



# 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

Bremen, 20. Februar 2026

# Inhaltsverzeichnis

<b>1</b>	<b>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</b>	<b>Grundlagen</b>	<b>2</b>
2.1	Lebensqualität und urbane Indizes . . . . .	2
2.2	Geoinformationssysteme und räumliche Datenbanken . . . . .	2
2.3	Koordinatenreferenzsysteme . . . . .	2
2.4	REST-APIs und das Client-Server-Modell . . . . .	3
2.5	Das BLoC-Entwurfsmuster . . . . .	3
<b>3</b>	<b>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</b>	<b>Systemarchitektur</b>	<b>6</b>
4.1	Architekturüberblick . . . . .	6
4.2	Technologiestack . . . . .	6
4.3	Projektstruktur . . . . .	6
4.4	Deployment-Architektur . . . . .	6
<b>5</b>	<b>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</b>	<b>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</b>	<b>Datenerfassung</b>	<b>12</b>
7.1	OSM-Datenerfassung . . . . .	12
7.1.1	Ablauf . . . . .	12
7.1.2	Geometriekonvertierung . . . . .	12
7.2	Unfallatlas-Datenerfassung . . . . .	12
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</b>	<b>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</b>	<b>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	Adresssuche . . . . .	17
<b>10</b>	<b>Testing und Deployment</b>	<b>18</b>
10.1	Backend-Tests . . . . .	18
10.1.1	Code Coverage . . . . .	18
10.2	Frontend-Tests . . . . .	18
10.3	Continuous Integration . . . . .	18
10.4	Deployment . . . . .	19
<b>11</b>	<b>Ergebnisse und Diskussion</b>	<b>20</b>
11.1	Exemplarische Standortbewertungen . . . . .	20
11.1.1	Interpretation . . . . .	20
11.2	Stärken des Systems . . . . .	20
11.3	Limitierungen . . . . .	20
11.4	Verbesserungspotenzial . . . . .	21
<b>12</b>	<b>Fazit und Ausblick</b>	<b>22</b>
12.1	Zusammenfassung . . . . .	22
12.2	Beantwortung der Zielsetzung . . . . .	22
12.3	Ausblick . . . . .	22
	<b>Literaturverzeichnis</b>	<b>23</b>



# 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 GeodatenverarbeitungsPipeline, 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 produktionsstaugliche Webanwendung mit Flutter-Frontend und FastAPI-Backend realisiert. Der vollständige Quellcode ist öffentlich auf GitHub verfügbar:

<https://github.com/Milad9A/Bremen-Livability-Index>

Die Webanwendung ist erreichbar unter:

<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

Kapitel 2 führt in die theoretischen Grundlagen der Geodatenverarbeitung und bestehender Lebensqualitätsindizes ein. Kapitel 3 beschreibt die verwendeten Datenquellen im Detail. Die Systemarchitektur wird in Kapitel 4 vorgestellt, gefolgt vom Datenbankdesign (Kapitel 5) und der Bewertungsmethodik (Kapitel 6). Kapitel 7 erläutert die automatisierte Datenerfassung. Die Implementierung von Backend-API (Kapitel 8) und Frontend (Kapitel 9) wird anschließend beschrieben. Kapitel 10 behandelt Testing und Deployment. Abschließend werden in Kapitel 11 die Ergebnisse diskutiert und in Kapitel 12 ein Fazit sowie ein Ausblick formuliert.

## 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 Entfernungsberechnung 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 Koordinatenreferenzsysteme

Im Projekt kommen zwei **Coordinate Reference System (Koordinatenreferenzsystem)** zum Einsatz:

Tabelle 2.1: Verwendete Koordinatenreferenzsysteme

EPSG-Code	Bezeichnung	Typ	Verwendung
4326	<b>WGS 84</b>	Geographisch	Primäres <b>CRS</b> der Datenbank, <b>API</b> -Ein-/Ausgabe, <b>OSM</b> -Daten
25832	ETRS89 / <b>UTM</b> Zone 32N	Projiziert	Quellformat der Unfallatlas-Daten; wird bei der Datenerfassung nach EPSG:4326 reprojiziert

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).

## 2.5 Das BLoC-Entwurfsmuster

Das [Business Logic Component \(BLoC\)](#)-Entwurfsmuster (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.

## 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 [dl-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	<b>fatal</b>	Unfall mit Getöteten
2	<b>severe</b>	Unfall mit Schwerverletzten
3	<b>minor</b>	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 Koordinatenspalten (XGCSWGS84/YGCSWGS84 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 beschreibt die zentralen Komponenten und deren Zusammenspiel. Durch die Trennung der Schichten lassen sich Backend und Frontend unabhängig voneinander entwickeln, testen und deployen.

**Architekturdiagramm (textuell)**

```
Flutter-App (Web / iOS / Android / Desktop)
↓ HTTP/JSON
FastAPI-Backend (Docker-Container auf Render.com)
↓ SQL / PostGIS-Queries
PostgreSQL 16 + PostGIS 3.4 (Neon.tech, Serverless)
```

Nebenkomponenten:

- Firebase Auth → Authentifizierung
- Firebase Firestore → Favoritensynchronisation
- Nominatim API → Adressgeokodierung
- Overpass API → OSM-Datenerfassung (offline)

Abbildung 4.1: Überblick der Systemarchitektur

### 4.2 Technologiestack

### 4.3 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.

### 4.4 Deployment-Architektur

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

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)
<b>ORM</b>	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	<b>BLoC</b> -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)

- **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:** 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.
- **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). 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 drei 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`
3. **Benutzerverwaltung** (2 Tabellen): `users`, `favorite_addresses`

Jede Geo-Tabelle folgt einem einheitlichen Aufbau:

Tabelle 5.1: Gemeinsames Spaltenschema der Geo-Tabellen

Spalte	Datentyp	Beschreibung
<code>id</code>	<code>SERIAL PRIMARY KEY</code>	Auto-Inkrement-ID
<code>osm_id</code>	<code>BIGINT</code>	OpenStreetMap-Objekt-ID (entfällt bei Unfalldaten)
<code>name</code>	<code>TEXT</code>	Bezeichnung (optional)
<code>geometry</code>	<code>GEOGRAPHY(type, 4326)</code>	Räumliches Objekt im <a href="#">WGS 84</a> -System
<code>created_at</code>	<code>TIMESTAMP</code>	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`. Die Benutzertabellen speichern Firebase-UIDs, E-Mail-Adressen und Favoriten-Koordinaten.

## 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 Tabellen erben von einer gemeinsamen Basisklasse `GISBase` mit der Konfiguration `arbitrary_types_allowed = True`.

## 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  $f(n) = \min(f_{\text{max}}, \ln(1+n) \cdot k)$ , die übrigen lineare Skalierung  $f(n) = \min(f_{\text{max}}, n \cdot k)$ .

### 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

Fünf Bewertungsbänder ordnen den Score ein: 80–100 *Hervorragend*, 60–79 *Gut*, 40–59 *Durchschnittlich*, 20–39 *Unterdurchschnittlich*, 0–19 *Schlecht*. Der Basiswert 40 stuft infrastrukturfreie Standorte als „durchschnittlich“ ein.

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.

# 7 Datenerfassung

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

## 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:

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

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 Koordinatenspalten (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.

## 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 <a href="#">2024a</a> )
GET	/preferences/defaults	Gibt Standardpräferenzen, Multiplikatoren und Faktoren zurück
POST	/users	Benutzer anlegen oder aktualisieren
GET	/users/{user_id}/favorites	Favoriten eines Benutzers abrufen
POST	/users/{user_id}/favorites	Favorit hinzufügen
DELETE	/users/{user_id}/favorites/{fav_id}	Favorit löschen

## 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     "greenery": "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`.

## 8.3 Dependency Injection und CORS

FastAPI nutzt Dependency Injection für Datenbankverbindungen: Die Funktion `get_session()` liefert eine `SQLModelSession` 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**, **FloatingSearchBar**.

#### **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, gefrorenete 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 Gast (Firebase Auth wird übersprungen)

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 und das Backend (`/users/{user_id}/favorites`).
- **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 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; bei Auswahl wird die Karte zur entsprechenden Koordinate navigiert und automatisch eine Score-Berechnung ausgelöst.

# 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>	–	FastAPI-Endpunkt-Tests mit <code>TestClient</code>
<code>test_database.py</code>	–	Datenbank-Verbindungstests
<code>test_main.py</code>	–	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

Durch die Aufteilung in drei separate Workflows können Backend- und Frontend-Tests parallel und unabhängig voneinander ausgeführt werden.

## 10.4 Deployment

Das Deployment erfolgt über die deklarative Datei `render.yaml` auf Render.com (Render Inc. 2024). Bei jedem Merge auf den `master`-Branch werden Backend und Frontend automatisch neu deployt.

- **Backend:** Render.com baut den Docker-Container aus `backend/Dockerfile` und startet Uvicorn auf Port 8000. Die `DATABASE_URL` wird als Secret konfiguriert.
- **Frontend:** Das Build-Skript `render_build.sh` führt `flutter build web` aus; das Ergebnis wird als statische Website gehostet (SPA-Rewrite auf `/index.html`).
- **Datenbank:** PostgreSQL 16 mit PostGIS 3.4 auf Neon.tech (Neon Inc. 2024) (Serverless, EU-Standort Frankfurt, automatische Skalierung auf null bei Inaktivität).

# 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.*

### 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 Webanwendung bereitgestellt. Das System erfüllt alle fünf in Kapitel 1 definierten Ziele: automatisierte Erfassung von über 60.000 Geodaten-Objekten, Echtzeit-Scoring (0–100) für beliebige Standorte, transparente Faktoraufschlüsselung mit GeoJSON-Visualisierung, interaktive Kartenanwendung und individuelle Gewichtung durch den Nutzer.

## 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. Die exemplarischen Ergebnisse (Kapitel 11) zeigen plausible Unterschiede zwischen verschiedenen Standorttypen.

## 12.3 Ausblick

Vielversprechende Weiterentwicklungen umfassen die Skalierung auf weitere Städte, die Integration von Echtzeitdaten (Luftqualität, Lärmkarten) und die empirische Validierung der Gewichtungen durch Nutzerbefragungen oder maschinelles Lernen.

# 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).