# 3 Technische Implementierungs-Spezifikation: Projekt zKalkulator

## Technische Implementierungs-Spezifikation: Projekt zKalkulator

## Executive Summary

Dieses Dokument fungiert als autoritativer technischer Leitfaden ("Builder's Guide") für die Architektur, Implementierung und Operationalisierung des Projekts **zKalkulator**. Es richtet sich an Senior Software Engineers, DevOps-Spezialisten und Systemarchitekten und definiert die Transformationsstrategie eines initialen Prototypen hin zu einer enterprise-fähigen, skalierbaren Anwendung.

Der Fokus liegt auf der Etablierung robuster **Enterprise-Standards**: strikte Typensicherheit über Systemgrenzen hinweg, vollständige Parität zwischen lokaler Entwicklungsumgebung und Produktionsinfrastruktur sowie automatisierte Pipelines für Qualitätssicherung und Deployment. Die gewählte Architektur basiert auf einem modernen **Jamstack-Ansatz**, der **Supabase** als Backend-as-a-Service (BaaS) und **React (Vite)** für das Frontend nutzt, orchestriert durch eine strenge Monorepo-Strategie. Besonders kritische Komponenten wie die PDF-Generierung und komplexe Berechnungslogik werden auf **Edge Functions** ausgelagert, um Latenz zu minimieren und geistiges Eigentum zu schützen.

Diese Spezifikation integriert Erkenntnisse aus umfangreichen technischen Recherchen und adressiert spezifische Herausforderungen wie das Speichermanagement in serverlosen Umgebungen, die Integration von Deno- und Node.js-Ökosystemen sowie die Implementierung granulare Sicherheitsmodelle mittels Row Level Security (RLS).

## 1. Systemarchitektur und Technologie-Stack

Die Architektur des zKalkulator-Systems folgt dem Paradigma einer **Event-Driven Serverless Architecture**, die speziell darauf ausgelegt ist, die Vorteile von Edge Computing mit der Stabilität relationaler Datenbanken zu verbinden.

### 1.1 Architektonische Grundprinzipien

Im Zentrum des Designs steht die Entscheidung für einen "Thin Client"-Ansatz. Während moderne Frontend-Frameworks wie React mächtig genug sind, um komplexe Logik im Browser auszuführen, verlangt der Enterprise-Kontext des zKalkulator-Projekts eine klare Trennung von Verantwortlichkeiten. Der Client dient primär der Datenerfassung, Validierung und Visualisierung. Die eigentliche Geschäftslogik – insbesondere wettbewerbskritische Berechnungsalgorithmen und ressourcenintensive Dokumentengenerierung – wird serverseitig in einer isolierten Umgebung ausgeführt. Dies schützt nicht nur den Quellcode der Berechnungen vor Client-Side-Inspection, sondern garantiert auch konsistente Ergebnisse unabhängig von der Hardwareleistung des Endgeräts.[1]

Die Datenhaltung erfolgt in **PostgreSQL**, bereitgestellt durch Supabase. Im Gegensatz zu NoSQL-Alternativen ermöglicht dies die Durchsetzung strikter Schemata und referenzieller Integrität direkt auf Datenbankebene, was für finanzmathematische oder technische Kalkulatoren essenziell ist. Der Zugriff auf diese Daten wird nicht durch einen traditionellen Applikationsserver vermittelt, sondern erfolgt über das von PostgREST generierte API-Layer, welches durch **Row Level Security (RLS)** abgesichert ist. Dies reduziert den Boilerplate-Code für Standard-CRUD-Operationen massiv und verlagert die Sicherheitslogik dorthin, wo die Daten leben.[2]

Für Logik, die über einfache CRUD-Operationen hinausgeht, kommen **Supabase Edge Functions** zum Einsatz. Diese laufen auf der Deno Runtime und werden global verteilt (CDN-Edge), was die Latenz für den Endnutzer minimiert. Die Wahl von Deno statt Node.js für diese Schicht ist strategisch: Deno bietet durch sein "Secure by Default"-Modell und die native TypeScript-Unterstützung eine robustere Basis für serverlose Funktionen, die oft externe Module laden oder komplexe Datenströme verarbeiten müssen.[1]

### 1.2 Technologie-Auswahl und Begründung

Die Selektion der Technologien erfolgte unter der Prämisse, Langlebigkeit, Wartbarkeit und Entwicklerproduktivität zu maximieren.

| Komponente | Technologie | Begründung für Enterprise-Standard |
|---|---|---|
| **Frontend Framework** | React + Vite | React ist der Industriestandard für komponentenbasierte UIs. Vite als Build-Tool ermöglicht extrem schnelle HMR (Hot Module Replacement) und optimierte Production-Builds. Zudem ist dieser Stack kompatibel mit Code-Exporten aus modernen Low-Code/No-Code Tools wie Lovable.dev, was Rapid Prototyping ermöglicht.[3] |
| **Programmiersprache** | TypeScript | Die Verwendung von TypeScript ist im gesamten Stack (Frontend & Backend) mandatorisch. Dies ermöglicht das Teilen von Typendefinitionen ("Shared Types") zwischen Datenbank, API und Client, was eine ganze Klasse von Laufzeitfehlern eliminiert. |
| **Backend / DB** | Supabase (PostgreSQL) | Bietet eine integrierte Suite aus Authentifizierung (GoTrue), Datenbank, Realtime-Subscriptions und Storage. Die PostgreSQL-Basis garantiert ACID-Konformität, während Extensions wie pg_graphql oder pg_net Erweiterbarkeit bieten.[5] |
| **Compute Runtime** | Deno (Edge Functions) | Deno startet schneller als Node.js (Cold Starts), unterstützt TypeScript nativ ohne Transpilierungsschritt zur Laufzeit und bietet Web-Standard-APIs (Request, Response, Fetch), was den Kontextwechsel für Frontend-Entwickler minimiert.[1] |
| **Testing Frameworks** | pgTAP & Vitest | pgTAP ermöglicht Unit-Tests direkt in der Datenbank (PL/pgSQL), um RLS-Policies und Trigger zu verifizieren.[7] Vitest dient als schnelles, Vite-natives Framework für Frontend- und Logik-Tests. |
| **Infrastructure as Code** | Supabase CLI / Git | Die gesamte Infrastrukturkonfiguration (Tabellen, Policies, Functions) wird in Migrationsdateien versioniert und über CI/CD-Pipelines ausgerollt, um "Configuration Drift" zu vermeiden.[8] |

**1.3 Strategie für Code-Sharing in einer hybriden Umgebung**

Eine der signifikantesten Herausforderungen bei der Entwicklung von Enterprise-Anwendungen in diesem Stack ist die Diskrepanz zwischen den Laufzeitumgebungen: Das Frontend wird für den Browser gebündelt (basierend auf Node.js-Build-Tools und NPM), während die Edge Functions in Deno laufen (basierend auf URL-Imports und modernen Web-Standards). Dies führt oft zu Code-Duplizierung, da Business-Logik oder Typen nicht einfach geteilt werden können.

Für **zKalkulator** wird daher eine **Hybrid-Monorepo-Struktur** spezifiziert. Diese Struktur muss so konzipiert sein, dass sie die Isolation der Runtimes respektiert, aber dennoch Code-Reuse erlaubt.

1. **Deno-Seite:** Deno nutzt traditionell keine package.json. Stattdessen werden Abhängigkeiten in einer deno.json oder import_map.json definiert.[9] Wir konfigurieren Deno so, dass es Module aus einem zentralen _shared-Verzeichnis importieren kann.
2. **Vite-Seite:** Vite und moderne Package Manager (wie pnpm oder yarn workspaces) erwarten eine package.json. Um Code aus dem _shared-Verzeichnis (das für Deno optimiert ist) im Frontend zu

nutzen, müssen wir sicherstellen, dass dieser Code "isomorph" ist – also keine Deno-spezifischen (z.B. Deno.readFile) oder Node-spezifischen APIs ohne Polyfills verwendet.

Die Lösung ist die Generierung von **Shared Types** als "Single Source of Truth". Supabase bietet hierfür Generatoren, die das Datenbankschema in TypeScript-Interfaces übersetzen. Diese generierten Dateien werden in das _shared-Verzeichnis gelegt. Sowohl die Deno-Funktionen (via relativem Import oder Import-Map) als auch das React-Frontend (via Alias in vite.config.ts) referenzieren diese Datei. Dies garantiert, dass eine Änderung im Datenbankschema sofort zu Kompilierfehlern im Frontend *und* Backend führt, falls der Code nicht angepasst wird – ein wesentliches Merkmal robuster Softwareentwicklung.[11]

## 2. Lokale Entwicklungsumgebung (Local Dev Parity)

Im Enterprise-Kontext ist die Parität zwischen lokaler Entwicklungsumgebung und Produktionsumgebung nicht verhandelbar. "It works on my machine" ist kein akzeptabler Status. Supabase ermöglicht durch seine CLI, den gesamten Stack – inklusive Datenbank, Auth-Service, Storage-API und Edge Functions – lokal in Docker-Containern zu emulieren.

### 2.1 Verzeichnisstruktur und Monorepo-Layout

Die Verzeichnisstruktur des Projekts muss Skalierbarkeit unterstützen und die klare Trennung der Domänen widerspiegeln. Wir definieren folgendes Layout:

```
zKalkulator-Monorepo/
├──.github/ # CI/CD Workflows und Actions Konfigurationen
├──.vscode/ # Editor-spezifische Settings (kritisch für Deno/TS Mix)
├── src/ # Frontend-Applikation (Vite + React)
│ ├── components/ # Wiederverwendbare UI-Komponenten
│ ├── hooks/ # Custom React Hooks (z.B. für Auth, Data Fetching)
│ ├── lib/ # Supabase Client Instanziierung
│ ├── services/ # Abstraktionsschicht für API-Calls
│ └── types/ # Frontend-spezifische Typenerweiterungen
├── supabase/ # Backend-Infrastruktur Definition
│ ├── functions/ # Deno Edge Functions Root
│ │ ├── _shared/ # Geteilter Code (Business Logic, DB Types, CORS Header)
│ │ ├── calculate-pdf/ # Spezifische Edge Function
│ │ │ ├── index.ts # Entry Point der Funktion
│ │ │ └── deno.json # Funktions-spezifische Konfiguration & Imports
│ │ ├── import_map.json # Globales Mapping für Deno-Module (Legacy Support)
│ │ └── deno.json # Globale Deno Config für den Workspace
│ ├── migrations/ # SQL Migrationsdateien (versioniert)
│ ├── tests/ # pgTAP Datenbank-Tests
│ ├── config.toml # Lokale Supabase Konfiguration (Ports, Auth Hooks)
│ └── seed.sql # Initiale Testdaten für lokale Entwicklung
├── package.json # Frontend Dependencies und Skripte
└── docker-compose.yml # Optionale lokale Services (z.B. Mail-Server Mock)
```

Diese Struktur isoliert den Deno-Code im supabase/functions-Verzeichnis, was für die Konfiguration der IDE essenziell ist.[12]

### 2.2 Setup der Deno-Umgebung für VS Code

Ein häufiges Problem in Monorepos ist der Konflikt zwischen dem TypeScript-Server (ts-server), der standardmäßig für React-Projekte verwendet wird, und dem Deno Language Server (LSP). Wenn man eine Datei im functions-Ordner öffnet, versucht VS Code oft, diese mit den Node.js-Regeln des Frontends zu validieren, was zu Fehlern bei Importen wie import... from "https://..." führt.[14]

Lösung: Workspace-Konfiguration.
Es muss zwingend eine .vscode/settings.json erstellt werden, die den Deno-LSP explizit nur für das Backend-Verzeichnis aktiviert. Dies verhindert Interferenzen.15
JSON

```
{
  "deno.enable": true,
  "deno.enablePaths": ["./supabase/functions"],
  "deno.unstable": ["bare-node-builtins", "byonm"],
  "deno.importMap": "./supabase/functions/import_map.json",
  "deno.lint": true,
  "editor.formatOnSave": true,
  "[typescript]": {
    "editor.defaultFormatter": "denoland.vscode-deno"
  }
}
```

Zusätzlich sollte für das Frontend-Verzeichnis (src) sichergestellt werden, dass der Deno-LSP dort *nicht* aktiv ist. Dies geschieht implizit durch deno.enablePaths, aber explizite Excludes können bei komplexen Setups helfen.

**2.3 Shared Types Management und Synchronisation**

Um die Typensicherheit zu gewährleisten, wird ein automatisierter Prozess etabliert. Supabase generiert TypeScript-Definitionen basierend auf dem aktuellen Datenbankschema (Introspection).

**Workflow:**

1. **Generierung:** Der Befehl supabase gen types typescript --local > supabase/functions/_shared/database.types.ts extrahiert die Typen aus der lokal laufenden Datenbank.[11]
2. **Backend-Nutzung:** Edge Functions importieren diese Datei direkt. Da sie im _shared-Ordner liegt, ist sie über relative Pfade oder über das Mapping in deno.json erreichbar.
   JSON

   ```
   // supabase/functions/deno.json
   {
     "imports": {
       "@shared/": "../_shared/"
     }
   }
   ```

9. **Frontend-Nutzung:** Da Vite standardmäßig keinen Zugriff auf Dateien außerhalb des src-Roots erlaubt (aus Sicherheitsgründen), gibt es zwei Strategien. Für Enterprise-Setups empfehlen wir die Synchronisation mittels eines npm-Skripts (sync-types), das die Datei nach der Generierung physisch nach src/types/supabase.ts kopiert. Dies entkoppelt den Build-Prozess des Frontends von der Backend-Struktur und vermeidet Symlink-Probleme in CI-Pipelines.

## 3. Datenbank-Design und Sicherheitsmodell (RLS)

Die Datenbank ist im zKalkulator-Projekt nicht nur Datenspeicher, sondern eine aktive Komponente der Sicherheitsarchitektur. Das Prinzip "Security in Depth" wird durch die konsequente Anwendung von Row Level Security (RLS) umgesetzt.

**3.1 Row Level Security (RLS) Strategie**

RLS ist ein Feature von PostgreSQL, das es erlaubt, Zugriffskontrollregeln direkt an Tabellenzeilen zu binden. Jede SQL-Abfrage wird gegen diese Regeln geprüft. Dies bedeutet, dass selbst wenn ein Entwickler im Frontend vergisst, einen WHERE user_id =...-Filter zu setzen, die Datenbank nur die erlaubten Datensätze zurückliefert.[2]

**Implementierungsvorgabe:**

- **Default Deny:** RLS muss für **alle** Tabellen im public-Schema aktiviert werden (ALTER TABLE... ENABLE ROW LEVEL SECURITY). Ohne explizite Policy ist der Zugriff standardmäßig blockiert.
- **Rollen-Modell:** Wir unterscheiden zwischen den Rollen anon (nicht authentifiziert), authenticated (eingeloggt) und service_role (interner Admin/Edge Function).

Performance-Kritische Entscheidung: Claims vs. Joins

Ein häufiger Performance-Engpass bei RLS in Multi-Tenant-Systemen (wie zKalkulator, wenn er von mehreren Firmen genutzt wird) ist die Prüfung von Zugehörigkeiten. Eine Policy wie auth.uid() IN (SELECT user_id FROM team_members...) erzwingt für jede Zeile eine Sub-Query (Join). Bei großen Datenmengen führt dies zu einem linearen Performance-Verlust.

Die Analyse der Datenbank-Performance zeigt deutlich:

- **Sub-Query Policies (Joins):** Die Abfragezeit steigt proportional zur Anzahl der Datensätze und zur Komplexität der verknüpften Tabellen. Bei 100.000 Zeilen kann eine einfache Select-Abfrage bereits Hunderte von Millisekunden dauern.
- **Claim-based Policies:** Wenn die Informationen (z.B. tenant_id oder role) direkt im JWT (JSON Web Token) des Benutzers als "Custom Claims" gespeichert sind, ist der Zugriff O(1) – also konstant schnell, unabhängig von der Datenmenge. Die Policy lautet dann einfach tenant_id = (auth.jwt() ->> 'tenant_id')::uuid.[16]

Daher ist für zKalkulator die Nutzung von **Custom Claims** via Auth Hooks (siehe Abschnitt 6.2) vorgeschrieben, um die Skalierbarkeit zu sichern.

**3.2 Atomare Transaktionen via RPC**

Komplexe Kalkulationen erfordern oft das Speichern von Daten in mehreren Tabellen gleichzeitig (z.B. calculations Header und calculation_items Details). Der Supabase-Client führt standardmäßig REST-Calls aus, die stateless sind. Ein Client-seitiges Promise.all von zwei Inserts ist **keine** Transaktion – schlägt der zweite Insert fehl, bleibt der erste bestehen, was zu inkonsistenten Daten ("Zombies") führt.[18]

Lösung: PostgreSQL Stored Procedures (RPC).

Wir verlagern die Transaktionslogik in die Datenbank. Eine PL/pgSQL-Funktion kapselt die Operationen in einem atomaren Block.

SQL

```
create or replace function save_full_calculation(
  p_user_id uuid,
  p_title text,
  p_items jsonb
) returns jsonb
language plpgsql
```

```
security definer -- Läuft mit Rechten des Erstellers (Vorsicht geboten!)
set search_path = public -- Security Definer Best Practice: Expliziter Pfad
as $$
declare
  v_calc_id bigint;
  v_item jsonb;
begin
  -- 1. Insert Header
  insert into calculations (user_id, title)
  values (p_user_id, p_title)
  returning id into v_calc_id;
  -- 2. Insert Items (Loop über JSON Array)
  foreach v_item in array p_items
  loop
    insert into calculation_items (calculation_id, name, value)
    values (v_calc_id, v_item->>'name', (v_item->>'value')::numeric);
  end loop;
  return jsonb_build_object('id', v_calc_id, 'status', 'success');
exception when others then
  -- Automatischer Rollback bei Fehler
  raise exception 'Transaction failed: %', sqlerrm;
end;
$$;
```

Der Einsatz von security definer ist hier oft notwendig, um Schreibrechte auf Tabellen zu gewähren, die der User nicht direkt editieren darf, oder um komplexe RLS-Checks im Trigger-Kontext zu vereinfachen. Dies erfordert jedoch zwingend das Setzen des search_path, um "Search Path Hijacking" zu verhindern, sowie strikte GRANT/REVOKE-Statements, um den Zugriff auf die Funktion auf authentifizierte Nutzer zu beschränken.[19]

## 4. Backend-Implementierung (Edge Functions)

Für Berechnungen, die sensible Algorithmen enthalten oder externe Bibliotheken benötigen, nutzen wir Supabase Edge Functions. Diese bieten eine isolierte Laufzeitumgebung basierend auf Deno.

### 4.1 PDF-Generierung in der Edge

Eine der anspruchsvollsten Anforderungen für zKalkulator ist die Generierung von PDF-Reports. In serverlosen Umgebungen sind Speicher (RAM) und Ausführungszeit (CPU Wall Time) begrenzt. Standard-Pläne bieten oft nur 128MB bis 256MB RAM. Das Laden eines großen PDF-Dokuments in den Speicher kann zum sofortigen Absturz ("OOM Kill") führen.[21]

Bibliotheks-Wahl und Evaluation:

Die Recherche zeigt, dass viele populäre Node.js-Bibliotheken (wie puppeteer oder pdfkit) in der Deno-Edge-Umgebung problematisch sind, da sie oft auf Node-APIs (wie fs oder child_process) angewiesen sind, die in der Sandbox nicht oder nur eingeschränkt zur Verfügung stehen.

- **Empfehlung: pdf-lib**. Diese Bibliothek ist in reinem JavaScript/TypeScript geschrieben, hat keine nativen Abhängigkeiten und ist offiziell kompatibel mit Deno. Sie erlaubt das Modifizieren existierender PDFs sowie das Erstellen neuer.[23]
- **Alternative:** react-pdf ist beliebt im Frontend, aber im Backend schwierig, da es oft DOM-APIs erwartet. Zwar gibt es Ansätze via esm.sh, die Stabilität in Edge Functions ist jedoch oft mangelhaft.[25]

Strategie für Speichereffizienz: Streaming und Offloading
Um Speicherüberläufe zu vermeiden, darf das generierte PDF niemals vollständig als Base64-String im RAM gehalten oder an den Client zurückgeschickt werden (Base64 vergrößert die Daten um ca. 33%).

1. **Direct Upload:** Das PDF wird generiert (am besten unter Verwendung von Uint8Array statt Strings) und sofort per Stream in einen Supabase Storage Bucket hochgeladen.
2. **Signed URL:** Die Funktion gibt lediglich eine temporär gültige URL (signedUrl) zurück, über die der Client das PDF herunterladen kann.

**Implementierungs-Beispiel (Konzept):**
TypeScript

```
// supabase/functions/generate-pdf/index.ts
import { PDFDocument } from 'https://cdn.skypack.dev/pdf-lib';
import { createClient } from 'jsr:@supabase/supabase-js@2';
Deno.serve(async (req) => {
  // 1. Setup Client (mit Service Role für Storage Schreibrechte)
  const supabase = createClient(Deno.env.get('SUPABASE_URL')!,
Deno.env.get('SUPABASE_SERVICE_ROLE_KEY')!);
  // 2. Load Template (gestreamt oder gecached wenn möglich)
  // Für kleine Templates ist ArrayBuffer okay, für große Streams nötig.
  const formPdfBytes = await fetch(TEMPLATE_URL).then(res => res.arrayBuffer());
  const pdfDoc = await PDFDocument.load(formPdfBytes);

  // 3. Modifikation
  const form = pdfDoc.getForm();
  form.getTextField('CustomerName').setText('Max Mustermann');
  form.flatten(); // Formularfelder "einbacken"
  // 4. Save & Upload (Direkter Stream zu Storage wäre ideal, pdf-lib save() liefert Uint8Array)
  const pdfBytes = await pdfDoc.save();

  const fileName = `report_${crypto.randomUUID()}.pdf`;
  const { error: uploadError } = await supabase.storage
   .from('reports')
   .upload(fileName, pdfBytes, { contentType: 'application/pdf' });
  if (uploadError) throw uploadError;
  // 5. Generate URL
  const { data } = await supabase.storage
   .from('reports')
   .createSignedUrl(fileName, 60 * 5); // 5 Minuten gültig
  return new Response(JSON.stringify({ url: data?.signedUrl }), { headers: { "Content-Type":
"application/json" } });
});
```

Diese Architektur entkoppelt die Generierung vom Download und schont die Ressourcen des Clients und der Edge Function.[27]

## 5. Frontend-Implementierung (React & Vite)

### 5.1 Integration von Lovable und Vite

Tools wie Lovable.dev beschleunigen den initialen Aufbau der UI enorm, indem sie aus Beschreibungen React-Code generieren. Der Export ist typischerweise ein Standard-Vite-Projekt. Für die Enterprise-Integration müssen jedoch einige Anpassungen vorgenommen werden.[3]
**Härtung des Codes:**

- **Environment Variables:** Hardcodierte API-URLs im generierten Code müssen durch import.meta.env.VITE_SUPABASE_URL ersetzt werden.
- **Routing:** Lovable nutzt oft den react-router. Im Enterprise-Kontext sollten Routen durch Konstanten definiert und lazy-loaded werden (React.lazy), um die initiale Bundle-Größe zu minimieren.
- **Service Layer:** Generierter Code macht oft direkte API-Calls in Komponenten (useEffect). Dies muss refactored werden. Wir führen einen expliziten Service-Layer ein (z.B. src/services/calculationService.ts), der die Kommunikation mit Supabase kapselt. Dies erleichtert das Mocking in Tests und zentralisiert die Fehlerbehandlung.

### 5.2 Zustandsverwaltung und API-Integration
Für zKalkulator ist der lokale Zustand (während der Eingabe) und der Server-Zustand (gespeicherte Kalkulationen) zu unterscheiden.

- **Server State:** Nutzung von TanStack Query (früher React Query). Dies bietet Caching, De-duplication von Requests und automatische Revalidation, was für ein flüssiges Nutzererlebnis essenziell ist.
- **Local State:** Für komplexe Formulare empfiehlt sich react-hook-form in Kombination mit zod zur Schema-Validierung. Da wir TypeScript nutzen, können wir die zod-Schemas oft direkt aus den generierten Supabase-Typen ableiten, was Redundanz vermeidet.

## 6. Advanced Features: Auth Hooks und Multi-Tenancy
Im Enterprise-Umfeld reichen einfache Rollen wie "User" oft nicht aus. Es werden differenzierte Berechtigungen benötigt (RBAC - Role Based Access Control) oder Mandantenfähigkeit (Multi-Tenancy).
### 6.1 Custom Claims via Auth Hooks
Supabase ermöglicht es, den Authentifizierungsprozess mittels **Auth Hooks** zu erweitern. Dies sind PostgreSQL-Funktionen oder HTTP-Endpunkte, die *während* des Token-Issuance-Prozesses ausgeführt werden.
Für zKalkulator implementieren wir einen custom_access_token Hook. Wenn sich ein User einloggt, prüft dieser Hook in der Tabelle user_roles oder memberships die Berechtigungen des Users und schreibt diese direkt in das Access Token (JWT).[19]
**Konfiguration (supabase/config.toml):**
Ini, TOML
```
[auth.hook.custom_access_token]
enabled = true
uri = "pg-functions://postgres/public/custom_claims_hook"
```
**SQL Hook Implementierung:**
SQL
```
create or replace function public.custom_claims_hook(event jsonb)
returns jsonb language plpgsql stable as $$
declare
  claims jsonb;
  user_role text;
begin
```

```
  -- Rolle abrufen
  select role into user_role from public.user_roles where user_id = (event->>'user_id')::uuid;

  claims := event->'claims';

  -- Rolle ins JWT injizieren
  if user_role is not null then
    claims := jsonb_set(claims, '{user_role}', to_jsonb(user_role));
  end if;
  -- Event aktualisieren
  event := jsonb_set(event, '{claims}', claims);
  return event;
end;
$$;
```

Notwendige Berechtigungen: Der Hook läuft unter dem User supabase_auth_admin. Wir müssen diesem User explizit Rechte geben (GRANT EXECUTE) und den Zugriff für public entziehen (REVOKE), um Missbrauch zu verhindern.[19]

## 7. Qualitätssicherung: Testing Strategy

Ein Enterprise-Projekt definiert sich über seine Stabilität. Wir etablieren eine Test-Pyramide, die alle Schichten abdeckt.

### 7.1 Datenbank-Unit-Tests (pgTAP)

Logik, die in der Datenbank lebt (RLS, Trigger, RPCs), wird oft vernachlässigt. Mit pgTAP testen wir diese deklarativ.[7]

Testszenario: RLS-Validierung

Wir simulieren verschiedene User-Kontexte, um sicherzustellen, dass Datenlecks unmöglich sind.

SQL

```
-- Datei: supabase/tests/database/01_rls_security.sql
begin;
select plan(3); -- Wir erwarten 3 Tests
-- 1. Setup: Testdaten erstellen (Switch to Admin Role)
set local role service_role;
insert into calculations (id, user_id, title) values (100, 'uuid-user-a', 'Secret Calc A');
-- 2. Testfall: User A darf sein Dokument sehen
select tests.authenticate_as('uuid-user-a');
select results_eq(
  $$select title from calculations where id = 100$$,
  $$values ('Secret Calc A')$$,
  'Owner can see their calculation'
);
-- 3. Testfall: User B darf Dokument von A NICHT sehen
select tests.authenticate_as('uuid-user-b');
select is_empty(
  $$select * from calculations where id = 100$$,
  'Other user cannot see calculation'
);
select * from finish();
rollback;
```

Dieser Test läuft in einer Transaktion (begin... rollback), sodass die Testdatenbank nach jedem Lauf sauber bleibt.

### 7.2 Frontend und Integrationstests

- **Unit Tests (Vitest):** Testen isolierter Logik, z.B. mathematische Hilfsfunktionen im zKalkulator.
- **Component Tests (React Testing Library):** Prüfen, ob Komponenten korrekt rendern und auf Events reagieren.
- **End-to-End Tests (Playwright):** Simulieren einen echten Browser-Workflow: Login -> Kalkulation erstellen -> PDF generieren -> Logout. Diese Tests laufen gegen eine Staging-Umgebung.

## 8. Deployment & CI/CD Pipeline

Die Auslieferung erfolgt vollautomatisiert über GitHub Actions. Manuelle Deployments sind Fehlerquellen und daher im Produktionsumfeld untersagt.

### 8.1 Pipeline-Architektur

Die Pipeline (.github/workflows/ci.yml) ist so konzipiert, dass sie schnelles Feedback liefert ("Fail Fast").

1. **Change Detection:** Wir nutzen Actions wie dorny/paths-filter oder Hash-Vergleiche [31], um zu erkennen, ob sich Backend-Code (supabase/) oder Frontend-Code (src/) geändert hat. Unveränderte Teile werden nicht neu gebaut/getestet, was CI-Minuten spart.
2. **Supabase CLI Setup:** Die Action supabase/setup-cli installiert die CLI im Runner.
3. **Integration Testing:** Ein temporärer Supabase-Stack wird im CI-Runner gestartet (supabase start). Die pgTAP-Tests laufen gegen diese Instanz. Dies ist sicherer als Tests gegen eine externe Staging-DB, da die Umgebung komplett isoliert und ephemeral ist.
4. **Deployment:**
    - **Edge Functions:** supabase functions deploy
    - **Datenbank:** supabase db push (wendet Migrationen an)
    - **Frontend:** Build via npm run build, Upload zu Hosting-Provider (z.B. Vercel, Netlify oder S3).

### 8.2 Environment Management

Secrets (API Keys, DB Passwörter) werden niemals im Code hinterlegt. Sie werden in GitHub Secrets gespeichert und zur Laufzeit als Environment Variables injiziert. Die Supabase CLI unterstützt das Verlinken von Projekten (supabase link), sodass beim Deployment automatisch die korrekte Remote-Instanz (Staging oder Production) basierend auf dem Git-Branch (main vs. develop) gewählt werden kann.

## 9. Sicherheits-Audit und Compliance

Abschließend wird das System gegen gängige Angriffsvektoren gehärtet.

1. **RLS-Lücken:** Regelmäßige Prüfung, ob neue Tabellen RLS aktiviert haben. Ein CI-Skript kann dies prüfen (select count(*) from pg_tables where rowsecurity = false).
2. **Function Isolation:** Sicherstellen, dass security definer Funktionen minimalen Scope haben und der search_path gesetzt ist.
3. **Dependency Scanning:** Tools wie npm audit und deno audit (falls verfügbar) oder Dependabot überwachen die Lieferkette auf verwundbare Pakete.

## Fazit

Die hier spezifizierte Architektur transformiert **zKalkulator** von einem einfachen Tool zu einer robusten Plattform. Durch die Kombination von **Supabase** als skalierbares Backend, **Deno Edge Functions** für Hochleistungs-Logik und einer strikten **Monorepo-Struktur** mit **automatisierter QA** wird ein Fundament geschaffen, das zukünftiges Wachstum und komplexe Enterprise-Anforderungen problemlos trägt. Der initiale Aufwand für das Setup der lokalen Parität und der Test-Pipelines wird sich durch reduzierte Wartungskosten und erhöhte Ausfallsicherheit vielfach amortisieren.