5 Operational Workflow & Hybrid-Team Topology: Projekt zKalkulator

Operational Workflow & Hybrid-Team Topology: Projekt zKalkulator

Executive Summary

Das Projekt **zKalkulator** repräsentiert eine neue Klasse von Softwarearchitekturen, die wir als **AI-Augmented Hybrid Systems** definieren. Die Kombination aus **Supabase** (als Backend-as-a-Service und Edge-Computing-Plattform) und **Lovable** (als AI-getriebener Frontend-Generator) bietet enorme Geschwindigkeitsvorteile, birgt jedoch signifikante Risiken hinsichtlich der Code-Qualität, Wartbarkeit und der Integrität der Schnittstellen. In einer Ära, in der generative KI-Tools wie Lovable den Schritt vom bloßen "Copiloten" zum eigenständigen "Agenten" vollziehen, verschieben sich die Verantwortlichkeiten im Entwicklungsteam fundamental. Der menschliche Entwickler wandelt sich vom reinen Coder zum Systemarchitekten und Orchestrator, der die Leitplanken für die probabilistische Natur der KI definiert.
Dieser Bericht analysiert die spezifischen Reibungspunkte zwischen der manuellen, deterministischen Backend-Entwicklung (Deno/TypeScript) und der probabilistischen, AI-generierten Frontend-Entwicklung (React/Vite). Er liefert einen erschöpfenden **Operational Workflow**, der auf drei Säulen ruht: **Contract-First-Validierung** zur Sicherung der API-Integrität, **Code-Isolation** zum Schutz manueller Logik vor AI-Überschreibung und **DevEx-Optimierung** durch lokale Simulation und Daten-Anonymisierung. Ziel dieses Dokuments ist es, eine Topologie zu definieren, in der der menschliche Entwickler als Architekt und "Guardrail" fungiert, während die AI (Lovable) als hocheffizienter, aber streng geführter Implementierungspartner agiert. Wir werden detaillierte Verzeichnisstrukturen, Konfigurationsstrategien (OpenAPI, Zod, Mock Service Worker) und Sicherheitsmechanismen (RLS, Anonymisierung) vorstellen, um das Projekt zKalkulator skalierbar und robust zu gestalten.

1. Einführung: Der Paradigmenwechsel zu AI-Augmented Hybrid Systems

Die Softwareentwicklung steht an einem historischen Wendepunkt. Während frühere Iterationen der Entwicklerproduktivität durch Abstraktion (höhere Programmiersprachen, Frameworks, BaaS) getrieben wurden, wird der aktuelle Zyklus durch die Augmentierung kognitiver Arbeit mittels Generativer KI definiert. Plattformen wie Lovable.dev versprechen, den Frontend-Entwicklungsprozess durch "Natural Language to Code" (Vibe Coding) drastisch zu beschleunigen. Dies führt jedoch zu einer fundamentalen Asymmetrie im Technologiestack des Projekts zKalkulator. Auf der einen Seite steht ein hochgradig strukturiertes, typensicheres und sicherheitskritisches Backend, basierend auf Supabase (PostgreSQL) und Deno Edge Functions. Auf der anderen Seite operiert ein fluides, AI-generiertes Frontend, das auf React und Vite basiert und dessen Quellcode durch probabilistische LLM-Inferenzen entsteht.
Diese Asymmetrie erfordert ein Umdenken in der Projektorganisation. Ein traditioneller Workflow, bei dem Frontend und Backend parallel und in ständiger manueller Abstimmung entwickelt werden, scheitert hier an der Volatilität des AI-Outputs. Wenn die AI das Frontend refakturiert, ohne die impliziten Verträge der API zu kennen, entstehen subtile Regressionsfehler. Wenn das Backend seine Schnittstelle ändert, ohne dass die AI dies kontextuell erfasst, bricht die Anwendung. Daher ist die Etablierung einer rigiden **Hybrid-Team Topology** kein optionales Add-on, sondern die conditio sine qua non für den Projekterfolg.
Die Herausforderung besteht nicht nur in der technischen Integration zweier unterschiedlicher Runtimes (Deno vs. Node.js), sondern in der kognitiven Synchronisation zwischen einem deterministischen menschlichen Akteur und einem stochastischen künstlichen Akteur. Dieser Bericht widmet sich der Lösung dieses Konflikts durch technische Exzellenz und prozessuale Strenge.

2. Hybrid-Team Topology: Rollenverteilung und Kognitive Architektur

Die Einführung von AI-Coding-Tools wie Lovable verändert die klassische Teamstruktur fundamental. Anstelle der traditionellen Aufteilung in Frontend- und Backend-Entwickler, erfordert zKalkulator eine

Hybrid-Topology. In diesem Modell ist Lovable nicht nur ein Werkzeug, sondern ein aktiver "Agent" im Team, dessen Output jedoch kontinuierlicher Validierung bedarf.[1]

2.1. Die "Architect-Agent" Beziehung: Neudefinition von Verantwortlichkeit

Der menschliche Entwickler nimmt im zKalkulator-Projekt primär die Rolle des **Systemarchitekten und Backend-Guardians** ein. Die AI (Lovable) fungiert als **Frontend-Implementierer**. Diese Trennung ist notwendig, da AI-Modelle dazu neigen, Kontext zu verlieren ("Drifting") oder Sicherheitsbestimmungen zu halluzinieren, wenn sie nicht strikt geführt werden.[3] Es ist eine hierarchische Beziehung, keine partnerschaftliche auf Augenhöhe. Der Architekt definiert die unveränderlichen Wahrheiten des Systems – das Datenmodell, die Sicherheitsregeln (Row Level Security), die API-Schnittstellen und die Validierungslogik. Der Agent (Lovable) operiert innerhalb dieser definierten Grenzen, um die Benutzeroberfläche zu generieren, die diese Wahrheiten für den Endnutzer zugänglich macht.

Die Kernherausforderung besteht darin, dass Lovable ("Vibe Coding") dazu neigen kann, schnelle visuelle Ergebnisse zu priorisieren, oft auf Kosten von Typensicherheit oder effizienter API-Nutzung.[5] Ein typisches Fehlermuster ist, dass die AI versucht, komplexe Geschäftslogik im Frontend nachzubauen ("Client-Side Calculation"), anstatt die dafür vorgesehenen, sicheren Edge Functions aufzurufen. Dies führt nicht nur zu Redundanz, sondern öffnet Sicherheitslücken, da Client-Code manipulierbar ist.[7] Daher muss die Topologie erzwingen, dass **Backend-Regeln (Contracts) unveränderlich** sind und vom Frontend konsumiert werden müssen, ohne dass die AI diese Contracts neu erfindet.

2.2. Reibungspunkte der Topologie: Eine forensische Analyse

Die Analyse der verfügbaren Daten [8] identifiziert drei kritische Reibungszonen in dieser Topologie, die das Projekt ohne proaktives Management zum Scheitern bringen können. Diese Reibungspunkte sind nicht nur technischer Natur, sondern auch prozessual bedingt durch die unterschiedlichen Arbeitsgeschwindigkeiten von Mensch und Maschine.

Erstens besteht ein signifikanter **Konflikt der Runtimes** zwischen Deno und Node.js. Supabase Edge Functions nutzen die Deno-Runtime, die auf Web-Standards (Import Maps, URL-Imports) basiert und ein striktes Sicherheitsmodell verfolgt. Das React-Frontend, welches von Lovable generiert wird und auf Vite basiert, ist hingegen tief im Node.js-Ökosystem (NPM, package.json, CommonJS-Kompatibilität) verankert. Das Teilen von TypeScript-Typen und Validierungslogik (z.B. Zod-Schemas) ist ohne eine sorgfältig geplante Monorepo-Struktur extrem fehleranfällig, da Import-Maps (Deno) und package.json (Node) standardmäßig inkompatibel sind.[11] Wenn VS Code versucht, IntelliSense für beide Welten gleichzeitig bereitzustellen, führt dies oft zu Konflikten im Language Server Protocol (LSP), was die Developer Experience massiv beeinträchtigt.[13]

Zweitens leiden LLM-basierte Systeme unter **Kontext-Amnesie**. Lovable "vergisst" bei langen Chat-Verläufen oder umfangreichen Projekten oft vorherige Instruktionen. Es ignoriert existierende Utility-Funktionen und beginnt, Code zu duplizieren, anstatt ihn wiederzuverwenden.[3] Dies führt zu einer Fragmentierung der Codebasis, bei der Geschäftslogik an mehreren Stellen inkonsistent implementiert wird. Ohne eine "Single Source of Truth", die immer wieder in den Kontext geladen wird, driftet die AI ab.

Drittens neigt die AI zu **destruktiven Iterationen**. Da Lovable darauf trainiert ist, Nutzeranfragen direkt zu erfüllen ("Make the button blue"), übersieht es oft die systemischen Auswirkungen. Ohne Schutzmechanismen neigt die AI dazu, komplexe, manuell optimierte Logikkomponenten oder sorgfältig abgestimmte Typendefinitionen zu überschreiben, um eine triviale UI-Änderung durchzuführen.[7] Der menschliche Architekt muss daher Mechanismen etablieren, die bestimmte Code-Bereiche als "unantastbar" markieren.

3. Technische Reibungsanalyse: Deno vs. Node im Detail

Um die vorgeschlagene Lösung zu verstehen, ist eine detaillierte Betrachtung des Laufzeit-Konflikts notwendig. Supabase setzt für Edge Functions auf Deno, um Kaltstartzeiten zu minimieren und Sicherheit durch Isolation zu maximieren. Deno löst Abhängigkeiten über URLs auf (z.B. import { z } from

"https://deno.land/x/zod/mod.ts"). Node.js und das npm-Ökosystem, in dem React und Vite leben, lösen Abhängigkeiten über den node_modules-Ordner und die package.json auf (z.B. import { z } from "zod"). Dieser Unterschied führt zu einem fundamentalen Problem beim Code-Sharing: Eine Datei schema.ts, die zod importiert, kann nicht gleichzeitig von Deno (Backend) und Node (Frontend) konsumiert werden, wenn sie nicht speziell dafür vorbereitet ist.

- **Im Backend:** Deno erwartet entweder eine URL oder einen Eintrag in der Import Map.
- **Im Frontend:** Der Vite-Bundler erwartet ein npm-Paket in node_modules.

Naive Lösungsversuche, wie das Kopieren von Dateien, verletzen das DRY-Prinzip (Don't Repeat Yourself) und führen zu Inkonsistenzen. Komplexere Lösungen, wie das Publishing eines privaten npm-Pakets, erhöhen den CI/CD-Overhead drastisch. Die Lösung für zKalkulator muss daher "Zero-Copy" und "Zero-Publish" sein, was durch geschickte Konfiguration von Workspaces und Pfad-Aliasen erreicht wird.[9]
Ein weiteres Problem ist die IDE-Unterstützung. Der Deno-LSP für VS Code versucht, alle TypeScript-Dateien zu validieren. In einem Frontend-Ordner scheitert er jedoch an Node-spezifischen Konstrukten oder fehlenden Deno-Imports, was zu "roten Wellenlinien" im gesamten Editor führt. Umgekehrt versteht der Standard-TypeScript-Server (TSServer) die Deno-Import-Pfade nicht. Eine strikte Isolation auf Workspace-Ebene in VS Code ist daher zwingend erforderlich.[14]

4. Contract-First-Validierung: Die Single Source of Truth

Um die Diskrepanz zwischen manuellem Backend und AI-Frontend zu überbrücken, muss zKalkulator eine strikte **Contract-First-Strategie** verfolgen. Die AI darf niemals die API *raten*. Sie muss gegen einen harten Vertrag entwickeln, der maschinenlesbar ist und dessen Einhaltung erzwungen wird.

4.1. Zod als Nukleus der Wahrheit

Der Schlüssel zur Lösung des Deno/Node-Konflikts und zur Sicherstellung der Konsistenz liegt in der Verwendung von **Zod** als sprachenübergreifende Validierungsbibliothek. Zod hat den entscheidenden Vorteil, dass es sowohl TypeScript-Typen (statisch) als auch Validierungslogik (Laufzeit) aus einer einzigen Definition generiert.
Für zKalkulator wird folgende **Monorepo-Strategie** empfohlen, um Zod-Schemas zu teilen:

1. **Isolierter Package-Ordner:** Erstellung eines Ordners packages/shared-types. Dieser Ordner enthält *nur* TypeScript-Dateien, die Zod-Schemas exportieren. Er enthält *keine* laufzeitspezifischen Abhängigkeiten außer Zod selbst.
2. **Isomorpher Import:**
    - **Frontend (Node/Vite):** In der package.json des Frontends wird Zod als normale Dependency installiert. Der packages/shared-types Ordner wird über Vite-Aliase (@shared) eingebunden.
    - **Backend (Deno):** In der deno.json des Backends wird ein Import-Map-Eintrag erstellt, der zod auf die Deno-kompatible Version (z.B. via esm.sh oder deno.land) mappt. Ein weiterer Eintrag mappt @shared/ auf den relativen Pfad ../../packages/shared-types/src/.

Dadurch können beide Welten dieselbe physische Datei schema.ts lesen. Deno löst den Import import { z } from "zod" über seine Import Map auf die URL auf, während Vite ihn über node_modules auflöst. Dies ist der "heilige Gral" des Code-Sharing in hybriden Runtimes.[15]

4.2. OpenAPI als Vertrag für die AI

Während Zod die interne Konsistenz sichert, benötigt Lovable (die AI) eine verständliche Beschreibung der API. Hier kommt **OpenAPI (Swagger)** ins Spiel. Supabase generiert automatisch eine

OpenAPI-Spezifikation für die Datenbank (PostgREST) [16], aber Edge Functions bleiben oft undokumentiert.
Wir empfehlen für zKalkulator den Einsatz von Frameworks innerhalb der Edge Functions, die OpenAPI unterstützen. Speziell Hono (ein Web-Framework für Edge) bietet in Kombination mit @hono/zod-openapi eine erstklassige Lösung.17
Der Workflow sieht wie folgt aus:

1. **Definition:** Der Backend-Entwickler definiert Input/Output-Schemas mit Zod im Shared-Package.
2. **Implementation:** Die Edge Function nutzt Hono und registriert Routen unter Verwendung dieser Schemas.
3. **Generierung:** Hono generiert zur Laufzeit (oder Build-Zeit) eine OpenAPI v3.1 JSON-Spezifikation (/doc Endpoint).
4. **Konsum:** Ein Skript (openapi-typescript oder openapi-fetch) generiert daraus TypeScript-Clients für das Frontend.[19]
5. **Instruktion:** Lovable wird angewiesen, diesen generierten Client zu nutzen.

Dieser Ansatz eliminiert die Möglichkeit, dass die AI falsche Endpunkte oder Parameter "halluziniert", da der Client-Code streng typisiert ist und direkt aus der Backend-Definition stammt.

5. Code-Isolation und Projektstruktur: Die Festung bauen

Die physische Struktur des Projekts ist der wichtigste Mechanismus zur Durchsetzung der Topologie. Wir müssen sicherstellen, dass Lovable niemals versehentlich Backend-Code modifiziert oder Konfigurationsdateien zerstört. Dies erfordert eine präzise Konfiguration von Dateisystem-Berechtigungen auf Ebene des AI-Tools.

5.1. Die Monorepo-Struktur

Basierend auf den Erkenntnissen zu Multi-Root Workspaces [14] definieren wir folgende Struktur für zKalkulator. Diese Struktur trennt die Runtimes physisch, erlaubt aber logische Verknüpfungen.
Verzeichnisbaum:

```
zKalkulator-Root/
├──.git/
├──.vscode/
│   └── settings.json       <-- Multi-Root Workspace Settings [14]
├──.lovableignore           <-- KRITISCH: Schützt Backend & Config
├── supabase/
│   ├── config.toml
│   ├── migrations/         <-- SQL Migrations (Manuell/Supabase CLI)
│   └── functions/
│       ├── deno.json       <-- Backend-Workspace Config [15]
│       ├── import_map.json  <-- Deno Imports [15]
│       ├── _shared/        <-- Shared Business Logic (Deno-optimiert)
│       └── calculate-api/   <-- Edge Function
│           └── index.ts
├── packages/
│   └── shared-types/       <-- Zod Schemas & Types (Isomorph)
```

│ ├── src/
│ │ └── index.ts <-- Exportiert Zod Schemas
│ └── package.json <-- Dummy package.json für Node-Resolution
├── apps/
│ └── frontend/ <-- Lovable Playground (Vite/React)
│ ├── src/
│ ├──.lovable/ <-- Knowledge Files
│ └── vite.config.ts <-- Enthält Aliase für @shared
└── README.md

5.2. Die Strategie der .lovableignore

Die Datei .lovableignore ist der wichtigste Schutzmechanismus. Sie funktioniert analog zu .gitignore, instruiert aber die AI, bestimmte Pfade weder zu lesen noch zu schreiben.[22] Dies ist entscheidend, um "Context Poisoning" zu verhindern – wenn die AI zu viel irrelevanten Code liest, sinkt die Qualität ihrer Antworten.

Empfohlener Inhalt für zKalkulator:

Backend Isolation: Lovable darf Backend nicht anfassen oder lesen

supabase/functions/
supabase/migrations/
supabase/config.toml

Shared Validation Logic: Darf nur konsumiert, nicht geändert werden

Lovable soll die Typen nutzen, aber nicht versuchen, sie umzuschreiben

packages/shared-types/

Configuration Protection: Verhindert, dass AI den Build kaputt macht

vite.config.ts
tsconfig.json
.github/

Insight: Durch das Ignorieren von supabase/functions/ zwingen wir die AI, die API als "Black Box" zu betrachten. Sie kann nicht versuchen, die Logik in der Edge Function zu ändern, um ein Frontend-Problem zu lösen (ein häufiges Verhalten von Coding-Assistenten). Sie *muss* das Frontend anpassen oder den menschlichen Entwickler um eine Backend-Änderung bitten.

5.3. VS Code Workspace Isolation

Da Deno und Node unterschiedliche LSPs (Language Server Protocols) verwenden, kommt es in Monorepos oft zu Fehlern, wo VS Code Node-Imports in Deno-Dateien vorschlägt und umgekehrt.[11] Dies führt zu einer frustrierenden Entwicklererfahrung ("Red Squiggly Lines everywhere").

Lösung in .vscode/settings.json:

JSON

```
{
  "deno.enable": false,
  "deno.enablePaths": ["./supabase/functions"],
  "deno.unstable": true,
  "typescript.validate.enable": true
}
```

Dies aktiviert den Deno-LSP *nur* im supabase/functions Ordner. Alle anderen Ordner nutzen den Standard TypeScript-LSP für Node. Dies löst das in [13] beschriebene "Intellisense-Chaos" und ermöglicht eine saubere Trennung der Arbeitsumgebungen innerhalb desselben Fensters.

6. Operational Workflow: Phasenplan und Zyklus

Der operative Betrieb von zKalkulator gliedert sich in vier Phasen, die zyklisch durchlaufen werden. Dieser Prozess minimiert das Risiko, dass die AI kritische Infrastruktur beschädigt und stellt sicher, dass menschliche Intelligenz dort eingesetzt wird, wo sie am wertvollsten ist: bei der Definition und Validierung.

Phase 1: Manuelle Kern-Entwicklung (Backend Dominanz)

In dieser Phase hat die AI (Lovable) **Schreibverbot** auf alle Backend-Dateien und Shared-Types. Der menschliche Architekt legt das Fundament.

1. **Datenmodellierung:** Definition der Tabellen in Supabase via SQL oder Table Editor.
2. **Row Level Security (RLS):** Implementierung der Sicherheitsrichtlinien. Lovable darf *niemals* RLS-Policies schreiben, da die AI Sicherheitslücken oft nicht erkennt oder Policies zu permissiv gestaltet ("TO public USING (true)").[21]
3. **Edge Function Logic:** Implementierung der komplexen Geschäftslogik (z.B. Kalkulationen) in Deno unter Verwendung der Shared Zod Schemas.
4. **Contract Publishing:** Push der aktualisierten Typen und OpenAPI-Specs in das _shared Verzeichnis und Generierung der TypeScript-Clients.

Phase 2: AI-Instruktion & Kontext-Setting

Bevor Lovable eine Zeile Frontend-Code schreibt, muss der Kontext geladen werden. Dies ist der "Briefing"-Schritt.

1. **Knowledge File Update:** Aktualisierung der knowledge.md mit den neuen Business-Regeln und API-Strukturen.[23]
2. **Prompt-Engineering:** Nutzung von Meta-Prompts, um Lovable auf die Rolle des "Frontend-Consumers" zu beschränken. Ein Prompt muss explizit verbieten, Mock-Daten zu erfinden, wenn eine echte API existiert.[3]
    - *Beispiel:* "Du bist ein Frontend-Entwickler. Nutze ausschließlich die Typen aus @shared/types. Erfinde keine eigenen Interfaces für API-Responses."

Phase 3: AI-gestützte Frontend-Implementierung ("Vibe Coding")

Hier übernimmt Lovable die Umsetzung. Dies ist die Phase der höchsten Geschwindigkeit.

1. **Component Generation:** Erstellung von UI-Komponenten basierend auf Shadcn/UI (Lovable Standard).
2. **Integration:** Lovable verbindet die Komponenten mit den Supabase-Hooks, die durch den Contract definiert wurden. Die AI nutzt openapi-fetch oder den Supabase JS Client, aber streng nach den Vorgaben der generierten Typen.
3. **Visual Feedback Loop:** Der Entwickler prüft visuell das Ergebnis im Lovable-Preview oder lokal via Vite.

Phase 4: Validierung & Refactoring

Nachdem die AI geliefert hat, tritt der Mensch wieder als "Gatekeeper" auf.

1. **Type Check:** Ein strikter tsc (TypeScript Compiler) Lauf prüft, ob Lovable Typen halluziniert hat.
2. **Linter:** eslint Regeln verhindern, dass any Typen eingeschmuggelt wurden.

3. **Review:** Der Code wird auf Einhaltung der Design-Patterns geprüft. Oft ist AI-Code funktional korrekt, aber schlecht strukturiert (z.B. riesige Komponenten). Hier greift der Mensch ein oder instruiert die AI zum Refactoring ("Zerlege diese Komponente in drei Sub-Komponenten").
4. **Commit:** Erst nach erfolgreicher Validierung wird der Code ins Repository gemerged.

7. DevEx-Optimierung: Lokale Simulation & Mocking

Eine hervorragende Developer Experience (DevEx) im Hybrid-Team bedeutet, dass das Frontend entwickelt werden kann, auch wenn das Backend noch nicht fertig ist oder instabil ist, und dass das Backend entwickelt werden kann, ohne das Frontend zu brechen. Dies erfordert fortgeschrittene Simulationstechniken.

7.1. AI-Code-Isolation durch Mock Service Worker (MSW)

Ein häufiges Problem bei AI-Frontends ist, dass sie bei fehlenden APIs "Fake-Code" direkt in die Komponenten schreiben (z.B. const data = [...]). Dies muss später mühsam entfernt werden und birgt das Risiko, dass Test-Daten in die Produktion gelangen.
Die Lösung für zKalkulator ist **MSW (Mock Service Worker)**. Wir generieren MSW-Handler automatisch aus der OpenAPI-Spec, die wir in Phase 1 erstellt haben.[24]
Workflow:

1. Das Frontend (generiert von Lovable) fordert Daten an (z.B. GET /calculation). Es nutzt dabei den generierten API-Client.
2. MSW fängt den Request im Browser (während der Entwicklung) ab.
3. MSW validiert den Request gegen das OpenAPI-Schema (Zod). Wenn der Request nicht dem Schema entspricht (z.B. falscher Parameter-Typ), wirft MSW einen Fehler. Dies gibt Lovable sofortiges Feedback ("Bad Request"), ohne dass das Backend involviert ist.
4. MSW antwortet mit Mock-Daten, die strikt dem Schema entsprechen. Diese Daten werden dynamisch generiert (z.B. mit faker.js), basierend auf den Typ-Definitionen in der OpenAPI-Spec.

Vorteil: Lovable schreibt von Anfang an "echten" Fetch-Code (supabase.functions.invoke(...)). MSW simuliert lediglich das Netzwerk. Wenn das Backend fertig ist, wird MSW einfach über eine Environment-Variable deaktiviert. Es ist kein Refactoring des AI-Codes nötig ("Code once, run everywhere").

7.2. Supabase Local Development & Isolation

Für die Backend-Entwicklung nutzen wir den lokalen Supabase Stack (supabase start). Dies stellt sicher, dass Edge Functions lokal in einer Docker-Umgebung laufen, die der Produktionsumgebung (Deno Deploy) sehr ähnlich ist.25
Ein spezifisches Problem hierbei ist das Hot Reloading. supabase functions serve unterstützt Hot Reloading, aber in komplexen Monorepos mit Shared Libraries funktioniert dies manchmal nicht zuverlässig, wenn Änderungen außerhalb des Funktions-Ordners vorgenommen werden.27
Workaround: Ein Watcher-Skript (nodemon oder deno task watch), das bei Änderungen im packages/shared-types Ordner den supabase functions serve Prozess neu startet.

8. Data Privacy & Anonymization: Synthetische Realität

Für die Entwicklung benötigt zKalkulator realistische Daten. Ein Dump der Produktionsdatenbank ist aus Datenschutzgründen (GDPR) strikt verboten. Die manuelle Erstellung von Seed-Daten ist jedoch aufwendig und deckt oft nicht alle Edge Cases ab (z.B. extrem lange Strings, Sonderzeichen).
Wir integrieren die **PostgreSQL Anonymizer** Extension direkt in den lokalen Supabase-Docker-Stack.[28]
Da Supabase auf Standard-Postgres aufbaut, können wir den Datenbank-Container anpassen oder initialisieren.
Strategie des "Anonymous Dump":

Die sicherste Methode für lokale Entwicklung ist ein Anonymized Dump aus der Produktion. Wir erstellen ein Skript, das auf der Produktionsdatenbank läuft (oder auf einem replizierten Staging-System) und einen SQL-Dump erzeugt, bei dem sensible Felder dynamisch maskiert werden.

1. **Maskierung:** Nutzung von pg_dump_anon (ein Wrapper um pg_dump), der SECURITY LABEL Regeln in der Datenbank respektiert.
    - SECURITY LABEL FOR anon ON COLUMN users.email IS 'MASKED WITH FUNCTION anon.fake_email()';
    - SECURITY LABEL FOR anon ON COLUMN users.name IS 'MASKED WITH FUNCTION anon.fake_last_name()';
2. **Export:** Das Skript exportiert die Daten. Das Ergebnis ist eine seed.sql Datei, die strukturell identisch zur Produktion ist, aber nur Fake-Daten enthält.[30]
3. **Import:** Lokal: supabase start lädt diesen seed.sql automatisch beim Start.

Dies ermöglicht Lovable, mit "echt aussehenden" Daten zu arbeiten, was die Qualität der generierten UIs massiv verbessert (z.B. korrekte Darstellung langer Namen, Pagination bei vielen Datensätzen), ohne Datenschutzrisiken einzugehen.[32] Die AI sieht nie echte Kundendaten.

9. Das "Knowledge File": Programmierung der kognitiven Schnittstelle

Das Knowledge File (.lovable/knowledge.md oder Systemeinstellungen) ist das Herzstück der AI-Steuerung im zKalkulator Projekt. Es ist nicht nur Dokumentation, sondern **ausführbare Anweisung** für das Large Language Model (LLM).[5]
Für zKalkulator muss dieses File extrem spezifisch sein. Allgemeine Anweisungen führen zu allgemeinem (und oft falschem) Code. Wir nutzen hier Techniken des "Meta-Prompting", bei denen wir der AI explizite Rollen und Grenzen zuweisen.

9.1. Template für zKalkulator Knowledge File

Nachfolgend ein Template, das auf den Best Practices [23] basiert und spezifisch auf die Hybrid-Topologie zugeschnitten ist:

zKalkulator Project Guidelines & Cognitive Context

1. Role & Boundaries (The "Prime Directives")

- **Role:** You are a Senior React Frontend Engineer specializing in Performance and Accessibility.
- **Restriction:** You DO NOT verify, modify, or suggest changes to Backend Logic (Supabase/Edge Functions). Treat the API as an immutable external service.
- **Validation:** You MUST use the provided Zod Schemas (@shared/types) for all form validations. Do not write custom regex.

2. Tech Stack Constraints

- **Framework:** React + Vite + TypeScript (Strict Mode).
- **Styling:** Tailwind CSS + Shadcn/UI ONLY. Do not invent new CSS classes or use inline styles.
- **State Management:** TanStack Query (React Query) v5 for all Async Data.
- **API Client:** Use supabase.functions.invoke() exclusively.

3. Critical Rules (DO NOT BREAK)

- **NO HARDCODED API KEYS:** Use import.meta.env. Warn the user if a key seems missing.
- **NO MOCK DATA IN COMPONENTS:** If data is missing, handle the isLoading state or ask the user for the API contract update. Do not hardcode JSON arrays in components.

- **RLS Awareness:** Assume all data queries are scoped to the current user. Do not try to query public tables unless instructed.
- **Import Strategy:**
    - Import shared types from @shared/types.
    - NEVER import directly from ../../supabase/functions. Relative imports crossing the apps/ boundary are FORBIDDEN.

4. Error Handling Pattern

- Every Supabase call must be wrapped in a standardized error handler that displays a Toast notification via sonner.
- Log errors to console only in development.

5. Coding Style

- Functional Components only.
- Use explicit return types for all functions.
- No any. Use unknown if strictly necessary and cast with Zod.

Insight: Der Abschnitt "Critical Rules" adressiert direkt die identifizierten Risiken: Halluzination von Keys [21], fehlerhafte Imports [13] und die Tendenz zur Erstellung von Mock-Daten ("Code Drifting").

10. Security & Governance: Schutz der Infrastruktur

In einem AI-getriebenen Projekt ist Sicherheit nicht nur eine Frage von Firewalls, sondern von Code-Integrität. Da die AI potenziell unsicheren Code generieren kann, müssen Sicherheitsmechanismen auf der Infrastrukturebene greifen, wo die AI keinen Zugriff hat.

10.1. Row Level Security (RLS) als letzte Verteidigungslinie

Supabase verlässt sich auf RLS, um Datenzugriffe zu steuern. Ein häufiger Fehler von AI-Modellen ist es, RLS-Policies zu ignorieren oder SQL-Queries zu generieren, die versuchen, RLS zu umgehen (z.B. durch Nutzung des service_role Keys im Client, was fatal wäre).
Maßnahme:

1. **Service Role Key Isolation:** Der service_role Key darf niemals im Frontend-Code oder in den Umgebungsvariablen vorkommen, auf die Lovable Zugriff hat. Er darf nur in den Edge Functions (Backend) existieren.
2. **RLS-Testing:** Wir implementieren automatisierte Tests (z.B. mit pgTAP), die sicherstellen, dass RLS-Policies aktiv sind. Diese Tests laufen in der CI/CD-Pipeline.
3. **Policy Audit:** Der menschliche Architekt muss jede neue Tabelle und deren RLS-Policy manuell reviewen. Lovable ist für RLS "blind".

10.2. Secret Management

Lovable erkennt API-Keys im Chat und warnt davor.21 Dennoch besteht das Risiko, dass Keys in den generierten Code gelangen.
Lösung:

- Nutzung von **Supabase Vault** oder Environment Variables für alle Secrets.
- Implementierung eines Pre-Commit Hooks (z.B. git-secrets oder trufflehog), der den Code auf Hardcoded Secrets scannt, bevor er committed wird. Da Lovable direkt in GitHub committen kann (via Integration), muss dieser Check in der GitHub Action laufen, die den Pull Request validiert.

11. CI/CD & Automatisierte Qualitätssicherung

Die Hybrid-Topologie erfordert eine CI/CD-Pipeline, die nicht nur testet, sondern auch **reguliert**. Sie ist der automatisierte Gatekeeper, der verhindert, dass "halluzinierter" Code die Produktion erreicht.

11.1. Der "Anti-Regression" Check

Da Lovable dazu neigt, bestehenden Code zu vereinfachen oder zu entfernen ("Code Golfing"), muss die CI prüfen, ob kritische Backend-Aufrufe verändert wurden.

- **Snapshot-Testing:** Wir nutzen Vitest Snapshots für die generierten API-Client-Aufrufe. Wenn Lovable die Art und Weise ändert, wie die API aufgerufen wird (z.B. Parameter-Reihenfolge), schlägt der Test fehl.
- **Type-Coverage:** Ein Script prüft, ob der Anteil an any-Typen im Codebase steigt. Ein Anstieg deutet auf "lazy AI coding" hin und blockiert den Merge.

11.2. Deployment Pipeline Architecture

Die Pipeline muss die Abhängigkeiten zwischen Backend (Deno) und Frontend (Node) respektieren.

1. **Backend Verification:** Zuerst werden die Supabase Functions und Migrationen gegen eine temporäre Datenbank getestet.
2. **Backend Deploy:** Bei Erfolg werden die Funktionen deployed (supabase functions deploy).
3. **Integration Test:** Ein Skript führt E2E-Tests (z.B. Playwright) gegen das *Staging*-Backend aus. Hier wird geprüft, ob das Frontend noch mit dem (potenziell neuen) Backend spricht.
4. **Frontend Build:** Erst wenn das Backend grün ist und die Integrationstests bestanden sind, baut Vite das Frontend. Dies verhindert, dass ein Frontend deployed wird, das auf eine kaputte oder nicht existente Backend-Version verweist.

12. Fazit

Das Projekt **zKalkulator** zeigt exemplarisch, wie moderne Softwareentwicklung im Zeitalter von AI-Tools aussehen muss. Es geht nicht mehr nur um Code, sondern um **Orchestrierung**. Lovable ist ein leistungsstarker Motor, aber Supabase und die manuelle Architektur sind das Lenkrad und die Bremsen. Die strikte Trennung durch .lovableignore, die Verbindung durch Shared-Types und Zod-Schemas, und die Simulation durch MSW schaffen eine Umgebung, in der Geschwindigkeit (AI) und Sicherheit (Backend) koexistieren können. Der Schlüssel liegt in der **Disziplin des Contracts**: Solange der Vertrag steht, kann die AI im Frontend "tanzen", ohne das Backend zum Einsturz zu bringen. Die Investition in diese robuste Hybrid-Topologie zahlt sich durch drastisch reduzierte Wartungskosten und eine höhere Systemstabilität aus. Der Mensch wird vom Coder zum Kurator, die AI vom Spielzeug zum Werkzeug.