtlinie stellt sicher, dass selbst bei einer Manipulation des Payloads durch einen böswilligen Benutzer die Datenbankoperation fehlschlägt oder keine Daten zurückliefert, da die tenant_id im WHERE-Clause der Query zwangsweise auf die ID aus dem JWT gesetzt wird.[11] Die Funktion auth.jwt() ist hierbei performant, da sie auf Sitzungsvariablen zugreift und keinen zusätzlichen Lookup in der auth.users Tabelle erfordert, was bei hochfrequenten Abfragen entscheidend ist.

2.2 Column-Level Security (CLS) durch Vertikale Partitionierung ("Split-Table")

Eine häufige Anforderung in Margenkalkulatoren ist, dass bestimmte Benutzer (z.B. Vertriebsmitarbeiter) zwar Produktdaten sehen dürfen (Name, SKU, Verkaufspreis), aber nicht die internen Kosten oder die berechnete Marge. PostgreSQLs native Column Level Privileges (GRANT SELECT (col1) ON table) sind für moderne Webanwendungen oft unzureichend.

Das Problem mit nativen Column Privileges ist, dass eine SELECT *-Abfrage (wie sie von vielen ORMs oder Frontend-Bibliotheken wie TanStack Query standardmäßig generiert wird) fehlschlägt, wenn der Benutzer auf auch nur eine einzige Spalte keinen Zugriff hat. Dies führt zu Laufzeitfehlern im Frontend anstatt zu einer gefilterten Datenrückgabe.[12]
Der "Split-Table"-Standard:
Für 'MargenKalkulator' müssen sensible Felder (z.B. cost_price, internal_margin_coefficient) zwingend in einer separaten Tabelle residieren, getrennt von den öffentlichen Feldern (z.B. product_name, display_price).
Die Strukturierung erfolgt nach folgendem Muster:

1. **Öffentliche Tabelle:** products (Enthält id, name, sku, public_description). RLS: Lesbar für authentifizierte Benutzer ("authenticated").
2. **Private Tabelle:** products_commercial (Enthält product_id, cost_basis, supplier_margin). RLS: Lesbar *ausschließlich* durch die Rolle service_role oder spezifische admin-Rollen, niemals durch Standard-Benutzer.

Diese architektonische Entscheidung, auch als vertikale Partitionierung bekannt, bietet einen robusten Schutzmechanismus. Visualisiert man die Datenbankstruktur, so stehen sich die öffentliche Tabelle products und die private Tabelle product_margins gegenüber, verbunden durch eine strikte 1:1-Beziehung. Während die öffentliche Tabelle metaphorisch "grün" und offen zugänglich für die Applikation ist, ist die private Tabelle "rot" und mit einem Schloss versehen. Der Zugriff auf die rote Tabelle ist für den normalen Datenverkehr gesperrt.
Dies stellt sicher, dass eine SELECT * FROM products-Abfrage durch das Lovable-Frontend nur unkritische Daten zurückliefert. Die Kalkulations-Engine (die im Backend läuft) kann mittels des sicheren Service-Role-Keys beide Tabellen joinen (zusammenführen), um die Mathematik durchzuführen, ohne jemals die Rohkostendaten über das Netzwerk an den Client zu senden.[13]

2.3 Erzwingung von 1-zu-1-Beziehungen

Bei der Verwendung des Split-Table-Patterns ist die Integrität der Datenbeziehung von höchster Wichtigkeit. Ein Produkt darf nicht ohne seine kommerziellen Daten existieren, und umgekehrt. Inkonsistenzen hier würden zu Fehlern in der Berechnung führen.
Standard:

1. Verwendung eines Foreign Key Constraints auf der id-Spalte der privaten Tabelle, der auf die öffentliche Tabelle referenziert.
2. Einsatz von DEFERRABLE INITIALLY DEFERRED Constraints, falls Einfügeoperationen in einer einzigen Transaktion, aber sequenziell erfolgen. Dies erlaubt es, innerhalb einer Transaktion kurzzeitig einen inkonsistenten Zustand zu haben, der jedoch vor dem Commit aufgelöst sein muss.
3. Idealerweise Nutzung einer Postgres-Transaktion oder einer Stored Procedure, um sicherzustellen, dass beide Zeilen atomar (gleichzeitig) erstellt werden.[15]

SQL

```
-- Standard Constraint für private Margentabellen
ALTER TABLE internal.product_margins
ADD CONSTRAINT fk_product
FOREIGN KEY (id)
REFERENCES public.products (id)
```

ON DELETE CASCADE;

Durch ON DELETE CASCADE wird sichergestellt, dass beim Löschen eines Produkts auch dessen sensible Margendaten entfernt werden, um Datenleichen ("Orphaned Rows") zu vermeiden, die ein Sicherheits- und Compliance-Risiko darstellen könnten.

3. "Black-Box"-Schutz und Logiksicherheit

Die Anforderung des Nutzers nach "Black-Box-Schutz" impliziert Maßnahmen gegen das Reverse-Engineering des Kalkulators. Dies ist der kritischste Standard im Bereich des IP-Schutzes. Wenn die Konkurrenz die Margenstruktur kennt, kann sie Preise systematisch unterbieten.

3.1 Das "Facade"-Pattern für API-Interaktion

Das direkte Exponieren von Datenbanktabellen über Supabase's REST API (PostgREST) ist für CRUD-Apps akzeptabel, aber für die Calculation Engine streng verboten. Wenn das Datenbankschema öffentlich einsehbar ist, gibt dies Hinweise auf das Datenmodell, was wiederum Rückschlüsse auf die Berechnungslogik zulässt.
Standard: Verwendung von Supabase Edge Functions als **API Facade**.
Das Frontend ruft einen generischen Endpunkt auf, z.B. POST /functions/v1/calculate-margin.

- **Input:** { "productId": "uuid", "volume": 100, "customerTier": "gold" }
- **Output:** { "finalPrice": 1250.00, "quoteId": "uuid" }

Die interne Komplexität – das Nachschlagen der Basiskosten, die Anwendung des Tier-Multiplikators, das Hinzufügen des Mengenrabatts, die Prüfung des Mindestpreises ("Floor Price") – bleibt vollständig verborgen. Die Antwort ("Response") darf *keine* Zwischenwerte wie "discount_applied" oder "base_cost" enthalten, es sei denn, dies ist explizit für die Rechnungsstellung an den Kunden erforderlich.[17] Dieses Muster entkoppelt zudem das Frontend von der Datenbankstruktur: Änderungen am Schema erfordern keine Änderungen am Frontend, solange die Facade-Schnittstelle stabil bleibt.

3.2 Obfuskation und Quellcode-Schutz

Da die Logik des 'MargenKalkulator' in Deno (TypeScript) residiert, wird der Quellcode auf der Supabase-Plattform gespeichert. Um Lecks während des Build-Prozesses oder durch Source Maps zu verhindern, sind folgende Maßnahmen erforderlich:

1. **Deaktivierung von Source Maps in der Produktion:** Es muss sichergestellt werden, dass die Build-Pipeline (falls ein Bundler wie esbuild oder Webpack vor dem Deployment auf Deno verwendet wird) alle Source Maps entfernt. Source Maps würden es einem Angreifer ermöglichen, den minifizierten Code im Browser (falls versehentlich ausgeliefert) oder bei Zugriff auf Artefakte in den Original-Quellcode zurückzuübersetzen.
2. **Isolierung von Umgebungsvariablen:** Geheimnisse, die für die Berechnung verwendet werden (z.B. GLOBAL_MARGIN_FLOOR), müssen in Supabase Secrets gespeichert werden, nicht im Code.
3. **Strikte "No-Client" Importe:** Nutzung von server-only Paketen oder strikte Trennung von .server.ts Dateien. Es muss sichergestellt werden, dass Berechnungslogik niemals in eine Komponente importiert werden kann. Falls ein Entwickler versehentlich ein Server-Modul in eine React-Komponente importiert, *muss* der Build fehlschlagen. Dies kann durch Linter-Regeln (ESLint no-restricted-imports) durchgesetzt werden.[4]

Um die Sicherheit weiter zu erhöhen, kann das Konzept der **Schema-Barrieren** angewendet werden. Hierbei werden zwei konzentrische Sicherheitszonen etabliert. Die äußere Zone ist die "Public API", die für Anfragen offensteht. Die innere Zone ist die "Interne Datenbank" (Internal Schema). Selbst wenn ein Angreifer die API kompromittiert, kann er nicht auf das interne Schema zugreifen, da dieses in der

PostgREST-Konfiguration von Supabase explizit nicht exponiert wird ("not exposed"). Der Zugriff auf die innere Zone ist ausschließlich über privilegierte Edge Functions (Service Role) möglich, die als Wächter ("Gatekeeper") fungieren.

3.3 Schutz gegen unautorisierte Workflow-Änderungen

Ein sophistizierter Angreifer (oder ein kompromittiertes Entwicklerkonto) könnte versuchen, den GitHub Actions Workflow zu manipulieren, um Secrets zu leaken oder Sicherheitschecks zu umgehen.
Standard:

- **CODEOWNERS Datei:** Das Verzeichnis .github/workflows muss strikt dem "DevSecOps"-Team oder dem Principal Architect zugeordnet sein. Änderungen an diesen Dateien erfordern zwingend deren Review.
- **Environment Protection Rules:** Das Deployment auf die Produktions-Supabase-Instanz muss eine Genehmigung durch einen designierten Administrator erfordern.
- **Checksum Verification ("Hash-Lock"):** Ein Workflow-Schritt muss die Prüfsumme der kritischen "margin-logic"-Dateien verifizieren. Wenn diese Dateien ohne eine entsprechende Aktualisierung des Checksummen-Manifests (welches von einem Admin genehmigt werden muss) geändert wurden, schlägt der Build fehl. Dies verhindert "stille" Änderungen an der Berechnungsformel, die sonst unbemerkt in die Produktion gelangen könnten.[8]

4. Serverseitige PDF-Generierung und Memory-Safety

Die Generierung von PDFs für Unternehmensangebote erfordert hohe Wiedergabetreue und Performance. Der 'MargenKalkulator' wird Dokumente generieren, die Tabellen mit Berechnungen, Haftungsausschlüsse und Branding enthalten. Dies stellt im Serverless-Kontext eine signifikante technische Herausforderung dar.

4.1 Speicherbeschränkungen in Edge Functions

Supabase Edge Functions laufen auf Deno in einer V8-Isolate-Umgebung. Das Standard-Speicherlimit ist oft restriktiv (z.B. 128MB oder 256MB in niedrigeren Tiers). Das Laden einer großen PDF-Bibliothek und des gesamten Dokuments in den Speicher führt unweigerlich zu Out Of Memory (OOM) Abstürzen, einem häufigen Fehlermodus bei der serverlosen PDF-Generierung.[20]
Ein klassisches Anti-Pattern ist der "Buffer Bloat". Ein Entwickler könnte naiv const pdfBytes = await doc.save(); schreiben. Diese Anweisung versucht, das gesamte PDF-Dokument in ein einziges Uint8Array im Javascript-Heap zu serialisieren. Wenn das generierte Angebots-PDF hochauflösende eingebettete Bilder (z.B. Produktfotos) oder umfangreiche Tabellen enthält, kann pdfBytes leicht 100MB überschreiten. Während der Serialisierung benötigt V8 oft das 2- bis 3-fache der Objektgröße an temporärem Speicher für String-Konkatenation und Puffer-Allokation. Ein 100MB PDF kann so einen 300MB Speicher-Spike verursachen und die Edge Function sofort zum Absturz bringen.
Standard:

- **Vermeidung von DOM-basierten Bibliotheken:** Bibliotheken wie jspdf oder react-pdf hängen oft von einem virtuellen DOM oder Browser-APIs ab, die in Deno schwergewichtig oder nicht existent sind.[20]
- **Verwendung von pdf-lib oder pdfkit:** Diese Bibliotheken sind leichtgewichtiger und können effizienter auf Buffern operieren. pdf-lib ist der empfohlene Standard für das Modifizieren existierender Templates (Formularausfüllung), während pdfkit besser für die Generierung von Grund auf ("from scratch") geeignet ist.[21]
- **Stream Processing:** Das PDF sollte idealerweise als Stream generiert und direkt an Supabase Storage oder die Client-Response gepiped werden, anstatt den vollen Blob im RAM aufzubauen.

4.2 Das "Template + Data" Hybrid-Modell

Um Designflexibilität (Lovable) mit Backend-Performance (Deno) zu balancieren, definieren wir einen hybriden Standard:

1. **Design-Phase:** Das PDF-Layout wird als statisches PDF-"Formular" mit benannten Feldern unter Verwendung eines Tools wie Adobe Acrobat oder einem freien Äquivalent entworfen. Dieses Template wird im Supabase Storage gespeichert.
2. **Ausführungs-Phase:**
    - Die Edge Function lädt das *leichtgewichtige* Template (Buffer).
    - Sie verwendet pdf-lib, um das Template zu laden.
    - Sie füllt die Formularfelder (z.B. Field_Price -> 1200€) und fügt dynamische Tabellen hinzu.
    - Sie "plättet" das Formular (flatten()), um es uneditierbar zu machen.
    - Sie speichert das Ergebnis im Storage und gibt eine signierte URL zurück.

Dieser Ansatz hat eine konstante Speicherkomplexität (O(1)) in Bezug auf die Designkomplexität, während die Generierung eines PDFs aus HTML/CSS mittels Puppeteer linear oder exponentiell (O(N)) mit der Komplexität wächst und extrem speicherintensiv ist (da eine Headless-Browser-Instanz benötigt wird, was in Standard Edge Functions oft unmöglich ist).[20]

4.3 Implementierungsdetail: Deno Importe und Asset-Optimierung

Deno handhabt Importe anders als Node.js. Der Standard erfordert die Verwendung von deno.json, um Abhängigkeiten sauber zu verwalten und Kompatibilitätsprobleme zu vermeiden.

Standard deno.json Konfiguration:

JSON

```
{
  "imports": {
    "pdf-lib": "https://esm.sh/pdf-lib@1.17.1",
    "supabase": "npm:@supabase/supabase-js@2"
  }
}
```

Die Verwendung von esm.sh oder npm: Spezifizierern ist obligatorisch, um die Kompatibilität mit der Edge Runtime sicherzustellen.[23]

Zusätzlich müssen Assets optimiert werden, bevor sie in das PDF eingebettet werden. Bilder *müssen* über Supabase Storage Image Transformations in der Größe angepasst und komprimiert werden, *bevor* sie von der Edge Function abgerufen werden. Die Funktion sollte niemals ein 5MB Rohbild abrufen, um es in eine 5cm Box auf dem PDF zu platzieren. Stattdessen sollte eine auf 50KB optimierte Version abgerufen werden. Zudem ist darauf zu achten, dass Referenzen auf große Assets (Bilder) sofort nach dem Einbetten freigegeben werden (z.B. durch Setzen auf null), damit der Garbage Collector (GC) diesen Speicher vor dem finalen Serialisierungsschritt zurückgewinnen kann.[5]

5. Detaillierte Engineering Standards: Implementierungsleitfaden

5.1 Projektstruktur & Trennung der Verantwortlichkeiten (Separation of Concerns)

Um die Anforderungen an "Black Box" und "Lovable Kompatibilität" zu erfüllen, muss das Repository folgende Struktur aufweisen:

- /supabase
    - /migrations (Datenbankschema & RLS Policies - Strikte SQL-Dateien)

        - /functions
            - _shared (Geteilte Typen und Utilities - **KEINE Geschäftslogik**)
            - calculate-margin (Die Black Box - Sensible Logik)
            - generate-quote-pdf (Memory-Safe PDF Engine)
- /src (React App)
    - /components (Lovable Generated UI)
    - /lib
        - /api (Facade Konnektoren)
        - /types (Öffentliche Typ-Definitionen)

Regel: Das Verzeichnis /supabase/functions/calculate-margin beinhaltet die "Kronjuwelen" des Projekts. Es sollte vom Kontext, auf den Lovable zugreifen kann, ausgeschlossen oder zumindest streng überwacht werden.

5.2 RLS Testing Standard (pgTAP)

Sicherheitsrichtlinien sind Code. Sie müssen getestet werden. Wir verwenden **pgTAP** für Datenbank-Unittests, da dies der De-Facto-Standard im PostgreSQL-Ökosystem ist.
Standard: Jede RLS-Policy muss einen korrespondierenden "positiven" Test (autorisierter Benutzer kann zugreifen) und einen "negativen" Test (unautorisierter Benutzer erhält null Zeilen oder einen Fehler) aufweisen.
Beispiel Test-Suite (tests/rls/margin_security.sql):
SQL

```
BEGIN;
SELECT plan(3);
-- 1. Setup: Erstelle einen Testbenutzer und eine Margenzeile
SELECT tests.create_supabase_user('test_user');
INSERT INTO internal.product_margins (id, cost) VALUES ('uuid...', 500);
-- 2. Test: Verifiziere, dass 'Anon' keine Daten sehen kann
SET ROLE anon;
SELECT is_empty(
  'SELECT * FROM internal.product_margins',
  'Anonyme Benutzer können keine Margendaten sehen'
);
-- 3. Test: Verifiziere, dass authentifizierte Benutzer (falscher Tenant) keine Daten sehen können
SET ROLE authenticated;
-- (Annahme: Auth-Kontext ist für einen anderen Tenant gesetzt)
SELECT is_empty(
  'SELECT * FROM internal.product_margins',
  'Benutzer anderer Tenants können keine Margendaten sehen'
);
SELECT * FROM finish();
ROLLBACK;
```

Dies stellt sicher, dass bei der Weiterentwicklung der Applikation die Sicherheitsgarantien intakt bleiben. Insbesondere bei komplexen Joins zwischen Tabellen können sich RLS-Fehler einschleichen, die durch solche Tests frühzeitig erkannt werden.[26]

5.3 CI/CD & GitHub Actions Sicherheit

Um zu verhindern, dass "Lovable" oder ein unvorsichtiger Entwickler den Build brechen oder die Sicherheit kompromittieren, wird ein strikter Workflow etabliert.
Workflow: security-audit.yml
Dieser Workflow läuft bei jedem Pull Request.

1. **Dependency Audit:** Prüft auf Schwachstellen in npm und deno Paketen.
2. **Linting:** Erzwingt "No Client Logic"-Regeln (z.B. Sicherstellung, dass keine Berechnungsfunktionen in .tsx Dateien importiert werden).
3. **RLS Verification:** Führt die pgTAP Suite gegen eine temporäre Supabase-Instanz aus.
4. **Hash Lock:** Verifiziert, dass die Prüfsumme des Verzeichnisses /supabase/functions/calculate-margin mit der genehmigten Signatur übereinstimmt. Wird Logik geändert, ohne dass das Manifest aktualisiert wurde (was Admin-Rechte erfordert), schlägt der Build fehl.[8]

6. Schlussfolgerung

Die Implementierung des 'MargenKalkulator' erfordert eine disziplinierte Abweichung von der Standard-"Radical Simplicity" der modernen Webentwicklung. Durch die Behandlung der Margenkalkulationslogik als **geschützten Microservice** (via Edge Functions) und der Datenschicht als **Zero-Trust-Tresor** (via RLS und Split Tables) stellt die Organisation sicher, dass ihr geistiges Eigentum geschützt und der Betrieb robust bleibt.
Die Integration von Lovable ist ein mächtiger Beschleuniger für die UI, muss aber innerhalb strikter architektonischer Grenzen "eingesperrt" werden. Das Frontend ist lediglich ein Betrachter; das Backend ist der Motor. Die Einhaltung der Standards zur Speichersicherheit bei der PDF-Generierung und die strikten Multi-Tenancy-Protokolle führen zu einem Enterprise-Grade-System, das in der Lage ist, sensible Finanzdaten mit der notwendigen Strenge zu verarbeiten.
Dieser Bericht dient als fundamentales Architekturdokument für das Projekt 'MargenKalkulator'. Die Einhaltung dieser Standards ist für alle am Entwicklungszyklus beteiligten Personen und automatisierten Agenten obligatorisch.