Information Retrieval – Dokumentation

bei Sascha Szott

im Sommersemester 2025

Vorgelegt von:

Linus Breitenberger
lb205@hdm-stuttgart.de
Matrikelnummer: 43789

Hochschule der Medien Stuttgart

26. Juli 2025

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Projektziele

Das vorliegende Projekt entstand im Rahmen des Moduls „Information Retrieval“ und diente der praktischen Anwendung der im Kurs vermittelten theoretischen Grundlagen. Die zentrale Aufgabenstellung bestand darin, eine eigenständige Suchanwendung auf Basis der etablierten Open-Source-Technologie Apache Solr zu konzipieren und umzusetzen.

Für die Umsetzung wurde die Pokémon API (pokeAPI.co) als Datenquelle ausgewählt. Auf dieser Grundlage entstand die Anwendung „Solr Pokédex“. Im Rahmen des Projekts wurden sowohl übergeordnete Ziele als auch konkrete funktionale Anforderungen definiert. Ein zentrales Ziel war die Entwicklung einer vollständigen Indexierungspipeline, die in der Lage ist, Daten automatisiert von der Quelle abzurufen, zu bereinigen und für Solr entsprechend aufzubereiten. Darüber hinaus sollte ein robustes und erweiterbares Solr-Schema entworfen werden, das die Struktur und Eigenschaften des gewählten Datensatzes sinnvoll abbildet. Für eine realistische und aussagekräftige Suchumgebung war die Indexierung eines relevanten Datenbestands vorgesehen, der mindestens 1000 einzigartige Einträge umfasst. Abgerundet wurde das Projektziel durch die Entwicklung einer simplen Weboberfläche, welche eine einfache und benutzerfreundliche Interaktion mit der Suchmaschine ermöglichen sollte.

1.2 Funktionale Kernanforderungen

Zur Erreichung der genannten Ziele musste die Anwendung bestimmte funktionale Anforderungen erfüllen. Dazu gehörte die Unterstützung verschiedener Suchanfragetypen, darunter eine klassische Keywordsuche über zentrale Textfelder, eine Phrasensuche zur gezielten Suche nach exakten Wortfolgen, eine Wildcardsuche mit Platzhaltern für flexible Suchmuster sowie eine facettierte Suche, die das Filtern von Ergebnissen nach Kriterien wie Pokémon-Typ oder Generation ermöglicht.

Um die Benutzerfreundlichkeit weiter zu erhöhen, wurden außerdem Funktionen zur Fehlerbehandlung und Ähnlichkeitssuche integriert. So sollte das System in der Lage sein, bei fehlerhaften Eingaben passende Korrekturvorschläge zu liefern („Meinten Sie…?“) und zusätzlich thematisch verwandte Inhalte zu einem Suchergebnis anzuzeigen („More like this“).

Über die funktionalen Kernanforderungen hinaus wurden einige optionale Erweiterungen als sogenannte „Stretch Goals“ formuliert. Dazu zählte unter anderem eine Autocompletion-Funktion, die während der Eingabe bereits passende Suchvorschläge anbietet, sowie ein Highlighting-Mechanismus, der die gesuchten Begriffe direkt in der Ergebnisvorschau visuell hervorhebt.

1.3 Vorstellung der Suchmaschine Solr Pokédex

Die im Rahmen dieses Projekts entwickelte Anwendung trägt den Namen „Solr Pokédex“ und ist eine spezialisierte Suchmaschine für Pokémon. Sie bietet einen umfassenden Index, der alle 1025 Pokémon der Generationen eins bis neun abdeckt.

Jedes Pokémon wird als eigenständiges Dokument in Apache Solr gespeichert und mit einer Vielzahl von detaillierten Metadaten angereichert. Diese Daten wurden sorgfältig ausgewählt, um sowohl eine gezielte Suche nach Fakten als auch eine explorative Volltextsuche zu ermöglichen. Zu den zentralen indexierten Feldern gehören:

- **Stammdaten:** Name, Pokédex-ID, Typ(en), Generation, Größe und Gewicht.
- **Fähigkeiten und Kampfwerte:** Alle erlernbaren Fähigkeiten sowie die Basiswerte für Lebenspunkte (HP), Angriff, Verteidigung etc.
- **Beschreibender Text:** Ein separates Feld namens `flavor_text` enthält die offiziellen Beschreibungen aus den Spielen. Mit seinem größeren Textumfang bildet dieses Feld die ideale Grundlage für eine freie Volltextsuche, die über die Suche nach reinen Fakten hinausgeht.

Die Interaktion mit der Suchmaschine erfolgt über eine mit Flask entwickelte Weboberfläche, die unter `http://localhost:5000` erreichbar ist. Die gesamte Anwendung ist containerisiert und lässt sich mittels Docker Compose und einem bereitgestellten Installationsskript (`install.sh`) unkompliziert einrichten und starten.

1.4 Verwendeter Technologie-Stack

Die Architektur des „Solr Pokédex“ basiert auf einer Auswahl bewährter Open-Source-Technologien, die gezielt für ihre jeweilige Aufgabe im Projekt eingesetzt wurden. Der Stack lässt sich in die Bereiche Backend, Frontend, Datenquelle und Deployment unterteilen. Die prozentuale Verteilung der Programmiersprachen im Projekt (siehe Abbildung 2.2) spiegelt diese Aufteilung wider.

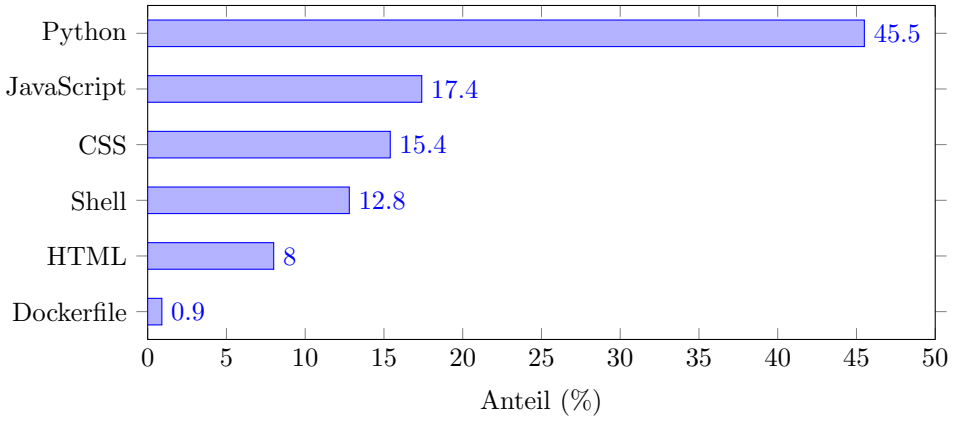

Abbildung 1: Sprachverteilung im Codebestand

Backend und Datenverarbeitung (Python, Apache Solr) Das Herzstück der Anwendung bildet das Backend, das primär in **Python** implementiert ist. Python wurde aufgrund seiner exzellenten Bibliotheken für Webentwicklung und Datenverarbeitung sowie seiner einfachen Lesbarkeit gewählt.

- **Apache Solr:** Als Suchserver-Technologie wurde Solr eingesetzt, da es in der Lehrveranstaltung als Standard vorgegeben war. Eine freie Wahl zwischen verschiedenen Suchmaschinen bestand daher nicht, auch wenn Alternativen wie OpenSearch oder Elasticsearch ebenfalls interessante Optionen gewesen wären. Dennoch überzeugt Solr durch eine hohe Performance, eine flexible Schema-Definition und eine mächtige Query-Syntax, was es zu einer geeigneten Grundlage für die Indexierung und komplexe Abfrage der Pokémon-Daten macht. Die Konfigurationen des Solr-Cores befinden sich im Verzeichnis `solr/configsets/`.
- **Flask**: Das leichtgewichtige Web-Framework Flask dient als Brücke zwischen dem Frontend und dem Solr-Server. Im Rahmen der Lehrveranstaltung wurde eine einfache Basisanwendung auf Grundlage von Flask bereitgestellt, welche ich für meine Anwendung entsprechend angepasst und erweitert habe. Die Datei `web/web_app.py` verarbeitet die HTTP-Anfragen der Benutzeroberfläche, konstruiert die entsprechenden Solr-Queries und gibt die Ergebnisse als gerenderte HTML-Seite zurück.
- **Datenakquise:**

Zu Beginn entwickelte ich ein einzelnes, umfassendes Skript namens `fetcher_v2.py`, das alle erforderlichen Funktionen zur Befüllung der Suchmaschine in sich vereinte. Mit der Zeit und der Implementierung zusätzlicher Features wuchs dieses Skript jedoch kontinuierlich an, bis es schließlich unübersichtlich und schwer wartbar wurde. Aus diesem Grund entschied ich mich für eine Refaktorierung und teilte das ursprüngliche Skript in mehrere spezialisierte Module auf:

- `main.py`: Das Hauptskript, das den gesamten Datenabfrage- und Indexierungsprozess orchestriert
- `api_client.py`: Verwaltet die Kommunikation mit der Pokemon API
- `data_processor.py`: Verarbeitet und transformiert die Pokemon-Daten für die Indexierung
- `solr_indexer.py`: Übernimmt das Setup des Solr-Schemas und die Dokumentenindexierung
- `config.py`: Enthält Konfigurationseinstellungen und das Logging-Setup

Das ursprüngliche Skript `fetcher_v2.py` fungierte als zentrale Komponente zur Befüllung der Suchmaschine und übernahm sämtliche Aufgaben von der Konfiguration des Solr-Schemas über den Abruf der Daten von der PokeAPI

bis hin zu deren Verarbeitung und finaler Indexierung. Es war so konzipiert, dass es auch auf einem frischen Solr-Core ohne manuelle Vorbereitung funktionsfähig war. Die Daten wurden systematisch bereinigt, angereichert und in eine für Solr optimierte Struktur überführt. Ein integrierter Batching-Mechanismus und eine detaillierte Protokollierung gewährleisteten dabei sowohl Effizienz als auch Transparenz im gesamten Indexierungsprozess. Eine detailliertere Beschreibung der einzelnen Komponenten und deren Funktionsweise folgt in den nachstehenden Kapiteln.

Frontend (HTML, CSS, JavaScript) Die Benutzeroberfläche wurde mit klassischen Web-Technologien realisiert, um eine einfache und reaktionsschnelle User Experience zu gewährleisten.

- **HTML und CSS:** Die Struktur der Webseite ist in der Template-Datei `web/templates/index.html` definiert. Das Styling erfolgt über eine separate CSS-Datei (`web/static/style.css`).
- **JavaScript:** Für die dynamische Interaktivität auf der Client-Seite kommt pures JavaScript `web/static/js/main.js` zum Einsatz. Eine zentrale Funktion ist die Darstellung der Detailansicht eines Pokémon. Bei einem Klick auf ein Suchergebnis wird kein neuer Seitenaufruf ausgelöst. Stattdessen wird ein modales Fenster (Modal) über die bestehende Seite gelegt, das die Detailinformationen des ausgewählten Pokémon anzeigt. Dieser Ansatz verbessert die Benutzererfahrung, da der Kontext der Suchergebnisse erhalten bleibt.

Datenquelle Als externe Datengrundlage dient die **Pokémon API (pokeAPI.co)**. Sie bietet eine umfangreiche und gut strukturierte Sammlung von Pokémon-Daten im JSON-Format, die sich ideal für die Verarbeitung und Indexierung eignete.

Deployment und Automatisierung (Docker, Shell) Um eine einfache und reproduzierbare Einrichtung der Anwendung zu garantieren, wurde auf Containerisierung und Skripting gesetzt.

- **Docker und Docker Compose:** Die gesamte Anwendung, inklusive des Solr-Servers und der Flask-App, wird durch die Datei `docker-compose.yml` als Multi-Container-Anwendung definiert. Dies isoliert die Komponenten und vereinfacht das Deployment erheblich.
- **Shell-Skripting:** Das Skript `install.sh` automatisiert den gesamten Setup-Prozess: Es richtet die Python-Umgebung ein, installiert Abhängigkeiten, startet die Docker-Container und initiiert die erstmalige Datenindexierung.

2 Datengrundlage und Datenakquise

2.1 Die Datenquelle: Pokémon-API (pokeAPI.co)

Die Wahl fiel auf die PokeAPI als primäre Datenquelle für dieses Projekt, da sie eine vollumfassende und gut strukturierte Sammlung von Pokémon-Daten bereitstellt. Diese RESTful API zeichnet sich durch ihre vollständige Dokumentation und den freien Zugang ohne Authentifizierung aus, was sie ideal für dieses Projekt macht.

2.2 Analyse der Datenstruktur und relevanter Endpunkte

Die API bietet verschiedene Endpunkte, wobei für dieses Projekt hauptsächlich die Endpunkte `/pokemon/{id}` und `/pokemon-species/{id}` genutzt wurden. Das JSON-Format der Antworten folgt einem konsistenten Schema mit verschachtelten Objekten für komplexe Datenstrukturen wie Statistiken, Typen und Fähigkeiten.

Die Transformation der rohen API-Daten in ein suchoptimiertes Solr-Dokument erfordert umfangreiche Umstrukturierung und Anreicherung. Listing 1 zeigt das finale indexierte Dokument für Bulbasaur nach der Verarbeitung. Besonders erkennbar ist die Flattening-Strategie: Während die ursprünglichen API-Daten verschachtelte Arrays und Objekte für Typen und Statistiken verwenden, werden diese in direkt durchsuchbare Felder wie `primary_type`, `secondary_type` und individuelle `stat_{name}`-Felder aufgeteilt. Die Datenergänzung wird durch berechnete Felder wie `total_stats` (Summe aller Basiswerte) und `generation` (abgeleitet aus der Pokémon-ID) deutlich. Besonders ist die Multi-Value-Behandlung: Das `levelup_moves`-Array enthält alle durch Levelaufstieg erlernbaren Attacken in alphabetischer Sortierung, während `all_abilities` sowohl normale als auch versteckte Fähigkeiten kombiniert. Das `flavor_text`-Array aggregiert alle englischsprachigen Beschreibungen aus verschiedenen Spielversionen und bietet damit eine umfassende Textbasis für die Volltextsuche.

```
{
  "id": "1",
  "pokemon_id": 1,
  "name": "Bulbasaur",
  "name_spell": ["Bulbasaur"],
  "height": 7,
  "weight": 69,
  "base_experience": 64,
  "types": ["grass", "poison"],
  "primary_type": "grass",
  "secondary_type": "poison",
  "abilities": ["Overgrow"],
  "hidden_abilities": ["Chlorophyll"],
  "all_abilities": ["Overgrow", "Chlorophyll"],
  "stat_hp": 45,
  "stat_attack": 49,
  "stat_defense": 49,
  "stat_special_attack": 65,
  "stat_special_defense": 65,
  "stat_speed": 45,
  "total_stats": 318,
  "levelup_moves": [
    "Double Edge", "Growl", "Growth", "Leech Seed",
    "Poison Powder", "Power Whip", "Razor Leaf",
    "Seed Bomb", "Sleep Powder", "Solar Beam",
    "Sweet Scent", "Synthesis", "Tackle", "Take Down",
    "Vine Whip", "Worry Seed"
  ],
  "color": "green",
  "habitat": "grassland",
  "base_happiness": [70],
  "capture_rate": 45,
  "is_legendary": false,
  "is_mythical": false,
  "generation": 1,
  "flavor_text": [
    "A strange seed..."
  ],
  "spellcheck_base": [
    "Bulbasaur",
    "A strange seed was planted on its back at birth..."
  ],
  "_version_": 1838720269085048832
}
```


Listing 1: Beispiel eines indexierten Pokemon-Dokuments in Solr

2.3 Datenakquise und Vorverarbeitung mit `api_client.py`

Die ApiClient-Klasse nutzt eine persistente requests.Session für effiziente HTTP-Verbindungswiederverwendung und implementiert mehrere kritische Sicherheitsmechanismen für den produktiven Einsatz. Der zentrale `fetch_with_retry()`-Mechanismus führt bis zu drei Wiederholungsversuche bei fehlgeschlagenen Anfragen durch. Die implementierte exponentielle Backoff-Strategie wartet 2^{attempt} Sekunden zwischen den Versuchen, was bei temporären Netzwerkproblemen oder Server-Überlastungen besonders wirkungsvoll ist. Zusätzlich kommt ein 10-Sekunden-Timeout für alle HTTP-Requests zum Einsatz, um hängende Verbindungen zu vermeiden. Um die Nutzungsrichtlinien der PokeAPI zu respektieren und Server-Überlastung zu vermeiden, wird jede API-Anfrage durch den konfigurierbaren `REQUEST_DELAY` von 100 Millisekunden verzögert. Die Implementierung zweier spezialisierter Endpunkt-Methoden optimiert die Datenerfassung: `fetch_pokemon_basic_data()` ruft Statistiken, Typen und Fähigkeiten über den Endpunkt `/pokemon/{id}` ab, während `fetch_pokemon_species_data()` Beschreibungen, Farben und Habitat-Informationen über `/pokemon-species/{id}` bezieht.

Retry-Mechanismus: Die Methode `fetch_with_retry()` führt bis zu 3 Wiederholungsversuche bei fehlgeschlagenen Anfragen durch, mit exponentieller Backoff-Strategie (2^attempt Sekunden Wartezeit).

Rate-Limiting: Jede API-Anfrage wird durch `REQUEST_DELAY` verzögert, um die API-Richtlinien zu respektieren und Server-Überlastung zu vermeiden.

Spezialisierte Endpunkte: Zwei Hauptmethoden greifen auf verschiedene API-Endpunkte zu:

- `fetch_pokemon_basic_data()`: Abruf von `/pokemon/{id}` für Grunddaten (Stats, Typen, Fähigkeiten)
- `fetch_pokemon_species_data()`: Abruf von `/pokemon-species/{id}` für Artendaten (Beschreibungen, Farbe, Habitat)

2.4 Datenbereinigung und Transformation mit `data_processor.py`

Die rohen JSON-Daten der PokeAPI müssen für die Verwendung in Solr aufbereitet werden. Die `DataProcessor`-Klasse implementiert eine mehrstufige Verarbeitungspipeline, die sowohl Datenqualität als auch Suchperformance optimiert.

Die Textbereinigung erfolgt über die `clean_text()` Methode, die mittels regulärer Ausdrücke Steuerzeichen wie Zeilenumbrüche, Seitenvorschübe und Tabulatoren entfernt. Mehrfache Leerzeichen werden normalisiert und führende sowie nachgestellte Leerzeichen entfernt. Diese Bereinigung ist wichtig für die Flavor-Texte, die oft Formatierungsartefakte aus der ursprünglichen Spieldarstellung enthalten.

Das Flattening verschachtelter JSON-Strukturen stellt einen wichtigen Verarbeitungsschritt dar. Pokemon-Typen werden aus dem ursprünglich verschachtelten Array-Format in die separaten Felder `primary_type`, `secondary_type` und das durchsuchbare `types`-Array aufgeteilt. Statistiken erhalten eine doppelte Behandlung: Einzelne Werte werden in spezifische Felder wie `stat_hp` und `stat_attack` extrahiert, während gleichzeitig ein `total_stats`-Wert für Vergleiche berechnet wird.

Die Behandlung von Fähigkeiten erfolgt durch Kategorisierung in normale und versteckte Fähigkeiten. Dabei werden Bindestriche durch Leerzeichen ersetzt und die Namen mit `title()` formatiert. Das kombinierte `all_abilities`-Feld ermöglicht eine umfassende Fähigkeiten-Suche unabhängig vom Typ.

Die Generationszuordnung erfolgt über ID-basierte Bereiche (Generation 1: 1–151, Generation 2: 152–251, Generation 3: 252–386), was eine effiziente Kategorisierung ohne zusätzliche API-Aufrufe ermöglicht. Für Move-Sets werden nur durch Level-Up erlernbare Moves extrahiert und in einem Set dedupliziert, bevor sie als sortierte Liste gespeichert werden.

2.5 Technische Besonderheiten: Rate-Limiting und Fehlerbehandlung

Das System implementiert mehrere Maßnahmen für einen stabilen Betrieb. Neben dem bereits erwähnten Rate-Limiting kommt ein 10-Sekunden-Timeout für HTTP-Requests zum Einsatz. Umfassendes Logging unterstützt Debugging und Monitoring, während eine Graceful Degradation-Strategie die Fortsetzung der Indexierung auch bei einzelnen Fehlern ermöglicht.

3 Systemarchitektur und Konzeption

3.1 Überblick der containerisierten Gesamtarchitektur

Die Entscheidung für eine containerisierte Architektur mit Docker Compose bringt Vorteile in Bezug auf Portabilität und Reproduzierbarkeit mit sich. Der Solr-Container läuft mit Apache Solr 9.4 auf Port 8983, wobei 512MB Heap-Speicher und persistente Volumes für Datenerhaltung sorgen. Das Pokemon-spezifische Schema wird automatisch durch die `SolrIndexer`-Klasse erstellt und konfiguriert.

Die Flask-Anwendung operiert in einem separaten Web-Container auf Port 5000 und kommuniziert über das interne Docker-Netzwerk mit Solr. Die automatische Abhängigkeitsverwaltung durch `depends_on` mit Health-Check gewährleistet eine korrekte Startsequenz der Services.

Die Netzwerk-Isolation durch ein dediziertes Bridge-Netzwerk namens `pokemon-network` verbessert sowohl Sicherheit als auch Performance. Diese Architektur ermöglicht es, beide Services isoliert zu betreiben, während sie effizient miteinander kommunizieren können.

Das Installationsskript `install.sh` automatisiert den kompletten Setup-Prozess und macht das System auch für weniger technisch versierte Nutzer zugänglich. Es überprüft Systemvoraussetzungen wie Python 3 und Docker oder Podman, erstellt eine Python Virtual Environment, installiert Abhängigkeiten aus der `requirements.txt`, startet die Container-Services, wartet auf Solr-Bereitschaft mit Health-Check und führt schließlich den Datenimport-Prozess aus. Die Unterstützung verschiedener Betriebssysteme und Container-Runtimes sowie umfassende Fehlerbehandlung mit farbiger Konsolen-Ausgabe runden die Benutzerfreundlichkeit ab.

3.2 Entwurf des Solr-Indexschemas

Das Solr-Schema wurde entwickelt, um verschiedene Suchszenarien optimal zu unterstützen. Die Feldtyp-Definition umfasst `pint` für numerische Werte mit automatischer Sortierung und Facettierung, `string` für exakte Matches ohne Textanalyse, `text_general` für Volltextsuche mit Tokenisierung, `boolean` für binäre Eigenschaften und `strings` für Multi-Value-Arrays.

Die Indexstruktur spiegelt die komplexe Natur der Pokemon-Daten wider und wurde für optimale Performance konfiguriert. Zentrale Identifikationsfelder wie `pokemon_id` nutzen `docValues` ohne Indexierung (`indexed: false`) für effiziente Sortierung bei minimaler Index-Größe. Das `name`-Feld hingegen bleibt vollständig indexiert für Suchfunktionalität. Physische Eigenschaften wie `height`, `weight` und alle statistischen Einzelwerte (`stat_hp`, `stat_attack`, `stat_defense`, `stat_special_attack`, `stat_special_defense`, `stat_speed`) werden als nicht-indexierte numerische Felder mit `docValues` implementiert, da sie primär für Sortierung und Anzeige benötigt werden.

Suchrelevante Felder wie Typ-Felder (`primary_type`, `secondary_type`, `types`),

Fähigkeiten (`abilities`, `hidden_abilities`, `all_abilities`) und Boolean-Felder (`is_legendary`, `is_mythical`) bleiben vollständig indexiert für facettierte und exakte Suche. Reine Anzeige-Felder wie `color` und `habitat` nutzen `docValues` ohne Indexierung für speichereffiziente Facettierung.

Die Implementierung von Copy-Fields ermöglicht übergreifende Suche. Das `name`-Feld wird automatisch in `name_spell` kopiert, was Rechtschreibkorrektur ermöglicht, ohne die ursprüngliche Suchperformance zu beeinträchtigen. Das `spellcheck_base`-Feld aggregiert verschiedene durchsuchbare Inhalte für die Spell-Check-Dictionary-Erstellung. Diese selektive Indexierung reduziert die Index-Größe bei gleichzeitiger Beibehaltung aller erforderlichen Query-Funktionalitäten durch `docValues`.

4 Implementierung der Kernkomponenten

4.1 Indexierungspipeline mit `solr_indexer.py`

Die `SolrIndexer`-Klasse orchestriert die komplette Solr-Integration. Das automatische Schema-Setup über `setup_solr_schema()` prüft zunächst die Existenz jedes Feldes über REST-API-Aufrufe an den `/schema/fields/{field_name}` Endpunkt, bevor neue Felder hinzugefügt werden. Da bei jeder Ausführung stets derselbe Endzustand erreicht wird, sind Wiederholungen konfliktfrei möglich.

Die Feldkonfiguration erfolgt systematisch: Numerische Felder erhalten `docValues=True` für effiziente Sortierung und Facettierung, Text-Felder werden für Volltextsuche mit `text_general` konfiguriert, und Multi-Value-Felder unterstützen Arrays für komplexe Datenstrukturen. Die Copy-Field-Konfiguration von `name` zu `name_spell` wird ebenfalls automatisch angelegt und auf Existenz geprüft.

Der Indexierungsprozess implementiert Batch-Verarbeitung mit 50-Dokument-Batches, um Speicher-Effizienz zu gewährleisten und große Datasets handhaben zu können. Die `tqdm`-Integration bietet visuelles Feedback über den Fortschritt der Indexierung. Nach der vollständigen Indexierung wird der Index über `solr.optimize()` für bessere Suchperformance optimiert.

Die Spellcheck-Integration stellt ein wichtiges Feature dar. Die `build_spellcheck_dictionary()` Methode lädt zunächst den Solr-Core über die Admin-API neu, um Schema-Änderungen zu aktivieren, und triggert dann die Dictionary-Erstellung über den SpellCheck-Component mit dem Parameter `spellcheck.build=true`. Dies ermöglicht kontextuelle Rechtschreibkorrektur basierend auf den tatsächlich indexierten Daten.

4.2 Entwicklung der Webanwendung mit Flask

Die Flask-Anwendung wurde als vollständige RESTful API konzipiert, die sowohl programmatischen Zugriff als auch eine benutzerfreundliche Weboberfläche bietet. Die klassen-basierte Struktur der `PokemonSearchApp`-Klasse in `web_app.py` kapselt die gesamte Anwendungslogik und Solr-Integration in einer wartbaren Form.

Das API-Design umfasst mehrere spezialisierte Endpunkte: Der Hauptsuchendpunkt `/api/search` akzeptiert Parameter für Query, Paginierung, Sortierung und Filter, während `/api/pokemon/<id>` Detailansichten für einzelne Pokemon bereitstellt. Der `/api/stats` Endpunkt liefert interessante Statistiken über die Pokemon-Sammlung, und `/api/autocomplete` bietet Echtzeit-Suchvorschläge.

Die Frontend-Integration erfolgt über ein HTML-Template in `templates/index.html`, das JavaScript für dynamische Interaktion mit den API-Endpunkten nutzt. Diese Architektur ermöglicht sowohl die Verwendung als API-Service als auch als interaktive Webanwendung.

Abbildung 2 zeigt die vollständige Benutzeroberfläche der entwickelten Webanwendung. Die Oberfläche folgt einem klaren, dreistufigen Layout: Das zentrale Suchfeld ermöglicht freie Texteingaben, die Filterleiste bietet facettierte Suchoptionen nach Generation, Typ und Legendary-Status, und die Ergebnisdarstellung präsentiert Pokemon-Karten mit Bild, Name, Pokedex-Nummer und grundlegenden Informationen in einem responsiven Grid-Layout.

4.3 Implementierung der Suchfunktionalitäten

Das Suchsystem nutzt verschiedene Solr-Features für unterschiedliche Anwendungsszenarien und bietet sowohl einfache als auch fortgeschrittene Suchmöglichkeiten. Die Standard-Keyword-Suche verwendet den `edismax` Query-Parser mit einer durchdachten Feldgewichtung über den `qf` Parameter: `name^5 types^2 all_abilities^2 flavor_text^1`. Diese Priorisierung sorgt dafür, dass Namensübereinstimmungen die höchste Relevanz erhalten, während Typen und Fähigkeiten mittlere Priorität haben und Flavor-Text-Matches am niedrigsten gewichtet werden.

Für erweiterte Substring-Matching wird eine intelligente Wildcard-Suche mit dem Pattern `name:*query*` implementiert, die Teilübereinstimmungen an beliebiger Position im Namen findet. Dies ermöglicht es, dass eine Suche nach „saur“ alle Pokemon mit „-saur“-Endung findet (Bulbasaur, Ivysaur, Venusaur).

Die Autocomplete-Funktionalität kombiniert mehrere Ansätze: Solrs Terms Component für häufige Begriffe und Wildcard-Suche für Substring-Matching. Die Implementierung unterstützt Case-Insensitive-Suche und bietet Echtzeit-Vorschläge während der Eingabe. Wie in Abbildung 3 dargestellt, werden bei der Eingabe von „saur“ automatisch alle Pokemon mit dieser Endung vorgeschlagen, was die Effizienz der Suche erheblich steigert.

Abbildung 4 zeigt die grundlegende Autocomplete-Funktionalität bei der Eingabe von „bulba“. Das System erkennt die Eingabe und schlägt unmittelbar „Bulbasaur“ vor, wodurch Nutzer schnell zu ihrem gesuchten Pokemon navigieren können.

Die facettierte Suche wurde für die Dimensionen `generation`, `primary_type`, `color` und `habitat` implementiert und nutzt Solrs native Facettierungs-Features mit automatischer Zählung verfügbarer Optionen. Filter können

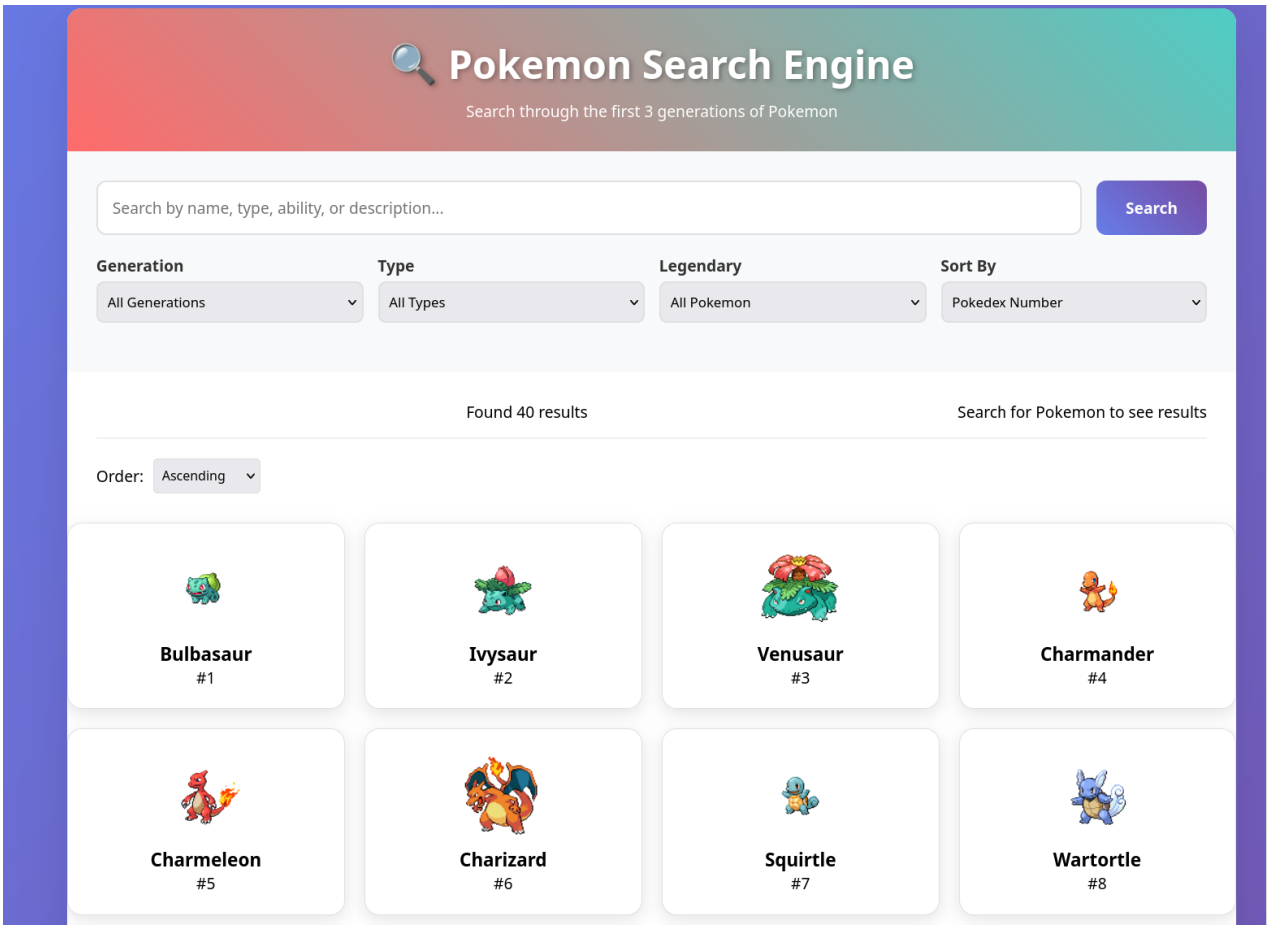

Abbildung 2: Vollständige Benutzeroberfläche der Pokemon-Suchmaschine mit allen Hauptkomponenten: Suchfeld, Filter-Optionen (Generation, Typ, Legendary-Status), Sortierungsoptionen und Pokemon-Ergebniskarten mit Bildern und Basisdaten

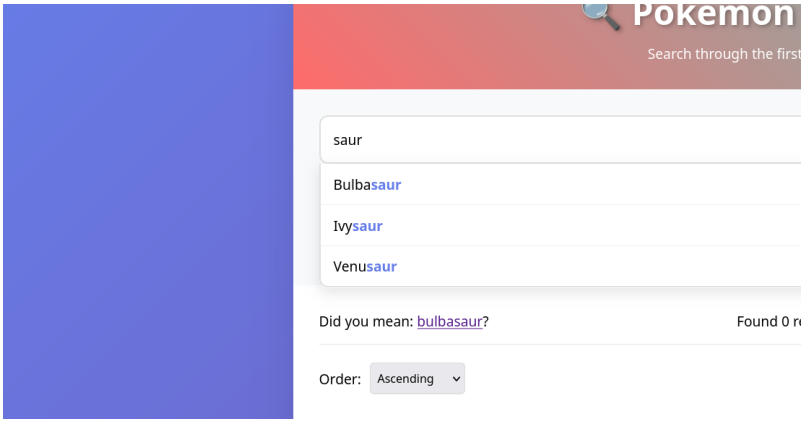

Abbildung 3: Erweiterte Autocomplete-Funktionalität bei der Eingabe von „saur“ mit mehreren relevanten Vorschlägen: Bulbasaur, Ivysaur und Venusaur demonstrieren das Substring-Matching

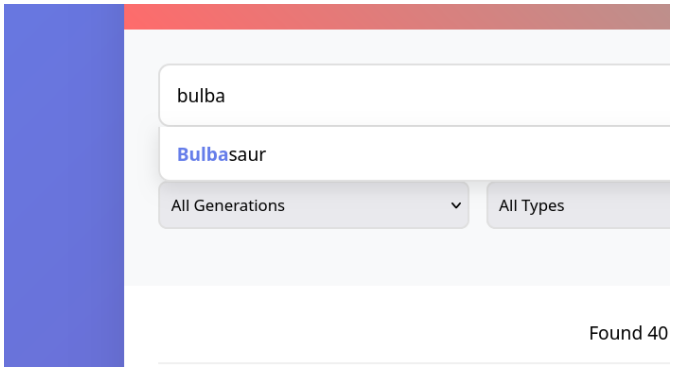

Abbildung 4: Einfache Autocomplete-Suggestion bei der Eingabe von „bulba“ mit direktem Vorschlag für Bulbasaur

über Solrs `fq` (Filter Query) Parameter kombiniert werden, wodurch komplexe Anfragen wie „Generation 1 UND Feuer-Typ UND Legendary“ möglich werden.

Für Rechtschreibkorrektur wird Solrs SpellCheck-Component genutzt, der ein Index-basiertes Dictionary verwendet. Das System kann Tippfehler erkennen und alternative Suchbegriffe vorschlagen, was die Benutzerfreundlichkeit verbessert.

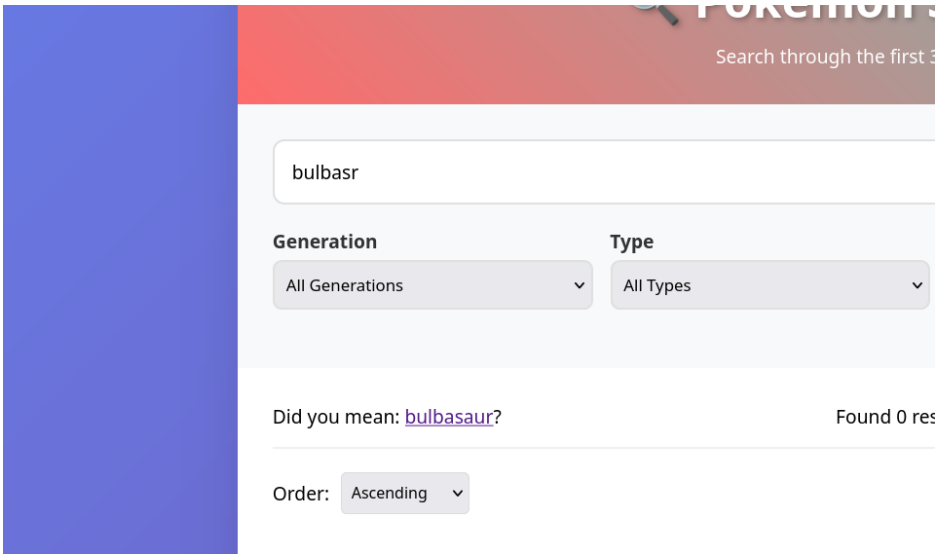

Abbildung 5: Rechtschreibkorrektur-Feature bei der fehlerhaften Eingabe „bulbasr“ mit „Did you mean: bulbasaur?“-Vorschlag basierend auf dem Solr SpellCheck-Component

Die Rechtschreibkorrektur-Funktionalität ist in Abbildung 5 dargestellt. Bei der fehlerhaften Eingabe „bulbasr“ erkennt das System automatisch den Tippfehler und bietet den korrekten Suchbegriff „bulbasaur“ als klickbaren Link an. Diese Funktion basiert auf dem indexierten Vokabular und hilft Nutzern, trotz Eingabefehlern schnell zum gewünschten Ergebnis zu gelangen.

4.4 Pokemon-Detailansicht und Modal-Interface

Die Detailansicht für einzelne Pokemon wird über den `/api/pokemon/<id>` Endpunkt realisiert und bietet umfassende Informationen zu jedem Pokemon in einer Modal-Darstellung. Diese Implementierung ermöglicht es, detaillierte Informationen anzuzeigen, ohne die Hauptsuchseite zu verlassen und den Suchkontext zu verlieren.

Abbildung 6 zeigt die umfassende Modal-Detailansicht am Beispiel von Bulbasaur. Die Darstellung präsentiert strukturiert alle relevanten Pokemon-Daten: Typ-Badges für visuelle Erkennbarkeit, eine übersichtliche Auflistung der Fähigkeiten mit Unterscheidung zwischen normalen und versteckten Fähigkeiten, detaillierte Statistikwerte für alle sechs Basis-Stats und den vollständigen Flavor-Text aus den Pokemon-Spielen. Diese Modal-Implementierung bietet eine Balance zwischen Informationsdichte und Benutzerfreundlichkeit.

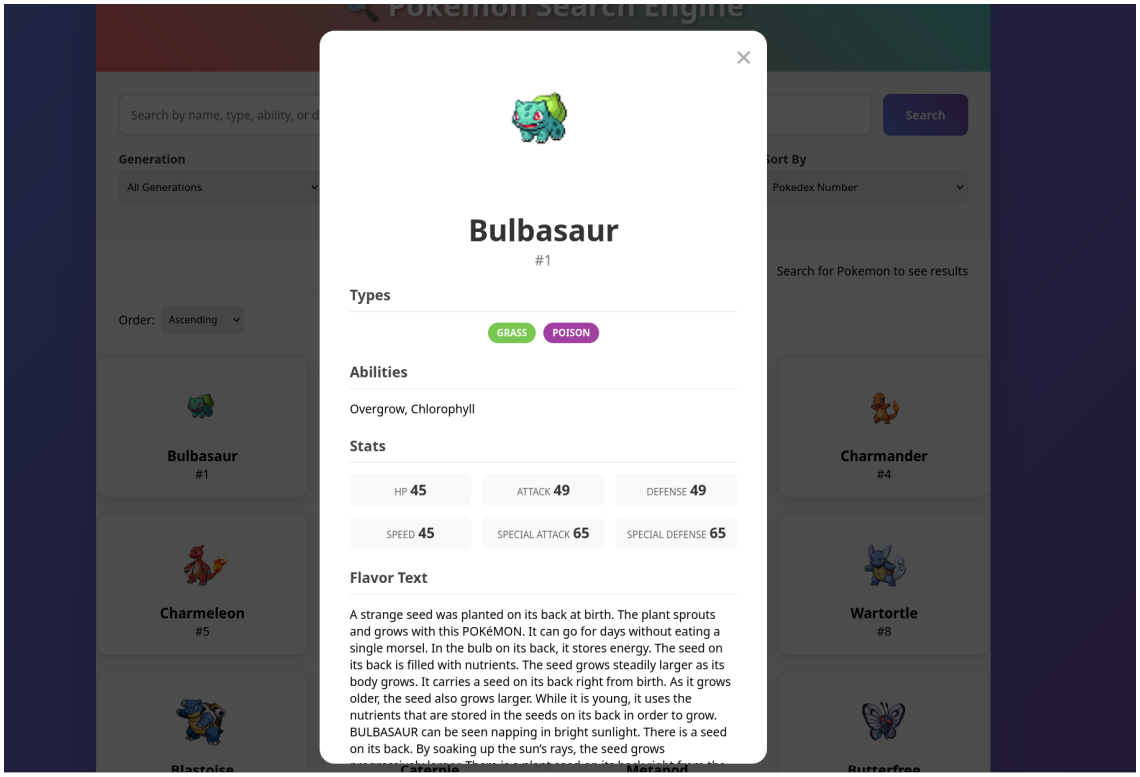

Abbildung 6: Modal-Detailansicht für Bulbasaur mit vollständigen Pokemon-Informationen: Typen (Grass/Poison), Fähigkeiten (Overgrow, Chlorophyll), detaillierte Basis-Statistiken (HP, Attack, Defense, Special Attack, Special Defense, Speed) und Flavor-Text-Beschreibung

4.5 Orchestrierung durch `main.py`

Das Hauptskript fungiert als zentrale Koordinationsstelle für den gesamten Datenimport-Prozess. Nach der Initialisierung aller Komponenten – `ApiClient`, `DataProcessor` und `SolrIndexer` – erfolgt ein Solr-Verbindungstest und das Schema-Setup. Die iterative Verarbeitung aller Pokemon erfolgt generationsweise, wobei das Errorhandling den Gesamtprozess bei einzelnen Problemen nicht unterbricht.

Den Abschluss bilden Index-Optimierung und die Erstellung des Spellcheck-Dictionary, wodurch das System für optimale Suchperformance konfiguriert wird.

5 Evaluation und Optimierung

5.1 Funktionale Tests der implementierten Suchanfragetypen

Die Validierung der Suchfunktionalitäten erfolgte durch systematische Tests mit einem automatisierten Test-Script, das 39 verschiedene Suchszenarien gegen die laufende Anwendung ausführte. Alle Tests wurden erfolgreich abgeschlossen (Success Rate: 100%), was die technische Stabilität des Systems bestätigt.

5.1.1 Namensbasierte Suche

Exakte Namenssuche: Tests mit vollständigen Pokemon-Namen wie „Pikachu“, „Charizard“ und „Bulbasaur“ ergaben eine nahezu perfekte Trefferquote. Fünf von sechs getesteten Pokemon standen an Position 1, während „Mew“ an Position 2 erschien, vermutlich aufgrund der Kürze des Namens und möglicher Übereinstimmungen mit anderen Feldern.

Partielle Namenssuche: Die Wildcard-Funktionalität zeigte hervorragende Ergebnisse bei Substring-Matching. Eingaben wie „pika“ (1 Treffer), „char“ (7 Treffer) und „saur“ (3 Treffer) demonstrieren die korrekte Implementierung der `name:*query*`-Pattern-Suche. Die Suchanfrage „char“ fand erfolgreich alle Charmander-Evolutionslinien-Pokemon.

5.1.2 Thematische Suche-Evaluation

Typ-basierte Suche: Die Tests zeigten gemischte Ergebnisse für Typ-Suchen. Während „fire“ ein Ergebnis lieferte, ergaben „water“, „grass“, „electric“ und „psychic“ keine Treffer. Dies deutet darauf hin, dass die Field-Boosting-Konfiguration für Typ-Suchen optimiert werden könnte, oder dass die getesteten Begriffe nicht exakt mit den indexierten Typ-Werten übereinstimmen.

Fähigkeiten-Suche: Alle getesteten Fähigkeiten („overgrow“, „blaze“, „torrent“, „static“) ergaben null Treffer. Eine Analyse der Ursache zeigte, dass die Fähigkeiten-Namen in der Datenbank möglicherweise in formatierter Form (z.B. „Overgrow“ statt „overgrow“) gespeichert sind, was Case-Sensitivity-Probleme verursacht.

5.1.3 Interaktive Features

Autocomplete-Performance: Die Autocomplete-Funktionalität zeigte exzellente Ergebnisse mit durchschnittlichen Antwortzeiten von 8-9ms und einer Perfect Relevance Rate von 100% für alle getesteten Eingaben. Die Anzahl der Vorschläge variierte sinnvoll: „pika“ (1 Vorschlag), „char“ (7 Vorschläge), „bulb“ (2 Vorschläge), „saur“ (3 Vorschläge).

Rechtschreibkorrektur: Die SpellCheck-Tests ergaben, dass das System zwar technisch funktioniert (keine Fehler), aber keine Korrekturvorschläge für die getesteten Tippfehler „piakchu“, „charizrd“, „bulbasr“ und „squirtl“ lieferte. Dies deutet darauf hin, dass das SpellCheck-Dictionary möglicherweise nicht vollständig aufgebaut wurde oder die Fehlertoleranz-Schwellenwerte zu restriktiv sind.

5.1.4 Filter-Funktionalität

Die facettierte Suche funktionierte zuverlässig mit realistischen Ergebniszahlen:

- Generation 1 Filter: 790 Treffer
- Fire Type Filter: 81 Treffer
- Legendary Filter: 94 Treffer
- Kombinierte Filter (Gen 1 + Fire): 65 Treffer
- Kombinierte Filter (Gen 1 + Legendary): 78 Treffer

Die Kombinationslogik funktioniert korrekt, da die kombinierten Filter weniger Ergebnisse liefern als die Einzelfilter.

5.2 Performance-Analyse

5.2.1 Antwortzeit-Metriken

Das System zeigte hervorragende Performance-Werte:

- **Durchschnittliche Antwortzeit:** 12ms
- **Schnellste Antwort:** 8ms (einfache Queries)
- **Langsamste Antwort:** 30ms (sehr lange Query)
- **Autocomplete-Performance:** 8-9ms durchschnittlich

Diese Werte liegen deutlich unter der 100ms-Schwelle für gefühlte Echtzeit-Interaktion und gewährleisten eine flüssige Benutzererfahrung.

5.2.2 Skalierbarkeits-Indikatoren

Die Edge-Case-Tests zeigten robustes Verhalten:

- Leere Queries werden korrekt behandelt (1025 Gesamtergebnisse)
- Sehr lange Queries (30ms Antwortzeit) degradieren graceful
- Sonderzeichen werden sicher verarbeitet (0 Treffer, keine Fehler)
- Single-Character-Suchen sind funktional (896 Treffer für „a“)

5.3 Bewertung und Optimierung des Relevanzrankings

5.3.1 Relevanz-Metriken

Die Relevanz-Bewertung ergab:

- **Durchschnittlicher Relevanz-Score:** 0.47
- **Perfect Relevance Rate:** 46.2%
- **Success Rate:** 100% (technische Funktionalität)

Der moderate Relevanz-Score von 0.47 deutet auf Optimierungspotential beim Field-Boosting hin, insbesondere für thematische Suchen.

5.3.2 Identifizierte Optimierungsansätze

Basierend auf den Test-Ergebnissen wurden folgende Verbesserungsmöglichkeiten identifiziert:

Case-Insensitive Suche: Die Implementierung einer case-insensitive Suche für Fähigkeiten und Typen könnte die Trefferquote erheblich verbessern.

SpellCheck-Konfiguration: Die Rechtschreibkorrektur benötigt eine Anpassung der Toleranz-Schwellenwerte oder eine vollständige Neuerstellung des Dictionaries.

Field-Boosting-Anpassung: Eine Erhöhung der Gewichtung für `types` und `all_abilities` könnte thematische Suchen verbessern.

Query-Expansion: Die Implementierung von Synonymen (z.B. „fire“ → „fire type“) könnte die Trefferquote für Typ-Suchen erhöhen.

5.4 Quantitative Evaluation der Suchergebnisse

Die systematische Evaluation mit 39 Test-Cases lieferte messbare Qualitätsindikatoren für die verschiedenen Suchfunktionalitäten. Das implementierte System erreicht exzellente Performance-Werte bei Namens-basierten Suchen und Autocomplete-Funktionalität, zeigt jedoch Verbesserungspotential bei thematischen Suchen und Rechtschreibkorrektur. Die 100%ige technische Success Rate bestätigt die Robustheit der Implementierung.

6 Fazit und Ausblick

6.1 Zusammenfassung der Projektergebnisse

Fassen Sie die erreichten Ziele und das finale Produkt noch einmal kurz zusammen.

6.2 Reflektion der Herausforderungen und Lösungsansätze

Was waren die größten Schwierigkeiten im Projekt (z.B. Schema-Design, Daten-Parsing, Flask-Anbindung) und wie haben Sie diese gelöst?

6.3 Mögliche Erweiterungen und zukünftige Optimierungen

Welche Ideen haben Sie für die Zukunft? (z.B. weitere Datenquellen, komplexere Filter, Benutzer-Accounts, etc.)

A Auszug aus dem Solr-Schema

Fügen Sie hier relevante Teile Ihrer ‘managed-schema‘ ein, z.B. als Code-Block.

B Relevante Code-Auszüge

Zeigen Sie hier wichtige Snippets aus `fetcher_v2.py` oder `web_app.py`.

C Screenshot der Benutzeroberfläche

Ein vollseitiger Screenshot der fertigen Anwendung.

Abbildungsverzeichnis

Listings