



## Bremen Livability Index

*Ein geodatenbasiertes Bewertungssystem  
für die Lebensqualität in Bremen*

Projektdokumentation im Modul

### Geodatenverarbeitung

Wintersemester 2025 / 2026

**Autor:** Milad Awad

**Matrikel-Nr.:** 5358834

**Dozent:** Dr.-Ing. Christian Seip

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>iii</b>
<b>Tabellenverzeichnis</b>	<b>iii</b>
<b>Listingverzeichnis</b>	<b>iii</b>
<b>Abkürzungsverzeichnis</b>	<b>iv</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Zielsetzung . . . . .	1
1.3 Abgrenzung . . . . .	1
1.4 Aufbau der Arbeit . . . . .	1
<b>2 Grundlagen</b>	<b>2</b>
2.1 Lebensqualität und urbane Indizes . . . . .	2
2.2 Geoinformationssysteme und räumliche Datenbanken . . . . .	2
2.3 Koordinatenreferenzsystem . . . . .	2
2.4 REST-APIs und das Client-Server-Modell . . . . .	3
2.5 Das BLoC-Design-Pattern . . . . .	3
<b>3 Datenquellen</b>	<b>4</b>
3.1 OpenStreetMap . . . . .	4
3.1.1 Abfrage über die Overpass API . . . . .	4
3.1.2 Datenkategorien . . . . .	4
3.1.3 Datenqualität und Vollständigkeit . . . . .	5
3.2 Unfallatlas . . . . .	5
3.2.1 Datenformat und Filterung . . . . .	5
3.2.2 Koordinatentransformation . . . . .	5
3.2.3 Zeitraum und Umfang . . . . .	5
<b>4 Systemarchitektur</b>	<b>6</b>
4.1 Architekturüberblick . . . . .	6
4.2 Projektstruktur . . . . .	6
4.3 Technologiestack . . . . .	7
4.4 Deployment-Architektur . . . . .	7
<b>5 Datenbankdesign</b>	<b>8</b>
5.1 Schema-Organisation . . . . .	8
5.2 Tabellenstruktur . . . . .	8
5.3 Geometrietypen . . . . .	8
5.4 Räumliche Indizierung . . . . .	9
5.5 ORM-Abbildung . . . . .	9
<b>6 Bewertungsmethodik</b>	<b>10</b>
6.1 Bewertungsformel . . . . .	10
6.2 Positive Faktoren . . . . .	10
6.3 Negative Faktoren . . . . .	10
6.4 Score-Interpretation und Implementierung . . . . .	11

<b>7 Datenerfassung</b>	<b>12</b>
7.1 OSM-Datenerfassung . . . . .	12
7.1.1 Ablauf . . . . .	12
7.1.2 Geometriekonvertierung . . . . .	12
7.2 Unfallatlas-Datenerfassung . . . . .	13
7.2.1 Download und Extraktion . . . . .	13
7.2.2 Filterung nach Bremen . . . . .	13
7.2.3 Koordinatentransformation . . . . .	13
7.2.4 Schweregrad-Mapping . . . . .	13
7.2.5 Verfügbare Jahrgänge . . . . .	13
<b>8 Backend-API</b>	<b>14</b>
8.1 Endpunkte . . . . .	14
8.2 Der /analyze-Endpunkt . . . . .	14
8.2.1 Request-Modell . . . . .	14
8.2.2 Response-Modell . . . . .	15
8.3 Dependency Injection und CORS . . . . .	15
8.4 Geokodierung . . . . .	15
<b>9 Frontend</b>	<b>16</b>
9.1 Architektur und State Management . . . . .	16
9.2 Kartenansicht . . . . .	16
9.3 Benutzeroberfläche – Liquid Glass Design . . . . .	16
9.4 Authentifizierung . . . . .	17
9.4.1 Cross-Device E-Mail-Link-Flow . . . . .	17
9.5 Favoritenverwaltung . . . . .	17
9.6 Nutzerpräferenzen . . . . .	18
9.7 Adresssuche . . . . .	18
<b>10 Testing und Deployment</b>	<b>19</b>
10.1 Backend-Tests . . . . .	19
10.1.1 Code Coverage . . . . .	19
10.2 Frontend-Tests . . . . .	19
10.3 Continuous Integration . . . . .	19
10.4 Deployment . . . . .	20
<b>11 Ergebnisse und Diskussion</b>	<b>21</b>
11.1 Exemplarische Standortbewertungen . . . . .	21
11.1.1 Interpretation . . . . .	22
11.2 Stärken des Systems . . . . .	22
11.3 Limitierungen . . . . .	22
11.4 Verbesserungspotenzial . . . . .	23
<b>12 Fazit und Ausblick</b>	<b>24</b>
12.1 Zusammenfassung . . . . .	24
12.2 Beantwortung der Zielsetzung . . . . .	24
12.3 Ausblick . . . . .	24
<b>Literaturverzeichnis</b>	<b>25</b>

# Abbildungsverzeichnis

2.1	Client-Server-Kommunikation über die REST-API . . . . .	3
2.2	BLoC-Pattern: Event-State-Kreislauf . . . . .	3
4.1	Systemarchitektur des Bremen Livability Index . . . . .	6
6.1	Berechnungsfluss des Livability Scores . . . . .	11
9.1	Anmeldebildschirm mit den verfügbaren Authentifizierungsmethoden . . . . .	17
9.2	Adresssuche mit Geokodierungsergebnissen für die Hochschule Bremen . . . . .	18
10.1	CI/CD-Pipeline: Pfadbasierte Auslösung nach dem Merge . . . . .	20
11.1	Mobile Ansichten der Anwendung (iOS) . . . . .	21
11.2	Webanwendung im Desktop-Browser: Kartenansicht mit Score-Aufschlüsselung . . . . .	22

# Tabellenverzeichnis

3.1	OSM-Datenkategorien und verwendete Tags . . . . .	4
3.2	Schweregrade der Unfälle im Unfallatlas . . . . .	5
4.1	Verwendete Technologien . . . . .	7
5.1	Gemeinsames Spaltenschema der Geo-Tabellen . . . . .	8
6.1	Positive Einflussfaktoren (Summe Max.: 60) . . . . .	10
6.2	Negative Einflussfaktoren (Summe Max.: 57) . . . . .	10
6.3	Farbcodierung des Livability Scores . . . . .	11
8.1	API-Endpunkte . . . . .	14
9.1	Unterstützte Anmeldemethoden nach Plattform . . . . .	17
10.1	Backend-Testdateien und Testumfang . . . . .	19
10.2	GitHub-Actions-Workflows . . . . .	20
11.1	Exemplarische Livability Scores für Bremer Standorte . . . . .	21

# Listingverzeichnis

3.1	Bounding-Box-Definition für Bremen . . . . .	4
5.1	Schema- und PostGIS-Initialisierung . . . . .	8
5.2	Beispiel: GiST-Index auf der Tabelle <code>trees</code> . . . . .	9
5.3	ORM-Basisklasse und Beispielmodell <code>Tree</code> . . . . .	9
6.1	Aggregation in <code>calculate_score</code> . . . . .	11
7.1	Filterung auf Bremen . . . . .	13
8.1	Beispiel-Request an <code>/analyze</code> . . . . .	14
8.2	Beispiel-Response von <code>/analyze</code> . . . . .	15

# Abkürzungsverzeichnis

## **API**

Application Programming Interface. [1–3](#), [14](#), [15](#), [17](#)

## **BLoC**

Business Logic Component. [3](#), [7](#), [16](#)

## **CI/CD**

Continuous Integration /Continuous Deployment. [18](#)

## **CORS**

Cross-Origin Resource Sharing. [15](#)

## **CRS**

Coordinate Reference System (Koordinatenreferenzsystem). [2](#)

## **GIS**

Geoinformationssystem. [1](#), [2](#)

## **ORM**

Object-Relational Mapping. [7](#)

## **OSM**

OpenStreetMap. [1](#), [2](#), [4](#), [5](#), [12](#), [15](#), [20](#)

## **REST**

Representational State Transfer. [3](#), [6](#), [14](#)

## **SQL**

Structured Query Language. [8](#)

## **UTM**

Universal Transverse Mercator. [2](#), [5](#), [13](#)

## **WGS 84**

World Geodetic System 1984. [2](#), [5](#), [8](#), [11](#), [12](#)

# 1 Einleitung

## 1.1 Motivation

Die Wahl des Wohnortes ist eine der weitreichendsten Entscheidungen im Alltag. Faktoren wie die Nähe zu Grünflächen, die Erreichbarkeit von Nahversorgung und öffentlichem Nahverkehr, aber auch potenzielle Belastungen durch Lärm, Verkehr oder Industrieanlagen beeinflussen die Lebensqualität eines Standortes erheblich. Obwohl zahlreiche Geodaten zu diesen Aspekten frei verfügbar sind – insbesondere über [OpenStreetMap \(OSM\)](#) und den Unfallatlas des Statistischen Bundesamtes – fehlt es an Werkzeugen, die diese heterogenen Datenquellen standortbezogen aggregieren und in einem leicht verständlichen Index zusammenfassen.

Internationale Lebensqualitätsrankings wie der *Global Liveability Index* der Economist Intelligence Unit (Economist Intelligence Unit [2024](#)) oder der *Quality of Living Index* von Mercer (Mercer LLC [2019](#)) bewerten Städte auf nationaler oder globaler Ebene, bieten jedoch keine Auflösung auf Stadtteil- oder gar Adressebene. Hier setzt das vorliegende Projekt an.

## 1.2 Zielsetzung

Ziel des Projekts **Bremen Livability Index (BLI)** ist die Entwicklung einer vollständigen Geodatenverarbeitungs-Pipeline, die:

1. räumliche Daten aus [OSM](#) und dem Unfallatlas automatisiert in eine PostGIS-Datenbank importiert,
2. für einen beliebigen Standort innerhalb Bremens einen **Livability Score** (0–100) in Echtzeit berechnet,
3. den Score in positive und negative Einflussfaktoren aufschlüsselt,
4. die umliegenden [GIS](#)-Features als GeoJSON-Objekte zur Visualisierung auf einer interaktiven Karte bereitstellt und
5. dem Nutzer die Möglichkeit gibt, individuelle Gewichtungen (*Präferenzen*) für die einzelnen Faktoren festzulegen.

Das System wird als produktionstaugliche Web-, Mobile- und Desktop-Applikation mit Flutter-Frontend und FastAPI-Backend realisiert. Quellcode: <https://github.com/Milad9A/Bremen-Livability-Index> – Webanwendung: <https://bremen-livability-frontend.onrender.com>

## 1.3 Abgrenzung

Die vorliegende Arbeit beschränkt sich räumlich auf das Gebiet der Freien Hansestadt Bremen (Bounding-Box 53,0° N – 53,2° N, 8,5° E – 9,0° E). Eine Übertragung auf andere Städte ist konzeptionell möglich, wird jedoch nicht umgesetzt. Es werden ausschließlich frei verfügbare, statische Datenquellen verwendet; Echtzeitdaten (z. B. aktuelle Lärmwerte, Luftqualität) werden nicht berücksichtigt.

## 1.4 Aufbau der Arbeit

Nach einer Einführung in Grundlagen und Datenquellen (Kapitel [2–3](#)) werden Systemarchitektur, Datenbankdesign und Bewertungsmethodik (Kapitel [4–6](#)) erläutert. Die automatisierte Datenerfassung (Kapitel [7](#)) sowie die Implementierung von Backend-[API](#) und Frontend (Kapitel [8–9](#)) werden anschließend beschrieben. Kapitel [10](#) behandelt Testing und Deployment; abschließend folgen Ergebnisdiskussion und Fazit (Kapitel [11–12](#)).

## 2 Grundlagen

### 2.1 Lebensqualität und urbane Indizes

Der Begriff *Lebensqualität* umfasst eine Vielzahl objektiv messbarer und subjektiv empfundener Dimensionen, darunter Gesundheitsversorgung, Bildung, Sicherheit, Umweltqualität und infrastrukturelle Erreichbarkeit (Economist Intelligence Unit 2024). Internationale Ansätze wie der *Global Liveability Index* (EIU) oder der *Quality of Living Index* (Mercer) bewerten Städte auf Grundlage makroskopischer Indikatoren (Mercer LLC 2019). Beide operieren auf Stadtebene und bieten keine feinräumige Auflösung innerhalb einer Stadt.

Ziel des Bremen Livability Index ist es, einen **mikroskaligen** Lebensqualitätsindex zu realisieren, der für *jeden beliebigen Punkt* innerhalb des Stadtgebiets einen Score berechnet. Dazu werden **Geoinformationssystem (GIS)**-Methoden eingesetzt, um die räumliche Nähe zu positiven und negativen Infrastrukturmerkmalen quantitativ zu erfassen.

### 2.2 Geoinformationssysteme und räumliche Datenbanken

Ein **Geoinformationssystem (GIS)** dient der Erfassung, Verwaltung, Analyse und Darstellung raumbezogener Daten. Im Kontext dieses Projekts werden insbesondere zwei Fähigkeiten benötigt:

- **Räumliche Abfragen:** Bestimmung aller Objekte innerhalb eines definierten Radius um einen Punkt (*proximity queries*).
- **Distanzberechnung:** Berechnung der ellipsoidalen Entfernung zwischen geographischen Koordinaten unter Berücksichtigung der Erdkrümmung.

Die Datenbank PostgreSQL bietet mit der Erweiterung **PostGIS** (PostGIS Project Steering Committee 2024) eine leistungsfähige räumliche Datenbanklösung. PostGIS unterstützt sowohl den Datentyp **GEOMETRY** (kartesische Ebene) als auch **GEOGRAPHY** (ellipsoidale Berechnung auf dem WGS 84-Ellipsoid). Für die exakte Entfernungs berechnung in Metern wird in diesem Projekt ausschließlich der Typ **GEOGRAPHY** verwendet.

Zentrale PostGIS-Funktionen im Projekt sind **ST\_DWithin** (Proximity-Prüfung mit GiST-Index), **ST\_Distance** (ellipsoidale Entfernung in Metern), **ST\_MakePoint/ST\_SetSRID** (Punkterzeugung mit **CRS**-Zuweisung) und **ST\_AsGeoJSON** (Konvertierung für das Frontend).

### 2.3 Koordinatenreferenzsystem

Das gesamte Projekt verwendet durchgängig **EPSG:4326 (WGS 84)** als einziges **Coordinate Reference System (Koordinatenreferenzsystem)** – für die Datenbank, die **API**-Ein- und Ausgabe sowie alle **OSM**-Quelldaten.

Die Rohdaten des Unfallatlas liegen im Quellformat EPSG:25832 (ETRS89 / **UTM** Zone 32N) vor. Sie werden jedoch während der Datenerfassung (Kapitel 7) automatisch nach EPSG:4326 reprojiziert, sodass EPSG:25832 ausschließlich als Eingangsformat des Ingestion-Skripts auftritt und im restlichen System nicht mehr in Erscheinung tritt.

Die Wahl von EPSG:4326 als Datenbankformat in Kombination mit dem PostGIS-Typ **GEOGRAPHY** stellt sicher, dass alle Distanzberechnungen korrekt auf dem **WGS 84**-Ellipsoid erfolgen – ohne die Notwendigkeit einer zusätzlichen Projektion (PostGIS Project Steering Committee 2024).

## 2.4 REST-APIs und das Client-Server-Modell

Die Kommunikation zwischen Frontend und Backend erfolgt über eine [Representational State Transfer \(REST\)-API](#). Das Backend stellt HTTP-Endpunkte bereit, die JSON-formatierte Anfragen entgegennehmen und Antworten zurückgeben. Dieses zustandslose Architekturmuster ermöglicht eine klare Trennung von Darstellungs- und Geschäftslogik sowie eine einfache Skalierbarkeit (Ramírez 2024a). Die API ist öffentlich erreichbar unter <https://bremen-livability-backend.onrender.com>; eine interaktive Swagger-Dokumentation steht unter /docs zur Verfügung. Abbildung 2.1 zeigt den Kommunikationsfluss.



Abbildung 2.1: Client-Server-Kommunikation über die REST-API

## 2.5 Das BLoC-Design-Pattern

Das [Business Logic Component \(BLoC\)](#)-Design-Pattern (Angelov 2024) trennt in Flutter-Anwendungen die UI-Schicht von der Geschäftslogik: Die Benutzeroberfläche sendet *Events* an den BLoC, der diese verarbeitet und neue *States* emittiert. Die UI reagiert reaktiv auf Zustandsänderungen. In Kombination mit dem [Freezed](#)-Codegenerator entstehen typsichere, unveränderliche (*immutable*) Zustandsobjekte, die eine vorhersagbare Zustandsverwaltung gewährleisten. Abbildung 2.2 veranschaulicht diesen Kreislauf.

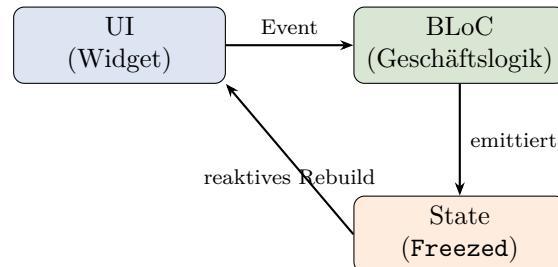


Abbildung 2.2: BLoC-Pattern: Event-State-Kreislauf

# 3 Datenquellen

Für die Berechnung des Livability Scores werden zwei komplementäre offene Datenquellen herangezogen: [OpenStreetMap \(OSM\)](#) für allgemeine Geodaten und der *Unfallatlas* für verkehrsbezogene Unfalldaten.

## 3.1 OpenStreetMap

OSM (OpenStreetMap Foundation 2024b) ist ein kollaboratives Kartenprojekt, das weltweit freie Geodaten unter der Open Database License (ODbL 1.0) bereitstellt (OpenStreetMap Foundation 2012). Die Daten werden von einer Community aus über 10 Millionen registrierten Nutzern gepflegt und umfassen Punkte (*Nodes*), Wege (*Ways*) und Relationen mit semantischen Tags.

### 3.1.1 Abfrage über die Overpass API

Der Zugriff auf die OSM-Daten erfolgt über die **Overpass API** (OpenStreetMap Wiki Contributors 2024), eine spezialisierte Leseschnittstelle für räumliche Abfragen. Die Abfragen verwenden die Overpass-QL-Syntax mit einem Bounding-Box-Filter für das Stadtgebiet Bremen:

```
1 BREMEN_BBOX = {  
2     "south": 53.0,  
3     "west": 8.5,  
4     "north": 53.2,  
5     "east": 9.0  
6 }
```

Listing 3.1: Bounding-Box-Definition für Bremen

Dieses Gebiet von ca. 420 km<sup>2</sup> umfasst das gesamte Stadtgebiet der Freien Hansestadt Bremen einschließlich Bremerhaven.

### 3.1.2 Datenkategorien

Aus OSM werden 20 thematische Kategorien extrahiert, die als positive oder negative Einflussfaktoren in die Bewertung einfließen (Tabelle 3.1).

Tabelle 3.1: OSM-Datenkategorien und verwendete Tags

Kategorie	OSM-Tags	Geom.	Einfl.
Bäume	natural=tree	Point	+
Parks	leisure=park	Polygon	+
Nahversorgung	amenity=supermarket cafe restaurant bank post_office bakery butcher	Point	+
ÖPNV	highway=bus_stop, railway=tram_stop	Point	+
Gesundheit	amenity=hospital pharmacy doctors clinic	Point	+
Fahrrad	highway=cycleway, cycleway=*, amenity=bicycle_parking	Pt/Ln	+
Bildung	amenity=school university college kindergarten library	Point	+
Sport/Freizeit	leisure=sports_centre swimming_pool playground pitch	Point	+
Fußgänger	highway=pedestrian footway	Line	+
Kultur	tourism=museum gallery, amenity=theatre cinema	Point	+
Industrie	landuse=industrial	Polygon	-
Hauptstraßen	highway=motorway trunk primary	Line	-
Lärm	amenity=nightclub bar pub fast_food car_repair	Point	-
Eisenbahn	railway=rail	Line	-
Tankstellen	amenity=fuel	Point	-
Abfall	landuse=landfill, amenity=recycling	Pt/Pg	-
Strom	power=substation plant generator	Pt/Pg	-
Parkplätze	amenity=parking (Polygon)	Polygon	-
Flughäfen	aeroway=aerodrome helipad	Pt/Pg	-
Baustellen	landuse=construction	Polygon	-

### 3.1.3 Datenqualität und Vollständigkeit

Die Qualität der OSM-Daten variiert je nach Kategorie. Für städtische Gebiete in Deutschland gilt OSM als weitgehend vollständig, insbesondere bei Straßen, Gebäuden und öffentlichen Einrichtungen. Bei Bäumen und Fahrradinfrastruktur bestehen dagegen Lücken, da diese Objekte häufig erst durch spezialisierte Mapping-Kampagnen erfasst werden. Eine systematische Validierung der Datenvollständigkeit liegt außerhalb des Projektumfangs, wird jedoch in Kapitel 11 diskutiert.

## 3.2 Unfallatlas

Der *Unfallatlas* (Statistisches Bundesamt 2024) ist ein Angebot der Statistischen Ämter des Bundes und der Länder und enthält georeferenzierte Straßenverkehrsunfälle mit Personenschaden ab dem Jahr 2016. Die Daten stehen als Open Data unter der Lizenz d1-de/by-2-0 zum Download bereit (Geobasis NRW 2024).

### 3.2.1 Datenformat und Filterung

Die Unfalldaten werden als gezippte CSV-Dateien im Koordinatenreferenzsystem EPSG:25832 (UTM Zone 32N) bereitgestellt. Für das Projekt werden die Daten nach dem Bundeslandschlüssel ULAND=4 (Bremen) gefiltert.

Tabelle 3.2: Schweregrade der Unfälle im Unfallatlas

UKATEGORIE	Schweregrad	Beschreibung
1	fatal	Unfall mit Getöteten
2	severe	Unfall mit Schwerverletzten
3	minor	Unfall mit Leichtverletzten

### 3.2.2 Koordinatentransformation

Da die Unfalldaten im projizierten System EPSG:25832 vorliegen, die Datenbank jedoch EPSG:4326 (WGS 84) erwartet, erfolgt bei der Datenerfassung eine automatische Reprojektion mithilfe der Python-Bibliothek GeoPandas (GeoPandas Contributors 2024). Die Koordinatenpalten (XGCSWGS84/YGCSWG S84 bzw. LINREFX/LINREFY) werden automatisch erkannt. Zudem wird das deutsche Dezimalkomma-Format (z. B. 53,0793) in Dezimalpunkt-Format konvertiert.

### 3.2.3 Zeitraum und Umfang

Es stehen Daten für die Jahre 2016–2024 zur Verfügung. Standardmäßig wird das aktuellste verfügbare Jahr (2024) importiert. Die Anzahl der Unfälle in Bremen variiert dabei je nach Jahr zwischen ca. 1.500 und 2.500 Datensätzen.

# 4 Systemarchitektur

Der Bremen Livability Index ist als klassische **Client-Server-Architektur** mit drei Schichten konzipiert: einer räumlichen Datenbank, einem REST-Backend und einem plattformübergreifenden Frontend. Dieses Kapitel gibt einen Überblick über den Gesamtaufbau und den Technologiestack.

## 4.1 Architekturüberblick

Abbildung 4.1 zeigt die zentralen Komponenten und deren Zusammenspiel. Durch die Trennung der Schichten lassen sich Backend und Frontend unabhängig voneinander entwickeln, testen und deployen.

Die Kernfunktionalität – räumliche Näheanalysen mittels PostGIS-Funktionen wie `ST_DWithin` und `GEOGRAPHY`-Datentypen – erfordert eine vollwertige räumliche Datenbank, die Firebase/Firestore nicht bieten kann. Gleichzeitig stellt Firebase Authentication bewährte OAuth- und Magic-Link-Flows bereit, deren Eigenimplementierung unverhältnismäßig aufwändig wäre. Die Architektur trennt daher bewusst: *geodatenintensive Logik* verbleibt im PostGIS-Backend, während *Identitätsverwaltung und Favoritensynchronisation* über Firebase-Dienste abgewickelt werden.

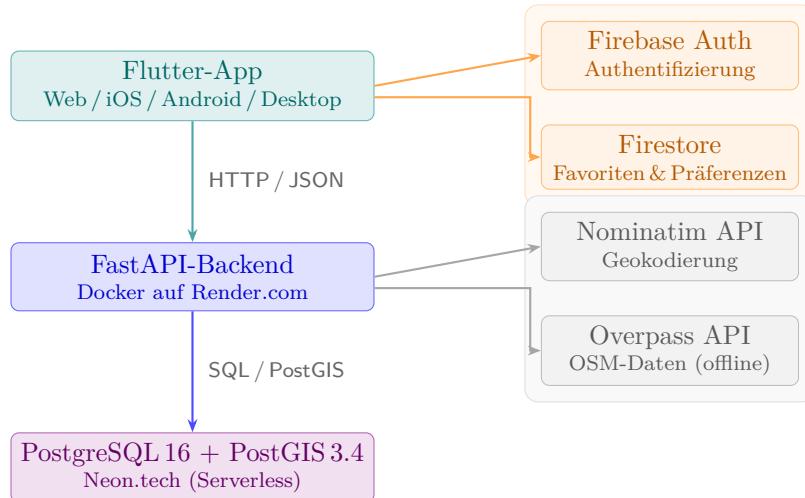


Abbildung 4.1: Systemarchitektur des Bremen Livability Index

## 4.2 Projektstruktur

Das Projekt ist als öffentliches GitHub-Repository unter <https://github.com/Milad9A/Bremen-Livability-Index> verfügbar und in drei Hauptverzeichnisse gegliedert:

### backend/

Enthält das FastAPI-Backend mit den Unterordnern `app/` (Endpunkte, Modelle), `core/` (Datenbank, Scoring, Logging), `services/` (Geokodierung), `scripts/` (Datenerfassung) und `tests/` (Unit-Tests).

### frontend/bli/

Enthält die Flutter-Anwendung mit der Verzeichnisstruktur `lib/core/` (Services, Theme, Widgets), `lib/features/` (Auth, Map, Favorites, Preferences, Onboarding) und plattformspezifischen Ordnern (`android/`, `ios/`, `web/`, `macos/`, `windows/`, `linux/`).

### documentation/

Enthält die vorliegende LaTeX-Dokumentation mit separaten Kapiteldateien und dem Literaturverzeichnis.

## 4.3 Technologiestack

Tabelle 4.1 gibt einen Überblick über alle eingesetzten Technologien und begründet deren Auswahl.

Tabelle 4.1: Verwendete Technologien

Schicht	Technologie	Begründung
Datenbank	PostgreSQL 16 + PostGIS 3.4	Leistungsfähigste Open-Source-Lösung für räumliche Daten; native <b>GEOGRAPHY</b> -Unterstützung (PostGIS Project Steering Committee 2024)
Backend	FastAPI 0.115 (Python)	Hohe Performance (ASGI), automatische OpenAPI-Doku, native Pydantic-Validierung (Ramírez 2024a)
Validierung	Pydantic 2.x	Laufzeit-Datenversionierung über Python-Typannotationen; bildet die Grundlage für FastAPI-Request-Validierung und SQLModel-Typisierung
ORM	SQLModel + GeoAlchemy2	SQLAlchemy-Leistung mit Pydantic-Typisierung; PostGIS-Funktionen als Python-Objekte (Ramírez 2024b; GeoAlchemy2 Contributors 2024)
Frontend	Flutter 3.x (Dart)	Plattformübergreifend aus einer Codebasis (Google LLC 2024b)
Karte	flutter_map 8.x	Open-Source-Kartenwidget; CartoDB Voyager Tiles (flutter_map Contributors 2024)
State Mgmt.	flutter_bloc 9.x	BLoC-Muster für reaktive, testbare Zustandsverwaltung (Angelov 2024)
Auth	Firebase Authentication	Google, GitHub, Magic-Link, anonym; serverseitige Token-Validierung (Google LLC 2024a)
Hosting	Render.com, Neon.tech	Containerbasiertes Hosting + Serverless-DB im Free Tier; EU-Standort (Render Inc. 2024; Neon Inc. 2024)
Container	Docker	Reproduzierbare Build- und Deployment-Umgebung (Docker Inc. 2024)

## 4.4 Deployment-Architektur

Das Deployment erfolgt vollständig cloudbasiert über Infrastruktur im Free Tier und wird über eine deklarative `render.yaml`-Konfiguration gesteuert:

- **Backend:** Docker-Container auf Render.com (Web Service, Region Frankfurt). Der Container wird aus dem `backend/Dockerfile` gebaut und stellt den FastAPI-Server auf Port 8000 bereit. Die Umgebungsvariable `DATABASE_URL` verbindet das Backend mit der Neon.tech-Datenbank.
- **Frontend (Web):** Statische Flutter-Webanwendung auf Render.com (Static Site). Das Build-Skript `render_build.sh` führt `flutter build web` aus. Eine SPA-Rewrite-Regel (`/* → /index.html`) stellt clientseitiges Routing sicher.
- **Frontend (Mobile & Desktop):** Ein GitHub-Actions-Workflow (`build-release.yml`) wird nach jedem erfolgreichen Frontend-Test auf dem `master`-Branch automatisch ausgelöst. Er baut die App für Android (APK), Windows (ZIP), Linux (tarball) und macOS (ZIP) und veröffentlicht alle vier Artefakte als einheitliches GitHub Release mit Zeitstempel-Tag. Desktop-Builds sind ad-hoc-signiert und erfordern ggf. eine Sicherheitsausnahme beim ersten Start.
- **Datenbank:** PostgreSQL 16 mit PostGIS 3.4 auf Neon.tech (Serverless). Die Datenbank skaliert automatisch und bietet branching-fähige Entwicklungsumgebungen.
- **E-Mail-Redirect:** Firebase Hosting leitet Magic-Link-URLs an die Flutter-App weiter.

# 5 Datenbankdesign

Die persistente Speicherung und räumliche Abfrage der Geodaten erfolgt über PostgreSQL 16 mit der Erweiterung PostGIS 3.4 (PostGIS Project Steering Committee 2024; The PostgreSQL Global Development Group 2024) mittels erweiterter Structured Query Language (SQL)-Abfragen. Dieses Kapitel beschreibt das Datenbankschema, die Tabellenstruktur und die Indexierung.

## 5.1 Schema-Organisation

Alle projektspezifischen Tabellen befinden sich im dedizierten Schema `gis_data`. Dieses Schema wird bei der erstmaligen Initialisierung durch das Skript `init_db.sql` angelegt, zusammen mit der PostGIS-Erweiterung:

```
1 CREATE EXTENSION IF NOT EXISTS postgis;
2 CREATE SCHEMA IF NOT EXISTS gis_data;
```

Listing 5.1: Schema- und PostGIS-Initialisierung

## 5.2 Tabellenstruktur

Das Schema umfasst **21 Tabellen**, die sich in zwei Gruppen unterteilen lassen:

1. **Positive Einflussfaktoren** (10 Tabellen): `trees`, `parks`, `amenities`, `public_transport`, `healthcare`, `bike_infrastructure`, `education`, `sports_leisure`, `pedestrian_infrastructure`, `cultural_venues`
2. **Negative Einflussfaktoren** (11 Tabellen): `accidents`, `industrial_areas`, `major_roads`, `noise_sources`, `railways`, `gas_stations`, `waste_facilities`, `power_infrastructure`, `parking_lots`, `airports`, `construction_sites`

Jede Geo-Tabelle folgt einem einheitlichen Aufbau:

Tabelle 5.1: Gemeinsames Spaltenschema der Geo-Tabellen

Spalte	Datentyp	Beschreibung
<code>id</code>	SERIAL PRIMARY KEY	Auto-Inkrement-ID
<code>osm_id</code>	BIGINT	OpenStreetMap-Objekt-ID (entfällt bei Unfalldaten)
<code>name</code>	TEXT	Bezeichnung (optional)
<code>geometry</code>	GEOGRAPHY(type, 4326)	Räumliches Objekt im <a href="#">WGS 84</a> -System
<code>created_at</code>	TIMESTAMP	Zeitstempel der Erfassung

Einige Tabellen verfügen über zusätzliche Typspalten, z. B. `amenity_type` (Art der Einrichtung), `severity` (Unfallschweregrad), `transport_type` (Bus/Tram) oder `healthcare_type`.

## 5.3 Geometrietypen

Abhängig von der Art der Geoobjekte werden drei Typen verwendet: POINT (z. B. Bäume, Haltestellen, Unfälle), LINESTRING (Straßen, Schienen, Rad-/Fußwege) und POLYGON (Parks, Industriegebiete, Parkplätze, Flughäfen). Alle Geometrien werden als GEOGRAPHY (nicht GEOMETRY) gespeichert, sodass Distanzberechnungen automatisch auf dem [WGS 84](#)-Ellipsoid in Metern erfolgen.

## 5.4 Räumliche Indizierung

Für jede Geometriespalte wird ein **GiST-Index** (*Generalized Search Tree*) angelegt. GiST-Indizes ermöglichen effiziente räumliche Abfragen, indem sie die Geometrien in hierarchische Bounding-Boxen partitionieren:

```

1 CREATE INDEX idx_trees_geom
2   ON gis_data.trees
3   USING GIST (geometry);

```

Listing 5.2: Beispiel: GiST-Index auf der Tabelle `trees`

Zusätzlich werden B-Tree-Indizes auf Typspalten (z. B. `amenity_type`, `transport_type`) erstellt, um Abfragen mit Typfiltern zu beschleunigen.

Die Kombination aus GiST-Index und `ST_DWithin` ermöglicht es, die räumlichen Abfragen des Scoring-Algorithmus in wenigen Millisekunden auszuführen – selbst bei Tabellen mit über 40.000 Einträgen (z. B. Bäume).

## 5.5 ORM-Abbildung

Die Datenbankmodelle werden im Backend durch **SQLModel**-Klassen (Ramírez 2024b) abgebildet, die sowohl als SQLAlchemy-ORM-Modelle als auch als Pydantic-Validierungsmodelle dienen. Die Geometriespalten verwenden den Typ `Geography` aus GeoAlchemy2 (GeoAlchemy2 Contributors 2024). Alle 21 Geo-Tabellen erben von einer gemeinsamen Basisklasse `GISBase` mit der Konfiguration `arbitrary_types_allowed = True`. Diese Einstellung ist notwendig, weil Pydantic standardmäßig nur bekannte Python-Typen validiert. Der Typ `Geography` aus GeoAlchemy2 ist kein nativer Python-Typ, sondern eine SQLAlchemy-Spaltendefinition. Ohne diese Einstellung würde Pydantic beim Initialisieren der Modellklassen mit einem `RuntimeError` abbrechen. Der Parameter weist Pydantic an, solche nicht-standardisierten Typen ohne eigene Validator-Logik zu akzeptieren.

Listing 5.3 zeigt die Basisklasse und ein konkretes Modell als Beispiel:

```

1 class GISBase(SQLModel):
2     """Base class for all 21 GIS models."""
3     class Config:
4         arbitrary_types_allowed = True
5
6     class Tree(GISBase, table=True):
7         __tablename__ = "trees"
8         __table_args__ = {"schema": "gis_data"}
9
10    id: Optional[int] = Field(default=None, primary_key=True)
11    osm_id: Optional[int] = Field(sa_column=Column(BigInteger))
12    name: Optional[str] = Field(sa_column=Column(Text))
13    geometry: Any = Field(
14        sa_column=Column(Geography("POINT", srid=4326))
15    )
16    created_at: Optional[datetime] = Field(
17        sa_column=Column(TIMESTAMP, default=datetime.utcnow)
18    )

```

Listing 5.3: ORM-Basisklasse und Beispielmodell `Tree`

# 6 Bewertungsmethodik

Das Kernstück des Bremen Livability Index ist der Bewertungsalgorithmus, der für einen gegebenen geographischen Punkt einen **Livability Score** zwischen 0 und 100 berechnet.

## 6.1 Bewertungsformel

Der Score setzt sich aus einem Basiswert, der Summe positiver Faktoren und der Summe negativer Faktoren zusammen:

$$\text{Score} = \text{clamp}\left(\underbrace{S_{\text{base}}}_{=40} + \sum_{i=1}^9 w_i \cdot f_i^+ - \sum_{j=1}^{11} w_j \cdot f_j^-, 0, 100\right) \quad (6.1)$$

Dabei ist  $S_{\text{base}} = 40$  ein neutraler Ausgangswert,  $f_i^+$  bzw.  $f_j^-$  die Einzelscores der positiven/negativen Faktoren und  $w \in \{0,0; 0,5; 1,0; 1,5\}$  der nutzerspezifische Gewichtungsmultiplikator (*ImportanceLevel: excluded, low, medium, high*).

Faktoren mit hohen Zählergebnissen verwenden logarithmische Skalierung  $f(n) = \min(f_{\max}, \ln(1+n) \cdot k)$ , die übrigen lineare Skalierung  $f(n) = \min(f_{\max}, n \cdot k)$ .

## 6.2 Positive Faktoren

Tabelle 6.1: Positive Einflussfaktoren (Summe Max.: 60)

Faktor	Max.	Radius	Formel
Grünflächen	14	175 m	$\min(9, \ln(1+n_B) \cdot 2,0) + \min(5, n_P \cdot 2,5)$
Nahversorgung	10	550 m	$\min(10, \ln(1+n) \cdot 2,8)$
ÖPNV	8	450 m	$\min(8, \ln(1+n) \cdot 3,5)$
Gesundheit	6	700 m	$\min(6, n \cdot 2,5)$
Fahrrad	6	275 m	$\min(6, \ln(1+n) \cdot 2,5)$
Bildung	5	500 m	$\min(5, n \cdot 1,5)$
Sport/Freizeit	4	700 m	$\min(4, \ln(1+n) \cdot 1,8)$
Kultur	4	500 m	$\min(4, n \cdot 2,0)$
Fußgänger	3	275 m	$\min(3, \ln(1+n) \cdot 1,2)$

## 6.3 Negative Faktoren

Tabelle 6.2: Negative Einflussfaktoren (Summe Max.: 57)

Faktor	Strafe	Radius	Typ
Industriegebiet	10	150 m	Binär
Unfälle	8	120 m	$\min(8, n \cdot 2,0)$
Flughafen	7	600 m	Binär
Hauptstraßen	6	60 m	Binär
Lärmquellen	6	75 m	$\min(6, n \cdot 2,0)$
Abfall	5	250 m	Binär
Eisenbahn	5	100 m	Binär
Tankstelle	3	75 m	Binär
Strom	3	75 m	Binär
Baustelle	2	125 m	Binär
Großparkplatz	2	50 m	Binär

Binäre Faktoren vergeben die volle Strafe, sobald mindestens ein Objekt im Radius vorhanden ist. Zählerbasierte (Unfälle, Lärm) steigen proportional, sind aber nach oben begrenzt. Die Suchradien

(50–700 m) spiegeln den Einflussbereich der Faktoren wider; alle Abfragen nutzen ST\_DWithin auf dem WGS 84-Ellipsoid.

## 6.4 Score-Interpretation und Implementierung

Die App visualisiert den Score durch eine dreistufige Farbcodierung (Tabelle 6.3). Der numerische Wert wird direkt angezeigt.

Tabelle 6.3: Farbcodierung des Livability Scores

Score-Bereich	Farbe	Bewertung
$\geq 70$	Grün	Gut
50–69	Orange	Mittel
< 50	Rot	Schlecht

Abbildung 6.1 veranschaulicht den Berechnungsfluss des Algorithmus: Der Basiswert wird mit den gewichteten positiven und negativen Teilscores verrechnet und das Ergebnis auf [0, 100] begrenzt.

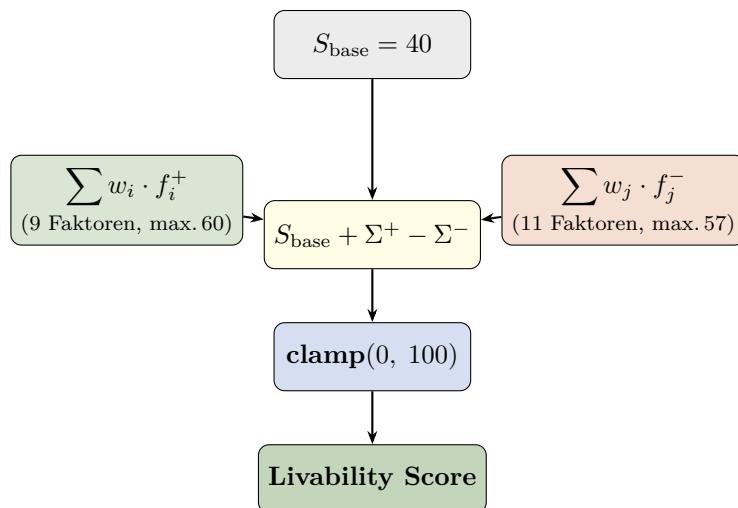


Abbildung 6.1: Berechnungsfluss des Livability Scores

Der Algorithmus ist in `LivabilityScorer` (`core/scoring.py`) implementiert. Jeder Faktor wird durch eine eigene statische Methode berechnet; `calculate_score` aggregiert alle Beiträge zu einem `LivabilityScoreResponse`-Objekt. Listing 6.1 zeigt den zentralen Aggregationsschritt.

```

1 positive = (greenery + amenities + transport + healthcare
2             + bike_infra + education + sports
3             + pedestrian + cultural)
4 negative = (accident_penalty + industrial_penalty
5              + roads_penalty + noise_penalty
6              + railway_penalty + gas_station_penalty
7              + waste_penalty + power_penalty
8              + parking_penalty + airport_penalty
9              + construction_penalty)
10 final_score = max(0.0, min(100.0,
11                     cls.BASE_SCORE + positive - negative))

```

Listing 6.1: Aggregation in `calculate_score`

# 7 Datenerfassung

Die Befüllung der Datenbank erfolgt automatisiert über Python-Skripte, die im Verzeichnis `backen/d/scripts/data_ingestion/` organisiert sind. Der Einstiegspunkt `ingest_all_data.py` ruft beide Ingestion-Module nacheinander auf.

Die Ingestion wird durch das Container-Startskript `entrypoint.sh` gesteuert. Beim Start des Backends prüft das Skript alle 21 Geo-Tabellen auf Leerheit. Ist mindestens eine Tabelle leer, wird die vollständige Ingestion ausgeführt; sind alle Tabellen befüllt, wird sie übersprungen um Zeit zu sparen. Dadurch läuft die Ingestion automatisch beim Erstdeploy auf Render.com sowie nach einem manuellen Datenbank-Reset, nicht aber bei regulären Neustarts des Containers. In der lokalen Entwicklungsumgebung (`start_server.sh`) muss `ingest_all_data.py` einmalig manuell ausgeführt werden.

## 7.1 OSM-Datenerfassung

Die Erfassung der 20 OSM-Kategorien erfolgt über die Python-Bibliothek `overpy` (PhiBo 2024), einen Wrapper für die Overpass API (OpenStreetMap Wiki Contributors 2024).

### 7.1.1 Ablauf

Für jede Kategorie wird folgende Pipeline durchlaufen:

1. **Overpass-Abfrage:** Eine Overpass-QL-Abfrage wird mit dem Bounding-Box-Filter für Bremen formuliert und an die Overpass API gesendet.
2. **Retry-Logik:** Bei Überlastung der API (HTTP 429 oder Gateway-Timeout) wird ein exponentieller Backoff mit Jitter angewendet:

$$\text{Wartezeit} = 10 \cdot 2^{\text{Versuch}} + r \quad r \sim \mathcal{U}(0, 5)$$

Der Faktor  $10 \cdot 2^{\text{Versuch}}$  verdoppelt die Wartezeit mit jedem Fehlversuch (Versuch 1: 20 s, Versuch 2: 40 s, Versuch 3: 80 s usw.), damit der Server Zeit bekommt, sich zu erholen. Der additive Zufallsterm  $r$  (Jitter) aus einer Gleichverteilung über  $[0, 5]$  verhindert, dass mehrere parallele Prozesse nach einem Fehler gleichzeitig einen neuen Versuch starten und die API erneut überlasten. Maximal werden 5 Versuche unternommen.

3. **Tabelle leeren:** `TRUNCATE TABLE gis_data.xxx CASCADE` entfernt alle bestehenden Daten, um eine idempotente Neuerfassung zu gewährleisten.
4. **Bulk-Insert:** Die empfangenen Nodes, Ways oder Relationen werden einzeln in die zugehörige Tabelle eingefügt.

### 7.1.2 Geometriekonvertierung

Die von der Overpass API zurückgegebenen Objekte werden je nach Typ unterschiedlich verarbeitet:

#### Punkte (Nodes)

werden mit `ST_MakePoint(lon, lat)` in einen PostGIS-Point konvertiert und mit `ST_SetSRID(..., 4326)` dem WGS 84-System zugewiesen.

#### Polygone (Ways)

Die Koordinaten der Way-Nodes werden zu einem WKT-String `POLYGON((coords))` zusammengesetzt. Falls der erste und letzte Node nicht identisch sind, wird der Ring automatisch geschlossen.

#### Linienzüge (Ways)

Analog werden die Koordinaten als `LINESTRING(coords)` zusammengesetzt.

Alle Geometrien werden abschließend nach `GEOGRAPHY(type, 4326)` gecastet. Die Datenbankverbindung wird über `psycopg2` direkt aufgebaut (ohne ORM), da Bulk-Inserts so effizienter sind.

## 7.2 Unfallatlas-Datenerfassung

Die Erfassung der Unfalldaten (Geobasis NRW 2024) umfasst mehrere Schritte, da die Quelldaten in einem anderen Format und Koordinatensystem vorliegen.

### 7.2.1 Download und Extraktion

Die Daten werden als gezippte CSV-Datei von der Open-Data-Plattform des Landes Nordrhein-Westfalen heruntergeladen:

```
1 https://www.opengeodata.nrw.de/produkte/transport_verkehr/
2 unfallatlas/Unfallorte2024_EPSG25832_CSV.zip
```

### 7.2.2 Filterung nach Bremen

Die CSV-Datei enthält Unfalldaten für ganz Deutschland. Die Filterung auf Bremen erfolgt anhand des Bundeslandschlüssels:

```
1 # ULAND = 4 entspricht dem Bundesland Bremen
2 df = df[df["ULAND"] == 4]
```

Listing 7.1: Filterung auf Bremen

### 7.2.3 Koordinatentransformation

Die Unfalldaten liegen im projizierten Koordinatensystem EPSG:25832 (UTM Zone 32N) vor (EPSG Geodetic Parameter Registry 2024). Für die Speicherung in der PostGIS-Datenbank (EPSG:4326) ist eine Reprojektion erforderlich.

Das Skript erkennt automatisch die Koordinatenpalten (XGCSWGS84/YGCSWGS84 bzw. LINREFX/LINREF Y), konvertiert das deutsche Dezimalkomma-Format und reprojiziert die Daten mithilfe von GeoPandas (GeoPandas Contributors 2024) nach EPSG:4326.

### 7.2.4 Schweregrad-Mapping

Das Feld UKATEGORIE wird in lesbare Schweregrade konvertiert (siehe Tabelle 3.2 in Kapitel 3):  
1 → fatal, 2 → severe, 3 → minor.

### 7.2.5 Verfügbare Jahrgänge

Es stehen Daten für die Jahrgänge 2016–2024 zur Verfügung. Das Ingestion-Skript akzeptiert das gewünschte Jahr als Parameter und versucht, bei Nicht-Verfügbarkeit automatisch das nächstältere Jahr zu verwenden.

# 8 Backend-API

Das Backend ist als REST-konforme API mit dem Python-Framework FastAPI (Ramírez 2024a) implementiert. Es stellt die zentrale Schnittstelle zwischen Frontend und Datenbank dar. Alle Endpunkte können über die interaktive Swagger-Dokumentation unter <https://bremen-livability-backend.onrender.com/docs> erkundet werden. Für manuelle Tests steht im Repository unter `backend/Bremen_Livability_Index.postman_collection.json` eine Postman-Collection bereit, die alle Endpunkte mit Beispielenfragen abdeckt.

## 8.1 Endpunkte

Tabelle 8.1 zeigt alle verfügbaren HTTP-Endpunkte der API.

Tabelle 8.1: API-Endpunkte

Methode	Pfad	Beschreibung
GET	/	API-Metadaten (Version, Endpunktliste)
GET	/health	Datenbank-Konnektivitätsprüfung
POST	/analyze	<b>Kern-Endpunkt:</b> Berechnung des Livability Scores für Koordinaten mit optionalen Präferenzen
POST	/geocode	Adresssuche über Nominatim (OpenStreetMap Foundation 2024a)
GET	/preferences/defaults	Gibt Standardpräferenzen, Multiplikatoren und Faktoren zurück

## 8.2 Der /analyze-Endpunkt

Der zentrale Endpunkt nimmt eine Anfrage vom Typ `LocationRequest` entgegen und gibt eine Antwort vom Typ `LivabilityScoreResponse` zurück. Der Ablauf umfasst die folgenden Schritte:

1. **Validierung:** Pydantic validiert die Eingabekoordinaten ( $-90 \leq \text{lat} \leq 90, -180 \leq \text{lon} \leq 180$ ).
2. **Räumliche Abfragen:** Für jeden nicht-ausgeschlossenen Faktor wird eine PostGIS-ST\_DWithin-Abfrage ausgeführt, die alle Objekte innerhalb des faktorspezifischen Radius ermittelt.
3. **Score-Berechnung:** Der `LivabilityScorer` berechnet den Einzelscore jedes Faktors und aggregiert den Gesamtscore (siehe Kapitel 6).
4. **GeoJSON-Erzeugung:** Die nahen Objekte werden mit `ST_AsGeoJSON` in GeoJSON konvertiert und in der Antwort als `nearby_features` zurückgegeben.
5. **Zusammenfassung:** Ein menschenlesbarer Zusammenfassungstext wird generiert.

### 8.2.1 Request-Modell

```
1  {
2      "latitude": 53.0793,
3      "longitude": 8.8017,
4      "preferences": {
5          "greener": "high",
6          "airport": "excluded",
7          "noise": "low"
8      }
9 }
```

Listing 8.1: Beispiel-Request an /analyze

### 8.2.2 Response-Modell

Die Antwort enthält den `score` (0–100), den `base_score` (40.0), die angefragten Koordinaten, eine `factors`-Liste mit Aufschlüsselung aller Einzelfaktoren, `nearby_features` als GeoJSON-Objekte gruppiert nach Kategorie sowie eine textuelle `summary`.

```

1  {
2      "score": 72.5,
3      "base_score": 40.0,
4      "location": { "latitude": 53.0793, "longitude": 8.8017 },
5      "factors": [
6          { "factor": "greenery",           "value": 18.0, "impact": "positive",
7              "description": "Parks and trees nearby" },
8          { "factor": "public_transport", "value": 7.0,   "impact": "positive",
9              "description": "Transit stops nearby" },
10         { "factor": "major_roads",     "value": -6.0,  "impact": "negative",
11             "description": "Major road within 60m" }
12     ],
13     "nearby_features": {
14         "parks": [
15             {
16                 "id": 42,
17                 "name": "Buergerpark",
18                 "type": "park",
19                 "subtype": null,
20                 "distance": 134.2,
21                 "geometry": { "type": "Point",
22                               "coordinates": [8.8021, 53.0797] }
23             }
24         ]
25     },
26     "summary": "Excellent amenities. Good transit access."
27 }
```

Listing 8.2: Beispiel-Response von `/analyze`

## 8.3 Dependency Injection und CORS

FastAPI nutzt Dependency Injection für Datenbankverbindungen: Die Funktion `get_session()` liefert eine SQLModel-Session als Generator, sodass jede Anfrage eine eigene Session erhält. Da Frontend und Backend auf unterschiedlichen Domains gehostet werden, wird eine [Cross-Origin Resource Sharing \(CORS\)](#)-Middleware konfiguriert, deren erlaubte Origins über die Umgebungsvariable `CORS_ORIGINS` gesteuert werden.

## 8.4 Geokodierung

Der `/geocode`-Endpunkt delegiert die Adresssuche an den `GeocodeService`, der intern die Nominatim-[API](#) (OpenStreetMap Foundation 2024a) anspricht. Nominatim ist ein freier Geokodierungsdienst, der [OSM](#)-Daten nutzt und keine API-Schlüssel erfordert. Die Ergebnisse enthalten Koordinaten, eine formatierte Adresse, einen Typ und einen Relevanzwert (`importance`).

# 9 Frontend

Das Frontend ist als plattformübergreifende Anwendung mit dem UI-Framework Flutter (Google LLC 2024b) implementiert und unterstützt Web, iOS, Android, macOS, Windows und Linux.

## 9.1 Architektur und State Management

Die Anwendung folgt dem [Business Logic Component \(BLoC\)](#)-Entwurfsmuster (vgl. Abschnitt 2.5), das die Geschäftslogik vollständig von der Darstellungsschicht trennt. Die Architektur gliedert sich in die folgenden Module:

### `lib/core/`

Querschnittskomponenten: `ApiService` (HTTP-Client via Dio), `DeepLinkService`, Theme-Definitionen (`AppTheme`, `AppColors`, `AppTextStyles`) und wiederverwendbare Widgets.

### `lib/features/auth/`

Authentifizierungslogik: `AuthBloc`, `AuthService`, Login-Screens.

### `lib/features/map/`

Kern-Feature: `MapBloc` (Karteninteraktionen), `MapScreen`, `ScoreCardView`, `FloatingearchBar`.

### `lib/features/favorites/`

Favoritenverwaltung: `FavoritesBloc`, Datenmodelle.

### `lib/features/preferences/`

Nutzerpräferenzen: `PreferencesBloc`, `PreferencesScreen`, `PreferencesService`.

### `lib/features/onboarding/`

Startbildschirm.

Alle Events und States werden mithilfe des Code-Generators `Freezed` als *Sealed Unions* definiert. Dies erzwingt typsichere, vollständige Pattern-Matching-Behandlung in der UI und verhindert undefinierte Zustände.

## 9.2 Kartenansicht

Die Kartenansicht bildet die zentrale Benutzeroberfläche. Sie basiert auf dem Widget `FlutterMap` (flutter\_map Contributors 2024) mit **CartoDB Voyager** Tiles – einem schlichten Kartenstil ohne störende POI-Beschriftungen.

- **Startzentrum:** 53,0793° N, 8,8017° E (Bremen Innenstadt)
- **Startzoom:** 13
- **Interaktion:** Ein Tipp auf die Karte löst ein `MapTapped`-Event aus, das den `MapBloc` veranlasst, den `ApiService` anzufragen. Der berechnete Score und die nahen Features werden anschließend als Marker und Score-Karte dargestellt.

Jede Feature-Kategorie erhält einen eigenen Marker mit einer standardisierten Farbpalette aus `AppColors`, sodass der Nutzer die Art der nahen Einrichtungen visuell unterscheiden kann.

## 9.3 Benutzeroberfläche – Liquid Glass Design

Das UI-Design folgt einem *Liquid Glass*-Konzept: Die Karte nimmt den gesamten Bildschirm ein, während alle interaktiven Elemente (Suchleiste, Score-Karte, Präferenzen) als halbtransparente, gefroste Glasflächen über der Karte schweben. Optische Effekte wie leichte Vergrößerung, Verzerrung und haptisches Feedback auf Mobilgeräten runden das Interaktionserlebnis ab.

## 9.4 Authentifizierung

Die Authentifizierung wird über Firebase Authentication (Google LLC 2024a) abgewickelt und unterstützt mehrere Anmeldemethoden:

Tabelle 9.1: Unterstützte Anmeldemethoden nach Plattform

Plattform	Anmeldemethoden
Web, iOS, Android	Google, GitHub, E-Mail-Magic-Link, Anonym
macOS, Windows, Linux	Nur als Guest (Firebase Auth wird übersprungen)

Abbildung 9.1 zeigt den Anmeldebildschirm mit den verfügbaren Authentifizierungsmethoden.

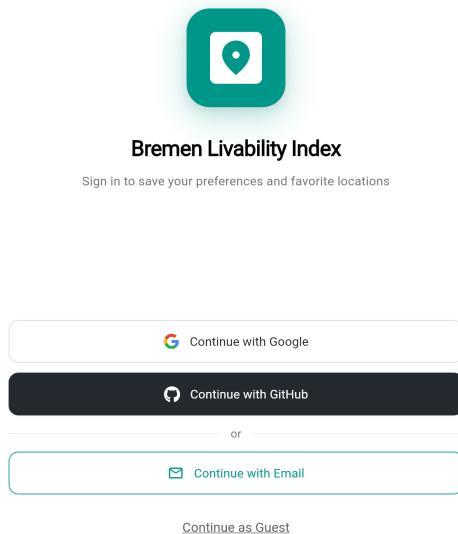


Abbildung 9.1: Anmeldebildschirm mit den verfügbaren Authentifizierungsmethoden

Auf Desktop-Plattformen wird die Firebase-Authentifizierung vollständig umgangen, da macOS-Keychains durch Sandboxing eingeschränkt sind und Windows/Linux keine nativen OAuth-Desktop-Flows unterstützen. Stattdessen wird ein lokales `AppUser.guest()`-Objekt ohne Firebase-Interaktion erzeugt.

### 9.4.1 Cross-Device E-Mail-Link-Flow

Wenn ein Nutzer einen Magic Link auf einem anderen Gerät öffnet als dem, auf dem er die Anmeldung initiiert hat, ist die E-Mail-Adresse nicht im lokalen Speicher hinterlegt. In diesem Fall erkennt der `DeepLinkService` den `oobCode`-Parameter in der URL, und der `AuthBloc` navigiert den Nutzer zu einem `EmailLinkPromptScreen`, auf dem er seine E-Mail-Adresse erneut eingeben kann.

## 9.5 Favoritenverwaltung

Nutzer können analysierte Standorte als Favoriten speichern. Die Datenhaltung ist zweistufig:

- **Angemeldete Nutzer:** Synchronisation über Firebase Firestore.
- **Gäste:** Lokale Speicherung über `SharedPreferences` (Key-Value-Store des Geräts).

Der FavoritesBloc abstrahiert die Speicherstrategie und stellt eine einheitliche Schnittstelle für das UI bereit.

## 9.6 Nutzerpräferenzen

Der PreferencesScreen erlaubt es dem Nutzer, für jeden der 20 Bewertungsfaktoren eine Wichtigkeit (`excluded`, `low`, `medium`, `high`) festzulegen. Jede Einstellung entspricht dem `ImportanceLevel`-Enum des Backends und wird beim nächsten `/analyze`-Aufruf als `preferences`-Objekt im Request-Body mitgesendet, wodurch der Score unmittelbar auf die persönlichen Prioritäten des Nutzers reagiert.

Die Persistenz der Präferenzen ist – analog zur Favoritenverwaltung (vgl. Abschnitt 9.5) – zweistufig: Gäste speichern ihre Einstellungen lokal via `SharedPreferences`; angemeldete Nutzer synchronisieren sie zusätzlich über Firestore (`users/{user_id}/preferences/factor_settings`), geräteübergreifend per Echtzeit-Stream, mit automatischem Löschen beim Abmelden.

Der PreferencesBloc koordiniert Lade- und Speichervorgänge und stellt den aktuellen UserPreferences-Zustand bereit, auf den MapBloc bei jeder Score-Anfrage zugreift.

## 9.7 Adresssuche

Die FloatingSearchBar ermöglicht die Suche nach Adressen. Eingaben werden über den ApiService an den `/geocode`-Endpunkt des Backends weitergeleitet, der wiederum die Nominatim-API abfragt. Die Suchergebnisse werden als Drop-down-Liste angezeigt (Abbildung 9.2); bei Auswahl wird die Karte zur entsprechenden Koordinate navigiert und automatisch eine Score-Berechnung ausgelöst.



Abbildung 9.2: Adresssuche mit Geokodierungsergebnissen für die Hochschule Bremen

# 10 Testing und Deployment

Dieses Kapitel beschreibt die Qualitätssicherung durch automatisierte Tests sowie die [Continuous Integration / Continuous Deployment \(CI/CD\)](#)-Pipeline und die Deployment-Strategie.

## 10.1 Backend-Tests

Das Backend wird mit [pytest](#) (pytest Contributors 2024) getestet. Die Testdateien befinden sich im Verzeichnis `backend/tests/` und decken vier Bereiche ab:

Tabelle 10.1: Backend-Testdateien und Testumfang

Datei	Tests	Schwerpunkt
<code>test_scoring.py</code>	58	Alle Scoring-Funktionen: Grenzwerte, Randfälle (0, 1, viele Objekte), logarithmische Skalierung, binäre Faktoren, Importance-Multiplikatoren
<code>test_api.py</code>	11	FastAPI-Endpunkt-Tests mit <code>TestClient</code>
<code>test_database.py</code>	5	Datenbank-Verbindungstests
<code>test_main.py</code>	9	Integrationstests der Hauptanwendung

Die Scoring-Tests sind besonders umfangreich, da der Bewertungsalgorithmus das Kernstück der Anwendung bildet. Jede der 20 Berechnungsfunktionen wird mit mindestens den folgenden Szenarien getestet:

- **Leereingabe:**  $n = 0$  muss Score 0,0 ergeben.
- **Einzelner Treffer:** Korrektheit der Formeln bei  $n = 1$ .
- **Sättigungsfall:** Sehr hohe  $n$ -Werte dürfen das jeweilige Maximum nicht überschreiten.
- **Binäre Faktoren:** `True/False` muss exakt die definierte Strafe bzw. 0,0 ergeben.

### 10.1.1 Code Coverage

Die Testabdeckung wird mit `pytest-cov` gemessen und an [Codecov](#) (Codecov Inc. 2024) übermittelt. Das Projekt erreicht eine Abdeckung von über 90 %.

## 10.2 Frontend-Tests

Die Flutter-Tests befinden sich in `frontend/bli/test/` und verwenden die Bibliotheken `bloc_test`, `mockito` und `mocktail` für BLoC-Tests mit gemockten Abhängigkeiten. Getestet werden insbesondere:

- **BLoC-Logik:** Korrekte Zustandsübergänge bei Events (z. B. `MapTapped` → `MapLoading` → `MapLoaded`)
- **Fehlerbehandlung:** Netzwerkfehler und ungültige Serverantworten
- **Authentifizierung:** Login-Flows für verschiedene Provider

## 10.3 Continuous Integration

Die [CI/CD](#)-Pipeline basiert auf [GitHub Actions](#) (GitHub Inc. 2024) und umfasst drei Workflows:

Tabelle 10.2: GitHub-Actions-Workflows

Workflow	Beschreibung
<code>backend-tests.yml</code>	Führt <code>pytest</code> bei jedem Push auf das Backend aus; übermittelt Coverage an Codecov
<code>frontend-tests.yml</code>	Führt <code>flutter test</code> bei jedem Push auf das Frontend aus
<code>build-release.yml</code>	Baut APK- (Android), Windows-, macOS- und Linux-Binaries und erstellt automatisch GitHub Releases

## 10.4 Deployment

Die Infrastruktur (Render.com, Neon.tech, Firebase) ist in Kapitel 4.4 beschrieben. Dieser Abschnitt fokussiert auf den automatisierten Ablauf vom Commit bis zur Produktivschaltung.

Bei jedem Push oder Merge auf `master` werden vier unabhängige Prozesse parallel ausgelöst:

1. **Backend-Tests:** `backend-tests.yml` führt `pytest` mit einer PostGIS-Service-Instanz aus und übermittelt die Coverage an Codecov (Flag: `backend`). Wird nur bei Änderungen unter `backend/` getriggert.
2. **Frontend-Tests:** `frontend-tests.yml` führt `flutter test --coverage` und `flutter analyze` aus und übermittelt die Coverage ebenfalls an Codecov (Flag: `frontend`). Wird nur bei Änderungen unter `frontend/` getriggert.
3. **Web-Deployment:** Render.com erkennt den Merge über einen GitHub-Webhook. Dank der `rootDir`-Konfiguration in `render.yaml` wird nur der Service neu gebaut, dessen Verzeichnis von der Änderung betroffen ist – unabhängig von den Testergebnissen:
  - **Backend:** Docker-Rebuild des FastAPI-Servers im Verzeichnis `backend/` (Region: Frankfurt).
  - **Frontend:** Statische Site, gebaut durch `render_build.sh` (`flutter build web --release`); das Ergebnis in `build/web` wird als Static Site mit SPA-Rewrite veröffentlicht.
4. **Native Builds:** `build-release.yml` wird durch den *erfolgreichen* Abschluss von `frontend-tests.yml` getriggert (`workflow_run`) und baut APK- (Android), Windows-, macOS- und Linux-Binaries, die als einheitliches GitHub Release veröffentlicht werden (vgl. Tabelle 10.2).

Abbildung 10.1 fasst den Ablauf visuell zusammen.

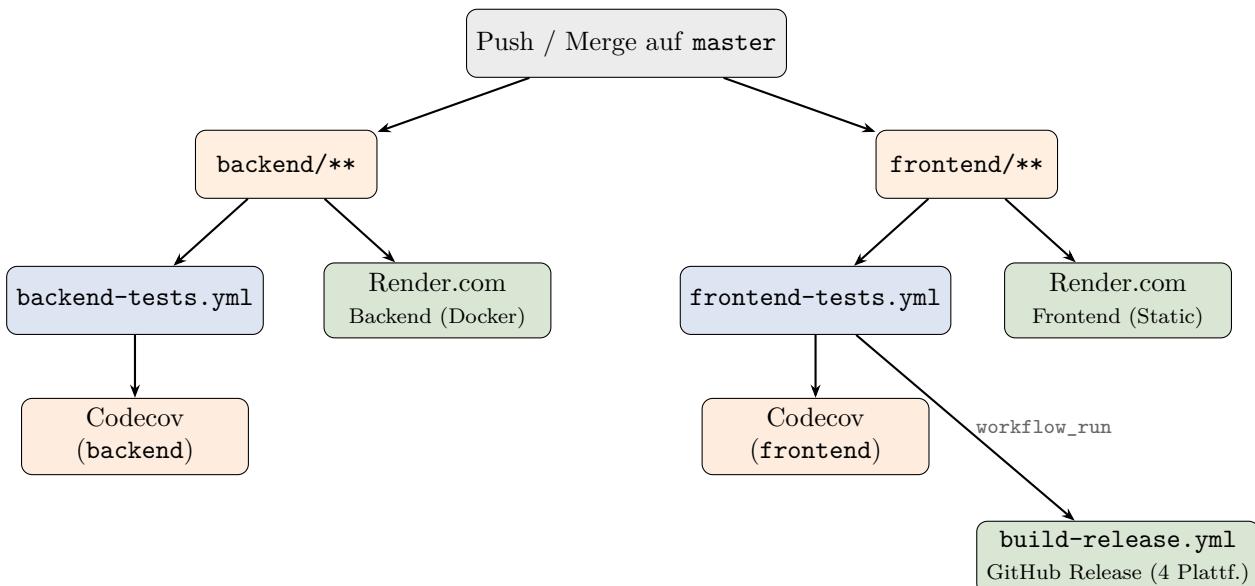


Abbildung 10.1: CI/CD-Pipeline: Pfadbasierte Auslösung nach dem Merge

# 11 Ergebnisse und Diskussion

Dieses Kapitel präsentiert exemplarische Ergebnisse des Bremen Livability Index und diskutiert Stärken, Schwächen sowie Limitierungen der gewählten Methodik.

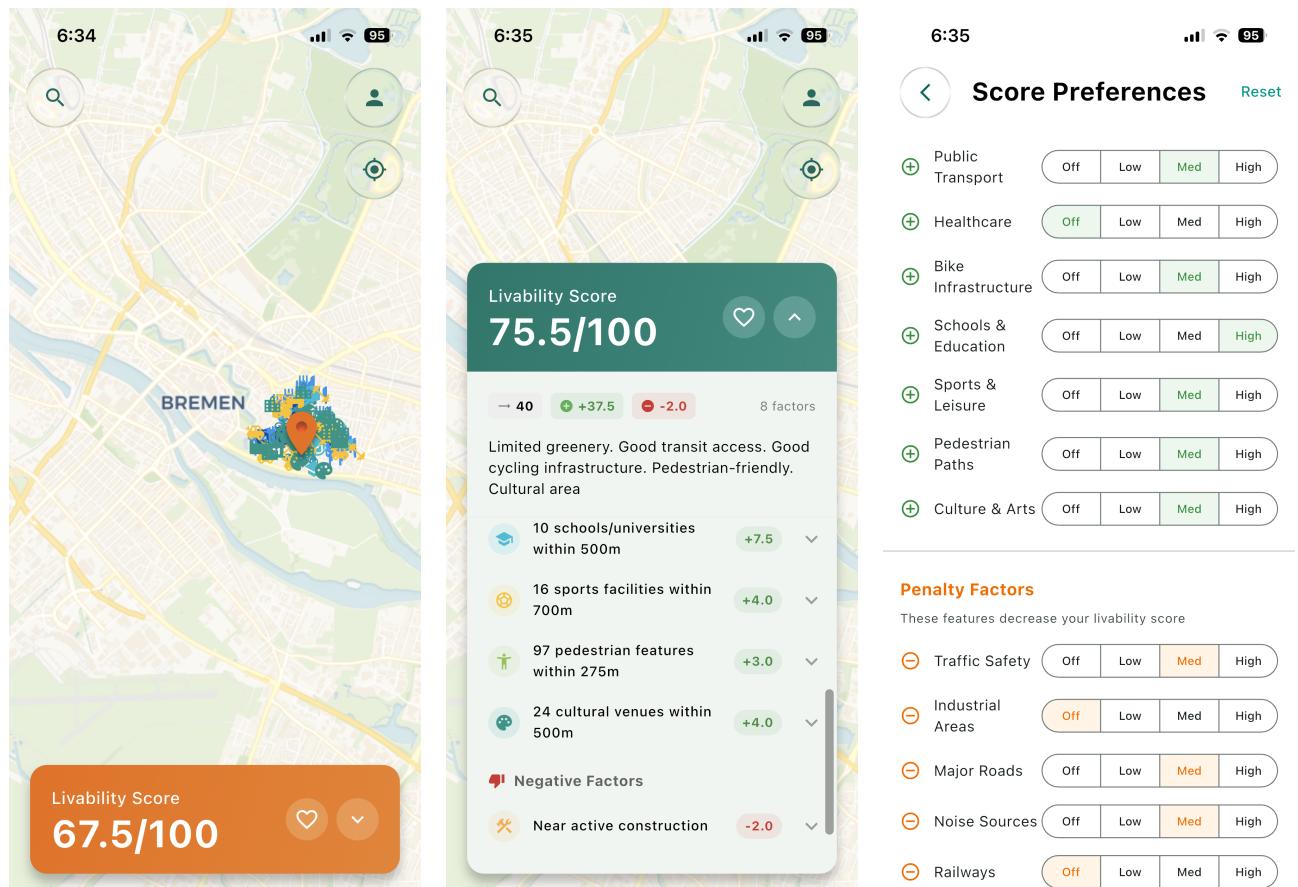
## 11.1 Exemplarische Standortbewertungen

Um die Funktionalität des Systems zu demonstrieren, wurden Livability Scores für fünf repräsentative Bremer Standorte mit den Standardpräferenzen (`medium` für alle Faktoren) berechnet:

Tabelle 11.1: Exemplarische Livability Scores für Bremer Standorte

Standort	Lat.	Lon.	Score
Marktplatz (Innenstadt)	53.076	8.808	~ 75 – 85
Bürgerpark (Schwachhausen)	53.092	8.822	~ 70 – 80
Universität Bremen	53.106	8.853	~ 65 – 75
Industriehafen	53.120	8.760	~ 25 – 35
Flughafen Bremen	53.047	8.787	~ 20 – 30

*Hinweis: Die exakten Scores variieren je nach aktuellem Datenbestand in der Datenbank und dem gewählten Abfragepunkt.*



(a) Kartenansicht mit GeoJSON-Markern

(b) Score-Aufschlüsselung nach Faktoren

(c) Individuelle Präferenzeinstellung

Abbildung 11.1: Mobile Ansichten der Anwendung (iOS)

Abbildung 11.2 zeigt die Webanwendung im Desktop-Browser mit Kartenansicht, farbcodierten GeoJSON-Markern und der Score-Aufschlüsselung nach positiven und negativen Faktoren.

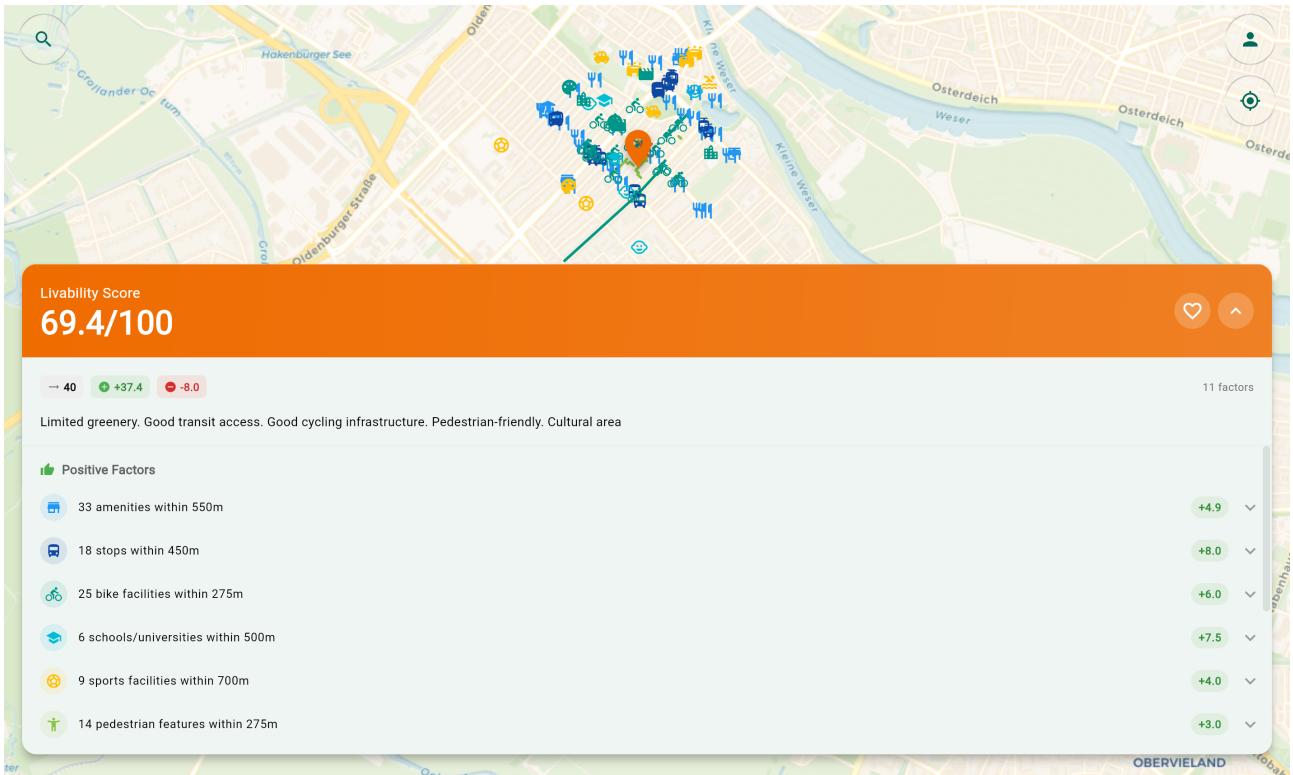


Abbildung 11.2: Webanwendung im Desktop-Browser: Kartenansicht mit Score-Aufschlüsselung

### 11.1.1 Interpretation

Die Ergebnisse zeigen ein plausibles Muster:

- **Innenstadt und Parks:** Hohe Scores aufgrund dichter Nahversorgung, guter ÖPNV-Anbindung und vieler Grünflächen.
- **Universität:** Guter Score durch Bildungseinrichtungen und ÖPNV, leicht geringer wegen weniger Nahversorgung.
- **Industriehafen:** Niedriger Score durch starke negative Faktoren (Industriegebiete, Hauptstraßen, Eisenbahn) bei gleichzeitig geringer positiver Infrastruktur.
- **Flughafen:** Niedrigster Score durch die Kombination aus Flughafen-Nähe (7 Strafpunkte), Hauptstraßen und fehlender Wohninfrastruktur.

## 11.2 Stärken des Systems

Das System bietet feinräumige Auflösung (Scores für jeden Punkt statt ganzer Städte), transparente Faktoraufschlüsselung, Personalisierbarkeit durch dynamische Gewichtung, Echtzeitberechnung dank GiST-Indizes, ausschließliche Nutzung offener Daten und plattformübergreifende Verfügbarkeit.

## 11.3 Limitierungen

Das System ist räumlich auf Bremen beschränkt. Die Vollständigkeit der OSM-Daten variiert je nach Kategorie (vgl. Abschnitt 3.1.3). Echtzeitdaten (Luftqualität, Lärmpegel) fließen nicht ein. Die Basisgewichtungen basieren auf Erfahrungswerten statt empirischen Studien. Binäre Faktoren differenzieren nicht nach Größe/Intensität. Unfalldaten werden nur für ein Jahr importiert.

## 11.4 Verbesserungspotenzial

Auf Basis der identifizierten Limitierungen lassen sich mehrere Weiterentwicklungen ableiten:

- **Skalierung:** Generalisierung der Bounding-Box-Konfiguration und Erweiterung auf weitere deutsche Städte oder den gesamten DACH-Raum.
- **Echtzeitdaten:** Integration von Echtzeit-APIs für Luftqualität (z. B. Umweltbundesamt), Lärmkarten und ÖPNV-Verspätungen.
- **Machine Learning:** Erlernung optimaler Gewichtungen aus Nutzerfeedback (implizit durch Favoriten, explizit durch Bewertungen).
- **Differenzierte negative Faktoren:** Abstufung der Strafpunkte basierend auf der Distanz zum negativen Objekt (*distance decay*).
- **Aggregierte Unfalldaten:** Import und Mittelung über mehrere Jahrgänge zur Glättung von Ausreißern.
- **Offene API:** Bereitstellung einer öffentlichen API für Drittanwendungen und Forschungszwecke.

# 12 Fazit und Ausblick

## 12.1 Zusammenfassung

Im Rahmen des Moduls *Geodatenverarbeitung* an der Hochschule Bremen wurde mit dem **Bremen Livability Index** ein vollständiges Geoinformationssystem konzipiert, implementiert und als produktionsstaugliche Anwendung bereitgestellt. Ausgangspunkt war der Wunsch, heterogene, frei verfügbare Geodaten – aus OpenStreetMap und dem Unfallatlas – zu einem einheitlichen, feinräumigen Lebensqualitätsindex zu verdichten. Über eine vollständig automatisierte Ingestion-Pipeline wurden mehr als 60.000 Objekte aus 20 thematischen Kategorien in eine PostGIS-Datenbank geladen und mit GiST-Indizes für Echtzeit-Radiusabfragen optimiert. Das FastAPI-Backend berechnet Scores innerhalb von Millisekunden und liefert die umliegenden **GIS**-Features als GeoJSON zurück. Die Flutter-Applikation stellt diese Ergebnisse plattformübergreifend – Web, iOS, Android, Desktop – auf einer interaktiven Karte dar und erlaubt individuelle Präferenzgewichtung. Authentifizierung und Favoritensynchronisation werden über Firebase abgewickelt, während die domänenspezifische Geo-Logik ausschließlich im Backend verbleibt. Das Projekt zeigt, dass sich aus frei verfügbaren Geodaten mit vertretbarem Aufwand ein praxistaugliches, feinräumiges Analysewerkzeug realisieren lässt.

## 12.2 Beantwortung der Zielsetzung

Die zentrale Fragestellung – ob sich aus frei verfügbaren Geodaten ein aussagekräftiger, feinräumiger Lebensqualitätsindex für Bremen ableiten lässt – kann positiv beantwortet werden. Alle fünf in Kapitel 1 formulierten Projektziele wurden erreicht: (1) Über 60.000 Geodatenobjekte werden automatisiert erfasst und aktualisiert; (2) der Livability Score (0–100) wird für beliebige Bremer Koordinaten in Echtzeit berechnet; (3) die Faktoraufschlüsselung macht positive und negative Einflüsse transparent nachvollziehbar; (4) die Kartenanwendung visualisiert die umgebenden **GIS**-Features interaktiv; und (5) Nutzer können die Gewichtung der Faktoren ihren persönlichen Präferenzen anpassen. Die exemplarischen Standortbewertungen (Kapitel 11) bestätigen, dass das Modell plausible Ergebnisse liefert: urban dichte, gut versorgte Standorte erzielen deutlich höhere Scores als periphere Industrie- oder Verkehrsflächen.

## 12.3 Ausblick

Konkrete technische Erweiterungsmöglichkeiten – von der Skalierung auf weitere Städte über Echtzeitdaten bis hin zu datengetriebener Kalibrierung – wurden bereits in Abschnitt 11.4 identifiziert. Darüber hinaus ergeben sich weiterreichende Perspektiven.

Der gewählte Architekturansatz – automatisierte OSM-Ingestion, konfigurierbares Scoring-Schema und plattformübergreifendes Frontend – ist nicht auf das Thema Lebensqualität beschränkt. Dieselbe Pipeline ließe sich mit angepassten Kategorien und Gewichtungen auf verwandte Fragestellungen übertragen, etwa die Bewertung von Gewerbestandorten, barrierefreier Infrastruktur oder Schulwegqualität.

Für eine institutionelle Nutzung in der Stadtplanung oder Immobilienwirtschaft wäre eine empirische Validierung der Gewichtungen unerlässlich – beispielsweise durch Korrelation der berechneten Scores mit Mietpreisspiegeln, Zufriedenheitsbefragungen oder soziodemographischen Indikatoren. Die Bereitstellung einer öffentlichen **API** könnte zudem die Integration in bestehende Geoinformations- und Planungswerkzeuge erleichtern und den Anwendungskreis über das akademische Umfeld hinaus erweitern.

# Literaturverzeichnis

- Angelov, F. (2024). *flutter\_bloc – Flutter Widgets for BLoC pattern*. URL: [https://pub.dev/packages/flutter\\_bloc](https://pub.dev/packages/flutter_bloc) (besucht am 01.12.2025).
- Codecov Inc. (2024). *Codecov Documentation*. URL: <https://docs codecov com/> (besucht am 01.12.2025).
- Docker Inc. (2024). *Docker Documentation*. URL: <https://docs.docker.com/> (besucht am 01.12.2025).
- Economist Intelligence Unit (2024). „The Global Liveability Index 2024“. In: *EIU Report*. URL: <https://www.eiu.com/n/campaigns/global-liveability-index-2024/>.
- EPSG Geodetic Parameter Registry (2024). *EPSG:25832 – ETRS89 / UTM zone 32N*. URL: <https://epsg.io/25832> (besucht am 01.12.2025).
- flutter\_map Contributors (2024). *flutter\_map – A versatile mapping package for Flutter*. URL: [https://pub.dev/packages/flutter\\_map](https://pub.dev/packages/flutter_map) (besucht am 01.12.2025).
- GeoAlchemy2 Contributors (2024). *GeoAlchemy2 Documentation*. URL: <https://geoalchemy-2.readthedocs.io/> (besucht am 01.12.2025).
- Geobasis NRW (2024). *Unfallatlas – Open Data Download*. URL: [https://www.opengeodata.nrw.de/produkte/transport\\_verkehr/unfallatlas/](https://www.opengeodata.nrw.de/produkte/transport_verkehr/unfallatlas/) (besucht am 01.12.2025).
- GeoPandas Contributors (2024). *GeoPandas – Geographic pandas extensions*. URL: <https://geopandas.org/> (besucht am 01.12.2025).
- GitHub Inc. (2024). *GitHub Actions Documentation*. URL: <https://docs.github.com/en/actions> (besucht am 01.12.2025).
- Google LLC (2024a). *Firebase Documentation*. URL: <https://firebase.google.com/docs> (besucht am 01.12.2025).
- Google LLC (2024b). *Flutter Documentation*. URL: <https://docs.flutter.dev/> (besucht am 01.12.2025).
- Mercer LLC (2019). „Quality of Living City Ranking“. In: *Mercer Global Report*. URL: <https://www.mercer.com/insights/quality-of-living/>.
- Neon Inc. (2024). *Neon – Serverless Postgres*. URL: <https://neon.tech/docs> (besucht am 01.12.2025).
- OpenStreetMap Foundation (2012). *Open Data Commons Open Database License (ODbL) v1.0*. URL: <https://opendatacommons.org/licenses/odbl/1-0/> (besucht am 01.12.2025).
- OpenStreetMap Foundation (2024a). *Nominatim – Geocoding with OpenStreetMap data*. URL: <https://nominatim.org/> (besucht am 01.12.2025).
- OpenStreetMap Foundation (2024b). *OpenStreetMap Wiki*. URL: <https://wiki.openstreetmap.org/> (besucht am 01.12.2025).
- OpenStreetMap Wiki Contributors (2024). *Overpass API*. URL: [https://wiki.openstreetmap.org/wiki/Overpass\\_API](https://wiki.openstreetmap.org/wiki/Overpass_API) (besucht am 01.12.2025).
- PhiBo (2024). *overpy – Python Wrapper for the Overpass API*. URL: <https://github.com/DinoTools/python-overpy> (besucht am 01.12.2025).
- PostGIS Project Steering Committee (2024). *PostGIS 3.4 Documentation*. URL: <https://postgis.net/docs/> (besucht am 01.12.2025).
- pytest Contributors (2024). *pytest Documentation*. URL: <https://docs.pytest.org/> (besucht am 01.12.2025).
- Ramírez, S. (2024a). *FastAPI Documentation*. URL: <https://fastapi.tiangolo.com/> (besucht am 01.12.2025).
- Ramírez, S. (2024b). *SQLModel Documentation*. URL: <https://sqlmodel.tiangolo.com/> (besucht am 01.12.2025).
- Render Inc. (2024). *Render Documentation*. URL: <https://docs.render.com/> (besucht am 01.12.2025).
- Statistisches Bundesamt (2024). *Unfallatlas – Kartenanwendung der regionalstatistischen Ergebnisse der Statistik der Straßenverkehrsunfälle*. URL: <https://unfallatlas.statistikportal.de/> (besucht am 01.12.2025).
- The PostgreSQL Global Development Group (2024). *PostgreSQL 16 Documentation*. URL: <https://www.postgresql.org/docs/16/> (besucht am 01.12.2025).